libpgf

C. Stamm

Version 7.21.07
Tue May 31 2022

Table of Contents

Table of contents

Hierarchical Index

Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

CDecoder 5
CEncoder 17
CDecoder::CMacroBlock 28
CEncoder::CMacroBlock 37
CPGFImage 48
CPGFStream 108
  CPGFFileStream 45
  CPGFMemoryStream 102

CSubband 110
CWaveletTransform 118
IOException 128
PGFHeader 130
PGFMagicVersion 133
  PGFPreHeader 136

PGFPostHeader 134
PGFRect 138
PGFVersionNumber 141
ROIBlockHeader::RBH 143
ROIBlockHeader 144

Class Index

Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

CDecoder (PGF decoder) 5
CEncoder (PGF encoder) 17
CDecoder::CMacroBlock (A macro block is a decoding unit of fixed size (uncoded)) 28
CEncoder::CMacroBlock (A macro block is an encoding unit of fixed size (uncoded)) 37
CPGFFileStream (File stream class) 45
CPGFImage (PGF main class) 48
CPGFMemoryStream (Memory stream class) 102
CPGFStream (Abstract stream base class) 108
CSubband (Wavelet channel class) 110
CWaveletTransform (PGF wavelet transform) 118
IOException (PGF exception) 128
PGFHeader (PGF header) 130
PGFMagicVersion (PGF identification and version) 133
PGFPostHeader (Optional PGF post-header) 134
PGFPreHeader (PGF pre-header) 136
PGFRect (Rectangle) 138
PGFVersionNumber (Version number stored in header since major version 7) 141
ROIBlockHeader::RBH (Named ROI block header (part of the union)) 143
ROIBlockHeader (Block header used with ROI coding scheme) 144

File Index

File List

Here is a list of all files with brief descriptions:

PGFimage.h (PGF image class) 146
PGFplatform.h (PGF platform specific definitions) 151
PGFstream.h (PGF stream class) 166
PGFtypes.h (PGF definitions) 169
BitStream.h 181
Decoder.cpp (PGF decoder class implementation) 191
Decoder.h (PGF decoder class) 206
Encoder.cpp (PGF encoder class implementation) 209
Encoder.h (PGF encoder class) 221
PGFimage.cpp (PGF image class implementation) 225
PGFstream.cpp (PGF stream class implementation) 264
Subband.cpp (PGF wavelet subband class implementation) 269
Subband.h (PGF wavelet subband class) 275
WaveletTransform.cpp (PGF wavelet transform class implementation) 278
WaveletTransform.h (PGF wavelet transform class) 287

Class Documentation

CDecoder Class Reference

PGF decoder.

`#include <Decoder.h>`

Classes

- class **CMacroBlock**
  A macro block is a decoding unit of fixed size (uncoded)

Public Member Functions

- **CDecoder** (**CPGFStream** *stream, **PGFPreHeader** &preHeader, **PGFHeader** &header, **PGFPostHeader** &postHeader, UINT32 *&levelLength, UINT64 &userDataPos, bool useOMP, UINT32 userDataPolicy)
- **~CDecoder** ()
  Destructor.
- void **Partition** (**CSubband** *band, int quantParam, int width, int height, int startPos, int pitch)
- void **DecodeInterleaved** (**CWaveletTransform** *wtChannel, int level, int quantParam)
- UINT32 **GetEncodedHeaderLength** () const
- void **SetStreamPosToStart** ()
  Resets stream position to beginning of PGF pre-header.
- void **SetStreamPosToData** ()
  Resets stream position to beginning of data block.
- void **Skip** (UINT64 offset)
- void **DequantizeValue** (**CSubband** *band, UINT32 bandPos, int quantParam)
- UINT32 **ReadEncodedData** (UINT8 *target, UINT32 len) const
- void **DecodeBuffer** ()
- **CPGFStream * GetStream** ()
- void **GetNextMacroBlock** ()

Private Member Functions

- void **ReadMacroBlock** (**CMacroBlock** *block)
  *throws **IOException***

Private Attributes

- **CPGFStream * m_stream**
  input PGF stream
- UINT64 **m_startPos**
  stream position at the beginning of the PGF pre-header
- UINT64 **m_streamSizeEstimation**
  estimation of stream size

- UINT32 **m_encodedHeaderLength**
  stream offset from startPos to the beginning of the data part (highest level)

- **CMacroBlock ** m_macroBlocks**
  array of macroblocks

- int **m_currentBlockIndex**
  index of current macro block

- int **m_macroBlockLen**
  array length

- int **m_macroBlocksAvailable**
  number of decoded macro blocks (including currently used macro block)

- **CMacroBlock * m_currentBlock**
  current macro block (used by main thread)

Detailed Description

PGF decoder.

PGF decoder class.

Author

C. Stamm, R. Spuler

Definition at line **46** of file **Decoder.h**.

Constructor & Destructor Documentation

CDecoder::CDecoder (CPGFStream * *stream*, PGFPreHeader & *preHeader*, PGFHeader & *header*, PGFPostHeader & *postHeader*, UINT32 *& *levelLength*, UINT64 & *userDataPos*, bool *useOMP*, UINT32 *userDataPolicy*)

Constructor: Read pre-header, header, and levelLength at current stream position. It might throw an **IOException**.

Parameters

stream	A PGF stream
preHeader	[out] A PGF pre-header
header	[out] A PGF header
postHeader	[out] A PGF post-header
levelLength	The location of the levelLength array. The array is allocated in this method. The caller has to delete this array.
userDataPos	The stream position of the user data (metadata)
useOMP	If true, then the decoder will use multi-threading based on openMP
userDataPolicy	Policy of user data (meta-data) handling while reading PGF headers.

Constructor Read pre-header, header, and levelLength It might throw an **IOException**.

Parameters

stream	A PGF stream
preHeader	[out] A PGF pre-header
header	[out] A PGF header
postHeader	[out] A PGF post-header
levelLength	The location of the levelLength array. The array is allocated in this method. The caller has to delete this array.
userDataPos	The stream position of the user data (metadata)
useOMP	If true, then the decoder will use multi-threading based on openMP
userDataPolicy	Policy of user data (meta-data) handling while reading PGF headers.

Definition at line **73** of file **Decoder.cpp**.

```
00076 : m_stream(stream)
00077 , m_startPos(0)
00078 , m_streamSizeEstimation(0)
00079 , m_encodedHeaderLength(0)
00080 , m_currentBlockIndex(0)
00081 , m_macroBlocksAvailable(0)
00082 #ifdef __PGFROISUPPORT__
00083 , m_roi(false)
00084 #endif
00085 {
00086         ASSERT(m_stream);
00087
00088         int count, expected;
00089
00090         // store current stream position
00091         m_startPos = m_stream->GetPos();
00092
00093         // read magic and version
00094         count = expected = MagicVersionSize;
00095         m_stream->Read(&count, &preHeader);
00096         if (count != expected) ReturnWithError(MissingData);
00097
00098         // read header size
00099         if (preHeader.version & Version6) {
00100                 // 32 bit header size since version 6
00101                 count = expected = 4;
00102         } else {
00103                 count = expected = 2;
00104         }
00105         m_stream->Read(&count, ((UINT8*)&preHeader) + MagicVersionSize);
00106         if (count != expected) ReturnWithError(MissingData);
00107
00108         // make sure the values are correct read
00109         preHeader.hSize = __VAL(preHeader.hSize);
00110
00111         // check magic number
00112         if (memcmp(preHeader.magic, PGFMagic, 3) != 0) {
00113                 // error condition: wrong Magic number
00114                 ReturnWithError(FormatCannotRead);
00115         }
00116
00117         // read file header
00118         count = expected = (preHeader.hSize < HeaderSize) ? preHeader.hSize :
HeaderSize;
00119         m_stream->Read(&count, &header);
00120         if (count != expected) ReturnWithError(MissingData);
00121
00122         // make sure the values are correct read
00123         header.height = __VAL(UINT32(header.height));
00124         header.width = __VAL(UINT32(header.width));
00125
00126         // be ready to read all versions including version 0
00127         if (preHeader.version > 0) {
00128 #ifndef __PGFROISUPPORT__
00129                 // check ROI usage
00130                 if (preHeader.version & PGFROI)
ReturnWithError(FormatCannotRead);
00131 #endif
00132
00133                 UINT32 size = preHeader.hSize;
00134
```

```
00135                   if (size > HeaderSize) {
00136                           size -= HeaderSize;
00137                           count = 0;
00138
00139                           // read post-header
00140                           if (header.mode == ImageModeIndexedColor) {
00141                                   if (size < ColorTableSize)
ReturnWithError(FormatCannotRead);
00142                                   // read color table
00143                                   count = expected = ColorTableSize;
00144                                   m_stream->Read(&count, postHeader.clut);
00145                                   if (count != expected)
ReturnWithError(MissingData);
00146                           }
00147
00148                           if (size > (UINT32)count) {
00149                                   size -= count;
00150
00151                                   // read/skip user data
00152                                   UserdataPolicy policy =
(UserdataPolicy)((userDataPolicy <= MaxUserDataSize) ? UP_CachePrefix : 0xFFFFFFFF -
userDataPolicy);
00153                                   userDataPos = m_stream->GetPos();
00154                                   postHeader.userDataLen = size;
00155
00156                                   if (policy == UP_Skip) {
00157                                           postHeader.cachedUserDataLen = 0;
00158                                           postHeader.userData = nullptr;
00159                                           Skip(size);
00160                                   } else {
00161                                           postHeader.cachedUserDataLen =
(policy == UP_CachePrefix) ? __min(size, userDataPolicy) : size;
00162
00163                                           // create user data memory block
00164                                           postHeader.userData =
new(std::nothrow) UINT8[postHeader.cachedUserDataLen];
00165                                           if (!postHeader.userData)
ReturnWithError(InsufficientMemory);
00166
00167                                           // read user data
00168                                           count = expected =
postHeader.cachedUserDataLen;
00169                                           m_stream->Read(&count,
postHeader.userData);
00170                                           if (count != expected)
ReturnWithError(MissingData);
00171
00172                                           // skip remaining user data
00173                                           if (postHeader.cachedUserDataLen <
size) Skip(size - postHeader.cachedUserDataLen);
00174                                   }
00175                           }
00176                   }
00177
00178                   // create levelLength
00179                   levelLength = new(std::nothrow) UINT32[header.nLevels];
00180                   if (!levelLength) ReturnWithError(InsufficientMemory);
00181
00182                   // read levelLength
00183                   count = expected = header.nLevels*WordBytes;
00184                   m_stream->Read(&count, levelLength);
00185                   if (count != expected) ReturnWithError(MissingData);
00186
00187 #ifdef PGF_USE_BIG_ENDIAN
00188                   // make sure the values are correct read
00189                   for (int i=0; i < header.nLevels; i++) {
00190                           levelLength[i] = __VAL(levelLength[i]);
00191                   }
00192 #endif
00193
00194                   // compute the total size in bytes; keep attention: level length
information is optional
00195                   for (int i=0; i < header.nLevels; i++) {
00196                           m_streamSizeEstimation += levelLength[i];
00197                   }
00198
00199           }
```

```
00200
00201         // store current stream position
00202         m_encodedHeaderLength = UINT32(m_stream->GetPos() - m_startPos);
00203
00204         // set number of threads
00205 #ifdef LIBPGF_USE_OPENMP
00206         m_macroBlockLen = omp_get_num_procs();
00207 #else
00208         m_macroBlockLen = 1;
00209 #endif
00210
00211         if (useOMP && m_macroBlockLen > 1) {
00212 #ifdef LIBPGF_USE_OPENMP
00213                 omp_set_num_threads(m_macroBlockLen);
00214 #endif
00215
00216                 // create macro block array
00217                 m_macroBlocks = new(std::nothrow)
CMacroBlock*[m_macroBlockLen];
00218                 if (!m_macroBlocks) ReturnWithError(InsufficientMemory);
00219                 for (int i = 0; i < m_macroBlockLen; i++) m_macroBlocks[i] = new
CMacroBlock();
00220                 m_currentBlock = m_macroBlocks[m_currentBlockIndex];
00221         } else {
00222                 m_macroBlocks = 0;
00223                 m_macroBlockLen = 1; // there is only one macro block
00224                 m_currentBlock = new(std::nothrow) CMacroBlock();
00225                 if (!m_currentBlock) ReturnWithError(InsufficientMemory);
00226         }
00227 }
```

CDecoder::~CDecoder ()

Destructor.

Definition at line **231** of file **Decoder.cpp**.

```
00231                 {
00232         if (m_macroBlocks) {
00233                 for (int i=0; i < m_macroBlockLen; i++) delete m_macroBlocks[i];
00234                 delete[] m_macroBlocks;
00235         } else {
00236                 delete m_currentBlock;
00237         }
00238 }
```

Member Function Documentation

void CDecoder::DecodeBuffer ()

Reads next block(s) from stream and decodes them It might throw an **IOException**.

Definition at line **494** of file **Decoder.cpp**.

```
00494                         {
00495         ASSERT(m_macroBlocksAvailable <= 0);
00496
00497         // macro block management
00498         if (m_macroBlockLen == 1) {
00499                 ASSERT(m_currentBlock);
00500                 ReadMacroBlock(m_currentBlock);
00501                 m_currentBlock->BitplaneDecode();
00502                 m_macroBlocksAvailable = 1;
00503         } else {
00504                 m_macroBlocksAvailable = 0;
00505                 for (int i=0; i < m_macroBlockLen; i++) {
00506                         // read sequentially several blocks
00507                         try {
00508                                 ReadMacroBlock(m_macroBlocks[i]);
00509                                 m_macroBlocksAvailable++;
00510                         } catch(IOException& ex) {
00511                                 if (ex.error == MissingData || ex.error ==
FormatCannotRead) {
```

```
00512                                           break; // no further data available or
the data isn't valid PGF data (might occur in streaming or PPPExt)
00513                                   } else {
00514                                           throw;
00515                                   }
00516                           }
00517                   }
00518 #ifdef LIBPGF_USE_OPENMP
00519                   // decode in parallel
00520                   #pragma omp parallel for default(shared) //no declared
exceptions in next block
00521 #endif
00522                   for (int i=0; i < m_macroBlocksAvailable; i++) {
00523                           m_macroBlocks[i]->BitplaneDecode();
00524                   }
00525
00526                   // prepare current macro block
00527                   m_currentBlockIndex = 0;
00528                   m_currentBlock = m_macroBlocks[m_currentBlockIndex];
00529           }
00530 }
```

void CDecoder::DecodeInterleaved (CWaveletTransform * *wtChannel*, int *level*, int *quantParam*)

Deccoding and dequantization of HL and LH subband (interleaved) using partitioning scheme. Partitioning scheme: The plane is partitioned in squares of side length InterBlockSize. It might throw an **IOException**.

Parameters

wtChannel	A wavelet transform channel containing the HL and HL band
level	Wavelet transform level
quantParam	Dequantization value

Definition at line **333** of file **Decoder.cpp**.

```
00333
{
00334           CSubband* hlBand = wtChannel->GetSubband(level, HL);
00335           CSubband* lhBand = wtChannel->GetSubband(level, LH);
00336           const div_t lhH = div(lhBand->GetHeight(), InterBlockSize);
00337           const div_t hlW = div(hlBand->GetWidth(), InterBlockSize);
00338           const int hlws = hlBand->GetWidth() - InterBlockSize;
00339           const int hlwr = hlBand->GetWidth() - hlW.rem;
00340           const int lhws = lhBand->GetWidth() - InterBlockSize;
00341           const int lhwr = lhBand->GetWidth() - hlW.rem;
00342           int hlPos, lhPos;
00343           int hlBase = 0, lhBase = 0, hlBase2, lhBase2;
00344
00345           ASSERT(lhBand->GetWidth() >= hlBand->GetWidth());
00346           ASSERT(hlBand->GetHeight() >= lhBand->GetHeight());
00347
00348           if (!hlBand->AllocMemory()) ReturnWithError(InsufficientMemory);
00349           if (!lhBand->AllocMemory()) ReturnWithError(InsufficientMemory);
00350
00351           // correct quantParam with normalization factor
00352           quantParam -= level;
00353           if (quantParam < 0) quantParam = 0;
00354
00355           // main height
00356           for (int i=0; i < lhH.quot; i++) {
00357                   // main width
00358                   hlBase2 = hlBase;
00359                   lhBase2 = lhBase;
00360                   for (int j=0; j < hlW.quot; j++) {
00361                           hlPos = hlBase2;
00362                           lhPos = lhBase2;
00363                           for (int y=0; y < InterBlockSize; y++) {
00364                                   for (int x=0; x < InterBlockSize; x++) {
00365                                           DequantizeValue(hlBand, hlPos,
quantParam);
00366                                           DequantizeValue(lhBand, lhPos,
quantParam);
00367                                           hlPos++;
00368                                           lhPos++;
```

```
00369                                }
00370                                hlPos += hlws;
00371                                lhPos += lhws;
00372                        }
00373                        hlBase2 += InterBlockSize;
00374                        lhBase2 += InterBlockSize;
00375                }
00376                // rest of width
00377                hlPos = hlBase2;
00378                lhPos = lhBase2;
00379                for (int y=0; y < InterBlockSize; y++) {
00380                        for (int x=0; x < hlW.rem; x++) {
00381                                DequantizeValue(hlBand, hlPos, quantParam);
00382                                DequantizeValue(lhBand, lhPos, quantParam);
00383                                hlPos++;
00384                                lhPos++;
00385                        }
00386                        // width difference between HL and LH
00387                        if (lhBand->GetWidth() > hlBand->GetWidth()) {
00388                                DequantizeValue(lhBand, lhPos, quantParam);
00389                        }
00390                        hlPos += hlwr;
00391                        lhPos += lhwr;
00392                        hlBase += hlBand->GetWidth();
00393                        lhBase += lhBand->GetWidth();
00394                }
00395        }
00396        // main width
00397        hlBase2 = hlBase;
00398        lhBase2 = lhBase;
00399        for (int j=0; j < hlW.quot; j++) {
00400                // rest of height
00401                hlPos = hlBase2;
00402                lhPos = lhBase2;
00403                for (int y=0; y < lhH.rem; y++) {
00404                        for (int x=0; x < InterBlockSize; x++) {
00405                                DequantizeValue(hlBand, hlPos, quantParam);
00406                                DequantizeValue(lhBand, lhPos, quantParam);
00407                                hlPos++;
00408                                lhPos++;
00409                        }
00410                        hlPos += hlws;
00411                        lhPos += lhws;
00412                }
00413                hlBase2 += InterBlockSize;
00414                lhBase2 += InterBlockSize;
00415        }
00416        // rest of height
00417        hlPos = hlBase2;
00418        lhPos = lhBase2;
00419        for (int y=0; y < lhH.rem; y++) {
00420                // rest of width
00421                for (int x=0; x < hlW.rem; x++) {
00422                        DequantizeValue(hlBand, hlPos, quantParam);
00423                        DequantizeValue(lhBand, lhPos, quantParam);
00424                        hlPos++;
00425                        lhPos++;
00426                }
00427                // width difference between HL and LH
00428                if (lhBand->GetWidth() > hlBand->GetWidth()) {
00429                        DequantizeValue(lhBand, lhPos, quantParam);
00430                }
00431                hlPos += hlwr;
00432                lhPos += lhwr;
00433                hlBase += hlBand->GetWidth();
00434        }
00435        // height difference between HL and LH
00436        if (hlBand->GetHeight() > lhBand->GetHeight()) {
00437                // total width
00438                hlPos = hlBase;
00439                for (int j=0; j < hlBand->GetWidth(); j++) {
00440                        DequantizeValue(hlBand, hlPos, quantParam);
00441                        hlPos++;
00442                }
00443        }
00444 }
```

void CDecoder::DequantizeValue (CSubband * *band*, UINT32 *bandPos*, int *quantParam*)

Dequantization of a single value at given position in subband. It might throw an **IOException**.

Parameters

band	A subband
bandPos	A valid position in subband band
quantParam	The quantization parameter

Dequantization of a single value at given position in subband. If encoded data is available, then stores dequantized band value into buffer m_value at position m_valuePos. Otherwise reads encoded data block and decodes it. It might throw an **IOException**.

Parameters

band	A subband
bandPos	A valid position in subband band
quantParam	The quantization parameter

Definition at line **462** of file **Decoder.cpp**.

```
00462
{
00463         ASSERT(m_currentBlock);
00464
00465         if (m_currentBlock->IsCompletelyRead()) {
00466                 // all data of current macro block has been read --> prepare next
macro block
00467                 GetNextMacroBlock();
00468         }
00469
00470         band->SetData(bandPos,
m_currentBlock->m_value[m_currentBlock->m_valuePos] << quantParam);
00471         m_currentBlock->m_valuePos++;
00472 }
```

UINT32 CDecoder::GetEncodedHeaderLength () const [inline]

Returns the length of all encoded headers in bytes.

Returns

The length of all encoded headers in bytes

Definition at line **136** of file **Decoder.h**.

```
00136 { return m_encodedHeaderLength; }
```

void CDecoder::GetNextMacroBlock ()

Gets next macro block It might throw an **IOException**.

Definition at line **477** of file **Decoder.cpp**.

```
00477                         {
00478         // current block has been read --> prepare next current block
00479         m_macroBlocksAvailable--;
00480
00481         if (m_macroBlocksAvailable > 0) {
00482                 m_currentBlock = m_macroBlocks[++m_currentBlockIndex];
00483         } else {
00484                 DecodeBuffer();
00485         }
00486         ASSERT(m_currentBlock);
00487 }
```

CPGFStream * CDecoder::GetStream () [inline]

Returns

Stream

Definition at line **174** of file **Decoder.h**.

```
00174 { return m_stream; }
```

void CDecoder::Partition (CSubband * *band*, int *quantParam*, int *width*, int *height*, int *startPos*, int *pitch*)

Unpartitions a rectangular region of a given subband. Partitioning scheme: The plane is partitioned in squares of side length LinBlockSize. Read wavelet coefficients from the output buffer of a macro block. It might throw an **IOException**.

Parameters

band	A subband
quantParam	Dequantization value
width	The width of the rectangle
height	The height of the rectangle
startPos	The relative subband position of the top left corner of the rectangular region
pitch	The number of bytes in row of the subband

Definition at line **266** of file **Decoder.cpp**.

```
00266
{
00267         ASSERT(band);
00268
00269         const div_t ww = div(width, LinBlockSize);
00270         const div_t hh = div(height, LinBlockSize);
00271         const int ws = pitch - LinBlockSize;
00272         const int wr = pitch - ww.rem;
00273         int pos, base = startPos, base2;
00274
00275         // main height
00276         for (int i=0; i < hh.quot; i++) {
00277                 // main width
00278                 base2 = base;
00279                 for (int j=0; j < ww.quot; j++) {
00280                         pos = base2;
00281                         for (int y=0; y < LinBlockSize; y++) {
00282                                 for (int x=0; x < LinBlockSize; x++) {
00283                                         DequantizeValue(band, pos,
quantParam);
00284                                         pos++;
00285                                 }
00286                                 pos += ws;
00287                         }
00288                         base2 += LinBlockSize;
00289                 }
00290                 // rest of width
00291                 pos = base2;
00292                 for (int y=0; y < LinBlockSize; y++) {
00293                         for (int x=0; x < ww.rem; x++) {
00294                                 DequantizeValue(band, pos, quantParam);
00295                                 pos++;
00296                         }
00297                         pos += wr;
00298                         base += pitch;
00299                 }
00300         }
00301         // main width
00302         base2 = base;
00303         for (int j=0; j < ww.quot; j++) {
00304                 // rest of height
00305                 pos = base2;
00306                 for (int y=0; y < hh.rem; y++) {
00307                         for (int x=0; x < LinBlockSize; x++) {
00308                                 DequantizeValue(band, pos, quantParam);
00309                                 pos++;
00310                         }
00311                         pos += ws;
00312                 }
00313                 base2 += LinBlockSize;
```

```
00314           }
00315           // rest of height
00316           pos = base2;
00317           for (int y=0; y < hh.rem; y++) {
00318                   // rest of width
00319                   for (int x=0; x < ww.rem; x++) {
00320                           DequantizeValue(band, pos, quantParam);
00321                           pos++;
00322                   }
00323                   pos += wr;
00324           }
00325 }
```

UINT32 CDecoder::ReadEncodedData (UINT8 * *target*, UINT32 *len*) const

Copies data from the open stream to a target buffer. It might throw an **IOException**.

Parameters

target	The target buffer
len	The number of bytes to read

Returns

The number of bytes copied to the target buffer

Definition at line **246** of file **Decoder.cpp**.

```
00246                                                                             {
00247           ASSERT(m_stream);
00248
00249           int count = len;
00250           m_stream->Read(&count, target);
00251
00252           return count;
00253 }
```

void CDecoder::ReadMacroBlock (CMacroBlock * *block*)[private]

throws **IOException**

Definition at line **535** of file **Decoder.cpp**.

```
00535                                                         {
00536           ASSERT(block);
00537
00538           UINT16 wordLen;
00539           ROIBlockHeader h(BufferSize);
00540           int count, expected;
00541
00542 #ifdef TRACE
00543           //UINT32 filePos = (UINT32)m_stream->GetPos();
00544           //printf("DecodeBuffer: %d\n", filePos);
00545 #endif
00546
00547           // read wordLen
00548           count = expected = sizeof(UINT16);
00549           m_stream->Read(&count, &wordLen);
00550           if (count != expected) ReturnWithError(MissingData);
00551           wordLen = __VAL(wordLen); // convert wordLen
00552           if (wordLen > BufferSize) ReturnWithError(FormatCannotRead);
00553
00554 #ifdef __PGFROISUPPORT__
00555           // read ROIBlockHeader
00556           if (m_roi) {
00557                   count = expected = sizeof(ROIBlockHeader);
00558                   m_stream->Read(&count, &h.val);
00559                   if (count != expected) ReturnWithError(MissingData);
00560                   h.val = __VAL(h.val); // convert ROIBlockHeader
00561           }
00562 #endif
00563           // save header
00564           block->m_header = h;
00565
00566           // read data
00567           count = expected = wordLen*WordBytes;
```

```
00568         m_stream->Read(&count, block->m_codeBuffer);
00569         if (count != expected) ReturnWithError(MissingData);
00570
00571 #ifdef PGF_USE_BIG_ENDIAN
00572         // convert data
00573         count /= WordBytes;
00574         for (int i=0; i < count; i++) {
00575                 block->m_codeBuffer[i] = __VAL(block->m_codeBuffer[i]);
00576         }
00577 #endif
00578
00579 #ifdef __PGFROISUPPORT__
00580         ASSERT(m_roi && h.rbh.bufferSize <= BufferSize || h.rbh.bufferSize ==
BufferSize);
00581 #else
00582         ASSERT(h.rbh.bufferSize == BufferSize);
00583 #endif
00584 }
```

void CDecoder::SetStreamPosToData () [inline]

Resets stream position to beginning of data block.

Definition at line **144** of file **Decoder.h**.

```
00144 { ASSERT(m_stream); m_stream->SetPos(FSFromStart, m_startPos +
m_encodedHeaderLength); }
```

void CDecoder::SetStreamPosToStart () [inline]

Resets stream position to beginning of PGF pre-header.

Definition at line **140** of file **Decoder.h**.

```
00140 { ASSERT(m_stream); m_stream->SetPos(FSFromStart, m_startPos); }
```

void CDecoder::Skip (UINT64 *offset*)

Skips a given number of bytes in the open stream. It might throw an **IOException**.

Definition at line **449** of file **Decoder.cpp**.

```
00449                              {
00450         m_stream->SetPos(FSFromCurrent, offset);
00451 }
```

Member Data Documentation

CMacroBlock* CDecoder::m_currentBlock [private]

current macro block (used by main thread)

Definition at line **209** of file **Decoder.h**.

int CDecoder::m_currentBlockIndex [private]

index of current macro block

Definition at line **206** of file **Decoder.h**.

UINT32 CDecoder::m_encodedHeaderLength [private]

stream offset from startPos to the beginning of the data part (highest level)

Definition at line **203** of file **Decoder.h**.

int CDecoder::m_macroBlockLen `[private]`

array length

Definition at line **207** of file **Decoder.h**.

CMacroBlock** CDecoder::m_macroBlocks `[private]`

array of macroblocks

Definition at line **205** of file **Decoder.h**.

int CDecoder::m_macroBlocksAvailable `[private]`

number of decoded macro blocks (including currently used macro block)

Definition at line **208** of file **Decoder.h**.

UINT64 CDecoder::m_startPos `[private]`

stream position at the beginning of the PGF pre-header

Definition at line **201** of file **Decoder.h**.

CPGFStream* CDecoder::m_stream `[private]`

input PGF stream

Definition at line **200** of file **Decoder.h**.

UINT64 CDecoder::m_streamSizeEstimation `[private]`

estimation of stream size

Definition at line **202** of file **Decoder.h**.

The documentation for this class was generated from the following files:

- **Decoder.h**
- **Decoder.cpp**

CEncoder Class Reference

PGF encoder.
`#include <Encoder.h>`

Classes

- class **CMacroBlock**
  A macro block is an encoding unit of fixed size (uncoded)

Public Member Functions

- **CEncoder** (**CPGFStream** *stream, **PGFPreHeader** preHeader, **PGFHeader** header, const **PGFPostHeader** &postHeader, UINT64 &userDataPos, bool useOMP)
- **~CEncoder** ()
  Destructor.
- void **FavorSpeedOverSize** ()
  Encoder favors speed over compression size.
- void **Flush** ()
- void **UpdatePostHeaderSize** (**PGFPreHeader** preHeader)
- UINT32 **WriteLevelLength** (UINT32 *&levelLength)
- UINT32 **UpdateLevelLength** ()
- void **Partition** (**CSubband** *band, int width, int height, int startPos, int pitch)
- void **SetEncodedLevel** (int currentLevel)
- void **WriteValue** (**CSubband** *band, int bandPos)
- INT64 **ComputeHeaderLength** () const
- INT64 **ComputeBufferLength** () const
- INT64 **ComputeOffset** () const
- void **SetStreamPosToStart** ()
  Resets stream position to beginning of PGF pre-header.
- void **SetBufferStartPos** ()
  Save current stream position as beginning of current level.

Private Member Functions

- void **EncodeBuffer** (**ROIBlockHeader** h)
- void **WriteMacroBlock** (**CMacroBlock** *block)

Private Attributes

- **CPGFStream * m_stream**
  output PMF stream
- UINT64 **m_startPosition**
  stream position of PGF start (PreHeader)
- UINT64 **m_levelLengthPos**
  stream position of Metadata

- UINT64 **m_bufferStartPos**
  stream position of encoded buffer

- **CMacroBlock** ** **m_macroBlocks**
  array of macroblocks

- int **m_macroBlockLen**
  array length

- int **m_lastMacroBlock**
  array index of the last created macro block

- **CMacroBlock** * **m_currentBlock**
  current macro block (used by main thread)

- UINT32 * **m_levelLength**
  temporary saves the level index

- int **m_currLevelIndex**
  counts where (=index) to save next value

- UINT8 **m_nLevels**
  number of levels

- bool **m_favorSpeed**
  favor speed over size

- bool **m_forceWriting**
  all macro blocks have to be written into the stream

Detailed Description

PGF encoder.

PGF encoder class.

Author

C. Stamm

Definition at line **46** of file **Encoder.h**.

Constructor & Destructor Documentation

CEncoder::CEncoder (CPGFStream * *stream*, PGFPreHeader *preHeader*, PGFHeader *header*, const PGFPostHeader & *postHeader*, UINT64 & *userDataPos*, bool *useOMP*)

Write pre-header, header, post-Header, and levelLength. It might throw an **IOException**.

Parameters

stream	A PGF stream
preHeader	A already filled in PGF pre-header
header	An already filled in PGF header
postHeader	[in] An already filled in PGF post-header (containing color table, user data, ...)
userDataPos	[out] File position of user data
useOMP	If true, then the encoder will use multi-threading based on openMP

Write pre-header, header, postHeader, and levelLength. It might throw an **IOException**.

Parameters

stream	A PGF stream
preHeader	A already filled in PGF pre-header
header	An already filled in PGF header
postHeader	[in] An already filled in PGF post-header (containing color table, user data, ...)
userDataPos	[out] File position of user data
useOMP	If true, then the encoder will use multi-threading based on openMP

Definition at line **70** of file **Encoder.cpp**.

```
00071 : m_stream(stream)
00072 , m_bufferStartPos(0)
00073 , m_currLevelIndex(0)
00074 , m_nLevels(header.nLevels)
00075 , m_favorSpeed(false)
00076 , m_forceWriting(false)
00077 #ifdef __PGFROISUPPORT__
00078 , m_roi(false)
00079 #endif
00080 {
00081         ASSERT(m_stream);
00082 
00083         int count;
00084         m_lastMacroBlock = 0;
00085         m_levelLength = nullptr;
00086 
00087         // set number of threads
00088 #ifdef LIBPGF_USE_OPENMP
00089         m_macroBlockLen = omp_get_num_procs();
00090 #else
00091         m_macroBlockLen = 1;
00092 #endif
00093 
00094         if (useOMP && m_macroBlockLen > 1) {
00095 #ifdef LIBPGF_USE_OPENMP
00096                 omp_set_num_threads(m_macroBlockLen);
00097 #endif
00098                 // create macro block array
00099                 m_macroBlocks = new(std::nothrow)
CMacroBlock*[m_macroBlockLen];
00100                 if (!m_macroBlocks) ReturnWithError(InsufficientMemory);
00101                 for (int i=0; i < m_macroBlockLen; i++) m_macroBlocks[i] = new
CMacroBlock(this);
00102                 m_currentBlock = m_macroBlocks[m_lastMacroBlock++];
00103         } else {
00104                 m_macroBlocks = 0;
00105                 m_macroBlockLen = 1;
00106                 m_currentBlock = new CMacroBlock(this);
00107         }
00108 
00109         // save file position
00110         m_startPosition = m_stream->GetPos();
00111 
00112         // write preHeader
00113         preHeader.hSize = __VAL(preHeader.hSize);
00114         count = PreHeaderSize;
00115         m_stream->Write(&count, &preHeader);
00116 
00117         // write file header
00118         header.height = __VAL(header.height);
00119         header.width = __VAL(header.width);
00120         count = HeaderSize;
00121         m_stream->Write(&count, &header);
```

```
00122
00123         // write postHeader
00124         if (header.mode == ImageModeIndexedColor) {
00125                 // write color table
00126                 count = ColorTableSize;
00127                 m_stream->Write(&count, (void *)postHeader.clut);
00128         }
00129         // save user data file position
00130         userDataPos = m_stream->GetPos();
00131         if (postHeader.userDataLen) {
00132                 if (postHeader.userData) {
00133                         // write user data
00134                         count = postHeader.userDataLen;
00135                         m_stream->Write(&count, postHeader.userData);
00136                 } else {
00137                         m_stream->SetPos(FSFromCurrent, count);
00138                 }
00139         }
00140
00141         // save level length file position
00142         m_levelLengthPos = m_stream->GetPos();
00143 }
```

CEncoder::~CEncoder ()

Destructor.

Definition at line **147** of file **Encoder.cpp**.

```
00147                         {
00148         if (m_macroBlocks) {
00149                 for (int i=0; i < m_macroBlockLen; i++) delete m_macroBlocks[i];
00150                 delete[] m_macroBlocks;
00151         } else {
00152                 delete m_currentBlock;
00153         }
00154 }
```

Member Function Documentation

INT64 CEncoder::ComputeBufferLength () const [inline]

Compute stream length of encoded buffer.

Returns

encoded buffer length

Definition at line **179** of file **Encoder.h**.

```
00179 { return m_stream->GetPos() - m_bufferStartPos; }
```

INT64 CEncoder::ComputeHeaderLength () const [inline]

Compute stream length of header.

Returns

header length

Definition at line **174** of file **Encoder.h**.

```
00174 { return m_levelLengthPos - m_startPosition; }
```

INT64 CEncoder::ComputeOffset () const [inline]

Compute file offset between real and expected levelLength position.

Returns

file offset

Definition at line **184** of file **Encoder.h**.

```
00184 { return m_stream->GetPos() - m_levelLengthPos; }
```

void CEncoder::EncodeBuffer (ROIBlockHeader *h*)[private]

Definition at line **341** of file **Encoder.cpp**.

```
00341                                       {
00342         ASSERT(m_currentBlock);
00343 #ifdef __PGFROISUPPORT__
00344         ASSERT(m_roi && h.rbh.bufferSize <= BufferSize || h.rbh.bufferSize ==
BufferSize);
00345 #else
00346         ASSERT(h.rbh.bufferSize == BufferSize);
00347 #endif
00348         m_currentBlock->m_header = h;
00349 
00350         // macro block management
00351         if (m_macroBlockLen == 1) {
00352                 m_currentBlock->BitplaneEncode();
00353                 WriteMacroBlock(m_currentBlock);
00354         } else {
00355                 // save last level index
00356                 int lastLevelIndex = m_currentBlock->m_lastLevelIndex;
00357 
00358                 if (m_forceWriting || m_lastMacroBlock == m_macroBlockLen) {
00359                         // encode macro blocks
00360                         /*
00361                         volatile OSError error = NoError;
00362                         #ifdef LIBPGF_USE_OPENMP
00363                         #pragma omp parallel for ordered default(shared)
00364                         #endif
00365                         for (int i=0; i < m_lastMacroBlock; i++) {
00366                                 if (error == NoError) {
00367                                         m_macroBlocks[i]->BitplaneEncode();
00368                                         #ifdef LIBPGF_USE_OPENMP
00369                                         #pragma omp ordered
00370                                         #endif
00371                                         {
00372                                                 try {
00373 
WriteMacroBlock(m_macroBlocks[i]);
00374                                                 } catch (IOException& e) {
00375                                                         error = e.error;
00376                                                 }
00377                                                 delete m_macroBlocks[i];
m_macroBlocks[i] = 0;
00378                                         }
00379                                 }
00380                         }
00381                         if (error != NoError) ReturnWithError(error);
00382                         */
00383 #ifdef LIBPGF_USE_OPENMP
00384                         #pragma omp parallel for default(shared) //no declared
exceptions in next block
00385 #endif
00386                         for (int i=0; i < m_lastMacroBlock; i++) {
00387                                 m_macroBlocks[i]->BitplaneEncode();
00388                         }
00389                         for (int i=0; i < m_lastMacroBlock; i++) {
00390                                 WriteMacroBlock(m_macroBlocks[i]);
00391                         }
00392 
00393                         // prepare for next round
00394                         m_forceWriting = false;
00395                         m_lastMacroBlock = 0;
00396                 }
00397                 // re-initialize macro block
00398                 m_currentBlock = m_macroBlocks[m_lastMacroBlock++];
00399                 m_currentBlock->Init(lastLevelIndex);
00400         }
00401 }
```

void CEncoder::FavorSpeedOverSize ()[inline]

Encoder favors speed over compression size.

Definition at line **121** of file **Encoder.h**.

```
00121 { m_favorSpeed = true; }
```

void CEncoder::Flush ()

Pad buffer with zeros and encode buffer. It might throw an **IOException**.

Definition at line **310** of file **Encoder.cpp**.

```
00310                     {
00311          if (m_currentBlock->m_valuePos > 0) {
00312                  // pad buffer with zeros
00313
memset(&(m_currentBlock->m_value[m_currentBlock->m_valuePos]), 0, (BufferSize -
m_currentBlock->m_valuePos)*DataTSize);
00314                  m_currentBlock->m_valuePos = BufferSize;
00315
00316                  // encode buffer
00317                  m_forceWriting = true;  // makes sure that the following
EncodeBuffer is really written into the stream
00318                  EncodeBuffer(ROIBlockHeader(m_currentBlock->m_valuePos,
true));
00319          }
00320 }
```

void CEncoder::Partition (CSubband * *band*, int *width*, int *height*, int *startPos*, int *pitch*)

Partitions a rectangular region of a given subband. Partitioning scheme: The plane is partitioned in squares of side length LinBlockSize. Write wavelet coefficients from subband into the input buffer of a macro block. It might throw an **IOException**.

Parameters

band	A subband
width	The width of the rectangle
height	The height of the rectangle
startPos	The absolute subband position of the top left corner of the rectangular region
pitch	The number of bytes in row of the subband

Definition at line **246** of file **Encoder.cpp**.

```
00246
{
00247          ASSERT(band);
00248
00249          const div_t hh = div(height, LinBlockSize);
00250          const div_t ww = div(width, LinBlockSize);
00251          const int ws = pitch - LinBlockSize;
00252          const int wr = pitch - ww.rem;
00253          int pos, base = startPos, base2;
00254
00255          // main height
00256          for (int i=0; i < hh.quot; i++) {
00257                  // main width
00258                  base2 = base;
00259                  for (int j=0; j < ww.quot; j++) {
00260                          pos = base2;
00261                          for (int y=0; y < LinBlockSize; y++) {
00262                                  for (int x=0; x < LinBlockSize; x++) {
00263                                          WriteValue(band, pos);
00264                                          pos++;
00265                                  }
00266                                  pos += ws;
00267                          }
00268                          base2 += LinBlockSize;
00269                  }
00270                  // rest of width
00271                  pos = base2;
00272                  for (int y=0; y < LinBlockSize; y++) {
00273                          for (int x=0; x < ww.rem; x++) {
00274                                  WriteValue(band, pos);
00275                                  pos++;
00276                          }
00277                          pos += wr;
```

```
00278                     base += pitch;
00279                 }
00280         }
00281         // main width
00282         base2 = base;
00283         for (int j=0; j < ww.quot; j++) {
00284                 // rest of height
00285                 pos = base2;
00286                 for (int y=0; y < hh.rem; y++) {
00287                         for (int x=0; x < LinBlockSize; x++) {
00288                                 WriteValue(band, pos);
00289                                 pos++;
00290                         }
00291                         pos += ws;
00292                 }
00293                 base2 += LinBlockSize;
00294         }
00295         // rest of height
00296         pos = base2;
00297         for (int y=0; y < hh.rem; y++) {
00298                 // rest of width
00299                 for (int x=0; x < ww.rem; x++) {
00300                         WriteValue(band, pos);
00301                         pos++;
00302                 }
00303                 pos += wr;
00304         }
00305 }
```

void CEncoder::SetBufferStartPos () [inline]

Save current stream position as beginning of current level.

Definition at line **192** of file **Encoder.h**.

```
00192 { m_bufferStartPos = m_stream->GetPos(); }
```

void CEncoder::SetEncodedLevel (int *currentLevel*) [inline]

Informs the encoder about the encoded level.

Parameters

currentLevel	encoded level [0, nLevels)

Definition at line **162** of file **Encoder.h**.

```
00162 { ASSERT(currentLevel >= 0); m_currentBlock->m_lastLevelIndex = m_nLevels -
currentLevel - 1; m_forceWriting = true; }
```

void CEncoder::SetStreamPosToStart () [inline]

Resets stream position to beginning of PGF pre-header.

Definition at line **188** of file **Encoder.h**.

```
00188 { ASSERT(m_stream); m_stream->SetPos(FSFromStart, m_startPosition); }
```

UINT32 CEncoder::UpdateLevelLength ()

Write new levelLength into stream. It might throw an **IOException**.

Returns

Written image bytes.

Definition at line **202** of file **Encoder.cpp**.

```
00202                                     {
00203         UINT64 curPos = m_stream->GetPos(); // end of image
00204 
00205         // set file pos to levelLength
00206         m_stream->SetPos(FSFromStart, m_levelLengthPos);
00207 
00208         if (m_levelLength) {
00209         #ifdef PGF_USE_BIG_ENDIAN
```

```
00210                   UINT32 levelLength;
00211                   int count = WordBytes;
00212
00213                   for (int i=0; i < m_currLevelIndex; i++) {
00214                           levelLength = __VAL(UINT32(m_levelLength[i]));
00215                           m_stream->Write(&count, &levelLength);
00216                   }
00217           #else
00218                   int count = m_currLevelIndex*WordBytes;
00219
00220                   m_stream->Write(&count, m_levelLength);
00221           #endif //PGF_USE_BIG_ENDIAN
00222           } else {
00223                   int count = m_currLevelIndex*WordBytes;
00224                   m_stream->SetPos(FSFromCurrent, count);
00225           }
00226
00227           // begin of image
00228           UINT32 retValue = UINT32(curPos - m_stream->GetPos());
00229
00230           // restore file position
00231           m_stream->SetPos(FSFromStart, curPos);
00232
00233           return retValue;
00234 }
```

void CEncoder::UpdatePostHeaderSize (PGFPreHeader *preHeader*)

Increase post-header size and write new size into stream.

Parameters

preHeader	An already filled in PGF pre-header It might throw an **IOException**.

Definition at line **160** of file **Encoder.cpp**.

```
00160                                                                 {
00161           UINT64 curPos = m_stream->GetPos(); // end of user data
00162           int count = PreHeaderSize;
00163
00164           // write preHeader
00165           SetStreamPosToStart();
00166           preHeader.hSize = __VAL(preHeader.hSize);
00167           m_stream->Write(&count, &preHeader);
00168
00169           m_stream->SetPos(FSFromStart, curPos);
00170 }
```

UINT32 CEncoder::WriteLevelLength (UINT32 *& *levelLength*)

Create level length data structure and write a place holder into stream. It might throw an **IOException**.

Parameters

levelLength	A reference to an integer array, large enough to save the relative file positions of all PGF levels

Returns

number of bytes written into stream

Definition at line **177** of file **Encoder.cpp**.

```
00177                                                         {
00178           // renew levelLength
00179           delete[] levelLength;
00180           levelLength = new(std::nothrow) UINT32[m_nLevels];
00181           if (!levelLength) ReturnWithError(InsufficientMemory);
00182           for (UINT8 l = 0; l < m_nLevels; l++) levelLength[l] = 0;
00183           m_levelLength = levelLength;
00184
00185           // save level length file position
00186           m_levelLengthPos = m_stream->GetPos();
00187
00188           // write dummy levelLength
00189           int count = m_nLevels*WordBytes;
00190           m_stream->Write(&count, m_levelLength);
```

```
00191
00192         // save current file position
00193         SetBufferStartPos();
00194
00195         return count;
00196 }
```

void CEncoder::WriteMacroBlock (CMacroBlock * *block*)[private]

Definition at line **406** of file **Encoder.cpp**.

```
00406                                                   {
00407         ASSERT(block);
00408 #ifdef __PGFROISUPPORT__
00409         ROIBlockHeader h = block->m_header;
00410 #endif
00411         UINT16 wordLen = UINT16(NumberOfWords(block->m_codePos));
ASSERT(wordLen <= CodeBufferLen);
00412         int count = sizeof(UINT16);
00413
00414 #ifdef TRACE
00415         //UINT32 filePos = (UINT32)m_stream->GetPos();
00416         //printf("EncodeBuffer: %d\n", filePos);
00417 #endif
00418
00419 #ifdef PGF_USE_BIG_ENDIAN
00420         // write wordLen
00421         UINT16 wl = __VAL(wordLen);
00422         m_stream->Write(&count, &wl); ASSERT(count == sizeof(UINT16));
00423
00424 #ifdef __PGFROISUPPORT__
00425         // write ROIBlockHeader
00426         if (m_roi) {
00427                 count = sizeof(ROIBlockHeader);
00428                 h.val = __VAL(h.val);
00429                 m_stream->Write(&count, &h.val); ASSERT(count ==
sizeof(ROIBlockHeader));
00430         }
00431 #endif // __PGFROISUPPORT__
00432
00433         // convert data
00434         for (int i=0; i < wordLen; i++) {
00435                 block->m_codeBuffer[i] = __VAL(block->m_codeBuffer[i]);
00436         }
00437 #else
00438         // write wordLen
00439         m_stream->Write(&count, &wordLen); ASSERT(count == sizeof(UINT16));
00440
00441 #ifdef __PGFROISUPPORT__
00442         // write ROIBlockHeader
00443         if (m_roi) {
00444                 count = sizeof(ROIBlockHeader);
00445                 m_stream->Write(&count, &h.val); ASSERT(count ==
sizeof(ROIBlockHeader));
00446         }
00447 #endif // __PGFROISUPPORT__
00448 #endif // PGF_USE_BIG_ENDIAN
00449
00450         // write encoded data into stream
00451         count = wordLen*WordBytes;
00452         m_stream->Write(&count, block->m_codeBuffer);
00453
00454         // store levelLength
00455         if (m_levelLength) {
00456                 // store level length
00457                 // EncodeBuffer has been called after m_lastLevelIndex has been
updated
00458                 ASSERT(m_currLevelIndex < m_nLevels);
00459                 m_levelLength[m_currLevelIndex] +=
(UINT32)ComputeBufferLength();
00460                 m_currLevelIndex = block->m_lastLevelIndex + 1;
00461
00462         }
00463
00464         // prepare for next buffer
```

```
00465         SetBufferStartPos();
00466
00467         // reset values
00468         block->m_valuePos = 0;
00469         block->m_maxAbsValue = 0;
00470 }
```

void CEncoder::WriteValue (CSubband * *band*, int *bandPos*)

Write a single value into subband at given position. It might throw an **IOException**.

Parameters

band	A subband
bandPos	A valid position in subband band

Definition at line **326** of file **Encoder.cpp**.

```
00326                                                          {
00327         if (m_currentBlock->m_valuePos == BufferSize) {
00328                 EncodeBuffer(ROIBlockHeader(BufferSize, false));
00329         }
00330         DataT val = m_currentBlock->m_value[m_currentBlock->m_valuePos++] =
band->GetData(bandPos);
00331         UINT32 v = abs(val);
00332         if (v > m_currentBlock->m_maxAbsValue) m_currentBlock->m_maxAbsValue =
v;
00333 }
```

Member Data Documentation

UINT64 CEncoder::m_bufferStartPos [private]

stream position of encoded buffer

Definition at line **216** of file **Encoder.h**.

CMacroBlock* CEncoder::m_currentBlock [private]

current macro block (used by main thread)

Definition at line **221** of file **Encoder.h**.

int CEncoder::m_currLevelIndex [private]

counts where (=index) to save next value

Definition at line **224** of file **Encoder.h**.

bool CEncoder::m_favorSpeed [private]

favor speed over size

Definition at line **226** of file **Encoder.h**.

bool CEncoder::m_forceWriting [private]

all macro blocks have to be written into the stream

Definition at line **227** of file **Encoder.h**.

int CEncoder::m_lastMacroBlock [private]

array index of the last created macro block

Definition at line **220** of file **Encoder.h**.

UINT32* CEncoder::m_levelLength [private]

temporary saves the level index

Definition at line **223** of file **Encoder.h**.

UINT64 CEncoder::m_levelLengthPos [private]

stream position of Metadata

Definition at line **215** of file **Encoder.h**.

int CEncoder::m_macroBlockLen [private]

array length

Definition at line **219** of file **Encoder.h**.

CMacroBlock** CEncoder::m_macroBlocks [private]

array of macroblocks

Definition at line **218** of file **Encoder.h**.

UINT8 CEncoder::m_nLevels [private]

number of levels

Definition at line **225** of file **Encoder.h**.

UINT64 CEncoder::m_startPosition [private]

stream position of PGF start (PreHeader)

Definition at line **214** of file **Encoder.h**.

CPGFStream* CEncoder::m_stream [private]

output PMF stream

Definition at line **213** of file **Encoder.h**.

The documentation for this class was generated from the following files:

- **Encoder.h**
- **Encoder.cpp**

CDecoder::CMacroBlock Class Reference

A macro block is a decoding unit of fixed size (uncoded)

Public Member Functions

- **CMacroBlock** ()
  Constructor: Initializes new macro block.

- bool **IsCompletelyRead** () const
- void **BitplaneDecode** ()

Public Attributes

- **ROIBlockHeader m_header**
  block header

- **DataT m_value** [**BufferSize**]
  output buffer of values with index m_valuePos

- UINT32 **m_codeBuffer** [**CodeBufferLen**]
  input buffer for encoded bitstream

- UINT32 **m_valuePos**
  current position in m_value

Private Member Functions

- UINT32 **ComposeBitplane** (UINT32 bufferSize, **DataT** planeMask, UINT32 *sigBits, UINT32 *refBits, UINT32 *signBits)
- UINT32 **ComposeBitplaneRLD** (UINT32 bufferSize, **DataT** planeMask, UINT32 sigPos, UINT32 *refBits)
- UINT32 **ComposeBitplaneRLD** (UINT32 bufferSize, **DataT** planeMask, UINT32 *sigBits, UINT32 *refBits, UINT32 signPos)
- void **SetBitAtPos** (UINT32 pos, **DataT** planeMask)
- void **SetSign** (UINT32 pos, bool sign)

Private Attributes

- bool **m_sigFlagVector** [**BufferSize**+1]

Detailed Description

A macro block is a decoding unit of fixed size (uncoded)

PGF decoder macro block class.

Author

C. Stamm, I. Bauersachs

Definition at line **51** of file **Decoder.h**.

Constructor & Destructor Documentation

CDecoder::CMacroBlock::CMacroBlock () [inline]

Constructor: Initializes new macro block.

Definition at line **55** of file **Decoder.h**.

```
00056                   : m_header(0)
// makes sure that IsCompletelyRead() returns true for an empty macro block
00057 #pragma warning( suppress : 4351 )
00058                   , m_value()
00059                   , m_codeBuffer()
00060                   , m_valuePos(0)
00061                   , m_sigFlagVector()
00062                   {
00063                   }
```

Member Function Documentation

void CDecoder::CMacroBlock::BitplaneDecode ()

Decodes already read input data into this macro block. Several macro blocks can be decoded in parallel. Call **CDecoder::ReadMacroBlock** before this method.

Definition at line **650** of file **Decoder.cpp**.

```
00650                                         {
00651           UINT32 bufferSize = m_header.rbh.bufferSize; ASSERT(bufferSize <=
BufferSize);
00652
00653           // clear significance vector
00654           for (UINT32 k=0; k < bufferSize; k++) {
00655                   m_sigFlagVector[k] = false;
00656           }
00657           m_sigFlagVector[bufferSize] = true; // sentinel
00658
00659           // clear output buffer
00660           for (UINT32 k=0; k < BufferSize; k++) {
00661                   m_value[k] = 0;
00662           }
00663
00664           // read number of bit planes
00665           // <nPlanes>
00666           UINT32 nPlanes = GetValueBlock(m_codeBuffer, 0, MaxBitPlanesLog);
00667           UINT32 codePos = MaxBitPlanesLog;
00668
00669           // loop through all bit planes
00670           if (nPlanes == 0) nPlanes = MaxBitPlanes + 1;
00671           ASSERT(0 < nPlanes && nPlanes <= MaxBitPlanes + 1);
00672           DataT planeMask = 1 << (nPlanes - 1);
00673
00674           for (int plane = nPlanes - 1; plane >= 0; plane--) {
00675                   UINT32 sigLen = 0;
00676
00677                   // read RL code
00678                   if (GetBit(m_codeBuffer, codePos)) {
00679                           // RL coding of sigBits is used
00680                           // <1><codeLen><codedSigAndSignBits>_<refBits>
00681                           codePos++;
00682
00683                           // read codeLen
00684                           UINT32 codeLen = GetValueBlock(m_codeBuffer, codePos,
RLblockSizeLen); ASSERT(codeLen <= MaxCodeLen);
00685
00686                           // position of encoded sigBits and signBits
00687                           UINT32 sigPos = codePos + RLblockSizeLen; ASSERT(sigPos
< CodeBufferBitLen);
00688
00689                           // refinement bits
00690                           codePos = AlignWordPos(sigPos + codeLen);
ASSERT(codePos < CodeBufferBitLen);
```

```
00691
00692                           // run-length decode significant bits and signs from
m_codeBuffer and
00693                           // read refinement bits from m_codeBuffer and compose
bit plane
00694                           sigLen = ComposeBitplaneRLD(bufferSize, planeMask,
sigPos, &m_codeBuffer[codePos >> WordWidthLog]);
00695
00696                   } else {
00697                           // no RL coding is used for sigBits and signBits together
00698                           // <0><sigLen>
00699                           codePos++;
00700
00701                           // read sigLen
00702                           sigLen = GetValueBlock(m_codeBuffer, codePos,
RLblockSizeLen); ASSERT(sigLen <= MaxCodeLen);
00703                           codePos += RLblockSizeLen; ASSERT(codePos <
CodeBufferBitLen);
00704
00705                           // read RL code for signBits
00706                           if (GetBit(m_codeBuffer, codePos)) {
00707                                   // RL coding is used just for signBits
00708                                   //
<1><codeLen><codedSignBits>_<sigBits>_<refBits>
00709                                   codePos++;
00710
00711                                   // read codeLen
00712                                   UINT32 codeLen = GetValueBlock(m_codeBuffer,
codePos, RLblockSizeLen); ASSERT(codeLen <= MaxCodeLen);
00713
00714                                   // sign bits
00715                                   UINT32 signPos = codePos + RLblockSizeLen;
ASSERT(signPos < CodeBufferBitLen);
00716
00717                                   // significant bits
00718                                   UINT32 sigPos = AlignWordPos(signPos +
codeLen); ASSERT(sigPos < CodeBufferBitLen);
00719
00720                                   // refinement bits
00721                                   codePos = AlignWordPos(sigPos + sigLen);
ASSERT(codePos < CodeBufferBitLen);
00722
00723                                   // read significant and refinement bitset from
m_codeBuffer
00724                                   sigLen = ComposeBitplaneRLD(bufferSize,
planeMask, &m_codeBuffer[sigPos >> WordWidthLog], &m_codeBuffer[codePos >>
WordWidthLog], signPos);
00725
00726                           } else {
00727                                   // RL coding of signBits was not efficient and
therefore not used
00728                                   //
<0><signLen>_<signBits>_<sigBits>_<refBits>
00729                                   codePos++;
00730
00731                                   // read signLen
00732                                   UINT32 signLen = GetValueBlock(m_codeBuffer,
codePos, RLblockSizeLen); ASSERT(signLen <= MaxCodeLen);
00733
00734                                   // sign bits
00735                                   UINT32 signPos = AlignWordPos(codePos +
RLblockSizeLen); ASSERT(signPos < CodeBufferBitLen);
00736
00737                                   // significant bits
00738                                   UINT32 sigPos = AlignWordPos(signPos +
signLen); ASSERT(sigPos < CodeBufferBitLen);
00739
00740                                   // refinement bits
00741                                   codePos = AlignWordPos(sigPos + sigLen);
ASSERT(codePos < CodeBufferBitLen);
00742
00743                                   // read significant and refinement bitset from
m_codeBuffer
00744                                   sigLen = ComposeBitplane(bufferSize,
planeMask, &m_codeBuffer[sigPos >> WordWidthLog], &m_codeBuffer[codePos >>
WordWidthLog], &m_codeBuffer[signPos >> WordWidthLog]);
00745                           }
```

```
00746                   }
00747
00748                   // start of next chunk
00749                   codePos = AlignWordPos(codePos + bufferSize - sigLen);
ASSERT(codePos < CodeBufferBitLen);
00750
00751                   // next plane
00752                   planeMask >>= 1;
00753           }
00754
00755           m_valuePos = 0;
00756 }
```

UINT32 CDecoder::CMacroBlock::ComposeBitplane (UINT32 *bufferSize*, DataT *planeMask*, UINT32 * *sigBits*, UINT32 * *refBits*, UINT32 * *signBits*) [private]

Definition at line **763** of file **Decoder.cpp**.

```
00763
{
00764           ASSERT(sigBits);
00765           ASSERT(refBits);
00766           ASSERT(signBits);
00767
00768           UINT32 valPos = 0, signPos = 0, refPos = 0, sigPos = 0;
00769
00770           while (valPos < bufferSize) {
00771                   // search next 1 in m_sigFlagVector using searching with sentinel
00772                   UINT32 sigEnd = valPos;
00773                   while(!m_sigFlagVector[sigEnd]) { sigEnd++; }
00774                   sigEnd -= valPos;
00775                   sigEnd += sigPos;
00776
00777                   // search 1's in sigBits[sigPos..sigEnd)
00778                   // these 1's are significant bits
00779                   while (sigPos < sigEnd) {
00780                           // search 0's
00781                           UINT32 zerocnt = SeekBitRange(sigBits, sigPos, sigEnd
- sigPos);
00782                           sigPos += zerocnt;
00783                           valPos += zerocnt;
00784                           if (sigPos < sigEnd) {
00785                                   // write bit to m_value
00786                                   SetBitAtPos(valPos, planeMask);
00787
00788                                   // copy sign bit
00789                                   SetSign(valPos, GetBit(signBits, signPos++));
00790
00791                                   // update significance flag vector
00792                                   m_sigFlagVector[valPos++] = true;
00793                                   sigPos++;
00794                           }
00795                   }
00796                   // refinement bit
00797                   if (valPos < bufferSize) {
00798                           // write one refinement bit
00799                           if (GetBit(refBits, refPos)) {
00800                                   SetBitAtPos(valPos, planeMask);
00801                           }
00802                           refPos++;
00803                           valPos++;
00804                   }
00805           }
00806           ASSERT(sigPos <= bufferSize);
00807           ASSERT(refPos <= bufferSize);
00808           ASSERT(signPos <= bufferSize);
00809           ASSERT(valPos == bufferSize);
00810
00811           return sigPos;
00812 }
```

UINT32 CDecoder::CMacroBlock::ComposeBitplaneRLD (UINT32 *bufferSize*, DataT *planeMask*, UINT32 * *sigBits*, UINT32 * *refBits*, UINT32 *signPos*) [private]

Definition at line **927** of file **Decoder.cpp**.

```
00927
{
00928           ASSERT(sigBits);
00929           ASSERT(refBits);
00930
00931           UINT32 valPos = 0, refPos = 0;
00932           UINT32 sigPos = 0, sigEnd;
00933           UINT32 zerocnt, count = 0;
00934           UINT32 k = 0;
00935           UINT32 runlen = 1 << k; // = 2^k
00936           bool signBit = false;
00937           bool zeroAfterRun = false;
00938
00939           while (valPos < bufferSize) {
00940                   // search next 1 in m_sigFlagVector using searching with sentinel
00941                   sigEnd = valPos;
00942                   while(!m_sigFlagVector[sigEnd]) { sigEnd++; }
00943                   sigEnd -= valPos;
00944                   sigEnd += sigPos;
00945
00946                   // search 1's in sigBits[sigPos..sigEnd)
00947                   // these 1's are significant bits
00948                   while (sigPos < sigEnd) {
00949                           // search 0's
00950                           zerocnt = SeekBitRange(sigBits, sigPos, sigEnd -
sigPos);
00951                           sigPos += zerocnt;
00952                           valPos += zerocnt;
00953                           if (sigPos < sigEnd) {
00954                                   // write bit to m_value
00955                                   SetBitAtPos(valPos, planeMask);
00956
00957                                   // check sign bit
00958                                   if (count == 0) {
00959                                           // all 1's have been set
00960                                           if (zeroAfterRun) {
00961                                                   // finish the run with a 0
00962                                                   signBit = false;
00963                                                   zeroAfterRun = false;
00964                                           } else {
00965                                                   // decode next sign bit
00966                                                   if (GetBit(m_codeBuffer,
signPos++)) {
00967                                                           // generate 1's run of
length 2^k
00968                                                           count = runlen - 1;
00969                                                           signBit = true;
00970
00971                                                           // adapt k (double
run-length interval)
00972                                                           if (k < WordWidth) {
00973                                                                   k++;
00974                                                                   runlen <<= 1;
00975                                                           }
00976                                                   } else {
00977                                                           // extract counter and
generate 1's run of length count
00978                                                           if (k > 0) {
00979                                                                   // extract
counter
00980                                                                   count =
GetValueBlock(m_codeBuffer, signPos, k);
00981                                                                   signPos += k;
00982
00983                                                                   // adapt k
(half run-length interval)
00984                                                                   k--;
00985                                                                   runlen >>= 1;
00986                                                           }
00987                                                           if (count > 0) {
```

```
00988                                                                                                    count--;
00989                                                                                                    signBit =
true;
00990                                                                                                    zeroAfterRun
= true;
00991                                                                                      } else {
00992                                                                                                    signBit =
false;
00993                                                                                      }
00994                                                                        }
00995                                                          }
00996                                           } else {
00997                                                          ASSERT(count > 0);
00998                                                          ASSERT(signBit);
00999                                                          count--;
01000                                           }
01001
01002                                           // copy sign bit
01003                                           SetSign(valPos, signBit);
01004
01005                                           // update significance flag vector
01006                                           m_sigFlagVector[valPos++] = true;
01007                                           sigPos++;
01008                             }
01009               }
01010
01011               // refinement bit
01012               if (valPos < bufferSize) {
01013                             // write one refinement bit
01014                             if (GetBit(refBits, refPos)) {
01015                                           SetBitAtPos(valPos, planeMask);
01016                             }
01017                             refPos++;
01018                             valPos++;
01019               }
01020        }
01021        ASSERT(sigPos <= bufferSize);
01022        ASSERT(refPos <= bufferSize);
01023        ASSERT(valPos == bufferSize);
01024
01025        return sigPos;
01026 }
```

UINT32 CDecoder::CMacroBlock::ComposeBitplaneRLD (UINT32 *bufferSize*, DataT *planeMask*, UINT32 *sigPos*, UINT32 * *refBits*) `[private]`

Definition at line **824** of file **Decoder.cpp**.

```
00824
{
00825        ASSERT(refBits);
00826
00827        UINT32 valPos = 0, refPos = 0;
00828        UINT32 sigPos = 0, sigEnd;
00829        UINT32 k = 3;
00830        UINT32 runlen = 1 << k; // = 2^k
00831        UINT32 count = 0, rest = 0;
00832        bool set1 = false;
00833
00834        while (valPos < bufferSize) {
00835               // search next 1 in m_sigFlagVector using searching with sentinel
00836               sigEnd = valPos;
00837               while(!m_sigFlagVector[sigEnd]) { sigEnd++; }
00838               sigEnd -= valPos;
00839               sigEnd += sigPos;
00840
00841               while (sigPos < sigEnd) {
00842                             if (rest || set1) {
00843                                           // rest of last run
00844                                           sigPos += rest;
00845                                           valPos += rest;
00846                                           rest = 0;
00847                             } else {
00848                                           // decode significant bits
00849                                           if (GetBit(m_codeBuffer, codePos++)) {
```

```
00850                                         // extract counter and generate zero
run of length count
00851                                         if (k > 0) {
00852                                                 // extract counter
00853                                                 count =
GetValueBlock(m_codeBuffer, codePos, k);
00854                                                 codePos += k;
00855                                                 if (count > 0) {
00856                                                         sigPos += count;
00857                                                         valPos += count;
00858                                                 }
00859
00860                                                 // adapt k (half run-length
interval)
00861                                                 k--;
00862                                                 runlen >>= 1;
00863                                         }
00864
00865                                         set1 = true;
00866
00867                                 } else {
00868                                         // generate zero run of length 2^k
00869                                         sigPos += runlen;
00870                                         valPos += runlen;
00871
00872                                         // adapt k (double run-length interval)
00873                                         if (k < WordWidth) {
00874                                                 k++;
00875                                                 runlen <<= 1;
00876                                         }
00877                                 }
00878                         }
00879
00880                         if (sigPos < sigEnd) {
00881                                 if (set1) {
00882                                         set1 = false;
00883
00884                                         // write 1 bit
00885                                         SetBitAtPos(valPos, planeMask);
00886
00887                                         // set sign bit
00888                                         SetSign(valPos, GetBit(m_codeBuffer,
codePos++));
00889
00890                                         // update significance flag vector
00891                                         m_sigFlagVector[valPos++] = true;
00892                                         sigPos++;
00893                                 }
00894                         } else {
00895                                 rest = sigPos - sigEnd;
00896                                 sigPos = sigEnd;
00897                                 valPos -= rest;
00898                         }
00899
00900                 }
00901
00902                 // refinement bit
00903                 if (valPos < bufferSize) {
00904                         // write one refinement bit
00905                         if (GetBit(refBits, refPos)) {
00906                                 SetBitAtPos(valPos, planeMask);
00907                         }
00908                         refPos++;
00909                         valPos++;
00910                 }
00911         }
00912         ASSERT(sigPos <= bufferSize);
00913         ASSERT(refPos <= bufferSize);
00914         ASSERT(valPos == bufferSize);
00915
00916         return sigPos;
00917 }
```

bool CDecoder::CMacroBlock::IsCompletelyRead () const [inline]

Returns true if this macro block has been completely read.

Returns

true if current value position is at block end

Definition at line **68** of file **Decoder.h**.

```
00068 { return m_valuePos >= m_header.rbh.bufferSize; }
```

void CDecoder::CMacroBlock::SetBitAtPos (UINT32 *pos*, DataT *planeMask*)[inline], [private]

Definition at line **85** of file **Decoder.h**.

```
00085 { (m_value[pos] >= 0) ? m_value[pos] |= planeMask : m_value[pos] -= planeMask;
}
```

void CDecoder::CMacroBlock::SetSign (UINT32 *pos*, bool *sign*)[inline], [private]

Definition at line **86** of file **Decoder.h**.

```
00086 { m_value[pos] = -m_value[pos]*sign + m_value[pos]*(!sign); }
```

Member Data Documentation

UINT32 CDecoder::CMacroBlock::m_codeBuffer[CodeBufferLen]

input buffer for encoded bitstream

Definition at line **78** of file **Decoder.h**.

ROIBlockHeader CDecoder::CMacroBlock::m_header

block header

Definition at line **76** of file **Decoder.h**.

bool CDecoder::CMacroBlock::m_sigFlagVector[BufferSize+1][private]

Definition at line **88** of file **Decoder.h**.

DataT CDecoder::CMacroBlock::m_value[BufferSize]

output buffer of values with index m_valuePos

Definition at line **77** of file **Decoder.h**.

UINT32 CDecoder::CMacroBlock::m_valuePos

current position in m_value

Definition at line **79** of file **Decoder.h**.

The documentation for this class was generated from the following files:

- **Decoder.h**
- **Decoder.cpp**

CEncoder::CMacroBlock Class Reference

A macro block is an encoding unit of fixed size (uncoded)

Public Member Functions

- **CMacroBlock** (**CEncoder** *encoder)
- void **Init** (int lastLevelIndex)
- void **BitplaneEncode** ()

Public Attributes

- **DataT m_value** [**BufferSize**]
  input buffer of values with index m_valuePos
- UINT32 **m_codeBuffer** [**CodeBufferLen**]
  output buffer for encoded bitstream
- **ROIBlockHeader m_header**
  block header
- UINT32 **m_valuePos**
  current buffer position
- UINT32 **m_maxAbsValue**
  maximum absolute coefficient in each buffer
- UINT32 **m_codePos**
  current position in encoded bitstream
- int **m_lastLevelIndex**
  index of last encoded level: [0, nLevels); used because a level-end can occur before a buffer is full

Private Member Functions

- UINT32 **RLESigns** (UINT32 codePos, UINT32 *signBits, UINT32 signLen)
- UINT32 **DecomposeBitplane** (UINT32 bufferSize, UINT32 planeMask, UINT32 codePos, UINT32 *sigBits, UINT32 *refBits, UINT32 *signBits, UINT32 &signLen, UINT32 &codeLen)
- UINT8 **NumberOfBitplanes** ()
- bool **GetBitAtPos** (UINT32 pos, UINT32 planeMask) const

Private Attributes

- **CEncoder** * **m_encoder**
- bool **m_sigFlagVector** [**BufferSize**+1]

Detailed Description

A macro block is an encoding unit of fixed size (uncoded)

PGF encoder macro block class.

Author

C. Stamm, I. Bauersachs

Definition at line **51** of file **Encoder.h**.

Constructor & Destructor Documentation

CEncoder::CMacroBlock::CMacroBlock (CEncoder * *encoder*) [inline]

Constructor: Initializes new macro block.

Parameters

encoder	Pointer to outer class.

Definition at line **56** of file **Encoder.h**.

```
00057                              : 4351 )
00058                 : m_value()
00059                 , m_codeBuffer()
00060                 , m_header(0)
00061                 , m_encoder(encoder)
00062                 , m_sigFlagVector()
00063                 {
00064                         ASSERT(m_encoder);
00065                         Init(-1);
00066                 }
```

Member Function Documentation

void CEncoder::CMacroBlock::BitplaneEncode ()

Encodes this macro block into internal code buffer. Several macro blocks can be encoded in parallel. Call **CEncoder::WriteMacroBlock** after this method.

Definition at line **482** of file **Encoder.cpp**.

```
00482                                  {
00483         UINT8   nPlanes;
00484         UINT32  sigLen, codeLen = 0, wordPos, refLen, signLen;
00485         UINT32  sigBits[BufferLen] = { 0 };
00486         UINT32  refBits[BufferLen] = { 0 };
00487         UINT32  signBits[BufferLen] = { 0 };
00488         UINT32  planeMask;
00489         UINT32  bufferSize = m_header.rbh.bufferSize; ASSERT(bufferSize <=
BufferSize);
00490         bool    useRL;
00491
00492 #ifdef TRACE
00493         //printf("which thread: %d\n", omp_get_thread_num());
00494 #endif
00495
00496         // clear significance vector
00497         for (UINT32 k=0; k < bufferSize; k++) {
00498                 m_sigFlagVector[k] = false;
00499         }
00500         m_sigFlagVector[bufferSize] = true; // sentinel
00501
00502         // clear output buffer
00503         for (UINT32 k=0; k < bufferSize; k++) {
00504                 m_codeBuffer[k] = 0;
00505         }
00506         m_codePos = 0;
00507
00508         // compute number of bit planes and split buffer into separate bit planes
00509         nPlanes = NumberOfBitplanes();
00510
00511         // write number of bit planes to m_codeBuffer
00512         // <nPlanes>
00513         SetValueBlock(m_codeBuffer, 0, nPlanes, MaxBitPlanesLog);
00514         m_codePos += MaxBitPlanesLog;
```

```
00515
00516         // loop through all bit planes
00517         if (nPlanes == 0) nPlanes = MaxBitPlanes + 1;
00518         planeMask = 1 << (nPlanes - 1);
00519
00520         for (int plane = nPlanes - 1; plane >= 0; plane--) {
00521                 // clear significant bitset
00522                 for (UINT32 k=0; k < BufferLen; k++) {
00523                         sigBits[k] = 0;
00524                 }
00525
00526                 // split bitplane in significant bitset and refinement bitset
00527                 sigLen = DecomposeBitplane(bufferSize, planeMask, m_codePos +
RLblockSizeLen + 1, sigBits, refBits, signBits, signLen, codeLen);
00528
00529                 if (sigLen > 0 && codeLen <= MaxCodeLen && codeLen <
AlignWordPos(sigLen) + AlignWordPos(signLen) + 2*RLblockSizeLen) {
00530                         // set RL code bit
00531                         // <1><codeLen>
00532                         SetBit(m_codeBuffer, m_codePos++);
00533
00534                         // write length codeLen to m_codeBuffer
00535                         SetValueBlock(m_codeBuffer, m_codePos, codeLen,
RLblockSizeLen);
00536                         m_codePos += RLblockSizeLen + codeLen;
00537                 } else {
00538                 #ifdef TRACE
00539                         //printf("new\n");
00540                         //for (UINT32 i=0; i < bufferSize; i++) {
00541                         //      printf("%s", (GetBit(sigBits, i)) ? "1" : "_");
00542                         //      if (i%120 == 119) printf("\n");
00543                         //}
00544                         //printf("\n");
00545                 #endif // TRACE
00546
00547                         // run-length coding wasn't efficient enough
00548                         // we don't use RL coding for sigBits
00549                         // <0><sigLen>
00550                         ClearBit(m_codeBuffer, m_codePos++);
00551
00552                         // write length sigLen to m_codeBuffer
00553                         ASSERT(sigLen <= MaxCodeLen);
00554                         SetValueBlock(m_codeBuffer, m_codePos, sigLen,
RLblockSizeLen);
00555                         m_codePos += RLblockSizeLen;
00556
00557                         if (m_encoder->m_favorSpeed || signLen == 0) {
00558                                 useRL = false;
00559                         } else {
00560                                 // overwrite m_codeBuffer
00561                                 useRL = true;
00562                                 // run-length encode m_sign and append them to
the m_codeBuffer
00563                                 codeLen = RLESigns(m_codePos + RLblockSizeLen
+ 1, signBits, signLen);
00564                         }
00565
00566                         if (useRL && codeLen <= MaxCodeLen && codeLen < signLen)
{
00567                                 // RL encoding of m_sign was efficient
00568                                 // <1><codeLen><codedSignBits>_
00569                                 // write RL code bit
00570                                 SetBit(m_codeBuffer, m_codePos++);
00571
00572                                 // write codeLen to m_codeBuffer
00573                                 SetValueBlock(m_codeBuffer, m_codePos,
codeLen, RLblockSizeLen);
00574
00575                                 // compute position of sigBits
00576                                 wordPos = NumberOfWords(m_codePos +
RLblockSizeLen + codeLen);
00577                                 ASSERT(0 <= wordPos && wordPos <
CodeBufferLen);
00578                         } else {
00579                                 // RL encoding of signBits wasn't efficient
00580                                 // <0><signLen>_<signBits>_
00581                                 // clear RL code bit
```

```
00582                                 ClearBit(m_codeBuffer, m_codePos++);
00583
00584                                 // write signLen to m_codeBuffer
00585                                 ASSERT(signLen <= MaxCodeLen);
00586                                 SetValueBlock(m_codeBuffer, m_codePos,
signLen, RLblockSizeLen);
00587
00588                                 // write signBits to m_codeBuffer
00589                                 wordPos = NumberOfWords(m_codePos +
RLblockSizeLen);
00590                                 ASSERT(0 <= wordPos && wordPos <
CodeBufferLen);
00591                                 codeLen = NumberOfWords(signLen);
00592
00593                                 for (UINT32 k=0; k < codeLen; k++) {
00594                                         m_codeBuffer[wordPos++] =
signBits[k];
00595                                 }
00596                         }
00597
00598                         // write sigBits
00599                         // <sigBits>_
00600                         ASSERT(0 <= wordPos && wordPos < CodeBufferLen);
00601                         refLen = NumberOfWords(sigLen);
00602
00603                         for (UINT32 k=0; k < refLen; k++) {
00604                                 m_codeBuffer[wordPos++] = sigBits[k];
00605                         }
00606                         m_codePos = wordPos << WordWidthLog;
00607                 }
00608
00609                 // append refinement bitset (aligned to word boundary)
00610                 // _<refBits>
00611                 wordPos = NumberOfWords(m_codePos);
00612                 ASSERT(0 <= wordPos && wordPos < CodeBufferLen);
00613                 refLen = NumberOfWords(bufferSize - sigLen);
00614
00615                 for (UINT32 k=0; k < refLen; k++) {
00616                         m_codeBuffer[wordPos++] = refBits[k];
00617                 }
00618                 m_codePos = wordPos << WordWidthLog;
00619                 planeMask >>= 1;
00620         }
00621         ASSERT(0 <= m_codePos && m_codePos <= CodeBufferBitLen);
00622 }
```

UINT32 CEncoder::CMacroBlock::DecomposeBitplane (UINT32 *bufferSize*, UINT32 *planeMask*, UINT32 *codePos*, UINT32 * *sigBits*, UINT32 * *refBits*, UINT32 * *signBits*, UINT32 & *signLen*, UINT32 & *codeLen*) `[private]`

Definition at line **634** of file **Encoder.cpp**.

```
00634
{
00635         ASSERT(sigBits);
00636         ASSERT(refBits);
00637         ASSERT(signBits);
00638         ASSERT(codePos < CodeBufferBitLen);
00639
00640         UINT32 sigPos = 0;
00641         UINT32 valuePos = 0, valueEnd;
00642         UINT32 refPos = 0;
00643
00644         // set output value
00645         signLen = 0;
00646
00647         // prepare RLE of Sigs and Signs
00648         const UINT32 outStartPos = codePos;
00649         UINT32 k = 3;
00650         UINT32 runlen = 1 << k; // = 2^k
00651         UINT32 count = 0;
00652
00653         while (valuePos < bufferSize) {
00654                 // search next 1 in m_sigFlagVector using searching with sentinel
```

```
00655                   valueEnd = valuePos;
00656                   while(!m_sigFlagVector[valueEnd]) { valueEnd++; }
00657
00658                   // search 1's in m_value[plane][valuePos..valueEnd)
00659                   // these 1's are significant bits
00660                   while (valuePos < valueEnd) {
00661                           if (GetBitAtPos(valuePos, planeMask)) {
00662                                   // RLE encoding
00663                                   // encode run of count 0's followed by a 1
00664                                   // with codeword: 1<count>(signBits[signPos])
00665                                   SetBit(m_codeBuffer, codePos++);
00666                                   if (k > 0) {
00667                                           SetValueBlock(m_codeBuffer, codePos,
count, k);
00668                                           codePos += k;
00669
00670                                           // adapt k (half the zero run-length)
00671                                           k--;
00672                                           runlen >>= 1;
00673                                   }
00674
00675                                   // copy and write sign bit
00676                                   if (m_value[valuePos] < 0) {
00677                                           SetBit(signBits, signLen++);
00678                                           SetBit(m_codeBuffer, codePos++);
00679                                   } else {
00680                                           ClearBit(signBits, signLen++);
00681                                           ClearBit(m_codeBuffer, codePos++);
00682                                   }
00683
00684                                   // write a 1 to sigBits
00685                                   SetBit(sigBits, sigPos++);
00686
00687                                   // update m_sigFlagVector
00688                                   m_sigFlagVector[valuePos] = true;
00689
00690                                   // prepare for next run
00691                                   count = 0;
00692                           } else {
00693                                   // RLE encoding
00694                                   count++;
00695                                   if (count == runlen) {
00696                                           // encode run of 2^k zeros by a single
0
00697                                           ClearBit(m_codeBuffer, codePos++);
00698                                           // adapt k (double the zero run-length)
00699                                           if (k < WordWidth) {
00700                                                   k++;
00701                                                   runlen <<= 1;
00702                                           }
00703
00704                                           // prepare for next run
00705                                           count = 0;
00706                                   }
00707
00708                                   // write 0 to sigBits
00709                                   sigPos++;
00710                           }
00711                           valuePos++;
00712                   }
00713                   // refinement bit
00714                   if (valuePos < bufferSize) {
00715                           // write one refinement bit
00716                           if (GetBitAtPos(valuePos++, planeMask)) {
00717                                   SetBit(refBits, refPos);
00718                           } else {
00719                                   ClearBit(refBits, refPos);
00720                           }
00721                           refPos++;
00722                   }
00723           }
00724           // RLE encoding of the rest of the plane
00725           // encode run of count 0's followed by a 1
00726           // with codeword: 1<count>(signBits[signPos])
00727           SetBit(m_codeBuffer, codePos++);
00728           if (k > 0) {
00729                   SetValueBlock(m_codeBuffer, codePos, count, k);
```

```
00730                 codePos += k;
00731         }
00732         // write dmmy sign bit
00733         SetBit(m_codeBuffer, codePos++);
00734 
00735         // write word filler zeros
00736 
00737         ASSERT(sigPos <= bufferSize);
00738         ASSERT(refPos <= bufferSize);
00739         ASSERT(signLen <= bufferSize);
00740         ASSERT(valuePos == bufferSize);
00741         ASSERT(codePos >= outStartPos && codePos < CodeBufferBitLen);
00742         codeLen = codePos - outStartPos;
00743 
00744         return sigPos;
00745 }
```

bool CEncoder::CMacroBlock::GetBitAtPos (UINT32 *pos*, UINT32 *planeMask*) const [inline], [private]

Definition at line **96** of file **Encoder.h**.

```
00096 { return (abs(m_value[pos]) & planeMask) > 0; }
```

void CEncoder::CMacroBlock::Init (int *lastLevelIndex*) [inline]

Reinitialzes this macro block (allows reusage).

Parameters

lastLevelIndex	Level length directory index of last encoded level: [0, nLevels)

Definition at line **71** of file **Encoder.h**.

```
00071                                              {                         //
initialize for reusage
00072                         m_valuePos = 0;
00073                         m_maxAbsValue = 0;
00074                         m_codePos = 0;
00075                         m_lastLevelIndex = lastLevelIndex;
00076                 }
```

UINT8 CEncoder::CMacroBlock::NumberOfBitplanes () [private]

Definition at line **750** of file **Encoder.cpp**.

```
00750                                             {
00751         UINT8 cnt = 0;
00752 
00753         // determine number of bitplanes for max value
00754         if (m_maxAbsValue > 0) {
00755                 while (m_maxAbsValue > 0) {
00756                         m_maxAbsValue >>= 1; cnt++;
00757                 }
00758                 if (cnt == MaxBitPlanes + 1) cnt = 0;
00759                 // end cs
00760                 ASSERT(cnt <= MaxBitPlanes);
00761                 ASSERT((cnt >> MaxBitPlanesLog) == 0);
00762                 return cnt;
00763         } else {
00764                 return 1;
00765         }
00766 }
```

UINT32 CEncoder::CMacroBlock::RLESigns (UINT32 *codePos*, UINT32 * *signBits*, UINT32 *signLen*) [private]

Definition at line **774** of file **Encoder.cpp**.

```
00774
{
00775         ASSERT(signBits);
```

```
00776         ASSERT(0 <= codePos && codePos < CodeBufferBitLen);
00777         ASSERT(0 < signLen && signLen <= BufferSize);
00778
00779         const UINT32  outStartPos = codePos;
00780         UINT32 k = 0;
00781         UINT32 runlen = 1 << k; // = 2^k
00782         UINT32 count = 0;
00783         UINT32 signPos = 0;
00784
00785         while (signPos < signLen) {
00786                 // search next 0 in signBits starting at position signPos
00787                 count = SeekBit1Range(signBits, signPos, __min(runlen, signLen
- signPos));
00788                 // count 1's found
00789                 if (count == runlen) {
00790                         // encode run of 2^k ones by a single 1
00791                         signPos += count;
00792                         SetBit(m_codeBuffer, codePos++);
00793                         // adapt k (double the 1's run-length)
00794                         if (k < WordWidth) {
00795                                 k++;
00796                                 runlen <<= 1;
00797                         }
00798                 } else {
00799                         // encode run of count 1's followed by a 0
00800                         // with codeword: 0(count)
00801                         signPos += count + 1;
00802                         ClearBit(m_codeBuffer, codePos++);
00803                         if (k > 0) {
00804                                 SetValueBlock(m_codeBuffer, codePos, count,
k);
00805                                 codePos += k;
00806                         }
00807                         // adapt k (half the 1's run-length)
00808                         if (k > 0) {
00809                                 k--;
00810                                 runlen >>= 1;
00811                         }
00812                 }
00813         }
00814         ASSERT(signPos == signLen || signPos == signLen + 1);
00815         ASSERT(codePos >= outStartPos && codePos < CodeBufferBitLen);
00816         return codePos - outStartPos;
00817 }
```

Member Data Documentation

UINT32 CEncoder::CMacroBlock::m_codeBuffer[CodeBufferLen]

output buffer for encoded bitstream

Definition at line **85** of file **Encoder.h**.

UINT32 CEncoder::CMacroBlock::m_codePos

current position in encoded bitstream

Definition at line **89** of file **Encoder.h**.

CEncoder* CEncoder::CMacroBlock::m_encoder `[private]`

Definition at line **98** of file **Encoder.h**.

ROIBlockHeader CEncoder::CMacroBlock::m_header

block header

Definition at line **86** of file **Encoder.h**.

int CEncoder::CMacroBlock::m_lastLevelIndex

index of last encoded level: [0, nLevels); used because a level-end can occur before a buffer is full

Definition at line **90** of file **Encoder.h**.

UINT32 CEncoder::CMacroBlock::m_maxAbsValue

maximum absolute coefficient in each buffer

Definition at line **88** of file **Encoder.h**.

bool CEncoder::CMacroBlock::m_sigFlagVector[BufferSize+1] `[private]`

Definition at line **99** of file **Encoder.h**.

DataT CEncoder::CMacroBlock::m_value[BufferSize]

input buffer of values with index m_valuePos

Definition at line **84** of file **Encoder.h**.

UINT32 CEncoder::CMacroBlock::m_valuePos

current buffer position

Definition at line **87** of file **Encoder.h**.

The documentation for this class was generated from the following files:

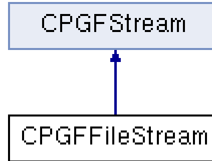
- **Encoder.h**
- **Encoder.cpp**

CPGFFileStream Class Reference

File stream class.

`#include <PGFstream.h>`

Inheritance diagram for CPGFFileStream:



Public Member Functions

- **CPGFFileStream** ()
- **CPGFFileStream** (HANDLE hFile)
- HANDLE **GetHandle** ()
- virtual **~CPGFFileStream** ()
- virtual void **Write** (int *count, void *buffer)
- virtual void **Read** (int *count, void *buffer)
- virtual void **SetPos** (short posMode, INT64 posOff)
- virtual UINT64 **GetPos** () const
- virtual bool **IsValid** () const

Protected Attributes

- HANDLE **m_hFile**
  file handle

Detailed Description

File stream class.

A PGF stream subclass for external storage files.

Author

C. Stamm

Definition at line **82** of file **PGFstream.h**.

Constructor & Destructor Documentation

CPGFFileStream::CPGFFileStream () `[inline]`

Definition at line **87** of file **PGFstream.h**.

```
00087 : m_hFile(0) {}
```

CPGFFileStream::CPGFFileStream (HANDLE *hFile*) `[inline]`

Constructor

Parameters

hFile	File handle

Definition at line **90** of file **PGFstream.h**.

```
00090 : m_hFile(hFile) {}
```

virtual CPGFFileStream::~CPGFFileStream () [inline], [virtual]

Definition at line **94** of file **PGFstream.h**.

```
00094 { m_hFile = 0; }
```

Member Function Documentation

HANDLE CPGFFileStream::GetHandle () [inline]

Returns

File handle

Definition at line **92** of file **PGFstream.h**.

```
00092 { return m_hFile; }
```

UINT64 CPGFFileStream::GetPos () const [virtual]

Get current stream position.

Returns

Current stream position

Implements **CPGFStream** (*p.109*).

Definition at line **64** of file **PGFstream.cpp**.

```
00064                                    {
00065         ASSERT(IsValid());
00066         OSError err;
00067         UINT64 pos = 0;
00068         if ((err = GetFPos(m_hFile, &pos)) != NoError) ReturnWithError2(err,
pos);
00069         return pos;
00070 }
```

virtual bool CPGFFileStream::IsValid () const [inline], [virtual]

Check stream validity.

Returns

True if stream and current position is valid

Implements **CPGFStream** (*p.109*).

Definition at line **99** of file **PGFstream.h**.

```
00099 { return m_hFile != 0; }
```

void CPGFFileStream::Read (int * *count*, void * *buffer*) [virtual]

Read some bytes from this stream and stores them into a buffer.

Parameters

count	A pointer to a value containing the number of bytes should be read. After this call it contains the number of read bytes.
buffer	A memory buffer

Implements **CPGFStream** (*p.109*).

Definition at line **48** of file **PGFstream.cpp**.

```
00048                                                   {
00049         ASSERT(count);
00050         ASSERT(buffPtr);
00051         ASSERT(IsValid());
00052         OSError err;
00053         if ((err = FileRead(m_hFile, count, buffPtr)) != NoError)
ReturnWithError(err);
```

```
00054 }
```

void CPGFFileStream::SetPos (short *posMode*, INT64 *posOff*)[virtual]

Set stream position either absolute or relative.

Parameters

posMode	A position mode (FSFromStart, FSFromCurrent, FSFromEnd)
posOff	A new stream position (absolute positioning) or a position offset (relative positioning)

Implements **CPGFStream** (*p.109*).

Definition at line **57** of file **PGFstream.cpp**.

```
00057                                                     {
00058         ASSERT(IsValid());
00059         OSError err;
00060         if ((err = SetFPos(m_hFile, posMode, posOff)) != NoError)
ReturnWithError(err);
00061 }
```

void CPGFFileStream::Write (int * *count*, void * *buffer*)[virtual]

Write some bytes out of a buffer into this stream.

Parameters

count	A pointer to a value containing the number of bytes should be written. After this call it contains the number of written bytes.
buffer	A memory buffer

Implements **CPGFStream** (*p.109*).

Definition at line **38** of file **PGFstream.cpp**.

```
00038                                                  {
00039         ASSERT(count);
00040         ASSERT(buffPtr);
00041         ASSERT(IsValid());
00042         OSError err;
00043         if ((err = FileWrite(m_hFile, count, buffPtr)) != NoError)
ReturnWithError(err);
00044
00045 }
```

Member Data Documentation

HANDLE CPGFFileStream::m_hFile[protected]

file handle

Definition at line **84** of file **PGFstream.h**.

The documentation for this class was generated from the following files:

- **PGFstream.h**
- **PGFstream.cpp**

CPGFImage Class Reference

PGF main class.
`#include <PGFimage.h>`

Public Member Functions

- **CPGFImage** ()
  Standard constructor.
- virtual **~CPGFImage** ()
  Destructor.
- void **Destroy** ()
- void **Open** (**CPGFStream** *stream)
- bool **IsOpen** () const
  Returns true if the PGF has been opened for reading.
- void **Read** (int level=0, CallbackPtr cb=nullptr, void *data=nullptr)
- void **Read** (**PGFRect** &rect, int level=0, CallbackPtr cb=nullptr, void *data=nullptr)
- void **ReadPreview** ()
- void **Reconstruct** (int level=0)
- void **GetBitmap** (int pitch, UINT8 *buff, BYTE bpp, int channelMap[]=nullptr, CallbackPtr cb=nullptr, void *data=nullptr) const
- void **GetYUV** (int pitch, **DataT** *buff, BYTE bpp, int channelMap[]=nullptr, CallbackPtr cb=nullptr, void *data=nullptr) const
- void **ImportBitmap** (int pitch, UINT8 *buff, BYTE bpp, int channelMap[]=nullptr, CallbackPtr cb=nullptr, void *data=nullptr)
- void **ImportYUV** (int pitch, **DataT** *buff, BYTE bpp, int channelMap[]=nullptr, CallbackPtr cb=nullptr, void *data=nullptr)
- void **Write** (**CPGFStream** *stream, UINT32 *nWrittenBytes=nullptr, CallbackPtr cb=nullptr, void *data=nullptr)
- UINT32 **WriteHeader** (**CPGFStream** *stream)
- UINT32 **WriteImage** (**CPGFStream** *stream, CallbackPtr cb=nullptr, void *data=nullptr)
- UINT32 **Write** (int level, CallbackPtr cb=nullptr, void *data=nullptr)
- void **ConfigureEncoder** (bool useOMP=true, bool favorSpeedOverSize=false)
- void **ConfigureDecoder** (bool useOMP=true, **UserdataPolicy** policy=**UP_CacheAll**, UINT32 prefixSize=0)
- void **ResetStreamPos** (bool startOfData)
- void **SetChannel** (**DataT** *channel, int c=0)
- void **SetHeader** (const **PGFHeader** &header, BYTE flags=0, const UINT8 *userData=0, UINT32 userDataLength=0)
- void **SetMaxValue** (UINT32 maxValue)
- void **SetProgressMode** (**ProgressMode** pm)
- void **SetRefreshCallback** (**RefreshCB** callback, void *arg)
- void **SetColorTable** (UINT32 iFirstColor, UINT32 nColors, const RGBQUAD *prgbColors)
- **DataT** * **GetChannel** (int c=0)
- void **GetColorTable** (UINT32 iFirstColor, UINT32 nColors, RGBQUAD *prgbColors) const
- const RGBQUAD * **GetColorTable** () const
- const **PGFHeader** * **GetHeader** () const
- UINT32 **GetMaxValue** () const
- UINT64 **GetUserDataPos** () const
- const UINT8 * **GetUserData** (UINT32 &cachedSize, UINT32 *pTotalSize=nullptr) const
- UINT32 **GetEncodedHeaderLength** () const

- UINT32 **GetEncodedLevelLength** (int level) const
- UINT32 **ReadEncodedHeader** (UINT8 *target, UINT32 targetLen) const
- UINT32 **ReadEncodedData** (int level, UINT8 *target, UINT32 targetLen) const
- UINT32 **ChannelWidth** (int c=0) const
- UINT32 **ChannelHeight** (int c=0) const
- BYTE **ChannelDepth** () const
- UINT32 **Width** (int level=0) const
- UINT32 **Height** (int level=0) const
- BYTE **Level** () const
- BYTE **Levels** () const
- bool **IsFullyRead** () const
  Return true if all levels have been read.

- BYTE **Quality** () const
- BYTE **Channels** () const
- BYTE **Mode** () const
- BYTE **BPP** () const
- bool **ROIisSupported** () const
- **PGFRect ComputeLevelROI** () const
- BYTE **UsedBitsPerChannel** () const
- BYTE **Version** () const

Static Public Member Functions

- static bool **ImportIsSupported** (BYTE mode)
- static UINT32 **LevelSizeL** (UINT32 size, int level)
- static UINT32 **LevelSizeH** (UINT32 size, int level)
- static BYTE **CodecMajorVersion** (BYTE version=**PGFVersion**)
  Return major version.

- static BYTE **MaxChannelDepth** (BYTE version=**PGFVersion**)

Protected Attributes

- **CWaveletTransform** * **m_wtChannel** [**MaxChannels**]
  wavelet transformed color channels

- **DataT** * **m_channel** [**MaxChannels**]
  untransformed channels in YUV format

- **CDecoder** * **m_decoder**
  PGF decoder.

- **CEncoder** * **m_encoder**
  PGF encoder.

- UINT32 * **m_levelLength**
  length of each level in bytes; first level starts immediately after this array

- UINT32 **m_width** [**MaxChannels**]
  width of each channel at current level

- UINT32 **m_height** [**MaxChannels**]

height of each channel at current level

- **PGFPreHeader m_preHeader**
  PGF pre-header.

- **PGFHeader m_header**
  PGF file header.

- **PGFPostHeader m_postHeader**
  PGF post-header.

- UINT64 **m_userDataPos**
  stream position of user data

- int **m_currentLevel**
  transform level of current image

- UINT32 **m_userDataPolicy**
  user data (metadata) policy during open

- BYTE **m_quant**
  quantization parameter

- bool **m_downsample**
  chrominance channels are downsampled

- bool **m_favorSpeedOverSize**
  favor encoding speed over compression ratio

- bool **m_useOMPinEncoder**
  use Open MP in encoder

- bool **m_useOMPinDecoder**
  use Open MP in decoder

- bool **m_streamReinitialized**
  stream has been reinitialized

- **PGFRect m_roi**
  region of interest

Private Member Functions

- void **Init** ()
- void **ComputeLevels** ()
- bool **CompleteHeader** ()
- void **RgbToYuv** (int pitch, UINT8 *rgbBuff, BYTE bpp, int channelMap[], CallbackPtr cb, void *data)

- void **Downsample** (int nChannel)
- UINT32 **UpdatePostHeaderSize** ()
- void **WriteLevel** ()
- **PGFRect GetAlignedROI** (int c=0) const
- void **SetROI** (**PGFRect** rect)
- UINT8 **Clamp4** (**DataT** v) const
- UINT16 **Clamp6** (**DataT** v) const
- UINT8 **Clamp8** (**DataT** v) const
- UINT16 **Clamp16** (**DataT** v) const
- UINT32 **Clamp31** (**DataT** v) const

Private Attributes

- **RefreshCB m_cb**
  pointer to refresh callback procedure

- void * **m_cbArg**
  refresh callback argument

- double **m_percent**
  progress [0..1]

- **ProgressMode m_progressMode**
  progress mode used in Read and Write; PM_Relative is default mode

---

Detailed Description

PGF main class.

PGF image class is the main class. You always need a PGF object for encoding or decoding image data. Decoding: **Open() Read() GetBitmap()** Encoding: **SetHeader() ImportBitmap() Write()**

Author

C. Stamm, R. Spuler

Definition at line **53** of file **PGFimage.h**.

Constructor & Destructor Documentation

CPGFImage::CPGFImage ()

Standard constructor.

Definition at line **64** of file **PGFimage.cpp**.

```
00064                         {
00065         Init();
00066 }
```

CPGFImage::~CPGFImage () [virtual]

Destructor.

Definition at line **117** of file **PGFimage.cpp**.

```
00117                       {
00118          m_currentLevel = -100; // unusual value used as marker in Destroy()
00119          Destroy();
00120 }
```

Member Function Documentation

BYTE CPGFImage::BPP () const [inline]

Return the number of bits per pixel. Valid values can be 1, 8, 12, 16, 24, 32, 48, 64.

Returns

Number of bits per pixel.

Definition at line **461** of file **PGFimage.h**.

```
00461 { return m_header.bpp; }
```

BYTE CPGFImage::ChannelDepth () const [inline]

Return bits per channel of the image's encoder.

Returns

Bits per channel

Definition at line **406** of file **PGFimage.h**.

```
00406 { return MaxChannelDepth(m_preHeader.version); }
```

UINT32 CPGFImage::ChannelHeight (int *c* = 0) const [inline]

Return current image height of given channel in pixels. The returned height depends on the levels read so far and on ROI.

Parameters

c	A channel index

Returns

Channel height in pixels

Definition at line **401** of file **PGFimage.h**.

```
00401 { ASSERT(c >= 0 && c < MaxChannels); return m_height[c]; }
```

BYTE CPGFImage::Channels () const [inline]

Return the number of image channels. An image of type RGB contains 3 image channels (B, G, R).

Returns

Number of image channels

Definition at line **448** of file **PGFimage.h**.

```
00448 { return m_header.channels; }
```

UINT32 CPGFImage::ChannelWidth (int *c* = 0) const [inline]

Return current image width of given channel in pixels. The returned width depends on the levels read so far and on ROI.

Parameters

c	A channel index

Returns

Channel width in pixels

Definition at line **394** of file **PGFimage.h**.

```
00394 { ASSERT(c >= 0 && c < MaxChannels); return m_width[c]; }
```

UINT16 CPGFImage::Clamp16 (DataT *v*) const [inline], [private]

Definition at line **573** of file **PGFimage.h**.

```
00573                                              {
00574                   if (v & 0xFFFF0000) return (v < 0) ? (UINT16)0: (UINT16)65535;
else return (UINT16)v;
00575           }
```

UINT32 CPGFImage::Clamp31 (DataT *v*) const [inline], [private]

Definition at line **576** of file **PGFimage.h**.

```
00576                                              {
00577                   return (v < 0) ? 0 : (UINT32)v;
00578           }
```

UINT8 CPGFImage::Clamp4 (DataT *v*) const [inline], [private]

Definition at line **563** of file **PGFimage.h**.

```
00563                                            {
00564                   if (v & 0xFFFFFFF0) return (v < 0) ? (UINT8)0: (UINT8)15; else
return (UINT8)v;
00565           }
```

UINT16 CPGFImage::Clamp6 (DataT *v*) const [inline], [private]

Definition at line **566** of file **PGFimage.h**.

```
00566                                             {
00567                   if (v & 0xFFFFFFC0) return (v < 0) ? (UINT16)0: (UINT16)63; else
return (UINT16)v;
00568           }
```

UINT8 CPGFImage::Clamp8 (DataT *v*) const [inline], [private]

Definition at line **569** of file **PGFimage.h**.

```
00569                                            {
00570                  // needs only one test in the normal case
00571                  if (v & 0xFFFFFF00) return (v < 0) ? (UINT8)0 : (UINT8)255; else
return (UINT8)v;
00572           }
```

BYTE CPGFImage::CodecMajorVersion (BYTE *version* = PGFVersion) [static]

Return major version.

Return codec major version.

Parameters

version	pgf pre-header version number

Returns

PGF major of given version

Definition at line **768** of file **PGFimage.cpp**.

```
00768                                                     {
00769         if (version & Version7) return 7;
00770         if (version & Version6) return 6;
00771         if (version & Version5) return 5;
00772         if (version & Version2) return 2;
00773         return 1;
```

```
00774 }
```

bool CPGFImage::CompleteHeader () [private]

Definition at line **218** of file **PGFimage.cpp**.

```
00218                                    {
00219        // set current codec version
00220        m_header.version = PGFVersionNumber(PGFMajorNumber, PGFYear, PGFWeek);
00221
00222        if (m_header.mode == ImageModeUnknown) {
00223                // undefined mode
00224                switch(m_header.bpp) {
00225                case 1: m_header.mode = ImageModeBitmap; break;
00226                case 8: m_header.mode = ImageModeGrayScale; break;
00227                case 12: m_header.mode = ImageModeRGB12; break;
00228                case 16: m_header.mode = ImageModeRGB16; break;
00229                case 24: m_header.mode = ImageModeRGBColor; break;
00230                case 32: m_header.mode = ImageModeRGBA; break;
00231                case 48: m_header.mode = ImageModeRGB48; break;
00232                default: m_header.mode = ImageModeRGBColor; break;
00233                }
00234        }
00235        if (!m_header.bpp) {
00236                // undefined bpp
00237                switch(m_header.mode) {
00238                case ImageModeBitmap:
00239                        m_header.bpp = 1;
00240                        break;
00241                case ImageModeIndexedColor:
00242                case ImageModeGrayScale:
00243                        m_header.bpp = 8;
00244                        break;
00245                case ImageModeRGB12:
00246                        m_header.bpp = 12;
00247                        break;
00248                case ImageModeRGB16:
00249                case ImageModeGray16:
00250                        m_header.bpp = 16;
00251                        break;
00252                case ImageModeRGBColor:
00253                case ImageModeLabColor:
00254                        m_header.bpp = 24;
00255                        break;
00256                case ImageModeRGBA:
00257                case ImageModeCMYKColor:
00258                case ImageModeGray32:
00259                        m_header.bpp = 32;
00260                        break;
00261                case ImageModeRGB48:
00262                case ImageModeLab48:
00263                        m_header.bpp = 48;
00264                        break;
00265                case ImageModeCMYK64:
00266                        m_header.bpp = 64;
00267                        break;
00268                default:
00269                        ASSERT(false);
00270                        m_header.bpp = 24;
00271                }
00272        }
00273        if (m_header.mode == ImageModeRGBColor && m_header.bpp == 32) {
00274                // change mode
00275                m_header.mode = ImageModeRGBA;
00276        }
00277        if (m_header.mode == ImageModeBitmap && m_header.bpp != 1) return false;
00278        if (m_header.mode == ImageModeIndexedColor && m_header.bpp != 8) return
false;
00279        if (m_header.mode == ImageModeGrayScale && m_header.bpp != 8) return
false;
00280        if (m_header.mode == ImageModeGray16 && m_header.bpp != 16) return false;
00281        if (m_header.mode == ImageModeGray32 && m_header.bpp != 32) return false;
00282        if (m_header.mode == ImageModeRGBColor && m_header.bpp != 24) return
false;
00283        if (m_header.mode == ImageModeRGBA && m_header.bpp != 32) return false;
```

```
00284         if (m_header.mode == ImageModeRGB12 && m_header.bpp != 12) return false;
00285         if (m_header.mode == ImageModeRGB16 && m_header.bpp != 16) return false;
00286         if (m_header.mode == ImageModeRGB48 && m_header.bpp != 48) return false;
00287         if (m_header.mode == ImageModeLabColor && m_header.bpp != 24) return
false;
00288         if (m_header.mode == ImageModeLab48 && m_header.bpp != 48) return false;
00289         if (m_header.mode == ImageModeCMYKColor && m_header.bpp != 32) return
false;
00290         if (m_header.mode == ImageModeCMYK64 && m_header.bpp != 64) return false;
00291
00292         // set number of channels
00293         if (!m_header.channels) {
00294                 switch(m_header.mode) {
00295                 case ImageModeBitmap:
00296                 case ImageModeIndexedColor:
00297                 case ImageModeGrayScale:
00298                 case ImageModeGray16:
00299                 case ImageModeGray32:
00300                         m_header.channels = 1;
00301                         break;
00302                 case ImageModeRGBColor:
00303                 case ImageModeRGB12:
00304                 case ImageModeRGB16:
00305                 case ImageModeRGB48:
00306                 case ImageModeLabColor:
00307                 case ImageModeLab48:
00308                         m_header.channels = 3;
00309                         break;
00310                 case ImageModeRGBA:
00311                 case ImageModeCMYKColor:
00312                 case ImageModeCMYK64:
00313                         m_header.channels = 4;
00314                         break;
00315                 default:
00316                         return false;
00317                 }
00318         }
00319
00320         // store used bits per channel
00321         UINT8 bpc = m_header.bpp/m_header.channels;
00322         if (bpc > 31) bpc = 31;
00323         if (!m_header.usedBitsPerChannel || m_header.usedBitsPerChannel > bpc)
{
00324                 m_header.usedBitsPerChannel = bpc;
00325         }
00326
00327         return true;
00328 }
```

PGFRect CPGFImage::ComputeLevelROI () const

Return ROI of channel 0 at current level in pixels. The returned rect is only valid after reading a ROI.

Returns

ROI in pixels

void CPGFImage::ComputeLevels () [private]

Definition at line **854** of file **PGFimage.cpp**.

```
00854                 {
00855         const int maxThumbnailWidth = 20*FilterSize;
00856         const int m = __min(m_header.width, m_header.height);
00857         int s = m;
00858
00859         if (m_header.nLevels < 1 || m_header.nLevels > MaxLevel) {
00860                 m_header.nLevels = 1;
00861                 // compute a good value depending on the size of the image
00862                 while (s > maxThumbnailWidth) {
00863                         m_header.nLevels++;
00864                         s >>= 1;
00865                 }
```

```
00866          }
00867
00868          int levels = m_header.nLevels; // we need a signed value during level
reduction
00869
00870          // reduce number of levels if the image size is smaller than
FilterSize*(2^levels)
00871          s = FilterSize*(1 << levels);   // must be at least the double filter
size because of subsampling
00872          while (m < s) {
00873                  levels--;
00874                  s >>= 1;
00875          }
00876          if (levels > MaxLevel) m_header.nLevels = MaxLevel;
00877          else if (levels < 0) m_header.nLevels = 0;
00878          else m_header.nLevels = (UINT8)levels;
00879
00880          // used in Write when PM_Absolute
00881          m_percent = pow(0.25, m_header.nLevels);
00882
00883          ASSERT(0 <= m_header.nLevels && m_header.nLevels <= MaxLevel);
00884 }
```

void CPGFImage::ConfigureDecoder (bool *useOMP* = `true`, UserdataPolicy *policy* = `UP_CacheAll`, UINT32 *prefixSize* = 0)`[inline]`

Configures the decoder.

Parameters

useOMP	Use parallel threading with Open MP during decoding. Default value: true. Influences the decoding only if the codec has been compiled with OpenMP support.
policy	The file might contain user data (e.g. metadata). The policy defines the behaviour during **Open()**. UP_CacheAll: User data is read and stored completely in a new allocated memory block. It can be accessed by **GetUserData()**. UP_CachePrefix: Only prefixSize bytes at the beginning of the user data are stored in a new allocated memory block. It can be accessed by **GetUserData()**. UP_Skip: User data is skipped and nothing is cached.
prefixSize	Is only used in combination with UP_CachePrefix. It defines the number of bytes cached.

Definition at line **260** of file **PGFimage.h**.

```
00260 { ASSERT(prefixSize <= MaxUserDataSize);  m_useOMPinDecoder = useOMP;
m_userDataPolicy = (UP_CachePrefix) ? prefixSize : 0xFFFFFFFF - policy; }
```

void CPGFImage::ConfigureEncoder (bool *useOMP* = `true`, bool *favorSpeedOverSize* = `false`)`[inline]`

Configures the encoder.

Parameters

useOMP	Use parallel threading with Open MP during encoding. Default value: true. Influences the encoding only if the codec has been compiled with OpenMP support.
favorSpeedOverSize	Favors encoding speed over compression ratio. Default value: false

Definition at line **250** of file **PGFimage.h**.

```
00250 { m_useOMPinEncoder = useOMP; m_favorSpeedOverSize = favorSpeedOverSize; }
```

void CPGFImage::Destroy ()

Definition at line **124** of file **PGFimage.cpp**.

```
00124                          {
00125          for (int i = 0; i < m_header.channels; i++) {
00126                  delete m_wtChannel[i]; // also deletes m_channel
00127          }
```

```
00128           delete[] m_postHeader.userData;
00129           delete[] m_levelLength;
00130           delete m_decoder;
00131           delete m_encoder;
00132
00133           if (m_currentLevel != -100) Init();
00134 }
```

void CPGFImage::Downsample (int *nChannel*) [private]

Definition at line **810** of file **PGFimage.cpp**.

```
00810                                  {
00811           ASSERT(ch > 0);
00812
00813           const int w = m_width[0];
00814           const int w2 = w/2;
00815           const int h2 = m_height[0]/2;
00816           const int oddW = w%2;                       // don't use bool ->
problems with MaxSpeed optimization
00817           const int oddH = m_height[0]%2;         // "
00818           int loPos = 0;
00819           int hiPos = w;
00820           int sampledPos = 0;
00821           DataT* buff = m_channel[ch]; ASSERT(buff);
00822
00823           for (int i=0; i < h2; i++) {
00824                   for (int j=0; j < w2; j++) {
00825                           // compute average of pixel block
00826                           buff[sampledPos] = (buff[loPos] + buff[loPos + 1] +
buff[hiPos] + buff[hiPos + 1]) >> 2;
00827                           loPos += 2; hiPos += 2;
00828                           sampledPos++;
00829                   }
00830                   if (oddW) {
00831                           buff[sampledPos] = (buff[loPos] + buff[hiPos]) >> 1;
00832                           loPos++; hiPos++;
00833                           sampledPos++;
00834                   }
00835                   loPos += w; hiPos += w;
00836           }
00837           if (oddH) {
00838                   for (int j=0; j < w2; j++) {
00839                           buff[sampledPos] = (buff[loPos] + buff[loPos+1]) >> 1;
00840                           loPos += 2; hiPos += 2;
00841                           sampledPos++;
00842                   }
00843                   if (oddW) {
00844                           buff[sampledPos] = buff[loPos];
00845                   }
00846           }
00847
00848           // downsampled image has half width and half height
00849           m_width[ch] = (m_width[ch] + 1)/2;
00850           m_height[ch] = (m_height[ch] + 1)/2;
00851 }
```

PGFRect CPGFImage::GetAlignedROI (int *c* = 0) const [private]

void CPGFImage::GetBitmap (int *pitch*, UINT8 * *buff*, BYTE *bpp*, int *channelMap*[] = nullptr, CallbackPtr *cb* = nullptr, void * *data* = nullptr) const

Get image data in interleaved format: (ordering of RGB data is BGR[A]) Upsampling, YUV to RGB transform and interleaving are done here to reduce the number of passes over the data. The absolute value of pitch is the number of bytes of an image row of the given image buffer. If pitch is negative, then the image buffer must point to the last row of a bottom-up image (first byte on last row). if pitch is positive, then the image buffer must point to the first row of a top-down image (first byte). The sequence of output channels in the output image buffer does not need to be the same as provided by PGF. In case of different sequences you have to provide a channelMap of size of expected

channels (depending on image mode). For example, PGF provides a channel sequence BGR in RGB color mode. If your provided image buffer expects a channel sequence ARGB, then the channelMap looks like { 3, 2, 1, 0 }. It might throw an **IOException**.

Parameters

pitch	The number of bytes of a row of the image buffer.
buff	An image buffer.
bpp	The number of bits per pixel used in image buffer.
channelMap	A integer array containing the mapping of PGF channel ordering to expected channel ordering.
cb	A pointer to a callback procedure. The procedure is called after each copied buffer row. If cb returns true, then it stops proceeding.
data	Data Pointer to C++ class container to host callback procedure.

Definition at line **1789** of file **PGFimage.cpp**.

```
01789
{
01790           ASSERT(buff);
01791           UINT32 w = m_width[0];  // width of decoded image
01792           UINT32 h = m_height[0]; // height of decoded image
01793           UINT32 yw = w;                  // y-channel width
01794           UINT32 uw = m_width[1]; // u-channel width
01795           UINT32 roiOffsetX = 0;
01796           UINT32 roiOffsetY = 0;
01797           UINT32 yOffset = 0;
01798           UINT32 uOffset = 0;
01799
01800 #ifdef __PGFROISUPPORT__
01801           const PGFRect& roi = GetAlignedROI(); // in pixels, roi is usually larger
than levelRoi
01802           ASSERT(w == roi.Width() && h == roi.Height());
01803           const PGFRect levelRoi = ComputeLevelROI();
01804           ASSERT(roi.left <= levelRoi.left && levelRoi.right <= roi.right);
01805           ASSERT(roi.top <= levelRoi.top && levelRoi.bottom <= roi.bottom);
01806
01807           if (ROIisSupported() && (levelRoi.Width() < w || levelRoi.Height() < h))
{
01808                   // ROI is used
01809                   w = levelRoi.Width();
01810                   h = levelRoi.Height();
01811                   roiOffsetX = levelRoi.left - roi.left;
01812                   roiOffsetY = levelRoi.top - roi.top;
01813                   yOffset = roiOffsetX + roiOffsetY*yw;
01814
01815                   if (m_downsample) {
01816                           const PGFRect& downsampledRoi = GetAlignedROI(1);
01817                           uOffset = levelRoi.left/2 - downsampledRoi.left +
(levelRoi.top/2 - downsampledRoi.top)*m_width[1];
01818                   } else {
01819                           uOffset = yOffset;
01820                   }
01821           }
01822 #endif
01823
01824           const double dP = 1.0/h;
01825           int defMap[] = { 0, 1, 2, 3, 4, 5, 6, 7 };
ASSERT(sizeof(defMap)/sizeof(defMap[0]) == MaxChannels);
01826           if (channelMap == nullptr) channelMap = defMap;
01827           DataT uAvg, vAvg;
01828           double percent = 0;
01829           UINT32 i, j;
01830
01831           switch(m_header.mode) {
01832           case ImageModeBitmap:
01833                   {
01834                           ASSERT(m_header.channels == 1);
01835                           ASSERT(m_header.bpp == 1);
01836                           ASSERT(bpp == 1);
01837
01838                           const UINT32 w2 = (w + 7)/8;
01839                           DataT* y = m_channel[0]; ASSERT(y);
01840
01841                           if (m_preHeader.version & Version7) {
```

```
01842                                   // new unpacked version has a little better
compression ratio
01843                                   // since version 7
01844                                   for (i = 0; i < h; i++) {
01845                                           UINT32 cnt = 0;
01846                                           for (j = 0; j < w2; j++) {
01847                                                   UINT8 byte = 0;
01848                                                   for (int k = 0; k < 8; k++) {
01849                                                           byte <<= 1;
01850                                                           UINT8 bit = 0;
01851                                                           if (cnt < w) {
01852                                                                   bit =
y[yOffset + cnt] & 1;
01853                                                           }
01854                                                           byte |= bit;
01855                                                           cnt++;
01856                                                   }
01857                                                   buff[j] = byte;
01858                                           }
01859                                           yOffset += yw;
01860                                           buff += pitch;
01861
01862                                           if (cb) {
01863                                                   percent += dP;
01864                                                   if ((*cb)(percent, true,
data)) ReturnWithError(EscapePressed);
01865                                           }
01866                                   }
01867                           } else {
01868                                   // old versions
01869                                   // packed pixels: 8 pixel in 1 byte of channel[0]
01870                                   if (!(m_preHeader.version & Version5)) yw = w2;
// not version 5 or 6
01871                                   yOffset = roiOffsetX/8 + roiOffsetY*yw; // 1
byte in y contains 8 pixel values
01872                                   for (i = 0; i < h; i++) {
01873                                           for (j = 0; j < w2; j++) {
01874                                                   buff[j] = Clamp8(y[yOffset +
j] + YUVoffset8);
01875                                           }
01876                                           yOffset += yw;
01877                                           buff += pitch;
01878
01879                                           if (cb) {
01880                                                   percent += dP;
01881                                                   if ((*cb)(percent, true,
data)) ReturnWithError(EscapePressed);
01882                                           }
01883                                   }
01884                           }
01885                           break;
01886                   }
01887           case ImageModeIndexedColor:
01888           case ImageModeGrayScale:
01889           case ImageModeHSLColor:
01890           case ImageModeHSBColor:
01891                   {
01892                           ASSERT(m_header.channels >= 1);
01893                           ASSERT(m_header.bpp == m_header.channels*8);
01894                           ASSERT(bpp%8 == 0);
01895
01896                           UINT32 cnt, channels = bpp/8; ASSERT(channels >=
m_header.channels);
01897
01898                           for (i=0; i < h; i++) {
01899                                   UINT32 yPos = yOffset;
01900                                   cnt = 0;
01901                                   for (j=0; j < w; j++) {
01902                                           for (UINT32 c=0; c < m_header.channels;
c++) {
01903                                                   buff[cnt + channelMap[c]] =
Clamp8(m_channel[c][yPos] + YUVoffset8);
01904                                           }
01905                                           cnt += channels;
01906                                           yPos++;
01907                                   }
01908                                   yOffset += yw;
```

```
01909                           buff += pitch;
01910
01911                           if (cb) {
01912                                   percent += dP;
01913                                   if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
01914                           }
01915                   }
01916                   break;
01917           }
01918   case ImageModeGray16:
01919           {
01920                   ASSERT(m_header.channels >= 1);
01921                   ASSERT(m_header.bpp == m_header.channels*16);
01922
01923                   const DataT yuvOffset16 = 1 << (UsedBitsPerChannel() -
1);
01924                   UINT32 cnt, channels;
01925
01926                   if (bpp%16 == 0) {
01927                           const int shift = 16 - UsedBitsPerChannel();
ASSERT(shift >= 0);
01928                           UINT16 *buff16 = (UINT16 *)buff;
01929                           int pitch16 = pitch/2;
01930                           channels = bpp/16; ASSERT(channels >=
m_header.channels);
01931
01932                           for (i=0; i < h; i++) {
01933                                   UINT32 yPos = yOffset;
01934                                   cnt = 0;
01935                                   for (j=0; j < w; j++) {
01936                                           for (UINT32 c=0; c <
m_header.channels; c++) {
01937                                                   buff16[cnt +
channelMap[c]] = Clamp16((m_channel[c][yPos] + yuvOffset16) << shift);
01938                                           }
01939                                           cnt += channels;
01940                                           yPos++;
01941                                   }
01942                                   yOffset += yw;
01943                                   buff16 += pitch16;
01944
01945                                   if (cb) {
01946                                           percent += dP;
01947                                           if ((*cb)(percent, true,
data)) ReturnWithError(EscapePressed);
01948                                   }
01949                           }
01950                   } else {
01951                           ASSERT(bpp%8 == 0);
01952                           const int shift = __max(0, UsedBitsPerChannel()
- 8);
01953                           channels = bpp/8; ASSERT(channels >=
m_header.channels);
01954
01955                           for (i=0; i < h; i++) {
01956                                   UINT32 yPos = yOffset;
01957                                   cnt = 0;
01958                                   for (j=0; j < w; j++) {
01959                                           for (UINT32 c=0; c <
m_header.channels; c++) {
01960                                                   buff[cnt +
channelMap[c]] = Clamp8((m_channel[c][yPos] + yuvOffset16) >> shift);
01961                                           }
01962                                           cnt += channels;
01963                                           yPos++;
01964                                   }
01965                                   yOffset += yw;
01966                                   buff += pitch;
01967
01968                                   if (cb) {
01969                                           percent += dP;
01970                                           if ((*cb)(percent, true,
data)) ReturnWithError(EscapePressed);
01971                                   }
01972                           }
01973                   }
```

```
01974                         break;
01975                 }
01976         case ImageModeRGBColor:
01977                 {
01978                         ASSERT(m_header.channels == 3);
01979                         ASSERT(m_header.bpp == m_header.channels*8);
01980                         ASSERT(bpp%8 == 0);
01981                         ASSERT(bpp >= m_header.bpp);
01982
01983                         DataT* y = m_channel[0]; ASSERT(y);
01984                         DataT* u = m_channel[1]; ASSERT(u);
01985                         DataT* v = m_channel[2]; ASSERT(v);
01986                         UINT8 *buffg = &buff[channelMap[1]],
01987                                   *buffr = &buff[channelMap[2]],
01988                                   *buffb = &buff[channelMap[0]];
01989                         UINT8 g;
01990                         UINT32 cnt, channels = bpp/8;
01991
01992                         if (m_downsample) {
01993                                 for (i=0; i < h; i++) {
01994                                         UINT32 uPos = uOffset;
01995                                         UINT32 yPos = yOffset;
01996                                         cnt = 0;
01997                                         for (j=0; j < w; j++) {
01998                                                 // u and v are downsampled
01999                                                 uAvg = u[uPos];
02000                                                 vAvg = v[uPos];
02001                                                 // Yuv
02002                                                 buffg[cnt] = g =
Clamp8(y[yPos] + YUVoffset8 - ((uAvg + vAvg ) >> 2)); // must be logical shift operator
02003                                                 buffr[cnt] = Clamp8(uAvg + g);
02004                                                 buffb[cnt] = Clamp8(vAvg + g);
02005                                                 cnt += channels;
02006                                                 if (j & 1) uPos++;
02007                                                 yPos++;
02008                                         }
02009                                         if (i & 1) uOffset += uw;
02010                                         yOffset += yw;
02011                                         buffb += pitch;
02012                                         buffg += pitch;
02013                                         buffr += pitch;
02014
02015                                         if (cb) {
02016                                                 percent += dP;
02017                                                 if ((*cb)(percent, true,
data)) ReturnWithError(EscapePressed);
02018                                         }
02019                                 }
02020
02021                         } else {
02022                                 for (i=0; i < h; i++) {
02023                                         cnt = 0;
02024                                         UINT32 yPos = yOffset;
02025                                         for (j = 0; j < w; j++) {
02026                                                 uAvg = u[yPos];
02027                                                 vAvg = v[yPos];
02028                                                 // Yuv
02029                                                 buffg[cnt] = g =
Clamp8(y[yPos] + YUVoffset8 - ((uAvg + vAvg ) >> 2)); // must be logical shift operator
02030                                                 buffr[cnt] = Clamp8(uAvg + g);
02031                                                 buffb[cnt] = Clamp8(vAvg + g);
02032                                                 cnt += channels;
02033                                                 yPos++;
02034                                         }
02035                                         yOffset += yw;
02036                                         buffb += pitch;
02037                                         buffg += pitch;
02038                                         buffr += pitch;
02039
02040                                         if (cb) {
02041                                                 percent += dP;
02042                                                 if ((*cb)(percent, true,
data)) ReturnWithError(EscapePressed);
02043                                         }
02044                                 }
02045                         }
02046                         break;
```

```
02047                   }
02048           case ImageModeRGB48:
02049                   {
02050                           ASSERT(m_header.channels == 3);
02051                           ASSERT(m_header.bpp == 48);
02052
02053                           const DataT yuvOffset16 = 1 << (UsedBitsPerChannel() -
1);
02054
02055                           DataT* y = m_channel[0]; ASSERT(y);
02056                           DataT* u = m_channel[1]; ASSERT(u);
02057                           DataT* v = m_channel[2]; ASSERT(v);
02058                           UINT32 cnt, channels;
02059                           DataT g;
02060
02061                           if (bpp >= 48 && bpp%16 == 0) {
02062                                   const int shift = 16 - UsedBitsPerChannel();
ASSERT(shift >= 0);
02063                                   UINT16 *buff16 = (UINT16 *)buff;
02064                                   int pitch16 = pitch/2;
02065                                   channels = bpp/16; ASSERT(channels >=
m_header.channels);
02066
02067                                   for (i=0; i < h; i++) {
02068                                           UINT32 uPos = uOffset;
02069                                           UINT32 yPos = yOffset;
02070                                           cnt = 0;
02071                                           for (j=0; j < w; j++) {
02072                                                   uAvg = u[uPos];
02073                                                   vAvg = v[uPos];
02074                                                   // Yuv
02075                                                   g = y[yPos] + yuvOffset16 -
((uAvg + vAvg ) >> 2); // must be logical shift operator
02076                                                   buff16[cnt + channelMap[1]] =
Clamp16(g << shift);
02077                                                   buff16[cnt + channelMap[2]] =
Clamp16((uAvg + g) << shift);
02078                                                   buff16[cnt + channelMap[0]] =
Clamp16((vAvg + g) << shift);
02079                                                   cnt += channels;
02080                                                   if (!m_downsample || (j & 1))
uPos++;
02081                                                   yPos++;
02082                                           }
02083                                           if (!m_downsample || (i & 1)) uOffset
+= uw;
02084                                           yOffset += yw;
02085                                           buff16 += pitch16;
02086
02087                                           if (cb) {
02088                                                   percent += dP;
02089                                                   if ((*cb)(percent, true,
data)) ReturnWithError(EscapePressed);
02090                                           }
02091                                   }
02092                           } else {
02093                                   ASSERT(bpp%8 == 0);
02094                                   const int shift = __max(0, UsedBitsPerChannel()
- 8);
02095                                   channels = bpp/8; ASSERT(channels >=
m_header.channels);
02096
02097                                   for (i=0; i < h; i++) {
02098                                           UINT32 uPos = uOffset;
02099                                           UINT32 yPos = yOffset;
02100                                           cnt = 0;
02101                                           for (j=0; j < w; j++) {
02102                                                   uAvg = u[uPos];
02103                                                   vAvg = v[uPos];
02104                                                   // Yuv
02105                                                   g = y[yPos] + yuvOffset16 -
((uAvg + vAvg ) >> 2); // must be logical shift operator
02106                                                   buff[cnt + channelMap[1]] =
Clamp8(g >> shift);
02107                                                   buff[cnt + channelMap[2]] =
Clamp8((uAvg + g) >> shift);
```

```
02108                                                   buff[cnt + channelMap[0]] =
Clamp8((vAvg + g) >> shift);
02109                                                   cnt += channels;
02110                                                   if (!m_downsample || (j & 1))
uPos++;
02111                                                   yPos++;
02112                                           }
02113                                           if (!m_downsample || (i & 1)) uOffset
+= uw;
02114                                           yOffset += yw;
02115                                           buff += pitch;
02116
02117                                           if (cb) {
02118                                                   percent += dP;
02119                                                   if ((*cb)(percent, true,
data)) ReturnWithError(EscapePressed);
02120                                           }
02121                                   }
02122                           }
02123                           break;
02124                   }
02125           case ImageModeLabColor:
02126                   {
02127                           ASSERT(m_header.channels == 3);
02128                           ASSERT(m_header.bpp == m_header.channels*8);
02129                           ASSERT(bpp%8 == 0);
02130
02131                           DataT* l = m_channel[0]; ASSERT(l);
02132                           DataT* a = m_channel[1]; ASSERT(a);
02133                           DataT* b = m_channel[2]; ASSERT(b);
02134                           UINT32 cnt, channels = bpp/8; ASSERT(channels >=
m_header.channels);
02135
02136                           for (i=0; i < h; i++) {
02137                                   UINT32 uPos = uOffset;
02138                                   UINT32 yPos = yOffset;
02139                                   cnt = 0;
02140                                   for (j=0; j < w; j++) {
02141                                           uAvg = a[uPos];
02142                                           vAvg = b[uPos];
02143                                           buff[cnt + channelMap[0]] =
Clamp8(l[yPos] + YUVoffset8);
02144                                           buff[cnt + channelMap[1]] =
Clamp8(uAvg + YUVoffset8);
02145                                           buff[cnt + channelMap[2]] =
Clamp8(vAvg + YUVoffset8);
02146                                           cnt += channels;
02147                                           if (!m_downsample || (j & 1)) uPos++;
02148                                           yPos++;
02149                                   }
02150                                   if (!m_downsample || (i & 1)) uOffset += uw;
02151                                   yOffset += yw;
02152                                   buff += pitch;
02153
02154                                   if (cb) {
02155                                           percent += dP;
02156                                           if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
02157                                   }
02158                           }
02159                           break;
02160                   }
02161           case ImageModeLab48:
02162                   {
02163                           ASSERT(m_header.channels == 3);
02164                           ASSERT(m_header.bpp == m_header.channels*16);
02165
02166                           const DataT yuvOffset16 = 1 << (UsedBitsPerChannel() -
1);
02167
02168                           DataT* l = m_channel[0]; ASSERT(l);
02169                           DataT* a = m_channel[1]; ASSERT(a);
02170                           DataT* b = m_channel[2]; ASSERT(b);
02171                           UINT32 cnt, channels;
02172
02173                           if (bpp%16 == 0) {
```

```
02174                             const int shift = 16 - UsedBitsPerChannel();
ASSERT(shift >= 0);
02175                             UINT16 *buff16 = (UINT16 *)buff;
02176                             int pitch16 = pitch/2;
02177                             channels = bpp/16; ASSERT(channels >=
m_header.channels);
02178
02179                             for (i=0; i < h; i++) {
02180                                     UINT32 uPos = uOffset;
02181                                     UINT32 yPos = yOffset;
02182                                     cnt = 0;
02183                                     for (j=0; j < w; j++) {
02184                                             uAvg = a[uPos];
02185                                             vAvg = b[uPos];
02186                                             buff16[cnt + channelMap[0]] =
Clamp16((l[yPos] + yuvOffset16) << shift);
02187                                             buff16[cnt + channelMap[1]] =
Clamp16((uAvg + yuvOffset16) << shift);
02188                                             buff16[cnt + channelMap[2]] =
Clamp16((vAvg + yuvOffset16) << shift);
02189                                             cnt += channels;
02190                                             if (!m_downsample || (j & 1))
uPos++;
02191                                             yPos++;
02192                                     }
02193                                     if (!m_downsample || (i & 1)) uOffset
+= uw;
02194                                     yOffset += yw;
02195                                     buff16 += pitch16;
02196
02197                                     if (cb) {
02198                                             percent += dP;
02199                                             if ((*cb)(percent, true,
data)) ReturnWithError(EscapePressed);
02200                                     }
02201                             }
02202                     } else {
02203                             ASSERT(bpp%8 == 0);
02204                             const int shift = __max(0, UsedBitsPerChannel()
- 8);
02205                             channels = bpp/8; ASSERT(channels >=
m_header.channels);
02206
02207                             for (i=0; i < h; i++) {
02208                                     UINT32 uPos = uOffset;
02209                                     UINT32 yPos = yOffset;
02210                                     cnt = 0;
02211                                     for (j=0; j < w; j++) {
02212                                             uAvg = a[uPos];
02213                                             vAvg = b[uPos];
02214                                             buff[cnt + channelMap[0]] =
Clamp8((l[yPos] + yuvOffset16) >> shift);
02215                                             buff[cnt + channelMap[1]] =
Clamp8((uAvg + yuvOffset16) >> shift);
02216                                             buff[cnt + channelMap[2]] =
Clamp8((vAvg + yuvOffset16) >> shift);
02217                                             cnt += channels;
02218                                             if (!m_downsample || (j & 1))
uPos++;
02219                                             yPos++;
02220                                     }
02221                                     if (!m_downsample || (i & 1)) uOffset
+= uw;
02222                                     yOffset += yw;
02223                                     buff += pitch;
02224
02225                                     if (cb) {
02226                                             percent += dP;
02227                                             if ((*cb)(percent, true,
data)) ReturnWithError(EscapePressed);
02228                                     }
02229                             }
02230                     }
02231                     break;
02232             }
02233     case ImageModeRGBA:
02234     case ImageModeCMYKColor:
```

```
02235                   {
02236                           ASSERT(m_header.channels == 4);
02237                           ASSERT(m_header.bpp == m_header.channels*8);
02238                           ASSERT(bpp%8 == 0);
02239
02240                           DataT* y = m_channel[0]; ASSERT(y);
02241                           DataT* u = m_channel[1]; ASSERT(u);
02242                           DataT* v = m_channel[2]; ASSERT(v);
02243                           DataT* a = m_channel[3]; ASSERT(a);
02244                           UINT8 g, aAvg;
02245                           UINT32 cnt, channels = bpp/8; ASSERT(channels >=
m_header.channels);
02246
02247                           for (i=0; i < h; i++) {
02248                                   UINT32 uPos = uOffset;
02249                                   UINT32 yPos = yOffset;
02250                                   cnt = 0;
02251                                   for (j=0; j < w; j++) {
02252                                           uAvg = u[uPos];
02253                                           vAvg = v[uPos];
02254                                           aAvg = Clamp8(a[uPos] + YUVoffset8);
02255                                           // Yuv
02256                                           buff[cnt + channelMap[1]] = g =
Clamp8(y[yPos] + YUVoffset8 - ((uAvg + vAvg ) >> 2)); // must be logical shift operator
02257                                           buff[cnt + channelMap[2]] =
Clamp8(uAvg + g);
02258                                           buff[cnt + channelMap[0]] =
Clamp8(vAvg + g);
02259                                           buff[cnt + channelMap[3]] = aAvg;
02260                                           cnt += channels;
02261                                           if (!m_downsample || (j & 1)) uPos++;
02262                                           yPos++;
02263                                   }
02264                                   if (!m_downsample || (i & 1)) uOffset += uw;
02265                                   yOffset += yw;
02266                                   buff += pitch;
02267
02268                                   if (cb) {
02269                                           percent += dP;
02270                                           if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
02271                                   }
02272                           }
02273                           break;
02274                   }
02275           case ImageModeCMYK64:
02276                   {
02277                           ASSERT(m_header.channels == 4);
02278                           ASSERT(m_header.bpp == 64);
02279
02280                           const DataT yuvOffset16 = 1 << (UsedBitsPerChannel() -
1);
02281
02282                           DataT* y = m_channel[0]; ASSERT(y);
02283                           DataT* u = m_channel[1]; ASSERT(u);
02284                           DataT* v = m_channel[2]; ASSERT(v);
02285                           DataT* a = m_channel[3]; ASSERT(a);
02286                           DataT g, aAvg;
02287                           UINT32 cnt, channels;
02288
02289                           if (bpp%16 == 0) {
02290                                   const int shift = 16 - UsedBitsPerChannel();
ASSERT(shift >= 0);
02291                                   UINT16 *buff16 = (UINT16 *)buff;
02292                                   int pitch16 = pitch/2;
02293                                   channels = bpp/16; ASSERT(channels >=
m_header.channels);
02294
02295                                   for (i=0; i < h; i++) {
02296                                           UINT32 uPos = uOffset;
02297                                           UINT32 yPos = yOffset;
02298                                           cnt = 0;
02299                                           for (j=0; j < w; j++) {
02300                                                   uAvg = u[uPos];
02301                                                   vAvg = v[uPos];
02302                                                   aAvg = a[uPos] + yuvOffset16;
02303                                                   // Yuv
```

```
02304                                                   g = y[yPos] + yuvOffset16 -
((uAvg + vAvg ) >> 2); // must be logical shift operator
02305                                                   buff16[cnt + channelMap[1]] =
Clamp16(g << shift);
02306                                                   buff16[cnt + channelMap[2]] =
Clamp16((uAvg + g) << shift);
02307                                                   buff16[cnt + channelMap[0]] =
Clamp16((vAvg + g) << shift);
02308                                                   buff16[cnt + channelMap[3]] =
Clamp16(aAvg << shift);
02309                                                   cnt += channels;
02310                                                   if (!m_downsample || (j & 1))
uPos++;
02311                                                   yPos++;
02312                                           }
02313                                           if (!m_downsample || (i & 1)) uOffset
+= uw;
02314                                           yOffset += yw;
02315                                           buff16 += pitch16;
02316
02317                                           if (cb) {
02318                                                   percent += dP;
02319                                                   if ((*cb)(percent, true,
data)) ReturnWithError(EscapePressed);
02320                                           }
02321                                   }
02322                           } else {
02323                                   ASSERT(bpp%8 == 0);
02324                                   const int shift = __max(0, UsedBitsPerChannel()
- 8);
02325                                   channels = bpp/8; ASSERT(channels >=
m_header.channels);
02326
02327                                   for (i=0; i < h; i++) {
02328                                           UINT32 uPos = uOffset;
02329                                           UINT32 yPos = yOffset;
02330                                           cnt = 0;
02331                                           for (j=0; j < w; j++) {
02332                                                   uAvg = u[uPos];
02333                                                   vAvg = v[uPos];
02334                                                   aAvg = a[uPos] + yuvOffset16;
02335                                                   // Yuv
02336                                                   g = y[yPos] + yuvOffset16 -
((uAvg + vAvg ) >> 2); // must be logical shift operator
02337                                                   buff[cnt + channelMap[1]] =
Clamp8(g >> shift);
02338                                                   buff[cnt + channelMap[2]] =
Clamp8((uAvg + g) >> shift);
02339                                                   buff[cnt + channelMap[0]] =
Clamp8((vAvg + g) >> shift);
02340                                                   buff[cnt + channelMap[3]] =
Clamp8(aAvg >> shift);
02341                                                   cnt += channels;
02342                                                   if (!m_downsample || (j & 1))
uPos++;
02343                                                   yPos++;
02344                                           }
02345                                           if (!m_downsample || (i & 1)) uOffset
+= uw;
02346                                           yOffset += yw;
02347                                           buff += pitch;
02348
02349                                           if (cb) {
02350                                                   percent += dP;
02351                                                   if ((*cb)(percent, true,
data)) ReturnWithError(EscapePressed);
02352                                           }
02353                                   }
02354                           }
02355                           break;
02356                   }
02357 #ifdef __PGF32SUPPORT__
02358           case ImageModeGray32:
02359                   {
02360                           ASSERT(m_header.channels == 1);
02361                           ASSERT(m_header.bpp == 32);
02362
```

```
02363                   const int yuvOffset31 = 1 << (UsedBitsPerChannel() - 1);
02364                   DataT* y = m_channel[0]; ASSERT(y);
02365
02366                   if (bpp == 32) {
02367                           const int shift = 31 - UsedBitsPerChannel();
ASSERT(shift >= 0);
02368                           UINT32 *buff32 = (UINT32 *)buff;
02369                           int pitch32 = pitch/4;
02370
02371                           for (i=0; i < h; i++) {
02372                                   UINT32 yPos = yOffset;
02373                                   for (j = 0; j < w; j++) {
02374                                           buff32[j] =
Clamp31((y[yPos++] + yuvOffset31) << shift);
02375                                   }
02376                                   yOffset += yw;
02377                                   buff32 += pitch32;
02378
02379                                   if (cb) {
02380                                           percent += dP;
02381                                           if ((*cb)(percent, true,
data)) ReturnWithError(EscapePressed);
02382                                   }
02383                           }
02384                   } else if (bpp == 16) {
02385                           const int usedBits = UsedBitsPerChannel();
02386                           UINT16 *buff16 = (UINT16 *)buff;
02387                           int pitch16 = pitch/2;
02388
02389                           if (usedBits < 16) {
02390                                   const int shift = 16 - usedBits;
02391                                   for (i=0; i < h; i++) {
02392                                           UINT32 yPos = yOffset;
02393                                           for (j = 0; j < w; j++) {
02394                                                   buff16[j] =
Clamp16((y[yPos++] + yuvOffset31) << shift);
02395                                           }
02396                                           yOffset += yw;
02397                                           buff16 += pitch16;
02398
02399                                           if (cb) {
02400                                                   percent += dP;
02401                                                   if ((*cb)(percent,
true, data)) ReturnWithError(EscapePressed);
02402                                           }
02403                                   }
02404                           } else {
02405                                   const int shift = __max(0, usedBits -
16);
02406                                   for (i=0; i < h; i++) {
02407                                           UINT32 yPos = yOffset;
02408                                           for (j = 0; j < w; j++) {
02409                                                   buff16[j] =
Clamp16((y[yPos++] + yuvOffset31) >> shift);
02410                                           }
02411                                           yOffset += yw;
02412                                           buff16 += pitch16;
02413
02414                                           if (cb) {
02415                                                   percent += dP;
02416                                                   if ((*cb)(percent,
true, data)) ReturnWithError(EscapePressed);
02417                                           }
02418                                   }
02419                           }
02420                   } else {
02421                           ASSERT(bpp == 8);
02422                           const int shift = __max(0, UsedBitsPerChannel()
- 8);
02423
02424                           for (i=0; i < h; i++) {
02425                                   UINT32 yPos = yOffset;
02426                                   for (j = 0; j < w; j++) {
02427                                           buff[j] = Clamp8((y[yPos++] +
yuvOffset31) >> shift);
02428                                   }
02429                                   yOffset += yw;
```

```
02430                                   buff += pitch;
02431
02432                                   if (cb) {
02433                                           percent += dP;
02434                                           if ((*cb)(percent, true,
data)) ReturnWithError(EscapePressed);
02435                                   }
02436                           }
02437                   }
02438                   break;
02439           }
02440 #endif
02441   case ImageModeRGB12:
02442           {
02443                   ASSERT(m_header.channels == 3);
02444                   ASSERT(m_header.bpp == m_header.channels*4);
02445                   ASSERT(bpp == m_header.channels*4);
02446                   ASSERT(!m_downsample);
02447
02448                   DataT* y = m_channel[0]; ASSERT(y);
02449                   DataT* u = m_channel[1]; ASSERT(u);
02450                   DataT* v = m_channel[2]; ASSERT(v);
02451                   UINT16 yval;
02452                   UINT32 cnt;
02453
02454                   for (i=0; i < h; i++) {
02455                           UINT32 yPos = yOffset;
02456                           cnt = 0;
02457                           for (j=0; j < w; j++) {
02458                                   // Yuv
02459                                   uAvg = u[yPos];
02460                                   vAvg = v[yPos];
02461                                   yval = Clamp4(y[yPos] + YUVoffset4 -
((uAvg + vAvg ) >> 2)); // must be logical shift operator
02462                                   if (j%2 == 0) {
02463                                           buff[cnt] = UINT8(Clamp4(vAvg
+ yval) | (yval << 4));
02464                                           cnt++;
02465                                           buff[cnt] = Clamp4(uAvg +
yval);
02466                                   } else {
02467                                           buff[cnt] |= Clamp4(vAvg +
yval) << 4;
02468                                           cnt++;
02469                                           buff[cnt] = UINT8(yval |
(Clamp4(uAvg + yval) << 4));
02470                                           cnt++;
02471                                   }
02472                                   yPos++;
02473                           }
02474                           yOffset += yw;
02475                           buff += pitch;
02476
02477                           if (cb) {
02478                                   percent += dP;
02479                                   if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
02480                           }
02481                   }
02482                   break;
02483           }
02484   case ImageModeRGB16:
02485           {
02486                   ASSERT(m_header.channels == 3);
02487                   ASSERT(m_header.bpp == 16);
02488                   ASSERT(bpp == 16);
02489                   ASSERT(!m_downsample);
02490
02491                   DataT* y = m_channel[0]; ASSERT(y);
02492                   DataT* u = m_channel[1]; ASSERT(u);
02493                   DataT* v = m_channel[2]; ASSERT(v);
02494                   UINT16 yval;
02495                   UINT16 *buff16 = (UINT16 *)buff;
02496                   int pitch16 = pitch/2;
02497
02498                   for (i=0; i < h; i++) {
02499                           UINT32 yPos = yOffset;
```

```
02500                                   for (j = 0; j < w; j++) {
02501                                           // Yuv
02502                                           uAvg = u[yPos];
02503                                           vAvg = v[yPos];
02504                                           yval = Clamp6(y[yPos++] + YUVoffset6 -
((uAvg + vAvg ) >> 2)); // must be logical shift operator
02505                                           buff16[j] = (yval << 5) | ((Clamp6(uAvg
+ yval) >> 1) << 11) | (Clamp6(vAvg + yval) >> 1);
02506                                   }
02507                                   yOffset += yw;
02508                                   buff16 += pitch16;
02509
02510                                   if (cb) {
02511                                           percent += dP;
02512                                           if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
02513                                   }
02514                           }
02515                           break;
02516                   }
02517           default:
02518                   ASSERT(false);
02519           }
02520
02521 #ifdef _DEBUG
02522           // display ROI (RGB) in debugger
02523           roiimage.width = w;
02524           roiimage.height = h;
02525           if (pitch > 0) {
02526                   roiimage.pitch = pitch;
02527                   roiimage.data = buff;
02528           } else {
02529                   roiimage.pitch = -pitch;
02530                   roiimage.data = buff + (h - 1)*pitch;
02531           }
02532 #endif
02533
02534 }
```

DataT * CPGFImage::GetChannel (int *c* = 0) [inline]

Return an internal YUV image channel.

Parameters

c	A channel index

Returns

An internal YUV image channel

Definition at line **317** of file **PGFimage.h**.

```
00317 { ASSERT(c >= 0 && c < MaxChannels); return m_channel[c]; }
```

const RGBQUAD * CPGFImage::GetColorTable () const [inline]

Returns

Address of color table

Definition at line **330** of file **PGFimage.h**.

```
00330 { return m_postHeader.clut; }
```

void CPGFImage::GetColorTable (UINT32 *iFirstColor*, UINT32 *nColors*, RGBQUAD * *prgbColors*) const

Retrieves red, green, blue (RGB) color values from a range of entries in the palette of the DIB section. It might throw an **IOException**.

Parameters

iFirstColor	The color table index of the first entry to retrieve.
nColors	The number of color table entries to retrieve.

prgbColors	A pointer to the array of RGBQUAD structures to retrieve the color table entries.

Definition at line **1350** of file **PGFimage.cpp**.

```
01350
{
01351         if (iFirstColor + nColors > ColorTableLen)
ReturnWithError(ColorTableError);
01352
01353         for (UINT32 i=iFirstColor, j=0; j < nColors; i++, j++) {
01354                 prgbColors[j] = m_postHeader.clut[i];
01355         }
01356 }
```

UINT32 CPGFImage::GetEncodedHeaderLength () const

Return the length of all encoded headers in bytes. Precondition: The PGF image has been opened with a call of Open(...).

Returns

The length of all encoded headers in bytes

Definition at line **648** of file **PGFimage.cpp**.

```
00648                                              {
00649         ASSERT(m_decoder);
00650         return m_decoder->GetEncodedHeaderLength();
00651 }
```

UINT32 CPGFImage::GetEncodedLevelLength (int *level*) const [inline]

Return the length of an encoded PGF level in bytes. Precondition: The PGF image has been opened with a call of Open(...).

Parameters

level	The image level

Returns

The length of a PGF level in bytes

Definition at line **367** of file **PGFimage.h**.

```
00367 { ASSERT(level >= 0 && level < m_header.nLevels); return
m_levelLength[m_header.nLevels - level - 1]; }
```

const PGFHeader * CPGFImage::GetHeader () const [inline]

Return the PGF header structure.

Returns

A PGF header structure

Definition at line **335** of file **PGFimage.h**.

```
00335 { return &m_header; }
```

UINT32 CPGFImage::GetMaxValue () const [inline]

Get maximum intensity value for image modes with more than eight bits per channel. Don't call this method before the PGF header has been read.

Returns

The maximum intensity value.

Definition at line **341** of file **PGFimage.h**.

```
00341 { return (1 << m_header.usedBitsPerChannel) - 1; }
```

const UINT8 * CPGFImage::GetUserData (UINT32 & *cachedSize*, UINT32 * *pTotalSize* = `nullptr`) const

Return user data and size of user data. Precondition: The PGF image has been opened with a call of Open(...).

Parameters

cachedSize	[out] Size of returned user data in bytes.
pTotalSize	[optional out] Pointer to return the size of user data stored in image header in bytes.

Returns

A pointer to user data or nullptr if there is no user data available.

Return user data and size of user data. Precondition: The PGF image has been opened with a call of Open(...). In an encoder scenario don't call this method before **WriteHeader()**.

Parameters

cachedSize	[out] Size of returned user data in bytes.
pTotalSize	[optional out] Pointer to return the size of user data stored in image header in bytes.

Returns

A pointer to user data or nullptr if there is no user data available.

Definition at line **337** of file **PGFimage.cpp**.

```
00337
{
00338         cachedSize = m_postHeader.cachedUserDataLen;
00339         if (pTotalSize) *pTotalSize = m_postHeader.userDataLen;
00340         return m_postHeader.userData;
00341 }
```

UINT64 CPGFImage::GetUserDataPos () const`[inline]`

Return the stream position of the user data or 0. Precondition: The PGF image has been opened with a call of Open(...).

Definition at line **346** of file **PGFimage.h**.

```
00346 { return m_userDataPos; }
```

void CPGFImage::GetYUV (int *pitch*, DataT * *buff*, BYTE *bpp*, int *channelMap*[] = `nullptr`, CallbackPtr *cb* = `nullptr`, void * *data* = `nullptr`) const

Get YUV image data in interleaved format: (ordering is YUV[A]) The absolute value of pitch is the number of bytes of an image row of the given image buffer. If pitch is negative, then the image buffer must point to the last row of a bottom-up image (first byte on last row). if pitch is positive, then the image buffer must point to the first row of a top-down image (first byte). The sequence of output channels in the output image buffer does not need to be the same as provided by PGF. In case of different sequences you have to provide a channelMap of size of expected channels (depending on image mode). For example, PGF provides a channel sequence BGR in RGB color mode. If your provided image buffer expects a channel sequence VUY, then the channelMap looks like { 2, 1, 0 }. It might throw an **IOException**.

Parameters

pitch	The number of bytes of a row of the image buffer.
buff	An image buffer.
bpp	The number of bits per pixel used in image buffer.
channelMap	A integer array containing the mapping of PGF channel ordering to expected channel ordering.
cb	A pointer to a callback procedure. The procedure is called after each copied buffer row. If cb returns true, then it stops proceeding.

data	Data Pointer to C++ class container to host callback procedure.

Get YUV image data in interleaved format: (ordering is YUV[A]) The absolute value of pitch is the number of bytes of an image row of the given image buffer. If pitch is negative, then the image buffer must point to the last row of a bottom-up image (first byte on last row). if pitch is positive, then the image buffer must point to the first row of a top-down image (first byte). The sequence of output channels in the output image buffer does not need to be the same as provided by PGF. In case of different sequences you have to provide a channelMap of size of expected channels (depending on image mode). For example, PGF provides a channel sequence BGR in RGB color mode. If your provided image buffer expects a channel sequence VUY, then the channelMap looks like { 2, 1, 0 }. It might throw an **IOException**.

Parameters

pitch	The number of bytes of a row of the image buffer.
buff	An image buffer.
bpp	The number of bits per pixel used in image buffer.
channelMap	A integer array containing the mapping of PGF channel ordering to expected channel ordering.
cb	A pointer to a callback procedure. The procedure is called after each copied buffer row. If cb returns true, then it stops proceeding.

Definition at line **2550** of file **PGFimage.cpp**.

```
02550
{
02551          ASSERT(buff);
02552          const UINT32 w = m_width[0];
02553          const UINT32 h = m_height[0];
02554          const bool wOdd = (1 == w%2);
02555          const int dataBits = DataTSize*8; ASSERT(dataBits == 16 || dataBits ==
32);
02556          const int pitch2 = pitch/DataTSize;
02557          const int yuvOffset = (dataBits == 16) ? YUVoffset8 : YUVoffset16;
02558          const double dP = 1.0/h;
02559
02560          int defMap[] = { 0, 1, 2, 3, 4, 5, 6, 7 };
ASSERT(sizeof(defMap)/sizeof(defMap[0]) == MaxChannels);
02561          if (channelMap == nullptr) channelMap = defMap;
02562          int sampledPos = 0, yPos = 0;
02563          DataT uAvg, vAvg;
02564          double percent = 0;
02565          UINT32 i, j;
02566
02567          if (m_header.channels == 3) {
02568                  ASSERT(bpp%dataBits == 0);
02569
02570                  DataT* y = m_channel[0]; ASSERT(y);
02571                  DataT* u = m_channel[1]; ASSERT(u);
02572                  DataT* v = m_channel[2]; ASSERT(v);
02573                  int cnt, channels = bpp/dataBits; ASSERT(channels >=
m_header.channels);
02574
02575                  for (i=0; i < h; i++) {
02576                          if (i%2) sampledPos -= (w + 1)/2;
02577                          cnt = 0;
02578                          for (j=0; j < w; j++) {
02579                                  if (m_downsample) {
02580                                          // image was downsampled
02581                                          uAvg = u[sampledPos];
02582                                          vAvg = v[sampledPos];
02583                                  } else {
02584                                          uAvg = u[yPos];
02585                                          vAvg = v[yPos];
02586                                  }
02587                                  buff[cnt + channelMap[0]] = y[yPos];
02588                                  buff[cnt + channelMap[1]] = uAvg;
02589                                  buff[cnt + channelMap[2]] = vAvg;
02590                                  yPos++;
02591                                  cnt += channels;
02592                                  if (j%2) sampledPos++;
02593                          }
```

```
02594                           buff += pitch2;
02595                           if (wOdd) sampledPos++;
02596
02597                           if (cb) {
02598                                   percent += dP;
02599                                   if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
02600                           }
02601                   }
02602           } else if (m_header.channels == 4) {
02603                   ASSERT(m_header.bpp == m_header.channels*8);
02604                   ASSERT(bpp%dataBits == 0);
02605
02606                   DataT* y = m_channel[0]; ASSERT(y);
02607                   DataT* u = m_channel[1]; ASSERT(u);
02608                   DataT* v = m_channel[2]; ASSERT(v);
02609                   DataT* a = m_channel[3]; ASSERT(a);
02610                   UINT8 aAvg;
02611                   int cnt, channels = bpp/dataBits; ASSERT(channels >=
m_header.channels);
02612
02613                   for (i=0; i < h; i++) {
02614                           if (i%2) sampledPos -= (w + 1)/2;
02615                           cnt = 0;
02616                           for (j=0; j < w; j++) {
02617                                   if (m_downsample) {
02618                                           // image was downsampled
02619                                           uAvg = u[sampledPos];
02620                                           vAvg = v[sampledPos];
02621                                           aAvg = Clamp8(a[sampledPos] +
yuvOffset);
02622                                   } else {
02623                                           uAvg = u[yPos];
02624                                           vAvg = v[yPos];
02625                                           aAvg = Clamp8(a[yPos] + yuvOffset);
02626                                   }
02627                                   // Yuv
02628                                   buff[cnt + channelMap[0]] = y[yPos];
02629                                   buff[cnt + channelMap[1]] = uAvg;
02630                                   buff[cnt + channelMap[2]] = vAvg;
02631                                   buff[cnt + channelMap[3]] = aAvg;
02632                                   yPos++;
02633                                   cnt += channels;
02634                                   if (j%2) sampledPos++;
02635                           }
02636                           buff += pitch2;
02637                           if (wOdd) sampledPos++;
02638
02639                           if (cb) {
02640                                   percent += dP;
02641                                   if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
02642                           }
02643                   }
02644           }
02645 }
```

UINT32 CPGFImage::Height (int *level* = 0) const [inline]

Return image height of channel 0 at given level in pixels. The returned height is independent of any Read-operations and ROI.

Parameters

level	A level

Returns

Image level height in pixels

Definition at line **420** of file **PGFimage.h**.

```
00420 { ASSERT(level >= 0); return LevelSizeL(m_header.height, level); }
```

void CPGFImage::ImportBitmap (int *pitch*, UINT8 * *buff*, BYTE *bpp*, int *channelMap*[] = `nullptr`, CallbackPtr *cb* = `nullptr`, void * *data* = `nullptr`)

Import an image from a specified image buffer. This method is usually called before Write(...) and after SetHeader(...). The absolute value of pitch is the number of bytes of an image row. If pitch is negative, then buff points to the last row of a bottom-up image (first byte on last row). If pitch is positive, then buff points to the first row of a top-down image (first byte). The sequence of input channels in the input image buffer does not need to be the same as expected from PGF. In case of different sequences you have to provide a channelMap of size of expected channels (depending on image mode). For example, PGF expects in RGB color mode a channel sequence BGR. If your provided image buffer contains a channel sequence ARGB, then the channelMap looks like { 3, 2, 1, 0 }. It might throw an **IOException**.

Parameters

pitch	The number of bytes of a row of the image buffer.
buff	An image buffer.
bpp	The number of bits per pixel used in image buffer.
channelMap	A integer array containing the mapping of input channel ordering to expected channel ordering.
cb	A pointer to a callback procedure. The procedure is called after each imported buffer row. If cb returns true, then it stops proceeding.
data	Data Pointer to C++ class container to host callback procedure.

Definition at line **792** of file **PGFimage.cpp**.

```
00792
{
00793         ASSERT(buff);
00794         ASSERT(m_channel[0]);
00795
00796         // color transform
00797         RgbToYuv(pitch, buff, bpp, channelMap, cb, data);
00798
00799         if (m_downsample) {
00800                 // Subsampling of the chrominance and alpha channels
00801                 for (int i=1; i < m_header.channels; i++) {
00802                         Downsample(i);
00803                 }
00804         }
00805 }
```

bool CPGFImage::ImportIsSupported (BYTE *mode*) `[static]`

Check for valid import image mode.

Parameters

mode	Image mode

Returns

True if an image of given mode can be imported with ImportBitmap(...)

Definition at line **1305** of file **PGFimage.cpp**.

```
01305                                            {
01306         size_t size = DataTSize;
01307
01308         if (size >= 2) {
01309                 switch(mode) {
01310                         case ImageModeBitmap:
01311                         case ImageModeIndexedColor:
01312                         case ImageModeGrayScale:
01313                         case ImageModeRGBColor:
01314                         case ImageModeCMYKColor:
01315                         case ImageModeHSLColor:
01316                         case ImageModeHSBColor:
01317                         //case ImageModeDuotone:
01318                         case ImageModeLabColor:
01319                         case ImageModeRGB12:
01320                         case ImageModeRGB16:
```

```
01321                         case ImageModeRGBA:
01322                                 return true;
01323                 }
01324         }
01325         if (size >= 3) {
01326                 switch(mode) {
01327                         case ImageModeGray16:
01328                         case ImageModeRGB48:
01329                         case ImageModeLab48:
01330                         case ImageModeCMYK64:
01331                         //case ImageModeDuotone16:
01332                                 return true;
01333                 }
01334         }
01335         if (size >=4) {
01336                 switch(mode) {
01337                         case ImageModeGray32:
01338                                 return true;
01339                 }
01340         }
01341         return false;
01342 }
```

void CPGFImage::ImportYUV (int *pitch*, DataT * *buff*, BYTE *bpp*, int *channelMap*[] = `nullptr`, CallbackPtr *cb* = `nullptr`, void * *data* = `nullptr`)

Import a YUV image from a specified image buffer. The absolute value of pitch is the number of bytes of an image row. If pitch is negative, then buff points to the last row of a bottom-up image (first byte on last row). If pitch is positive, then buff points to the first row of a top-down image (first byte). The sequence of input channels in the input image buffer does not need to be the same as expected from PGF. In case of different sequences you have to provide a channelMap of size of expected channels (depending on image mode). For example, PGF expects in RGB color mode a channel sequence BGR. If your provided image buffer contains a channel sequence VUY, then the channelMap looks like { 2, 1, 0 }. It might throw an **IOException**.

Parameters

pitch	The number of bytes of a row of the image buffer.
buff	An image buffer.
bpp	The number of bits per pixel used in image buffer.
channelMap	A integer array containing the mapping of input channel ordering to expected channel ordering.
cb	A pointer to a callback procedure. The procedure is called after each imported buffer row. If cb returns true, then it stops proceeding.
data	Data Pointer to C++ class container to host callback procedure.

Import a YUV image from a specified image buffer. The absolute value of pitch is the number of bytes of an image row. If pitch is negative, then buff points to the last row of a bottom-up image (first byte on last row). If pitch is positive, then buff points to the first row of a top-down image (first byte). The sequence of input channels in the input image buffer does not need to be the same as expected from PGF. In case of different sequences you have to provide a channelMap of size of expected channels (depending on image mode). For example, PGF expects in RGB color mode a channel sequence BGR. If your provided image buffer contains a channel sequence VUY, then the channelMap looks like { 2, 1, 0 }. It might throw an **IOException**.

Parameters

pitch	The number of bytes of a row of the image buffer.
buff	An image buffer.
bpp	The number of bits per pixel used in image buffer.
channelMap	A integer array containing the mapping of input channel ordering to expected channel ordering.
cb	A pointer to a callback procedure. The procedure is called after each imported buffer row. If cb returns true, then it stops proceeding.

Definition at line **2661** of file **PGFimage.cpp**.

```
02661
{
02662         ASSERT(buff);
02663         const double dP = 1.0/m_header.height;
02664         const int dataBits = DataTSize*8; ASSERT(dataBits == 16 || dataBits ==
32);
02665         const int pitch2 = pitch/DataTSize;
02666         const int yuvOffset = (dataBits == 16) ? YUVoffset8 : YUVoffset16;
02667
02668         int yPos = 0, cnt = 0;
02669         double percent = 0;
02670         int defMap[] = { 0, 1, 2, 3, 4, 5, 6, 7 };
ASSERT(sizeof(defMap)/sizeof(defMap[0]) == MaxChannels);
02671
02672         if (channelMap == nullptr) channelMap = defMap;
02673
02674         if (m_header.channels == 3)     {
02675                 ASSERT(bpp%dataBits == 0);
02676
02677                 DataT* y = m_channel[0]; ASSERT(y);
02678                 DataT* u = m_channel[1]; ASSERT(u);
02679                 DataT* v = m_channel[2]; ASSERT(v);
02680                 const int channels = bpp/dataBits; ASSERT(channels >=
m_header.channels);
02681
02682                 for (UINT32 h=0; h < m_header.height; h++) {
02683                         if (cb) {
02684                                 if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
02685                                 percent += dP;
02686                         }
02687
02688                         cnt = 0;
02689                         for (UINT32 w=0; w < m_header.width; w++) {
02690                                 y[yPos] = buff[cnt + channelMap[0]];
02691                                 u[yPos] = buff[cnt + channelMap[1]];
02692                                 v[yPos] = buff[cnt + channelMap[2]];
02693                                 yPos++;
02694                                 cnt += channels;
02695                         }
02696                         buff += pitch2;
02697                 }
02698         } else if (m_header.channels == 4) {
02699                 ASSERT(bpp%dataBits == 0);
02700
02701                 DataT* y = m_channel[0]; ASSERT(y);
02702                 DataT* u = m_channel[1]; ASSERT(u);
02703                 DataT* v = m_channel[2]; ASSERT(v);
02704                 DataT* a = m_channel[3]; ASSERT(a);
02705                 const int channels = bpp/dataBits; ASSERT(channels >=
m_header.channels);
02706
02707                 for (UINT32 h=0; h < m_header.height; h++) {
02708                         if (cb) {
02709                                 if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
02710                                 percent += dP;
02711                         }
02712
02713                         cnt = 0;
02714                         for (UINT32 w=0; w < m_header.width; w++) {
02715                                 y[yPos] = buff[cnt + channelMap[0]];
02716                                 u[yPos] = buff[cnt + channelMap[1]];
02717                                 v[yPos] = buff[cnt + channelMap[2]];
02718                                 a[yPos] = buff[cnt + channelMap[3]] -
yuvOffset;
02719                                 yPos++;
02720                                 cnt += channels;
02721                         }
02722                         buff += pitch2;
02723                 }
02724         }
02725
02726         if (m_downsample) {
02727                 // Subsampling of the chrominance and alpha channels
```

```
02728                   for (int i=1; i < m_header.channels; i++) {
02729                           Downsample(i);
02730                   }
02731           }
02732 }
```

void CPGFImage::Init () [private]

Definition at line **69** of file **PGFimage.cpp**.

```
00069                        {
00070           // init pointers
00071           m_decoder = nullptr;
00072           m_encoder = nullptr;
00073           m_levelLength = nullptr;
00074 
00075           // init members
00076 #ifdef __PGFROISUPPORT__
00077           m_streamReinitialized = false;
00078 #endif
00079           m_currentLevel = 0;
00080           m_quant = 0;
00081           m_userDataPos = 0;
00082           m_downsample = false;
00083           m_favorSpeedOverSize = false;
00084           m_useOMPinEncoder = true;
00085           m_useOMPinDecoder = true;
00086           m_cb = nullptr;
00087           m_cbArg = nullptr;
00088           m_progressMode = PM_Relative;
00089           m_percent = 0;
00090           m_userDataPolicy = UP_CacheAll;
00091 
00092           // init preHeader
00093           memcpy(m_preHeader.magic, PGFMagic, 3);
00094           m_preHeader.version = PGFVersion;
00095           m_preHeader.hSize = 0;
00096 
00097           // init postHeader
00098           m_postHeader.userData = nullptr;
00099           m_postHeader.userDataLen = 0;
00100           m_postHeader.cachedUserDataLen = 0;
00101 
00102           // init channels
00103           for (int i = 0; i < MaxChannels; i++) {
00104                   m_channel[i] = nullptr;
00105                   m_wtChannel[i] = nullptr;
00106           }
00107 
00108           // set image width and height
00109           for (int i = 0; i < MaxChannels; i++) {
00110                   m_width[0] = 0;
00111                   m_height[0] = 0;
00112           }
00113 }
```

bool CPGFImage::IsFullyRead () const [inline]

Return true if all levels have been read.

Definition at line **436** of file **PGFimage.h**.

```
00436 { return m_currentLevel == 0; }
```

bool CPGFImage::IsOpen () const [inline]

Returns true if the PGF has been opened for reading.

Definition at line **77** of file **PGFimage.h**.

```
00077 { return m_decoder != nullptr; }
```

BYTE CPGFImage::Level () const [inline]

Return current image level. Since Read(...) can be used to read each image level separately, it is helpful to know the current level. The current level immediately after Open(...) is **Levels()**.

Returns

Current image level

Definition at line **427** of file **PGFimage.h**.

```
00427 { return (BYTE)m_currentLevel; }
```

BYTE CPGFImage::Levels () const [inline]

Return the number of image levels.

Returns

Number of image levels

Definition at line **432** of file **PGFimage.h**.

```
00432 { return m_header.nLevels; }
```

static UINT32 CPGFImage::LevelSizeH (UINT32 *size*, int *level*) [inline], [static]

Compute and return image width/height of HH subband at given level.

Parameters

size	Original image size (e.g. width or height at level 0)
level	An image level

Returns

high pass size at given level in pixels

Definition at line **506** of file **PGFimage.h**.

```
00506 { ASSERT(level >= 0); UINT32 d = 1 << (level - 1); return (size + d - 1) >> level;
}
```

static UINT32 CPGFImage::LevelSizeL (UINT32 *size*, int *level*) [inline], [static]

Compute and return image width/height of LL subband at given level.

Parameters

size	Original image size (e.g. width or height at level 0)
level	An image level

Returns

Image width/height at given level in pixels

Definition at line **499** of file **PGFimage.h**.

```
00499 { ASSERT(level >= 0); UINT32 d = 1 << level; return (size + d - 1) >> level; }
```

static BYTE CPGFImage::MaxChannelDepth (BYTE *version* = PGFVersion) [inline], [static]

Return maximum channel depth.

Parameters

version	pgf pre-header version number

Returns

maximum channel depth in bit of given version (16 or 32 bit)

Definition at line **518** of file **PGFimage.h**.

```
00518 { return (version & PGF32) ? 32 : 16; }
```

BYTE CPGFImage::Mode () const[inline]

Return the image mode. An image mode is a predefined constant value (see also **PGFtypes.h**) compatible with Adobe Photoshop. It represents an image type and format.

Returns

Image mode

Definition at line **455** of file **PGFimage.h**.

```
00455 { return m_header.mode; }
```

void CPGFImage::Open (CPGFStream * *stream*)

Open a PGF image at current stream position: read pre-header, header, and ckeck image type. Precondition: The stream has been opened for reading. It might throw an **IOException**.

Parameters

stream	A PGF stream

Definition at line **141** of file **PGFimage.cpp**.

```
00141                                       {
00142         ASSERT(stream);
00143
00144         // create decoder and read PGFPreHeader PGFHeader PGFPostHeader
LevelLengths
00145         m_decoder = new CDecoder(stream, m_preHeader, m_header, m_postHeader,
m_levelLength,
00146                 m_userDataPos, m_useOMPinDecoder, m_userDataPolicy);
00147
00148         if (m_header.nLevels > MaxLevel) ReturnWithError(FormatCannotRead);
00149
00150         // set current level
00151         m_currentLevel = m_header.nLevels;
00152
00153         // set image width and height
00154         m_width[0] = m_header.width;
00155         m_height[0] = m_header.height;
00156
00157         // complete header
00158         if (!CompleteHeader()) ReturnWithError(FormatCannotRead);
00159
00160         // interpret quant parameter
00161         if (m_header.quality > DownsampleThreshold &&
00162                 (m_header.mode == ImageModeRGBColor ||
00163                  m_header.mode == ImageModeRGBA ||
00164                  m_header.mode == ImageModeRGB48 ||
00165                  m_header.mode == ImageModeCMYKColor ||
00166                  m_header.mode == ImageModeCMYK64 ||
00167                  m_header.mode == ImageModeLabColor ||
00168                  m_header.mode == ImageModeLab48)) {
00169                 m_downsample = true;
00170                 m_quant = m_header.quality - 1;
00171         } else {
00172                 m_downsample = false;
00173                 m_quant = m_header.quality;
00174         }
00175
00176         // set channel dimensions (chrominance is subsampled by factor 2)
00177         if (m_downsample) {
00178                 for (int i=1; i < m_header.channels; i++) {
00179                         m_width[i] = (m_width[0] + 1) >> 1;
00180                         m_height[i] = (m_height[0] + 1) >> 1;
00181                 }
00182         } else {
00183                 for (int i=1; i < m_header.channels; i++) {
00184                         m_width[i] = m_width[0];
00185                         m_height[i] = m_height[0];
00186                 }
00187         }
00188
00189         if (m_header.nLevels > 0) {
00190                 // init wavelet subbands
```

```
00191                   for (int i=0; i < m_header.channels; i++) {
00192                           m_wtChannel[i] = new CWaveletTransform(m_width[i],
m_height[i], m_header.nLevels);
00193                   }
00194
00195                   // used in Read when PM_Absolute
00196                   m_percent = pow(0.25, m_header.nLevels);
00197
00198           } else {
00199                   // very small image: we don't use DWT and encoding
00200
00201                   // read channels
00202                   for (int c=0; c < m_header.channels; c++) {
00203                           const UINT32 size = m_width[c]*m_height[c];
00204                           m_channel[c] = new(std::nothrow) DataT[size];
00205                           if (!m_channel[c])
ReturnWithError(InsufficientMemory);
00206
00207                           // read channel data from stream
00208                           for (UINT32 i=0; i < size; i++) {
00209                                   int count = DataTSize;
00210                                   stream->Read(&count, &m_channel[c][i]);
00211                                   if (count != DataTSize)
ReturnWithError(MissingData);
00212                           }
00213                   }
00214           }
00215 }
```

BYTE CPGFImage::Quality () const`[inline]`

Return the PGF quality. The quality is inbetween 0 and MaxQuality. PGF quality 0 means lossless quality.

Returns

PGF quality

Definition at line **442** of file **PGFimage.h**.

```
00442 { return m_header.quality; }
```

void CPGFImage::Read (int *level* = 0, CallbackPtr *cb* = `nullptr`, void * *data* = `nullptr`)

Read and decode some levels of a PGF image at current stream position. A PGF image is structered in levels, numbered between 0 and **Levels()** - 1. Each level can be seen as a single image, containing the same content as all other levels, but in a different size (width, height). The image size at level i is double the size (width, height) of the image at level i+1. The image at level 0 contains the original size. Precondition: The PGF image has been opened with a call of Open(...). It might throw an **IOException**.

Parameters

level	[0, nLevels) The image level of the resulting image in the internal image buffer.
cb	A pointer to a callback procedure. The procedure is called after reading a single level. If cb returns true, then it stops proceeding.
data	Data Pointer to C++ class container to host callback procedure.

Definition at line **402** of file **PGFimage.cpp**.

```
00402                                                                        {
00403           ASSERT((level >= 0 && level < m_header.nLevels) || m_header.nLevels ==
0); // m_header.nLevels == 0: image didn't use wavelet transform
00404           ASSERT(m_decoder);
00405
00406 #ifdef __PGFROISUPPORT__
00407           if (ROIisSupported() && m_header.nLevels > 0) {
00408                   // new encoding scheme supporting ROI
00409                   PGFRect rect(0, 0, m_header.width, m_header.height);
00410                   Read(rect, level, cb, data);
00411                   return;
00412           }
```

```
00413 #endif
00414
00415         if (m_header.nLevels == 0) {
00416                 if (level == 0) {
00417                         // the data has already been read during open
00418                         // now update progress
00419                         if (cb) {
00420                                 if ((*cb)(1.0, true, data))
ReturnWithError(EscapePressed);
00421                         }
00422                 }
00423         } else {
00424                 const int levelDiff = m_currentLevel - level;
00425                 double percent = (m_progressMode == PM_Relative) ? pow(0.25,
levelDiff) : m_percent;
00426
00427                 // encoding scheme without ROI
00428                 while (m_currentLevel > level) {
00429                         for (int i=0; i < m_header.channels; i++) {
00430                                 CWaveletTransform* wtChannel = m_wtChannel[i];
00431                                 ASSERT(wtChannel);
00432
00433                                 // decode file and write stream to m_wtChannel
00434                                 if (m_currentLevel == m_header.nLevels) {
00435                                         // last level also has LL band
00436
wtChannel->GetSubband(m_currentLevel, LL)->PlaceTile(*m_decoder, m_quant);
00437                                 }
00438                                 if (m_preHeader.version & Version5) {
00439                                         // since version 5
00440
wtChannel->GetSubband(m_currentLevel, HL)->PlaceTile(*m_decoder, m_quant);
00441
wtChannel->GetSubband(m_currentLevel, LH)->PlaceTile(*m_decoder, m_quant);
00442                                 } else {
00443                                         // until version 4
00444
m_decoder->DecodeInterleaved(wtChannel, m_currentLevel, m_quant);
00445                                 }
00446                                 wtChannel->GetSubband(m_currentLevel,
HH)->PlaceTile(*m_decoder, m_quant);
00447                         }
00448
00449                         volatile OSError error = NoError; // volatile prevents
optimizations
00450 #ifdef LIBPGF_USE_OPENMP
00451                         #pragma omp parallel for default(shared)
00452 #endif
00453                         for (int i=0; i < m_header.channels; i++) {
00454                                 // inverse transform from m_wtChannel to
m_channel
00455                                 if (error == NoError) {
00456                                         OSError err =
m_wtChannel[i]->InverseTransform(m_currentLevel, &m_width[i], &m_height[i],
&m_channel[i]);
00457                                         if (err != NoError) error = err;
00458                                 }
00459                                 ASSERT(m_channel[i]);
00460                         }
00461                         if (error != NoError) ReturnWithError(error);
00462
00463                         // set new level: must be done before refresh callback
00464                         m_currentLevel--;
00465
00466                         // now we have to refresh the display
00467                         if (m_cb) m_cb(m_cbArg);
00468
00469                         // now update progress
00470                         if (cb) {
00471                                 percent *= 4;
00472                                 if (m_progressMode == PM_Absolute) m_percent =
percent;
00473                                 if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
00474                         }
00475                 }
00476         }
```

```
00477 }
```

void CPGFImage::Read (PGFRect & *rect*, int *level* = 0, CallbackPtr *cb* = nullptr, void * *data* = nullptr)

Read a rectangular region of interest of a PGF image at current stream position. The origin of the coordinate axis is the top-left corner of the image. All coordinates are measured in pixels. It might throw an **IOException**.

Parameters

rect	[inout] Rectangular region of interest (ROI) at level 0. The rect might be cropped.
level	[0, nLevels) The image level of the resulting image in the internal image buffer.
cb	A pointer to a callback procedure. The procedure is called after reading a single level. If cb returns true, then it stops proceeding.
data	Data Pointer to C++ class container to host callback procedure.

UINT32 CPGFImage::ReadEncodedData (int *level*, UINT8 * *target*, UINT32 *targetLen*) const

Reads the data of an encoded PGF level and copies it to a target buffer without decoding. Precondition: The PGF image has been opened with a call of Open(...). It might throw an **IOException**.

Parameters

level	The image level
target	The target buffer
targetLen	The length of the target buffer in bytes

Returns

The number of bytes copied to the target buffer

Definition at line **707** of file **PGFimage.cpp**.

```
00707
{
00708         ASSERT(level >= 0 && level < m_header.nLevels);
00709         ASSERT(target);
00710         ASSERT(targetLen > 0);
00711         ASSERT(m_decoder);
00712
00713         // reset stream position
00714         m_decoder->SetStreamPosToData();
00715
00716         // position stream
00717         UINT64 offset = 0;
00718
00719         for (int i=m_header.nLevels - 1; i > level; i--) {
00720                 offset += m_levelLength[m_header.nLevels - 1 - i];
00721         }
00722         m_decoder->Skip(offset);
00723
00724         // compute number of bytes to read
00725         UINT32 len = __min(targetLen, GetEncodedLevelLength(level));
00726
00727         // read data
00728         len = m_decoder->ReadEncodedData(target, len);
00729         ASSERT(len >= 0 && len <= targetLen);
00730
00731         return len;
00732 }
```

UINT32 CPGFImage::ReadEncodedHeader (UINT8 * *target*, UINT32 *targetLen*) const

Reads the encoded PGF header and copies it to a target buffer. Precondition: The PGF image has been opened with a call of Open(...). It might throw an **IOException**.

Parameters

target	The target buffer
targetLen	The length of the target buffer in bytes

Returns

The number of bytes copied to the target buffer

Definition at line **660** of file **PGFimage.cpp**.

```
00660                                                                          {
00661         ASSERT(target);
00662         ASSERT(targetLen > 0);
00663         ASSERT(m_decoder);
00664 
00665         // reset stream position
00666         m_decoder->SetStreamPosToStart();
00667 
00668         // compute number of bytes to read
00669         UINT32 len = __min(targetLen, GetEncodedHeaderLength());
00670 
00671         // read data
00672         len = m_decoder->ReadEncodedData(target, len);
00673         ASSERT(len >= 0 && len <= targetLen);
00674 
00675         return len;
00676 }
```

void CPGFImage::ReadPreview () `[inline]`

Read and decode smallest level of a PGF image at current stream position. For details, please refert to Read(...) Precondition: The PGF image has been opened with a call of Open(...). It might throw an **IOException**.

Definition at line **111** of file **PGFimage.h**.

```
00111 { Read(Levels() - 1); }
```

void CPGFImage::Reconstruct (int *level* = 0)

After you've written a PGF image, you can call this method followed by GetBitmap/GetYUV to get a quick reconstruction (coded -> decoded image). It might throw an **IOException**.

Parameters

level	The image level of the resulting image in the internal image buffer.

Definition at line **348** of file **PGFimage.cpp**.

```
00348                                      {
00349         if (m_header.nLevels == 0) {
00350                 // image didn't use wavelet transform
00351                 if (level == 0) {
00352                         for (int i=0; i < m_header.channels; i++) {
00353                                 ASSERT(m_wtChannel[i]);
00354                                 m_channel[i] = m_wtChannel[i]->GetSubband(0,
LL)->GetBuffer();
00355                         }
00356                 }
00357         } else {
00358                 int currentLevel = m_header.nLevels;
00359 
00360         #ifdef __PGFROISUPPORT__
00361                 if (ROIisSupported()) {
00362                         // enable ROI reading
00363                         SetROI(PGFRect(0, 0, m_header.width,
m_header.height));
00364                 }
00365         #endif
00366 
00367                 while (currentLevel > level) {
00368                         for (int i=0; i < m_header.channels; i++) {
00369                                 ASSERT(m_wtChannel[i]);
00370                                 // dequantize subbands
00371                                 if (currentLevel == m_header.nLevels) {
00372                                         // last level also has LL band
```

```
00373
m_wtChannel[i]->GetSubband(currentLevel, LL)->Dequantize(m_quant);
00374                                           }
00375                                           m_wtChannel[i]->GetSubband(currentLevel,
HL)->Dequantize(m_quant);
00376                                           m_wtChannel[i]->GetSubband(currentLevel,
LH)->Dequantize(m_quant);
00377                                           m_wtChannel[i]->GetSubband(currentLevel,
HH)->Dequantize(m_quant);
00378
00379                                           // inverse transform from m_wtChannel to
m_channel
00380                                           OSError err =
m_wtChannel[i]->InverseTransform(currentLevel, &m_width[i], &m_height[i],
&m_channel[i]);
00381                                           if (err != NoError) ReturnWithError(err);
00382                                           ASSERT(m_channel[i]);
00383                                   }
00384
00385                                   currentLevel--;
00386                           }
00387           }
00388 }
```

void CPGFImage::ResetStreamPos (bool *startOfData*)

Reset stream position to start of PGF pre-header or start of data. Must not be called before **Open()** or before **Write()**. Use this method after **Read()** if you want to read the same image several times, e.g. reading different ROIs.

Parameters

startOfData	true: you want to read the same image several times. false: resets stream position to the initial position

Definition at line **682** of file **PGFimage.cpp**.

```
00682                                                   {
00683           m_currentLevel = 0;
00684           if (startOfData) {
00685                   ASSERT(m_decoder);
00686                   m_decoder->SetStreamPosToData();
00687           } else {
00688                   if (m_decoder) {
00689                           m_decoder->SetStreamPosToStart();
00690                   } else if (m_encoder) {
00691                           m_encoder->SetStreamPosToStart();
00692                   } else {
00693                           ASSERT(false);
00694                   }
00695           }
00696 }
```

void CPGFImage::RgbToYuv (int *pitch*, UINT8 * *rgbBuff*, BYTE *bpp*, int *channelMap*[], CallbackPtr *cb*, void * *data*)`[private]`

Definition at line **1389** of file **PGFimage.cpp**.

```
01389
{
01390           ASSERT(buff);
01391           UINT32 yPos = 0, cnt = 0;
01392           double percent = 0;
01393           const double dP = 1.0/m_header.height;
01394           int defMap[] = { 0, 1, 2, 3, 4, 5, 6, 7 };
ASSERT(sizeof(defMap)/sizeof(defMap[0]) == MaxChannels);
01395
01396           if (channelMap == nullptr) channelMap = defMap;
01397
01398           switch(m_header.mode) {
01399           case ImageModeBitmap:
01400                   {
01401                           ASSERT(m_header.channels == 1);
01402                           ASSERT(m_header.bpp == 1);
01403                           ASSERT(bpp == 1);
```

```
01404
01405                       const UINT32 w = m_header.width;
01406                       const UINT32 w2 = (m_header.width + 7)/8;
01407                       DataT* y = m_channel[0]; ASSERT(y);
01408
01409                       // new unpacked version since version 7
01410                       for (UINT32 h = 0; h < m_header.height; h++) {
01411                               if (cb) {
01412                                       if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
01413                                       percent += dP;
01414                               }
01415                               cnt = 0;
01416                               for (UINT32 j = 0; j < w2; j++) {
01417                                       UINT8 byte = buff[j];
01418                                       for (int k = 0; k < 8; k++) {
01419                                               UINT8 bit = (byte & 0x80) >> 7;
01420                                               if (cnt < w) y[yPos++] = bit;
01421                                               byte <<= 1;
01422                                               cnt++;
01423                                       }
01424                               }
01425                               buff += pitch;
01426                       }
01427                       /* old version: packed values: 8 pixels in 1 byte
01428                       for (UINT32 h = 0; h < m_header.height; h++) {
01429                               if (cb) {
01430                                       if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
01431                                       percent += dP;
01432                               }
01433
01434                               for (UINT32 j = 0; j < w2; j++) {
01435                                       y[yPos++] = buff[j] - YUVoffset8;
01436                               }
01437                               // version 5 and 6
01438                               // for (UINT32 j = w2; j < w; j++) {
01439                               //      y[yPos++] = YUVoffset8;
01440                               //}
01441                               buff += pitch;
01442                       }
01443                       */
01444               }
01445               break;
01446       case ImageModeIndexedColor:
01447       case ImageModeGrayScale:
01448       case ImageModeHSLColor:
01449       case ImageModeHSBColor:
01450       case ImageModeLabColor:
01451               {
01452                       ASSERT(m_header.channels >= 1);
01453                       ASSERT(m_header.bpp == m_header.channels*8);
01454                       ASSERT(bpp%8 == 0);
01455                       const int channels = bpp/8; ASSERT(channels >=
m_header.channels);
01456
01457                       for (UINT32 h=0; h < m_header.height; h++) {
01458                               if (cb) {
01459                                       if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
01460                                       percent += dP;
01461                               }
01462
01463                               cnt = 0;
01464                               for (UINT32 w=0; w < m_header.width; w++) {
01465                                       for (int c=0; c < m_header.channels;
c++) {
01466                                               m_channel[c][yPos] = buff[cnt
+ channelMap[c]] - YUVoffset8;
01467                                       }
01468                                       cnt += channels;
01469                                       yPos++;
01470                               }
01471                               buff += pitch;
01472                       }
01473               }
01474               break;
```

```
01475         case ImageModeGray16:
01476         case ImageModeLab48:
01477                 {
01478                         ASSERT(m_header.channels >= 1);
01479                         ASSERT(m_header.bpp == m_header.channels*16);
01480                         ASSERT(bpp%16 == 0);
01481
01482                         UINT16 *buff16 = (UINT16 *)buff;
01483                         const int pitch16 = pitch/2;
01484                         const int channels = bpp/16; ASSERT(channels >=
m_header.channels);
01485                         const int shift = 16 - UsedBitsPerChannel();
ASSERT(shift >= 0);
01486                         const DataT yuvOffset16 = 1 << (UsedBitsPerChannel() -
1);
01487
01488                         for (UINT32 h=0; h < m_header.height; h++) {
01489                                 if (cb) {
01490                                         if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
01491                                         percent += dP;
01492                                 }
01493
01494                                 cnt = 0;
01495                                 for (UINT32 w=0; w < m_header.width; w++) {
01496                                         for (int c=0; c < m_header.channels;
c++) {
01497                                                 m_channel[c][yPos] =
(buff16[cnt + channelMap[c]] >> shift) - yuvOffset16;
01498                                         }
01499                                         cnt += channels;
01500                                         yPos++;
01501                                 }
01502                                 buff16 += pitch16;
01503                         }
01504                 }
01505                 break;
01506         case ImageModeRGBColor:
01507                 {
01508                         ASSERT(m_header.channels == 3);
01509                         ASSERT(m_header.bpp == m_header.channels*8);
01510                         ASSERT(bpp%8 == 0);
01511
01512                         DataT* y = m_channel[0]; ASSERT(y);
01513                         DataT* u = m_channel[1]; ASSERT(u);
01514                         DataT* v = m_channel[2]; ASSERT(v);
01515                         const int channels = bpp/8; ASSERT(channels >=
m_header.channels);
01516                         UINT8 b, g, r;
01517
01518                         for (UINT32 h=0; h < m_header.height; h++) {
01519                                 if (cb) {
01520                                         if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
01521                                         percent += dP;
01522                                 }
01523
01524                                 cnt = 0;
01525                                 for (UINT32 w=0; w < m_header.width; w++) {
01526                                         b = buff[cnt + channelMap[0]];
01527                                         g = buff[cnt + channelMap[1]];
01528                                         r = buff[cnt + channelMap[2]];
01529                                         // Yuv
01530                                         y[yPos] = ((b + (g << 1) + r) >> 2) -
YUVoffset8;
01531                                         u[yPos] = r - g;
01532                                         v[yPos] = b - g;
01533                                         yPos++;
01534                                         cnt += channels;
01535                                 }
01536                                 buff += pitch;
01537                         }
01538                 }
01539                 break;
01540         case ImageModeRGB48:
01541                 {
01542                         ASSERT(m_header.channels == 3);
```

```
01543                           ASSERT(m_header.bpp == m_header.channels*16);
01544                           ASSERT(bpp%16 == 0);
01545
01546                           UINT16 *buff16 = (UINT16 *)buff;
01547                           const int pitch16 = pitch/2;
01548                           const int channels = bpp/16; ASSERT(channels >=
m_header.channels);
01549                           const int shift = 16 - UsedBitsPerChannel();
ASSERT(shift >= 0);
01550                           const DataT yuvOffset16 = 1 << (UsedBitsPerChannel() -
1);
01551
01552                           DataT* y = m_channel[0]; ASSERT(y);
01553                           DataT* u = m_channel[1]; ASSERT(u);
01554                           DataT* v = m_channel[2]; ASSERT(v);
01555                           UINT16 b, g, r;
01556
01557                           for (UINT32 h=0; h < m_header.height; h++) {
01558                                   if (cb) {
01559                                           if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
01560                                           percent += dP;
01561                                   }
01562
01563                                   cnt = 0;
01564                                   for (UINT32 w=0; w < m_header.width; w++) {
01565                                           b = buff16[cnt + channelMap[0]] >>
shift;
01566                                           g = buff16[cnt + channelMap[1]] >>
shift;
01567                                           r = buff16[cnt + channelMap[2]] >>
shift;
01568                                           // Yuv
01569                                           y[yPos] = ((b + (g << 1) + r) >> 2) -
yuvOffset16;
01570                                           u[yPos] = r - g;
01571                                           v[yPos] = b - g;
01572                                           yPos++;
01573                                           cnt += channels;
01574                                   }
01575                                   buff16 += pitch16;
01576                           }
01577                   }
01578                   break;
01579           case ImageModeRGBA:
01580           case ImageModeCMYKColor:
01581                   {
01582                           ASSERT(m_header.channels == 4);
01583                           ASSERT(m_header.bpp == m_header.channels*8);
01584                           ASSERT(bpp%8 == 0);
01585                           const int channels = bpp/8; ASSERT(channels >=
m_header.channels);
01586
01587                           DataT* y = m_channel[0]; ASSERT(y);
01588                           DataT* u = m_channel[1]; ASSERT(u);
01589                           DataT* v = m_channel[2]; ASSERT(v);
01590                           DataT* a = m_channel[3]; ASSERT(a);
01591                           UINT8 b, g, r;
01592
01593                           for (UINT32 h=0; h < m_header.height; h++) {
01594                                   if (cb) {
01595                                           if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
01596                                           percent += dP;
01597                                   }
01598
01599                                   cnt = 0;
01600                                   for (UINT32 w=0; w < m_header.width; w++) {
01601                                           b = buff[cnt + channelMap[0]];
01602                                           g = buff[cnt + channelMap[1]];
01603                                           r = buff[cnt + channelMap[2]];
01604                                           // Yuv
01605                                           y[yPos] = ((b + (g << 1) + r) >> 2) -
YUVoffset8;
01606                                           u[yPos] = r - g;
01607                                           v[yPos] = b - g;
```

```
01608                                   a[yPos++] = buff[cnt + channelMap[3]]
- YUVoffset8;
01609                                   cnt += channels;
01610                           }
01611                           buff += pitch;
01612                   }
01613           }
01614           break;
01615   case ImageModeCMYK64:
01616           {
01617                   ASSERT(m_header.channels == 4);
01618                   ASSERT(m_header.bpp == m_header.channels*16);
01619                   ASSERT(bpp%16 == 0);
01620
01621                   UINT16 *buff16 = (UINT16 *)buff;
01622                   const int pitch16 = pitch/2;
01623                   const int channels = bpp/16; ASSERT(channels >=
m_header.channels);
01624                   const int shift = 16 - UsedBitsPerChannel();
ASSERT(shift >= 0);
01625                   const DataT yuvOffset16 = 1 << (UsedBitsPerChannel() -
1);
01626
01627                   DataT* y = m_channel[0]; ASSERT(y);
01628                   DataT* u = m_channel[1]; ASSERT(u);
01629                   DataT* v = m_channel[2]; ASSERT(v);
01630                   DataT* a = m_channel[3]; ASSERT(a);
01631                   UINT16 b, g, r;
01632
01633                   for (UINT32 h=0; h < m_header.height; h++) {
01634                           if (cb) {
01635                                   if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
01636                                   percent += dP;
01637                           }
01638
01639                           cnt = 0;
01640                           for (UINT32 w=0; w < m_header.width; w++) {
01641                                   b = buff16[cnt + channelMap[0]] >>
shift;
01642                                   g = buff16[cnt + channelMap[1]] >>
shift;
01643                                   r = buff16[cnt + channelMap[2]] >>
shift;
01644                                   // Yuv
01645                                   y[yPos] = ((b + (g << 1) + r) >> 2) -
yuvOffset16;
01646                                   u[yPos] = r - g;
01647                                   v[yPos] = b - g;
01648                                   a[yPos++] = (buff16[cnt +
channelMap[3]] >> shift) - yuvOffset16;
01649                                   cnt += channels;
01650                           }
01651                           buff16 += pitch16;
01652                   }
01653           }
01654           break;
01655 #ifdef __PGF32SUPPORT__
01656   case ImageModeGray32:
01657           {
01658                   ASSERT(m_header.channels == 1);
01659                   ASSERT(m_header.bpp == 32);
01660                   ASSERT(bpp == 32);
01661                   ASSERT(DataTSize == sizeof(UINT32));
01662
01663                   DataT* y = m_channel[0]; ASSERT(y);
01664
01665                   UINT32 *buff32 = (UINT32 *)buff;
01666                   const int pitch32 = pitch/4;
01667                   const int shift = 31 - UsedBitsPerChannel();
ASSERT(shift >= 0);
01668                   const DataT yuvOffset31 = 1 << (UsedBitsPerChannel() -
1);
01669
01670                   for (UINT32 h=0; h < m_header.height; h++) {
01671                           if (cb) {
```

```
01672                                   if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
01673                                   percent += dP;
01674                           }
01675
01676                           for (UINT32 w=0; w < m_header.width; w++) {
01677                                   y[yPos++] = (buff32[w] >> shift) -
yuvOffset31;
01678                           }
01679                           buff32 += pitch32;
01680                   }
01681           }
01682           break;
01683 #endif
01684   case ImageModeRGB12:
01685           {
01686                   ASSERT(m_header.channels == 3);
01687                   ASSERT(m_header.bpp == m_header.channels*4);
01688                   ASSERT(bpp == m_header.channels*4);
01689
01690                   DataT* y = m_channel[0]; ASSERT(y);
01691                   DataT* u = m_channel[1]; ASSERT(u);
01692                   DataT* v = m_channel[2]; ASSERT(v);
01693
01694                   UINT8 rgb = 0, b, g, r;
01695
01696                   for (UINT32 h=0; h < m_header.height; h++) {
01697                           if (cb) {
01698                                   if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
01699                                   percent += dP;
01700                           }
01701
01702                           cnt = 0;
01703                           for (UINT32 w=0; w < m_header.width; w++) {
01704                                   if (w%2 == 0) {
01705                                           // even pixel position
01706                                           rgb = buff[cnt];
01707                                           b = rgb & 0x0F;
01708                                           g = (rgb & 0xF0) >> 4;
01709                                           cnt++;
01710                                           rgb = buff[cnt];
01711                                           r = rgb & 0x0F;
01712                                   } else {
01713                                           // odd pixel position
01714                                           b = (rgb & 0xF0) >> 4;
01715                                           cnt++;
01716                                           rgb = buff[cnt];
01717                                           g = rgb & 0x0F;
01718                                           r = (rgb & 0xF0) >> 4;
01719                                           cnt++;
01720                                   }
01721
01722                                   // Yuv
01723                                   y[yPos] = ((b + (g << 1) + r) >> 2) -
YUVoffset4;
01724                                   u[yPos] = r - g;
01725                                   v[yPos] = b - g;
01726                                   yPos++;
01727                           }
01728                           buff += pitch;
01729                   }
01730           }
01731           break;
01732   case ImageModeRGB16:
01733           {
01734                   ASSERT(m_header.channels == 3);
01735                   ASSERT(m_header.bpp == 16);
01736                   ASSERT(bpp == 16);
01737
01738                   DataT* y = m_channel[0]; ASSERT(y);
01739                   DataT* u = m_channel[1]; ASSERT(u);
01740                   DataT* v = m_channel[2]; ASSERT(v);
01741
01742                   UINT16 *buff16 = (UINT16 *)buff;
01743                   UINT16 rgb, b, g, r;
01744                   const int pitch16 = pitch/2;
```

```
01745
01746                     for (UINT32 h=0; h < m_header.height; h++) {
01747                             if (cb) {
01748                                     if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
01749                                     percent += dP;
01750                             }
01751                             for (UINT32 w=0; w < m_header.width; w++) {
01752                                     rgb = buff16[w];
01753                                     r = (rgb & 0xF800) >> 10;       //
highest 5 bits
01754                                     g = (rgb & 0x07E0) >> 5;        //
middle 6 bits
01755                                     b = (rgb & 0x001F) << 1;        //
lowest 5 bits
01756                                     // Yuv
01757                                     y[yPos] = ((b + (g << 1) + r) >> 2) -
YUVoffset6;
01758                                     u[yPos] = r - g;
01759                                     v[yPos] = b - g;
01760                                     yPos++;
01761                             }
01762
01763                             buff16 += pitch16;
01764                     }
01765             }
01766             break;
01767         default:
01768             ASSERT(false);
01769         }
01770 }
```

bool CPGFImage::ROIisSupported () const [inline]

Return true if the pgf image supports Region Of Interest (ROI).

Returns

true if the pgf image supports ROI.

Definition at line **466** of file **PGFimage.h**.

```
00466 { return (m_preHeader.version & PGFROI) == PGFROI; }
```

void CPGFImage::SetChannel (DataT * *channel*, int *c* = 0) [inline]

Set internal PGF image buffer channel.

Parameters

channel	A YUV data channel
c	A channel index

Definition at line **272** of file **PGFimage.h**.

```
00272 { ASSERT(c >= 0 && c < MaxChannels); m_channel[c] = channel; }
```

void CPGFImage::SetColorTable (UINT32 *iFirstColor*, UINT32 *nColors*, const RGBQUAD * *prgbColors*)

Sets the red, green, blue (RGB) color values for a range of entries in the palette (clut). It might throw an **IOException**.

Parameters

iFirstColor	The color table index of the first entry to set.
nColors	The number of color table entries to set.
prgbColors	A pointer to the array of RGBQUAD structures to set the color table entries.

Definition at line **1364** of file **PGFimage.cpp**.

```
01364
{
01365         if (iFirstColor + nColors > ColorTableLen)
ReturnWithError(ColorTableError);
01366
01367         for (UINT32 i=iFirstColor, j=0; j < nColors; i++, j++) {
01368                 m_postHeader.clut[i] = prgbColors[j];
```

```
01369           }
01370 }
```

void CPGFImage::SetHeader (const PGFHeader & *header*, BYTE *flags* = 0, const UINT8 * *userData* = 0, UINT32 *userDataLength* = 0)

Set PGF header and user data. Precondition: The PGF image has been never opened with Open(...). It might throw an **IOException**.

Parameters

header	A valid and already filled in PGF header structure
flags	A combination of additional version flags. In case you use level-wise encoding then set flag = PGFROI.
userData	A user-defined memory block containing any kind of cached metadata.
userDataLength	The size of user-defined memory block in bytes

Definition at line **894** of file **PGFimage.cpp**.

```
00894
{
00895           ASSERT(!m_decoder);      // current image must be closed
00896           ASSERT(header.quality <= MaxQuality);
00897           ASSERT(userDataLength <= MaxUserDataSize);
00898
00899           // init state
00900 #ifdef __PGFROISUPPORT__
00901           m_streamReinitialized = false;
00902 #endif
00903
00904           // init preHeader
00905           memcpy(m_preHeader.magic, PGFMagic, 3);
00906           m_preHeader.version = PGFVersion | flags;
00907           m_preHeader.hSize = HeaderSize;
00908
00909           // copy header
00910           memcpy(&m_header, &header, HeaderSize);
00911
00912           // check quality
00913           if (m_header.quality > MaxQuality) m_header.quality = MaxQuality;
00914
00915           // complete header
00916           CompleteHeader();
00917
00918           // check and set number of levels
00919           ComputeLevels();
00920
00921           // check for downsample
00922           if (m_header.quality > DownsampleThreshold &&  (m_header.mode ==
ImageModeRGBColor ||
00923
m_header.mode == ImageModeRGBA ||
00924
m_header.mode == ImageModeRGB48 ||
00925
m_header.mode == ImageModeCMYKColor ||
00926
m_header.mode == ImageModeCMYK64 ||
00927
m_header.mode == ImageModeLabColor ||
00928
m_header.mode == ImageModeLab48)) {
00929                   m_downsample = true;
00930                   m_quant = m_header.quality - 1;
00931           } else {
00932                   m_downsample = false;
00933                   m_quant = m_header.quality;
00934           }
00935
00936           // update header size and copy user data
00937           if (m_header.mode == ImageModeIndexedColor) {
00938                   // update header size
00939                   m_preHeader.hSize += ColorTableSize;
00940           }
00941           if (userDataLength && userData) {
```

```
00942                   if (userDataLength > MaxUserDataSize) userDataLength =
MaxUserDataSize;
00943                   m_postHeader.userData = new(std::nothrow)
UINT8[userDataLength];
00944                   if (!m_postHeader.userData)
ReturnWithError(InsufficientMemory);
00945                   m_postHeader.userDataLen = m_postHeader.cachedUserDataLen =
userDataLength;
00946                   memcpy(m_postHeader.userData, userData, userDataLength);
00947                   // update header size
00948                   m_preHeader.hSize += userDataLength;
00949           }
00950
00951           // allocate channels
00952           for (int i=0; i < m_header.channels; i++) {
00953                   // set current width and height
00954                   m_width[i] = m_header.width;
00955                   m_height[i] = m_header.height;
00956
00957                   // allocate channels
00958                   ASSERT(!m_channel[i]);
00959                   m_channel[i] = new(std::nothrow)
DataT[m_header.width*m_header.height];
00960                   if (!m_channel[i]) {
00961                           if (i) i--;
00962                           while(i) {
00963                                   delete[] m_channel[i]; m_channel[i] = 0;
00964                                   i--;
00965                           }
00966                           ReturnWithError(InsufficientMemory);
00967                   }
00968           }
00969 }
```

void CPGFImage::SetMaxValue (UINT32 *maxValue*)

Set maximum intensity value for image modes with more than eight bits per channel. Call this method after SetHeader, but before ImportBitmap.

Parameters

maxValue	The maximum intensity value.

Definition at line **738** of file **PGFimage.cpp**.

```
00738                                   {
00739           const BYTE bpc = m_header.bpp/m_header.channels;
00740           BYTE pot = 0;
00741
00742           while(maxValue > 0) {
00743                   pot++;
00744                   maxValue >>= 1;
00745           }
00746           // store bits per channel
00747           if (pot > bpc) pot = bpc;
00748           if (pot > 31) pot = 31;
00749           m_header.usedBitsPerChannel = pot;
00750 }
```

void CPGFImage::SetProgressMode (ProgressMode *pm*)[inline]

Set progress mode used in Read and Write. Default mode is PM_Relative. This method must be called before **Open()** or **SetHeader()**. PM_Relative: 100% = level difference between current level and target level of Read/Write PM_Absolute: 100% = number of levels

Definition at line **296** of file **PGFimage.h**.

```
00296 { m_progressMode = pm; }
```

void CPGFImage::SetRefreshCallback (RefreshCB *callback*, void * *arg*)[inline]

Set refresh callback procedure and its parameter. The refresh callback is called during Read(...) after each level read.

Parameters

callback	A refresh callback procedure
arg	A parameter of the refresh callback procedure

Definition at line **303** of file **PGFimage.h**.

```
00303 { m_cb = callback; m_cbArg = arg; }
```

void CPGFImage::SetROI (PGFRect *rect*) [private]

UINT32 CPGFImage::UpdatePostHeaderSize () [private]

Definition at line **1124** of file **PGFimage.cpp**.

```
01124                                    {
01125         ASSERT(m_encoder);
01126 
01127         INT64 offset = m_encoder->ComputeOffset(); ASSERT(offset >= 0);
01128 
01129         if (offset > 0) {
01130                 // update post-header size and rewrite pre-header
01131                 m_preHeader.hSize += (UINT32)offset;
01132                 m_encoder->UpdatePostHeaderSize(m_preHeader);
01133         }
01134 
01135         // write dummy levelLength into stream
01136         return m_encoder->WriteLevelLength(m_levelLength);
01137 }
```

BYTE CPGFImage::UsedBitsPerChannel () const

Returns number of used bits per input/output image channel. Precondition: header must be initialized.

Returns

number of used bits per input/output image channel.

Definition at line **756** of file **PGFimage.cpp**.

```
00756                                      {
00757         const BYTE bpc = m_header.bpp/m_header.channels;
00758 
00759         if (bpc > 8) {
00760                 return m_header.usedBitsPerChannel;
00761         } else {
00762                 return bpc;
00763         }
00764 }
```

BYTE CPGFImage::Version () const [inline]

Returns the used codec major version of a pgf image

Returns

PGF codec major version of this image

Definition at line **484** of file **PGFimage.h**.

```
00484 { BYTE ver = CodecMajorVersion(m_preHeader.version); return (ver <= 7) ? ver :
(BYTE)m_header.version.major; }
```

UINT32 CPGFImage::Width (int *level* = 0) const [inline]

Return image width of channel 0 at given level in pixels. The returned width is independent of any Read-operations and ROI.

Parameters

level	A level

Returns

Image level width in pixels

Definition at line **413** of file **PGFimage.h**.

```
00413 { ASSERT(level >= 0); return LevelSizeL(m_header.width, level); }
```

void CPGFImage::Write (CPGFStream * *stream*, UINT32 * *nWrittenBytes* = `nullptr`, CallbackPtr *cb* = `nullptr`, void * *data* = `nullptr`)

Encode and write an entire PGF image (header and image) at current stream position. A PGF image is structered in levels, numbered between 0 and **Levels()** - 1. Each level can be seen as a single image, containing the same content as all other levels, but in a different size (width, height). The image size at level i is double the size (width, height) of the image at level i+1. The image at level 0 contains the original size. Precondition: the PGF image contains a valid header (see also SetHeader(...)). It might throw an **IOException**.

Parameters

stream	A PGF stream
nWrittenBytes	[in-out] The number of bytes written into stream are added to the input value.
cb	A pointer to a callback procedure. The procedure is called after writing a single level. If cb returns true, then it stops proceeding.
data	Data Pointer to C++ class container to host callback procedure.

Definition at line **1221** of file **PGFimage.cpp**.

```
01221
{
01222         ASSERT(stream);
01223         ASSERT(m_preHeader.hSize);
01224
01225         // create wavelet transform channels and encoder
01226         UINT32 nBytes = WriteHeader(stream);
01227
01228         // write image
01229         nBytes += WriteImage(stream, cb, data);
01230
01231         // return written bytes
01232         if (nWrittenBytes) *nWrittenBytes += nBytes;
01233 }
```

UINT32 CPGFImage::Write (int *level*, CallbackPtr *cb* = `nullptr`, void * *data* = `nullptr`)

Encode and write down to given level at current stream position. A PGF image is structered in levels, numbered between 0 and **Levels()** - 1. Each level can be seen as a single image, containing the same content as all other levels, but in a different size (width, height). The image size at level i is double the size (width, height) of the image at level i+1. The image at level 0 contains the original size. Preconditions: the PGF image contains a valid header (see also SetHeader(...)) and **WriteHeader()** has been called before. **Levels()** > 0. The ROI encoding scheme must be used (see also SetHeader(...)). It might throw an **IOException**.

Parameters

level	[0, nLevels) The image level of the resulting image in the internal image buffer.
cb	A pointer to a callback procedure. The procedure is called after writing a single level. If cb returns true, then it stops proceeding.
data	Data Pointer to C++ class container to host callback procedure.

Returns

The number of bytes written into stream.

UINT32 CPGFImage::WriteHeader (CPGFStream * *stream*)

Create wavelet transform channels and encoder. Write header at current stream position. Call this method before your first call of Write(int level) or **WriteImage()**, but after

SetHeader(). This method is called inside of Write(stream, ...). It might throw an **IOException**.

Parameters

stream	A PGF stream

Returns

The number of bytes written into stream.

Create wavelet transform channels and encoder. Write header at current stream position. Performs forward FWT. Call this method before your first call of Write(int level) or **WriteImage()**, but after **SetHeader()**. This method is called inside of Write(stream, ...). It might throw an **IOException**.

Parameters

stream	A PGF stream

Returns

The number of bytes written into stream.

Definition at line **979** of file **PGFimage.cpp**.

```
00979                                                        {
00980           ASSERT(m_header.nLevels <= MaxLevel);
00981           ASSERT(m_header.quality <= MaxQuality); // quality is already
initialized
00982
00983           if (m_header.nLevels > 0) {
00984                   volatile OSError error = NoError; // volatile prevents
optimizations
00985                   // create new wt channels
00986 #ifdef LIBPGF_USE_OPENMP
00987                   #pragma omp parallel for default(shared)
00988 #endif
00989                   for (int i=0; i < m_header.channels; i++) {
00990                           DataT *temp = nullptr;
00991                           if (error == NoError) {
00992                                   if (m_wtChannel[i]) {
00993                                           ASSERT(m_channel[i]);
00994                                           // copy m_channel to temp
00995                                           int size = m_height[i]*m_width[i];
00996                                           temp = new(std::nothrow) DataT[size];
00997                                           if (temp) {
00998                                                   memcpy(temp, m_channel[i],
size*DataTSize);
00999                                                   delete m_wtChannel[i];  //
also deletes m_channel
01000                                                   m_channel[i] = nullptr;
01001                                           } else {
01002                                                   error = InsufficientMemory;
01003                                           }
01004                                   }
01005                                   if (error == NoError) {
01006                                           if (temp) {
01007                                                   ASSERT(!m_channel[i]);
01008                                                   m_channel[i] = temp;
01009                                           }
01010                                           m_wtChannel[i] = new
CWaveletTransform(m_width[i], m_height[i], m_header.nLevels, m_channel[i]);
01011                                           if (m_wtChannel[i]) {
01012                                           #ifdef __PGFROISUPPORT__
01013
m_wtChannel[i]->SetROI(PGFRect(0, 0, m_width[i], m_height[i]));
01014                                           #endif
01015
01016                                                   // wavelet subband
decomposition
01017                                                   for (int l=0; error == NoError
&& l < m_header.nLevels; l++) {
01018                                                           OSError err =
m_wtChannel[i]->ForwardTransform(l, m_quant);
01019                                                           if (err != NoError)
error = err;
01020                                                   }
```

```
01021                                   } else {
01022                                           delete[] m_channel[i];
01023                                           error = InsufficientMemory;
01024                                   }
01025                           }
01026                   }
01027           }
01028           if (error != NoError) {
01029                   // free already allocated memory
01030                   for (int i=0; i < m_header.channels; i++) {
01031                           delete m_wtChannel[i];
01032                   }
01033                   ReturnWithError(error);
01034           }
01035
01036           m_currentLevel = m_header.nLevels;
01037
01038           // create encoder, write headers and user data, but not
level-length area
01039           m_encoder = new CEncoder(stream, m_preHeader, m_header,
m_postHeader, m_userDataPos, m_useOMPinEncoder);
01040           if (m_favorSpeedOverSize) m_encoder->FavorSpeedOverSize();
01041
01042   #ifdef __PGFROISUPPORT__
01043           if (ROIisSupported()) {
01044                   // new encoding scheme supporting ROI
01045                   m_encoder->SetROI();
01046           }
01047   #endif
01048
01049   } else {
01050           // very small image: we don't use DWT and encoding
01051
01052           // create encoder, write headers and user data, but not
level-length area
01053           m_encoder = new CEncoder(stream, m_preHeader, m_header,
m_postHeader, m_userDataPos, m_useOMPinEncoder);
01054   }
01055
01056   INT64 nBytes = m_encoder->ComputeHeaderLength();
01057   return (nBytes > 0) ? (UINT32)nBytes : 0;
01058 }
```

UINT32 CPGFImage::WriteImage (CPGFStream * *stream*, CallbackPtr *cb* = `nullptr`, void * *data* = `nullptr`)

Encode and write an image at current stream position. Call this method after **WriteHeader()**. In case you want to write uncached metadata, then do that after **WriteHeader()** and before **WriteImage()**. This method is called inside of Write(stream, ...). It might throw an **IOException**.

Parameters

stream	A PGF stream
cb	A pointer to a callback procedure. The procedure is called after writing a single level. If cb returns true, then it stops proceeding.
data	Data Pointer to C++ class container to host callback procedure.

Returns

The number of bytes written into stream.

Definition at line **1150** of file **PGFimage.cpp**.

```
01150
{
01151   ASSERT(stream);
01152   ASSERT(m_preHeader.hSize);
01153
01154   int levels = m_header.nLevels;
01155   double percent = pow(0.25, levels);
01156
01157   // update post-header size, rewrite pre-header, and write dummy
levelLength
01158   UINT32 nWrittenBytes = UpdatePostHeaderSize();
```

```
01159
01160          if (levels == 0) {
01161                  // for very small images: write channels uncoded
01162                  for (int c=0; c < m_header.channels; c++) {
01163                          const UINT32 size = m_width[c]*m_height[c];
01164
01165                          // write channel data into stream
01166                          for (UINT32 i=0; i < size; i++) {
01167                                  int count = DataTSize;
01168                                  stream->Write(&count, &m_channel[c][i]);
01169                          }
01170                  }
01171
01172                  // now update progress
01173                  if (cb) {
01174                          if ((*cb)(1, true, data))
ReturnWithError(EscapePressed);
01175                  }
01176
01177          } else {
01178                  // encode quantized wavelet coefficients and write to PGF file
01179                  // encode subbands, higher levels first
01180                  // color channels are interleaved
01181
01182                  // encode all levels
01183                  for (m_currentLevel = levels; m_currentLevel > 0; ) {
01184                          WriteLevel(); // decrements m_currentLevel
01185
01186                          // now update progress
01187                          if (cb) {
01188                                  percent *= 4;
01189                                  if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
01190                          }
01191                  }
01192
01193                  // flush encoder and write level lengths
01194                  m_encoder->Flush();
01195          }
01196
01197          // update level lengths
01198          nWrittenBytes += m_encoder->UpdateLevelLength(); // return written
image bytes
01199
01200          // delete encoder
01201          delete m_encoder; m_encoder = nullptr;
01202
01203          ASSERT(!m_encoder);
01204
01205          return nWrittenBytes;
01206 }
```

void CPGFImage::WriteLevel () [private]

Definition at line **1068** of file **PGFimage.cpp**.

```
01068                          {
01069          ASSERT(m_encoder);
01070          ASSERT(m_currentLevel > 0);
01071          ASSERT(m_header.nLevels > 0);
01072
01073 #ifdef __PGFROISUPPORT__
01074          if (ROIisSupported()) {
01075                  const int lastChannel = m_header.channels - 1;
01076
01077                  for (int i=0; i < m_header.channels; i++) {
01078                          // get number of tiles and tile indices
01079                          const UINT32 nTiles =
m_wtChannel[i]->GetNofTiles(m_currentLevel);
01080                          const UINT32 lastTile = nTiles - 1;
01081
01082                          if (m_currentLevel == m_header.nLevels) {
01083                                  // last level also has LL band
01084                                  ASSERT(nTiles == 1);
```

```
01085                                   m_wtChannel[i]->GetSubband(m_currentLevel,
LL)->ExtractTile(*m_encoder);
01086                                   m_encoder->EncodeTileBuffer(); // encode macro
block with tile-end = true
01087                           }
01088                           for (UINT32 tileY=0; tileY < nTiles; tileY++) {
01089                                   for (UINT32 tileX=0; tileX < nTiles; tileX++)
{
01090                                           // extract tile to macro block and
encode already filled macro blocks with tile-end = false
01091
m_wtChannel[i]->GetSubband(m_currentLevel, HL)->ExtractTile(*m_encoder, true, tileX,
tileY);
01092
m_wtChannel[i]->GetSubband(m_currentLevel, LH)->ExtractTile(*m_encoder, true, tileX,
tileY);
01093
m_wtChannel[i]->GetSubband(m_currentLevel, HH)->ExtractTile(*m_encoder, true, tileX,
tileY);
01094                                           if (i == lastChannel && tileY ==
lastTile && tileX == lastTile) {
01095                                                   // all necessary data are
buffered. next call of EncodeTileBuffer will write the last piece of data of the current
level.
01096
m_encoder->SetEncodedLevel(--m_currentLevel);
01097                                           }
01098                                           m_encoder->EncodeTileBuffer(); //
encode last macro block with tile-end = true
01099                                   }
01100                           }
01101                   }
01102           } else
01103 #endif
01104           {
01105                   for (int i=0; i < m_header.channels; i++) {
01106                           ASSERT(m_wtChannel[i]);
01107                           if (m_currentLevel == m_header.nLevels) {
01108                                   // last level also has LL band
01109                                   m_wtChannel[i]->GetSubband(m_currentLevel,
LL)->ExtractTile(*m_encoder);
01110                           }
01111                           //encoder.EncodeInterleaved(m_wtChannel[i],
m_currentLevel, m_quant); // until version 4
01112                           m_wtChannel[i]->GetSubband(m_currentLevel,
HL)->ExtractTile(*m_encoder); // since version 5
01113                           m_wtChannel[i]->GetSubband(m_currentLevel,
LH)->ExtractTile(*m_encoder); // since version 5
01114                           m_wtChannel[i]->GetSubband(m_currentLevel,
HH)->ExtractTile(*m_encoder);
01115                   }
01116
01117                   // all necessary data are buffered. next call of EncodeBuffer
will write the last piece of data of the current level.
01118                   m_encoder->SetEncodedLevel(--m_currentLevel);
01119           }
01120 }
```

Member Data Documentation

RefreshCB CPGFImage::m_cb [private]

pointer to refresh callback procedure

Definition at line **545** of file **PGFimage.h**.

void* CPGFImage::m_cbArg [private]

refresh callback argument

Definition at line **546** of file **PGFimage.h**.

DataT* CPGFImage::m_channel[MaxChannels] `[protected]`

untransformed channels in YUV format

Definition at line **522** of file **PGFimage.h**.

int CPGFImage::m_currentLevel `[protected]`

transform level of current image

Definition at line **532** of file **PGFimage.h**.

CDecoder* CPGFImage::m_decoder `[protected]`

PGF decoder.

Definition at line **523** of file **PGFimage.h**.

bool CPGFImage::m_downsample `[protected]`

chrominance channels are downsampled

Definition at line **535** of file **PGFimage.h**.

CEncoder* CPGFImage::m_encoder `[protected]`

PGF encoder.

Definition at line **524** of file **PGFimage.h**.

bool CPGFImage::m_favorSpeedOverSize `[protected]`

favor encoding speed over compression ratio

Definition at line **536** of file **PGFimage.h**.

PGFHeader CPGFImage::m_header `[protected]`

PGF file header.

Definition at line **529** of file **PGFimage.h**.

UINT32 CPGFImage::m_height[MaxChannels] `[protected]`

height of each channel at current level

Definition at line **527** of file **PGFimage.h**.

UINT32* CPGFImage::m_levelLength `[protected]`

length of each level in bytes; first level starts immediately after this array

Definition at line **525** of file **PGFimage.h**.

double CPGFImage::m_percent `[private]`

progress [0..1]

Definition at line **547** of file **PGFimage.h**.

PGFPostHeader CPGFImage::m_postHeader `[protected]`

PGF post-header.

Definition at line **530** of file **PGFimage.h**.

PGFPreHeader CPGFImage::m_preHeader `[protected]`

PGF pre-header.

Definition at line **528** of file **PGFimage.h**.

ProgressMode CPGFImage::m_progressMode `[private]`

progress mode used in Read and Write; PM_Relative is default mode

Definition at line **548** of file **PGFimage.h**.

BYTE CPGFImage::m_quant `[protected]`

quantization parameter

Definition at line **534** of file **PGFimage.h**.

PGFRect CPGFImage::m_roi `[protected]`

region of interest

Definition at line **541** of file **PGFimage.h**.

bool CPGFImage::m_streamReinitialized `[protected]`

stream has been reinitialized

Definition at line **540** of file **PGFimage.h**.

bool CPGFImage::m_useOMPinDecoder `[protected]`

use Open MP in decoder

Definition at line **538** of file **PGFimage.h**.

bool CPGFImage::m_useOMPinEncoder `[protected]`

use Open MP in encoder

Definition at line **537** of file **PGFimage.h**.

UINT32 CPGFImage::m_userDataPolicy `[protected]`

user data (metadata) policy during open

Definition at line **533** of file **PGFimage.h**.

UINT64 CPGFImage::m_userDataPos `[protected]`

stream position of user data

Definition at line **531** of file **PGFimage.h**.

UINT32 CPGFImage::m_width[MaxChannels] `[protected]`

width of each channel at current level

Definition at line **526** of file **PGFimage.h**.

CWaveletTransform* CPGFImage::m_wtChannel[MaxChannels] `[protected]`

wavelet transformed color channels

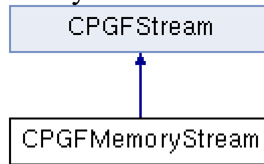
Definition at line **521** of file **PGFimage.h**.

The documentation for this class was generated from the following files:

- **PGFimage.h**
- **PGFimage.cpp**

CPGFMemoryStream Class Reference

Memory stream class.
`#include <PGFstream.h>`
Inheritance diagram for CPGFMemoryStream:



Public Member Functions

- **CPGFMemoryStream** (size_t size)
- **CPGFMemoryStream** (UINT8 *pBuffer, size_t size)
- void **Reinitialize** (UINT8 *pBuffer, size_t size)
- virtual **~CPGFMemoryStream** ()
- virtual void **Write** (int *count, void *buffer)
- virtual void **Read** (int *count, void *buffer)
- virtual void **SetPos** (short posMode, INT64 posOff)
- virtual UINT64 **GetPos** () const
- virtual bool **IsValid** () const
- size_t **GetSize** () const
- const UINT8 * **GetBuffer** () const
- UINT8 * **GetBuffer** ()
- UINT64 **GetEOS** () const
- void **SetEOS** (UINT64 length)

Protected Attributes

- UINT8 * **m_buffer**
- UINT8 * **m_pos**
  buffer start address and current buffer address
- UINT8 * **m_eos**
  end of stream (first address beyond written area)
- size_t **m_size**
  buffer size
- bool **m_allocated**
  indicates a new allocated buffer

Detailed Description

Memory stream class.

A PGF stream subclass for internal memory.

Author

C. Stamm

Definition at line **106** of file **PGFstream.h**.

Constructor & Destructor Documentation

CPGFMemoryStream::CPGFMemoryStream (size_t *size*)

Constructor

Parameters

size	Size of new allocated memory buffer

Allocate memory block of given size

Parameters

size	Memory size

Definition at line **78** of file **PGFstream.cpp**.

```
00079 : m_size(size)
00080 , m_allocated(true) {
00081         m_buffer = m_pos = m_eos = new(std::nothrow) UINT8[m_size];
00082         if (!m_buffer) ReturnWithError(InsufficientMemory);
00083 }
```

CPGFMemoryStream::CPGFMemoryStream (UINT8 * *pBuffer*, size_t *size*)

Constructor. Use already allocated memory of given size

Parameters

pBuffer	Memory location
size	Memory size

Use already allocated memory of given size

Parameters

pBuffer	Memory location
size	Memory size

Definition at line **89** of file **PGFstream.cpp**.

```
00090 : m_buffer(pBuffer)
00091 , m_pos(pBuffer)
00092 , m_eos(pBuffer + size)
00093 , m_size(size)
00094 , m_allocated(false) {
00095         ASSERT(IsValid());
00096 }
```

virtual CPGFMemoryStream::~CPGFMemoryStream () [inline], [virtual]

Definition at line **128** of file **PGFstream.h**.

```
00128                                         {
00129                 m_pos = 0;
00130                 if (m_allocated) {
00131                         // the memory buffer has been allocated inside of
CPMFmemoryStream constructor
00132                         delete[] m_buffer; m_buffer = 0;
00133                 }
00134         }
```

Member Function Documentation

UINT8 * CPGFMemoryStream::GetBuffer () [inline]

Returns

Memory buffer

Definition at line **147** of file **PGFstream.h**.

```
00147 { return m_buffer; }
```

const UINT8 * CPGFMemoryStream::GetBuffer () const[inline]

Returns

Memory buffer

Definition at line **145** of file **PGFstream.h**.

```
00145 { return m_buffer; }
```

UINT64 CPGFMemoryStream::GetEOS () const[inline]

Returns

relative position of end of stream (= stream length)

Definition at line **149** of file **PGFstream.h**.

```
00149 { ASSERT(IsValid()); return m_eos - m_buffer; }
```

virtual UINT64 CPGFMemoryStream::GetPos () const[inline], [virtual]

Get current stream position.

Returns

Current stream position

Implements **CPGFStream** (*p.109*).

Definition at line **139** of file **PGFstream.h**.

```
00139 { ASSERT(IsValid()); return m_pos - m_buffer; }
```

size_t CPGFMemoryStream::GetSize () const[inline]

Returns

Memory size

Definition at line **143** of file **PGFstream.h**.

```
00143 { return m_size; }
```

virtual bool CPGFMemoryStream::IsValid () const[inline], [virtual]

Check stream validity.

Returns

True if stream and current position is valid

Implements **CPGFStream** (*p.109*).

Definition at line **140** of file **PGFstream.h**.

```
00140 { return m_buffer != 0; }
```

void CPGFMemoryStream::Read (int * *count*, void * *buffer*)[virtual]

Read some bytes from this stream and stores them into a buffer.

Parameters

count	A pointer to a value containing the number of bytes should be read. After this call it contains the number of read bytes.

buffer	A memory buffer

Implements **CPGFStream** (*p.109*).

Definition at line **148** of file **PGFstream.cpp**.

```
00148                                                              {
00149         ASSERT(IsValid());
00150         ASSERT(count);
00151         ASSERT(buffPtr);
00152         ASSERT(m_buffer + m_size >= m_eos);
00153         ASSERT(m_pos <= m_eos);
00154
00155         if (m_pos + *count <= m_eos) {
00156                 memcpy(buffPtr, m_pos, *count);
00157                 m_pos += *count;
00158         } else {
00159                 // end of memory block reached -> read only until end
00160                 *count = (int)__max(0, m_eos - m_pos);
00161                 memcpy(buffPtr, m_pos, *count);
00162                 m_pos += *count;
00163         }
00164         ASSERT(m_pos <= m_eos);
00165 }
```

void CPGFMemoryStream::Reinitialize (UINT8 * *pBuffer*, size_t *size*)

Use already allocated memory of given size

Parameters

pBuffer	Memory location
size	Memory size

Definition at line **102** of file **PGFstream.cpp**.

```
00102                                                                        {
00103         if (!m_allocated) {
00104                 m_buffer = m_pos = pBuffer;
00105                 m_size = size;
00106                 m_eos = m_buffer + size;
00107         }
00108 }
```

void CPGFMemoryStream::SetEOS (UINT64 *length*)`[inline]`

Parameters

length	Stream length (= relative position of end of stream)

Definition at line **151** of file **PGFstream.h**.

```
00151 { ASSERT(IsValid()); m_eos = m_buffer + length; }
```

void CPGFMemoryStream::SetPos (short *posMode*, INT64 *posOff*)`[virtual]`

Set stream position either absolute or relative.

Parameters

posMode	A position mode (FSFromStart, FSFromCurrent, FSFromEnd)
posOff	A new stream position (absolute positioning) or a position offset (relative positioning)

Implements **CPGFStream** (*p.109*).

Definition at line **168** of file **PGFstream.cpp**.

```
00168                                                                 {
00169         ASSERT(IsValid());
00170         switch(posMode) {
00171         case FSFromStart:
00172                 m_pos = m_buffer + posOff;
00173                 break;
00174         case FSFromCurrent:
00175                 m_pos += posOff;
00176                 break;
00177         case FSFromEnd:
```

```
00178                   m_pos = m_eos + posOff;
00179                   break;
00180           default:
00181                   ASSERT(false);
00182           }
00183           if (m_pos > m_eos)
00184                   ReturnWithError(InvalidStreamPos);
00185 }
```

void CPGFMemoryStream::Write (int * *count*, void * *buffer*) [virtual]

Write some bytes out of a buffer into this stream.

Parameters

count	A pointer to a value containing the number of bytes should be written. After this call it contains the number of written bytes.
buffer	A memory buffer

Implements **CPGFStream** (*p.109*).

Definition at line **111** of file **PGFstream.cpp**.

```
00111                                                       {
00112           ASSERT(count);
00113           ASSERT(buffPtr);
00114           ASSERT(IsValid());
00115           const size_t deltaSize = 0x4000 + *count;
00116
00117           if (m_pos + *count <= m_buffer + m_size) {
00118                   memcpy(m_pos, buffPtr, *count);
00119                   m_pos += *count;
00120                   if (m_pos > m_eos) m_eos = m_pos;
00121           } else if (m_allocated) {
00122                   // memory block is too small -> reallocate a deltaSize larger
block
00123                   size_t offset = m_pos - m_buffer;
00124                   UINT8 *buf_tmp = (UINT8 *)realloc(m_buffer, m_size + deltaSize);
00125                   if (!buf_tmp) {
00126                           delete[] m_buffer;
00127                           m_buffer = 0;
00128                           ReturnWithError(InsufficientMemory);
00129                   } else {
00130                           m_buffer = buf_tmp;
00131                   }
00132                   m_size += deltaSize;
00133
00134                   // reposition m_pos
00135                   m_pos = m_buffer + offset;
00136
00137                   // write block
00138                   memcpy(m_pos, buffPtr, *count);
00139                   m_pos += *count;
00140                   if (m_pos > m_eos) m_eos = m_pos;
00141           } else {
00142                   ReturnWithError(InsufficientMemory);
00143           }
00144           ASSERT(m_pos <= m_eos);
00145 }
```

Member Data Documentation

bool CPGFMemoryStream::m_allocated [protected]

indicates a new allocated buffer

Definition at line **111** of file **PGFstream.h**.

UINT8* CPGFMemoryStream::m_buffer [protected]

Definition at line **108** of file **PGFstream.h**.

UINT8* CPGFMemoryStream::m_eos [protected]

end of stream (first address beyond written area)

Definition at line **109** of file **PGFstream.h**.

UINT8 * CPGFMemoryStream::m_pos [protected]

buffer start address and current buffer address

Definition at line **108** of file **PGFstream.h**.

size_t CPGFMemoryStream::m_size [protected]

buffer size

Definition at line **110** of file **PGFstream.h**.

The documentation for this class was generated from the following files:

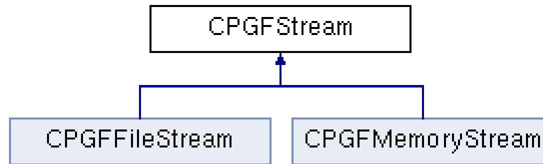
- **PGFstream.h**
- **PGFstream.cpp**

CPGFStream Class Reference

Abstract stream base class.

`#include <PGFstream.h>`

Inheritance diagram for CPGFStream:



Public Member Functions

- **CPGFStream** ()
  Standard constructor.
- virtual **~CPGFStream** ()
  Standard destructor.
- virtual void **Write** (int *count, void *buffer)=0
- virtual void **Read** (int *count, void *buffer)=0
- virtual void **SetPos** (short posMode, INT64 posOff)=0
- virtual UINT64 **GetPos** () const =0
- virtual bool **IsValid** () const =0

Detailed Description

Abstract stream base class.

Abstract stream base class.

Author

C. Stamm

Definition at line **39** of file **PGFstream.h**.

Constructor & Destructor Documentation

CPGFStream::CPGFStream () `[inline]`

Standard constructor.

Definition at line **43** of file **PGFstream.h**.

```
00043 {}
```

virtual CPGFStream::~CPGFStream () `[inline]`, `[virtual]`

Standard destructor.

Definition at line **47** of file **PGFstream.h**.

```
00047 {}
```

Member Function Documentation

virtual UINT64 CPGFStream::GetPos () const`[pure virtual]`

Get current stream position.

Returns

Current stream position

Implemented in **CPGFFileStream** (*p.46*), and **CPGFMemoryStream** (*p.104*).

virtual bool CPGFStream::IsValid () const`[pure virtual]`

Check stream validity.

Returns

True if stream and current position is valid

Implemented in **CPGFFileStream** (*p.46*), and **CPGFMemoryStream** (*p.104*).

virtual void CPGFStream::Read (int * *count*, void * *buffer*)`[pure virtual]`

Read some bytes from this stream and stores them into a buffer.

Parameters

count	A pointer to a value containing the number of bytes should be read. After this call it contains the number of read bytes.
buffer	A memory buffer

Implemented in **CPGFFileStream** (*p.46*), and **CPGFMemoryStream** (*p.104*).

virtual void CPGFStream::SetPos (short *posMode*, INT64 *posOff*)`[pure virtual]`

Set stream position either absolute or relative.

Parameters

posMode	A position mode (FSFromStart, FSFromCurrent, FSFromEnd)
posOff	A new stream position (absolute positioning) or a position offset (relative positioning)

Implemented in **CPGFFileStream** (*p.47*), and **CPGFMemoryStream** (*p.105*).

virtual void CPGFStream::Write (int * *count*, void * *buffer*)`[pure virtual]`

Write some bytes out of a buffer into this stream.

Parameters

count	A pointer to a value containing the number of bytes should be written. After this call it contains the number of written bytes.
buffer	A memory buffer

Implemented in **CPGFFileStream** (*p.47*), and **CPGFMemoryStream** (*p.106*).

The documentation for this class was generated from the following file:

- **PGFstream.h**

CSubband Class Reference

Wavelet channel class.

`#include <Subband.h>`

Public Member Functions

- **CSubband** ()
  Standard constructor.

- **~CSubband** ()
  Destructor.

- bool **AllocMemory** ()
- void **FreeMemory** ()
  Delete the memory buffer of this subband.

- void **ExtractTile** (**CEncoder** &encoder, bool tile=false, UINT32 tileX=0, UINT32 tileY=0)
- void **PlaceTile** (**CDecoder** &decoder, int quantParam, bool tile=false, UINT32 tileX=0, UINT32 tileY=0)
- void **Quantize** (int quantParam)
- void **Dequantize** (int quantParam)
- void **SetData** (UINT32 pos, **DataT** v)
- **DataT** * **GetBuffer** ()
- **DataT** **GetData** (UINT32 pos) const
- int **GetLevel** () const
- int **GetHeight** () const
- int **GetWidth** () const
- **Orientation** **GetOrientation** () const

Private Member Functions

- void **Initialize** (UINT32 width, UINT32 height, int level, **Orientation** orient)
- void **WriteBuffer** (**DataT** val)
- void **SetBuffer** (**DataT** *b)
- **DataT** **ReadBuffer** ()
- UINT32 **GetBuffPos** () const
- void **InitBuffPos** ()

Private Attributes

- UINT32 **m_width**
  width in pixels

- UINT32 **m_height**
  height in pixels

- UINT32 **m_size**
  size of data buffer m_data

- int **m_level**
  recursion level

- **Orientation m_orientation**
  0=LL, 1=HL, 2=LH, 3=HH L=lowpass filtered, H=highpass filterd

- UINT32 **m_dataPos**
  current position in m_data

- **DataT * m_data**
  buffer

Friends

- class **CWaveletTransform**
- class **CRoiIndices**

Detailed Description

Wavelet channel class.

PGF wavelet channel subband class.

Author

C. Stamm, R. Spuler

Definition at line **42** of file **Subband.h**.

Constructor & Destructor Documentation

CSubband::CSubband ()

Standard constructor.

Definition at line **35** of file **Subband.cpp**.

```
00036 : m_width(0)
00037 , m_height(0)
00038 , m_size(0)
00039 , m_level(0)
00040 , m_orientation(LL)
00041 , m_data(0)
00042 , m_dataPos(0)
00043 #ifdef __PGFROISUPPORT__
00044 , m_nTiles(0)
00045 #endif
00046 {
00047 }
```

CSubband::~CSubband ()

Destructor.

Definition at line **51** of file **Subband.cpp**.

```
00051                         {
00052         FreeMemory();
00053 }
```

Member Function Documentation

bool CSubband::AllocMemory ()

Allocate a memory buffer to store all wavelet coefficients of this subband.

Returns

True if the allocation did work without any problems

Definition at line **77** of file **Subband.cpp**.

```
00077                        {
00078         UINT32 oldSize = m_size;
00079
00080 #ifdef __PGFROISUPPORT__
00081         m_size = BufferWidth()*m_ROI.Height();
00082 #endif
00083         ASSERT(m_size > 0);
00084
00085         if (m_data) {
00086                 if (oldSize >= m_size) {
00087                         return true;
00088                 } else {
00089                         delete[] m_data;
00090                         m_data = new(std::nothrow) DataT[m_size];
00091                         return (m_data != 0);
00092                 }
00093         } else {
00094                 m_data = new(std::nothrow) DataT[m_size];
00095                 return (m_data != 0);
00096         }
00097 }
```

void CSubband::Dequantize (int *quantParam*)

Perform subband dequantization with given quantization parameter. A scalar quantization (with dead-zone) is used. A large quantization value results in strong quantization and therefore in big quality loss.

Parameters

quantParam	A quantization parameter (larger or equal to 0)

Definition at line **154** of file **Subband.cpp**.

```
00154                                    {
00155         if (m_orientation == LL) {
00156                 quantParam -= m_level + 1;
00157         } else if (m_orientation == HH) {
00158                 quantParam -= m_level - 1;
00159         } else {
00160                 quantParam -= m_level;
00161         }
00162         if (quantParam > 0) {
00163                 for (UINT32 i=0; i < m_size; i++) {
00164                         m_data[i] <<= quantParam;
00165                 }
00166         }
00167 }
```

void CSubband::ExtractTile (CEncoder & *encoder*, bool *tile* = `false`, UINT32 *tileX* = `0`, UINT32 *tileY* = `0`)

Extracts a rectangular subregion of this subband. Write wavelet coefficients into buffer. It might throw an **IOException**.

Parameters

encoder	An encoder instance
tile	True if just a rectangular region is extracted, false if the entire subband is extracted.
tileX	Tile index in x-direction
tileY	Tile index in y-direction

Definition at line **177** of file **Subband.cpp**.

```
00177
{
00178 #ifdef __PGFROISUPPORT__
00179         if (tile) {
00180                 // compute tile position and size
00181                 UINT32 xPos, yPos, w, h;
00182                 TilePosition(tileX, tileY, xPos, yPos, w, h);
00183
00184                 // write values into buffer using partitiong scheme
00185                 encoder.Partition(this, w, h, xPos + yPos*m_width, m_width);
00186         } else
00187 #endif
00188         {
00189                 tileX; tileY; tile; // prevents from unreferenced formal
parameter warning
00190                 // write values into buffer using partitiong scheme
00191                 encoder.Partition(this, m_width, m_height, 0, m_width);
00192         }
00193 }
```

void CSubband::FreeMemory ()

Delete the memory buffer of this subband.

Definition at line **101** of file **Subband.cpp**.

```
00101                          {
00102         if (m_data) {
00103                 delete[] m_data; m_data = 0;
00104         }
00105 }
```

DataT * CSubband::GetBuffer () [inline]

Get a pointer to an array of all wavelet coefficients of this subband.

Returns

Pointer to array of wavelet coefficients

Definition at line **107** of file **Subband.h**.

```
00107 { return m_data; }
```

UINT32 CSubband::GetBuffPos () const [inline], [private]

Definition at line **151** of file **Subband.h**.

```
00151 { return m_dataPos; }
```

DataT CSubband::GetData (UINT32 *pos*) const [inline]

Return wavelet coefficient at given position.

Parameters

pos	A subband position (>= 0)

Returns

Wavelet coefficient

Definition at line **113** of file **Subband.h**.

```
00113 { ASSERT(pos < m_size); return m_data[pos]; }
```

int CSubband::GetHeight () const [inline]

Return height of this subband.

Returns

Height of this subband (in pixels)

Definition at line **123** of file **Subband.h**.

```
00123 { return m_height; }
```

int CSubband::GetLevel () const [inline]

Return level of this subband.

Returns

Level of this subband

Definition at line **118** of file **Subband.h**.

```
00118 { return m_level; }
```

Orientation CSubband::GetOrientation () const [inline]

Return orientation of this subband. LL LH HL HH

Returns

Orientation of this subband (LL, HL, LH, HH)

Definition at line **135** of file **Subband.h**.

```
00135 { return m_orientation; }
```

int CSubband::GetWidth () const [inline]

Return width of this subband.

Returns

Width of this subband (in pixels)

Definition at line **128** of file **Subband.h**.

```
00128 { return m_width; }
```

void CSubband::InitBuffPos () [inline], [private]

Definition at line **162** of file **Subband.h**.

```
00162 { m_dataPos = 0; }
```

void CSubband::Initialize (UINT32 *width*, UINT32 *height*, int *level*, Orientation *orient*) [private]

Definition at line **57** of file **Subband.cpp**.

```
00057
{
00058         m_width = width;
00059         m_height = height;
00060         m_size = m_width*m_height;
00061         m_level = level;
00062         m_orientation = orient;
00063         m_data = 0;
00064         m_dataPos = 0;
00065 #ifdef __PGFROISUPPORT__
00066         m_ROI.left = 0;
00067         m_ROI.top = 0;
00068         m_ROI.right = m_width;
00069         m_ROI.bottom = m_height;
00070         m_nTiles = 0;
00071 #endif
00072 }
```

void CSubband::PlaceTile (CDecoder & *decoder*, int *quantParam*, bool *tile* = false, UINT32 *tileX* = 0, UINT32 *tileY* = 0)

Decoding and dequantization of this subband. It might throw an **IOException**.

Parameters

decoder	A decoder instance
quantParam	Dequantization value
tile	True if just a rectangular region is placed, false if the entire subband is placed.
tileX	Tile index in x-direction
tileY	Tile index in y-direction

Definition at line **203** of file **Subband.cpp**.

```
00203
{
00204         // allocate memory
00205         if (!AllocMemory()) ReturnWithError(InsufficientMemory);
00206
00207         // correct quantParam with normalization factor
00208         if (m_orientation == LL) {
00209                 quantParam -= m_level + 1;
00210         } else if (m_orientation == HH) {
00211                 quantParam -= m_level - 1;
00212         } else {
00213                 quantParam -= m_level;
00214         }
00215         if (quantParam < 0) quantParam = 0;
00216
00217 #ifdef __PGFROISUPPORT__
00218         if (tile) {
00219                 UINT32 xPos, yPos, w, h;
00220
00221                 // compute tile position and size
00222                 TilePosition(tileX, tileY, xPos, yPos, w, h);
00223
00224                 ASSERT(xPos >= m_ROI.left && yPos >= m_ROI.top);
00225                 decoder.Partition(this, quantParam, w, h, (xPos - m_ROI.left)
+ (yPos - m_ROI.top)*BufferWidth(), BufferWidth());
00226         } else
00227 #endif
00228         {
00229                 tileX; tileY; tile; // prevents from unreferenced formal
parameter warning
00230                 // read values into buffer using partitiong scheme
00231                 decoder.Partition(this, quantParam, m_width, m_height, 0,
m_width);
00232         }
00233 }
```

void CSubband::Quantize (int *quantParam*)

Perform subband quantization with given quantization parameter. A scalar quantization (with dead-zone) is used. A large quantization value results in strong quantization and therefore in big quality loss.

Parameters

quantParam	A quantization parameter (larger or equal to 0)

Definition at line **112** of file **Subband.cpp**.

```
00112                                         {
00113         if (m_orientation == LL) {
00114                 quantParam -= (m_level + 1);
00115                 // uniform rounding quantization
00116                 if (quantParam > 0) {
00117                         quantParam--;
00118                         for (UINT32 i=0; i < m_size; i++) {
00119                                 if (m_data[i] < 0) {
00120                                         m_data[i] = -(((-m_data[i] >>
quantParam) + 1) >> 1);
00121                                 } else {
00122                                         m_data[i] = ((m_data[i] >> quantParam)
+ 1) >> 1;
00123                                 }
00124                         }
00125                 }
00126         } else {
00127                 if (m_orientation == HH) {
00128                         quantParam -= (m_level - 1);
```

```
00129                   } else {
00130                           quantParam -= m_level;
00131                   }
00132                   // uniform deadzone quantization
00133                   if (quantParam > 0) {
00134                           int threshold = ((1 << quantParam) * 7)/5;      // good
value
00135                           quantParam--;
00136                           for (UINT32 i=0; i < m_size; i++) {
00137                                   if (m_data[i] < -threshold) {
00138                                           m_data[i] = -(((-m_data[i] >>
quantParam) + 1) >> 1);
00139                                   } else if (m_data[i] > threshold) {
00140                                           m_data[i] = ((m_data[i] >> quantParam)
+ 1) >> 1;
00141                                   } else {
00142                                           m_data[i] = 0;
00143                                   }
00144                           }
00145                   }
00146           }
00147 }
```

DataT CSubband::ReadBuffer () [inline], [private]

Definition at line **149** of file **Subband.h**.

```
00149 { ASSERT(m_dataPos < m_size); return m_data[m_dataPos++]; }
```

void CSubband::SetBuffer (DataT * *b*) [inline], [private]

Definition at line **148** of file **Subband.h**.

```
00148 { ASSERT(b); m_data = b; }
```

void CSubband::SetData (UINT32 *pos*, DataT *v*) [inline]

Store wavelet coefficient in subband at given position.

Parameters

pos	A subband position (>= 0)
v	A wavelet coefficient

Definition at line **102** of file **Subband.h**.

```
00102 { ASSERT(pos < m_size); m_data[pos] = v; }
```

void CSubband::WriteBuffer (DataT *val*) [inline], [private]

Definition at line **147** of file **Subband.h**.

```
00147 { ASSERT(m_dataPos < m_size); m_data[m_dataPos++] = val; }
```

Friends And Related Function Documentation

friend class CRoiIndices [friend]

Definition at line **44** of file **Subband.h**.

friend class CWaveletTransform [friend]

Definition at line **43** of file **Subband.h**.

Member Data Documentation

DataT* CSubband::m_data `[private]`

buffer

Definition at line **172** of file **Subband.h**.

UINT32 CSubband::m_dataPos `[private]`

current position in m_data

Definition at line **171** of file **Subband.h**.

UINT32 CSubband::m_height `[private]`

height in pixels

Definition at line **167** of file **Subband.h**.

int CSubband::m_level `[private]`

recursion level

Definition at line **169** of file **Subband.h**.

Orientation CSubband::m_orientation `[private]`

0=LL, 1=HL, 2=LH, 3=HH L=lowpass filtered, H=highpass filterd

Definition at line **170** of file **Subband.h**.

UINT32 CSubband::m_size `[private]`

size of data buffer m_data

Definition at line **168** of file **Subband.h**.

UINT32 CSubband::m_width `[private]`

width in pixels

Definition at line **166** of file **Subband.h**.

The documentation for this class was generated from the following files:

- **Subband.h**
- **Subband.cpp**

CWaveletTransform Class Reference

PGF wavelet transform.

`#include <WaveletTransform.h>`

Public Member Functions

- **CWaveletTransform** (UINT32 width, UINT32 height, int levels, **DataT** *data=nullptr)
- **~CWaveletTransform** ()
  Destructor.
- OSError **ForwardTransform** (int level, int quant)
- OSError **InverseTransform** (int level, UINT32 *width, UINT32 *height, **DataT** **data)
- **CSubband** * **GetSubband** (int level, **Orientation** orientation)

Private Member Functions

- void **Destroy** ()
- void **InitSubbands** (UINT32 width, UINT32 height, **DataT** *data)
- void **ForwardRow** (**DataT** *buff, UINT32 width)
- void **InverseRow** (**DataT** *buff, UINT32 width)
- void **InterleavedToSubbands** (int destLevel, **DataT** *loRow, **DataT** *hiRow, UINT32 width)
- void **SubbandsToInterleaved** (int srcLevel, **DataT** *loRow, **DataT** *hiRow, UINT32 width)

Private Attributes

- int **m_nLevels**
  number of LL levels: one more than header.nLevels in PGFimage
- **CSubband**(* **m_subband**)[**NSubbands**]
  quadtree of subbands: LL HL LH HH

Friends

- class **CSubband**

Detailed Description

PGF wavelet transform.

PGF wavelet transform class.

Author

C. Stamm, R. Spuler

Definition at line **55** of file **WaveletTransform.h**.

Constructor & Destructor Documentation

CWaveletTransform::CWaveletTransform (UINT32 *width*, UINT32 *height*, int *levels*, DataT * *data* = `nullptr`)

Constructor: Constructs a wavelet transform pyramid of given size and levels.

Parameters

width	The width of the original image (at level 0) in pixels
height	The height of the original image (at level 0) in pixels
levels	The number of levels (>= 0)
data	Input data of subband LL at level 0

Definition at line **40** of file **WaveletTransform.cpp**.

```
00041 : m_nLevels(levels + 1) // m_nLevels in CPGFImage determines the number of FWT
steps; this.m_nLevels determines the number subband-planes
00042 , m_subband(nullptr)
00043 #ifdef __PGFROISUPPORT__
00044 , m_indices(nullptr)
00045 #endif
00046 {
00047         ASSERT(m_nLevels > 0 && m_nLevels <= MaxLevel + 1);
00048         InitSubbands(width, height, data);
00049 }
```

CWaveletTransform::~CWaveletTransform () [inline]

Destructor.

Definition at line **69** of file **WaveletTransform.h**.

```
00069 { Destroy(); }
```

Member Function Documentation

void CWaveletTransform::Destroy () [inline], [private]

Definition at line **125** of file **WaveletTransform.h**.

```
00125                            {
00126                 delete[] m_subband; m_subband = nullptr;
00127         #ifdef __PGFROISUPPORT__
00128                 delete[] m_indices; m_indices = nullptr;
00129         #endif
00130         }
```

void CWaveletTransform::ForwardRow (DataT * *buff*, UINT32 *width*) [private]

Definition at line **180** of file **WaveletTransform.cpp**.

```
00180                                                              {
00181         if (width >= FilterSize) {
00182                 UINT32 i = 3;
00183 
00184                 // left border handling
00185                 src[1] -= ((src[0] + src[2] + c1) >> 1); // high pass
00186                 src[0] += ((src[1] + c1) >> 1); // low pass
00187 
00188                 // middle part
00189                 for (; i < width-1; i += 2) {
00190                         src[i] -= ((src[i-1] + src[i+1] + c1) >> 1); // high pass
00191                         src[i-1] += ((src[i-2] + src[i] + c2) >> 2); // low pass
00192                 }
00193 
00194                 // right border handling
00195                 if (width & 1) {
00196                         src[i-1] += ((src[i-2] + c1) >> 1); // low pass
00197                 } else {
00198                         src[i] -= src[i-1]; // high pass
00199                         src[i-1] += ((src[i-2] + src[i] + c2) >> 2); // low pass
00200                 }
00201         }
00202 }
```

OSError CWaveletTransform::ForwardTransform (int *level*, int *quant*)

Compute fast forward wavelet transform of LL subband at given level and stores result in all 4 subbands of level + 1.

Parameters

level	A wavelet transform pyramid level (>= 0 && < Levels())
quant	A quantization value (linear scalar quantization)

Returns

error in case of a memory allocation problem

Definition at line **88** of file **WaveletTransform.cpp**.

```
00088                                                                  {
00089         ASSERT(level >= 0 && level < m_nLevels - 1);
00090         const int destLevel = level + 1;
00091         ASSERT(m_subband[destLevel]);
00092         CSubband* srcBand = &m_subband[level][LL]; ASSERT(srcBand);
00093         const UINT32 width = srcBand->GetWidth();
00094         const UINT32 height = srcBand->GetHeight();
00095         DataT* src = srcBand->GetBuffer(); ASSERT(src);
00096         DataT *row0, *row1, *row2, *row3;
00097
00098         // Allocate memory for next transform level
00099         for (int i=0; i < NSubbands; i++) {
00100                 if (!m_subband[destLevel][i].AllocMemory()) return
InsufficientMemory;
00101         }
00102
00103         if (height >= FilterSize) { // changed from FilterSizeH to FilterSize
00104                 // top border handling
00105                 row0 = src; row1 = row0 + width; row2 = row1 + width;
00106                 ForwardRow(row0, width);
00107                 ForwardRow(row1, width);
00108                 ForwardRow(row2, width);
00109                 for (UINT32 k=0; k < width; k++) {
00110                         row1[k] -= ((row0[k] + row2[k] + c1) >> 1); // high pass
00111                         row0[k] += ((row1[k] + c1) >> 1); // low pass
00112                 }
00113                 InterleavedToSubbands(destLevel, row0, row1, width);
00114                 row0 = row1; row1 = row2; row2 += width; row3 = row2 + width;
00115
00116                 // middle part
00117                 for (UINT32 i=3; i < height-1; i += 2) {
00118                         ForwardRow(row2, width);
00119                         ForwardRow(row3, width);
00120                         for (UINT32 k=0; k < width; k++) {
00121                                 row2[k] -= ((row1[k] + row3[k] + c1) >> 1); //
high pass filter
00122                                 row1[k] += ((row0[k] + row2[k] + c2) >> 2); //
low pass filter
00123                         }
00124                         InterleavedToSubbands(destLevel, row1, row2, width);
00125                         row0 = row2; row1 = row3; row2 = row3 + width; row3 =
row2 + width;
00126                 }
00127
00128                 // bottom border handling
00129                 if (height & 1) {
00130                         for (UINT32 k=0; k < width; k++) {
00131                                 row1[k] += ((row0[k] + c1) >> 1); // low pass
00132                         }
00133                         InterleavedToSubbands(destLevel, row1, nullptr,
width);
00134                         row0 = row1; row1 += width;
00135                 } else {
00136                         ForwardRow(row2, width);
00137                         for (UINT32 k=0; k < width; k++) {
00138                                 row2[k] -= row1[k]; // high pass
00139                                 row1[k] += ((row0[k] + row2[k] + c2) >> 2); //
low pass
00140                         }
00141                         InterleavedToSubbands(destLevel, row1, row2, width);
00142                         row0 = row1; row1 = row2; row2 += width;
00143                 }
```

```
00144           } else {
00145                   // if height is too small
00146                   row0 = src; row1 = row0 + width;
00147                   // first part
00148                   for (UINT32 k=0; k < height; k += 2) {
00149                           ForwardRow(row0, width);
00150                           ForwardRow(row1, width);
00151                           InterleavedToSubbands(destLevel, row0, row1, width);
00152                           row0 += width << 1; row1 += width << 1;
00153                   }
00154                   // bottom
00155                   if (height & 1) {
00156                           InterleavedToSubbands(destLevel, row0, nullptr,
width);
00157                   }
00158           }
00159
00160           if (quant > 0) {
00161                   // subband quantization (without LL)
00162                   for (int i=1; i < NSubbands; i++) {
00163                           m_subband[destLevel][i].Quantize(quant);
00164                   }
00165                   // LL subband quantization
00166                   if (destLevel == m_nLevels - 1) {
00167                           m_subband[destLevel][LL].Quantize(quant);
00168                   }
00169           }
00170
00171           // free source band
00172           srcBand->FreeMemory();
00173           return NoError;
00174 }
```

CSubband * CWaveletTransform::GetSubband (int *level*, Orientation *orientation*) [inline]

Get pointer to one of the 4 subband at a given level.

Parameters

level	A wavelet transform pyramid level (>= 0 && <= Levels())
orientation	A quarter of the subband (LL, LH, HL, HH)

Definition at line **93** of file **WaveletTransform.h**.

```
00093                                                                       {
00094                   ASSERT(level >= 0 && level < m_nLevels);
00095                   return &m_subband[level][orientation];
00096           }
```

void CWaveletTransform::InitSubbands (UINT32 *width*, UINT32 *height*, DataT * *data*) [private]

Definition at line **53** of file **WaveletTransform.cpp**.

```
00053
{
00054           if (m_subband) Destroy();
00055
00056           // create subbands
00057           m_subband = new CSubband[m_nLevels][NSubbands];
00058
00059           // init subbands
00060           UINT32 loWidth = width;
00061           UINT32 hiWidth = width;
00062           UINT32 loHeight = height;
00063           UINT32 hiHeight = height;
00064
00065           for (int level = 0; level < m_nLevels; level++) {
00066                   m_subband[level][LL].Initialize(loWidth, loHeight, level, LL);
// LL
00067                   m_subband[level][HL].Initialize(hiWidth, loHeight, level, HL);
//      HL
00068                   m_subband[level][LH].Initialize(loWidth, hiHeight, level, LH);
// LH
```

```
00069                 m_subband[level][HH].Initialize(hiWidth, hiHeight, level, HH);
//    HH
00070                 hiWidth = loWidth >> 1;                 hiHeight = loHeight >>
1;
00071                 loWidth = (loWidth + 1) >> 1;   loHeight = (loHeight + 1) >> 1;
00072         }
00073         if (data) {
00074                 m_subband[0][LL].SetBuffer(data);
00075         }
00076 }
```

void CWaveletTransform::InterleavedToSubbands (int *destLevel*, DataT * *loRow*, DataT * *hiRow*, UINT32 *width*) `[private]`

Definition at line **206** of file **WaveletTransform.cpp**.

```
00206
{
00207         const UINT32 wquot = width >> 1;
00208         const bool wrem = (width & 1);
00209         CSubband &ll = m_subband[destLevel][LL], &hl =
m_subband[destLevel][HL];
00210         CSubband &lh = m_subband[destLevel][LH], &hh =
m_subband[destLevel][HH];
00211
00212         if (hiRow) {
00213                 for (UINT32 i=0; i < wquot; i++) {
00214                         ll.WriteBuffer(*loRow++);       // first access, than
increment
00215                         hl.WriteBuffer(*loRow++);
00216                         lh.WriteBuffer(*hiRow++);       // first access, than
increment
00217                         hh.WriteBuffer(*hiRow++);
00218                 }
00219                 if (wrem) {
00220                         ll.WriteBuffer(*loRow);
00221                         lh.WriteBuffer(*hiRow);
00222                 }
00223         } else {
00224                 for (UINT32 i=0; i < wquot; i++) {
00225                         ll.WriteBuffer(*loRow++);       // first access, than
increment
00226                         hl.WriteBuffer(*loRow++);
00227                 }
00228                 if (wrem) ll.WriteBuffer(*loRow);
00229         }
00230 }
```

void CWaveletTransform::InverseRow (DataT * *buff*, UINT32 *width*) `[private]`

Definition at line **419** of file **WaveletTransform.cpp**.

```
00419                                                           {
00420         if (width >= FilterSize) {
00421                 UINT32 i = 2;
00422
00423                 // left border handling
00424                 dest[0] -= ((dest[1] + c1) >> 1); // even
00425
00426                 // middle part
00427                 for (; i < width - 1; i += 2) {
00428                         dest[i] -= ((dest[i-1] + dest[i+1] + c2) >> 2); // even
00429                         dest[i-1] += ((dest[i-2] + dest[i] + c1) >> 1); // odd
00430                 }
00431
00432                 // right border handling
00433                 if (width & 1) {
00434                         dest[i] -= ((dest[i-1] + c1) >> 1); // even
00435                         dest[i-1] += ((dest[i-2] + dest[i] + c1) >> 1); // odd
00436                 } else {
00437                         dest[i-1] += dest[i-2]; // odd
00438                 }
00439         }
```

```
00440 }
```

OSError CWaveletTransform::InverseTransform (int ***level*****, UINT32 *** ***width*****, UINT32 *** ***height*****, DataT **** ***data*****)**

Compute fast inverse wavelet transform of all 4 subbands of given level and stores result in LL subband of level - 1.

Parameters

level	A wavelet transform pyramid level (> 0 && <= Levels())
width	A pointer to the returned width of subband LL (in pixels)
height	A pointer to the returned height of subband LL (in pixels)
data	A pointer to the returned array of image data

Returns

error in case of a memory allocation problem

Definition at line **245** of file **WaveletTransform.cpp**.

```
00245
{
00246         ASSERT(srcLevel > 0 && srcLevel < m_nLevels);
00247         const int destLevel = srcLevel - 1;
00248         ASSERT(m_subband[destLevel]);
00249         CSubband* destBand = &m_subband[destLevel][LL];
00250         UINT32 width, height;
00251
00252         // allocate memory for the results of the inverse transform
00253         if (!destBand->AllocMemory()) return InsufficientMemory;
00254         DataT *origin = destBand->GetBuffer(), *row0, *row1, *row2, *row3;
00255
00256 #ifdef __PGFROISUPPORT__
00257         PGFRect destROI = destBand->GetAlignedROI();
00258         const UINT32 destWidth  = destROI.Width();  // destination buffer width
00259         const UINT32 destHeight = destROI.Height(); // destination buffer height
00260         width = destWidth;              // destination working width
00261         height = destHeight;    // destination working height
00262
00263         // update destination ROI
00264         if (destROI.top & 1) {
00265                 destROI.top++;
00266                 origin += destWidth;
00267                 height--;
00268         }
00269         if (destROI.left & 1) {
00270                 destROI.left++;
00271                 origin++;
00272                 width--;
00273         }
00274
00275         // init source buffer position
00276         const UINT32 leftD = destROI.left >> 1;
00277         const UINT32 left0 = m_subband[srcLevel][LL].GetAlignedROI().left;
00278         const UINT32 left1 = m_subband[srcLevel][HL].GetAlignedROI().left;
00279         const UINT32 topD = destROI.top >> 1;
00280         const UINT32 top0 = m_subband[srcLevel][LL].GetAlignedROI().top;
00281         const UINT32 top1 = m_subband[srcLevel][LH].GetAlignedROI().top;
00282         ASSERT(m_subband[srcLevel][LH].GetAlignedROI().left == left0);
00283         ASSERT(m_subband[srcLevel][HH].GetAlignedROI().left == left1);
00284         ASSERT(m_subband[srcLevel][HL].GetAlignedROI().top == top0);
00285         ASSERT(m_subband[srcLevel][HH].GetAlignedROI().top == top1);
00286
00287         UINT32 srcOffsetX[2] = { 0, 0 };
00288         UINT32 srcOffsetY[2] = { 0, 0 };
00289
00290         if (leftD >= __max(left0, left1)) {
00291                 srcOffsetX[0] = leftD - left0;
00292                 srcOffsetX[1] = leftD - left1;
00293         } else {
00294                 if (left0 <= left1) {
00295                         const UINT32 dx = (left1 - leftD) << 1;
00296                         destROI.left += dx;
00297                         origin += dx;
00298                         width -= dx;
```

```
00299                           srcOffsetX[0] = left1 - left0;
00300                   } else {
00301                           const UINT32 dx = (left0 - leftD) << 1;
00302                           destROI.left += dx;
00303                           origin += dx;
00304                           width -= dx;
00305                           srcOffsetX[1] = left0 - left1;
00306                   }
00307           }
00308           if (topD >= __max(top0, top1)) {
00309                   srcOffsetY[0] = topD - top0;
00310                   srcOffsetY[1] = topD - top1;
00311           } else {
00312                   if (top0 <= top1) {
00313                           const UINT32 dy = (top1 - topD) << 1;
00314                           destROI.top += dy;
00315                           origin += dy*destWidth;
00316                           height -= dy;
00317                           srcOffsetY[0] = top1 - top0;
00318                   } else {
00319                           const UINT32 dy = (top0 - topD) << 1;
00320                           destROI.top += dy;
00321                           origin += dy*destWidth;
00322                           height -= dy;
00323                           srcOffsetY[1] = top0 - top1;
00324                   }
00325           }
00326
00327           m_subband[srcLevel][LL].InitBuffPos(srcOffsetX[0], srcOffsetY[0]);
00328           m_subband[srcLevel][HL].InitBuffPos(srcOffsetX[1], srcOffsetY[0]);
00329           m_subband[srcLevel][LH].InitBuffPos(srcOffsetX[0], srcOffsetY[1]);
00330           m_subband[srcLevel][HH].InitBuffPos(srcOffsetX[1], srcOffsetY[1]);
00331
00332 #else
00333           width = destBand->GetWidth();
00334           height = destBand->GetHeight();
00335           PGFRect destROI(0, 0, width, height);
00336           const UINT32 destWidth = width; // destination buffer width
00337           const UINT32 destHeight = height; // destination buffer height
00338
00339           // init source buffer position
00340           for (int i = 0; i < NSubbands; i++) {
00341                   m_subband[srcLevel][i].InitBuffPos();
00342           }
00343 #endif
00344
00345           if (destHeight >= FilterSize) { // changed from FilterSizeH to FilterSize
00346                   // top border handling
00347                   row0 = origin; row1 = row0 + destWidth;
00348                   SubbandsToInterleaved(srcLevel, row0, row1, width);
00349                   for (UINT32 k = 0; k < width; k++) {
00350                           row0[k] -= ((row1[k] + c1) >> 1); // even
00351                   }
00352
00353                   // middle part
00354                   row2 = row1 + destWidth; row3 = row2 + destWidth;
00355                   for (UINT32 i = destROI.top + 2; i < destROI.bottom - 1; i +=
2) {
00356                           SubbandsToInterleaved(srcLevel, row2, row3, width);
00357                           for (UINT32 k = 0; k < width; k++) {
00358                                   row2[k] -= ((row1[k] + row3[k] + c2) >> 2); //
even
00359                                   row1[k] += ((row0[k] + row2[k] + c1) >> 1); //
odd
00360                           }
00361                           InverseRow(row0, width);
00362                           InverseRow(row1, width);
00363                           row0 = row2; row1 = row3; row2 = row1 + destWidth; row3
= row2 + destWidth;
00364                   }
00365
00366                   // bottom border handling
00367                   if (height & 1) {
00368                           SubbandsToInterleaved(srcLevel, row2, nullptr, width);
00369                           for (UINT32 k = 0; k < width; k++) {
00370                                   row2[k] -= ((row1[k] + c1) >> 1); // even
```

```
00371                           row1[k] += ((row0[k] + row2[k] + c1) >> 1); //
odd
00372                   }
00373                   InverseRow(row0, width);
00374                   InverseRow(row1, width);
00375                   InverseRow(row2, width);
00376                   row0 = row1; row1 = row2; row2 += destWidth;
00377           } else {
00378                   for (UINT32 k = 0; k < width; k++) {
00379                           row1[k] += row0[k];
00380                   }
00381                   InverseRow(row0, width);
00382                   InverseRow(row1, width);
00383                   row0 = row1; row1 += destWidth;
00384           }
00385   } else {
00386           // height is too small
00387           row0 = origin; row1 = row0 + destWidth;
00388           // first part
00389           for (UINT32 k = 0; k < height; k += 2) {
00390                   SubbandsToInterleaved(srcLevel, row0, row1, width);
00391                   InverseRow(row0, width);
00392                   InverseRow(row1, width);
00393                   row0 += destWidth << 1; row1 += destWidth << 1;
00394           }
00395           // bottom
00396           if (height & 1) {
00397                   SubbandsToInterleaved(srcLevel, row0, nullptr, width);
00398                   InverseRow(row0, width);
00399           }
00400   }
00401
00402   // free memory of the current srcLevel
00403   for (int i = 0; i < NSubbands; i++) {
00404           m_subband[srcLevel][i].FreeMemory();
00405   }
00406
00407   // return info
00408   *w = destWidth;
00409   *h = destHeight;
00410   *data = destBand->GetBuffer();
00411   return NoError;
00412 }
```

void CWaveletTransform::SubbandsToInterleaved (int ***srcLevel*****, DataT *** ***loRow*****, DataT *** ***hiRow*****, UINT32** ***width*****)** `[private]`

Definition at line **444** of file **WaveletTransform.cpp**.

```
00444
{
00445   const UINT32 wquot = width >> 1;
00446   const bool wrem = (width & 1);
00447   CSubband &ll = m_subband[srcLevel][LL], &hl = m_subband[srcLevel][HL];
00448   CSubband &lh = m_subband[srcLevel][LH], &hh = m_subband[srcLevel][HH];
00449
00450   if (hiRow) {
00451   #ifdef __PGFROISUPPORT__
00452           const bool storePos = wquot < ll.BufferWidth();
00453           UINT32 llPos = 0, hlPos = 0, lhPos = 0, hhPos = 0;
00454
00455           if (storePos) {
00456                   // save current src buffer positions
00457                   llPos = ll.GetBuffPos();
00458                   hlPos = hl.GetBuffPos();
00459                   lhPos = lh.GetBuffPos();
00460                   hhPos = hh.GetBuffPos();
00461           }
00462   #endif
00463
00464           for (UINT32 i=0; i < wquot; i++) {
00465                   *loRow++ = ll.ReadBuffer();// first access, than
increment
00466                   *loRow++ = hl.ReadBuffer();// first access, than
increment
```

```
00467                          *hiRow++ = lh.ReadBuffer();// first access, than
increment
00468                          *hiRow++ = hh.ReadBuffer();// first access, than
increment
00469                  }
00470
00471                  if (wrem) {
00472                          *loRow++ = ll.ReadBuffer();// first access, than
increment
00473                          *hiRow++ = lh.ReadBuffer();// first access, than
increment
00474                  }
00475
00476          #ifdef __PGFROISUPPORT__
00477                  if (storePos) {
00478                          // increment src buffer positions
00479                          ll.IncBuffRow(llPos);
00480                          hl.IncBuffRow(hlPos);
00481                          lh.IncBuffRow(lhPos);
00482                          hh.IncBuffRow(hhPos);
00483                  }
00484          #endif
00485
00486          } else {
00487          #ifdef __PGFROISUPPORT__
00488                  const bool storePos = wquot < ll.BufferWidth();
00489                  UINT32 llPos = 0, hlPos = 0;
00490
00491                  if (storePos) {
00492                          // save current src buffer positions
00493                          llPos = ll.GetBuffPos();
00494                          hlPos = hl.GetBuffPos();
00495                  }
00496          #endif
00497
00498                  for (UINT32 i=0; i < wquot; i++) {
00499                          *loRow++ = ll.ReadBuffer();// first access, than
increment
00500                          *loRow++ = hl.ReadBuffer();// first access, than
increment
00501                  }
00502                  if (wrem) *loRow++ = ll.ReadBuffer();
00503
00504          #ifdef __PGFROISUPPORT__
00505                  if (storePos) {
00506                          // increment src buffer positions
00507                          ll.IncBuffRow(llPos);
00508                          hl.IncBuffRow(hlPos);
00509                  }
00510          #endif
00511          }
00512 }
```

Friends And Related Function Documentation

friend class CSubband [friend]

Definition at line **56** of file **WaveletTransform.h**.

Member Data Documentation

int CWaveletTransform::m_nLevels [private]

number of LL levels: one more than header.nLevels in PGFimage

Definition at line **141** of file **WaveletTransform.h**.

CSubband(* CWaveletTransform::m_subband)[NSubbands] `[private]`

quadtree of subbands: LL HL LH HH

Definition at line **142** of file **WaveletTransform.h**.

The documentation for this class was generated from the following files:

- **WaveletTransform.h**
- **WaveletTransform.cpp**

IOException Struct Reference

PGF exception.
`#include <PGFtypes.h>`

Public Member Functions

- **IOException** ()
  Standard constructor.

- **IOException** (OSError err)

Public Attributes

- OSError **error**
  operating system error code

Detailed Description

PGF exception.

PGF I/O exception

Author

C. Stamm

Definition at line **210** of file **PGFtypes.h**.

Constructor & Destructor Documentation

IOException::IOException () [inline]

Standard constructor.

Definition at line **214** of file **PGFtypes.h**.

```
00214 : error(NoError) {}
```

IOException::IOException (OSError *err*) [inline]

Constructor

Parameters

err	Run-time error

Definition at line **218** of file **PGFtypes.h**.

```
00218 : error(err) {}
```

Member Data Documentation

OSError IOException::error

operating system error code

Definition at line **211** of file **PGFtypes.h**.

The documentation for this struct was generated from the following file:

- **PGFtypes.h**

PGFHeader Struct Reference

PGF header.

```
#include <PGFtypes.h>
```

Public Member Functions

- **PGFHeader** ()

Public Attributes

- UINT32 **width**
  image width in pixels
- UINT32 **height**
  image height in pixels
- UINT8 **nLevels**
  number of FWT transforms
- UINT8 **quality**
  quantization parameter: 0=lossless, 4=standard, 6=poor quality
- UINT8 **bpp**
  bits per pixel
- UINT8 **channels**
  number of channels
- UINT8 **mode**
  image mode according to Adobe's image modes
- UINT8 **usedBitsPerChannel**
  number of used bits per channel in 16- and 32-bit per channel modes
- **PGFVersionNumber version**
  codec version number: (since Version 7)

Detailed Description

PGF header.

PGF header contains image information

Author

C. Stamm

Definition at line **151** of file **PGFtypes.h**.

Constructor & Destructor Documentation

PGFHeader::PGFHeader () [inline]

Definition at line **152** of file **PGFtypes.h**.

```
00152 : width(0), height(0), nLevels(0), quality(0), bpp(0), channels(0),
mode(ImageModeUnknown), usedBitsPerChannel(0), version(0, 0, 0) {}
```

Member Data Documentation

UINT8 PGFHeader::bpp

bits per pixel

Definition at line **157** of file **PGFtypes.h**.

UINT8 PGFHeader::channels

number of channels

Definition at line **158** of file **PGFtypes.h**.

UINT32 PGFHeader::height

image height in pixels

Definition at line **154** of file **PGFtypes.h**.

UINT8 PGFHeader::mode

image mode according to Adobe's image modes

Definition at line **159** of file **PGFtypes.h**.

UINT8 PGFHeader::nLevels

number of FWT transforms

Definition at line **155** of file **PGFtypes.h**.

UINT8 PGFHeader::quality

quantization parameter: 0=lossless, 4=standard, 6=poor quality

Definition at line **156** of file **PGFtypes.h**.

UINT8 PGFHeader::usedBitsPerChannel

number of used bits per channel in 16- and 32-bit per channel modes

Definition at line **160** of file **PGFtypes.h**.

PGFVersionNumber PGFHeader::version

codec version number: (since Version 7)

Definition at line **161** of file **PGFtypes.h**.

UINT32 PGFHeader::width

image width in pixels

Definition at line **153** of file **PGFtypes.h**.

The documentation for this struct was generated from the following file:

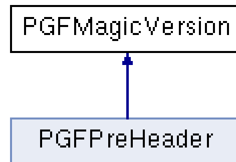
- **PGFtypes.h**

PGFMagicVersion Struct Reference

PGF identification and version.

`#include <PGFtypes.h>`

Inheritance diagram for PGFMagicVersion:



Public Attributes

- char **magic** [3]
  PGF identification = "PGF".

- UINT8 **version**
  PGF version.

Detailed Description

PGF identification and version.

general PGF file structure **PGFPreHeader** **PGFHeader** [**PGFPostHeader**] LevelLengths Level_n-1 Level_n-2 ... Level_0 **PGFPostHeader** ::= [ColorTable] [UserData] LevelLengths ::= UINT32[nLevels] PGF magic and version (part of PGF pre-header)

Author

C. Stamm

Definition at line **113** of file **PGFtypes.h**.

Member Data Documentation

char PGFMagicVersion::magic[3]

PGF identification = "PGF".

Definition at line **114** of file **PGFtypes.h**.

UINT8 PGFMagicVersion::version

PGF version.

Definition at line **115** of file **PGFtypes.h**.

The documentation for this struct was generated from the following file:

- **PGFtypes.h**

PGFPostHeader Struct Reference

Optional PGF post-header.
`#include <PGFtypes.h>`

Public Attributes

- RGBQUAD **clut** [**ColorTableLen**]
  color table for indexed color images (optional part of file header)
- UINT8 * **userData**
  user data of size userDataLen (optional part of file header)
- UINT32 **userDataLen**
  user data size in bytes (not part of file header)
- UINT32 **cachedUserDataLen**
  cached user data size in bytes (not part of file header)

Detailed Description

Optional PGF post-header.

PGF post-header is optional. It contains color table and user data

Author

C. Stamm

Definition at line **169** of file **PGFtypes.h**.

Member Data Documentation

UINT32 PGFPostHeader::cachedUserDataLen

cached user data size in bytes (not part of file header)

Definition at line **173** of file **PGFtypes.h**.

RGBQUAD PGFPostHeader::clut[ColorTableLen]

color table for indexed color images (optional part of file header)

Definition at line **170** of file **PGFtypes.h**.

UINT8* PGFPostHeader::userData

user data of size userDataLen (optional part of file header)

Definition at line **171** of file **PGFtypes.h**.

UINT32 PGFPostHeader::userDataLen

user data size in bytes (not part of file header)

Definition at line **172** of file **PGFtypes.h**.

The documentation for this struct was generated from the following file:

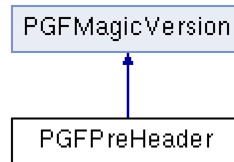
- **PGFtypes.h**

PGFPreHeader Struct Reference

PGF pre-header.

`#include <PGFtypes.h>`

Inheritance diagram for PGFPreHeader:



Public Attributes

- UINT32 **hSize**
  total size of ***PGFHeader****, [ColorTable], and [UserData] in bytes (since Version 6: 4 Bytes)*
- char **magic** [3]
  PGF identification = "PGF".
- UINT8 **version**
  PGF version.

Detailed Description

PGF pre-header.

PGF pre-header defined header length and PGF identification and version

Author

C. Stamm

Definition at line **123** of file **PGFtypes.h**.

Member Data Documentation

UINT32 PGFPreHeader::hSize

total size of **PGFHeader**, [ColorTable], and [UserData] in bytes (since Version 6: 4 Bytes)

Definition at line **124** of file **PGFtypes.h**.

char PGFMagicVersion::magic[3] `[inherited]`

PGF identification = "PGF".

Definition at line **114** of file **PGFtypes.h**.

UINT8 PGFMagicVersion::version `[inherited]`

PGF version.

Definition at line **115** of file **PGFtypes.h**.

The documentation for this struct was generated from the following file:

- **PGFtypes.h**

PGFRect Struct Reference

Rectangle.

`#include <PGFtypes.h>`

Public Member Functions

- **PGFRect** ()
  Standard constructor.

- **PGFRect** (UINT32 x, UINT32 y, UINT32 width, UINT32 height)
- UINT32 **Width** () const
- UINT32 **Height** () const
- bool **IsInside** (UINT32 x, UINT32 y) const

Public Attributes

- UINT32 **left**
- UINT32 **top**
- UINT32 **right**
- UINT32 **bottom**

Detailed Description

Rectangle.

Rectangle

Author

C. Stamm

Definition at line **225** of file **PGFtypes.h**.

Constructor & Destructor Documentation

PGFRect::PGFRect () `[inline]`

Standard constructor.

Definition at line **229** of file **PGFtypes.h**.

```
00229 : left(0), top(0), right(0), bottom(0) {}
```

PGFRect::PGFRect (UINT32 *x*, UINT32 *y*, UINT32 *width*, UINT32 *height*) `[inline]`

Constructor

Parameters

x	Left offset
y	Top offset
width	Rectangle width
height	Rectangle height

Definition at line **236** of file **PGFtypes.h**.

```
00236 : left(x), top(y), right(x + width), bottom(y + height) {}
```

Member Function Documentation

UINT32 PGFRect::Height () const [inline]

Returns

Rectangle height

Definition at line **259** of file **PGFtypes.h**.

```
00259 { return bottom - top; }
```

bool PGFRect::IsInside (UINT32 *x*, UINT32 *y*) const [inline]

Test if point (x,y) is inside this rectangle (inclusive top-left edges, exclusive bottom-right edges)

Parameters

x	Point coordinate x
y	Point coordinate y

Returns

True if point (x,y) is inside this rectangle (inclusive top-left edges, exclusive bottom-right edges)

Definition at line **265** of file **PGFtypes.h**.

```
00265 { return (x >= left && x < right && y >= top && y < bottom); }
```

UINT32 PGFRect::Width () const [inline]

Returns

Rectangle width

Definition at line **256** of file **PGFtypes.h**.

```
00256 { return right - left; }
```

Member Data Documentation

UINT32 PGFRect::bottom

Definition at line **226** of file **PGFtypes.h**.

UINT32 PGFRect::left

Definition at line **226** of file **PGFtypes.h**.

UINT32 PGFRect::right

Definition at line **226** of file **PGFtypes.h**.

UINT32 PGFRect::top

Definition at line **226** of file **PGFtypes.h**.

The documentation for this struct was generated from the following file:

- **PGFtypes.h**

PGFVersionNumber Struct Reference

version number stored in header since major version 7

#include <PGFtypes.h>

Public Member Functions

- **PGFVersionNumber** (UINT8 _major, UINT8 _year, UINT8 _week)

Public Attributes

- UINT16 **major**: 4
  major version number

- UINT16 **year**: 6
  year since 2000 (year 2001 = 1)

- UINT16 **week**: 6
  week number in a year

Detailed Description

version number stored in header since major version 7

Version number since major version 7

Author

C. Stamm

Definition at line **132** of file **PGFtypes.h**.

Constructor & Destructor Documentation

PGFVersionNumber::PGFVersionNumber (UINT8 *_major*, UINT8 *_year*, UINT8 *_week*) [inline]

Definition at line **133** of file **PGFtypes.h**.

```
00133 : major(_major), year(_year), week(_week) {}
```

Member Data Documentation

UINT16 PGFVersionNumber::major

major version number

Definition at line **140** of file **PGFtypes.h**.

UINT16 PGFVersionNumber::week

week number in a year

Definition at line **142** of file **PGFtypes.h**.

UINT16 PGFVersionNumber::year

year since 2000 (year 2001 = 1)

Definition at line **141** of file **PGFtypes.h**.

The documentation for this struct was generated from the following file:

- **PGFtypes.h**

ROIBlockHeader::RBH Struct Reference

Named ROI block header (part of the union)
`#include <PGFtypes.h>`

Public Attributes

- UINT16 **bufferSize**: **RLblockSizeLen**
  number of uncoded UINT32 values in a block

- UINT16 **tileEnd**: 1
  1: last part of a tile

Detailed Description

Named ROI block header (part of the union)

Definition at line **183** of file **PGFtypes.h**.

Member Data Documentation

UINT16 ROIBlockHeader::RBH::bufferSize

number of uncoded UINT32 values in a block

Definition at line **188** of file **PGFtypes.h**.

UINT16 ROIBlockHeader::RBH::tileEnd

1: last part of a tile

Definition at line **189** of file **PGFtypes.h**.

The documentation for this struct was generated from the following file:

- **PGFtypes.h**

ROIBlockHeader Union Reference

Block header used with ROI coding scheme

`#include <PGFtypes.h>`

Classes

- struct **RBH**
  Named ROI block header (part of the union)

Public Member Functions

- **ROIBlockHeader** (UINT16 v)
- **ROIBlockHeader** (UINT32 size, bool end)

Public Attributes

- UINT16 **val**
- struct **ROIBlockHeader::RBH rbh**
  ROI block header.

Detailed Description

Block header used with ROI coding scheme

ROI block header is used with ROI coding scheme. It contains block size and tile end flag

Author

C. Stamm

Definition at line **180** of file **PGFtypes.h**.

Constructor & Destructor Documentation

ROIBlockHeader::ROIBlockHeader (UINT16 *v*)`[inline]`

Constructor

Parameters

v	Buffer size

Definition at line **196** of file **PGFtypes.h**.

```
00196 { val = v; }
```

ROIBlockHeader::ROIBlockHeader (UINT32 *size*, bool *end*)`[inline]`

Constructor

Parameters

size	Buffer size
end	0/1 Flag; 1: last part of a tile

Definition at line **201** of file **PGFtypes.h**.

```
00201 { ASSERT(size < (1 << RLblockSizeLen)); rbh.bufferSize = size; rbh.tileEnd = end;
}
```

Member Data Documentation

struct ROIBlockHeader::RBH ROIBlockHeader::rbh

ROI block header.

UINT16 ROIBlockHeader::val

unstructured union value

Definition at line **181** of file **PGFtypes.h**.

The documentation for this union was generated from the following file:

- **PGFtypes.h**

File Documentation

PGFimage.h File Reference

PGF image class.
`#include "PGFstream.h"`

Classes

- class **CPGFImage**
  PGF main class.

Detailed Description

PGF image class.

Author

C. Stamm

Definition in file **PGFimage.h**.

PGFimage.h

Go to the documentation of this file.
```
00001 /*
00002  * The Progressive Graphics File; http://www.libpgf.org
00003  *
00004  * $Date: 2007-02-03 13:04:21 +0100 (Sa, 03 Feb 2007) $
00005  * $Revision: 280 $
00006  *
00007  * This file Copyright (C) 2006 xeraina GmbH, Switzerland
00008  *
00009  * This program is free software; you can redistribute it and/or
00010  * modify it under the terms of the GNU LESSER GENERAL PUBLIC LICENSE
00011  * as published by the Free Software Foundation; either version 2.1
00012  * of the License, or (at your option) any later version.
00013  *
00014  * This program is distributed in the hope that it will be useful,
00015  * but WITHOUT ANY WARRANTY; without even the implied warranty of
00016  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00017  * GNU General Public License for more details.
00018  *
00019  * You should have received a copy of the GNU General Public License
00020  * along with this program; if not, write to the Free Software
00021  * Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
00022  */
00023
00028
00029 #ifndef PGF_PGFIMAGE_H
00030 #define PGF_PGFIMAGE_H
00031
00032 #include "PGFstream.h"
00033
00035 // prototypes
00036 class CDecoder;
00037 class CEncoder;
00038 class CWaveletTransform;
00039
00053 class CPGFImage {
00054 public:
00055
00058         CPGFImage();
00059
00062         virtual ~CPGFImage();
00063
00065         // Destroy internal data structures. Object state after this is the same as
after CPGFImage().
00066         void Destroy();
00067
00073         void Open(CPGFStream* stream);
00074
00077         bool IsOpen() const     { return m_decoder != nullptr; }
00078
00091         void Read(int level = 0, CallbackPtr cb = nullptr, void *data = nullptr);
00092
00093 #ifdef __PGFROISUPPORT__
00103         void Read(PGFRect& rect, int level = 0, CallbackPtr cb = nullptr, void *data
= nullptr);
00104 #endif
00105
00111         void ReadPreview()
{ Read(Levels() - 1); }
00112
00118         void Reconstruct(int level = 0);
00119
00137         void GetBitmap(int pitch, UINT8* buff, BYTE bpp, int channelMap[] = nullptr,
CallbackPtr cb = nullptr, void *data = nullptr) const; // throws IOException
00138
00154         void GetYUV(int pitch, DataT* buff, BYTE bpp, int channelMap[] = nullptr,
CallbackPtr cb = nullptr, void *data = nullptr) const; // throws IOException
00155
00172         void ImportBitmap(int pitch, UINT8 *buff, BYTE bpp, int channelMap[] =
nullptr, CallbackPtr cb = nullptr, void *data = nullptr);
00173
00189         void ImportYUV(int pitch, DataT *buff, BYTE bpp, int channelMap[] = nullptr,
CallbackPtr cb = nullptr, void *data = nullptr);
00190
```

```
00204         void Write(CPGFStream* stream, UINT32* nWrittenBytes = nullptr, CallbackPtr
cb = nullptr, void *data = nullptr);
00205
00213         UINT32 WriteHeader(CPGFStream* stream);
00214
00225         UINT32 WriteImage(CPGFStream* stream, CallbackPtr cb = nullptr, void *data
= nullptr);
00226
00227 #ifdef __PGFROISUPPORT__
00243         UINT32 Write(int level, CallbackPtr cb = nullptr, void *data = nullptr);
00244 #endif
00245
00250         void ConfigureEncoder(bool useOMP = true, bool favorSpeedOverSize = false)
{ m_useOMPinEncoder = useOMP; m_favorSpeedOverSize = favorSpeedOverSize; }
00251
00260         void ConfigureDecoder(bool useOMP = true, UserdataPolicy policy =
UP_CacheAll, UINT32 prefixSize = 0) { ASSERT(prefixSize <= MaxUserDataSize);
m_useOMPinDecoder = useOMP; m_userDataPolicy = (UP_CachePrefix) ? prefixSize : 0xFFFFFFFF
- policy; }
00261
00266         void ResetStreamPos(bool startOfData);
00267
00272         void SetChannel(DataT* channel, int c = 0)
{ ASSERT(c >= 0 && c < MaxChannels); m_channel[c] = channel; }
00273
00282         void SetHeader(const PGFHeader& header, BYTE flags = 0, const UINT8* userData
= 0, UINT32 userDataLength = 0); // throws IOException
00283
00288         void SetMaxValue(UINT32 maxValue);
00289
00296         void SetProgressMode(ProgressMode pm)
{ m_progressMode = pm; }
00297
00303         void SetRefreshCallback(RefreshCB callback, void* arg)                 {
m_cb = callback; m_cbArg = arg; }
00304
00311         void SetColorTable(UINT32 iFirstColor, UINT32 nColors, const RGBQUAD*
prgbColors);
00312
00317         DataT* GetChannel(int c = 0)
{ ASSERT(c >= 0 && c < MaxChannels); return m_channel[c]; }
00318
00325         void GetColorTable(UINT32 iFirstColor, UINT32 nColors, RGBQUAD* prgbColors)
const;
00326
00328         // Returns address of internal color table
00330         const RGBQUAD* GetColorTable() const
{ return m_postHeader.clut; }
00331
00335         const PGFHeader* GetHeader() const
{ return &m_header; }
00336
00341         UINT32 GetMaxValue() const
{ return (1 << m_header.usedBitsPerChannel) - 1; }
00342
00346         UINT64 GetUserDataPos() const
{ return m_userDataPos; }
00347
00354         const UINT8* GetUserData(UINT32& cachedSize, UINT32* pTotalSize = nullptr)
const;
00355
00360         UINT32 GetEncodedHeaderLength() const;
00361
00367         UINT32 GetEncodedLevelLength(int level) const
{ ASSERT(level >= 0 && level < m_header.nLevels); return m_levelLength[m_header.nLevels
- level - 1]; }
00368
00376         UINT32 ReadEncodedHeader(UINT8* target, UINT32 targetLen) const;
00377
00387         UINT32 ReadEncodedData(int level, UINT8* target, UINT32 targetLen) const;
00388
00394         UINT32 ChannelWidth(int c = 0) const
{ ASSERT(c >= 0 && c < MaxChannels); return m_width[c]; }
00395
00401         UINT32 ChannelHeight(int c = 0) const
{ ASSERT(c >= 0 && c < MaxChannels); return m_height[c]; }
00402
```

```
00406         BYTE ChannelDepth() const
{ return MaxChannelDepth(m_preHeader.version); }
00407
00413         UINT32 Width(int level = 0) const
{ ASSERT(level >= 0); return LevelSizeL(m_header.width, level); }
00414
00420         UINT32 Height(int level = 0) const
{ ASSERT(level >= 0); return LevelSizeL(m_header.height, level); }
00421
00427         BYTE Level() const
{ return (BYTE)m_currentLevel; }
00428
00432         BYTE Levels() const
{ return m_header.nLevels; }
00433
00436         bool IsFullyRead() const
{ return m_currentLevel == 0; }
00437
00442         BYTE Quality() const
{ return m_header.quality; }
00443
00448         BYTE Channels() const
{ return m_header.channels; }
00449
00455         BYTE Mode() const
{ return m_header.mode; }
00456
00461         BYTE BPP() const
{ return m_header.bpp; }
00462
00466         bool ROIisSupported() const
{ return (m_preHeader.version & PGFROI) == PGFROI; }
00467
00468 #ifdef __PGFROISUPPORT__
00472         PGFRect ComputeLevelROI() const;
00473 #endif
00474
00479         BYTE UsedBitsPerChannel() const;
00480
00484         BYTE Version() const
{ BYTE ver = CodecMajorVersion(m_preHeader.version); return (ver <= 7) ? ver :
(BYTE)m_header.version.major; }
00485
00486         //class methods
00487
00492         static bool ImportIsSupported(BYTE mode);
00493
00499         static UINT32 LevelSizeL(UINT32 size, int level)
{ ASSERT(level >= 0); UINT32 d = 1 << level; return (size + d - 1) >> level; }
00500
00506         static UINT32 LevelSizeH(UINT32 size, int level)
{ ASSERT(level >= 0); UINT32 d = 1 << (level - 1); return (size + d - 1) >> level; }
00507
00512         static BYTE CodecMajorVersion(BYTE version = PGFVersion);
00513
00518         static BYTE MaxChannelDepth(BYTE version = PGFVersion)                  {
return (version & PGF32) ? 32 : 16; }
00519
00520 protected:
00521         CWaveletTransform* m_wtChannel[MaxChannels];
00522         DataT* m_channel[MaxChannels];
00523         CDecoder* m_decoder;
00524         CEncoder* m_encoder;
00525         UINT32* m_levelLength;
00526         UINT32 m_width[MaxChannels];
00527         UINT32 m_height[MaxChannels];
00528         PGFPreHeader m_preHeader;
00529         PGFHeader m_header;
00530         PGFPostHeader m_postHeader;
00531         UINT64 m_userDataPos;
00532         int m_currentLevel;
00533         UINT32 m_userDataPolicy;
00534         BYTE m_quant;
00535         bool m_downsample;
00536         bool m_favorSpeedOverSize;
00537         bool m_useOMPinEncoder;
00538         bool m_useOMPinDecoder;
```

```
00539 #ifdef __PGFROISUPPORT__
00540         bool m_streamReinitialized;
00541         PGFRect m_roi;
00542 #endif
00543
00544 private:
00545         RefreshCB m_cb;
00546         void *m_cbArg;
00547         double m_percent;
00548         ProgressMode m_progressMode;
00549
00550         void Init();
00551         void ComputeLevels();
00552         bool CompleteHeader();
00553         void RgbToYuv(int pitch, UINT8* rgbBuff, BYTE bpp, int channelMap[],
CallbackPtr cb, void *data);
00554         void Downsample(int nChannel);
00555         UINT32 UpdatePostHeaderSize();
00556         void WriteLevel();
00557
00558 #ifdef __PGFROISUPPORT__
00559         PGFRect GetAlignedROI(int c = 0) const;
00560         void SetROI(PGFRect rect);
00561 #endif
00562
00563         UINT8 Clamp4(DataT v) const {
00564                 if (v & 0xFFFFFFF0) return (v < 0) ? (UINT8)0: (UINT8)15; else return
(UINT8)v;
00565         }
00566         UINT16 Clamp6(DataT v) const {
00567                 if (v & 0xFFFFFFC0) return (v < 0) ? (UINT16)0: (UINT16)63; else
return (UINT16)v;
00568         }
00569         UINT8 Clamp8(DataT v) const {
00570                 // needs only one test in the normal case
00571                 if (v & 0xFFFFFF00) return (v < 0) ? (UINT8)0 : (UINT8)255; else
return (UINT8)v;
00572         }
00573         UINT16 Clamp16(DataT v) const {
00574                 if (v & 0xFFFF0000) return (v < 0) ? (UINT16)0: (UINT16)65535; else
return (UINT16)v;
00575         }
00576         UINT32 Clamp31(DataT v) const {
00577                 return (v < 0) ? 0 : (UINT32)v;
00578         }
00579 };
00580
00581 #endif //PGF_PGFIMAGE_H
```

PGFplatform.h File Reference

PGF platform specific definitions.

```
#include <cassert>
#include <cmath>
#include <cstdlib>
```

Macros

- #define **__PGFROISUPPORT__**
- #define **__PGF32SUPPORT__**
- #define **WordWidth** 32
  *WordBytes*8.*
- #define **WordWidthLog** 5
  ld of WordWidth
- #define **WordMask** 0xFFFFFFE0
  least WordWidthLog bits are zero
- #define **WordBytes** 4
  sizeof(UINT32)
- #define **WordBytesMask** 0xFFFFFFFC
  least WordBytesLog bits are zero
- #define **WordBytesLog** 2
  ld of WordBytes
- #define **DWWIDTHBITS**(bits) (((bits) + **WordWidth** - 1) & **WordMask**)
  aligns scanline width in bits to DWORD value
- #define **DWWIDTH**(bytes) (((bytes) + **WordBytes** - 1) & **WordBytesMask**)
  aligns scanline width in bytes to DWORD value
- #define **DWWIDTHREST**(bytes) ((**WordBytes** - (bytes)%**WordBytes**)%**WordBytes**)
  ***DWWIDTH(bytes)** - bytes.*
- #define **__min**(x, y) ((x) <= (y) ? (x) : (y))
- #define **__max**(x, y) ((x) >= (y) ? (x) : (y))
- #define **ImageModeBitmap** 0
- #define **ImageModeGrayScale** 1
- #define **ImageModeIndexedColor** 2
- #define **ImageModeRGBColor** 3
- #define **ImageModeCMYKColor** 4
- #define **ImageModeHSLColor** 5
- #define **ImageModeHSBColor** 6
- #define **ImageModeMultichannel** 7
- #define **ImageModeDuotone** 8
- #define **ImageModeLabColor** 9

- #define **ImageModeGray16**   10
- #define **ImageModeRGB48**   11
- #define **ImageModeLab48**   12
- #define **ImageModeCMYK64**   13
- #define **ImageModeDeepMultichannel**   14
- #define **ImageModeDuotone16**   15
- #define **ImageModeRGBA**   17
- #define **ImageModeGray32**   18
- #define **ImageModeRGB12**   19
- #define **ImageModeRGB16**   20
- #define **ImageModeUnknown**   255
- #define **__VAL**(x)   (x)

Detailed Description

PGF platform specific definitions.

Author

C. Stamm

Definition in file **PGFplatform.h**.

Macro Definition Documentation

#define __max(x, y)   ((x) >= (y) ? (x) : (y))

Definition at line **92** of file **PGFplatform.h**.

#define __min(x, y)   ((x) <= (y) ? (x) : (y))

Definition at line **91** of file **PGFplatform.h**.

#define __PGF32SUPPORT__

Definition at line **67** of file **PGFplatform.h**.

#define __PGFROISUPPORT__

Definition at line **60** of file **PGFplatform.h**.

#define __VAL(x)   (x)

Definition at line **603** of file **PGFplatform.h**.

#define DWWIDTH(bytes)   (((bytes) + WordBytes - 1) & WordBytesMask)

aligns scanline width in bytes to DWORD value

Definition at line **84** of file **PGFplatform.h**.

#define DWWIDTHBITS(bits)   (((bits) + WordWidth - 1) & WordMask)

aligns scanline width in bits to DWORD value

Definition at line **83** of file **PGFplatform.h**.

#define DWWIDTHREST(bytes)   ((WordBytes - (bytes)%WordBytes)%WordBytes)

DWWIDTH(bytes) - bytes.

Definition at line **85** of file **PGFplatform.h**.

#define ImageModeBitmap   0

Definition at line **98** of file **PGFplatform.h**.

#define ImageModeCMYK64   13

Definition at line **111** of file **PGFplatform.h**.

#define ImageModeCMYKColor   4

Definition at line **102** of file **PGFplatform.h**.

#define ImageModeDeepMultichannel   14

Definition at line **112** of file **PGFplatform.h**.

#define ImageModeDuotone   8

Definition at line **106** of file **PGFplatform.h**.

#define ImageModeDuotone16   15

Definition at line **113** of file **PGFplatform.h**.

#define ImageModeGray16   10

Definition at line **108** of file **PGFplatform.h**.

#define ImageModeGray32   18

Definition at line **116** of file **PGFplatform.h**.

#define ImageModeGrayScale   1

Definition at line **99** of file **PGFplatform.h**.

#define ImageModeHSBColor 6

Definition at line **104** of file **PGFplatform.h**.

#define ImageModeHSLColor 5

Definition at line **103** of file **PGFplatform.h**.

#define ImageModeIndexedColor 2

Definition at line **100** of file **PGFplatform.h**.

#define ImageModeLab48 12

Definition at line **110** of file **PGFplatform.h**.

#define ImageModeLabColor 9

Definition at line **107** of file **PGFplatform.h**.

#define ImageModeMultichannel 7

Definition at line **105** of file **PGFplatform.h**.

#define ImageModeRGB12 19

Definition at line **117** of file **PGFplatform.h**.

#define ImageModeRGB16 20

Definition at line **118** of file **PGFplatform.h**.

#define ImageModeRGB48 11

Definition at line **109** of file **PGFplatform.h**.

#define ImageModeRGBA 17

Definition at line **115** of file **PGFplatform.h**.

#define ImageModeRGBColor 3

Definition at line **101** of file **PGFplatform.h**.

#define ImageModeUnknown 255

Definition at line **119** of file **PGFplatform.h**.

#define WordBytes 4

sizeof(UINT32)

Definition at line **76** of file **PGFplatform.h**.

#define WordBytesLog 2

ld of WordBytes

Definition at line **78** of file **PGFplatform.h**.

#define WordBytesMask 0xFFFFFFFC

least WordBytesLog bits are zero

Definition at line **77** of file **PGFplatform.h**.

#define WordMask 0xFFFFFFE0

least WordWidthLog bits are zero

Definition at line **75** of file **PGFplatform.h**.

#define WordWidth 32

WordBytes*8.

Definition at line **73** of file **PGFplatform.h**.

#define WordWidthLog 5

ld of WordWidth

Definition at line **74** of file **PGFplatform.h**.

PGFplatform.h

```
Go to the documentation of this file.00001 /*
00002  * The Progressive Graphics File; http://www.libpgf.org
00003  *
00004  * $Date: 2007-06-12 19:27:47 +0200 (Di, 12 Jun 2007) $
00005  * $Revision: 307 $
00006  *
00007  * This file Copyright (C) 2006 xeraina GmbH, Switzerland
00008  *
00009  * This program is free software; you can redistribute it and/or
00010  * modify it under the terms of the GNU LESSER GENERAL PUBLIC LICENSE
00011  * as published by the Free Software Foundation; either version 2.1
00012  * of the License, or (at your option) any later version.
00013  *
00014  * This program is distributed in the hope that it will be useful,
00015  * but WITHOUT ANY WARRANTY; without even the implied warranty of
00016  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00017  * GNU General Public License for more details.
00018  *
00019  * You should have received a copy of the GNU General Public License
00020  * along with this program; if not, write to the Free Software
00021  * Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
00022  */
00023
00028
00029 #ifndef PGF_PGFPLATFORM_H
00030 #define PGF_PGFPLATFORM_H
00031
00032 #include <cassert>
00033 #include <cmath>
00034 #include <cstdlib>
00035
00036
//-------------------------------------------------------------------------------
00037 // Endianess detection taken from lcms2 header.
00038 // This list can be endless, so only some checks are performed over here.
00039
//-------------------------------------------------------------------------------
00040 #if defined(_HOST_BIG_ENDIAN) || defined(__BIG_ENDIAN__) ||
defined(WORDS_BIGENDIAN)
00041 #define PGF_USE_BIG_ENDIAN 1
00042 #endif
00043
00044 #if defined(__sgi__) || defined(__sgi) || defined(__powerpc__) || defined(__sparc)
|| defined(__sparc__)
00045 #define PGF_USE_BIG_ENDIAN 1
00046 #endif
00047
00048 #if defined(__ppc__) || defined(__s390__) || defined(__s390x__)
00049 #define PGF_USE_BIG_ENDIAN 1
00050 #endif
00051
00052 #ifdef TARGET_CPU_PPC
00053 #define PGF_USE_BIG_ENDIAN 1
00054 #endif
00055
00056
//-------------------------------------------------------------------------------
00057 // ROI support
00058
//-------------------------------------------------------------------------------
00059 #ifndef NPGFROI
00060 #define __PGFROISUPPORT__ // without ROI support the program code gets simpler and
smaller
00061 #endif
00062
00063
//-------------------------------------------------------------------------------
00064 // 32 bit per channel support
00065
//-------------------------------------------------------------------------------
00066 #ifndef NPGF32
00067 #define __PGF32SUPPORT__ // without 32 bit the memory consumption during encoding
and decoding is much lesser
```

```
00068 #endif
00069
00070
//----------------------------------------------------------------------------------
00071 //      32 Bit platform constants
00072
//----------------------------------------------------------------------------------
00073 #define WordWidth                       32
00074 #define WordWidthLog            5
00075 #define WordMask                        0xFFFFFFE0
00076 #define WordBytes                       4
00077 #define WordBytesMask           0xFFFFFFFC
00078 #define WordBytesLog            2
00079
00080
//----------------------------------------------------------------------------------
00081 // Alignment macros (used in PGF based libraries)
00082
//----------------------------------------------------------------------------------
00083 #define DWWIDTHBITS(bits)       (((bits) + WordWidth - 1) & WordMask)
00084 #define DWWIDTH(bytes)          (((bytes) + WordBytes - 1) & WordBytesMask)
00085 #define DWWIDTHREST(bytes)      ((WordBytes - (bytes)%WordBytes)%WordBytes)
00086
00087
//----------------------------------------------------------------------------------
00088 // Min-Max macros
00089
//----------------------------------------------------------------------------------
00090 #ifndef __min
00091         #define __min(x, y)             ((x) <= (y) ? (x) : (y))
00092         #define __max(x, y)             ((x) >= (y) ? (x) : (y))
00093 #endif // __min
00094
00095
//----------------------------------------------------------------------------------
00096 //      Defines -- Adobe image modes.
00097
//----------------------------------------------------------------------------------
00098 #define ImageModeBitmap                         0
00099 #define ImageModeGrayScale                      1
00100 #define ImageModeIndexedColor           2
00101 #define ImageModeRGBColor                       3
00102 #define ImageModeCMYKColor                      4
00103 #define ImageModeHSLColor                       5
00104 #define ImageModeHSBColor                       6
00105 #define ImageModeMultichannel           7
00106 #define ImageModeDuotone                        8
00107 #define ImageModeLabColor                       9
00108 #define ImageModeGray16                         10              // 565
00109 #define ImageModeRGB48                          11
00110 #define ImageModeLab48                          12
00111 #define ImageModeCMYK64                         13
00112 #define ImageModeDeepMultichannel       14
00113 #define ImageModeDuotone16                      15
00114 // pgf extension
00115 #define ImageModeRGBA                           17
00116 #define ImageModeGray32                         18              // MSB is 0 (can be
interpreted as signed 15.16 fixed point format)
00117 #define ImageModeRGB12                          19
00118 #define ImageModeRGB16                          20
00119 #define ImageModeUnknown                        255
00120
00121
00122
//----------------------------------------------------------------------------------
00123 // WINDOWS
00124
//----------------------------------------------------------------------------------
00125 #if defined(WIN32) || defined(WINCE) || defined(WIN64)
00126 #define VC_EXTRALEAN            // Exclude rarely-used stuff from Windows headers
00127
00128
//----------------------------------------------------------------------------------
00129 // MFC
00130
//----------------------------------------------------------------------------------
00131 #ifdef _MFC_VER
```

```
00132 #ifndef _WIN32_WINNT            // Specifies that the minimum required platform is
Windows Vista.
00133 #define _WIN32_WINNT 0x0600     // Change this to the appropriate value to target
other versions of Windows.
00134 #endif
00135 #include <afx.h>
00136 #include <afxwin.h>         // MFC core and standard components
00137 #include <afxext.h>         // MFC extensions
00138 #include <afxdtctl.h>           // MFC support for Internet Explorer 4 Common
Controls
00139 #ifndef _AFX_NO_AFXCMN_SUPPORT
00140 #include <afxcmn.h>                     // MFC support for Windows Common Controls
00141 #endif // _AFX_NO_AFXCMN_SUPPORT
00142
00143 #else
00144
00145 #include <windows.h>
00146 #include <ole2.h>
00147
00148 #endif // _MFC_VER
00149
//--------------------------------------------------------------------------------
00150
00151 #define DllExport   __declspec( dllexport )
00152
00153
//--------------------------------------------------------------------------------
00154 // unsigned number type definitions
00155
//--------------------------------------------------------------------------------
00156 typedef unsigned char           UINT8;
00157 typedef unsigned char           BYTE;
00158 typedef unsigned short          UINT16;
00159 typedef unsigned short      WORD;
00160 typedef unsigned int            UINT32;
00161 typedef unsigned long       DWORD;
00162 typedef unsigned long       ULONG;
00163 typedef unsigned __int64        UINT64;
00164 typedef unsigned __int64        ULONGLONG;
00165
00166
//--------------------------------------------------------------------------------
00167 // signed number type definitions
00168
//--------------------------------------------------------------------------------
00169 typedef signed char                     INT8;
00170 typedef signed short            INT16;
00171 typedef signed int                      INT32;
00172 typedef signed int                      BOOL;
00173 typedef signed long                     LONG;
00174 typedef signed __int64          INT64;
00175 typedef signed __int64          LONGLONG;
00176
00177
//--------------------------------------------------------------------------------
00178 // other types
00179
//--------------------------------------------------------------------------------
00180 typedef int OSError;
00181 typedef bool (__cdecl *CallbackPtr)(double percent, bool escapeAllowed, void *data);
00182
00183
//--------------------------------------------------------------------------------
00184 // struct type definitions
00185
//--------------------------------------------------------------------------------
00186
00187
//--------------------------------------------------------------------------------
00188 // DEBUG macros
00189
//--------------------------------------------------------------------------------
00190 #ifndef ASSERT
00191         #ifdef _DEBUG
00192                 #define ASSERT(x)       assert(x)
00193         #else
00194                 #if defined(__GNUC__)
```

```
00195                         #define ASSERT(ignore)((void) 0)
00196                 #elif _MSC_VER >= 1300
00197                         #define ASSERT          __noop
00198                 #else
00199                         #define ASSERT ((void)0)
00200                 #endif
00201         #endif //_DEBUG
00202 #endif //ASSERT
00203
00204
//-------------------------------------------------------------------------------
00205 // Exception handling macros
00206
//-------------------------------------------------------------------------------
00207 #ifdef NEXCEPTIONS
00208         extern OSError _PGF_Error_;
00209         extern OSError GetLastPGFError();
00210
00211         #define ReturnWithError(err) { _PGF_Error_ = err; return; }
00212         #define ReturnWithError2(err, ret) { _PGF_Error_ = err; return ret; }
00213 #else
00214         #define ReturnWithError(err) throw IOException(err)
00215         #define ReturnWithError2(err, ret) throw IOException(err)
00216 #endif //NEXCEPTIONS
00217
00218
//-------------------------------------------------------------------------------
00219 // constants
00220
//-------------------------------------------------------------------------------
00221 #define FSFromStart             FILE_BEGIN                              // 0
00222 #define FSFromCurrent   FILE_CURRENT                    // 1
00223 #define FSFromEnd               FILE_END                                // 2
00224
00225 #define INVALID_SET_FILE_POINTER ((DWORD)-1)
00226
00227
//-------------------------------------------------------------------------------
00228 // IO Error constants
00229
//-------------------------------------------------------------------------------
00230 #define NoError                         ERROR_SUCCESS
00231 #define AppError                        0x20000000
00232 #define InsufficientMemory      0x20000001
00233 #define InvalidStreamPos        0x20000002
00234 #define EscapePressed           0x20000003
00235 #define WrongVersion            0x20000004
00236 #define FormatCannotRead        0x20000005
00237 #define ImageTooSmall           0x20000006
00238 #define ZlibError                       0x20000007
00239 #define ColorTableError         0x20000008
00240 #define PNGError                        0x20000009
00241 #define MissingData                     0x2000000A
00242
00243
//-------------------------------------------------------------------------------
00244 // methods
00245
//-------------------------------------------------------------------------------
00246 inline OSError FileRead(HANDLE hFile, int *count, void *buffPtr) {
00247         if (ReadFile(hFile, buffPtr, *count, (ULONG *)count, nullptr)) {
00248                 return NoError;
00249         } else {
00250                 return GetLastError();
00251         }
00252 }
00253
00254 inline OSError FileWrite(HANDLE hFile, int *count, void *buffPtr) {
00255         if (WriteFile(hFile, buffPtr, *count, (ULONG *)count, nullptr)) {
00256                 return NoError;
00257         } else {
00258                 return GetLastError();
00259         }
00260 }
00261
00262 inline OSError GetFPos(HANDLE hFile, UINT64 *pos) {
00263 #ifdef WINCE
```

```
00264         LARGE_INTEGER li;
00265         li.QuadPart = 0;
00266
00267         li.LowPart = SetFilePointer (hFile, li.LowPart, &li.HighPart,
FILE_CURRENT);
00268         if (li.LowPart == INVALID_SET_FILE_POINTER) {
00269                 OSError err = GetLastError();
00270                 if (err != NoError) {
00271                         return err;
00272                 }
00273         }
00274         *pos = li.QuadPart;
00275         return NoError;
00276 #else
00277         LARGE_INTEGER li;
00278         li.QuadPart = 0;
00279         if (SetFilePointerEx(hFile, li, (PLARGE_INTEGER)pos, FILE_CURRENT)) {
00280                 return NoError;
00281         } else {
00282                 return GetLastError();
00283         }
00284 #endif
00285 }
00286
00287 inline OSError SetFPos(HANDLE hFile, int posMode, INT64 posOff) {
00288 #ifdef WINCE
00289         LARGE_INTEGER li;
00290         li.QuadPart = posOff;
00291
00292         if (SetFilePointer (hFile, li.LowPart, &li.HighPart, posMode) ==
INVALID_SET_FILE_POINTER) {
00293                 OSError err = GetLastError();
00294                 if (err != NoError) {
00295                         return err;
00296                 }
00297         }
00298         return NoError;
00299 #else
00300         LARGE_INTEGER li;
00301         li.QuadPart = posOff;
00302         if (SetFilePointerEx(hFile, li, nullptr, posMode)) {
00303                 return NoError;
00304         } else {
00305                 return GetLastError();
00306         }
00307 #endif
00308 }
00309 #endif //WIN32
00310
00311
00312
//---------------------------------------------------------------------------------
00313 // Apple OSX
00314
//---------------------------------------------------------------------------------
00315 #ifdef __APPLE__
00316 #define __POSIX__
00317 #endif // __APPLE__
00318
00319
00320
//---------------------------------------------------------------------------------
00321 // LINUX
00322
//---------------------------------------------------------------------------------
00323 #if defined(__linux__) || defined(__GLIBC__)
00324 #define __POSIX__
00325 #endif // __linux__ or __GLIBC__
00326
00327
00328
//---------------------------------------------------------------------------------
00329 // SOLARIS
00330
//---------------------------------------------------------------------------------
00331 #ifdef __sun
00332 #define __POSIX__
```

```
00333 #endif // __sun
00334
00335
00336
//---------------------------------------------------------------------------------
00337 // *BSD
00338
//---------------------------------------------------------------------------------
00339 #if defined(__NetBSD__) || defined(__OpenBSD__) || defined(__FreeBSD__)
00340 #ifndef __POSIX__
00341 #define __POSIX__
00342 #endif
00343
00344 #ifndef off64_t
00345 #define off64_t off_t
00346 #endif
00347
00348 #ifndef lseek64
00349 #define lseek64 lseek
00350 #endif
00351
00352 #endif // __NetBSD__ or __OpenBSD__ or __FreeBSD__
00353
00354
00355
//---------------------------------------------------------------------------------
00356 // POSIX *NIXes
00357
//---------------------------------------------------------------------------------
00358
00359 #ifdef __POSIX__
00360 #include <unistd.h>
00361 #include <errno.h>
00362 #include <stdint.h>             // for int64_t and uint64_t
00363 #include <string.h>             // memcpy()
00364
00365 #undef major
00366
00367
//---------------------------------------------------------------------------------
00368 // unsigned number type definitions
00369
//---------------------------------------------------------------------------------
00370
00371 typedef unsigned char           UINT8;
00372 typedef unsigned char           BYTE;
00373 typedef unsigned short          UINT16;
00374 typedef unsigned short          WORD;
00375 typedef unsigned int            UINT32;
00376 typedef unsigned int            DWORD;
00377 typedef unsigned long           ULONG;
00378 typedef unsigned long long   __Uint64;
00379 typedef __Uint64                        UINT64;
00380 typedef __Uint64                        ULONGLONG;
00381
00382
//---------------------------------------------------------------------------------
00383 // signed number type definitions
00384
//---------------------------------------------------------------------------------
00385 typedef signed char                     INT8;
00386 typedef signed short            INT16;
00387 typedef signed int                      INT32;
00388 typedef signed int                      BOOL;
00389 typedef signed long                     LONG;
00390 typedef int64_t                         INT64;
00391 typedef int64_t                         LONGLONG;
00392
00393
//---------------------------------------------------------------------------------
00394 // other types
00395
//---------------------------------------------------------------------------------
00396 typedef int                                     OSError;
00397 typedef int                                     HANDLE;
00398 typedef unsigned long           ULONG_PTR;
00399 typedef void*                           PVOID;
```

```
00400 typedef char*                          LPTSTR;
00401 typedef bool (*CallbackPtr)(double percent, bool escapeAllowed, void *data);
00402
00403
//----------------------------------------------------------------------------------
00404 // struct type definitions
00405
//----------------------------------------------------------------------------------
00406 typedef struct tagRGBTRIPLE {
00407         BYTE rgbtBlue;
00408         BYTE rgbtGreen;
00409         BYTE rgbtRed;
00410 } RGBTRIPLE;
00411
00412 typedef struct tagRGBQUAD {
00413         BYTE rgbBlue;
00414         BYTE rgbGreen;
00415         BYTE rgbRed;
00416         BYTE rgbReserved;
00417 } RGBQUAD;
00418
00419 typedef union _LARGE_INTEGER {
00420   struct {
00421     DWORD LowPart;
00422     LONG HighPart;
00423   } u;
00424   LONGLONG QuadPart;
00425 } LARGE_INTEGER, *PLARGE_INTEGER;
00426 #endif // __POSIX__
00427
00428
00429 #if defined(__POSIX__) || defined(WINCE)
00430 // CMYK macros
00431 #define GetKValue(cmyk)      ((BYTE)(cmyk))
00432 #define GetYValue(cmyk)      ((BYTE)((cmyk)>> 8))
00433 #define GetMValue(cmyk)      ((BYTE)((cmyk)>>16))
00434 #define GetCValue(cmyk)      ((BYTE)((cmyk)>>24))
00435 #define CMYK(c,m,y,k)
((COLORREF)((((BYTE)(k)|((WORD)((BYTE)(y))<<8))|(((DWORD)(BYTE)(m))<<16))|(((DWORD)(BY
TE)(c))<<24)))
00436
00437
//----------------------------------------------------------------------------------
00438 // methods
00439
//----------------------------------------------------------------------------------
00440 /* The MulDiv function multiplies two 32-bit values and then divides the 64-bit
00441  * result by a third 32-bit value. The return value is rounded up or down to
00442  * the nearest integer.
00443  *
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winprog/winprog/muldi
v.asp
00444  * */
00445 __inline int MulDiv(int nNumber, int nNumerator, int nDenominator) {
00446         INT64 multRes = nNumber*nNumerator;
00447         INT32 divRes = INT32(multRes/nDenominator);
00448         return divRes;
00449 }
00450 #endif // __POSIX__ or WINCE
00451
00452
00453 #ifdef __POSIX__
00454
//----------------------------------------------------------------------------------
00455 // DEBUG macros
00456
//----------------------------------------------------------------------------------
00457 #ifndef ASSERT
00458         #ifdef _DEBUG
00459                 #define ASSERT(x)       assert(x)
00460         #else
00461                 #define ASSERT(x)
00462         #endif //_DEBUG
00463 #endif //ASSERT
00464
00465
//----------------------------------------------------------------------------------
```

```
00466 // Exception handling macros
00467
//-------------------------------------------------------------------------------
00468 #ifdef NEXCEPTIONS
00469         extern OSError _PGF_Error_;
00470         extern OSError GetLastPGFError();
00471
00472         #define ReturnWithError(err) { _PGF_Error_ = err; return; }
00473         #define ReturnWithError2(err, ret) { _PGF_Error_ = err; return ret; }
00474 #else
00475         #define ReturnWithError(err) throw IOException(err)
00476         #define ReturnWithError2(err, ret) throw IOException(err)
00477 #endif //NEXCEPTIONS
00478
00479 #define THROW_ throw(IOException)
00480 #define CONST const
00481
00482
//-------------------------------------------------------------------------------
00483 // constants
00484
//-------------------------------------------------------------------------------
00485 #define FSFromStart                     SEEK_SET
00486 #define FSFromCurrent           SEEK_CUR
00487 #define FSFromEnd                       SEEK_END
00488 #if defined(__cplusplus) && __cplusplus < 201103L
00489         #define nullptr                 NULL
00490 #endif
00491
00492
//-------------------------------------------------------------------------------
00493 // IO Error constants
00494
//-------------------------------------------------------------------------------
00495 #define NoError                                 0x0000
00496 #define AppError                                0x2000
00497 #define InsufficientMemory              0x2001
00498 #define InvalidStreamPos                0x2002
00499 #define EscapePressed                   0x2003
00500 #define WrongVersion                    0x2004
00501 #define FormatCannotRead                0x2005
00502 #define ImageTooSmall                   0x2006
00503 #define ZlibError                               0x2007
00504 #define ColorTableError                 0x2008
00505 #define PNGError                                0x2009
00506 #define MissingData                             0x200A
00507
00508
//-------------------------------------------------------------------------------
00509 // methods
00510
//-------------------------------------------------------------------------------
00511 __inline OSError FileRead(HANDLE hFile, int *count, void *buffPtr) {
00512         *count = (int)read(hFile, buffPtr, *count);
00513         if (*count != -1) {
00514                 return NoError;
00515         } else {
00516                 return errno;
00517         }
00518 }
00519
00520 __inline OSError FileWrite(HANDLE hFile, int *count, void *buffPtr) {
00521         *count = (int)write(hFile, buffPtr, (size_t)*count);
00522         if (*count != -1) {
00523                 return NoError;
00524         } else {
00525                 return errno;
00526         }
00527 }
00528
00529 __inline OSError GetFPos(HANDLE hFile, UINT64 *pos) {
00530         #ifdef __APPLE__
00531                 off_t ret;
00532                 if ((ret = lseek(hFile, 0, SEEK_CUR)) == -1) {
00533                         return errno;
00534                 } else {
00535                         *pos = (UINT64)ret;
```

```
00536                         return NoError;
00537                 }
00538         #else
00539                 off64_t ret;
00540                 if ((ret = lseek64(hFile, 0, SEEK_CUR)) == -1) {
00541                         return errno;
00542                 } else {
00543                         *pos = (UINT64)ret;
00544                         return NoError;
00545                 }
00546         #endif
00547 }
00548
00549 __inline OSError SetFPos(HANDLE hFile, int posMode, INT64 posOff) {
00550         #ifdef __APPLE__
00551                 if ((lseek(hFile, (off_t)posOff, posMode)) == -1) {
00552                         return errno;
00553                 } else {
00554                         return NoError;
00555                 }
00556         #else
00557                 if ((lseek64(hFile, (off64_t)posOff, posMode)) == -1) {
00558                         return errno;
00559                 } else {
00560                         return NoError;
00561                 }
00562         #endif
00563 }
00564
00565 #endif /* __POSIX__ */
00566
//-------------------------------------------------------------------------------
00567
00568
00569
//-------------------------------------------------------------------------------
00570 //      Big Endian
00571
//-------------------------------------------------------------------------------
00572 #ifdef PGF_USE_BIG_ENDIAN
00573
00574 #ifndef _lrotl
00575         #define _lrotl(x,n)     (((x) << ((UINT32)(n))) | ((x) >> (32 -
(UINT32)(n))))
00576 #endif
00577
00578 __inline UINT16 ByteSwap(UINT16 wX) {
00579         return ((wX & 0xFF00) >> 8) | ((wX & 0x00FF) << 8);
00580 }
00581
00582 __inline UINT32 ByteSwap(UINT32 dwX) {
00583 #ifdef _X86_
00584         _asm mov eax, dwX
00585         _asm bswap eax
00586         _asm mov dwX, eax
00587         return dwX;
00588 #else
00589         return _lrotl(((dwX & 0xFF00FF00) >> 8) | ((dwX & 0x00FF00FF) << 8), 16);
00590 #endif
00591 }
00592
00593 #if defined(WIN32) || defined(WIN64)
00594 __inline UINT64 ByteSwap(UINT64 ui64) {
00595         return _byteswap_uint64(ui64);
00596 }
00597 #endif
00598
00599 #define __VAL(x) ByteSwap(x)
00600
00601 #else //PGF_USE_BIG_ENDIAN
00602
00603         #define __VAL(x) (x)
00604
00605 #endif //PGF_USE_BIG_ENDIAN
00606
00607 // OpenMP rules (inspired from libraw project)
00608 // NOTE: Use LIBPGF_DISABLE_OPENMP to disable OpenMP support in whole libpgf
```

```
00609 #ifndef LIBPGF_DISABLE_OPENMP
00610 # if defined (_OPENMP)
00611 #  if defined (WIN32) || defined(WIN64)
00612 #   if defined (_MSC_VER) && (_MSC_VER >= 1500)
00613 //    VS2008 SP1 and VS2010+ : OpenMP works OK
00614 #    define LIBPGF_USE_OPENMP
00615 #   elif defined (__INTEL_COMPILER) && (__INTEL_COMPILER >=910)
00616 //    untested on 9.x and 10.x, Intel documentation claims OpenMP 2.5 support in 9.1
00617 #    define LIBPGF_USE_OPENMP
00618 #   else
00619 #    undef LIBPGF_USE_OPENMP
00620 #   endif
00621 //  Not Win32
00622 #  elif (defined(__APPLE__) || defined(__MACOSX__)) && defined(_REENTRANT)
00623 #   undef LIBPGF_USE_OPENMP
00624 #  else
00625 #   define LIBPGF_USE_OPENMP
00626 #  endif
00627 # endif // defined (_OPENMP)
00628 #endif // ifndef LIBPGF_DISABLE_OPENMP
00629 #ifdef LIBPGF_USE_OPENMP
00630 #include <omp.h>
00631 #endif
00632 
00633 #endif //PGF_PGFPLATFORM_H
```

PGFstream.h File Reference

PGF stream class.

```
#include "PGFtypes.h"
#include <new>
```

Classes

- class **CPGFStream**
  Abstract stream base class.
- class **CPGFFileStream**
  File stream class.
- class **CPGFMemoryStream**
  Memory stream class.

Detailed Description

PGF stream class.

Author

C. Stamm

Definition in file **PGFstream.h**.

PGFstream.h

```
Go to the documentation of this file.00001 /*
00002  * The Progressive Graphics File; http://www.libpgf.org
00003  *
00004  * $Date: 2007-06-11 10:56:17 +0200 (Mo, 11 Jun 2007) $
00005  * $Revision: 299 $
00006  *
00007  * This file Copyright (C) 2006 xeraina GmbH, Switzerland
00008  *
00009  * This program is free software; you can redistribute it and/or
00010  * modify it under the terms of the GNU LESSER GENERAL PUBLIC LICENSE
00011  * as published by the Free Software Foundation; either version 2.1
00012  * of the License, or (at your option) any later version.
00013  *
00014  * This program is distributed in the hope that it will be useful,
00015  * but WITHOUT ANY WARRANTY; without even the implied warranty of
00016  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00017  * GNU General Public License for more details.
00018  *
00019  * You should have received a copy of the GNU General Public License
00020  * along with this program; if not, write to the Free Software
00021  * Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
00022  */
00023
00028
00029 #ifndef PGF_STREAM_H
00030 #define PGF_STREAM_H
00031
00032 #include "PGFtypes.h"
00033 #include <new>
00034
00039 class CPGFStream {
00040 public:
00043         CPGFStream() {}
00044
00047         virtual ~CPGFStream() {}
00048
00053         virtual void Write(int *count, void *buffer)=0;
00054
00059         virtual void Read(int *count, void *buffer)=0;
00060
00065         virtual void SetPos(short posMode, INT64 posOff)=0;
00066
00070         virtual UINT64 GetPos() const=0;
00071
00075         virtual bool IsValid() const=0;
00076 };
00077
00082 class CPGFFileStream : public CPGFStream {
00083 protected:
00084         HANDLE m_hFile;
00085
00086 public:
00087         CPGFFileStream() : m_hFile(0) {}
00090         CPGFFileStream(HANDLE hFile) : m_hFile(hFile) {}
00092         HANDLE GetHandle() { return m_hFile; }
00093
00094         virtual ~CPGFFileStream() { m_hFile = 0; }
00095         virtual void Write(int *count, void *buffer); // throws IOException
00096         virtual void Read(int *count, void *buffer); // throws IOException
00097         virtual void SetPos(short posMode, INT64 posOff); // throws IOException
00098         virtual UINT64 GetPos() const; // throws IOException
00099         virtual bool   IsValid() const  { return m_hFile != 0; }
00100 };
00101
00106 class CPGFMemoryStream : public CPGFStream {
00107 protected:
00108         UINT8 *m_buffer, *m_pos;
00109         UINT8 *m_eos;
00110         size_t m_size;
00111         bool   m_allocated;
00112
00113 public:
00116         CPGFMemoryStream(size_t size);
```

```
00117
00121         CPGFMemoryStream(UINT8 *pBuffer, size_t size);
00122
00126         void Reinitialize(UINT8 *pBuffer, size_t size);
00127
00128         virtual ~CPGFMemoryStream() {
00129                 m_pos = 0;
00130                 if (m_allocated) {
00131                         // the memory buffer has been allocated inside of
CPMFmemoryStream constructor
00132                         delete[] m_buffer; m_buffer = 0;
00133                 }
00134         }
00135
00136         virtual void Write(int *count, void *buffer); // throws IOException
00137         virtual void Read(int *count, void *buffer);
00138         virtual void SetPos(short posMode, INT64 posOff); // throws IOException
00139         virtual UINT64 GetPos() const { ASSERT(IsValid()); return m_pos - m_buffer;
}
00140         virtual bool   IsValid() const  { return m_buffer != 0; }
00141
00143         size_t GetSize() const                  { return m_size; }
00145         const UINT8* GetBuffer() const  { return m_buffer; }
00147         UINT8* GetBuffer()                              { return m_buffer; }
00149         UINT64 GetEOS() const                   { ASSERT(IsValid()); return m_eos
- m_buffer; }
00151         void SetEOS(UINT64 length)              { ASSERT(IsValid()); m_eos =
m_buffer + length; }
00152 };
00153
00158 #ifdef _MFC_VER
00159 class CPGFMemFileStream : public CPGFStream {
00160 protected:
00161         CMemFile *m_memFile;
00162 public:
00163         CPGFMemFileStream(CMemFile *memFile) : m_memFile(memFile) {}
00164         virtual bool    IsValid() const { return m_memFile != nullptr; }
00165         virtual ~CPGFMemFileStream() {}
00166         virtual void Write(int *count, void *buffer); // throws IOException
00167         virtual void Read(int *count, void *buffer); // throws IOException
00168         virtual void SetPos(short posMode, INT64 posOff); // throws IOException
00169         virtual UINT64 GetPos() const; // throws IOException
00170 };
00171 #endif
00172
00177 #if defined(WIN32) || defined(WINCE)
00178 class CPGFIStream : public CPGFStream {
00179 protected:
00180         IStream *m_stream;
00181 public:
00182         CPGFIStream(IStream *stream) : m_stream(stream) { m_stream->AddRef(); }
00183         virtual bool IsValid() const    { return m_stream != 0; }
00184         virtual ~CPGFIStream() { m_stream->Release(); }
00185         virtual void Write(int *count, void *buffer); // throws IOException
00186         virtual void Read(int *count, void *buffer); // throws IOException
00187         virtual void SetPos(short posMode, INT64 posOff); // throws IOException
00188         virtual UINT64 GetPos() const; // throws IOException
00189         IStream* GetIStream() const { return m_stream; }
00190 };
00191 #endif
00192
00193 #endif // PGF_STREAM_H
```

PGFtypes.h File Reference

PGF definitions.
`#include "PGFplatform.h"`

Classes

- struct **PGFMagicVersion**
  PGF identification and version.
- struct **PGFPreHeader**
  PGF pre-header.
- struct **PGFVersionNumber**
  version number stored in header since major version 7
- struct **PGFHeader**
  PGF header.
- struct **PGFPostHeader**
  Optional PGF post-header.
- union **ROIBlockHeader**
  Block header used with ROI coding scheme
- struct **ROIBlockHeader::RBH**
  Named ROI block header (part of the union)
- struct **IOException**
  PGF exception.
- struct **PGFRect**
  Rectangle.

Macros

- #define **PGFMajorNumber** 7
- #define **PGFYear** 21
- #define **PGFWeek** 07
- #define **PPCAT_NX**(A, B) A ## B
- #define **PPCAT**(A, B) **PPCAT_NX**(A, B)
- #define **STRINGIZE_NX**(A) #A
- #define **STRINGIZE**(A) **STRINGIZE_NX**(A)
- #define **PGFCodecVersionID** **PPCAT**(**PPCAT**(**PPCAT**(0x0, **PGFMajorNumber**), **PGFYear**), **PGFWeek**)
- #define **PGFCodecVersion** **STRINGIZE**(**PPCAT**(**PPCAT**(**PPCAT**(**PPCAT**(**PGFMajorNumber**, .), **PGFYear**), .), **PGFWeek**))
- #define **PGFMagic** "PGF"
  PGF identification.
- #define **MaxLevel** 30
  maximum number of transform levels
- #define **NSubbands** 4
  number of subbands per level

- #define **MaxChannels** 8
  maximum number of (color) channels

- #define **DownsampleThreshold** 3
  if quality is larger than this threshold than downsampling is used

- #define **ColorTableLen** 256
  size of color lookup table (clut)

- #define **Version2** 2
  data structure ***PGFHeader*** *of major version 2*

- #define **PGF32** 4
  32 bit values are used -> allows at maximum 30 input bits, otherwise 16 bit values are used -> allows at maximum 14 input bits

- #define **PGFROI** 8
  supports Regions Of Interest

- #define **Version5** 16
  new coding scheme since major version 5

- #define **Version6** 32
  hSize in ***PGFPreHeader*** *uses 32 bits instead of 16 bits*

- #define **Version7** 64
  Codec major and minor version number stored in ***PGFHeader****.*

- #define **PGFVersion** (**Version2** | **PGF32** | **Version5** | **Version6** | **Version7**)
  current standard version

- #define **BufferSize** 16384
  must be a multiple of WordWidth, BufferSize <= UINT16_MAX

- #define **RLblockSizeLen** 15
  *block size length (< 16): ld(BufferSize) < RLblockSizeLen <= 2*ld(BufferSize)*

- #define **LinBlockSize** 8
  side length of a coefficient block in a HH or LL subband

- #define **InterBlockSize** 4
  side length of a coefficient block in a HL or LH subband

- #define **MaxBitPlanes** 31
  maximum number of bit planes of m_value: 32 minus sign bit

- #define **MaxBitPlanesLog** 5

number of bits to code the maximum number of bit planes (in 32 or 16 bit mode)

- #define **MaxQuality** **MaxBitPlanes**
  maximum quality

- #define **MagicVersionSize** sizeof(**PGFMagicVersion**)
- #define **PreHeaderSize** sizeof(**PGFPreHeader**)
- #define **HeaderSize** sizeof(**PGFHeader**)
- #define **ColorTableSize** (**ColorTableLen***sizeof(RGBQUAD))
- #define **DataTSize** sizeof(**DataT**)
- #define **MaxUserDataSize** 0x7FFFFFFF

Typedefs

- typedef INT32 **DataT**
- typedef void(* **RefreshCB**) (void *p)

Enumerations

- enum **Orientation** { **LL** = 0, **HL** = 1, **LH** = 2, **HH** = 3 }
- enum **ProgressMode** { **PM_Relative**, **PM_Absolute** }
- enum **UserdataPolicy** { **UP_Skip** = 0, **UP_CachePrefix** = 1, **UP_CacheAll** = 2 }

Detailed Description

PGF definitions.

Author

C. Stamm

Definition in file **PGFtypes.h**.

Macro Definition Documentation

#define BufferSize 16384

must be a multiple of WordWidth, BufferSize <= UINT16_MAX

Definition at line **84** of file **PGFtypes.h**.

#define ColorTableLen 256

size of color lookup table (clut)

Definition at line **66** of file **PGFtypes.h**.

#define ColorTableSize (ColorTableLen*sizeof(RGBQUAD))

Definition at line **282** of file **PGFtypes.h**.

#define DataTSize sizeof(DataT)

Definition at line **283** of file **PGFtypes.h**.

#define DownsampleThreshold 3

if quality is larger than this threshold than downsampling is used

Definition at line **65** of file **PGFtypes.h**.

#define HeaderSize sizeof(PGFHeader)

Definition at line **281** of file **PGFtypes.h**.

#define InterBlockSize 4

side length of a coefficient block in a HL or LH subband

Definition at line **87** of file **PGFtypes.h**.

#define LinBlockSize 8

side length of a coefficient block in a HH or LL subband

Definition at line **86** of file **PGFtypes.h**.

#define MagicVersionSize sizeof(PGFMagicVersion)

Definition at line **279** of file **PGFtypes.h**.

#define MaxBitPlanes 31

maximum number of bit planes of m_value: 32 minus sign bit

Definition at line **89** of file **PGFtypes.h**.

#define MaxBitPlanesLog 5

number of bits to code the maximum number of bit planes (in 32 or 16 bit mode)

Definition at line **93** of file **PGFtypes.h**.

#define MaxChannels 8

maximum number of (color) channels

Definition at line **64** of file **PGFtypes.h**.

#define MaxLevel 30

maximum number of transform levels

Definition at line **62** of file **PGFtypes.h**.

#define MaxQuality MaxBitPlanes

maximum quality

Definition at line **94** of file **PGFtypes.h**.

#define MaxUserDataSize   0x7FFFFFFF

Definition at line **284** of file **PGFtypes.h**.

#define NSubbands   4

number of subbands per level

Definition at line **63** of file **PGFtypes.h**.

#define PGF32   4

32 bit values are used -> allows at maximum 30 input bits, otherwise 16 bit values are used -> allows at maximum 14 input bits

Definition at line **69** of file **PGFtypes.h**.

#define PGFCodecVersion   STRINGIZE(PPCAT(PPCAT(PPCAT(PPCAT(PGFMajorNumber, .), PGFYear), .), PGFWeek))

Definition at line **56** of file **PGFtypes.h**.

#define PGFCodecVersionID   PPCAT(PPCAT(PPCAT(0x0, PGFMajorNumber), PGFYear), PGFWeek)

Definition at line **54** of file **PGFtypes.h**.

#define PGFMagic   "PGF"

PGF identification.

Definition at line **61** of file **PGFtypes.h**.

#define PGFMajorNumber   7

Definition at line **44** of file **PGFtypes.h**.

#define PGFROI   8

supports Regions Of Interest

Definition at line **70** of file **PGFtypes.h**.

#define PGFVersion   (Version2 | PGF32 | Version5 | Version6 | Version7)

current standard version

Definition at line **76** of file **PGFtypes.h**.

#define PGFWeek 07

Definition at line **46** of file **PGFtypes.h**.

#define PGFYear 21

Definition at line **45** of file **PGFtypes.h**.

#define PPCAT(A, B) PPCAT_NX(A, B)

Definition at line **49** of file **PGFtypes.h**.

#define PPCAT_NX(A, B) A ## B

Definition at line **48** of file **PGFtypes.h**.

#define PreHeaderSize sizeof(PGFPreHeader)

Definition at line **280** of file **PGFtypes.h**.

#define RLblockSizeLen 15

block size length (< 16): ld(BufferSize) < RLblockSizeLen <= 2*ld(BufferSize)

Definition at line **85** of file **PGFtypes.h**.

#define STRINGIZE(A) STRINGIZE_NX(A)

Definition at line **51** of file **PGFtypes.h**.

#define STRINGIZE_NX(A) #A

Definition at line **50** of file **PGFtypes.h**.

#define Version2 2

data structure **PGFHeader** of major version 2

Definition at line **68** of file **PGFtypes.h**.

#define Version5 16

new coding scheme since major version 5

Definition at line **71** of file **PGFtypes.h**.

#define Version6 32

hSize in **PGFPreHeader** uses 32 bits instead of 16 bits

Definition at line **72** of file **PGFtypes.h**.

#define Version7   64

Codec major and minor version number stored in **PGFHeader**.

Definition at line **73** of file **PGFtypes.h**.

Typedef Documentation

typedef INT32 DataT

Definition at line **269** of file **PGFtypes.h**.

typedef void(* RefreshCB) (void *p)

Definition at line **274** of file **PGFtypes.h**.

Enumeration Type Documentation

enum Orientation

Enumerator:

LL	
HL	
LH	
HH	

Definition at line **99** of file **PGFtypes.h**.

```
00099 { LL = 0, HL = 1, LH = 2, HH = 3 };
```

enum ProgressMode

Enumerator:

PM_Relative	
PM_Absolute	

Definition at line **100** of file **PGFtypes.h**.

```
00100 { PM_Relative, PM_Absolute };
```

enum UserdataPolicy

Enumerator:

UP_Skip	
UP_CachePrefix	
UP_CacheAll	

Definition at line **101** of file **PGFtypes.h**.

```
00101 { UP_Skip = 0, UP_CachePrefix = 1, UP_CacheAll = 2 };
```

PGFtypes.h

Go to the documentation of this file.
```
00001 /*
00002  * The Progressive Graphics File; http://www.libpgf.org
00003  *
00004  * $Date: 2007-06-11 10:56:17 +0200 (Mo, 11 Jun 2007) $
00005  * $Revision: 299 $
00006  *
00007  * This file Copyright (C) 2006 xeraina GmbH, Switzerland
00008  *
00009  * This program is free software; you can redistribute it and/or
00010  * modify it under the terms of the GNU LESSER GENERAL PUBLIC LICENSE
00011  * as published by the Free Software Foundation; either version 2.1
00012  * of the License, or (at your option) any later version.
00013  *
00014  * This program is distributed in the hope that it will be useful,
00015  * but WITHOUT ANY WARRANTY; without even the implied warranty of
00016  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00017  * GNU General Public License for more details.
00018  *
00019  * You should have received a copy of the GNU General Public License
00020  * along with this program; if not, write to the Free Software
00021  * Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
00022  */
00023
00028
00029 #ifndef PGF_PGFTYPES_H
00030 #define PGF_PGFTYPES_H
00031
00032 #include "PGFplatform.h"
00033
00034
//-------------------------------------------------------------------------------
00035 //      Codec versions
00036 //
00037 // Version 2:   modified data structure PGFHeader (backward compatibility assured)
00038 // Version 4:   DataT: INT32 instead of INT16, allows 30 bit per pixel and channel
(backward compatibility assured)
00039 // Version 5:   ROI, new block-reordering scheme (backward compatibility assured)
00040 // Version 6:   modified data structure PGFPreHeader: hSize (header size) is now a
UINT32 instead of a UINT16 (backward compatibility assured)
00041 // Version 7:   last two bytes in header are now used for extended version numbers;
new data representation for bitmaps (backward compatibility assured)
00042 //
00043
//-------------------------------------------------------------------------------
00044 #define PGFMajorNumber          7
00045 #define PGFYear                         21              // leading zeros are
possible
00046 #define PGFWeek                         07              // leading zeros are
possible
00047
00048 #define PPCAT_NX(A, B) A ## B
00049 #define PPCAT(A, B) PPCAT_NX(A, B)
00050 #define STRINGIZE_NX(A) #A
00051 #define STRINGIZE(A) STRINGIZE_NX(A)
00052
00053 //#define PGFCodecVersionID             0x072102
00054 #define PGFCodecVersionID PPCAT(PPCAT(PPCAT(0x0, PGFMajorNumber), PGFYear),
PGFWeek)
00055 //#define PGFCodecVersion               "7.21.02"               ///< Major number,
Minor number: Year (2) Week (2)
00056 #define PGFCodecVersion STRINGIZE(PPCAT(PPCAT(PPCAT(PPCAT(PGFMajorNumber, .),
PGFYear), .), PGFWeek))
00057
00058
//-------------------------------------------------------------------------------
00059 //      Image constants
00060
//-------------------------------------------------------------------------------
00061 #define PGFMagic                        "PGF"
00062 #define MaxLevel                        30
00063 #define NSubbands                       4
00064 #define MaxChannels                     8
00065 #define DownsampleThreshold 3
```

```
00066 #define ColorTableLen           256
00067 // version flags
00068 #define Version2                        2
00069 #define PGF32                           4
00070 #define PGFROI                          8
00071 #define Version5                        16
00072 #define Version6                        32
00073 #define Version7                        64
00074 // version numbers
00075 #ifdef __PGF32SUPPORT__
00076 #define PGFVersion                      (Version2 | PGF32 | Version5 | Version6 |
Version7)
00077 #else
00078 #define PGFVersion                      (Version2 |         Version5 | Version6 |
Version7)
00079 #endif
00080
00081
//-------------------------------------------------------------------------------
00082 //      Coder constants
00083
//-------------------------------------------------------------------------------
00084 #define BufferSize                      16384
00085 #define RLblockSizeLen          15
00086 #define LinBlockSize            8
00087 #define InterBlockSize          4
00088 #ifdef __PGF32SUPPORT__
00089         #define MaxBitPlanes    31
00090 #else
00091         #define MaxBitPlanes    15
00092 #endif
00093 #define MaxBitPlanesLog         5
00094 #define MaxQuality                      MaxBitPlanes
00095
00096
//-------------------------------------------------------------------------------
00097 // Types
00098
//-------------------------------------------------------------------------------
00099 enum Orientation                { LL = 0, HL = 1, LH = 2, HH = 3 };
00100 enum ProgressMode               { PM_Relative, PM_Absolute };
00101 enum UserdataPolicy             { UP_Skip = 0, UP_CachePrefix = 1, UP_CacheAll = 2
};
00102
00107
00108 #pragma pack(1)
00113 struct PGFMagicVersion {
00114         char magic[3];
00115         UINT8 version;
00116         // total: 4 Bytes
00117 };
00118
00123 struct PGFPreHeader : PGFMagicVersion {
00124         UINT32 hSize;
00125         // total: 8 Bytes
00126 };
00127
00132 struct PGFVersionNumber {
00133         PGFVersionNumber(UINT8 _major, UINT8 _year, UINT8 _week) : major(_major),
year(_year), week(_week) {}
00134
00135 #ifdef PGF_USE_BIG_ENDIAN
00136         UINT16 week  : 6;
00137         UINT16 year  : 6;
00138         UINT16 major : 4;
00139 #else
00140         UINT16 major : 4;
00141         UINT16 year  : 6;
00142         UINT16 week  : 6;
00143 #endif // PGF_USE_BIG_ENDIAN
00144         // total: 2 Bytes
00145 };
00146
00151 struct PGFHeader {
00152         PGFHeader() : width(0), height(0), nLevels(0), quality(0), bpp(0),
channels(0), mode(ImageModeUnknown), usedBitsPerChannel(0), version(0, 0, 0) {}
00153         UINT32 width;
```

```
00154         UINT32 height;
00155         UINT8 nLevels;
00156         UINT8 quality;
00157         UINT8 bpp;
00158         UINT8 channels;
00159         UINT8 mode;
00160         UINT8 usedBitsPerChannel;
00161         PGFVersionNumber version;
00162         // total: 16 Bytes
00163 };
00164
00169 struct PGFPostHeader {
00170         RGBQUAD clut[ColorTableLen];
00171         UINT8 *userData;
00172         UINT32 userDataLen;
00173         UINT32 cachedUserDataLen;
00174 };
00175
00180 union ROIBlockHeader {
00181         UINT16 val;
00183         struct RBH {
00184 #ifdef PGF_USE_BIG_ENDIAN
00185                 UINT16 tileEnd   :                              1;
00186                 UINT16 bufferSize: RLblockSizeLen;
00187 #else
00188                 UINT16 bufferSize: RLblockSizeLen;
00189                 UINT16 tileEnd   :                              1;
00190 #endif // PGF_USE_BIG_ENDIAN
00191         } rbh;
00192         // total: 2 Bytes
00193
00196         ROIBlockHeader(UINT16 v) { val = v; }
00197
00201         ROIBlockHeader(UINT32 size, bool end) { ASSERT(size < (1 <<
RLblockSizeLen)); rbh.bufferSize = size; rbh.tileEnd = end; }
00202 };
00203
00204 #pragma pack()
00205
00210 struct IOException {
00211         OSError error;
00212
00214         IOException() : error(NoError) {}
00215
00218         IOException(OSError err) : error(err) {}
00219 };
00220
00225 struct PGFRect {
00226         UINT32 left, top, right, bottom;
00227
00229         PGFRect() : left(0), top(0), right(0), bottom(0) {}
00230
00236         PGFRect(UINT32 x, UINT32 y, UINT32 width, UINT32 height) : left(x), top(y),
right(x + width), bottom(y + height) {}
00237
00238 #ifdef WIN32
00239         PGFRect(const RECT& rect) : left(rect.left), top(rect.top),
right(rect.right), bottom(rect.bottom) {
00240                 ASSERT(rect.left >= 0 && rect.right >= 0 && rect.left <= rect.right);
00241                 ASSERT(rect.top >= 0 && rect.bottom >= 0 && rect.top <= rect.bottom);
00242         }
00243
00244         PGFRect& operator=(const RECT& rect) {
00245                 left = rect.left; top = rect.top; right = rect.right; bottom =
rect.bottom;
00246                 return *this;
00247         }
00248
00249         operator RECT() {
00250                 RECT rect = { (LONG)left, (LONG)top, (LONG)right, (LONG)bottom };
00251                 return rect;
00252         }
00253 #endif
00254
00256         UINT32 Width() const                                    { return right -
left; }
00257
```

```
00259         UINT32 Height() const                                     { return bottom -
top; }
00260
00265         bool IsInside(UINT32 x, UINT32 y) const { return (x >= left && x < right &&
y >= top && y < bottom); }
00266 };
00267
00268 #ifdef __PGF32SUPPORT__
00269 typedef INT32 DataT;
00270 #else
00271 typedef INT16 DataT;
00272 #endif
00273
00274 typedef void (*RefreshCB)(void *p);
00275
00276
//-------------------------------------------------------------------------------
00277 // Image constants
00278
//-------------------------------------------------------------------------------
00279 #define MagicVersionSize        sizeof(PGFMagicVersion)
00280 #define PreHeaderSize           sizeof(PGFPreHeader)
00281 #define HeaderSize                      sizeof(PGFHeader)
00282 #define ColorTableSize          (ColorTableLen*sizeof(RGBQUAD))
00283 #define DataTSize                       sizeof(DataT)
00284 #define MaxUserDataSize         0x7FFFFFFF
00285
00286 #endif //PGF_PGFTYPES_H
```

BitStream.h File Reference

`#include "PGFtypes.h"`

Macros

- #define **MAKEU64**(a, b) ((UINT64) (((UINT32) (a)) | ((UINT64) ((UINT32) (b))) << 32))
  Make 64 bit unsigned integer from two 32 bit unsigned integers.

Functions

- void **SetBit** (UINT32 *stream, UINT32 pos)
- void **ClearBit** (UINT32 *stream, UINT32 pos)
- bool **GetBit** (UINT32 *stream, UINT32 pos)
- bool **CompareBitBlock** (UINT32 *stream, UINT32 pos, UINT32 k, UINT32 val)
- void **SetValueBlock** (UINT32 *stream, UINT32 pos, UINT32 val, UINT32 k)
- UINT32 **GetValueBlock** (UINT32 *stream, UINT32 pos, UINT32 k)
- void **ClearBitBlock** (UINT32 *stream, UINT32 pos, UINT32 len)
- void **SetBitBlock** (UINT32 *stream, UINT32 pos, UINT32 len)
- UINT32 **SeekBitRange** (UINT32 *stream, UINT32 pos, UINT32 len)
- UINT32 **SeekBit1Range** (UINT32 *stream, UINT32 pos, UINT32 len)
- UINT32 **AlignWordPos** (UINT32 pos)
- UINT32 **NumberOfWords** (UINT32 pos)

Variables

- static const UINT32 **Filled** = 0xFFFFFFFF

Macro Definition Documentation

#define MAKEU64(a, b) ((UINT64) (((UINT32) (a)) | ((UINT64) ((UINT32) (b))) << 32))

Make 64 bit unsigned integer from two 32 bit unsigned integers.

Definition at line **41** of file **BitStream.h**.

Function Documentation

UINT32 AlignWordPos (UINT32 *pos*)[inline]

Compute bit position of the next 32-bit word

Parameters

pos	current bit stream position

Returns

bit position of next 32-bit word

Definition at line **328** of file **BitStream.h**.

```
00328                                           {
00329 //      return ((pos + WordWidth - 1) >> WordWidthLog) << WordWidthLog;
00330         return DWWIDTHBITS(pos);
00331 }
```

void ClearBit (UINT32 * *stream*, UINT32 *pos*)[inline]

Set one bit of a bit stream to 0

Parameters

stream	A bit stream stored in array of unsigned integers
pos	A valid zero-based position in the bit stream

Definition at line **70** of file **BitStream.h**.

```
00070                                               {
00071           stream[pos >> WordWidthLog] &= ~(1 << (pos%WordWidth));
00072 }
```

void ClearBitBlock (UINT32 * *stream*, UINT32 *pos*, UINT32 *len*) [inline]

Clear block of size at least len at position pos in stream

Parameters

stream	A bit stream stored in array of unsigned integers
pos	A valid zero-based position in the bit stream
len	Number of bits set to 0

Definition at line **169** of file **BitStream.h**.

```
00169                                                                     {
00170           ASSERT(len > 0);
00171           const UINT32 iFirstInt = pos >> WordWidthLog;
00172           const UINT32 iLastInt = (pos + len - 1) >> WordWidthLog;
00173
00174           const UINT32 startMask = Filled << (pos%WordWidth);
00175 //        const UINT32 endMask=Filled>>(WordWidth-1-((pos+len-1)%WordWidth));
00176
00177           if (iFirstInt == iLastInt) {
00178                   stream[iFirstInt] &= ~(startMask /*& endMask*/);
00179           } else {
00180                   stream[iFirstInt] &= ~startMask;
00181                   for (UINT32 i = iFirstInt + 1; i <= iLastInt; i++) { // changed
<=
00182                           stream[i] = 0;
00183                   }
00184                   //stream[iLastInt] &= ~endMask;
00185           }
00186 }
```

bool CompareBitBlock (UINT32 * *stream*, UINT32 *pos*, UINT32 *k*, UINT32 *val*) [inline]

Compare k-bit binary representation of stream at position pos with val

Parameters

stream	A bit stream stored in array of unsigned integers
pos	A valid zero-based position in the bit stream
k	Number of bits to compare
val	Value to compare with

Returns

true if equal

Definition at line **91** of file **BitStream.h**.

```
00091
{
00092           const UINT32 iLoInt = pos >> WordWidthLog;
00093           const UINT32 iHiInt = (pos + k - 1) >> WordWidthLog;
00094           ASSERT(iLoInt <= iHiInt);
00095           const UINT32 mask = (Filled >> (WordWidth - k));
00096
00097           if (iLoInt == iHiInt) {
00098                   // fits into one integer
00099                   val &= mask;
00100                   val <<= (pos%WordWidth);
00101                   return (stream[iLoInt] & val) == val;
00102           } else {
00103                   // must be splitted over integer boundary
00104                   UINT64 v1 = MAKEU64(stream[iLoInt], stream[iHiInt]);
00105                   UINT64 v2 = UINT64(val & mask) << (pos%WordWidth);
00106                   return (v1 & v2) == v2;
```

```
00107         }
00108 }
```

bool GetBit (UINT32 * *stream*, UINT32 *pos*)`[inline]`

Return one bit of a bit stream

Parameters

stream	A bit stream stored in array of unsigned integers
pos	A valid zero-based position in the bit stream

Returns

bit at position pos of bit stream stream

Definition at line **79** of file **BitStream.h**.

```
00079                                                      {
00080         return (stream[pos >> WordWidthLog] & (1 << (pos%WordWidth))) > 0;
00081
00082 }
```

UINT32 GetValueBlock (UINT32 * *stream*, UINT32 *pos*, UINT32 *k*)`[inline]`

Read k-bit number from stream at position pos

Parameters

stream	A bit stream stored in array of unsigned integers
pos	A valid zero-based position in the bit stream
k	Number of bits to read: 1 <= k <= 32

Definition at line **142** of file **BitStream.h**.

```
00142                                                                          {
00143         UINT32 count, hiCount;
00144         const UINT32 iLoInt = pos >> WordWidthLog;
// integer of first bit
00145         const UINT32 iHiInt = (pos + k - 1) >> WordWidthLog;          // integer
of last bit
00146         const UINT32 loMask = Filled << (pos%WordWidth);
00147         const UINT32 hiMask = Filled >> (WordWidth - 1 - ((pos + k -
1)%WordWidth));
00148
00149         if (iLoInt == iHiInt) {
00150                 // inside integer boundary
00151                 count = stream[iLoInt] & (loMask & hiMask);
00152                 count >>= pos%WordWidth;
00153         } else {
00154                 // overlapping integer boundary
00155                 count = stream[iLoInt] & loMask;
00156                 count >>= pos%WordWidth;
00157                 hiCount = stream[iHiInt] & hiMask;
00158                 hiCount <<= WordWidth - (pos%WordWidth);
00159                 count |= hiCount;
00160         }
00161         return count;
00162 }
```

UINT32 NumberOfWords (UINT32 *pos*)`[inline]`

Compute number of the 32-bit words

Parameters

pos	Current bit stream position

Returns

Number of 32-bit words

Definition at line **337** of file **BitStream.h**.

```
00337                                          {
00338         return (pos + WordWidth - 1) >> WordWidthLog;
00339 }
```

UINT32 SeekBit1Range (UINT32 * *stream*, UINT32 *pos*, UINT32 *len*) [inline]

Returns the distance to the next 0 in stream at position pos. If no 0 is found within len bits, then len is returned.

Parameters

stream	A bit stream stored in array of unsigned integers
pos	A valid zero-based position in the bit stream
len	size of search area (in bits) return The distance to the next 0 in stream at position pos

Definition at line **249** of file **BitStream.h**.

```
00249                                                                     {
00250         UINT32 count = 0;
00251         UINT32 testMask = 1 << (pos%WordWidth);
00252         UINT32* word = stream + (pos >> WordWidthLog);
00253
00254         while (((*word & testMask) != 0) && (count < len)) {
00255                 count++;
00256                 testMask <<= 1;
00257                 if (!testMask) {
00258                         word++; testMask = 1;
00259
00260                         // fast steps if all bits in a word are one
00261                         while ((count + WordWidth <= len) && (*word == Filled))
{
00262                                 word++;
00263                                 count += WordWidth;
00264                         }
00265                 }
00266         }
00267         return count;
00268 }
```

UINT32 SeekBitRange (UINT32 * *stream*, UINT32 *pos*, UINT32 *len*) [inline]

Returns the distance to the next 1 in stream at position pos. If no 1 is found within len bits, then len is returned.

Parameters

stream	A bit stream stored in array of unsigned integers
pos	A valid zero-based position in the bit stream
len	size of search area (in bits) return The distance to the next 1 in stream at position pos

Definition at line **220** of file **BitStream.h**.

```
00220                                                                    {
00221         UINT32 count = 0;
00222         UINT32 testMask = 1 << (pos%WordWidth);
00223         UINT32* word = stream + (pos >> WordWidthLog);
00224
00225         while (((*word & testMask) == 0) && (count < len)) {
00226                 count++;
00227                 testMask <<= 1;
00228                 if (!testMask) {
00229                         word++; testMask = 1;
00230
00231                         // fast steps if all bits in a word are zero
00232                         while ((count + WordWidth <= len) && (*word == 0)) {
00233                                 word++;
00234                                 count += WordWidth;
00235                         }
00236                 }
00237         }
00238
00239         return count;
00240 }
```

void SetBit (UINT32 * *stream*, UINT32 *pos*) [inline]

Set one bit of a bit stream to 1

Parameters

stream	A bit stream stored in array of unsigned integers
pos	A valid zero-based position in the bit stream

Definition at line **62** of file **BitStream.h**.

```
00062                                                {
00063           stream[pos >> WordWidthLog] |= (1 << (pos%WordWidth));
00064 }
```

void SetBitBlock (UINT32 * *stream*, UINT32 *pos*, UINT32 *len*) `[inline]`

Set block of size at least len at position pos in stream

Parameters

stream	A bit stream stored in array of unsigned integers
pos	A valid zero-based position in the bit stream
len	Number of bits set to 1

Definition at line **193** of file **BitStream.h**.

```
00193                                                                 {
00194           ASSERT(len > 0);
00195
00196           const UINT32 iFirstInt = pos >> WordWidthLog;
00197           const UINT32 iLastInt = (pos + len - 1) >> WordWidthLog;
00198
00199           const UINT32 startMask = Filled << (pos%WordWidth);
00200 //        const UINT32 endMask=Filled>>(WordWidth-1-((pos+len-1)%WordWidth));
00201
00202           if (iFirstInt == iLastInt) {
00203                   stream[iFirstInt] |= (startMask /*& endMask*/);
00204           } else {
00205                   stream[iFirstInt] |= startMask;
00206                   for (UINT32 i = iFirstInt + 1; i <= iLastInt; i++) { // changed
<=
00207                           stream[i] = Filled;
00208                   }
00209                   //stream[iLastInt] &= ~endMask;
00210           }
00211 }
```

void SetValueBlock (UINT32 * *stream*, UINT32 *pos*, UINT32 *val*, UINT32 *k*) `[inline]`

Store k-bit binary representation of val in stream at position pos

Parameters

stream	A bit stream stored in array of unsigned integers
pos	A valid zero-based position in the bit stream
val	Value to store in stream at position pos
k	Number of bits of integer representation of val

Definition at line **116** of file **BitStream.h**.

```
00116
{
00117           const UINT32 offset = pos%WordWidth;
00118           const UINT32 iLoInt = pos >> WordWidthLog;
00119           const UINT32 iHiInt = (pos + k - 1) >> WordWidthLog;
00120           ASSERT(iLoInt <= iHiInt);
00121           const UINT32 loMask = Filled << offset;
00122           const UINT32 hiMask = Filled >> (WordWidth - 1 - ((pos + k -
1)%WordWidth));
00123
00124           if (iLoInt == iHiInt) {
00125                   // fits into one integer
00126                   stream[iLoInt] &= ~(loMask & hiMask); // clear bits
00127                   stream[iLoInt] |= val << offset; // write value
00128           } else {
00129                   // must be splitted over integer boundary
00130                   stream[iLoInt] &= ~loMask; // clear bits
00131                   stream[iLoInt] |= val << offset; // write lower part of value
00132                   stream[iHiInt] &= ~hiMask; // clear bits
```

```
00133                 stream[iHiInt] |= val >> (WordWidth - offset); // write higher
part of value
00134         }
00135 }
```

Variable Documentation

const UINT32 Filled = 0xFFFFFFFF [static]

Definition at line **38** of file **BitStream.h**.

BitStream.h

```
Go to the documentation of this file.00001 /*
00002  * The Progressive Graphics File; http://www.libpgf.org
00003  *
00004  * $Date: 2006-06-04 22:05:59 +0200 (So, 04 Jun 2006) $
00005  * $Revision: 229 $
00006  *
00007  * This file Copyright (C) 2006 xeraina GmbH, Switzerland
00008  *
00009  * This program is free software; you can redistribute it and/or
00010  * modify it under the terms of the GNU LESSER GENERAL PUBLIC LICENSE
00011  * as published by the Free Software Foundation; either version 2.1
00012  * of the License, or (at your option) any later version.
00013  *
00014  * This program is distributed in the hope that it will be useful,
00015  * but WITHOUT ANY WARRANTY; without even the implied warranty of
00016  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00017  * GNU General Public License for more details.
00018  *
00019  * You should have received a copy of the GNU General Public License
00020  * along with this program; if not, write to the Free Software
00021  * Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
00022  */
00023
00028
00029 #ifndef PGF_BITSTREAM_H
00030 #define PGF_BITSTREAM_H
00031
00032 #include "PGFtypes.h"
00033
00035 // constants
00036 //static const WordWidth = 32;
00037 //static const WordWidthLog = 5;
00038 static const UINT32 Filled = 0xFFFFFFFF;
00039
00041 #define MAKEU64(a, b) ((UINT64) (((UINT32) (a)) | ((UINT64) ((UINT32) (b))) << 32))
00042
00043 /*
00044 static UINT8 lMask[] = {
00045         0x00,                       // 00000000
00046         0x80,                       // 10000000
00047         0xc0,                       // 11000000
00048         0xe0,                       // 11100000
00049         0xf0,                       // 11110000
00050         0xf8,                       // 11111000
00051         0xfc,                       // 11111100
00052         0xfe,                       // 11111110
00053         0xff,                       // 11111111
00054 };
00055 */
00056 // these procedures have to be inlined because of performance reasons
00057
00062 inline void SetBit(UINT32* stream, UINT32 pos) {
00063         stream[pos >> WordWidthLog] |= (1 << (pos%WordWidth));
00064 }
00065
00070 inline void ClearBit(UINT32* stream, UINT32 pos) {
00071         stream[pos >> WordWidthLog] &= ~(1 << (pos%WordWidth));
00072 }
00073
00079 inline bool GetBit(UINT32* stream, UINT32 pos)  {
00080         return (stream[pos >> WordWidthLog] & (1 << (pos%WordWidth))) > 0;
00081
00082 }
00083
00091 inline bool CompareBitBlock(UINT32* stream, UINT32 pos, UINT32 k, UINT32 val) {
00092         const UINT32 iLoInt = pos >> WordWidthLog;
00093         const UINT32 iHiInt = (pos + k - 1) >> WordWidthLog;
00094         ASSERT(iLoInt <= iHiInt);
00095         const UINT32 mask = (Filled >> (WordWidth - k));
00096
00097         if (iLoInt == iHiInt) {
00098                 // fits into one integer
00099                 val &= mask;
```

```
00100                   val <<= (pos%WordWidth);
00101                   return (stream[iLoInt] & val) == val;
00102           } else {
00103                   // must be splitted over integer boundary
00104                   UINT64 v1 = MAKEU64(stream[iLoInt], stream[iHiInt]);
00105                   UINT64 v2 = UINT64(val & mask) << (pos%WordWidth);
00106                   return (v1 & v2) == v2;
00107           }
00108 }
00109
00116 inline void SetValueBlock(UINT32* stream, UINT32 pos, UINT32 val, UINT32 k) {
00117           const UINT32 offset = pos%WordWidth;
00118           const UINT32 iLoInt = pos >> WordWidthLog;
00119           const UINT32 iHiInt = (pos + k - 1) >> WordWidthLog;
00120           ASSERT(iLoInt <= iHiInt);
00121           const UINT32 loMask = Filled << offset;
00122           const UINT32 hiMask = Filled >> (WordWidth - 1 - ((pos + k - 1)%WordWidth));
00123
00124           if (iLoInt == iHiInt) {
00125                   // fits into one integer
00126                   stream[iLoInt] &= ~(loMask & hiMask); // clear bits
00127                   stream[iLoInt] |= val << offset; // write value
00128           } else {
00129                   // must be splitted over integer boundary
00130                   stream[iLoInt] &= ~loMask; // clear bits
00131                   stream[iLoInt] |= val << offset; // write lower part of value
00132                   stream[iHiInt] &= ~hiMask; // clear bits
00133                   stream[iHiInt] |= val >> (WordWidth - offset); // write higher part
of value
00134           }
00135 }
00136
00142 inline UINT32 GetValueBlock(UINT32* stream, UINT32 pos, UINT32 k) {
00143           UINT32 count, hiCount;
00144           const UINT32 iLoInt = pos >> WordWidthLog;                              //
integer of first bit
00145           const UINT32 iHiInt = (pos + k - 1) >> WordWidthLog;            // integer
of last bit
00146           const UINT32 loMask = Filled << (pos%WordWidth);
00147           const UINT32 hiMask = Filled >> (WordWidth - 1 - ((pos + k - 1)%WordWidth));
00148
00149           if (iLoInt == iHiInt) {
00150                   // inside integer boundary
00151                   count = stream[iLoInt] & (loMask & hiMask);
00152                   count >>= pos%WordWidth;
00153           } else {
00154                   // overlapping integer boundary
00155                   count = stream[iLoInt] & loMask;
00156                   count >>= pos%WordWidth;
00157                   hiCount = stream[iHiInt] & hiMask;
00158                   hiCount <<= WordWidth - (pos%WordWidth);
00159                   count |= hiCount;
00160           }
00161           return count;
00162 }
00163
00169 inline void ClearBitBlock(UINT32* stream, UINT32 pos, UINT32 len) {
00170           ASSERT(len > 0);
00171           const UINT32 iFirstInt = pos >> WordWidthLog;
00172           const UINT32 iLastInt = (pos + len - 1) >> WordWidthLog;
00173
00174           const UINT32 startMask = Filled << (pos%WordWidth);
00175 //        const UINT32 endMask=Filled>>(WordWidth-1-((pos+len-1)%WordWidth));
00176
00177           if (iFirstInt == iLastInt) {
00178                   stream[iFirstInt] &= ~(startMask /*& endMask*/);
00179           } else {
00180                   stream[iFirstInt] &= ~startMask;
00181                   for (UINT32 i = iFirstInt + 1; i <= iLastInt; i++) { // changed <=
00182                           stream[i] = 0;
00183                   }
00184                   //stream[iLastInt] &= ~endMask;
00185           }
00186 }
00187
00193 inline void SetBitBlock(UINT32* stream, UINT32 pos, UINT32 len) {
00194           ASSERT(len > 0);
```

```
00195
00196        const UINT32 iFirstInt = pos >> WordWidthLog;
00197        const UINT32 iLastInt = (pos + len - 1) >> WordWidthLog;
00198
00199        const UINT32 startMask = Filled << (pos%WordWidth);
00200 //     const UINT32 endMask=Filled>>(WordWidth-1-((pos+len-1)%WordWidth));
00201
00202        if (iFirstInt == iLastInt) {
00203                stream[iFirstInt] |= (startMask /*& endMask*/);
00204        } else {
00205                stream[iFirstInt] |= startMask;
00206                for (UINT32 i = iFirstInt + 1; i <= iLastInt; i++) { // changed <=
00207                        stream[i] = Filled;
00208                }
00209                //stream[iLastInt] &= ~endMask;
00210        }
00211 }
00212
00220 inline UINT32 SeekBitRange(UINT32* stream, UINT32 pos, UINT32 len) {
00221        UINT32 count = 0;
00222        UINT32 testMask = 1 << (pos%WordWidth);
00223        UINT32* word = stream + (pos >> WordWidthLog);
00224
00225        while (((*word & testMask) == 0) && (count < len)) {
00226                count++;
00227                testMask <<= 1;
00228                if (!testMask) {
00229                        word++; testMask = 1;
00230
00231                        // fast steps if all bits in a word are zero
00232                        while ((count + WordWidth <= len) && (*word == 0)) {
00233                                word++;
00234                                count += WordWidth;
00235                        }
00236                }
00237        }
00238
00239        return count;
00240 }
00241
00249 inline UINT32 SeekBit1Range(UINT32* stream, UINT32 pos, UINT32 len) {
00250        UINT32 count = 0;
00251        UINT32 testMask = 1 << (pos%WordWidth);
00252        UINT32* word = stream + (pos >> WordWidthLog);
00253
00254        while (((*word & testMask) != 0) && (count < len)) {
00255                count++;
00256                testMask <<= 1;
00257                if (!testMask) {
00258                        word++; testMask = 1;
00259
00260                        // fast steps if all bits in a word are one
00261                        while ((count + WordWidth <= len) && (*word == Filled)) {
00262                                word++;
00263                                count += WordWidth;
00264                        }
00265                }
00266        }
00267        return count;
00268 }
00269 /*
00274 inline void BitCopy(const UINT8 *sStream, UINT32 sPos, UINT8 *dStream, UINT32 dPos,
UINT32 k) {
00275        ASSERT(k > 0);
00276
00277        div_t divS = div(sPos, 8);
00278        div_t divD = div(dPos, 8);
00279        UINT32 sOff = divS.rem;
00280        UINT32 dOff = divD.rem;
00281        INT32 tmp = div(dPos + k - 1, 8).quot;
00282
00283        const UINT8 *sAddr = sStream + divS.quot;
00284        UINT8 *dAddrS = dStream + divD.quot;
00285        UINT8 *dAddrE = dStream + tmp;
00286        UINT8 eMask;
00287
00288        UINT8 destSB = *dAddrS;
```

```
00289         UINT8 destEB = *dAddrE;
00290         UINT8 *dAddr;
00291         UINT8 prec;
00292         INT32 shiftl, shiftr;
00293
00294         if (dOff > sOff) {
00295                 prec = 0;
00296                 shiftr = dOff - sOff;
00297                 shiftl = 8 - dOff + sOff;
00298         } else {
00299                 prec = *sAddr << (sOff - dOff);
00300                 shiftr = 8 - sOff + dOff;
00301                 shiftl = sOff - dOff;
00302                 sAddr++;
00303         }
00304
00305         for (dAddr = dAddrS; dAddr < dAddrE; dAddr++, sAddr++) {
00306                 *dAddr = prec | (*sAddr >> shiftr);
00307                 prec = *sAddr << shiftl;
00308         }
00309
00310         if ((sPos + k)%8 == 0) {
00311                 *dAddr = prec;
00312         } else {
00313                 *dAddr = prec | (*sAddr >> shiftr);
00314         }
00315
00316         eMask = lMask[dOff];
00317         *dAddrS = (destSB & eMask) | (*dAddrS & (~eMask));
00318
00319         INT32 mind = (dPos + k) % 8;
00320         eMask = (mind) ? lMask[mind] : lMask[8];
00321         *dAddrE = (destEB & (~eMask)) | (*dAddrE & eMask);
00322 }
00323 */
00328 inline UINT32 AlignWordPos(UINT32 pos) {
00329 //      return ((pos + WordWidth - 1) >> WordWidthLog) << WordWidthLog;
00330         return DWWIDTHBITS(pos);
00331 }
00332
00337 inline UINT32 NumberOfWords(UINT32 pos) {
00338         return (pos + WordWidth - 1) >> WordWidthLog;
00339 }
00340
00341 #endif //PGF_BITSTREAM_H
```

Decoder.cpp File Reference

PGF decoder class implementation.
`#include "Decoder.h"`

Macros

- #define **CodeBufferBitLen** (**CodeBufferLen*****WordWidth**)
  max number of bits in m_codeBuffer

- #define **MaxCodeLen** ((1 << **RLblockSizeLen**) - 1)
  max length of RL encoded block

Detailed Description

PGF decoder class implementation.

Author

C. Stamm, R. Spuler

Definition in file **Decoder.cpp**.

Macro Definition Documentation

#define CodeBufferBitLen (CodeBufferLen*WordWidth)

max number of bits in m_codeBuffer

Definition at line **58** of file **Decoder.cpp**.

#define MaxCodeLen ((1 << RLblockSizeLen) - 1)

max length of RL encoded block

Definition at line **59** of file **Decoder.cpp**.

Decoder.cpp

```
Go to the documentation of this file.00001 /*
00002  * The Progressive Graphics File; http://www.libpgf.org
00003  *
00004  * $Date: 2006-06-04 22:05:59 +0200 (So, 04 Jun 2006) $
00005  * $Revision: 229 $
00006  *
00007  * This file Copyright (C) 2006 xeraina GmbH, Switzerland
00008  *
00009  * This program is free software; you can redistribute it and/or
00010  * modify it under the terms of the GNU LESSER GENERAL PUBLIC LICENSE
00011  * as published by the Free Software Foundation; either version 2.1
00012  * of the License, or (at your option) any later version.
00013  *
00014  * This program is distributed in the hope that it will be useful,
00015  * but WITHOUT ANY WARRANTY; without even the implied warranty of
00016  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00017  * GNU General Public License for more details.
00018  *
00019  * You should have received a copy of the GNU General Public License
00020  * along with this program; if not, write to the Free Software
00021  * Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
00022  */
00023
00028
00029 #include "Decoder.h"
00030 #ifdef TRACE
00031         #include <stdio.h>
00032 #endif
00033
00035 // PGF: file structure
00036 //
00037 // PGFPreHeader PGFHeader [PGFPostHeader] LevelLengths Level_n-1 Level_n-2 ...
Level_0
00038 // PGFPostHeader ::= [ColorTable] [UserData]
00039 // LevelLengths  ::= UINT32[nLevels]
00040
00042 // Decoding scheme
00043 // input:  binary file
00044 // output: wavelet coefficients stored in subbands
00045 //
00046 //                    file      (for each buffer: packedLength (16 bit), packed
bits)
00047 //                      |
00048 //                m_codeBuffer  (for each plane: RLcodeLength (16 bit), RLcoded
sigBits + m_sign, refBits)
00049 //                |     |     |
00050 //           m_sign  sigBits  refBits   [BufferLen, BufferLen, BufferLen]
00051 //                |     |     |
00052 //                   m_value    [BufferSize]
00053 //                      |
00054 //                   subband
00055 //
00056
00057 // Constants
00058 #define CodeBufferBitLen                (CodeBufferLen*WordWidth)
00059 #define MaxCodeLen                              ((1 << RLblockSizeLen) - 1)
00060
00073 CDecoder::CDecoder(CPGFStream* stream, PGFPreHeader& preHeader, PGFHeader& header,
00074                                         PGFPostHeader& postHeader, UINT32*&
levelLength, UINT64& userDataPos,
00075                                         bool useOMP, UINT32 userDataPolicy)
00076 : m_stream(stream)
00077 , m_startPos(0)
00078 , m_streamSizeEstimation(0)
00079 , m_encodedHeaderLength(0)
00080 , m_currentBlockIndex(0)
00081 , m_macroBlocksAvailable(0)
00082 #ifdef __PGFROISUPPORT__
00083 , m_roi(false)
00084 #endif
00085 {
00086         ASSERT(m_stream);
00087
```

```
00088         int count, expected;
00089 
00090         // store current stream position
00091         m_startPos = m_stream->GetPos();
00092 
00093         // read magic and version
00094         count = expected = MagicVersionSize;
00095         m_stream->Read(&count, &preHeader);
00096         if (count != expected) ReturnWithError(MissingData);
00097 
00098         // read header size
00099         if (preHeader.version & Version6) {
00100                 // 32 bit header size since version 6
00101                 count = expected = 4;
00102         } else {
00103                 count = expected = 2;
00104         }
00105         m_stream->Read(&count, ((UINT8*)&preHeader) + MagicVersionSize);
00106         if (count != expected) ReturnWithError(MissingData);
00107 
00108         // make sure the values are correct read
00109         preHeader.hSize = __VAL(preHeader.hSize);
00110 
00111         // check magic number
00112         if (memcmp(preHeader.magic, PGFMagic, 3) != 0) {
00113                 // error condition: wrong Magic number
00114                 ReturnWithError(FormatCannotRead);
00115         }
00116 
00117         // read file header
00118         count = expected = (preHeader.hSize < HeaderSize) ? preHeader.hSize :
HeaderSize;
00119         m_stream->Read(&count, &header);
00120         if (count != expected) ReturnWithError(MissingData);
00121 
00122         // make sure the values are correct read
00123         header.height = __VAL(UINT32(header.height));
00124         header.width = __VAL(UINT32(header.width));
00125 
00126         // be ready to read all versions including version 0
00127         if (preHeader.version > 0) {
00128 #ifndef __PGFROISUPPORT__
00129                 // check ROI usage
00130                 if (preHeader.version & PGFROI) ReturnWithError(FormatCannotRead);
00131 #endif
00132 
00133                 UINT32 size = preHeader.hSize;
00134 
00135                 if (size > HeaderSize) {
00136                         size -= HeaderSize;
00137                         count = 0;
00138 
00139                         // read post-header
00140                         if (header.mode == ImageModeIndexedColor) {
00141                                 if (size < ColorTableSize)
ReturnWithError(FormatCannotRead);
00142                                 // read color table
00143                                 count = expected = ColorTableSize;
00144                                 m_stream->Read(&count, postHeader.clut);
00145                                 if (count != expected)
ReturnWithError(MissingData);
00146                         }
00147 
00148                         if (size > (UINT32)count) {
00149                                 size -= count;
00150 
00151                                 // read/skip user data
00152                                 UserdataPolicy policy =
(UserdataPolicy)((userDataPolicy <= MaxUserDataSize) ? UP_CachePrefix : 0xFFFFFFFF -
userDataPolicy);
00153                                 userDataPos = m_stream->GetPos();
00154                                 postHeader.userDataLen = size;
00155 
00156                                 if (policy == UP_Skip) {
00157                                         postHeader.cachedUserDataLen = 0;
00158                                         postHeader.userData = nullptr;
00159                                         Skip(size);
```

```
00160                                   } else {
00161                                           postHeader.cachedUserDataLen = (policy ==
UP_CachePrefix) ? __min(size, userDataPolicy) : size;
00162
00163                                           // create user data memory block
00164                                           postHeader.userData = new(std::nothrow)
UINT8[postHeader.cachedUserDataLen];
00165                                           if (!postHeader.userData)
ReturnWithError(InsufficientMemory);
00166
00167                                           // read user data
00168                                           count = expected =
postHeader.cachedUserDataLen;
00169                                           m_stream->Read(&count,
postHeader.userData);
00170                                           if (count != expected)
ReturnWithError(MissingData);
00171
00172                                           // skip remaining user data
00173                                           if (postHeader.cachedUserDataLen < size)
Skip(size - postHeader.cachedUserDataLen);
00174                                   }
00175                           }
00176                   }
00177
00178                   // create levelLength
00179                   levelLength = new(std::nothrow) UINT32[header.nLevels];
00180                   if (!levelLength) ReturnWithError(InsufficientMemory);
00181
00182                   // read levelLength
00183                   count = expected = header.nLevels*WordBytes;
00184                   m_stream->Read(&count, levelLength);
00185                   if (count != expected) ReturnWithError(MissingData);
00186
00187 #ifdef PGF_USE_BIG_ENDIAN
00188                   // make sure the values are correct read
00189                   for (int i=0; i < header.nLevels; i++) {
00190                           levelLength[i] = __VAL(levelLength[i]);
00191                   }
00192 #endif
00193
00194                   // compute the total size in bytes; keep attention: level length
information is optional
00195                   for (int i=0; i < header.nLevels; i++) {
00196                           m_streamSizeEstimation += levelLength[i];
00197                   }
00198
00199           }
00200
00201           // store current stream position
00202           m_encodedHeaderLength = UINT32(m_stream->GetPos() - m_startPos);
00203
00204           // set number of threads
00205 #ifdef LIBPGF_USE_OPENMP
00206           m_macroBlockLen = omp_get_num_procs();
00207 #else
00208           m_macroBlockLen = 1;
00209 #endif
00210
00211           if (useOMP && m_macroBlockLen > 1) {
00212 #ifdef LIBPGF_USE_OPENMP
00213                   omp_set_num_threads(m_macroBlockLen);
00214 #endif
00215
00216                   // create macro block array
00217                   m_macroBlocks = new(std::nothrow) CMacroBlock*[m_macroBlockLen];
00218                   if (!m_macroBlocks) ReturnWithError(InsufficientMemory);
00219                   for (int i = 0; i < m_macroBlockLen; i++) m_macroBlocks[i] = new
CMacroBlock();
00220                   m_currentBlock = m_macroBlocks[m_currentBlockIndex];
00221           } else {
00222                   m_macroBlocks = 0;
00223                   m_macroBlockLen = 1; // there is only one macro block
00224                   m_currentBlock = new(std::nothrow) CMacroBlock();
00225                   if (!m_currentBlock) ReturnWithError(InsufficientMemory);
00226           }
00227 }
```

```
00228
00230 // Destructor
00231 CDecoder::~CDecoder() {
00232         if (m_macroBlocks) {
00233                 for (int i=0; i < m_macroBlockLen; i++) delete m_macroBlocks[i];
00234                 delete[] m_macroBlocks;
00235         } else {
00236                 delete m_currentBlock;
00237         }
00238 }
00239
00246 UINT32 CDecoder::ReadEncodedData(UINT8* target, UINT32 len) const {
00247         ASSERT(m_stream);
00248
00249         int count = len;
00250         m_stream->Read(&count, target);
00251
00252         return count;
00253 }
00254
00266 void CDecoder::Partition(CSubband* band, int quantParam, int width, int height, int
startPos, int pitch) {
00267         ASSERT(band);
00268
00269         const div_t ww = div(width, LinBlockSize);
00270         const div_t hh = div(height, LinBlockSize);
00271         const int ws = pitch - LinBlockSize;
00272         const int wr = pitch - ww.rem;
00273         int pos, base = startPos, base2;
00274
00275         // main height
00276         for (int i=0; i < hh.quot; i++) {
00277                 // main width
00278                 base2 = base;
00279                 for (int j=0; j < ww.quot; j++) {
00280                         pos = base2;
00281                         for (int y=0; y < LinBlockSize; y++) {
00282                                 for (int x=0; x < LinBlockSize; x++) {
00283                                         DequantizeValue(band, pos, quantParam);
00284                                         pos++;
00285                                 }
00286                                 pos += ws;
00287                         }
00288                         base2 += LinBlockSize;
00289                 }
00290                 // rest of width
00291                 pos = base2;
00292                 for (int y=0; y < LinBlockSize; y++) {
00293                         for (int x=0; x < ww.rem; x++) {
00294                                 DequantizeValue(band, pos, quantParam);
00295                                 pos++;
00296                         }
00297                         pos += wr;
00298                         base += pitch;
00299                 }
00300         }
00301         // main width
00302         base2 = base;
00303         for (int j=0; j < ww.quot; j++) {
00304                 // rest of height
00305                 pos = base2;
00306                 for (int y=0; y < hh.rem; y++) {
00307                         for (int x=0; x < LinBlockSize; x++) {
00308                                 DequantizeValue(band, pos, quantParam);
00309                                 pos++;
00310                         }
00311                         pos += ws;
00312                 }
00313                 base2 += LinBlockSize;
00314         }
00315         // rest of height
00316         pos = base2;
00317         for (int y=0; y < hh.rem; y++) {
00318                 // rest of width
00319                 for (int x=0; x < ww.rem; x++) {
00320                         DequantizeValue(band, pos, quantParam);
00321                         pos++;
```

```
00322                   }
00323                   pos += wr;
00324           }
00325 }
00326
00328 // Decodes and dequantizes HL, and LH band of one level
00329 // LH and HH are interleaved in the codestream and must be split
00330 // Deccoding and dequantization of HL and LH Band (interleaved) using partitioning
scheme
00331 // partitions the plane in squares of side length InterBlockSize
00332 // It might throw an IOException.
00333 void CDecoder::DecodeInterleaved(CWaveletTransform* wtChannel, int level, int
quantParam) {
00334           CSubband* hlBand = wtChannel->GetSubband(level, HL);
00335           CSubband* lhBand = wtChannel->GetSubband(level, LH);
00336           const div_t lhH = div(lhBand->GetHeight(), InterBlockSize);
00337           const div_t hlW = div(hlBand->GetWidth(), InterBlockSize);
00338           const int hlws = hlBand->GetWidth() - InterBlockSize;
00339           const int hlwr = hlBand->GetWidth() - hlW.rem;
00340           const int lhws = lhBand->GetWidth() - InterBlockSize;
00341           const int lhwr = lhBand->GetWidth() - hlW.rem;
00342           int hlPos, lhPos;
00343           int hlBase = 0, lhBase = 0, hlBase2, lhBase2;
00344
00345           ASSERT(lhBand->GetWidth() >= hlBand->GetWidth());
00346           ASSERT(hlBand->GetHeight() >= lhBand->GetHeight());
00347
00348           if (!hlBand->AllocMemory()) ReturnWithError(InsufficientMemory);
00349           if (!lhBand->AllocMemory()) ReturnWithError(InsufficientMemory);
00350
00351           // correct quantParam with normalization factor
00352           quantParam -= level;
00353           if (quantParam < 0) quantParam = 0;
00354
00355           // main height
00356           for (int i=0; i < lhH.quot; i++) {
00357                   // main width
00358                   hlBase2 = hlBase;
00359                   lhBase2 = lhBase;
00360                   for (int j=0; j < hlW.quot; j++) {
00361                           hlPos = hlBase2;
00362                           lhPos = lhBase2;
00363                           for (int y=0; y < InterBlockSize; y++) {
00364                                   for (int x=0; x < InterBlockSize; x++) {
00365                                           DequantizeValue(hlBand, hlPos,
quantParam);
00366                                           DequantizeValue(lhBand, lhPos,
quantParam);
00367                                           hlPos++;
00368                                           lhPos++;
00369                                   }
00370                                   hlPos += hlws;
00371                                   lhPos += lhws;
00372                           }
00373                           hlBase2 += InterBlockSize;
00374                           lhBase2 += InterBlockSize;
00375                   }
00376                   // rest of width
00377                   hlPos = hlBase2;
00378                   lhPos = lhBase2;
00379                   for (int y=0; y < InterBlockSize; y++) {
00380                           for (int x=0; x < hlW.rem; x++) {
00381                                   DequantizeValue(hlBand, hlPos, quantParam);
00382                                   DequantizeValue(lhBand, lhPos, quantParam);
00383                                   hlPos++;
00384                                   lhPos++;
00385                           }
00386                           // width difference between HL and LH
00387                           if (lhBand->GetWidth() > hlBand->GetWidth()) {
00388                                   DequantizeValue(lhBand, lhPos, quantParam);
00389                           }
00390                           hlPos += hlwr;
00391                           lhPos += lhwr;
00392                           hlBase += hlBand->GetWidth();
00393                           lhBase += lhBand->GetWidth();
00394                   }
00395           }
```

```
00396         // main width
00397         hlBase2 = hlBase;
00398         lhBase2 = lhBase;
00399         for (int j=0; j < hlW.quot; j++) {
00400                 // rest of height
00401                 hlPos = hlBase2;
00402                 lhPos = lhBase2;
00403                 for (int y=0; y < lhH.rem; y++) {
00404                         for (int x=0; x < InterBlockSize; x++) {
00405                                 DequantizeValue(hlBand, hlPos, quantParam);
00406                                 DequantizeValue(lhBand, lhPos, quantParam);
00407                                 hlPos++;
00408                                 lhPos++;
00409                         }
00410                         hlPos += hlws;
00411                         lhPos += lhws;
00412                 }
00413                 hlBase2 += InterBlockSize;
00414                 lhBase2 += InterBlockSize;
00415         }
00416         // rest of height
00417         hlPos = hlBase2;
00418         lhPos = lhBase2;
00419         for (int y=0; y < lhH.rem; y++) {
00420                 // rest of width
00421                 for (int x=0; x < hlW.rem; x++) {
00422                         DequantizeValue(hlBand, hlPos, quantParam);
00423                         DequantizeValue(lhBand, lhPos, quantParam);
00424                         hlPos++;
00425                         lhPos++;
00426                 }
00427                 // width difference between HL and LH
00428                 if (lhBand->GetWidth() > hlBand->GetWidth()) {
00429                         DequantizeValue(lhBand, lhPos, quantParam);
00430                 }
00431                 hlPos += hlwr;
00432                 lhPos += lhwr;
00433                 hlBase += hlBand->GetWidth();
00434         }
00435         // height difference between HL and LH
00436         if (hlBand->GetHeight() > lhBand->GetHeight()) {
00437                 // total width
00438                 hlPos = hlBase;
00439                 for (int j=0; j < hlBand->GetWidth(); j++) {
00440                         DequantizeValue(hlBand, hlPos, quantParam);
00441                         hlPos++;
00442                 }
00443         }
00444 }
00445
00449 void CDecoder::Skip(UINT64 offset) {
00450         m_stream->SetPos(FSFromCurrent, offset);
00451 }
00452
00462 void CDecoder::DequantizeValue(CSubband* band, UINT32 bandPos, int quantParam) {
00463         ASSERT(m_currentBlock);
00464
00465         if (m_currentBlock->IsCompletelyRead()) {
00466                 // all data of current macro block has been read --> prepare next
macro block
00467                 GetNextMacroBlock();
00468         }
00469
00470         band->SetData(bandPos,
m_currentBlock->m_value[m_currentBlock->m_valuePos] << quantParam);
00471         m_currentBlock->m_valuePos++;
00472 }
00473
00475 // Gets next macro block
00476 // It might throw an IOException.
00477 void CDecoder::GetNextMacroBlock() {
00478         // current block has been read --> prepare next current block
00479         m_macroBlocksAvailable--;
00480
00481         if (m_macroBlocksAvailable > 0) {
00482                 m_currentBlock = m_macroBlocks[++m_currentBlockIndex];
00483         } else {
```

```
00484                   DecodeBuffer();
00485           }
00486           ASSERT(m_currentBlock);
00487 }
00488
00490 // Reads next block(s) from stream and decodes them
00491 // Decoding scheme: <wordLen>(16 bits) [ ROI ] data
00492 //                  ROI       ::= <bufferSize>(15 bits) <eofTile>(1 bit)
00493 // It might throw an IOException.
00494 void CDecoder::DecodeBuffer() {
00495           ASSERT(m_macroBlocksAvailable <= 0);
00496
00497           // macro block management
00498           if (m_macroBlockLen == 1) {
00499                   ASSERT(m_currentBlock);
00500                   ReadMacroBlock(m_currentBlock);
00501                   m_currentBlock->BitplaneDecode();
00502                   m_macroBlocksAvailable = 1;
00503           } else {
00504                   m_macroBlocksAvailable = 0;
00505                   for (int i=0; i < m_macroBlockLen; i++) {
00506                           // read sequentially several blocks
00507                           try {
00508                                   ReadMacroBlock(m_macroBlocks[i]);
00509                                   m_macroBlocksAvailable++;
00510                           } catch(IOException& ex) {
00511                                   if (ex.error == MissingData || ex.error ==
FormatCannotRead) {
00512                                           break; // no further data available or the
data isn't valid PGF data (might occur in streaming or PPPExt)
00513                                   } else {
00514                                           throw;
00515                                   }
00516                           }
00517                   }
00518 #ifdef LIBPGF_USE_OPENMP
00519                   // decode in parallel
00520                   #pragma omp parallel for default(shared) //no declared exceptions
in next block
00521 #endif
00522                   for (int i=0; i < m_macroBlocksAvailable; i++) {
00523                           m_macroBlocks[i]->BitplaneDecode();
00524                   }
00525
00526                   // prepare current macro block
00527                   m_currentBlockIndex = 0;
00528                   m_currentBlock = m_macroBlocks[m_currentBlockIndex];
00529           }
00530 }
00531
00533 // Reads next block from stream and stores it in the given macro block
00534 // It might throw an IOException.
00535 void CDecoder::ReadMacroBlock(CMacroBlock* block) {
00536           ASSERT(block);
00537
00538           UINT16 wordLen;
00539           ROIBlockHeader h(BufferSize);
00540           int count, expected;
00541
00542 #ifdef TRACE
00543           //UINT32 filePos = (UINT32)m_stream->GetPos();
00544           //printf("DecodeBuffer: %d\n", filePos);
00545 #endif
00546
00547           // read wordLen
00548           count = expected = sizeof(UINT16);
00549           m_stream->Read(&count, &wordLen);
00550           if (count != expected) ReturnWithError(MissingData);
00551           wordLen = __VAL(wordLen); // convert wordLen
00552           if (wordLen > BufferSize) ReturnWithError(FormatCannotRead);
00553
00554 #ifdef __PGFROISUPPORT__
00555           // read ROIBlockHeader
00556           if (m_roi) {
00557                   count = expected = sizeof(ROIBlockHeader);
00558                   m_stream->Read(&count, &h.val);
00559                   if (count != expected) ReturnWithError(MissingData);
```

```
00560                   h.val = __VAL(h.val); // convert ROIBlockHeader
00561           }
00562 #endif
00563           // save header
00564           block->m_header = h;
00565
00566           // read data
00567           count = expected = wordLen*WordBytes;
00568           m_stream->Read(&count, block->m_codeBuffer);
00569           if (count != expected) ReturnWithError(MissingData);
00570
00571 #ifdef PGF_USE_BIG_ENDIAN
00572           // convert data
00573           count /= WordBytes;
00574           for (int i=0; i < count; i++) {
00575                   block->m_codeBuffer[i] = __VAL(block->m_codeBuffer[i]);
00576           }
00577 #endif
00578
00579 #ifdef __PGFROISUPPORT__
00580           ASSERT(m_roi && h.rbh.bufferSize <= BufferSize || h.rbh.bufferSize ==
BufferSize);
00581 #else
00582           ASSERT(h.rbh.bufferSize == BufferSize);
00583 #endif
00584 }
00585
00586 #ifdef __PGFROISUPPORT__
00588 // Resets stream position to next tile.
00589 // Used with ROI encoding scheme only.
00590 // Reads several next blocks from stream but doesn't decode them into macro blocks
00591 // Encoding scheme: <wordLen>(16 bits) ROI data
00592 //                  ROI      ::= <bufferSize>(15 bits) <eofTile>(1 bit)
00593 // It might throw an IOException.
00594 void CDecoder::SkipTileBuffer() {
00595           ASSERT(m_roi);
00596
00597           // current macro block belongs to the last tile, so go to the next macro block
00598           m_macroBlocksAvailable--;
00599           m_currentBlockIndex++;
00600
00601           // check if pre-decoded data is available
00602           while (m_macroBlocksAvailable > 0 &&
!m_macroBlocks[m_currentBlockIndex]->m_header.rbh.tileEnd) {
00603                   m_macroBlocksAvailable--;
00604                   m_currentBlockIndex++;
00605           }
00606           if (m_macroBlocksAvailable > 0) {
00607                   // set new current macro block
00608                   m_currentBlock = m_macroBlocks[m_currentBlockIndex];
00609                   ASSERT(m_currentBlock->m_header.rbh.tileEnd);
00610                   return;
00611           }
00612
00613           ASSERT(m_macroBlocksAvailable <= 0);
00614           m_macroBlocksAvailable = 0;
00615           UINT16 wordLen;
00616           ROIBlockHeader h(0);
00617           int count, expected;
00618
00619           // skips all blocks until tile end
00620           do {
00621                   // read wordLen
00622                   count = expected = sizeof(wordLen);
00623                   m_stream->Read(&count, &wordLen);
00624                   if (count != expected) ReturnWithError(MissingData);
00625                   wordLen = __VAL(wordLen); // convert wordLen
00626                   if (wordLen > BufferSize) ReturnWithError(FormatCannotRead);
00627
00628                   // read ROIBlockHeader
00629                   count = expected = sizeof(ROIBlockHeader);
00630                   m_stream->Read(&count, &h.val);
00631                   if (count != expected) ReturnWithError(MissingData);
00632                   h.val = __VAL(h.val); // convert ROIBlockHeader
00633
00634                   // skip data
00635                   m_stream->SetPos(FSFromCurrent, wordLen*WordBytes);
```

```
00636         } while (!h.rbh.tileEnd);
00637 }
00638 #endif
00639
00641 // Decodes macro block into buffer of given size using bit plane coding.
00642 // A buffer contains bufferLen UINT32 values, thus, bufferSize bits per bit plane.
00643 // Following coding scheme is used:
00644 //              Buffer          ::= <nPlanes>(5 bits) foreach(plane i): Plane[i]
00645 //              Plane[i]        ::= [ Sig1 | Sig2 ] [DWORD alignment] refBits
00646 //              Sig1            ::= 1 <codeLen>(15 bits) codedSigAndSignBits
00647 //              Sig2            ::= 0 <sigLen>(15 bits) [Sign1 | Sign2 ] [DWORD
alignment] sigBits
00648 //              Sign1           ::= 1 <codeLen>(15 bits) codedSignBits
00649 //              Sign2           ::= 0 <signLen>(15 bits) [DWORD alignment] signBits
00650 void CDecoder::CMacroBlock::BitplaneDecode() {
00651         UINT32 bufferSize = m_header.rbh.bufferSize; ASSERT(bufferSize <=
BufferSize);
00652
00653         // clear significance vector
00654         for (UINT32 k=0; k < bufferSize; k++) {
00655                 m_sigFlagVector[k] = false;
00656         }
00657         m_sigFlagVector[bufferSize] = true; // sentinel
00658
00659         // clear output buffer
00660         for (UINT32 k=0; k < BufferSize; k++) {
00661                 m_value[k] = 0;
00662         }
00663
00664         // read number of bit planes
00665         // <nPlanes>
00666         UINT32 nPlanes = GetValueBlock(m_codeBuffer, 0, MaxBitPlanesLog);
00667         UINT32 codePos = MaxBitPlanesLog;
00668
00669         // loop through all bit planes
00670         if (nPlanes == 0) nPlanes = MaxBitPlanes + 1;
00671         ASSERT(0 < nPlanes && nPlanes <= MaxBitPlanes + 1);
00672         DataT planeMask = 1 << (nPlanes - 1);
00673
00674         for (int plane = nPlanes - 1; plane >= 0; plane--) {
00675                 UINT32 sigLen = 0;
00676
00677                 // read RL code
00678                 if (GetBit(m_codeBuffer, codePos)) {
00679                         // RL coding of sigBits is used
00680                         // <1><codeLen><codedSigAndSignBits>_<refBits>
00681                         codePos++;
00682
00683                         // read codeLen
00684                         UINT32 codeLen = GetValueBlock(m_codeBuffer, codePos,
RLblockSizeLen); ASSERT(codeLen <= MaxCodeLen);
00685
00686                         // position of encoded sigBits and signBits
00687                         UINT32 sigPos = codePos + RLblockSizeLen; ASSERT(sigPos <
CodeBufferBitLen);
00688
00689                         // refinement bits
00690                         codePos = AlignWordPos(sigPos + codeLen); ASSERT(codePos <
CodeBufferBitLen);
00691
00692                         // run-length decode significant bits and signs from
m_codeBuffer and
00693                         // read refinement bits from m_codeBuffer and compose bit
plane
00694                         sigLen = ComposeBitplaneRLD(bufferSize, planeMask, sigPos,
&m_codeBuffer[codePos >> WordWidthLog]);
00695
00696                 } else {
00697                         // no RL coding is used for sigBits and signBits together
00698                         // <0><sigLen>
00699                         codePos++;
00700
00701                         // read sigLen
00702                         sigLen = GetValueBlock(m_codeBuffer, codePos,
RLblockSizeLen); ASSERT(sigLen <= MaxCodeLen);
00703                         codePos += RLblockSizeLen; ASSERT(codePos <
CodeBufferBitLen);
```

```
00704
00705                           // read RL code for signBits
00706                           if (GetBit(m_codeBuffer, codePos)) {
00707                                   // RL coding is used just for signBits
00708                                   //
<1><codeLen><codedSignBits>_<sigBits>_<refBits>
00709                                   codePos++;
00710
00711                                   // read codeLen
00712                                   UINT32 codeLen = GetValueBlock(m_codeBuffer,
codePos, RLblockSizeLen); ASSERT(codeLen <= MaxCodeLen);
00713
00714                                   // sign bits
00715                                   UINT32 signPos = codePos + RLblockSizeLen;
ASSERT(signPos < CodeBufferBitLen);
00716
00717                                   // significant bits
00718                                   UINT32 sigPos = AlignWordPos(signPos + codeLen);
ASSERT(sigPos < CodeBufferBitLen);
00719
00720                                   // refinement bits
00721                                   codePos = AlignWordPos(sigPos + sigLen);
ASSERT(codePos < CodeBufferBitLen);
00722
00723                                   // read significant and refinement bitset from
m_codeBuffer
00724                                   sigLen = ComposeBitplaneRLD(bufferSize,
planeMask, &m_codeBuffer[sigPos >> WordWidthLog], &m_codeBuffer[codePos >> WordWidthLog],
signPos);
00725
00726                           } else {
00727                                   // RL coding of signBits was not efficient and
therefore not used
00728                                   // <0><signLen>_<signBits>_<sigBits>_<refBits>
00729                                   codePos++;
00730
00731                                   // read signLen
00732                                   UINT32 signLen = GetValueBlock(m_codeBuffer,
codePos, RLblockSizeLen); ASSERT(signLen <= MaxCodeLen);
00733
00734                                   // sign bits
00735                                   UINT32 signPos = AlignWordPos(codePos +
RLblockSizeLen); ASSERT(signPos < CodeBufferBitLen);
00736
00737                                   // significant bits
00738                                   UINT32 sigPos = AlignWordPos(signPos + signLen);
ASSERT(sigPos < CodeBufferBitLen);
00739
00740                                   // refinement bits
00741                                   codePos = AlignWordPos(sigPos + sigLen);
ASSERT(codePos < CodeBufferBitLen);
00742
00743                                   // read significant and refinement bitset from
m_codeBuffer
00744                                   sigLen = ComposeBitplane(bufferSize, planeMask,
&m_codeBuffer[sigPos >> WordWidthLog], &m_codeBuffer[codePos >> WordWidthLog],
&m_codeBuffer[signPos >> WordWidthLog]);
00745                           }
00746                   }
00747
00748                   // start of next chunk
00749                   codePos = AlignWordPos(codePos + bufferSize - sigLen);
ASSERT(codePos < CodeBufferBitLen);
00750
00751                   // next plane
00752                   planeMask >>= 1;
00753           }
00754
00755           m_valuePos = 0;
00756 }
00757
00759 // Reconstructs bitplane from significant bitset and refinement bitset
00760 // returns length [bits] of sigBits
00761 // input:  sigBits, refBits, signBits
00762 // output: m_value
00763 UINT32 CDecoder::CMacroBlock::ComposeBitplane(UINT32 bufferSize, DataT planeMask,
UINT32* sigBits, UINT32* refBits, UINT32* signBits) {
```

```
00764         ASSERT(sigBits);
00765         ASSERT(refBits);
00766         ASSERT(signBits);
00767
00768         UINT32 valPos = 0, signPos = 0, refPos = 0, sigPos = 0;
00769
00770         while (valPos < bufferSize) {
00771                 // search next 1 in m_sigFlagVector using searching with sentinel
00772                 UINT32 sigEnd = valPos;
00773                 while(!m_sigFlagVector[sigEnd]) { sigEnd++; }
00774                 sigEnd -= valPos;
00775                 sigEnd += sigPos;
00776
00777                 // search 1's in sigBits[sigPos..sigEnd)
00778                 // these 1's are significant bits
00779                 while (sigPos < sigEnd) {
00780                         // search 0's
00781                         UINT32 zerocnt = SeekBitRange(sigBits, sigPos, sigEnd -
sigPos);
00782                         sigPos += zerocnt;
00783                         valPos += zerocnt;
00784                         if (sigPos < sigEnd) {
00785                                 // write bit to m_value
00786                                 SetBitAtPos(valPos, planeMask);
00787
00788                                 // copy sign bit
00789                                 SetSign(valPos, GetBit(signBits, signPos++));
00790
00791                                 // update significance flag vector
00792                                 m_sigFlagVector[valPos++] = true;
00793                                 sigPos++;
00794                         }
00795                 }
00796                 // refinement bit
00797                 if (valPos < bufferSize) {
00798                         // write one refinement bit
00799                         if (GetBit(refBits, refPos)) {
00800                                 SetBitAtPos(valPos, planeMask);
00801                         }
00802                         refPos++;
00803                         valPos++;
00804                 }
00805         }
00806         ASSERT(sigPos <= bufferSize);
00807         ASSERT(refPos <= bufferSize);
00808         ASSERT(signPos <= bufferSize);
00809         ASSERT(valPos == bufferSize);
00810
00811         return sigPos;
00812 }
00813
00815 // Reconstructs bitplane from significant bitset and refinement bitset
00816 // returns length [bits] of decoded significant bits
00817 // input:  RL encoded sigBits and signBits in m_codeBuffer, refBits
00818 // output: m_value
00819 // RLE:
00820 // - Decode run of 2^k zeros by a single 0.
00821 // - Decode run of count 0's followed by a 1 with codeword: 1<count>x
00822 // - x is 0: if a positive sign has been stored, otherwise 1
00823 // - Read each bit from m_codeBuffer[codePos] and increment codePos.
00824 UINT32 CDecoder::CMacroBlock::ComposeBitplaneRLD(UINT32 bufferSize, DataT
planeMask, UINT32 codePos, UINT32* refBits) {
00825         ASSERT(refBits);
00826
00827         UINT32 valPos = 0, refPos = 0;
00828         UINT32 sigPos = 0, sigEnd;
00829         UINT32 k = 3;
00830         UINT32 runlen = 1 << k; // = 2^k
00831         UINT32 count = 0, rest = 0;
00832         bool set1 = false;
00833
00834         while (valPos < bufferSize) {
00835                 // search next 1 in m_sigFlagVector using searching with sentinel
00836                 sigEnd = valPos;
00837                 while(!m_sigFlagVector[sigEnd]) { sigEnd++; }
00838                 sigEnd -= valPos;
00839                 sigEnd += sigPos;
```

```
00840
00841                   while (sigPos < sigEnd) {
00842                           if (rest || set1) {
00843                                   // rest of last run
00844                                   sigPos += rest;
00845                                   valPos += rest;
00846                                   rest = 0;
00847                           } else {
00848                                   // decode significant bits
00849                                   if (GetBit(m_codeBuffer, codePos++)) {
00850                                           // extract counter and generate zero run of
length count
00851                                           if (k > 0) {
00852                                                   // extract counter
00853                                                   count =
GetValueBlock(m_codeBuffer, codePos, k);
00854                                                   codePos += k;
00855                                                   if (count > 0) {
00856                                                           sigPos += count;
00857                                                           valPos += count;
00858                                                   }
00859
00860                                                   // adapt k (half run-length
interval)
00861                                                   k--;
00862                                                   runlen >>= 1;
00863                                           }
00864
00865                                           set1 = true;
00866
00867                                   } else {
00868                                           // generate zero run of length 2^k
00869                                           sigPos += runlen;
00870                                           valPos += runlen;
00871
00872                                           // adapt k (double run-length interval)
00873                                           if (k < WordWidth) {
00874                                                   k++;
00875                                                   runlen <<= 1;
00876                                           }
00877                                   }
00878                           }
00879
00880                           if (sigPos < sigEnd) {
00881                                   if (set1) {
00882                                           set1 = false;
00883
00884                                           // write 1 bit
00885                                           SetBitAtPos(valPos, planeMask);
00886
00887                                           // set sign bit
00888                                           SetSign(valPos, GetBit(m_codeBuffer,
codePos++));
00889
00890                                           // update significance flag vector
00891                                           m_sigFlagVector[valPos++] = true;
00892                                           sigPos++;
00893                                   }
00894                           } else {
00895                                   rest = sigPos - sigEnd;
00896                                   sigPos = sigEnd;
00897                                   valPos -= rest;
00898                           }
00899
00900                   }
00901
00902                   // refinement bit
00903                   if (valPos < bufferSize) {
00904                           // write one refinement bit
00905                           if (GetBit(refBits, refPos)) {
00906                                   SetBitAtPos(valPos, planeMask);
00907                           }
00908                           refPos++;
00909                           valPos++;
00910                   }
00911           }
00912           ASSERT(sigPos <= bufferSize);
```

```
00913         ASSERT(refPos <= bufferSize);
00914         ASSERT(valPos == bufferSize);
00915
00916         return sigPos;
00917 }
00918
00920 // Reconstructs bitplane from significant bitset, refinement bitset, and RL encoded
sign bits
00921 // returns length [bits] of sigBits
00922 // input:  sigBits, refBits, RL encoded signBits
00923 // output: m_value
00924 // RLE:
00925 // decode run of 2^k 1's by a single 1
00926 // decode run of count 1's followed by a 0 with codeword: 0<count>
00927 UINT32 CDecoder::CMacroBlock::ComposeBitplaneRLD(UINT32 bufferSize, DataT
planeMask, UINT32* sigBits, UINT32* refBits, UINT32 signPos) {
00928         ASSERT(sigBits);
00929         ASSERT(refBits);
00930
00931         UINT32 valPos = 0, refPos = 0;
00932         UINT32 sigPos = 0, sigEnd;
00933         UINT32 zerocnt, count = 0;
00934         UINT32 k = 0;
00935         UINT32 runlen = 1 << k; // = 2^k
00936         bool signBit = false;
00937         bool zeroAfterRun = false;
00938
00939         while (valPos < bufferSize) {
00940                 // search next 1 in m_sigFlagVector using searching with sentinel
00941                 sigEnd = valPos;
00942                 while(!m_sigFlagVector[sigEnd]) { sigEnd++; }
00943                 sigEnd -= valPos;
00944                 sigEnd += sigPos;
00945
00946                 // search 1's in sigBits[sigPos..sigEnd)
00947                 // these 1's are significant bits
00948                 while (sigPos < sigEnd) {
00949                         // search 0's
00950                         zerocnt = SeekBitRange(sigBits, sigPos, sigEnd - sigPos);
00951                         sigPos += zerocnt;
00952                         valPos += zerocnt;
00953                         if (sigPos < sigEnd) {
00954                                 // write bit to m_value
00955                                 SetBitAtPos(valPos, planeMask);
00956
00957                                 // check sign bit
00958                                 if (count == 0) {
00959                                         // all 1's have been set
00960                                         if (zeroAfterRun) {
00961                                                 // finish the run with a 0
00962                                                 signBit = false;
00963                                                 zeroAfterRun = false;
00964                                         } else {
00965                                                 // decode next sign bit
00966                                                 if (GetBit(m_codeBuffer,
signPos++)) {
00967                                                         // generate 1's run of
length 2^k
00968                                                         count = runlen - 1;
00969                                                         signBit = true;
00970
00971                                                         // adapt k (double
run-length interval)
00972                                                         if (k < WordWidth) {
00973                                                                 k++;
00974                                                                 runlen <<= 1;
00975                                                         }
00976                                                 } else {
00977                                                         // extract counter and
generate 1's run of length count
00978                                                         if (k > 0) {
00979                                                                 // extract
counter
00980                                                                 count =
GetValueBlock(m_codeBuffer, signPos, k);
00981                                                                 signPos += k;
00982
```

```
00983                                                                   // adapt k (half
run-length interval)
00984                                                                   k--;
00985                                                                   runlen >>= 1;
00986                                                           }
00987                                                           if (count > 0) {
00988                                                                   count--;
00989                                                                   signBit = true;
00990                                                                   zeroAfterRun =
true;
00991                                                           } else {
00992                                                                   signBit = false;
00993                                                           }
00994                                                   }
00995                                           }
00996                                   } else {
00997                                           ASSERT(count > 0);
00998                                           ASSERT(signBit);
00999                                           count--;
01000                                   }
01001
01002                                   // copy sign bit
01003                                   SetSign(valPos, signBit);
01004
01005                                   // update significance flag vector
01006                                   m_sigFlagVector[valPos++] = true;
01007                                   sigPos++;
01008                           }
01009                   }
01010
01011                   // refinement bit
01012                   if (valPos < bufferSize) {
01013                           // write one refinement bit
01014                           if (GetBit(refBits, refPos)) {
01015                                   SetBitAtPos(valPos, planeMask);
01016                           }
01017                           refPos++;
01018                           valPos++;
01019                   }
01020           }
01021           ASSERT(sigPos <= bufferSize);
01022           ASSERT(refPos <= bufferSize);
01023           ASSERT(valPos == bufferSize);
01024
01025           return sigPos;
01026 }
01027
01029 #ifdef TRACE
01030 void CDecoder::DumpBuffer() {
01031           //printf("\nDump\n");
01032           //for (int i=0; i < BufferSize; i++) {
01033           //      printf("%d", m_value[i]);
01034           //}
01035 }
01036 #endif //TRACE
```

Decoder.h File Reference

PGF decoder class.

```
#include "PGFstream.h"
#include "BitStream.h"
#include "Subband.h"
#include "WaveletTransform.h"
```

Classes

- class **CDecoder**
  PGF decoder.
- class **CDecoder::CMacroBlock**
  A macro block is a decoding unit of fixed size (uncoded)

Macros

- #define **BufferLen**   (**BufferSize**/**WordWidth**)
  number of words per buffer
- #define **CodeBufferLen**   **BufferSize**
  number of words in code buffer (CodeBufferLen > BufferLen)

Detailed Description

PGF decoder class.

Author

C. Stamm, R. Spuler

Definition in file **Decoder.h**.

Macro Definition Documentation

#define BufferLen   (BufferSize/WordWidth)

number of words per buffer

Definition at line **39** of file **Decoder.h**.

#define CodeBufferLen   BufferSize

number of words in code buffer (CodeBufferLen > BufferLen)

Definition at line **40** of file **Decoder.h**.

Decoder.h

```
Go to the documentation of this file.00001 /*
00002  * The Progressive Graphics File; http://www.libpgf.org
00003  *
00004  * $Date: 2006-06-04 22:05:59 +0200 (So, 04 Jun 2006) $
00005  * $Revision: 229 $
00006  *
00007  * This file Copyright (C) 2006 xeraina GmbH, Switzerland
00008  *
00009  * This program is free software; you can redistribute it and/or
00010  * modify it under the terms of the GNU LESSER GENERAL PUBLIC LICENSE
00011  * as published by the Free Software Foundation; either version 2.1
00012  * of the License, or (at your option) any later version.
00013  *
00014  * This program is distributed in the hope that it will be useful,
00015  * but WITHOUT ANY WARRANTY; without even the implied warranty of
00016  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00017  * GNU General Public License for more details.
00018  *
00019  * You should have received a copy of the GNU General Public License
00020  * along with this program; if not, write to the Free Software
00021  * Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
00022  */
00023
00028
00029 #ifndef PGF_DECODER_H
00030 #define PGF_DECODER_H
00031
00032 #include "PGFstream.h"
00033 #include "BitStream.h"
00034 #include "Subband.h"
00035 #include "WaveletTransform.h"
00036
00038 // Constants
00039 #define BufferLen                       (BufferSize/WordWidth)
00040 #define CodeBufferLen           BufferSize
00041
00046 class CDecoder {
00051         class CMacroBlock {
00052         public:
00055                 CMacroBlock()
00056                 : m_header(0)
// makes sure that IsCompletelyRead() returns true for an empty macro block
00057 #pragma warning( suppress : 4351 )
00058                 , m_value()
00059                 , m_codeBuffer()
00060                 , m_valuePos(0)
00061                 , m_sigFlagVector()
00062                 {
00063                 }
00064
00068                 bool IsCompletelyRead() const   { return m_valuePos >=
m_header.rbh.bufferSize; }
00069
00074                 void BitplaneDecode();
00075
00076                 ROIBlockHeader m_header;
00077                 DataT  m_value[BufferSize];
00078                 UINT32 m_codeBuffer[CodeBufferLen];
00079                 UINT32 m_valuePos;
00080
00081         private:
00082                 UINT32 ComposeBitplane(UINT32 bufferSize, DataT planeMask, UINT32*
sigBits, UINT32* refBits, UINT32* signBits);
00083                 UINT32 ComposeBitplaneRLD(UINT32 bufferSize, DataT planeMask,
UINT32 sigPos, UINT32* refBits);
00084                 UINT32 ComposeBitplaneRLD(UINT32 bufferSize, DataT planeMask,
UINT32* sigBits, UINT32* refBits, UINT32 signPos);
00085                 void  SetBitAtPos(UINT32 pos, DataT planeMask)                {
(m_value[pos] >= 0) ? m_value[pos] |= planeMask : m_value[pos] -= planeMask; }
00086                 void  SetSign(UINT32 pos, bool sign)
{ m_value[pos] = -m_value[pos]*sign + m_value[pos]*(!sign); }
00087
```

```
00088                 bool m_sigFlagVector[BufferSize+1];                    // see
paper from Malvar, Fast Progressive Wavelet Coder
00089         };
00090
00091 public:
00103         CDecoder(CPGFStream* stream, PGFPreHeader& preHeader, PGFHeader& header,
00104                      PGFPostHeader& postHeader, UINT32*& levelLength, UINT64&
userDataPos,
00105                         bool useOMP, UINT32 userDataPolicy); // throws IOException
00106
00109         ~CDecoder();
00110
00122         void Partition(CSubband* band, int quantParam, int width, int height, int
startPos, int pitch);
00123
00131         void DecodeInterleaved(CWaveletTransform* wtChannel, int level, int
quantParam);
00132
00136         UINT32 GetEncodedHeaderLength() const                  { return
m_encodedHeaderLength; }
00137
00140         void SetStreamPosToStart()                             {
ASSERT(m_stream); m_stream->SetPos(FSFromStart, m_startPos); }
00141
00144         void SetStreamPosToData()                              {
ASSERT(m_stream); m_stream->SetPos(FSFromStart, m_startPos + m_encodedHeaderLength); }
00145
00149         void Skip(UINT64 offset);
00150
00157         void DequantizeValue(CSubband* band, UINT32 bandPos, int quantParam);
00158
00165         UINT32 ReadEncodedData(UINT8* target, UINT32 len) const;
00166
00170         void DecodeBuffer();
00171
00174         CPGFStream* GetStream()
{ return m_stream; }
00175
00179         void GetNextMacroBlock();
00180
00181 #ifdef __PGFROISUPPORT__
00186         void SkipTileBuffer();
00187
00190         void SetROI()                                  { m_roi = true; }
00191 #endif
00192
00193 #ifdef TRACE
00194         void DumpBuffer();
00195 #endif
00196
00197 private:
00198         void ReadMacroBlock(CMacroBlock* block);
00199
00200         CPGFStream *m_stream;
00201         UINT64 m_startPos;
00202         UINT64 m_streamSizeEstimation;
00203         UINT32 m_encodedHeaderLength;
00204
00205         CMacroBlock **m_macroBlocks;
00206         int m_currentBlockIndex;
00207         int     m_macroBlockLen;
00208         int     m_macroBlocksAvailable;
00209         CMacroBlock *m_currentBlock;
00210
00211 #ifdef __PGFROISUPPORT__
00212         bool   m_roi;
00213 #endif
00214 };
00215
00216 #endif //PGF_DECODER_H
```

Encoder.cpp File Reference

PGF encoder class implementation.
`#include "Encoder.h"`

Macros

- #define **CodeBufferBitLen** (**CodeBufferLen*WordWidth**)
  max number of bits in m_codeBuffer

- #define **MaxCodeLen** ((1 << **RLblockSizeLen**) - 1)
  max length of RL encoded block

Detailed Description

PGF encoder class implementation.

Author

C. Stamm, R. Spuler

Definition in file **Encoder.cpp**.

Macro Definition Documentation

#define CodeBufferBitLen (CodeBufferLen*WordWidth)

max number of bits in m_codeBuffer

Definition at line **58** of file **Encoder.cpp**.

#define MaxCodeLen ((1 << RLblockSizeLen) - 1)

max length of RL encoded block

Definition at line **59** of file **Encoder.cpp**.

Encoder.cpp

```
Go to the documentation of this file.00001 /*
00002  * The Progressive Graphics File; http://www.libpgf.org
00003  *
00004  * $Date: 2007-02-03 13:04:21 +0100 (Sa, 03 Feb 2007) $
00005  * $Revision: 280 $
00006  *
00007  * This file Copyright (C) 2006 xeraina GmbH, Switzerland
00008  *
00009  * This program is free software; you can redistribute it and/or
00010  * modify it under the terms of the GNU LESSER GENERAL PUBLIC LICENSE
00011  * as published by the Free Software Foundation; either version 2.1
00012  * of the License, or (at your option) any later version.
00013  *
00014  * This program is distributed in the hope that it will be useful,
00015  * but WITHOUT ANY WARRANTY; without even the implied warranty of
00016  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00017  * GNU General Public License for more details.
00018  *
00019  * You should have received a copy of the GNU General Public License
00020  * along with this program; if not, write to the Free Software
00021  * Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
00022  */
00023
00028
00029 #include "Encoder.h"
00030 #ifdef TRACE
00031         #include <stdio.h>
00032 #endif
00033
00035 // PGF: file structure
00036 //
00037 // PGFPreHeader PGFHeader [PGFPostHeader] LevelLengths Level_n-1 Level_n-2 ...
Level_0
00038 // PGFPostHeader ::= [ColorTable] [UserData]
00039 // LevelLengths  ::= UINT32[nLevels]
00040
00042 // Encoding scheme
00043 // input: wavelet coefficients stored in subbands
00044 // output: binary file
00045 //
00046 //                   subband
00047 //                      |
00048 //                   m_value    [BufferSize]
00049 //                |     |     |
00050 //           m_sign  sigBits  refBits   [BufferSize, BufferLen, BufferLen]
00051 //                |     |     |
00052 //                m_codeBuffer  (for each plane: RLcodeLength (16 bit), RLcoded
sigBits + m_sign, refBits)
00053 //                      |
00054 //                    file      (for each buffer: packedLength (16 bit), packed
bits)
00055 //
00056
00057 // Constants
00058 #define CodeBufferBitLen                (CodeBufferLen*WordWidth)
00059 #define MaxCodeLen                              ((1 << RLblockSizeLen) - 1)
00060
00070 CEncoder::CEncoder(CPGFStream* stream, PGFPreHeader preHeader, PGFHeader header,
const PGFPostHeader& postHeader, UINT64& userDataPos, bool useOMP)
00071 : m_stream(stream)
00072 , m_bufferStartPos(0)
00073 , m_currLevelIndex(0)
00074 , m_nLevels(header.nLevels)
00075 , m_favorSpeed(false)
00076 , m_forceWriting(false)
00077 #ifdef __PGFROISUPPORT__
00078 , m_roi(false)
00079 #endif
00080 {
00081         ASSERT(m_stream);
00082
00083         int count;
00084         m_lastMacroBlock = 0;
```

```
00085         m_levelLength = nullptr;
00086
00087         // set number of threads
00088 #ifdef LIBPGF_USE_OPENMP
00089         m_macroBlockLen = omp_get_num_procs();
00090 #else
00091         m_macroBlockLen = 1;
00092 #endif
00093
00094         if (useOMP && m_macroBlockLen > 1) {
00095 #ifdef LIBPGF_USE_OPENMP
00096                 omp_set_num_threads(m_macroBlockLen);
00097 #endif
00098                 // create macro block array
00099                 m_macroBlocks = new(std::nothrow) CMacroBlock*[m_macroBlockLen];
00100                 if (!m_macroBlocks) ReturnWithError(InsufficientMemory);
00101                 for (int i=0; i < m_macroBlockLen; i++) m_macroBlocks[i] = new
CMacroBlock(this);
00102                 m_currentBlock = m_macroBlocks[m_lastMacroBlock++];
00103         } else {
00104                 m_macroBlocks = 0;
00105                 m_macroBlockLen = 1;
00106                 m_currentBlock = new CMacroBlock(this);
00107         }
00108
00109         // save file position
00110         m_startPosition = m_stream->GetPos();
00111
00112         // write preHeader
00113         preHeader.hSize = __VAL(preHeader.hSize);
00114         count = PreHeaderSize;
00115         m_stream->Write(&count, &preHeader);
00116
00117         // write file header
00118         header.height = __VAL(header.height);
00119         header.width = __VAL(header.width);
00120         count = HeaderSize;
00121         m_stream->Write(&count, &header);
00122
00123         // write postHeader
00124         if (header.mode == ImageModeIndexedColor) {
00125                 // write color table
00126                 count = ColorTableSize;
00127                 m_stream->Write(&count, (void *)postHeader.clut);
00128         }
00129         // save user data file position
00130         userDataPos = m_stream->GetPos();
00131         if (postHeader.userDataLen) {
00132                 if (postHeader.userData) {
00133                         // write user data
00134                         count = postHeader.userDataLen;
00135                         m_stream->Write(&count, postHeader.userData);
00136                 } else {
00137                         m_stream->SetPos(FSFromCurrent, count);
00138                 }
00139         }
00140
00141         // save level length file position
00142         m_levelLengthPos = m_stream->GetPos();
00143 }
00144
00146 // Destructor
00147 CEncoder::~CEncoder() {
00148         if (m_macroBlocks) {
00149                 for (int i=0; i < m_macroBlockLen; i++) delete m_macroBlocks[i];
00150                 delete[] m_macroBlocks;
00151         } else {
00152                 delete m_currentBlock;
00153         }
00154 }
00155
00160 void CEncoder::UpdatePostHeaderSize(PGFPreHeader preHeader) {
00161         UINT64 curPos = m_stream->GetPos(); // end of user data
00162         int count = PreHeaderSize;
00163
00164         // write preHeader
00165         SetStreamPosToStart();
```

```
00166         preHeader.hSize = __VAL(preHeader.hSize);
00167         m_stream->Write(&count, &preHeader);
00168
00169         m_stream->SetPos(FSFromStart, curPos);
00170 }
00171
00177 UINT32 CEncoder::WriteLevelLength(UINT32*& levelLength) {
00178         // renew levelLength
00179         delete[] levelLength;
00180         levelLength = new(std::nothrow) UINT32[m_nLevels];
00181         if (!levelLength) ReturnWithError(InsufficientMemory);
00182         for (UINT8 l = 0; l < m_nLevels; l++) levelLength[l] = 0;
00183         m_levelLength = levelLength;
00184
00185         // save level length file position
00186         m_levelLengthPos = m_stream->GetPos();
00187
00188         // write dummy levelLength
00189         int count = m_nLevels*WordBytes;
00190         m_stream->Write(&count, m_levelLength);
00191
00192         // save current file position
00193         SetBufferStartPos();
00194
00195         return count;
00196 }
00197
00202 UINT32 CEncoder::UpdateLevelLength() {
00203         UINT64 curPos = m_stream->GetPos(); // end of image
00204
00205         // set file pos to levelLength
00206         m_stream->SetPos(FSFromStart, m_levelLengthPos);
00207
00208         if (m_levelLength) {
00209         #ifdef PGF_USE_BIG_ENDIAN
00210                 UINT32 levelLength;
00211                 int count = WordBytes;
00212
00213                 for (int i=0; i < m_currLevelIndex; i++) {
00214                         levelLength = __VAL(UINT32(m_levelLength[i]));
00215                         m_stream->Write(&count, &levelLength);
00216                 }
00217         #else
00218                 int count = m_currLevelIndex*WordBytes;
00219
00220                 m_stream->Write(&count, m_levelLength);
00221         #endif //PGF_USE_BIG_ENDIAN
00222         } else {
00223                 int count = m_currLevelIndex*WordBytes;
00224                 m_stream->SetPos(FSFromCurrent, count);
00225         }
00226
00227         // begin of image
00228         UINT32 retValue = UINT32(curPos - m_stream->GetPos());
00229
00230         // restore file position
00231         m_stream->SetPos(FSFromStart, curPos);
00232
00233         return retValue;
00234 }
00235
00246 void CEncoder::Partition(CSubband* band, int width, int height, int startPos, int
pitch) {
00247         ASSERT(band);
00248
00249         const div_t hh = div(height, LinBlockSize);
00250         const div_t ww = div(width, LinBlockSize);
00251         const int ws = pitch - LinBlockSize;
00252         const int wr = pitch - ww.rem;
00253         int pos, base = startPos, base2;
00254
00255         // main height
00256         for (int i=0; i < hh.quot; i++) {
00257                 // main width
00258                 base2 = base;
00259                 for (int j=0; j < ww.quot; j++) {
00260                         pos = base2;
```

```
00261                         for (int y=0; y < LinBlockSize; y++) {
00262                                 for (int x=0; x < LinBlockSize; x++) {
00263                                         WriteValue(band, pos);
00264                                         pos++;
00265                                 }
00266                                 pos += ws;
00267                         }
00268                         base2 += LinBlockSize;
00269                 }
00270                 // rest of width
00271                 pos = base2;
00272                 for (int y=0; y < LinBlockSize; y++) {
00273                         for (int x=0; x < ww.rem; x++) {
00274                                 WriteValue(band, pos);
00275                                 pos++;
00276                         }
00277                         pos += wr;
00278                         base += pitch;
00279                 }
00280         }
00281         // main width
00282         base2 = base;
00283         for (int j=0; j < ww.quot; j++) {
00284                 // rest of height
00285                 pos = base2;
00286                 for (int y=0; y < hh.rem; y++) {
00287                         for (int x=0; x < LinBlockSize; x++) {
00288                                 WriteValue(band, pos);
00289                                 pos++;
00290                         }
00291                         pos += ws;
00292                 }
00293                 base2 += LinBlockSize;
00294         }
00295         // rest of height
00296         pos = base2;
00297         for (int y=0; y < hh.rem; y++) {
00298                 // rest of width
00299                 for (int x=0; x < ww.rem; x++) {
00300                         WriteValue(band, pos);
00301                         pos++;
00302                 }
00303                 pos += wr;
00304         }
00305 }
00306
00310 void CEncoder::Flush() {
00311         if (m_currentBlock->m_valuePos > 0) {
00312                 // pad buffer with zeros
00313                 memset(&(m_currentBlock->m_value[m_currentBlock->m_valuePos]),
0, (BufferSize - m_currentBlock->m_valuePos)*DataTSize);
00314                 m_currentBlock->m_valuePos = BufferSize;
00315
00316                 // encode buffer
00317                 m_forceWriting = true;  // makes sure that the following
EncodeBuffer is really written into the stream
00318                 EncodeBuffer(ROIBlockHeader(m_currentBlock->m_valuePos, true));
00319         }
00320 }
00321
00323 // Stores band value from given position bandPos into buffer m_value at position
m_valuePos
00324 // If buffer is full encode it to file
00325 // It might throw an IOException.
00326 void CEncoder::WriteValue(CSubband* band, int bandPos) {
00327         if (m_currentBlock->m_valuePos == BufferSize) {
00328                 EncodeBuffer(ROIBlockHeader(BufferSize, false));
00329         }
00330         DataT val = m_currentBlock->m_value[m_currentBlock->m_valuePos++] =
band->GetData(bandPos);
00331         UINT32 v = abs(val);
00332         if (v > m_currentBlock->m_maxAbsValue) m_currentBlock->m_maxAbsValue = v;
00333 }
00334
00336 // Encode buffer and write data into stream.
00337 // h contains buffer size and flag indicating end of tile.
00338 // Encoding scheme: <wordLen>(16 bits) [ ROI ] data
```

```
00339 //               ROI       ::= <bufferSize>(15 bits) <eofTile>(1 bit)
00340 // It might throw an IOException.
00341 void CEncoder::EncodeBuffer(ROIBlockHeader h) {
00342         ASSERT(m_currentBlock);
00343 #ifdef __PGFROISUPPORT__
00344         ASSERT(m_roi && h.rbh.bufferSize <= BufferSize || h.rbh.bufferSize ==
BufferSize);
00345 #else
00346         ASSERT(h.rbh.bufferSize == BufferSize);
00347 #endif
00348         m_currentBlock->m_header = h;
00349 
00350         // macro block management
00351         if (m_macroBlockLen == 1) {
00352                 m_currentBlock->BitplaneEncode();
00353                 WriteMacroBlock(m_currentBlock);
00354         } else {
00355                 // save last level index
00356                 int lastLevelIndex = m_currentBlock->m_lastLevelIndex;
00357 
00358                 if (m_forceWriting || m_lastMacroBlock == m_macroBlockLen) {
00359                         // encode macro blocks
00360                         /*
00361                         volatile OSError error = NoError;
00362                         #ifdef LIBPGF_USE_OPENMP
00363                         #pragma omp parallel for ordered default(shared)
00364                         #endif
00365                         for (int i=0; i < m_lastMacroBlock; i++) {
00366                                 if (error == NoError) {
00367                                         m_macroBlocks[i]->BitplaneEncode();
00368                                         #ifdef LIBPGF_USE_OPENMP
00369                                         #pragma omp ordered
00370                                         #endif
00371                                         {
00372                                                 try {
00373 
WriteMacroBlock(m_macroBlocks[i]);
00374                                                 } catch (IOException& e) {
00375                                                         error = e.error;
00376                                                 }
00377                                                 delete m_macroBlocks[i];
m_macroBlocks[i] = 0;
00378                                         }
00379                                 }
00380                         }
00381                         if (error != NoError) ReturnWithError(error);
00382                         */
00383 #ifdef LIBPGF_USE_OPENMP
00384                         #pragma omp parallel for default(shared) //no declared
exceptions in next block
00385 #endif
00386                         for (int i=0; i < m_lastMacroBlock; i++) {
00387                                 m_macroBlocks[i]->BitplaneEncode();
00388                         }
00389                         for (int i=0; i < m_lastMacroBlock; i++) {
00390                                 WriteMacroBlock(m_macroBlocks[i]);
00391                         }
00392 
00393                         // prepare for next round
00394                         m_forceWriting = false;
00395                         m_lastMacroBlock = 0;
00396                 }
00397                 // re-initialize macro block
00398                 m_currentBlock = m_macroBlocks[m_lastMacroBlock++];
00399                 m_currentBlock->Init(lastLevelIndex);
00400         }
00401 }
00402 
00404 // Write encoded macro block into stream.
00405 // It might throw an IOException.
00406 void CEncoder::WriteMacroBlock(CMacroBlock* block) {
00407         ASSERT(block);
00408 #ifdef __PGFROISUPPORT__
00409         ROIBlockHeader h = block->m_header;
00410 #endif
00411         UINT16 wordLen = UINT16(NumberOfWords(block->m_codePos)); ASSERT(wordLen
<= CodeBufferLen);
```

```
00412         int count = sizeof(UINT16);
00413
00414 #ifdef TRACE
00415         //UINT32 filePos = (UINT32)m_stream->GetPos();
00416         //printf("EncodeBuffer: %d\n", filePos);
00417 #endif
00418
00419 #ifdef PGF_USE_BIG_ENDIAN
00420         // write wordLen
00421         UINT16 wl = __VAL(wordLen);
00422         m_stream->Write(&count, &wl); ASSERT(count == sizeof(UINT16));
00423
00424 #ifdef __PGFROISUPPORT__
00425         // write ROIBlockHeader
00426         if (m_roi) {
00427                 count = sizeof(ROIBlockHeader);
00428                 h.val = __VAL(h.val);
00429                 m_stream->Write(&count, &h.val); ASSERT(count ==
sizeof(ROIBlockHeader));
00430         }
00431 #endif // __PGFROISUPPORT__
00432
00433         // convert data
00434         for (int i=0; i < wordLen; i++) {
00435                 block->m_codeBuffer[i] = __VAL(block->m_codeBuffer[i]);
00436         }
00437 #else
00438         // write wordLen
00439         m_stream->Write(&count, &wordLen); ASSERT(count == sizeof(UINT16));
00440
00441 #ifdef __PGFROISUPPORT__
00442         // write ROIBlockHeader
00443         if (m_roi) {
00444                 count = sizeof(ROIBlockHeader);
00445                 m_stream->Write(&count, &h.val); ASSERT(count ==
sizeof(ROIBlockHeader));
00446         }
00447 #endif // __PGFROISUPPORT__
00448 #endif // PGF_USE_BIG_ENDIAN
00449
00450         // write encoded data into stream
00451         count = wordLen*WordBytes;
00452         m_stream->Write(&count, block->m_codeBuffer);
00453
00454         // store levelLength
00455         if (m_levelLength) {
00456                 // store level length
00457                 // EncodeBuffer has been called after m_lastLevelIndex has been
updated
00458                 ASSERT(m_currLevelIndex < m_nLevels);
00459                 m_levelLength[m_currLevelIndex] += (UINT32)ComputeBufferLength();
00460                 m_currLevelIndex = block->m_lastLevelIndex + 1;
00461
00462         }
00463
00464         // prepare for next buffer
00465         SetBufferStartPos();
00466
00467         // reset values
00468         block->m_valuePos = 0;
00469         block->m_maxAbsValue = 0;
00470 }
00471
00473 // Encode buffer of given size using bit plane coding.
00474 // A buffer contains bufferLen UINT32 values, thus, bufferSize bits per bit plane.
00475 // Following coding scheme is used:
00476 //              Buffer          ::= <nPlanes>(5 bits) foreach(plane i): Plane[i]
00477 //              Plane[i]        ::= [ Sig1 | Sig2 ] [DWORD alignment] refBits
00478 //              Sig1            ::= 1 <codeLen>(15 bits) codedSigAndSignBits
00479 //              Sig2            ::= 0 <sigLen>(15 bits) [Sign1 | Sign2 ] [DWORD
alignment] sigBits
00480 //              Sign1           ::= 1 <codeLen>(15 bits) codedSignBits
00481 //              Sign2           ::= 0 <signLen>(15 bits) [DWORD alignment] signBits
00482 void CEncoder::CMacroBlock::BitplaneEncode() {
00483         UINT8   nPlanes;
00484         UINT32  sigLen, codeLen = 0, wordPos, refLen, signLen;
00485         UINT32  sigBits[BufferLen] = { 0 };
```

```
00486         UINT32  refBits[BufferLen] = { 0 };
00487         UINT32  signBits[BufferLen] = { 0 };
00488         UINT32  planeMask;
00489         UINT32  bufferSize = m_header.rbh.bufferSize; ASSERT(bufferSize <=
BufferSize);
00490         bool    useRL;
00491
00492 #ifdef TRACE
00493         //printf("which thread: %d\n", omp_get_thread_num());
00494 #endif
00495
00496         // clear significance vector
00497         for (UINT32 k=0; k < bufferSize; k++) {
00498                 m_sigFlagVector[k] = false;
00499         }
00500         m_sigFlagVector[bufferSize] = true; // sentinel
00501
00502         // clear output buffer
00503         for (UINT32 k=0; k < bufferSize; k++) {
00504                 m_codeBuffer[k] = 0;
00505         }
00506         m_codePos = 0;
00507
00508         // compute number of bit planes and split buffer into separate bit planes
00509         nPlanes = NumberOfBitplanes();
00510
00511         // write number of bit planes to m_codeBuffer
00512         // <nPlanes>
00513         SetValueBlock(m_codeBuffer, 0, nPlanes, MaxBitPlanesLog);
00514         m_codePos += MaxBitPlanesLog;
00515
00516         // loop through all bit planes
00517         if (nPlanes == 0) nPlanes = MaxBitPlanes + 1;
00518         planeMask = 1 << (nPlanes - 1);
00519
00520         for (int plane = nPlanes - 1; plane >= 0; plane--) {
00521                 // clear significant bitset
00522                 for (UINT32 k=0; k < BufferLen; k++) {
00523                         sigBits[k] = 0;
00524                 }
00525
00526                 // split bitplane in significant bitset and refinement bitset
00527                 sigLen = DecomposeBitplane(bufferSize, planeMask, m_codePos +
RLblockSizeLen + 1, sigBits, refBits, signBits, signLen, codeLen);
00528
00529                 if (sigLen > 0 && codeLen <= MaxCodeLen && codeLen <
AlignWordPos(sigLen) + AlignWordPos(signLen) + 2*RLblockSizeLen) {
00530                         // set RL code bit
00531                         // <1><codeLen>
00532                         SetBit(m_codeBuffer, m_codePos++);
00533
00534                         // write length codeLen to m_codeBuffer
00535                         SetValueBlock(m_codeBuffer, m_codePos, codeLen,
RLblockSizeLen);
00536                         m_codePos += RLblockSizeLen + codeLen;
00537                 } else {
00538                 #ifdef TRACE
00539                         //printf("new\n");
00540                         //for (UINT32 i=0; i < bufferSize; i++) {
00541                         //      printf("%s", (GetBit(sigBits, i)) ? "1" : "_");
00542                         //      if (i%120 == 119) printf("\n");
00543                         //}
00544                         //printf("\n");
00545                 #endif // TRACE
00546
00547                         // run-length coding wasn't efficient enough
00548                         // we don't use RL coding for sigBits
00549                         // <0><sigLen>
00550                         ClearBit(m_codeBuffer, m_codePos++);
00551
00552                         // write length sigLen to m_codeBuffer
00553                         ASSERT(sigLen <= MaxCodeLen);
00554                         SetValueBlock(m_codeBuffer, m_codePos, sigLen,
RLblockSizeLen);
00555                         m_codePos += RLblockSizeLen;
00556
00557                         if (m_encoder->m_favorSpeed || signLen == 0) {
```

```
00558                                   useRL = false;
00559                           } else {
00560                                   // overwrite m_codeBuffer
00561                                   useRL = true;
00562                                   // run-length encode m_sign and append them to the
m_codeBuffer
00563                                   codeLen = RLESigns(m_codePos + RLblockSizeLen + 1,
signBits, signLen);
00564                           }
00565
00566                           if (useRL && codeLen <= MaxCodeLen && codeLen < signLen) {
00567                                   // RL encoding of m_sign was efficient
00568                                   // <1><codeLen><codedSignBits>_
00569                                   // write RL code bit
00570                                   SetBit(m_codeBuffer, m_codePos++);
00571
00572                                   // write codeLen to m_codeBuffer
00573                                   SetValueBlock(m_codeBuffer, m_codePos, codeLen,
RLblockSizeLen);
00574
00575                                   // compute position of sigBits
00576                                   wordPos = NumberOfWords(m_codePos + RLblockSizeLen
+ codeLen);
00577                                   ASSERT(0 <= wordPos && wordPos < CodeBufferLen);
00578                           } else {
00579                                   // RL encoding of signBits wasn't efficient
00580                                   // <0><signLen>_<signBits>_
00581                                   // clear RL code bit
00582                                   ClearBit(m_codeBuffer, m_codePos++);
00583
00584                                   // write signLen to m_codeBuffer
00585                                   ASSERT(signLen <= MaxCodeLen);
00586                                   SetValueBlock(m_codeBuffer, m_codePos, signLen,
RLblockSizeLen);
00587
00588                                   // write signBits to m_codeBuffer
00589                                   wordPos = NumberOfWords(m_codePos +
RLblockSizeLen);
00590                                   ASSERT(0 <= wordPos && wordPos < CodeBufferLen);
00591                                   codeLen = NumberOfWords(signLen);
00592
00593                                   for (UINT32 k=0; k < codeLen; k++) {
00594                                           m_codeBuffer[wordPos++] = signBits[k];
00595                                   }
00596                           }
00597
00598                           // write sigBits
00599                           // <sigBits>_
00600                           ASSERT(0 <= wordPos && wordPos < CodeBufferLen);
00601                           refLen = NumberOfWords(sigLen);
00602
00603                           for (UINT32 k=0; k < refLen; k++) {
00604                                   m_codeBuffer[wordPos++] = sigBits[k];
00605                           }
00606                           m_codePos = wordPos << WordWidthLog;
00607                   }
00608
00609                   // append refinement bitset (aligned to word boundary)
00610                   // _<refBits>
00611                   wordPos = NumberOfWords(m_codePos);
00612                   ASSERT(0 <= wordPos && wordPos < CodeBufferLen);
00613                   refLen = NumberOfWords(bufferSize - sigLen);
00614
00615                   for (UINT32 k=0; k < refLen; k++) {
00616                           m_codeBuffer[wordPos++] = refBits[k];
00617                   }
00618                   m_codePos = wordPos << WordWidthLog;
00619                   planeMask >>= 1;
00620           }
00621           ASSERT(0 <= m_codePos && m_codePos <= CodeBufferBitLen);
00622 }
00623
00625 // Split bitplane of length bufferSize into significant and refinement bitset
00626 // returns length [bits] of significant bits
00627 // input:  bufferSize, planeMask, codePos
00628 // output: sigBits, refBits, signBits, signLen [bits], codeLen [bits]
00629 // RLE
```

```
00630 // - Encode run of 2^k zeros by a single 0.
00631 // - Encode run of count 0's followed by a 1 with codeword: 1<count>x
00632 // - x is 0: if a positive sign is stored, otherwise 1
00633 // - Store each bit in m_codeBuffer[codePos] and increment codePos.
00634 UINT32 CEncoder::CMacroBlock::DecomposeBitplane(UINT32 bufferSize, UINT32
planeMask, UINT32 codePos, UINT32* sigBits, UINT32* refBits, UINT32* signBits, UINT32&
signLen, UINT32& codeLen) {
00635         ASSERT(sigBits);
00636         ASSERT(refBits);
00637         ASSERT(signBits);
00638         ASSERT(codePos < CodeBufferBitLen);
00639
00640         UINT32 sigPos = 0;
00641         UINT32 valuePos = 0, valueEnd;
00642         UINT32 refPos = 0;
00643
00644         // set output value
00645         signLen = 0;
00646
00647         // prepare RLE of Sigs and Signs
00648         const UINT32 outStartPos = codePos;
00649         UINT32 k = 3;
00650         UINT32 runlen = 1 << k; // = 2^k
00651         UINT32 count = 0;
00652
00653         while (valuePos < bufferSize) {
00654                 // search next 1 in m_sigFlagVector using searching with sentinel
00655                 valueEnd = valuePos;
00656                 while(!m_sigFlagVector[valueEnd]) { valueEnd++; }
00657
00658                 // search 1's in m_value[plane][valuePos..valueEnd)
00659                 // these 1's are significant bits
00660                 while (valuePos < valueEnd) {
00661                         if (GetBitAtPos(valuePos, planeMask)) {
00662                                 // RLE encoding
00663                                 // encode run of count 0's followed by a 1
00664                                 // with codeword: 1<count>(signBits[signPos])
00665                                 SetBit(m_codeBuffer, codePos++);
00666                                 if (k > 0) {
00667                                         SetValueBlock(m_codeBuffer, codePos,
count, k);
00668                                         codePos += k;
00669
00670                                         // adapt k (half the zero run-length)
00671                                         k--;
00672                                         runlen >>= 1;
00673                                 }
00674
00675                                 // copy and write sign bit
00676                                 if (m_value[valuePos] < 0) {
00677                                         SetBit(signBits, signLen++);
00678                                         SetBit(m_codeBuffer, codePos++);
00679                                 } else {
00680                                         ClearBit(signBits, signLen++);
00681                                         ClearBit(m_codeBuffer, codePos++);
00682                                 }
00683
00684                                 // write a 1 to sigBits
00685                                 SetBit(sigBits, sigPos++);
00686
00687                                 // update m_sigFlagVector
00688                                 m_sigFlagVector[valuePos] = true;
00689
00690                                 // prepare for next run
00691                                 count = 0;
00692                         } else {
00693                                 // RLE encoding
00694                                 count++;
00695                                 if (count == runlen) {
00696                                         // encode run of 2^k zeros by a single 0
00697                                         ClearBit(m_codeBuffer, codePos++);
00698                                         // adapt k (double the zero run-length)
00699                                         if (k < WordWidth) {
00700                                                 k++;
00701                                                 runlen <<= 1;
00702                                         }
00703
```

```
00704                                         // prepare for next run
00705                                         count = 0;
00706                                 }
00707
00708                                 // write 0 to sigBits
00709                                 sigPos++;
00710                         }
00711                         valuePos++;
00712                 }
00713                 // refinement bit
00714                 if (valuePos < bufferSize) {
00715                         // write one refinement bit
00716                         if (GetBitAtPos(valuePos++, planeMask)) {
00717                                 SetBit(refBits, refPos);
00718                         } else {
00719                                 ClearBit(refBits, refPos);
00720                         }
00721                         refPos++;
00722                 }
00723         }
00724         // RLE encoding of the rest of the plane
00725         // encode run of count 0's followed by a 1
00726         // with codeword: 1<count>(signBits[signPos])
00727         SetBit(m_codeBuffer, codePos++);
00728         if (k > 0) {
00729                 SetValueBlock(m_codeBuffer, codePos, count, k);
00730                 codePos += k;
00731         }
00732         // write dmmy sign bit
00733         SetBit(m_codeBuffer, codePos++);
00734
00735         // write word filler zeros
00736
00737         ASSERT(sigPos <= bufferSize);
00738         ASSERT(refPos <= bufferSize);
00739         ASSERT(signLen <= bufferSize);
00740         ASSERT(valuePos == bufferSize);
00741         ASSERT(codePos >= outStartPos && codePos < CodeBufferBitLen);
00742         codeLen = codePos - outStartPos;
00743
00744         return sigPos;
00745 }
00746
00747
00749 // Compute number of bit planes needed
00750 UINT8 CEncoder::CMacroBlock::NumberOfBitplanes() {
00751         UINT8 cnt = 0;
00752
00753         // determine number of bitplanes for max value
00754         if (m_maxAbsValue > 0) {
00755                 while (m_maxAbsValue > 0) {
00756                         m_maxAbsValue >>= 1; cnt++;
00757                 }
00758                 if (cnt == MaxBitPlanes + 1) cnt = 0;
00759                 // end cs
00760                 ASSERT(cnt <= MaxBitPlanes);
00761                 ASSERT((cnt >> MaxBitPlanesLog) == 0);
00762                 return cnt;
00763         } else {
00764                 return 1;
00765         }
00766 }
00767
00769 // Adaptive Run-Length encoder for long sequences of ones.
00770 // Returns length of output in bits.
00771 // - Encode run of 2^k ones by a single 1.
00772 // - Encode run of count 1's followed by a 0 with codeword: 0<count>.
00773 // - Store each bit in m_codeBuffer[codePos] and increment codePos.
00774 UINT32 CEncoder::CMacroBlock::RLESigns(UINT32 codePos, UINT32* signBits, UINT32
signLen) {
00775         ASSERT(signBits);
00776         ASSERT(0 <= codePos && codePos < CodeBufferBitLen);
00777         ASSERT(0 < signLen && signLen <= BufferSize);
00778
00779         const UINT32  outStartPos = codePos;
00780         UINT32 k = 0;
00781         UINT32 runlen = 1 << k; // = 2^k
```

```
00782         UINT32 count = 0;
00783         UINT32 signPos = 0;
00784
00785         while (signPos < signLen) {
00786                 // search next 0 in signBits starting at position signPos
00787                 count = SeekBit1Range(signBits, signPos, __min(runlen, signLen -
signPos));
00788                 // count 1's found
00789                 if (count == runlen) {
00790                         // encode run of 2^k ones by a single 1
00791                         signPos += count;
00792                         SetBit(m_codeBuffer, codePos++);
00793                         // adapt k (double the 1's run-length)
00794                         if (k < WordWidth) {
00795                                 k++;
00796                                 runlen <<= 1;
00797                         }
00798                 } else {
00799                         // encode run of count 1's followed by a 0
00800                         // with codeword: 0(count)
00801                         signPos += count + 1;
00802                         ClearBit(m_codeBuffer, codePos++);
00803                         if (k > 0) {
00804                                 SetValueBlock(m_codeBuffer, codePos, count, k);
00805                                 codePos += k;
00806                         }
00807                         // adapt k (half the 1's run-length)
00808                         if (k > 0) {
00809                                 k--;
00810                                 runlen >>= 1;
00811                         }
00812                 }
00813         }
00814         ASSERT(signPos == signLen || signPos == signLen + 1);
00815         ASSERT(codePos >= outStartPos && codePos < CodeBufferBitLen);
00816         return codePos - outStartPos;
00817 }
00818
00820 #ifdef TRACE
00821 void CEncoder::DumpBuffer() const {
00822         //printf("\nDump\n");
00823         //for (UINT32 i=0; i < BufferSize; i++) {
00824         //      printf("%d", m_value[i]);
00825         //}
00826         //printf("\n");
00827 }
00828 #endif //TRACE
00829
00830
```

Encoder.h File Reference

PGF encoder class.

```
#include "PGFstream.h"
#include "BitStream.h"
#include "Subband.h"
#include "WaveletTransform.h"
```

Classes

- class **CEncoder**
  PGF encoder.
- class **CEncoder::CMacroBlock**
  A macro block is an encoding unit of fixed size (uncoded)

Macros

- #define **BufferLen** (**BufferSize**/**WordWidth**)
  number of words per buffer
- #define **CodeBufferLen** **BufferSize**
  number of words in code buffer (CodeBufferLen > BufferLen)

Detailed Description

PGF encoder class.

Author

C. Stamm, R. Spuler

Definition in file **Encoder.h**.

Macro Definition Documentation

#define BufferLen (BufferSize/WordWidth)

number of words per buffer

Definition at line **39** of file **Encoder.h**.

#define CodeBufferLen BufferSize

number of words in code buffer (CodeBufferLen > BufferLen)

Definition at line **40** of file **Encoder.h**.

Encoder.h

```
Go to the documentation of this file.00001 /*
00002  * The Progressive Graphics File; http://www.libpgf.org
00003  *
00004  * $Date: 2006-06-04 22:05:59 +0200 (So, 04 Jun 2006) $
00005  * $Revision: 229 $
00006  *
00007  * This file Copyright (C) 2006 xeraina GmbH, Switzerland
00008  *
00009  * This program is free software; you can redistribute it and/or
00010  * modify it under the terms of the GNU LESSER GENERAL PUBLIC LICENSE
00011  * as published by the Free Software Foundation; either version 2.1
00012  * of the License, or (at your option) any later version.
00013  *
00014  * This program is distributed in the hope that it will be useful,
00015  * but WITHOUT ANY WARRANTY; without even the implied warranty of
00016  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00017  * GNU General Public License for more details.
00018  *
00019  * You should have received a copy of the GNU General Public License
00020  * along with this program; if not, write to the Free Software
00021  * Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
00022  */
00023 
00028 
00029 #ifndef PGF_ENCODER_H
00030 #define PGF_ENCODER_H
00031 
00032 #include "PGFstream.h"
00033 #include "BitStream.h"
00034 #include "Subband.h"
00035 #include "WaveletTransform.h"
00036 
00038 // Constants
00039 #define BufferLen                       (BufferSize/WordWidth)
00040 #define CodeBufferLen           BufferSize
00041 
00046 class CEncoder {
00051         class CMacroBlock {
00052         public:
00056                 CMacroBlock(CEncoder *encoder)
00057 #pragma warning( suppress : 4351 )
00058                 : m_value()
00059                 , m_codeBuffer()
00060                 , m_header(0)
00061                 , m_encoder(encoder)
00062                 , m_sigFlagVector()
00063                 {
00064                         ASSERT(m_encoder);
00065                         Init(-1);
00066                 }
00067 
00071                 void Init(int lastLevelIndex) {                         //
initialize for reusage
00072                         m_valuePos = 0;
00073                         m_maxAbsValue = 0;
00074                         m_codePos = 0;
00075                         m_lastLevelIndex = lastLevelIndex;
00076                 }
00077 
00082                 void BitplaneEncode();
00083 
00084                 DataT   m_value[BufferSize];
00085                 UINT32  m_codeBuffer[CodeBufferLen];
00086                 ROIBlockHeader m_header;
00087                 UINT32  m_valuePos;
00088                 UINT32  m_maxAbsValue;
00089                 UINT32  m_codePos;
00090                 int             m_lastLevelIndex;
00091 
00092         private:
00093                 UINT32 RLESigns(UINT32 codePos, UINT32* signBits, UINT32 signLen);
```

```
00094                 UINT32 DecomposeBitplane(UINT32 bufferSize, UINT32 planeMask,
UINT32 codePos, UINT32* sigBits, UINT32* refBits, UINT32* signBits, UINT32& signLen,
UINT32& codeLen);
00095                 UINT8  NumberOfBitplanes();
00096                 bool   GetBitAtPos(UINT32 pos, UINT32 planeMask) const  { return
(abs(m_value[pos]) & planeMask) > 0; }
00097
00098                 CEncoder *m_encoder;
// encoder instance
00099                 bool    m_sigFlagVector[BufferSize+1];         // see paper from
Malvar, Fast Progressive Wavelet Coder
00100         };
00101
00102 public:
00112         CEncoder(CPGFStream* stream, PGFPreHeader preHeader, PGFHeader header,
const PGFPostHeader& postHeader,
00113                 UINT64& userDataPos, bool useOMP); // throws IOException
00114
00117         ~CEncoder();
00118
00121         void FavorSpeedOverSize() { m_favorSpeed = true; }
00122
00126         void Flush();
00127
00132         void UpdatePostHeaderSize(PGFPreHeader preHeader);
00133
00139         UINT32 WriteLevelLength(UINT32*& levelLength);
00140
00145         UINT32 UpdateLevelLength();
00146
00157         void Partition(CSubband* band, int width, int height, int startPos, int
pitch);
00158
00162         void SetEncodedLevel(int currentLevel) { ASSERT(currentLevel >= 0);
m_currentBlock->m_lastLevelIndex = m_nLevels - currentLevel - 1; m_forceWriting = true;
}
00163
00169         void WriteValue(CSubband* band, int bandPos);
00170
00174         INT64 ComputeHeaderLength() const { return m_levelLengthPos -
m_startPosition; }
00175
00179         INT64 ComputeBufferLength() const { return m_stream->GetPos() -
m_bufferStartPos; }
00180
00184         INT64 ComputeOffset() const { return m_stream->GetPos() - m_levelLengthPos;
}
00185
00188         void SetStreamPosToStart() { ASSERT(m_stream);
m_stream->SetPos(FSFromStart, m_startPosition); }
00189
00192         void SetBufferStartPos() { m_bufferStartPos = m_stream->GetPos(); }
00193
00194 #ifdef __PGFROISUPPORT__
00198         void EncodeTileBuffer() { ASSERT(m_currentBlock &&
m_currentBlock->m_valuePos >= 0 && m_currentBlock->m_valuePos <= BufferSize);
EncodeBuffer(ROIBlockHeader(m_currentBlock->m_valuePos, true)); }
00199
00202         void SetROI()                                    { m_roi = true; }
00203 #endif
00204
00205 #ifdef TRACE
00206         void DumpBuffer() const;
00207 #endif
00208
00209 private:
00210         void EncodeBuffer(ROIBlockHeader h); // throws IOException
00211         void WriteMacroBlock(CMacroBlock* block); // throws IOException
00212
00213         CPGFStream *m_stream;
00214         UINT64  m_startPosition;
00215         UINT64  m_levelLengthPos;
00216         UINT64  m_bufferStartPos;
00217
00218         CMacroBlock **m_macroBlocks;
00219         int             m_macroBlockLen;
00220         int             m_lastMacroBlock;
```

```
00221         CMacroBlock *m_currentBlock;
00222
00223         UINT32* m_levelLength;
00224         int      m_currLevelIndex;
00225         UINT8    m_nLevels;
00226         bool     m_favorSpeed;
00227         bool     m_forceWriting;
00228 #ifdef __PGFROISUPPORT__
00229         bool     m_roi;
00230 #endif
00231 };
00232
00233 #endif //PGF_ENCODER
```

PGFimage.cpp File Reference

PGF image class implementation.

```
#include "PGFimage.h"
#include "Decoder.h"
#include "Encoder.h"
#include "BitStream.h"
#include <cmath>
#include <cstring>
```

Macros

- #define **YUVoffset4**   8
- #define **YUVoffset6**   32
- #define **YUVoffset8**   128
- #define **YUVoffset16**   32768

Detailed Description

PGF image class implementation.

Author

C. Stamm

Definition in file **PGFimage.cpp**.

Macro Definition Documentation

#define YUVoffset16   32768

Definition at line **39** of file **PGFimage.cpp**.

#define YUVoffset4   8

Definition at line **36** of file **PGFimage.cpp**.

#define YUVoffset6   32

Definition at line **37** of file **PGFimage.cpp**.

#define YUVoffset8   128

Definition at line **38** of file **PGFimage.cpp**.

PGFimage.cpp

```
Go to the documentation of this file.00001 /*
00002  * The Progressive Graphics File; http://www.libpgf.org
00003  *
00004  * $Date: 2007-02-03 13:04:21 +0100 (Sa, 03 Feb 2007) $
00005  * $Revision: 280 $
00006  *
00007  * This file Copyright (C) 2006 xeraina GmbH, Switzerland
00008  *
00009  * This program is free software; you can redistribute it and/or
00010  * modify it under the terms of the GNU LESSER GENERAL PUBLIC LICENSE
00011  * as published by the Free Software Foundation; either version 2.1
00012  * of the License, or (at your option) any later version.
00013  *
00014  * This program is distributed in the hope that it will be useful,
00015  * but WITHOUT ANY WARRANTY; without even the implied warranty of
00016  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00017  * GNU General Public License for more details.
00018  *
00019  * You should have received a copy of the GNU General Public License
00020  * along with this program; if not, write to the Free Software
00021  * Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
00022  */
00023 
00028 
00029 #include "PGFimage.h"
00030 #include "Decoder.h"
00031 #include "Encoder.h"
00032 #include "BitStream.h"
00033 #include <cmath>
00034 #include <cstring>
00035 
00036 #define YUVoffset4              8                               // 2^3
00037 #define YUVoffset6              32                              // 2^5
00038 #define YUVoffset8              128                             // 2^7
00039 #define YUVoffset16             32768                   // 2^15
00040 //#define YUVoffset31           1073741824              // 2^30
00041 
00043 // global methods and variables
00044 #ifdef NEXCEPTIONS
00045         OSError _PGF_Error_;
00046 
00047         OSError GetLastPGFError() {
00048                 OSError tmp = _PGF_Error_;
00049                 _PGF_Error_ = NoError;
00050                 return tmp;
00051         }
00052 #endif
00053 
00054 #ifdef _DEBUG
00055         // allows RGB and RGBA image visualization inside Visual Studio Debugger
00056         struct DebugBGRImage {
00057                 int width, height, pitch;
00058                 BYTE *data;
00059         } roiimage;
00060 #endif
00061 
00063 // Standard constructor
00064 CPGFImage::CPGFImage() {
00065         Init();
00066 }
00067 
00069 void CPGFImage::Init() {
00070         // init pointers
00071         m_decoder = nullptr;
00072         m_encoder = nullptr;
00073         m_levelLength = nullptr;
00074 
00075         // init members
00076 #ifdef __PGFROISUPPORT__
00077         m_streamReinitialized = false;
00078 #endif
00079         m_currentLevel = 0;
00080         m_quant = 0;
```

```
00081           m_userDataPos = 0;
00082           m_downsample = false;
00083           m_favorSpeedOverSize = false;
00084           m_useOMPinEncoder = true;
00085           m_useOMPinDecoder = true;
00086           m_cb = nullptr;
00087           m_cbArg = nullptr;
00088           m_progressMode = PM_Relative;
00089           m_percent = 0;
00090           m_userDataPolicy = UP_CacheAll;
00091
00092           // init preHeader
00093           memcpy(m_preHeader.magic, PGFMagic, 3);
00094           m_preHeader.version = PGFVersion;
00095           m_preHeader.hSize = 0;
00096
00097           // init postHeader
00098           m_postHeader.userData = nullptr;
00099           m_postHeader.userDataLen = 0;
00100           m_postHeader.cachedUserDataLen = 0;
00101
00102           // init channels
00103           for (int i = 0; i < MaxChannels; i++) {
00104                   m_channel[i] = nullptr;
00105                   m_wtChannel[i] = nullptr;
00106           }
00107
00108           // set image width and height
00109           for (int i = 0; i < MaxChannels; i++) {
00110                   m_width[0] = 0;
00111                   m_height[0] = 0;
00112           }
00113 }
00114
00116 // Destructor: Destroy internal data structures.
00117 CPGFImage::~CPGFImage() {
00118           m_currentLevel = -100; // unusual value used as marker in Destroy()
00119           Destroy();
00120 }
00121
00123 // Destroy internal data structures. Object state after this is the same as after
CPGFImage().
00124 void CPGFImage::Destroy() {
00125           for (int i = 0; i < m_header.channels; i++) {
00126                   delete m_wtChannel[i]; // also deletes m_channel
00127           }
00128           delete[] m_postHeader.userData;
00129           delete[] m_levelLength;
00130           delete m_decoder;
00131           delete m_encoder;
00132
00133           if (m_currentLevel != -100) Init();
00134 }
00135
00137 // Open a PGF image at current stream position: read pre-header, header, levelLength,
and ckeck image type.
00138 // Precondition: The stream has been opened for reading.
00139 // It might throw an IOException.
00140 // @param stream A PGF stream
00141 void CPGFImage::Open(CPGFStream *stream) {
00142           ASSERT(stream);
00143
00144           // create decoder and read PGFPreHeader PGFHeader PGFPostHeader LevelLengths
00145           m_decoder = new CDecoder(stream, m_preHeader, m_header, m_postHeader,
m_levelLength,
00146                   m_userDataPos, m_useOMPinDecoder, m_userDataPolicy);
00147
00148           if (m_header.nLevels > MaxLevel) ReturnWithError(FormatCannotRead);
00149
00150           // set current level
00151           m_currentLevel = m_header.nLevels;
00152
00153           // set image width and height
00154           m_width[0] = m_header.width;
00155           m_height[0] = m_header.height;
00156
00157           // complete header
```

```
00158         if (!CompleteHeader()) ReturnWithError(FormatCannotRead);
00159
00160         // interpret quant parameter
00161         if (m_header.quality > DownsampleThreshold &&
00162                 (m_header.mode == ImageModeRGBColor ||
00163                  m_header.mode == ImageModeRGBA ||
00164                  m_header.mode == ImageModeRGB48 ||
00165                  m_header.mode == ImageModeCMYKColor ||
00166                  m_header.mode == ImageModeCMYK64 ||
00167                  m_header.mode == ImageModeLabColor ||
00168                  m_header.mode == ImageModeLab48)) {
00169                 m_downsample = true;
00170                 m_quant = m_header.quality - 1;
00171         } else {
00172                 m_downsample = false;
00173                 m_quant = m_header.quality;
00174         }
00175
00176         // set channel dimensions (chrominance is subsampled by factor 2)
00177         if (m_downsample) {
00178                 for (int i=1; i < m_header.channels; i++) {
00179                         m_width[i] = (m_width[0] + 1) >> 1;
00180                         m_height[i] = (m_height[0] + 1) >> 1;
00181                 }
00182         } else {
00183                 for (int i=1; i < m_header.channels; i++) {
00184                         m_width[i] = m_width[0];
00185                         m_height[i] = m_height[0];
00186                 }
00187         }
00188
00189         if (m_header.nLevels > 0) {
00190                 // init wavelet subbands
00191                 for (int i=0; i < m_header.channels; i++) {
00192                         m_wtChannel[i] = new CWaveletTransform(m_width[i],
m_height[i], m_header.nLevels);
00193                 }
00194
00195                 // used in Read when PM_Absolute
00196                 m_percent = pow(0.25, m_header.nLevels);
00197
00198         } else {
00199                 // very small image: we don't use DWT and encoding
00200
00201                 // read channels
00202                 for (int c=0; c < m_header.channels; c++) {
00203                         const UINT32 size = m_width[c]*m_height[c];
00204                         m_channel[c] = new(std::nothrow) DataT[size];
00205                         if (!m_channel[c]) ReturnWithError(InsufficientMemory);
00206
00207                         // read channel data from stream
00208                         for (UINT32 i=0; i < size; i++) {
00209                                 int count = DataTSize;
00210                                 stream->Read(&count, &m_channel[c][i]);
00211                                 if (count != DataTSize)
ReturnWithError(MissingData);
00212                         }
00213                 }
00214         }
00215 }
00216
00218 bool CPGFImage::CompleteHeader() {
00219         // set current codec version
00220         m_header.version = PGFVersionNumber(PGFMajorNumber, PGFYear, PGFWeek);
00221
00222         if (m_header.mode == ImageModeUnknown) {
00223                 // undefined mode
00224                 switch(m_header.bpp) {
00225                 case 1: m_header.mode = ImageModeBitmap; break;
00226                 case 8: m_header.mode = ImageModeGrayScale; break;
00227                 case 12: m_header.mode = ImageModeRGB12; break;
00228                 case 16: m_header.mode = ImageModeRGB16; break;
00229                 case 24: m_header.mode = ImageModeRGBColor; break;
00230                 case 32: m_header.mode = ImageModeRGBA; break;
00231                 case 48: m_header.mode = ImageModeRGB48; break;
00232                 default: m_header.mode = ImageModeRGBColor; break;
00233                 }
```

```
00234         }
00235         if (!m_header.bpp) {
00236                 // undefined bpp
00237                 switch(m_header.mode) {
00238                 case ImageModeBitmap:
00239                         m_header.bpp = 1;
00240                         break;
00241                 case ImageModeIndexedColor:
00242                 case ImageModeGrayScale:
00243                         m_header.bpp = 8;
00244                         break;
00245                 case ImageModeRGB12:
00246                         m_header.bpp = 12;
00247                         break;
00248                 case ImageModeRGB16:
00249                 case ImageModeGray16:
00250                         m_header.bpp = 16;
00251                         break;
00252                 case ImageModeRGBColor:
00253                 case ImageModeLabColor:
00254                         m_header.bpp = 24;
00255                         break;
00256                 case ImageModeRGBA:
00257                 case ImageModeCMYKColor:
00258                 case ImageModeGray32:
00259                         m_header.bpp = 32;
00260                         break;
00261                 case ImageModeRGB48:
00262                 case ImageModeLab48:
00263                         m_header.bpp = 48;
00264                         break;
00265                 case ImageModeCMYK64:
00266                         m_header.bpp = 64;
00267                         break;
00268                 default:
00269                         ASSERT(false);
00270                         m_header.bpp = 24;
00271                 }
00272         }
00273         if (m_header.mode == ImageModeRGBColor && m_header.bpp == 32) {
00274                 // change mode
00275                 m_header.mode = ImageModeRGBA;
00276         }
00277         if (m_header.mode == ImageModeBitmap && m_header.bpp != 1) return false;
00278         if (m_header.mode == ImageModeIndexedColor && m_header.bpp != 8) return
false;
00279         if (m_header.mode == ImageModeGrayScale && m_header.bpp != 8) return false;
00280         if (m_header.mode == ImageModeGray16 && m_header.bpp != 16) return false;
00281         if (m_header.mode == ImageModeGray32 && m_header.bpp != 32) return false;
00282         if (m_header.mode == ImageModeRGBColor && m_header.bpp != 24) return false;
00283         if (m_header.mode == ImageModeRGBA && m_header.bpp != 32) return false;
00284         if (m_header.mode == ImageModeRGB12 && m_header.bpp != 12) return false;
00285         if (m_header.mode == ImageModeRGB16 && m_header.bpp != 16) return false;
00286         if (m_header.mode == ImageModeRGB48 && m_header.bpp != 48) return false;
00287         if (m_header.mode == ImageModeLabColor && m_header.bpp != 24) return false;
00288         if (m_header.mode == ImageModeLab48 && m_header.bpp != 48) return false;
00289         if (m_header.mode == ImageModeCMYKColor && m_header.bpp != 32) return false;
00290         if (m_header.mode == ImageModeCMYK64 && m_header.bpp != 64) return false;
00291
00292         // set number of channels
00293         if (!m_header.channels) {
00294                 switch(m_header.mode) {
00295                 case ImageModeBitmap:
00296                 case ImageModeIndexedColor:
00297                 case ImageModeGrayScale:
00298                 case ImageModeGray16:
00299                 case ImageModeGray32:
00300                         m_header.channels = 1;
00301                         break;
00302                 case ImageModeRGBColor:
00303                 case ImageModeRGB12:
00304                 case ImageModeRGB16:
00305                 case ImageModeRGB48:
00306                 case ImageModeLabColor:
00307                 case ImageModeLab48:
00308                         m_header.channels = 3;
00309                         break;
```

```
00310                   case ImageModeRGBA:
00311                   case ImageModeCMYKColor:
00312                   case ImageModeCMYK64:
00313                           m_header.channels = 4;
00314                           break;
00315                   default:
00316                           return false;
00317                   }
00318           }
00319
00320           // store used bits per channel
00321           UINT8 bpc = m_header.bpp/m_header.channels;
00322           if (bpc > 31) bpc = 31;
00323           if (!m_header.usedBitsPerChannel || m_header.usedBitsPerChannel > bpc) {
00324                   m_header.usedBitsPerChannel = bpc;
00325           }
00326
00327           return true;
00328 }
00329
00337 const UINT8* CPGFImage::GetUserData(UINT32& cachedSize, UINT32* pTotalSize /*=
nullptr*/) const {
00338           cachedSize = m_postHeader.cachedUserDataLen;
00339           if (pTotalSize) *pTotalSize = m_postHeader.userDataLen;
00340           return m_postHeader.userData;
00341 }
00342
00348 void CPGFImage::Reconstruct(int level /*= 0*/) {
00349           if (m_header.nLevels == 0) {
00350                   // image didn't use wavelet transform
00351                   if (level == 0) {
00352                           for (int i=0; i < m_header.channels; i++) {
00353                                   ASSERT(m_wtChannel[i]);
00354                                   m_channel[i] = m_wtChannel[i]->GetSubband(0,
LL)->GetBuffer();
00355                           }
00356                   }
00357           } else {
00358                   int currentLevel = m_header.nLevels;
00359
00360           #ifdef __PGFROISUPPORT__
00361                   if (ROIisSupported()) {
00362                           // enable ROI reading
00363                           SetROI(PGFRect(0, 0, m_header.width, m_header.height));
00364                   }
00365           #endif
00366
00367                   while (currentLevel > level) {
00368                           for (int i=0; i < m_header.channels; i++) {
00369                                   ASSERT(m_wtChannel[i]);
00370                                   // dequantize subbands
00371                                   if (currentLevel == m_header.nLevels) {
00372                                           // last level also has LL band
00373                                           m_wtChannel[i]->GetSubband(currentLevel,
LL)->Dequantize(m_quant);
00374                                   }
00375                                   m_wtChannel[i]->GetSubband(currentLevel,
HL)->Dequantize(m_quant);
00376                                   m_wtChannel[i]->GetSubband(currentLevel,
LH)->Dequantize(m_quant);
00377                                   m_wtChannel[i]->GetSubband(currentLevel,
HH)->Dequantize(m_quant);
00378
00379                                   // inverse transform from m_wtChannel to m_channel
00380                                   OSError err =
m_wtChannel[i]->InverseTransform(currentLevel, &m_width[i], &m_height[i],
&m_channel[i]);
00381                                   if (err != NoError) ReturnWithError(err);
00382                                   ASSERT(m_channel[i]);
00383                           }
00384
00385                           currentLevel--;
00386                   }
00387           }
00388 }
00389
00391 // Read and decode some levels of a PGF image at current stream position.
```

```
00392 // A PGF image is structered in levels, numbered between 0 and Levels() - 1.
00393 // Each level can be seen as a single image, containing the same content
00394 // as all other levels, but in a different size (width, height).
00395 // The image size at level i is double the size (width, height) of the image at level
i+1.
00396 // The image at level 0 contains the original size.
00397 // Precondition: The PGF image has been opened with a call of Open(...).
00398 // It might throw an IOException.
00399 // @param level The image level of the resulting image in the internal image buffer.
00400 // @param cb A pointer to a callback procedure. The procedure is called after reading
a single level. If cb returns true, then it stops proceeding.
00401 // @param data Data Pointer to C++ class container to host callback procedure.
00402 void CPGFImage::Read(int level /*= 0*/, CallbackPtr cb /*= nullptr*/, void *data
/*=nullptr*/) {
00403         ASSERT((level >= 0 && level < m_header.nLevels) || m_header.nLevels == 0);
// m_header.nLevels == 0: image didn't use wavelet transform
00404         ASSERT(m_decoder);
00405 
00406 #ifdef __PGFROISUPPORT__
00407         if (ROIisSupported() && m_header.nLevels > 0) {
00408                 // new encoding scheme supporting ROI
00409                 PGFRect rect(0, 0, m_header.width, m_header.height);
00410                 Read(rect, level, cb, data);
00411                 return;
00412         }
00413 #endif
00414 
00415         if (m_header.nLevels == 0) {
00416                 if (level == 0) {
00417                         // the data has already been read during open
00418                         // now update progress
00419                         if (cb) {
00420                                 if ((*cb)(1.0, true, data))
ReturnWithError(EscapePressed);
00421                         }
00422                 }
00423         } else {
00424                 const int levelDiff = m_currentLevel - level;
00425                 double percent = (m_progressMode == PM_Relative) ? pow(0.25,
levelDiff) : m_percent;
00426 
00427                 // encoding scheme without ROI
00428                 while (m_currentLevel > level) {
00429                         for (int i=0; i < m_header.channels; i++) {
00430                                 CWaveletTransform* wtChannel = m_wtChannel[i];
00431                                 ASSERT(wtChannel);
00432 
00433                                 // decode file and write stream to m_wtChannel
00434                                 if (m_currentLevel == m_header.nLevels) {
00435                                         // last level also has LL band
00436                                         wtChannel->GetSubband(m_currentLevel,
LL)->PlaceTile(*m_decoder, m_quant);
00437                                 }
00438                                 if (m_preHeader.version & Version5) {
00439                                         // since version 5
00440                                         wtChannel->GetSubband(m_currentLevel,
HL)->PlaceTile(*m_decoder, m_quant);
00441                                         wtChannel->GetSubband(m_currentLevel,
LH)->PlaceTile(*m_decoder, m_quant);
00442                                 } else {
00443                                         // until version 4
00444                                         m_decoder->DecodeInterleaved(wtChannel,
m_currentLevel, m_quant);
00445                                 }
00446                                 wtChannel->GetSubband(m_currentLevel,
HH)->PlaceTile(*m_decoder, m_quant);
00447                         }
00448 
00449                         volatile OSError error = NoError; // volatile prevents
optimizations
00450 #ifdef LIBPGF_USE_OPENMP
00451                         #pragma omp parallel for default(shared)
00452 #endif
00453                         for (int i=0; i < m_header.channels; i++) {
00454                                 // inverse transform from m_wtChannel to m_channel
00455                                 if (error == NoError) {
```

```
00456                                           OSError err =
m_wtChannel[i]->InverseTransform(m_currentLevel, &m_width[i], &m_height[i],
&m_channel[i]);
00457                                           if (err != NoError) error = err;
00458                                   }
00459                                   ASSERT(m_channel[i]);
00460                           }
00461                           if (error != NoError) ReturnWithError(error);
00462
00463                           // set new level: must be done before refresh callback
00464                           m_currentLevel--;
00465
00466                           // now we have to refresh the display
00467                           if (m_cb) m_cb(m_cbArg);
00468
00469                           // now update progress
00470                           if (cb) {
00471                                   percent *= 4;
00472                                   if (m_progressMode == PM_Absolute) m_percent =
percent;
00473                                   if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
00474                           }
00475                   }
00476           }
00477 }
00478
00479 #ifdef __PGFROISUPPORT__
00489 void CPGFImage::Read(PGFRect& rect, int level /*= 0*/, CallbackPtr cb /*= nullptr*/,
void *data /*=nullptr*/) {
00490           ASSERT((level >= 0 && level < m_header.nLevels) || m_header.nLevels == 0);
// m_header.nLevels == 0: image didn't use wavelet transform
00491           ASSERT(m_decoder);
00492
00493           if (m_header.nLevels == 0 || !ROIisSupported()) {
00494                   rect.left = rect.top = 0;
00495                   rect.right = m_header.width; rect.bottom = m_header.height;
00496                   Read(level, cb, data);
00497           } else {
00498                   ASSERT(ROIisSupported());
00499                   // new encoding scheme supporting ROI
00500                   ASSERT(rect.left < m_header.width && rect.top < m_header.height);
00501
00502                   // check rectangle
00503                   if (rect.right == 0 || rect.right > m_header.width) rect.right =
m_header.width;
00504                   if (rect.bottom == 0 || rect.bottom > m_header.height) rect.bottom
= m_header.height;
00505
00506                   const int levelDiff = m_currentLevel - level;
00507                   double percent = (m_progressMode == PM_Relative) ? pow(0.25,
levelDiff) : m_percent;
00508
00509                   // check level difference
00510                   if (levelDiff <= 0) {
00511                           // it is a new read call, probably with a new ROI
00512                           m_currentLevel = m_header.nLevels;
00513                           m_decoder->SetStreamPosToData();
00514                   }
00515
00516                   // enable ROI decoding and reading
00517                   SetROI(rect);
00518
00519                   while (m_currentLevel > level) {
00520                           for (int i=0; i < m_header.channels; i++) {
00521                                   CWaveletTransform* wtChannel = m_wtChannel[i];
00522                                   ASSERT(wtChannel);
00523
00524                                   // get number of tiles and tile indices
00525                                   const UINT32 nTiles =
wtChannel->GetNofTiles(m_currentLevel); // independent of ROI
00526
00527                                   // decode file and write stream to m_wtChannel
00528                                   if (m_currentLevel == m_header.nLevels) { // last
level also has LL band
00529                                           ASSERT(nTiles == 1);
00530                                           m_decoder->GetNextMacroBlock();
```

```
00531                                       wtChannel->GetSubband(m_currentLevel,
LL)->PlaceTile(*m_decoder, m_quant);
00532                               }
00533                               for (UINT32 tileY=0; tileY < nTiles; tileY++) {
00534                                       for (UINT32 tileX=0; tileX < nTiles;
tileX++) {
00535                                               // check relevance of tile
00536                                               if
(wtChannel->TileIsRelevant(m_currentLevel, tileX, tileY)) {
00537
m_decoder->GetNextMacroBlock();
00538
wtChannel->GetSubband(m_currentLevel, HL)->PlaceTile(*m_decoder, m_quant, true, tileX,
tileY);
00539
wtChannel->GetSubband(m_currentLevel, LH)->PlaceTile(*m_decoder, m_quant, true, tileX,
tileY);
00540
wtChannel->GetSubband(m_currentLevel, HH)->PlaceTile(*m_decoder, m_quant, true, tileX,
tileY);
00541                                               } else {
00542                                                       // skip tile
00543
m_decoder->SkipTileBuffer();
00544                                               }
00545                                       }
00546                               }
00547                       }
00548
00549                       volatile OSError error = NoError; // volatile prevents
optimizations
00550 #ifdef LIBPGF_USE_OPENMP
00551                       #pragma omp parallel for default(shared)
00552 #endif
00553                       for (int i=0; i < m_header.channels; i++) {
00554                               // inverse transform from m_wtChannel to m_channel
00555                               if (error == NoError) {
00556                                       OSError err =
m_wtChannel[i]->InverseTransform(m_currentLevel, &m_width[i], &m_height[i],
&m_channel[i]);
00557                                       if (err != NoError) error = err;
00558                               }
00559                               ASSERT(m_channel[i]);
00560                       }
00561                       if (error != NoError) ReturnWithError(error);
00562
00563                       // set new level: must be done before refresh callback
00564                       m_currentLevel--;
00565
00566                       // now we have to refresh the display
00567                       if (m_cb) m_cb(m_cbArg);
00568
00569                       // now update progress
00570                       if (cb) {
00571                               percent *= 4;
00572                               if (m_progressMode == PM_Absolute) m_percent =
percent;
00573                               if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
00574                       }
00575               }
00576       }
00577 }
00578
00583 PGFRect CPGFImage::ComputeLevelROI() const {
00584       if (m_currentLevel == 0) {
00585               return m_roi;
00586       } else {
00587               const UINT32 rLeft = LevelSizeL(m_roi.left, m_currentLevel);
00588               const UINT32 rRight = LevelSizeL(m_roi.right, m_currentLevel);
00589               const UINT32 rTop = LevelSizeL(m_roi.top, m_currentLevel);
00590               const UINT32 rBottom = LevelSizeL(m_roi.bottom, m_currentLevel);
00591               return PGFRect(rLeft, rTop, rRight - rLeft, rBottom - rTop);
00592       }
00593 }
00594
00598 PGFRect CPGFImage::GetAlignedROI(int c /*= 0*/) const {
```

```
00599         PGFRect roi(0, 0, m_width[c], m_height[c]);
00600
00601         if (ROIisSupported()) {
00602                 ASSERT(m_wtChannel[c]);
00603
00604                 roi = m_wtChannel[c]->GetAlignedROI(m_currentLevel);
00605         }
00606         ASSERT(roi.Width() == m_width[c]);
00607         ASSERT(roi.Height() == m_height[c]);
00608         return roi;
00609 }
00610
00615 void CPGFImage::SetROI(PGFRect rect) {
00616         ASSERT(m_decoder);
00617         ASSERT(ROIisSupported());
00618         ASSERT(m_wtChannel[0]);
00619
00620         // store ROI for a later call of GetBitmap
00621         m_roi = rect;
00622
00623         // enable ROI decoding
00624         m_decoder->SetROI();
00625
00626         // prepare wavelet channels for using ROI
00627         m_wtChannel[0]->SetROI(rect);
00628
00629         if (m_downsample && m_header.channels > 1) {
00630                 // all further channels are downsampled, therefore downsample ROI
00631                 rect.left >>= 1;
00632                 rect.top >>= 1;
00633                 rect.right = (rect.right + 1) >> 1;
00634                 rect.bottom = (rect.bottom + 1) >> 1;
00635         }
00636         for (int i=1; i < m_header.channels; i++) {
00637                 ASSERT(m_wtChannel[i]);
00638                 m_wtChannel[i]->SetROI(rect);
00639         }
00640 }
00641
00642 #endif // __PGFROISUPPORT__
00643
00648 UINT32 CPGFImage::GetEncodedHeaderLength() const {
00649         ASSERT(m_decoder);
00650         return m_decoder->GetEncodedHeaderLength();
00651 }
00652
00660 UINT32 CPGFImage::ReadEncodedHeader(UINT8* target, UINT32 targetLen) const {
00661         ASSERT(target);
00662         ASSERT(targetLen > 0);
00663         ASSERT(m_decoder);
00664
00665         // reset stream position
00666         m_decoder->SetStreamPosToStart();
00667
00668         // compute number of bytes to read
00669         UINT32 len = __min(targetLen, GetEncodedHeaderLength());
00670
00671         // read data
00672         len = m_decoder->ReadEncodedData(target, len);
00673         ASSERT(len >= 0 && len <= targetLen);
00674
00675         return len;
00676 }
00677
00682 void CPGFImage::ResetStreamPos(bool startOfData) {
00683         m_currentLevel = 0;
00684         if (startOfData) {
00685                 ASSERT(m_decoder);
00686                 m_decoder->SetStreamPosToData();
00687         } else {
00688                 if (m_decoder) {
00689                         m_decoder->SetStreamPosToStart();
00690                 } else if (m_encoder) {
00691                         m_encoder->SetStreamPosToStart();
00692                 } else {
00693                         ASSERT(false);
00694                 }
```

```
00695         }
00696 }
00697
00707 UINT32 CPGFImage::ReadEncodedData(int level, UINT8* target, UINT32 targetLen) const
{
00708         ASSERT(level >= 0 && level < m_header.nLevels);
00709         ASSERT(target);
00710         ASSERT(targetLen > 0);
00711         ASSERT(m_decoder);
00712
00713         // reset stream position
00714         m_decoder->SetStreamPosToData();
00715
00716         // position stream
00717         UINT64 offset = 0;
00718
00719         for (int i=m_header.nLevels - 1; i > level; i--) {
00720                 offset += m_levelLength[m_header.nLevels - 1 - i];
00721         }
00722         m_decoder->Skip(offset);
00723
00724         // compute number of bytes to read
00725         UINT32 len = __min(targetLen, GetEncodedLevelLength(level));
00726
00727         // read data
00728         len = m_decoder->ReadEncodedData(target, len);
00729         ASSERT(len >= 0 && len <= targetLen);
00730
00731         return len;
00732 }
00733
00738 void CPGFImage::SetMaxValue(UINT32 maxValue) {
00739         const BYTE bpc = m_header.bpp/m_header.channels;
00740         BYTE pot = 0;
00741
00742         while(maxValue > 0) {
00743                 pot++;
00744                 maxValue >>= 1;
00745         }
00746         // store bits per channel
00747         if (pot > bpc) pot = bpc;
00748         if (pot > 31) pot = 31;
00749         m_header.usedBitsPerChannel = pot;
00750 }
00751
00756 BYTE CPGFImage::UsedBitsPerChannel() const {
00757         const BYTE bpc = m_header.bpp/m_header.channels;
00758
00759         if (bpc > 8) {
00760                 return m_header.usedBitsPerChannel;
00761         } else {
00762                 return bpc;
00763         }
00764 }
00765
00768 BYTE CPGFImage::CodecMajorVersion(BYTE version) {
00769         if (version & Version7) return 7;
00770         if (version & Version6) return 6;
00771         if (version & Version5) return 5;
00772         if (version & Version2) return 2;
00773         return 1;
00774 }
00775
00777 // Import an image from a specified image buffer.
00778 // This method is usually called before Write(...) and after SetHeader(...).
00779 // It might throw an IOException.
00780 // The absolute value of pitch is the number of bytes of an image row.
00781 // If pitch is negative, then buff points to the last row of a bottom-up image (first
byte on last row).
00782 // If pitch is positive, then buff points to the first row of a top-down image (first
byte).
00783 // The sequence of input channels in the input image buffer does not need to be the
same as expected from PGF. In case of different sequences you have to
00784 // provide a channelMap of size of expected channels (depending on image mode). For
example, PGF expects in RGB color mode a channel sequence BGR.
00785 // If your provided image buffer contains a channel sequence ARGB, then the channelMap
looks like { 3, 2, 1 }.
```

```
00786 // @param pitch The number of bytes of a row of the image buffer.
00787 // @param buff An image buffer.
00788 // @param bpp The number of bits per pixel used in image buffer.
00789 // @param channelMap A integer array containing the mapping of input channel ordering
to expected channel ordering.
00790 // @param cb A pointer to a callback procedure. The procedure is called after each
imported buffer row. If cb returns true, then it stops proceeding.
00791 // @param data Data Pointer to C++ class container to host callback procedure.
00792 void CPGFImage::ImportBitmap(int pitch, UINT8 *buff, BYTE bpp, int channelMap[] /*=
nullptr */, CallbackPtr cb /*= nullptr*/, void *data /*=nullptr*/) {
00793         ASSERT(buff);
00794         ASSERT(m_channel[0]);
00795 
00796         // color transform
00797         RgbToYuv(pitch, buff, bpp, channelMap, cb, data);
00798 
00799         if (m_downsample) {
00800                 // Subsampling of the chrominance and alpha channels
00801                 for (int i=1; i < m_header.channels; i++) {
00802                         Downsample(i);
00803                 }
00804         }
00805 }
00806 
00808 // Bilinerar Subsampling of channel ch by a factor 2
00809 // Called before Write()
00810 void CPGFImage::Downsample(int ch) {
00811         ASSERT(ch > 0);
00812 
00813         const int w = m_width[0];
00814         const int w2 = w/2;
00815         const int h2 = m_height[0]/2;
00816         const int oddW = w%2;                           // don't use bool ->
problems with MaxSpeed optimization
00817         const int oddH = m_height[0]%2;         // "
00818         int loPos = 0;
00819         int hiPos = w;
00820         int sampledPos = 0;
00821         DataT* buff = m_channel[ch]; ASSERT(buff);
00822 
00823         for (int i=0; i < h2; i++) {
00824                 for (int j=0; j < w2; j++) {
00825                         // compute average of pixel block
00826                         buff[sampledPos] = (buff[loPos] + buff[loPos + 1] +
buff[hiPos] + buff[hiPos + 1]) >> 2;
00827                         loPos += 2; hiPos += 2;
00828                         sampledPos++;
00829                 }
00830                 if (oddW) {
00831                         buff[sampledPos] = (buff[loPos] + buff[hiPos]) >> 1;
00832                         loPos++; hiPos++;
00833                         sampledPos++;
00834                 }
00835                 loPos += w; hiPos += w;
00836         }
00837         if (oddH) {
00838                 for (int j=0; j < w2; j++) {
00839                         buff[sampledPos] = (buff[loPos] + buff[loPos+1]) >> 1;
00840                         loPos += 2; hiPos += 2;
00841                         sampledPos++;
00842                 }
00843                 if (oddW) {
00844                         buff[sampledPos] = buff[loPos];
00845                 }
00846         }
00847 
00848         // downsampled image has half width and half height
00849         m_width[ch] = (m_width[ch] + 1)/2;
00850         m_height[ch] = (m_height[ch] + 1)/2;
00851 }
00852 
00854 void CPGFImage::ComputeLevels() {
00855         const int maxThumbnailWidth = 20*FilterSize;
00856         const int m = __min(m_header.width, m_header.height);
00857         int s = m;
00858 
00859         if (m_header.nLevels < 1 || m_header.nLevels > MaxLevel) {
```

```
00860                   m_header.nLevels = 1;
00861                   // compute a good value depending on the size of the image
00862                   while (s > maxThumbnailWidth) {
00863                           m_header.nLevels++;
00864                           s >>= 1;
00865                   }
00866           }
00867
00868           int levels = m_header.nLevels; // we need a signed value during level
reduction
00869
00870           // reduce number of levels if the image size is smaller than
FilterSize*(2^levels)
00871           s = FilterSize*(1 << levels);   // must be at least the double filter size
because of subsampling
00872           while (m < s) {
00873                   levels--;
00874                   s >>= 1;
00875           }
00876           if (levels > MaxLevel) m_header.nLevels = MaxLevel;
00877           else if (levels < 0) m_header.nLevels = 0;
00878           else m_header.nLevels = (UINT8)levels;
00879
00880           // used in Write when PM_Absolute
00881           m_percent = pow(0.25, m_header.nLevels);
00882
00883           ASSERT(0 <= m_header.nLevels && m_header.nLevels <= MaxLevel);
00884 }
00885
00894 void CPGFImage::SetHeader(const PGFHeader& header, BYTE flags /*=0*/, const UINT8*
userData /*= 0*/, UINT32 userDataLength /*= 0*/) {
00895           ASSERT(!m_decoder);     // current image must be closed
00896           ASSERT(header.quality <= MaxQuality);
00897           ASSERT(userDataLength <= MaxUserDataSize);
00898
00899           // init state
00900 #ifdef __PGFROISUPPORT__
00901           m_streamReinitialized = false;
00902 #endif
00903
00904           // init preHeader
00905           memcpy(m_preHeader.magic, PGFMagic, 3);
00906           m_preHeader.version = PGFVersion | flags;
00907           m_preHeader.hSize = HeaderSize;
00908
00909           // copy header
00910           memcpy(&m_header, &header, HeaderSize);
00911
00912           // check quality
00913           if (m_header.quality > MaxQuality) m_header.quality = MaxQuality;
00914
00915           // complete header
00916           CompleteHeader();
00917
00918           // check and set number of levels
00919           ComputeLevels();
00920
00921           // check for downsample
00922           if (m_header.quality > DownsampleThreshold &&  (m_header.mode ==
ImageModeRGBColor ||
00923
m_header.mode == ImageModeRGBA ||
00924
m_header.mode == ImageModeRGB48 ||
00925
m_header.mode == ImageModeCMYKColor ||
00926
m_header.mode == ImageModeCMYK64 ||
00927
m_header.mode == ImageModeLabColor ||
00928
m_header.mode == ImageModeLab48)) {
00929                   m_downsample = true;
00930                   m_quant = m_header.quality - 1;
00931           } else {
00932                   m_downsample = false;
00933                   m_quant = m_header.quality;
```

```
00934         }
00935
00936         // update header size and copy user data
00937         if (m_header.mode == ImageModeIndexedColor) {
00938                 // update header size
00939                 m_preHeader.hSize += ColorTableSize;
00940         }
00941         if (userDataLength && userData) {
00942                 if (userDataLength > MaxUserDataSize) userDataLength =
MaxUserDataSize;
00943                 m_postHeader.userData = new(std::nothrow) UINT8[userDataLength];
00944                 if (!m_postHeader.userData) ReturnWithError(InsufficientMemory);
00945                 m_postHeader.userDataLen = m_postHeader.cachedUserDataLen =
userDataLength;
00946                 memcpy(m_postHeader.userData, userData, userDataLength);
00947                 // update header size
00948                 m_preHeader.hSize += userDataLength;
00949         }
00950
00951         // allocate channels
00952         for (int i=0; i < m_header.channels; i++) {
00953                 // set current width and height
00954                 m_width[i] = m_header.width;
00955                 m_height[i] = m_header.height;
00956
00957                 // allocate channels
00958                 ASSERT(!m_channel[i]);
00959                 m_channel[i] = new(std::nothrow)
DataT[m_header.width*m_header.height];
00960                 if (!m_channel[i]) {
00961                         if (i) i--;
00962                         while(i) {
00963                                 delete[] m_channel[i]; m_channel[i] = 0;
00964                                 i--;
00965                         }
00966                         ReturnWithError(InsufficientMemory);
00967                 }
00968         }
00969 }
00970
00979 UINT32 CPGFImage::WriteHeader(CPGFStream* stream) {
00980         ASSERT(m_header.nLevels <= MaxLevel);
00981         ASSERT(m_header.quality <= MaxQuality); // quality is already initialized
00982
00983         if (m_header.nLevels > 0) {
00984                 volatile OSError error = NoError; // volatile prevents optimizations
00985                 // create new wt channels
00986 #ifdef LIBPGF_USE_OPENMP
00987                 #pragma omp parallel for default(shared)
00988 #endif
00989                 for (int i=0; i < m_header.channels; i++) {
00990                         DataT *temp = nullptr;
00991                         if (error == NoError) {
00992                                 if (m_wtChannel[i]) {
00993                                         ASSERT(m_channel[i]);
00994                                         // copy m_channel to temp
00995                                         int size = m_height[i]*m_width[i];
00996                                         temp = new(std::nothrow) DataT[size];
00997                                         if (temp) {
00998                                                 memcpy(temp, m_channel[i],
size*DataTSize);
00999                                                 delete m_wtChannel[i];  // also
deletes m_channel
01000                                                 m_channel[i] = nullptr;
01001                                         } else {
01002                                                 error = InsufficientMemory;
01003                                         }
01004                                 }
01005                                 if (error == NoError) {
01006                                         if (temp) {
01007                                                 ASSERT(!m_channel[i]);
01008                                                 m_channel[i] = temp;
01009                                         }
01010                                         m_wtChannel[i] = new
CWaveletTransform(m_width[i], m_height[i], m_header.nLevels, m_channel[i]);
01011                                         if (m_wtChannel[i]) {
01012                                         #ifdef __PGFROISUPPORT__
```

```
01013
m_wtChannel[i]->SetROI(PGFRect(0, 0, m_width[i], m_height[i]));
01014                                   #endif
01015
01016                                           // wavelet subband decomposition
01017                                           for (int l=0; error == NoError &&
l < m_header.nLevels; l++) {
01018                                                   OSError err =
m_wtChannel[i]->ForwardTransform(l, m_quant);
01019                                                   if (err != NoError) error
= err;
01020                                           }
01021                                   } else {
01022                                           delete[] m_channel[i];
01023                                           error = InsufficientMemory;
01024                                   }
01025                           }
01026                   }
01027           }
01028           if (error != NoError) {
01029                   // free already allocated memory
01030                   for (int i=0; i < m_header.channels; i++) {
01031                           delete m_wtChannel[i];
01032                   }
01033                   ReturnWithError(error);
01034           }
01035
01036           m_currentLevel = m_header.nLevels;
01037
01038           // create encoder, write headers and user data, but not level-length
area
01039           m_encoder = new CEncoder(stream, m_preHeader, m_header,
m_postHeader, m_userDataPos, m_useOMPinEncoder);
01040           if (m_favorSpeedOverSize) m_encoder->FavorSpeedOverSize();
01041
01042   #ifdef __PGFROISUPPORT__
01043           if (ROIisSupported()) {
01044                   // new encoding scheme supporting ROI
01045                   m_encoder->SetROI();
01046           }
01047   #endif
01048
01049   } else {
01050           // very small image: we don't use DWT and encoding
01051
01052           // create encoder, write headers and user data, but not level-length
area
01053           m_encoder = new CEncoder(stream, m_preHeader, m_header,
m_postHeader, m_userDataPos, m_useOMPinEncoder);
01054   }
01055
01056   INT64 nBytes = m_encoder->ComputeHeaderLength();
01057   return (nBytes > 0) ? (UINT32)nBytes : 0;
01058 }
01059
01061 // Encode and write next level of a PGF image at current stream position.
01062 // A PGF image is structered in levels, numbered between 0 and Levels() - 1.
01063 // Each level can be seen as a single image, containing the same content
01064 // as all other levels, but in a different size (width, height).
01065 // The image size at level i is double the size (width, height) of the image at level
i+1.
01066 // The image at level 0 contains the original size.
01067 // It might throw an IOException.
01068 void CPGFImage::WriteLevel() {
01069   ASSERT(m_encoder);
01070   ASSERT(m_currentLevel > 0);
01071   ASSERT(m_header.nLevels > 0);
01072
01073 #ifdef __PGFROISUPPORT__
01074   if (ROIisSupported()) {
01075           const int lastChannel = m_header.channels - 1;
01076
01077           for (int i=0; i < m_header.channels; i++) {
01078                   // get number of tiles and tile indices
01079                   const UINT32 nTiles =
m_wtChannel[i]->GetNofTiles(m_currentLevel);
01080                   const UINT32 lastTile = nTiles - 1;
```

```
01081
01082                         if (m_currentLevel == m_header.nLevels) {
01083                                 // last level also has LL band
01084                                 ASSERT(nTiles == 1);
01085                                 m_wtChannel[i]->GetSubband(m_currentLevel,
LL)->ExtractTile(*m_encoder);
01086                                 m_encoder->EncodeTileBuffer(); // encode macro
block with tile-end = true
01087                         }
01088                         for (UINT32 tileY=0; tileY < nTiles; tileY++) {
01089                                 for (UINT32 tileX=0; tileX < nTiles; tileX++) {
01090                                         // extract tile to macro block and encode
already filled macro blocks with tile-end = false
01091
m_wtChannel[i]->GetSubband(m_currentLevel, HL)->ExtractTile(*m_encoder, true, tileX,
tileY);
01092
m_wtChannel[i]->GetSubband(m_currentLevel, LH)->ExtractTile(*m_encoder, true, tileX,
tileY);
01093
m_wtChannel[i]->GetSubband(m_currentLevel, HH)->ExtractTile(*m_encoder, true, tileX,
tileY);
01094                                         if (i == lastChannel && tileY == lastTile
&& tileX == lastTile) {
01095                                                 // all necessary data are
buffered. next call of EncodeTileBuffer will write the last piece of data of the current
level.
01096
m_encoder->SetEncodedLevel(--m_currentLevel);
01097                                         }
01098                                         m_encoder->EncodeTileBuffer(); // encode
last macro block with tile-end = true
01099                                 }
01100                         }
01101                 }
01102         } else
01103 #endif
01104         {
01105                 for (int i=0; i < m_header.channels; i++) {
01106                         ASSERT(m_wtChannel[i]);
01107                         if (m_currentLevel == m_header.nLevels) {
01108                                 // last level also has LL band
01109                                 m_wtChannel[i]->GetSubband(m_currentLevel,
LL)->ExtractTile(*m_encoder);
01110                         }
01111                         //encoder.EncodeInterleaved(m_wtChannel[i],
m_currentLevel, m_quant); // until version 4
01112                         m_wtChannel[i]->GetSubband(m_currentLevel,
HL)->ExtractTile(*m_encoder); // since version 5
01113                         m_wtChannel[i]->GetSubband(m_currentLevel,
LH)->ExtractTile(*m_encoder); // since version 5
01114                         m_wtChannel[i]->GetSubband(m_currentLevel,
HH)->ExtractTile(*m_encoder);
01115                 }
01116
01117                 // all necessary data are buffered. next call of EncodeBuffer will
write the last piece of data of the current level.
01118                 m_encoder->SetEncodedLevel(--m_currentLevel);
01119         }
01120 }
01121
01123 // Return written levelLength bytes
01124 UINT32 CPGFImage::UpdatePostHeaderSize() {
01125         ASSERT(m_encoder);
01126
01127         INT64 offset = m_encoder->ComputeOffset(); ASSERT(offset >= 0);
01128
01129         if (offset > 0) {
01130                 // update post-header size and rewrite pre-header
01131                 m_preHeader.hSize += (UINT32)offset;
01132                 m_encoder->UpdatePostHeaderSize(m_preHeader);
01133         }
01134
01135         // write dummy levelLength into stream
01136         return m_encoder->WriteLevelLength(m_levelLength);
01137 }
01138
```

```
01150 UINT32 CPGFImage::WriteImage(CPGFStream* stream, CallbackPtr cb /*= nullptr*/, void
*data /*= nullptr*/) {
01151         ASSERT(stream);
01152         ASSERT(m_preHeader.hSize);
01153
01154         int levels = m_header.nLevels;
01155         double percent = pow(0.25, levels);
01156
01157         // update post-header size, rewrite pre-header, and write dummy levelLength
01158         UINT32 nWrittenBytes = UpdatePostHeaderSize();
01159
01160         if (levels == 0) {
01161                 // for very small images: write channels uncoded
01162                 for (int c=0; c < m_header.channels; c++) {
01163                         const UINT32 size = m_width[c]*m_height[c];
01164
01165                         // write channel data into stream
01166                         for (UINT32 i=0; i < size; i++) {
01167                                 int count = DataTSize;
01168                                 stream->Write(&count, &m_channel[c][i]);
01169                         }
01170                 }
01171
01172                 // now update progress
01173                 if (cb) {
01174                         if ((*cb)(1, true, data)) ReturnWithError(EscapePressed);
01175                 }
01176
01177         } else {
01178                 // encode quantized wavelet coefficients and write to PGF file
01179                 // encode subbands, higher levels first
01180                 // color channels are interleaved
01181
01182                 // encode all levels
01183                 for (m_currentLevel = levels; m_currentLevel > 0; ) {
01184                         WriteLevel(); // decrements m_currentLevel
01185
01186                         // now update progress
01187                         if (cb) {
01188                                 percent *= 4;
01189                                 if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
01190                         }
01191                 }
01192
01193                 // flush encoder and write level lengths
01194                 m_encoder->Flush();
01195         }
01196
01197         // update level lengths
01198         nWrittenBytes += m_encoder->UpdateLevelLength(); // return written image
bytes
01199
01200         // delete encoder
01201         delete m_encoder; m_encoder = nullptr;
01202
01203         ASSERT(!m_encoder);
01204
01205         return nWrittenBytes;
01206 }
01207
01221 void CPGFImage::Write(CPGFStream* stream, UINT32* nWrittenBytes /*= nullptr*/,
CallbackPtr cb /*= nullptr*/, void *data /*=nullptr*/) {
01222         ASSERT(stream);
01223         ASSERT(m_preHeader.hSize);
01224
01225         // create wavelet transform channels and encoder
01226         UINT32 nBytes = WriteHeader(stream);
01227
01228         // write image
01229         nBytes += WriteImage(stream, cb, data);
01230
01231         // return written bytes
01232         if (nWrittenBytes) *nWrittenBytes += nBytes;
01233 }
01234
01235 #ifdef __PGFROISUPPORT__
```

```
01237 // Encode and write down to given level at current stream position.
01238 // A PGF image is structered in levels, numbered between 0 and Levels() - 1.
01239 // Each level can be seen as a single image, containing the same content
01240 // as all other levels, but in a different size (width, height).
01241 // The image size at level i is double the size (width, height) of the image at level
i+1.
01242 // The image at level 0 contains the original size.
01243 // Precondition: the PGF image contains a valid header (see also SetHeader(...)) and
WriteHeader() has been called before.
01244 // The ROI encoding scheme is used.
01245 // It might throw an IOException.
01246 // @param level The image level of the resulting image in the internal image buffer.
01247 // @param cb A pointer to a callback procedure. The procedure is called after writing
a single level. If cb returns true, then it stops proceeding.
01248 // @param data Data Pointer to C++ class container to host callback procedure.
01249 // @return The number of bytes written into stream.
01250 UINT32 CPGFImage::Write(int level, CallbackPtr cb /*= nullptr*/, void *data
/*=nullptr*/) {
01251         ASSERT(m_header.nLevels > 0);
01252         ASSERT(0 <= level && level < m_header.nLevels);
01253         ASSERT(m_encoder);
01254         ASSERT(ROIisSupported());
01255
01256         const int levelDiff = m_currentLevel - level;
01257         double percent = (m_progressMode == PM_Relative) ? pow(0.25, levelDiff) :
m_percent;
01258         UINT32 nWrittenBytes = 0;
01259
01260         if (m_currentLevel == m_header.nLevels) {
01261                 // update post-header size, rewrite pre-header, and write dummy
levelLength
01262                 nWrittenBytes = UpdatePostHeaderSize();
01263         } else {
01264                 // prepare for next level: save current file position, because the
stream might have been reinitialized
01265                 if (m_encoder->ComputeBufferLength()) {
01266                         m_streamReinitialized = true;
01267                 }
01268         }
01269
01270         // encoding scheme with ROI
01271         while (m_currentLevel > level) {
01272                 WriteLevel();   // decrements m_currentLevel
01273
01274                 if (m_levelLength) {
01275                         nWrittenBytes += m_levelLength[m_header.nLevels -
m_currentLevel - 1];
01276                 }
01277
01278                 // now update progress
01279                 if (cb) {
01280                         percent *= 4;
01281                         if (m_progressMode == PM_Absolute) m_percent = percent;
01282                         if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
01283                 }
01284         }
01285
01286         // automatically closing
01287         if (m_currentLevel == 0) {
01288                 if (!m_streamReinitialized) {
01289                         // don't write level lengths, if the stream position changed
inbetween two Write operations
01290                         m_encoder->UpdateLevelLength();
01291                 }
01292                 // delete encoder
01293                 delete m_encoder; m_encoder = nullptr;
01294         }
01295
01296         return nWrittenBytes;
01297 }
01298 #endif // __PGFROISUPPORT__
01299
01300
01302 // Check for valid import image mode.
01303 // @param mode Image mode
01304 // @return True if an image of given mode can be imported with ImportBitmap(...)
```

```
01305 bool CPGFImage::ImportIsSupported(BYTE mode) {
01306         size_t size = DataTSize;
01307
01308         if (size >= 2) {
01309                 switch(mode) {
01310                         case ImageModeBitmap:
01311                         case ImageModeIndexedColor:
01312                         case ImageModeGrayScale:
01313                         case ImageModeRGBColor:
01314                         case ImageModeCMYKColor:
01315                         case ImageModeHSLColor:
01316                         case ImageModeHSBColor:
01317                         //case ImageModeDuotone:
01318                         case ImageModeLabColor:
01319                         case ImageModeRGB12:
01320                         case ImageModeRGB16:
01321                         case ImageModeRGBA:
01322                                 return true;
01323                 }
01324         }
01325         if (size >= 3) {
01326                 switch(mode) {
01327                         case ImageModeGray16:
01328                         case ImageModeRGB48:
01329                         case ImageModeLab48:
01330                         case ImageModeCMYK64:
01331                         //case ImageModeDuotone16:
01332                                 return true;
01333                 }
01334         }
01335         if (size >=4) {
01336                 switch(mode) {
01337                         case ImageModeGray32:
01338                                 return true;
01339                 }
01340         }
01341         return false;
01342 }
01343
01350 void CPGFImage::GetColorTable(UINT32 iFirstColor, UINT32 nColors, RGBQUAD*
prgbColors) const {
01351         if (iFirstColor + nColors > ColorTableLen)
ReturnWithError(ColorTableError);
01352
01353         for (UINT32 i=iFirstColor, j=0; j < nColors; i++, j++) {
01354                 prgbColors[j] = m_postHeader.clut[i];
01355         }
01356 }
01357
01364 void CPGFImage::SetColorTable(UINT32 iFirstColor, UINT32 nColors, const RGBQUAD*
prgbColors) {
01365         if (iFirstColor + nColors > ColorTableLen)
ReturnWithError(ColorTableError);
01366
01367         for (UINT32 i=iFirstColor, j=0; j < nColors; i++, j++) {
01368                 m_postHeader.clut[i] = prgbColors[j];
01369         }
01370 }
01371
01373 // Buffer transform from interleaved to channel seperated format
01374 // the absolute value of pitch is the number of bytes of an image row
01375 // if pitch is negative, then buff points to the last row of a bottom-up image (first
byte on last row)
01376 // if pitch is positive, then buff points to the first row of a top-down image (first
byte)
01377 // bpp is the number of bits per pixel used in image buffer buff
01378 //
01379 // RGB is transformed into YUV format (ordering of buffer data is BGR[A])
01380 // Y = (R + 2*G + B)/4 -128
01381 // U = R - G
01382 // V = B - G
01383 //
01384 // Since PGF Codec version 2.0 images are stored in top-down direction
01385 //
01386 // The sequence of input channels in the input image buffer does not need to be the
same as expected from PGF. In case of different sequences you have to
```

```
01387 // provide a channelMap of size of expected channels (depending on image mode). For
example, PGF expects in RGB color mode a channel sequence BGR.
01388 // If your provided image buffer contains a channel sequence ARGB, then the channelMap
looks like { 3, 2, 1 }.
01389 void CPGFImage::RgbToYuv(int pitch, UINT8* buff, BYTE bpp, int channelMap[],
CallbackPtr cb, void *data /*=nullptr*/) {
01390         ASSERT(buff);
01391         UINT32 yPos = 0, cnt = 0;
01392         double percent = 0;
01393         const double dP = 1.0/m_header.height;
01394         int defMap[] = { 0, 1, 2, 3, 4, 5, 6, 7 };
ASSERT(sizeof(defMap)/sizeof(defMap[0]) == MaxChannels);
01395
01396         if (channelMap == nullptr) channelMap = defMap;
01397
01398         switch(m_header.mode) {
01399         case ImageModeBitmap:
01400                 {
01401                         ASSERT(m_header.channels == 1);
01402                         ASSERT(m_header.bpp == 1);
01403                         ASSERT(bpp == 1);
01404
01405                         const UINT32 w = m_header.width;
01406                         const UINT32 w2 = (m_header.width + 7)/8;
01407                         DataT* y = m_channel[0]; ASSERT(y);
01408
01409                         // new unpacked version since version 7
01410                         for (UINT32 h = 0; h < m_header.height; h++) {
01411                                 if (cb) {
01412                                         if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
01413                                         percent += dP;
01414                                 }
01415                                 cnt = 0;
01416                                 for (UINT32 j = 0; j < w2; j++) {
01417                                         UINT8 byte = buff[j];
01418                                         for (int k = 0; k < 8; k++) {
01419                                                 UINT8 bit = (byte & 0x80) >> 7;
01420                                                 if (cnt < w) y[yPos++] = bit;
01421                                                 byte <<= 1;
01422                                                 cnt++;
01423                                         }
01424                                 }
01425                                 buff += pitch;
01426                         }
01427                         /* old version: packed values: 8 pixels in 1 byte
01428                         for (UINT32 h = 0; h < m_header.height; h++) {
01429                                 if (cb) {
01430                                         if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
01431                                         percent += dP;
01432                                 }
01433
01434                                 for (UINT32 j = 0; j < w2; j++) {
01435                                         y[yPos++] = buff[j] - YUVoffset8;
01436                                 }
01437                                 // version 5 and 6
01438                                 // for (UINT32 j = w2; j < w; j++) {
01439                                 //      y[yPos++] = YUVoffset8;
01440                                 //}
01441                                 buff += pitch;
01442                         }
01443                         */
01444                 }
01445                 break;
01446         case ImageModeIndexedColor:
01447         case ImageModeGrayScale:
01448         case ImageModeHSLColor:
01449         case ImageModeHSBColor:
01450         case ImageModeLabColor:
01451                 {
01452                         ASSERT(m_header.channels >= 1);
01453                         ASSERT(m_header.bpp == m_header.channels*8);
01454                         ASSERT(bpp%8 == 0);
01455                         const int channels = bpp/8; ASSERT(channels >=
m_header.channels);
01456
```

```
01457                     for (UINT32 h=0; h < m_header.height; h++) {
01458                         if (cb) {
01459                             if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
01460                             percent += dP;
01461                         }
01462
01463                         cnt = 0;
01464                         for (UINT32 w=0; w < m_header.width; w++) {
01465                             for (int c=0; c < m_header.channels; c++)
{
01466                                 m_channel[c][yPos] = buff[cnt +
channelMap[c]] - YUVoffset8;
01467                             }
01468                             cnt += channels;
01469                             yPos++;
01470                         }
01471                         buff += pitch;
01472                     }
01473                 }
01474                 break;
01475         case ImageModeGray16:
01476         case ImageModeLab48:
01477                 {
01478                     ASSERT(m_header.channels >= 1);
01479                     ASSERT(m_header.bpp == m_header.channels*16);
01480                     ASSERT(bpp%16 == 0);
01481
01482                     UINT16 *buff16 = (UINT16 *)buff;
01483                     const int pitch16 = pitch/2;
01484                     const int channels = bpp/16; ASSERT(channels >=
m_header.channels);
01485                     const int shift = 16 - UsedBitsPerChannel(); ASSERT(shift
>= 0);
01486                     const DataT yuvOffset16 = 1 << (UsedBitsPerChannel() - 1);
01487
01488                     for (UINT32 h=0; h < m_header.height; h++) {
01489                         if (cb) {
01490                             if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
01491                             percent += dP;
01492                         }
01493
01494                         cnt = 0;
01495                         for (UINT32 w=0; w < m_header.width; w++) {
01496                             for (int c=0; c < m_header.channels; c++)
{
01497                                 m_channel[c][yPos] = (buff16[cnt
+ channelMap[c]] >> shift) - yuvOffset16;
01498                             }
01499                             cnt += channels;
01500                             yPos++;
01501                         }
01502                         buff16 += pitch16;
01503                     }
01504                 }
01505                 break;
01506         case ImageModeRGBColor:
01507                 {
01508                     ASSERT(m_header.channels == 3);
01509                     ASSERT(m_header.bpp == m_header.channels*8);
01510                     ASSERT(bpp%8 == 0);
01511
01512                     DataT* y = m_channel[0]; ASSERT(y);
01513                     DataT* u = m_channel[1]; ASSERT(u);
01514                     DataT* v = m_channel[2]; ASSERT(v);
01515                     const int channels = bpp/8; ASSERT(channels >=
m_header.channels);
01516                     UINT8 b, g, r;
01517
01518                     for (UINT32 h=0; h < m_header.height; h++) {
01519                         if (cb) {
01520                             if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
01521                             percent += dP;
01522                         }
01523
```

```
01524                                   cnt = 0;
01525                                   for (UINT32 w=0; w < m_header.width; w++) {
01526                                           b = buff[cnt + channelMap[0]];
01527                                           g = buff[cnt + channelMap[1]];
01528                                           r = buff[cnt + channelMap[2]];
01529                                           // Yuv
01530                                           y[yPos] = ((b + (g << 1) + r) >> 2) -
YUVoffset8;
01531                                           u[yPos] = r - g;
01532                                           v[yPos] = b - g;
01533                                           yPos++;
01534                                           cnt += channels;
01535                                   }
01536                                   buff += pitch;
01537                           }
01538                   }
01539                   break;
01540           case ImageModeRGB48:
01541                   {
01542                           ASSERT(m_header.channels == 3);
01543                           ASSERT(m_header.bpp == m_header.channels*16);
01544                           ASSERT(bpp%16 == 0);
01545
01546                           UINT16 *buff16 = (UINT16 *)buff;
01547                           const int pitch16 = pitch/2;
01548                           const int channels = bpp/16; ASSERT(channels >=
m_header.channels);
01549                           const int shift = 16 - UsedBitsPerChannel(); ASSERT(shift
>= 0);
01550                           const DataT yuvOffset16 = 1 << (UsedBitsPerChannel() - 1);
01551
01552                           DataT* y = m_channel[0]; ASSERT(y);
01553                           DataT* u = m_channel[1]; ASSERT(u);
01554                           DataT* v = m_channel[2]; ASSERT(v);
01555                           UINT16 b, g, r;
01556
01557                           for (UINT32 h=0; h < m_header.height; h++) {
01558                                   if (cb) {
01559                                           if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
01560                                           percent += dP;
01561                                   }
01562
01563                                   cnt = 0;
01564                                   for (UINT32 w=0; w < m_header.width; w++) {
01565                                           b = buff16[cnt + channelMap[0]] >> shift;
01566                                           g = buff16[cnt + channelMap[1]] >> shift;
01567                                           r = buff16[cnt + channelMap[2]] >> shift;
01568                                           // Yuv
01569                                           y[yPos] = ((b + (g << 1) + r) >> 2) -
yuvOffset16;
01570                                           u[yPos] = r - g;
01571                                           v[yPos] = b - g;
01572                                           yPos++;
01573                                           cnt += channels;
01574                                   }
01575                                   buff16 += pitch16;
01576                           }
01577                   }
01578                   break;
01579           case ImageModeRGBA:
01580           case ImageModeCMYKColor:
01581                   {
01582                           ASSERT(m_header.channels == 4);
01583                           ASSERT(m_header.bpp == m_header.channels*8);
01584                           ASSERT(bpp%8 == 0);
01585                           const int channels = bpp/8; ASSERT(channels >=
m_header.channels);
01586
01587                           DataT* y = m_channel[0]; ASSERT(y);
01588                           DataT* u = m_channel[1]; ASSERT(u);
01589                           DataT* v = m_channel[2]; ASSERT(v);
01590                           DataT* a = m_channel[3]; ASSERT(a);
01591                           UINT8 b, g, r;
01592
01593                           for (UINT32 h=0; h < m_header.height; h++) {
01594                                   if (cb) {
```

```
01595                                       if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
01596                                       percent += dP;
01597                               }
01598
01599                               cnt = 0;
01600                               for (UINT32 w=0; w < m_header.width; w++) {
01601                                       b = buff[cnt + channelMap[0]];
01602                                       g = buff[cnt + channelMap[1]];
01603                                       r = buff[cnt + channelMap[2]];
01604                                       // Yuv
01605                                       y[yPos] = ((b + (g << 1) + r) >> 2) -
YUVoffset8;
01606                                       u[yPos] = r - g;
01607                                       v[yPos] = b - g;
01608                                       a[yPos++] = buff[cnt + channelMap[3]] -
YUVoffset8;
01609                                       cnt += channels;
01610                               }
01611                               buff += pitch;
01612                       }
01613               }
01614               break;
01615       case ImageModeCMYK64:
01616               {
01617                       ASSERT(m_header.channels == 4);
01618                       ASSERT(m_header.bpp == m_header.channels*16);
01619                       ASSERT(bpp%16 == 0);
01620
01621                       UINT16 *buff16 = (UINT16 *)buff;
01622                       const int pitch16 = pitch/2;
01623                       const int channels = bpp/16; ASSERT(channels >=
m_header.channels);
01624                       const int shift = 16 - UsedBitsPerChannel(); ASSERT(shift
>= 0);
01625                       const DataT yuvOffset16 = 1 << (UsedBitsPerChannel() - 1);
01626
01627                       DataT* y = m_channel[0]; ASSERT(y);
01628                       DataT* u = m_channel[1]; ASSERT(u);
01629                       DataT* v = m_channel[2]; ASSERT(v);
01630                       DataT* a = m_channel[3]; ASSERT(a);
01631                       UINT16 b, g, r;
01632
01633                       for (UINT32 h=0; h < m_header.height; h++) {
01634                               if (cb) {
01635                                       if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
01636                                       percent += dP;
01637                               }
01638
01639                               cnt = 0;
01640                               for (UINT32 w=0; w < m_header.width; w++) {
01641                                       b = buff16[cnt + channelMap[0]] >> shift;
01642                                       g = buff16[cnt + channelMap[1]] >> shift;
01643                                       r = buff16[cnt + channelMap[2]] >> shift;
01644                                       // Yuv
01645                                       y[yPos] = ((b + (g << 1) + r) >> 2) -
yuvOffset16;
01646                                       u[yPos] = r - g;
01647                                       v[yPos] = b - g;
01648                                       a[yPos++] = (buff16[cnt + channelMap[3]]
>> shift) - yuvOffset16;
01649                                       cnt += channels;
01650                               }
01651                               buff16 += pitch16;
01652                       }
01653               }
01654               break;
01655 #ifdef __PGF32SUPPORT__
01656       case ImageModeGray32:
01657               {
01658                       ASSERT(m_header.channels == 1);
01659                       ASSERT(m_header.bpp == 32);
01660                       ASSERT(bpp == 32);
01661                       ASSERT(DataTSize == sizeof(UINT32));
01662
01663                       DataT* y = m_channel[0]; ASSERT(y);
```

```
01664
01665                           UINT32 *buff32 = (UINT32 *)buff;
01666                           const int pitch32 = pitch/4;
01667                           const int shift = 31 - UsedBitsPerChannel(); ASSERT(shift
>= 0);
01668                           const DataT yuvOffset31 = 1 << (UsedBitsPerChannel() - 1);
01669
01670                           for (UINT32 h=0; h < m_header.height; h++) {
01671                                   if (cb) {
01672                                           if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
01673                                           percent += dP;
01674                                   }
01675
01676                                   for (UINT32 w=0; w < m_header.width; w++) {
01677                                           y[yPos++] = (buff32[w] >> shift) -
yuvOffset31;
01678                                   }
01679                                   buff32 += pitch32;
01680                           }
01681                   }
01682                   break;
01683 #endif
01684           case ImageModeRGB12:
01685                   {
01686                           ASSERT(m_header.channels == 3);
01687                           ASSERT(m_header.bpp == m_header.channels*4);
01688                           ASSERT(bpp == m_header.channels*4);
01689
01690                           DataT* y = m_channel[0]; ASSERT(y);
01691                           DataT* u = m_channel[1]; ASSERT(u);
01692                           DataT* v = m_channel[2]; ASSERT(v);
01693
01694                           UINT8 rgb = 0, b, g, r;
01695
01696                           for (UINT32 h=0; h < m_header.height; h++) {
01697                                   if (cb) {
01698                                           if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
01699                                           percent += dP;
01700                                   }
01701
01702                                   cnt = 0;
01703                                   for (UINT32 w=0; w < m_header.width; w++) {
01704                                           if (w%2 == 0) {
01705                                                   // even pixel position
01706                                                   rgb = buff[cnt];
01707                                                   b = rgb & 0x0F;
01708                                                   g = (rgb & 0xF0) >> 4;
01709                                                   cnt++;
01710                                                   rgb = buff[cnt];
01711                                                   r = rgb & 0x0F;
01712                                           } else {
01713                                                   // odd pixel position
01714                                                   b = (rgb & 0xF0) >> 4;
01715                                                   cnt++;
01716                                                   rgb = buff[cnt];
01717                                                   g = rgb & 0x0F;
01718                                                   r = (rgb & 0xF0) >> 4;
01719                                                   cnt++;
01720                                           }
01721
01722                                           // Yuv
01723                                           y[yPos] = ((b + (g << 1) + r) >> 2) -
YUVoffset4;
01724                                           u[yPos] = r - g;
01725                                           v[yPos] = b - g;
01726                                           yPos++;
01727                                   }
01728                                   buff += pitch;
01729                           }
01730                   }
01731                   break;
01732           case ImageModeRGB16:
01733                   {
01734                           ASSERT(m_header.channels == 3);
01735                           ASSERT(m_header.bpp == 16);
```

```
01736                   ASSERT(bpp == 16);
01737
01738                   DataT* y = m_channel[0]; ASSERT(y);
01739                   DataT* u = m_channel[1]; ASSERT(u);
01740                   DataT* v = m_channel[2]; ASSERT(v);
01741
01742                   UINT16 *buff16 = (UINT16 *)buff;
01743                   UINT16 rgb, b, g, r;
01744                   const int pitch16 = pitch/2;
01745
01746                   for (UINT32 h=0; h < m_header.height; h++) {
01747                           if (cb) {
01748                                   if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
01749                                   percent += dP;
01750                           }
01751                           for (UINT32 w=0; w < m_header.width; w++) {
01752                                   rgb = buff16[w];
01753                                   r = (rgb & 0xF800) >> 10;       // highest
5 bits
01754                                   g = (rgb & 0x07E0) >> 5;        // middle
6 bits
01755                                   b = (rgb & 0x001F) << 1;        // lowest
5 bits
01756                                   // Yuv
01757                                   y[yPos] = ((b + (g << 1) + r) >> 2) -
YUVoffset6;
01758                                   u[yPos] = r - g;
01759                                   v[yPos] = b - g;
01760                                   yPos++;
01761                           }
01762
01763                           buff16 += pitch16;
01764                   }
01765           }
01766           break;
01767   default:
01768           ASSERT(false);
01769   }
01770 }
01771
01773 // Get image data in interleaved format: (ordering of RGB data is BGR[A])
01774 // Upsampling, YUV to RGB transform and interleaving are done here to reduce the number
01775 // of passes over the data.
01776 // The absolute value of pitch is the number of bytes of an image row of the given
image buffer.
01777 // If pitch is negative, then the image buffer must point to the last row of a bottom-up
image (first byte on last row).
01778 // if pitch is positive, then the image buffer must point to the first row of a top-down
image (first byte).
01779 // The sequence of output channels in the output image buffer does not need to be
the same as provided by PGF. In case of different sequences you have to
01780 // provide a channelMap of size of expected channels (depending on image mode). For
example, PGF provides a channel sequence BGR in RGB color mode.
01781 // If your provided image buffer expects a channel sequence ARGB, then the channelMap
looks like { 3, 2, 1 }.
01782 // It might throw an IOException.
01783 // @param pitch The number of bytes of a row of the image buffer.
01784 // @param buff An image buffer.
01785 // @param bpp The number of bits per pixel used in image buffer.
01786 // @param channelMap A integer array containing the mapping of PGF channel ordering
to expected channel ordering.
01787 // @param cb A pointer to a callback procedure. The procedure is called after each
copied buffer row. If cb returns true, then it stops proceeding.
01788 // @param data Data Pointer to C++ class container to host callback procedure.
01789 void CPGFImage::GetBitmap(int pitch, UINT8* buff, BYTE bpp, int channelMap[] /*=
nullptr */, CallbackPtr cb /*= nullptr*/, void *data /*=nullptr*/) const {
01790   ASSERT(buff);
01791   UINT32 w = m_width[0];  // width of decoded image
01792   UINT32 h = m_height[0]; // height of decoded image
01793   UINT32 yw = w;                  // y-channel width
01794   UINT32 uw = m_width[1]; // u-channel width
01795   UINT32 roiOffsetX = 0;
01796   UINT32 roiOffsetY = 0;
01797   UINT32 yOffset = 0;
01798   UINT32 uOffset = 0;
01799
```

```
01800 #ifdef __PGFROISUPPORT__
01801         const PGFRect& roi = GetAlignedROI(); // in pixels, roi is usually larger
than levelRoi
01802         ASSERT(w == roi.Width() && h == roi.Height());
01803         const PGFRect levelRoi = ComputeLevelROI();
01804         ASSERT(roi.left <= levelRoi.left && levelRoi.right <= roi.right);
01805         ASSERT(roi.top <= levelRoi.top && levelRoi.bottom <= roi.bottom);
01806
01807         if (ROIisSupported() && (levelRoi.Width() < w || levelRoi.Height() < h)) {
01808                 // ROI is used
01809                 w = levelRoi.Width();
01810                 h = levelRoi.Height();
01811                 roiOffsetX = levelRoi.left - roi.left;
01812                 roiOffsetY = levelRoi.top - roi.top;
01813                 yOffset = roiOffsetX + roiOffsetY*yw;
01814
01815                 if (m_downsample) {
01816                         const PGFRect& downsampledRoi = GetAlignedROI(1);
01817                         uOffset = levelRoi.left/2 - downsampledRoi.left +
(levelRoi.top/2 - downsampledRoi.top)*m_width[1];
01818                 } else {
01819                         uOffset = yOffset;
01820                 }
01821         }
01822 #endif
01823
01824         const double dP = 1.0/h;
01825         int defMap[] = { 0, 1, 2, 3, 4, 5, 6, 7 };
ASSERT(sizeof(defMap)/sizeof(defMap[0]) == MaxChannels);
01826         if (channelMap == nullptr) channelMap = defMap;
01827         DataT uAvg, vAvg;
01828         double percent = 0;
01829         UINT32 i, j;
01830
01831         switch(m_header.mode) {
01832         case ImageModeBitmap:
01833                 {
01834                         ASSERT(m_header.channels == 1);
01835                         ASSERT(m_header.bpp == 1);
01836                         ASSERT(bpp == 1);
01837
01838                         const UINT32 w2 = (w + 7)/8;
01839                         DataT* y = m_channel[0]; ASSERT(y);
01840
01841                         if (m_preHeader.version & Version7) {
01842                                 // new unpacked version has a little better
compression ratio
01843                                 // since version 7
01844                                 for (i = 0; i < h; i++) {
01845                                         UINT32 cnt = 0;
01846                                         for (j = 0; j < w2; j++) {
01847                                                 UINT8 byte = 0;
01848                                                 for (int k = 0; k < 8; k++) {
01849                                                         byte <<= 1;
01850                                                         UINT8 bit = 0;
01851                                                         if (cnt < w) {
01852                                                                 bit = y[yOffset +
cnt] & 1;
01853                                                         }
01854                                                         byte |= bit;
01855                                                         cnt++;
01856                                                 }
01857                                                 buff[j] = byte;
01858                                         }
01859                                         yOffset += yw;
01860                                         buff += pitch;
01861
01862                                         if (cb) {
01863                                                 percent += dP;
01864                                                 if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
01865                                         }
01866                                 }
01867                         } else {
01868                                 // old versions
01869                                 // packed pixels: 8 pixel in 1 byte of channel[0]
```

```
01870                                   if (!(m_preHeader.version & Version5)) yw = w2; //
not version 5 or 6
01871                                   yOffset = roiOffsetX/8 + roiOffsetY*yw; // 1 byte
in y contains 8 pixel values
01872                                   for (i = 0; i < h; i++) {
01873                                           for (j = 0; j < w2; j++) {
01874                                                   buff[j] = Clamp8(y[yOffset + j] +
YUVoffset8);
01875                                           }
01876                                           yOffset += yw;
01877                                           buff += pitch;
01878
01879                                           if (cb) {
01880                                                   percent += dP;
01881                                                   if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
01882                                           }
01883                                   }
01884                           }
01885                           break;
01886                   }
01887           case ImageModeIndexedColor:
01888           case ImageModeGrayScale:
01889           case ImageModeHSLColor:
01890           case ImageModeHSBColor:
01891                   {
01892                           ASSERT(m_header.channels >= 1);
01893                           ASSERT(m_header.bpp == m_header.channels*8);
01894                           ASSERT(bpp%8 == 0);
01895
01896                           UINT32 cnt, channels = bpp/8; ASSERT(channels >=
m_header.channels);
01897
01898                           for (i=0; i < h; i++) {
01899                                   UINT32 yPos = yOffset;
01900                                   cnt = 0;
01901                                   for (j=0; j < w; j++) {
01902                                           for (UINT32 c=0; c < m_header.channels;
c++) {
01903                                                   buff[cnt + channelMap[c]] =
Clamp8(m_channel[c][yPos] + YUVoffset8);
01904                                           }
01905                                           cnt += channels;
01906                                           yPos++;
01907                                   }
01908                                   yOffset += yw;
01909                                   buff += pitch;
01910
01911                                   if (cb) {
01912                                           percent += dP;
01913                                           if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
01914                                   }
01915                           }
01916                           break;
01917                   }
01918           case ImageModeGray16:
01919                   {
01920                           ASSERT(m_header.channels >= 1);
01921                           ASSERT(m_header.bpp == m_header.channels*16);
01922
01923                           const DataT yuvOffset16 = 1 << (UsedBitsPerChannel() - 1);
01924                           UINT32 cnt, channels;
01925
01926                           if (bpp%16 == 0) {
01927                                   const int shift = 16 - UsedBitsPerChannel();
ASSERT(shift >= 0);
01928                                   UINT16 *buff16 = (UINT16 *)buff;
01929                                   int pitch16 = pitch/2;
01930                                   channels = bpp/16; ASSERT(channels >=
m_header.channels);
01931
01932                                   for (i=0; i < h; i++) {
01933                                           UINT32 yPos = yOffset;
01934                                           cnt = 0;
01935                                           for (j=0; j < w; j++) {
```

```
01936                                                   for (UINT32 c=0; c <
m_header.channels; c++) {
01937                                                           buff16[cnt +
channelMap[c]] = Clamp16((m_channel[c][yPos] + yuvOffset16) << shift);
01938                                                   }
01939                                                   cnt += channels;
01940                                                   yPos++;
01941                                           }
01942                                           yOffset += yw;
01943                                           buff16 += pitch16;
01944
01945                                           if (cb) {
01946                                                   percent += dP;
01947                                                   if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
01948                                           }
01949                                   }
01950                           } else {
01951                                   ASSERT(bpp%8 == 0);
01952                                   const int shift = __max(0, UsedBitsPerChannel() -
8);
01953                                   channels = bpp/8; ASSERT(channels >=
m_header.channels);
01954
01955                                   for (i=0; i < h; i++) {
01956                                           UINT32 yPos = yOffset;
01957                                           cnt = 0;
01958                                           for (j=0; j < w; j++) {
01959                                                   for (UINT32 c=0; c <
m_header.channels; c++) {
01960                                                           buff[cnt + channelMap[c]]
= Clamp8((m_channel[c][yPos] + yuvOffset16) >> shift);
01961                                                   }
01962                                                   cnt += channels;
01963                                                   yPos++;
01964                                           }
01965                                           yOffset += yw;
01966                                           buff += pitch;
01967
01968                                           if (cb) {
01969                                                   percent += dP;
01970                                                   if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
01971                                           }
01972                                   }
01973                           }
01974                           break;
01975                   }
01976           case ImageModeRGBColor:
01977                   {
01978                           ASSERT(m_header.channels == 3);
01979                           ASSERT(m_header.bpp == m_header.channels*8);
01980                           ASSERT(bpp%8 == 0);
01981                           ASSERT(bpp >= m_header.bpp);
01982
01983                           DataT* y = m_channel[0]; ASSERT(y);
01984                           DataT* u = m_channel[1]; ASSERT(u);
01985                           DataT* v = m_channel[2]; ASSERT(v);
01986                           UINT8 *buffg = &buff[channelMap[1]],
01987                                         *buffr = &buff[channelMap[2]],
01988                                         *buffb = &buff[channelMap[0]];
01989                           UINT8 g;
01990                           UINT32 cnt, channels = bpp/8;
01991
01992                           if (m_downsample) {
01993                                   for (i=0; i < h; i++) {
01994                                           UINT32 uPos = uOffset;
01995                                           UINT32 yPos = yOffset;
01996                                           cnt = 0;
01997                                           for (j=0; j < w; j++) {
01998                                                   // u and v are downsampled
01999                                                   uAvg = u[uPos];
02000                                                   vAvg = v[uPos];
02001                                                   // Yuv
02002                                                   buffg[cnt] = g = Clamp8(y[yPos] +
YUVoffset8 - ((uAvg + vAvg ) >> 2)); // must be logical shift operator
02003                                                   buffr[cnt] = Clamp8(uAvg + g);
```

```
02004                                           buffb[cnt] = Clamp8(vAvg + g);
02005                                           cnt += channels;
02006                                           if (j & 1) uPos++;
02007                                           yPos++;
02008                                   }
02009                                   if (i & 1) uOffset += uw;
02010                                   yOffset += yw;
02011                                   buffb += pitch;
02012                                   buffg += pitch;
02013                                   buffr += pitch;
02014
02015                                   if (cb) {
02016                                           percent += dP;
02017                                           if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
02018                                   }
02019                           }
02020
02021                   } else {
02022                           for (i=0; i < h; i++) {
02023                                   cnt = 0;
02024                                   UINT32 yPos = yOffset;
02025                                   for (j = 0; j < w; j++) {
02026                                           uAvg = u[yPos];
02027                                           vAvg = v[yPos];
02028                                           // Yuv
02029                                           buffg[cnt] = g = Clamp8(y[yPos] +
YUVoffset8 - ((uAvg + vAvg ) >> 2)); // must be logical shift operator
02030                                           buffr[cnt] = Clamp8(uAvg + g);
02031                                           buffb[cnt] = Clamp8(vAvg + g);
02032                                           cnt += channels;
02033                                           yPos++;
02034                                   }
02035                                   yOffset += yw;
02036                                   buffb += pitch;
02037                                   buffg += pitch;
02038                                   buffr += pitch;
02039
02040                                   if (cb) {
02041                                           percent += dP;
02042                                           if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
02043                                   }
02044                           }
02045                   }
02046                   break;
02047           }
02048   case ImageModeRGB48:
02049           {
02050                   ASSERT(m_header.channels == 3);
02051                   ASSERT(m_header.bpp == 48);
02052
02053                   const DataT yuvOffset16 = 1 << (UsedBitsPerChannel() - 1);
02054
02055                   DataT* y = m_channel[0]; ASSERT(y);
02056                   DataT* u = m_channel[1]; ASSERT(u);
02057                   DataT* v = m_channel[2]; ASSERT(v);
02058                   UINT32 cnt, channels;
02059                   DataT g;
02060
02061                   if (bpp >= 48 && bpp%16 == 0) {
02062                           const int shift = 16 - UsedBitsPerChannel();
ASSERT(shift >= 0);
02063                           UINT16 *buff16 = (UINT16 *)buff;
02064                           int pitch16 = pitch/2;
02065                           channels = bpp/16; ASSERT(channels >=
m_header.channels);
02066
02067                           for (i=0; i < h; i++) {
02068                                   UINT32 uPos = uOffset;
02069                                   UINT32 yPos = yOffset;
02070                                   cnt = 0;
02071                                   for (j=0; j < w; j++) {
02072                                           uAvg = u[uPos];
02073                                           vAvg = v[uPos];
02074                                           // Yuv
```

```
02075                                                           g = y[yPos] + yuvOffset16 - ((uAvg
+ vAvg ) >> 2); // must be logical shift operator
02076                                                           buff16[cnt + channelMap[1]] =
Clamp16(g << shift);
02077                                                           buff16[cnt + channelMap[2]] =
Clamp16((uAvg + g) << shift);
02078                                                           buff16[cnt + channelMap[0]] =
Clamp16((vAvg + g) << shift);
02079                                                           cnt += channels;
02080                                                           if (!m_downsample || (j & 1))
uPos++;
02081                                                           yPos++;
02082                                                   }
02083                                                   if (!m_downsample || (i & 1)) uOffset += uw;
02084                                                   yOffset += yw;
02085                                                   buff16 += pitch16;
02086
02087                                                   if (cb) {
02088                                                           percent += dP;
02089                                                           if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
02090                                                   }
02091                                           }
02092                                   } else {
02093                                           ASSERT(bpp%8 == 0);
02094                                           const int shift = __max(0, UsedBitsPerChannel() -
8);
02095                                           channels = bpp/8; ASSERT(channels >=
m_header.channels);
02096
02097                                           for (i=0; i < h; i++) {
02098                                                   UINT32 uPos = uOffset;
02099                                                   UINT32 yPos = yOffset;
02100                                                   cnt = 0;
02101                                                   for (j=0; j < w; j++) {
02102                                                           uAvg = u[uPos];
02103                                                           vAvg = v[uPos];
02104                                                           // Yuv
02105                                                           g = y[yPos] + yuvOffset16 - ((uAvg
+ vAvg ) >> 2); // must be logical shift operator
02106                                                           buff[cnt + channelMap[1]] =
Clamp8(g >> shift);
02107                                                           buff[cnt + channelMap[2]] =
Clamp8((uAvg + g) >> shift);
02108                                                           buff[cnt + channelMap[0]] =
Clamp8((vAvg + g) >> shift);
02109                                                           cnt += channels;
02110                                                           if (!m_downsample || (j & 1))
uPos++;
02111                                                           yPos++;
02112                                                   }
02113                                                   if (!m_downsample || (i & 1)) uOffset += uw;
02114                                                   yOffset += yw;
02115                                                   buff += pitch;
02116
02117                                                   if (cb) {
02118                                                           percent += dP;
02119                                                           if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
02120                                                   }
02121                                           }
02122                                   }
02123                                   break;
02124                           }
02125           case ImageModeLabColor:
02126                           {
02127                                   ASSERT(m_header.channels == 3);
02128                                   ASSERT(m_header.bpp == m_header.channels*8);
02129                                   ASSERT(bpp%8 == 0);
02130
02131                                   DataT* l = m_channel[0]; ASSERT(l);
02132                                   DataT* a = m_channel[1]; ASSERT(a);
02133                                   DataT* b = m_channel[2]; ASSERT(b);
02134                                   UINT32 cnt, channels = bpp/8; ASSERT(channels >=
m_header.channels);
02135
02136                                   for (i=0; i < h; i++) {
```

```
02137                               UINT32 uPos = uOffset;
02138                               UINT32 yPos = yOffset;
02139                               cnt = 0;
02140                               for (j=0; j < w; j++) {
02141                                       uAvg = a[uPos];
02142                                       vAvg = b[uPos];
02143                                       buff[cnt + channelMap[0]] = Clamp8(l[yPos]
+ YUVoffset8);
02144                                       buff[cnt + channelMap[1]] = Clamp8(uAvg +
YUVoffset8);
02145                                       buff[cnt + channelMap[2]] = Clamp8(vAvg +
YUVoffset8);
02146                                       cnt += channels;
02147                                       if (!m_downsample || (j & 1)) uPos++;
02148                                       yPos++;
02149                               }
02150                               if (!m_downsample || (i & 1)) uOffset += uw;
02151                               yOffset += yw;
02152                               buff += pitch;
02153
02154                               if (cb) {
02155                                       percent += dP;
02156                                       if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
02157                               }
02158                       }
02159                       break;
02160               }
02161       case ImageModeLab48:
02162               {
02163                       ASSERT(m_header.channels == 3);
02164                       ASSERT(m_header.bpp == m_header.channels*16);
02165
02166                       const DataT yuvOffset16 = 1 << (UsedBitsPerChannel() - 1);
02167
02168                       DataT* l = m_channel[0]; ASSERT(l);
02169                       DataT* a = m_channel[1]; ASSERT(a);
02170                       DataT* b = m_channel[2]; ASSERT(b);
02171                       UINT32 cnt, channels;
02172
02173                       if (bpp%16 == 0) {
02174                               const int shift = 16 - UsedBitsPerChannel();
ASSERT(shift >= 0);
02175                               UINT16 *buff16 = (UINT16 *)buff;
02176                               int pitch16 = pitch/2;
02177                               channels = bpp/16; ASSERT(channels >=
m_header.channels);
02178
02179                               for (i=0; i < h; i++) {
02180                                       UINT32 uPos = uOffset;
02181                                       UINT32 yPos = yOffset;
02182                                       cnt = 0;
02183                                       for (j=0; j < w; j++) {
02184                                               uAvg = a[uPos];
02185                                               vAvg = b[uPos];
02186                                               buff16[cnt + channelMap[0]] =
Clamp16((l[yPos] + yuvOffset16) << shift);
02187                                               buff16[cnt + channelMap[1]] =
Clamp16((uAvg + yuvOffset16) << shift);
02188                                               buff16[cnt + channelMap[2]] =
Clamp16((vAvg + yuvOffset16) << shift);
02189                                               cnt += channels;
02190                                               if (!m_downsample || (j & 1))
uPos++;
02191                                               yPos++;
02192                                       }
02193                                       if (!m_downsample || (i & 1)) uOffset += uw;
02194                                       yOffset += yw;
02195                                       buff16 += pitch16;
02196
02197                                       if (cb) {
02198                                               percent += dP;
02199                                               if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
02200                                       }
02201                               }
02202                       } else {
```

```
02203                                   ASSERT(bpp%8 == 0);
02204                                   const int shift = __max(0, UsedBitsPerChannel() -
8);
02205                                   channels = bpp/8; ASSERT(channels >=
m_header.channels);
02206
02207                                   for (i=0; i < h; i++) {
02208                                           UINT32 uPos = uOffset;
02209                                           UINT32 yPos = yOffset;
02210                                           cnt = 0;
02211                                           for (j=0; j < w; j++) {
02212                                                   uAvg = a[uPos];
02213                                                   vAvg = b[uPos];
02214                                                   buff[cnt + channelMap[0]] =
Clamp8((l[yPos] + yuvOffset16) >> shift);
02215                                                   buff[cnt + channelMap[1]] =
Clamp8((uAvg + yuvOffset16) >> shift);
02216                                                   buff[cnt + channelMap[2]] =
Clamp8((vAvg + yuvOffset16) >> shift);
02217                                                   cnt += channels;
02218                                                   if (!m_downsample || (j & 1))
uPos++;
02219                                                   yPos++;
02220                                           }
02221                                           if (!m_downsample || (i & 1)) uOffset += uw;
02222                                           yOffset += yw;
02223                                           buff += pitch;
02224
02225                                           if (cb) {
02226                                                   percent += dP;
02227                                                   if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
02228                                           }
02229                                   }
02230                           }
02231                           break;
02232                   }
02233           case ImageModeRGBA:
02234           case ImageModeCMYKColor:
02235                   {
02236                           ASSERT(m_header.channels == 4);
02237                           ASSERT(m_header.bpp == m_header.channels*8);
02238                           ASSERT(bpp%8 == 0);
02239
02240                           DataT* y = m_channel[0]; ASSERT(y);
02241                           DataT* u = m_channel[1]; ASSERT(u);
02242                           DataT* v = m_channel[2]; ASSERT(v);
02243                           DataT* a = m_channel[3]; ASSERT(a);
02244                           UINT8 g, aAvg;
02245                           UINT32 cnt, channels = bpp/8; ASSERT(channels >=
m_header.channels);
02246
02247                           for (i=0; i < h; i++) {
02248                                   UINT32 uPos = uOffset;
02249                                   UINT32 yPos = yOffset;
02250                                   cnt = 0;
02251                                   for (j=0; j < w; j++) {
02252                                           uAvg = u[uPos];
02253                                           vAvg = v[uPos];
02254                                           aAvg = Clamp8(a[uPos] + YUVoffset8);
02255                                           // Yuv
02256                                           buff[cnt + channelMap[1]] = g =
Clamp8(y[yPos] + YUVoffset8 - ((uAvg + vAvg ) >> 2)); // must be logical shift operator
02257                                           buff[cnt + channelMap[2]] = Clamp8(uAvg +
g);
02258                                           buff[cnt + channelMap[0]] = Clamp8(vAvg +
g);
02259                                           buff[cnt + channelMap[3]] = aAvg;
02260                                           cnt += channels;
02261                                           if (!m_downsample || (j & 1)) uPos++;
02262                                           yPos++;
02263                                   }
02264                                   if (!m_downsample || (i & 1)) uOffset += uw;
02265                                   yOffset += yw;
02266                                   buff += pitch;
02267
02268                                   if (cb) {
```

```
02269                                           percent += dP;
02270                                           if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
02271                                   }
02272                           }
02273                           break;
02274                   }
02275           case ImageModeCMYK64:
02276                   {
02277                           ASSERT(m_header.channels == 4);
02278                           ASSERT(m_header.bpp == 64);
02279
02280                           const DataT yuvOffset16 = 1 << (UsedBitsPerChannel() - 1);
02281
02282                           DataT* y = m_channel[0]; ASSERT(y);
02283                           DataT* u = m_channel[1]; ASSERT(u);
02284                           DataT* v = m_channel[2]; ASSERT(v);
02285                           DataT* a = m_channel[3]; ASSERT(a);
02286                           DataT g, aAvg;
02287                           UINT32 cnt, channels;
02288
02289                           if (bpp%16 == 0) {
02290                                   const int shift = 16 - UsedBitsPerChannel();
ASSERT(shift >= 0);
02291                                   UINT16 *buff16 = (UINT16 *)buff;
02292                                   int pitch16 = pitch/2;
02293                                   channels = bpp/16; ASSERT(channels >=
m_header.channels);
02294
02295                                   for (i=0; i < h; i++) {
02296                                           UINT32 uPos = uOffset;
02297                                           UINT32 yPos = yOffset;
02298                                           cnt = 0;
02299                                           for (j=0; j < w; j++) {
02300                                                   uAvg = u[uPos];
02301                                                   vAvg = v[uPos];
02302                                                   aAvg = a[uPos] + yuvOffset16;
02303                                                   // Yuv
02304                                                   g = y[yPos] + yuvOffset16 - ((uAvg
+ vAvg ) >> 2); // must be logical shift operator
02305                                                   buff16[cnt + channelMap[1]] =
Clamp16(g << shift);
02306                                                   buff16[cnt + channelMap[2]] =
Clamp16((uAvg + g) << shift);
02307                                                   buff16[cnt + channelMap[0]] =
Clamp16((vAvg + g) << shift);
02308                                                   buff16[cnt + channelMap[3]] =
Clamp16(aAvg << shift);
02309                                                   cnt += channels;
02310                                                   if (!m_downsample || (j & 1))
uPos++;
02311                                                   yPos++;
02312                                           }
02313                                           if (!m_downsample || (i & 1)) uOffset += uw;
02314                                           yOffset += yw;
02315                                           buff16 += pitch16;
02316
02317                                           if (cb) {
02318                                                   percent += dP;
02319                                                   if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
02320                                           }
02321                                   }
02322                           } else {
02323                                   ASSERT(bpp%8 == 0);
02324                                   const int shift = __max(0, UsedBitsPerChannel() -
8);
02325                                   channels = bpp/8; ASSERT(channels >=
m_header.channels);
02326
02327                                   for (i=0; i < h; i++) {
02328                                           UINT32 uPos = uOffset;
02329                                           UINT32 yPos = yOffset;
02330                                           cnt = 0;
02331                                           for (j=0; j < w; j++) {
02332                                                   uAvg = u[uPos];
02333                                                   vAvg = v[uPos];
```

```
02334                                                           aAvg = a[uPos] + yuvOffset16;
02335                                                           // Yuv
02336                                                           g = y[yPos] + yuvOffset16 - ((uAvg
+ vAvg ) >> 2); // must be logical shift operator
02337                                                           buff[cnt + channelMap[1]] =
Clamp8(g >> shift);
02338                                                           buff[cnt + channelMap[2]] =
Clamp8((uAvg + g) >> shift);
02339                                                           buff[cnt + channelMap[0]] =
Clamp8((vAvg + g) >> shift);
02340                                                           buff[cnt + channelMap[3]] =
Clamp8(aAvg >> shift);
02341                                                           cnt += channels;
02342                                                           if (!m_downsample || (j & 1))
uPos++;
02343                                                           yPos++;
02344                                                   }
02345                                                   if (!m_downsample || (i & 1)) uOffset += uw;
02346                                                   yOffset += yw;
02347                                                   buff += pitch;
02348
02349                                                   if (cb) {
02350                                                           percent += dP;
02351                                                           if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
02352                                                   }
02353                                           }
02354                                   }
02355                                   break;
02356                           }
02357 #ifdef __PGF32SUPPORT__
02358           case ImageModeGray32:
02359                           {
02360                                   ASSERT(m_header.channels == 1);
02361                                   ASSERT(m_header.bpp == 32);
02362
02363                                   const int yuvOffset31 = 1 << (UsedBitsPerChannel() - 1);
02364                                   DataT* y = m_channel[0]; ASSERT(y);
02365
02366                                   if (bpp == 32) {
02367                                           const int shift = 31 - UsedBitsPerChannel();
ASSERT(shift >= 0);
02368                                           UINT32 *buff32 = (UINT32 *)buff;
02369                                           int pitch32 = pitch/4;
02370
02371                                           for (i=0; i < h; i++) {
02372                                                   UINT32 yPos = yOffset;
02373                                                   for (j = 0; j < w; j++) {
02374                                                           buff32[j] = Clamp31((y[yPos++] +
yuvOffset31) << shift);
02375                                                   }
02376                                                   yOffset += yw;
02377                                                   buff32 += pitch32;
02378
02379                                                   if (cb) {
02380                                                           percent += dP;
02381                                                           if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
02382                                                   }
02383                                           }
02384                                   } else if (bpp == 16) {
02385                                           const int usedBits = UsedBitsPerChannel();
02386                                           UINT16 *buff16 = (UINT16 *)buff;
02387                                           int pitch16 = pitch/2;
02388
02389                                           if (usedBits < 16) {
02390                                                   const int shift = 16 - usedBits;
02391                                                   for (i=0; i < h; i++) {
02392                                                           UINT32 yPos = yOffset;
02393                                                           for (j = 0; j < w; j++) {
02394                                                                   buff16[j] =
Clamp16((y[yPos++] + yuvOffset31) << shift);
02395                                                           }
02396                                                           yOffset += yw;
02397                                                           buff16 += pitch16;
02398
02399                                                           if (cb) {
```

```
02400                                                   percent += dP;
02401                                                   if ((*cb)(percent, true,
data)) ReturnWithError(EscapePressed);
02402                                           }
02403                                   }
02404                           } else {
02405                                   const int shift = __max(0, usedBits - 16);
02406                                   for (i=0; i < h; i++) {
02407                                           UINT32 yPos = yOffset;
02408                                           for (j = 0; j < w; j++) {
02409                                                   buff16[j] =
Clamp16((y[yPos++] + yuvOffset31) >> shift);
02410                                           }
02411                                           yOffset += yw;
02412                                           buff16 += pitch16;
02413
02414                                           if (cb) {
02415                                                   percent += dP;
02416                                                   if ((*cb)(percent, true,
data)) ReturnWithError(EscapePressed);
02417                                           }
02418                                   }
02419                           }
02420                   } else {
02421                           ASSERT(bpp == 8);
02422                           const int shift = __max(0, UsedBitsPerChannel() -
8);
02423
02424                           for (i=0; i < h; i++) {
02425                                   UINT32 yPos = yOffset;
02426                                   for (j = 0; j < w; j++) {
02427                                           buff[j] = Clamp8((y[yPos++] +
yuvOffset31) >> shift);
02428                                   }
02429                                   yOffset += yw;
02430                                   buff += pitch;
02431
02432                                   if (cb) {
02433                                           percent += dP;
02434                                           if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
02435                                   }
02436                           }
02437                   }
02438                   break;
02439           }
02440 #endif
02441   case ImageModeRGB12:
02442           {
02443                   ASSERT(m_header.channels == 3);
02444                   ASSERT(m_header.bpp == m_header.channels*4);
02445                   ASSERT(bpp == m_header.channels*4);
02446                   ASSERT(!m_downsample);
02447
02448                   DataT* y = m_channel[0]; ASSERT(y);
02449                   DataT* u = m_channel[1]; ASSERT(u);
02450                   DataT* v = m_channel[2]; ASSERT(v);
02451                   UINT16 yval;
02452                   UINT32 cnt;
02453
02454                   for (i=0; i < h; i++) {
02455                           UINT32 yPos = yOffset;
02456                           cnt = 0;
02457                           for (j=0; j < w; j++) {
02458                                   // Yuv
02459                                   uAvg = u[yPos];
02460                                   vAvg = v[yPos];
02461                                   yval = Clamp4(y[yPos] + YUVoffset4 - ((uAvg
+ vAvg ) >> 2)); // must be logical shift operator
02462                                   if (j%2 == 0) {
02463                                           buff[cnt] = UINT8(Clamp4(vAvg +
yval) | (yval << 4));
02464                                           cnt++;
02465                                           buff[cnt] = Clamp4(uAvg + yval);
02466                                   } else {
02467                                           buff[cnt] |= Clamp4(vAvg + yval)
<< 4;
```

```
02468                                           cnt++;
02469                                           buff[cnt] = UINT8(yval |
(Clamp4(uAvg + yval) << 4));
02470                                           cnt++;
02471                                   }
02472                                   yPos++;
02473                           }
02474                           yOffset += yw;
02475                           buff += pitch;
02476
02477                           if (cb) {
02478                                   percent += dP;
02479                                   if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
02480                           }
02481                   }
02482                   break;
02483           }
02484   case ImageModeRGB16:
02485           {
02486                   ASSERT(m_header.channels == 3);
02487                   ASSERT(m_header.bpp == 16);
02488                   ASSERT(bpp == 16);
02489                   ASSERT(!m_downsample);
02490
02491                   DataT* y = m_channel[0]; ASSERT(y);
02492                   DataT* u = m_channel[1]; ASSERT(u);
02493                   DataT* v = m_channel[2]; ASSERT(v);
02494                   UINT16 yval;
02495                   UINT16 *buff16 = (UINT16 *)buff;
02496                   int pitch16 = pitch/2;
02497
02498                   for (i=0; i < h; i++) {
02499                           UINT32 yPos = yOffset;
02500                           for (j = 0; j < w; j++) {
02501                                   // Yuv
02502                                   uAvg = u[yPos];
02503                                   vAvg = v[yPos];
02504                                   yval = Clamp6(y[yPos++] + YUVoffset6 -
((uAvg + vAvg ) >> 2)); // must be logical shift operator
02505                                   buff16[j] = (yval << 5) | ((Clamp6(uAvg +
yval) >> 1) << 11) | (Clamp6(vAvg + yval) >> 1);
02506                           }
02507                           yOffset += yw;
02508                           buff16 += pitch16;
02509
02510                           if (cb) {
02511                                   percent += dP;
02512                                   if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
02513                           }
02514                   }
02515                   break;
02516           }
02517   default:
02518           ASSERT(false);
02519   }
02520
02521 #ifdef _DEBUG
02522   // display ROI (RGB) in debugger
02523   roiimage.width = w;
02524   roiimage.height = h;
02525   if (pitch > 0) {
02526           roiimage.pitch = pitch;
02527           roiimage.data = buff;
02528   } else {
02529           roiimage.pitch = -pitch;
02530           roiimage.data = buff + (h - 1)*pitch;
02531   }
02532 #endif
02533
02534 }
02535
02550 void CPGFImage::GetYUV(int pitch, DataT* buff, BYTE bpp, int channelMap[] /*=
nullptr*/, CallbackPtr cb /*= nullptr*/, void *data /*=nullptr*/) const {
02551   ASSERT(buff);
02552   const UINT32 w = m_width[0];
```

```
02553         const UINT32 h = m_height[0];
02554         const bool wOdd = (1 == w%2);
02555         const int dataBits = DataTSize*8; ASSERT(dataBits == 16 || dataBits == 32);
02556         const int pitch2 = pitch/DataTSize;
02557         const int yuvOffset = (dataBits == 16) ? YUVoffset8 : YUVoffset16;
02558         const double dP = 1.0/h;
02559
02560         int defMap[] = { 0, 1, 2, 3, 4, 5, 6, 7 };
ASSERT(sizeof(defMap)/sizeof(defMap[0]) == MaxChannels);
02561         if (channelMap == nullptr) channelMap = defMap;
02562         int sampledPos = 0, yPos = 0;
02563         DataT uAvg, vAvg;
02564         double percent = 0;
02565         UINT32 i, j;
02566
02567         if (m_header.channels == 3) {
02568                 ASSERT(bpp%dataBits == 0);
02569
02570                 DataT* y = m_channel[0]; ASSERT(y);
02571                 DataT* u = m_channel[1]; ASSERT(u);
02572                 DataT* v = m_channel[2]; ASSERT(v);
02573                 int cnt, channels = bpp/dataBits; ASSERT(channels >=
m_header.channels);
02574
02575                 for (i=0; i < h; i++) {
02576                         if (i%2) sampledPos -= (w + 1)/2;
02577                         cnt = 0;
02578                         for (j=0; j < w; j++) {
02579                                 if (m_downsample) {
02580                                         // image was downsampled
02581                                         uAvg = u[sampledPos];
02582                                         vAvg = v[sampledPos];
02583                                 } else {
02584                                         uAvg = u[yPos];
02585                                         vAvg = v[yPos];
02586                                 }
02587                                 buff[cnt + channelMap[0]] = y[yPos];
02588                                 buff[cnt + channelMap[1]] = uAvg;
02589                                 buff[cnt + channelMap[2]] = vAvg;
02590                                 yPos++;
02591                                 cnt += channels;
02592                                 if (j%2) sampledPos++;
02593                         }
02594                         buff += pitch2;
02595                         if (wOdd) sampledPos++;
02596
02597                         if (cb) {
02598                                 percent += dP;
02599                                 if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
02600                         }
02601                 }
02602         } else if (m_header.channels == 4) {
02603                 ASSERT(m_header.bpp == m_header.channels*8);
02604                 ASSERT(bpp%dataBits == 0);
02605
02606                 DataT* y = m_channel[0]; ASSERT(y);
02607                 DataT* u = m_channel[1]; ASSERT(u);
02608                 DataT* v = m_channel[2]; ASSERT(v);
02609                 DataT* a = m_channel[3]; ASSERT(a);
02610                 UINT8 aAvg;
02611                 int cnt, channels = bpp/dataBits; ASSERT(channels >=
m_header.channels);
02612
02613                 for (i=0; i < h; i++) {
02614                         if (i%2) sampledPos -= (w + 1)/2;
02615                         cnt = 0;
02616                         for (j=0; j < w; j++) {
02617                                 if (m_downsample) {
02618                                         // image was downsampled
02619                                         uAvg = u[sampledPos];
02620                                         vAvg = v[sampledPos];
02621                                         aAvg = Clamp8(a[sampledPos] + yuvOffset);
02622                                 } else {
02623                                         uAvg = u[yPos];
02624                                         vAvg = v[yPos];
02625                                         aAvg = Clamp8(a[yPos] + yuvOffset);
```

```
02626                               }
02627                               // Yuv
02628                               buff[cnt + channelMap[0]] = y[yPos];
02629                               buff[cnt + channelMap[1]] = uAvg;
02630                               buff[cnt + channelMap[2]] = vAvg;
02631                               buff[cnt + channelMap[3]] = aAvg;
02632                               yPos++;
02633                               cnt += channels;
02634                               if (j%2) sampledPos++;
02635                        }
02636                        buff += pitch2;
02637                        if (wOdd) sampledPos++;
02638
02639                        if (cb) {
02640                               percent += dP;
02641                               if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
02642                        }
02643                 }
02644          }
02645 }
02646
02661 void CPGFImage::ImportYUV(int pitch, DataT *buff, BYTE bpp, int channelMap[] /*=
nullptr*/, CallbackPtr cb /*= nullptr*/, void *data /*=nullptr*/) {
02662          ASSERT(buff);
02663          const double dP = 1.0/m_header.height;
02664          const int dataBits = DataTSize*8; ASSERT(dataBits == 16 || dataBits == 32);
02665          const int pitch2 = pitch/DataTSize;
02666          const int yuvOffset = (dataBits == 16) ? YUVoffset8 : YUVoffset16;
02667
02668          int yPos = 0, cnt = 0;
02669          double percent = 0;
02670          int defMap[] = { 0, 1, 2, 3, 4, 5, 6, 7 };
ASSERT(sizeof(defMap)/sizeof(defMap[0]) == MaxChannels);
02671
02672          if (channelMap == nullptr) channelMap = defMap;
02673
02674          if (m_header.channels == 3)     {
02675                 ASSERT(bpp%dataBits == 0);
02676
02677                 DataT* y = m_channel[0]; ASSERT(y);
02678                 DataT* u = m_channel[1]; ASSERT(u);
02679                 DataT* v = m_channel[2]; ASSERT(v);
02680                 const int channels = bpp/dataBits; ASSERT(channels >=
m_header.channels);
02681
02682                 for (UINT32 h=0; h < m_header.height; h++) {
02683                        if (cb) {
02684                               if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
02685                               percent += dP;
02686                        }
02687
02688                        cnt = 0;
02689                        for (UINT32 w=0; w < m_header.width; w++) {
02690                               y[yPos] = buff[cnt + channelMap[0]];
02691                               u[yPos] = buff[cnt + channelMap[1]];
02692                               v[yPos] = buff[cnt + channelMap[2]];
02693                               yPos++;
02694                               cnt += channels;
02695                        }
02696                        buff += pitch2;
02697                 }
02698          } else if (m_header.channels == 4) {
02699                 ASSERT(bpp%dataBits == 0);
02700
02701                 DataT* y = m_channel[0]; ASSERT(y);
02702                 DataT* u = m_channel[1]; ASSERT(u);
02703                 DataT* v = m_channel[2]; ASSERT(v);
02704                 DataT* a = m_channel[3]; ASSERT(a);
02705                 const int channels = bpp/dataBits; ASSERT(channels >=
m_header.channels);
02706
02707                 for (UINT32 h=0; h < m_header.height; h++) {
02708                        if (cb) {
02709                               if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
```

```
02710                                     percent += dP;
02711                             }
02712 
02713                             cnt = 0;
02714                             for (UINT32 w=0; w < m_header.width; w++) {
02715                                     y[yPos] = buff[cnt + channelMap[0]];
02716                                     u[yPos] = buff[cnt + channelMap[1]];
02717                                     v[yPos] = buff[cnt + channelMap[2]];
02718                                     a[yPos] = buff[cnt + channelMap[3]] - yuvOffset;
02719                                     yPos++;
02720                                     cnt += channels;
02721                             }
02722                             buff += pitch2;
02723                     }
02724             }
02725 
02726             if (m_downsample) {
02727                     // Subsampling of the chrominance and alpha channels
02728                     for (int i=1; i < m_header.channels; i++) {
02729                             Downsample(i);
02730                     }
02731             }
02732 }
02733 
```

PGFstream.cpp File Reference

PGF stream class implementation.
`#include "PGFstream.h"`

Detailed Description

PGF stream class implementation.

Author

C. Stamm

Definition in file **PGFstream.cpp**.

PGFstream.cpp

```
Go to the documentation of this file.00001 /*
00002  * The Progressive Graphics File; http://www.libpgf.org
00003  *
00004  * $Date: 2007-01-19 11:51:24 +0100 (Fr, 19 Jan 2007) $
00005  * $Revision: 268 $
00006  *
00007  * This file Copyright (C) 2006 xeraina GmbH, Switzerland
00008  *
00009  * This program is free software; you can redistribute it and/or
00010  * modify it under the terms of the GNU LESSER GENERAL PUBLIC LICENSE
00011  * as published by the Free Software Foundation; either version 2.1
00012  * of the License, or (at your option) any later version.
00013  *
00014  * This program is distributed in the hope that it will be useful,
00015  * but WITHOUT ANY WARRANTY; without even the implied warranty of
00016  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00017  * GNU General Public License for more details.
00018  *
00019  * You should have received a copy of the GNU General Public License
00020  * along with this program; if not, write to the Free Software
00021  * Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
00022  */
00023
00028
00029 #include "PGFstream.h"
00030
00031 #ifdef WIN32
00032 #include <malloc.h>
00033 #endif
00034
00036 // CPGFFileStream
00038 void CPGFFileStream::Write(int *count, void *buffPtr) {
00039         ASSERT(count);
00040         ASSERT(buffPtr);
00041         ASSERT(IsValid());
00042         OSError err;
00043         if ((err = FileWrite(m_hFile, count, buffPtr)) != NoError)
ReturnWithError(err);
00044
00045 }
00046
00048 void CPGFFileStream::Read(int *count, void *buffPtr) {
00049         ASSERT(count);
00050         ASSERT(buffPtr);
00051         ASSERT(IsValid());
00052         OSError err;
00053         if ((err = FileRead(m_hFile, count, buffPtr)) != NoError)
ReturnWithError(err);
00054 }
00055
00057 void CPGFFileStream::SetPos(short posMode, INT64 posOff) {
00058         ASSERT(IsValid());
00059         OSError err;
00060         if ((err = SetFPos(m_hFile, posMode, posOff)) != NoError)
ReturnWithError(err);
00061 }
00062
00064 UINT64 CPGFFileStream::GetPos() const {
00065         ASSERT(IsValid());
00066         OSError err;
00067         UINT64 pos = 0;
00068         if ((err = GetFPos(m_hFile, &pos)) != NoError) ReturnWithError2(err, pos);
00069         return pos;
00070 }
00071
00072
00074 // CPGFMemoryStream
00078 CPGFMemoryStream::CPGFMemoryStream(size_t size)
00079 : m_size(size)
00080 , m_allocated(true) {
00081         m_buffer = m_pos = m_eos = new(std::nothrow) UINT8[m_size];
00082         if (!m_buffer) ReturnWithError(InsufficientMemory);
00083 }
```

```
00084
00089 CPGFMemoryStream::CPGFMemoryStream(UINT8 *pBuffer, size_t size)
00090 : m_buffer(pBuffer)
00091 , m_pos(pBuffer)
00092 , m_eos(pBuffer + size)
00093 , m_size(size)
00094 , m_allocated(false) {
00095         ASSERT(IsValid());
00096 }
00097
00102 void CPGFMemoryStream::Reinitialize(UINT8 *pBuffer, size_t size) {
00103         if (!m_allocated) {
00104                 m_buffer = m_pos = pBuffer;
00105                 m_size = size;
00106                 m_eos = m_buffer + size;
00107         }
00108 }
00109
00111 void CPGFMemoryStream::Write(int *count, void *buffPtr) {
00112         ASSERT(count);
00113         ASSERT(buffPtr);
00114         ASSERT(IsValid());
00115         const size_t deltaSize = 0x4000 + *count;
00116
00117         if (m_pos + *count <= m_buffer + m_size) {
00118                 memcpy(m_pos, buffPtr, *count);
00119                 m_pos += *count;
00120                 if (m_pos > m_eos) m_eos = m_pos;
00121         } else if (m_allocated) {
00122                 // memory block is too small -> reallocate a deltaSize larger block
00123                 size_t offset = m_pos - m_buffer;
00124                 UINT8 *buf_tmp = (UINT8 *)realloc(m_buffer, m_size + deltaSize);
00125                 if (!buf_tmp) {
00126                         delete[] m_buffer;
00127                         m_buffer = 0;
00128                         ReturnWithError(InsufficientMemory);
00129                 } else {
00130                         m_buffer = buf_tmp;
00131                 }
00132                 m_size += deltaSize;
00133
00134                 // reposition m_pos
00135                 m_pos = m_buffer + offset;
00136
00137                 // write block
00138                 memcpy(m_pos, buffPtr, *count);
00139                 m_pos += *count;
00140                 if (m_pos > m_eos) m_eos = m_pos;
00141         } else {
00142                 ReturnWithError(InsufficientMemory);
00143         }
00144         ASSERT(m_pos <= m_eos);
00145 }
00146
00148 void CPGFMemoryStream::Read(int *count, void *buffPtr) {
00149         ASSERT(IsValid());
00150         ASSERT(count);
00151         ASSERT(buffPtr);
00152         ASSERT(m_buffer + m_size >= m_eos);
00153         ASSERT(m_pos <= m_eos);
00154
00155         if (m_pos + *count <= m_eos) {
00156                 memcpy(buffPtr, m_pos, *count);
00157                 m_pos += *count;
00158         } else {
00159                 // end of memory block reached -> read only until end
00160                 *count = (int)__max(0, m_eos - m_pos);
00161                 memcpy(buffPtr, m_pos, *count);
00162                 m_pos += *count;
00163         }
00164         ASSERT(m_pos <= m_eos);
00165 }
00166
00168 void CPGFMemoryStream::SetPos(short posMode, INT64 posOff) {
00169         ASSERT(IsValid());
00170         switch(posMode) {
00171         case FSFromStart:
```

```
00172                   m_pos = m_buffer + posOff;
00173                   break;
00174           case FSFromCurrent:
00175                   m_pos += posOff;
00176                   break;
00177           case FSFromEnd:
00178                   m_pos = m_eos + posOff;
00179                   break;
00180           default:
00181                   ASSERT(false);
00182           }
00183           if (m_pos > m_eos)
00184                   ReturnWithError(InvalidStreamPos);
00185 }
00186
00187
00189 // CPGFMemFileStream
00190 #ifdef _MFC_VER
00192 void CPGFMemFileStream::Write(int *count, void *buffPtr) {
00193           ASSERT(count);
00194           ASSERT(buffPtr);
00195           ASSERT(IsValid());
00196           m_memFile->Write(buffPtr, *count);
00197 }
00198
00200 void CPGFMemFileStream::Read(int *count, void *buffPtr) {
00201           ASSERT(count);
00202           ASSERT(buffPtr);
00203           ASSERT(IsValid());
00204           m_memFile->Read(buffPtr, *count);
00205 }
00206
00208 void CPGFMemFileStream::SetPos(short posMode, INT64 posOff) {
00209           ASSERT(IsValid());
00210           m_memFile->Seek(posOff, posMode);
00211 }
00212
00214 UINT64 CPGFMemFileStream::GetPos() const {
00215           return (UINT64)m_memFile->GetPosition();
00216 }
00217 #endif // _MFC_VER
00218
00220 // CPGFIStream
00221 #if defined(WIN32) || defined(WINCE)
00223 void CPGFIStream::Write(int *count, void *buffPtr) {
00224           ASSERT(count);
00225           ASSERT(buffPtr);
00226           ASSERT(IsValid());
00227
00228           HRESULT hr = m_stream->Write(buffPtr, *count, (ULONG *)count);
00229           if (FAILED(hr)) {
00230                   ReturnWithError(hr);
00231           }
00232 }
00233
00235 void CPGFIStream::Read(int *count, void *buffPtr) {
00236           ASSERT(count);
00237           ASSERT(buffPtr);
00238           ASSERT(IsValid());
00239
00240           HRESULT hr = m_stream->Read(buffPtr, *count, (ULONG *)count);
00241           if (FAILED(hr)) {
00242                   ReturnWithError(hr);
00243           }
00244 }
00245
00247 void CPGFIStream::SetPos(short posMode, INT64 posOff) {
00248           ASSERT(IsValid());
00249
00250           LARGE_INTEGER li;
00251           li.QuadPart = posOff;
00252
00253           HRESULT hr = m_stream->Seek(li, posMode, nullptr);
00254           if (FAILED(hr)) {
00255                   ReturnWithError(hr);
00256           }
00257 }
```

```
00258
00260 UINT64 CPGFIStream::GetPos() const {
00261         ASSERT(IsValid());
00262
00263         LARGE_INTEGER n;
00264         ULARGE_INTEGER pos;
00265         n.QuadPart = 0;
00266
00267         HRESULT hr = m_stream->Seek(n, FSFromCurrent, &pos);
00268         if (SUCCEEDED(hr)) {
00269                 return pos.QuadPart;
00270         } else {
00271                 ReturnWithError2(hr, pos.QuadPart);
00272         }
00273 }
00274 #endif // WIN32 || WINCE
```

Subband.cpp File Reference

PGF wavelet subband class implementation.
```
#include "Subband.h"
#include "Encoder.h"
#include "Decoder.h"
```

Detailed Description

PGF wavelet subband class implementation.

Author

C. Stamm

Definition in file **Subband.cpp**.

Subband.cpp

Go to the documentation of this file.
```
00001 /*
00002  * The Progressive Graphics File; http://www.libpgf.org
00003  *
00004  * $Date: 2006-06-04 22:05:59 +0200 (So, 04 Jun 2006) $
00005  * $Revision: 229 $
00006  *
00007  * This file Copyright (C) 2006 xeraina GmbH, Switzerland
00008  *
00009  * This program is free software; you can redistribute it and/or
00010  * modify it under the terms of the GNU LESSER GENERAL PUBLIC LICENSE
00011  * as published by the Free Software Foundation; either version 2.1
00012  * of the License, or (at your option) any later version.
00013  *
00014  * This program is distributed in the hope that it will be useful,
00015  * but WITHOUT ANY WARRANTY; without even the implied warranty of
00016  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00017  * GNU General Public License for more details.
00018  *
00019  * You should have received a copy of the GNU General Public License
00020  * along with this program; if not, write to the Free Software
00021  * Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
00022  */
00023
00028
00029 #include "Subband.h"
00030 #include "Encoder.h"
00031 #include "Decoder.h"
00032
00034 // Default constructor
00035 CSubband::CSubband()
00036 : m_width(0)
00037 , m_height(0)
00038 , m_size(0)
00039 , m_level(0)
00040 , m_orientation(LL)
00041 , m_data(0)
00042 , m_dataPos(0)
00043 #ifdef __PGFROISUPPORT__
00044 , m_nTiles(0)
00045 #endif
00046 {
00047 }
00048
00050 // Destructor
00051 CSubband::~CSubband() {
00052         FreeMemory();
00053 }
00054
00056 // Initialize subband parameters
00057 void CSubband::Initialize(UINT32 width, UINT32 height, int level, Orientation
orient) {
00058         m_width = width;
00059         m_height = height;
00060         m_size = m_width*m_height;
00061         m_level = level;
00062         m_orientation = orient;
00063         m_data = 0;
00064         m_dataPos = 0;
00065 #ifdef __PGFROISUPPORT__
00066         m_ROI.left = 0;
00067         m_ROI.top = 0;
00068         m_ROI.right = m_width;
00069         m_ROI.bottom = m_height;
00070         m_nTiles = 0;
00071 #endif
00072 }
00073
00075 // Allocate a memory buffer to store all wavelet coefficients of this subband.
00076 // @return True if the allocation works without any problems
00077 bool CSubband::AllocMemory() {
00078         UINT32 oldSize = m_size;
00079
00080 #ifdef __PGFROISUPPORT__
```

```
00081         m_size = BufferWidth()*m_ROI.Height();
00082 #endif
00083         ASSERT(m_size > 0);
00084
00085         if (m_data) {
00086                 if (oldSize >= m_size) {
00087                         return true;
00088                 } else {
00089                         delete[] m_data;
00090                         m_data = new(std::nothrow) DataT[m_size];
00091                         return (m_data != 0);
00092                 }
00093         } else {
00094                 m_data = new(std::nothrow) DataT[m_size];
00095                 return (m_data != 0);
00096         }
00097 }
00098
00100 // Delete the memory buffer of this subband.
00101 void CSubband::FreeMemory() {
00102         if (m_data) {
00103                 delete[] m_data; m_data = 0;
00104         }
00105 }
00106
00108 // Perform subband quantization with given quantization parameter.
00109 // A scalar quantization (with dead-zone) is used. A large quantization value
00110 // results in strong quantization and therefore in big quality loss.
00111 // @param quantParam A quantization parameter (larger or equal to 0)
00112 void CSubband::Quantize(int quantParam) {
00113         if (m_orientation == LL) {
00114                 quantParam -= (m_level + 1);
00115                 // uniform rounding quantization
00116                 if (quantParam > 0) {
00117                         quantParam--;
00118                         for (UINT32 i=0; i < m_size; i++) {
00119                                 if (m_data[i] < 0) {
00120                                         m_data[i] = -(((-m_data[i] >> quantParam)
+ 1) >> 1);
00121                                 } else {
00122                                         m_data[i] = ((m_data[i] >> quantParam) + 1)
>> 1;
00123                                 }
00124                         }
00125                 }
00126         } else {
00127                 if (m_orientation == HH) {
00128                         quantParam -= (m_level - 1);
00129                 } else {
00130                         quantParam -= m_level;
00131                 }
00132                 // uniform deadzone quantization
00133                 if (quantParam > 0) {
00134                         int threshold = ((1 << quantParam) * 7)/5;      // good
value
00135                         quantParam--;
00136                         for (UINT32 i=0; i < m_size; i++) {
00137                                 if (m_data[i] < -threshold) {
00138                                         m_data[i] = -(((-m_data[i] >> quantParam)
+ 1) >> 1);
00139                                 } else if (m_data[i] > threshold) {
00140                                         m_data[i] = ((m_data[i] >> quantParam) + 1)
>> 1;
00141                                 } else {
00142                                         m_data[i] = 0;
00143                                 }
00144                         }
00145                 }
00146         }
00147 }
00148
00154 void CSubband::Dequantize(int quantParam) {
00155         if (m_orientation == LL) {
00156                 quantParam -= m_level + 1;
00157         } else if (m_orientation == HH) {
00158                 quantParam -= m_level - 1;
00159         } else {
```

```
00160                   quantParam -= m_level;
00161           }
00162           if (quantParam > 0) {
00163                   for (UINT32 i=0; i < m_size; i++) {
00164                           m_data[i] <<= quantParam;
00165                   }
00166           }
00167 }
00168
00177 void CSubband::ExtractTile(CEncoder& encoder, bool tile /*= false*/, UINT32 tileX
/*= 0*/, UINT32 tileY /*= 0*/) {
00178 #ifdef __PGFROISUPPORT__
00179           if (tile) {
00180                   // compute tile position and size
00181                   UINT32 xPos, yPos, w, h;
00182                   TilePosition(tileX, tileY, xPos, yPos, w, h);
00183
00184                   // write values into buffer using partitiong scheme
00185                   encoder.Partition(this, w, h, xPos + yPos*m_width, m_width);
00186           } else
00187 #endif
00188           {
00189                   tileX; tileY; tile; // prevents from unreferenced formal parameter
warning
00190                   // write values into buffer using partitiong scheme
00191                   encoder.Partition(this, m_width, m_height, 0, m_width);
00192           }
00193 }
00194
00203 void CSubband::PlaceTile(CDecoder& decoder, int quantParam, bool tile /*= false*/,
UINT32 tileX /*= 0*/, UINT32 tileY /*= 0*/) {
00204           // allocate memory
00205           if (!AllocMemory()) ReturnWithError(InsufficientMemory);
00206
00207           // correct quantParam with normalization factor
00208           if (m_orientation == LL) {
00209                   quantParam -= m_level + 1;
00210           } else if (m_orientation == HH) {
00211                   quantParam -= m_level - 1;
00212           } else {
00213                   quantParam -= m_level;
00214           }
00215           if (quantParam < 0) quantParam = 0;
00216
00217 #ifdef __PGFROISUPPORT__
00218           if (tile) {
00219                   UINT32 xPos, yPos, w, h;
00220
00221                   // compute tile position and size
00222                   TilePosition(tileX, tileY, xPos, yPos, w, h);
00223
00224                   ASSERT(xPos >= m_ROI.left && yPos >= m_ROI.top);
00225                   decoder.Partition(this, quantParam, w, h, (xPos - m_ROI.left) +
(yPos - m_ROI.top)*BufferWidth(), BufferWidth());
00226           } else
00227 #endif
00228           {
00229                   tileX; tileY; tile; // prevents from unreferenced formal parameter
warning
00230                   // read values into buffer using partitiong scheme
00231                   decoder.Partition(this, quantParam, m_width, m_height, 0,
m_width);
00232           }
00233 }
00234
00235
00236
00237 #ifdef __PGFROISUPPORT__
00240 void CSubband::SetAlignedROI(const PGFRect& roi) {
00241           ASSERT(roi.left <= m_width);
00242           ASSERT(roi.top <= m_height);
00243
00244           m_ROI = roi;
00245           if (m_ROI.right > m_width) m_ROI.right = m_width;
00246           if (m_ROI.bottom > m_height) m_ROI.bottom = m_height;
00247 }
00248
```

```
00257 void CSubband::TilePosition(UINT32 tileX, UINT32 tileY, UINT32& xPos, UINT32& yPos,
UINT32& w, UINT32& h) const {
00258         ASSERT(tileX < m_nTiles); ASSERT(tileY < m_nTiles);
00259         // example
00260         // band = HH, w = 30, ldTiles = 2 -> 4 tiles in a row/column
00261         // --> tile widths
00262         // 8 7 8 7
00263         //
00264         // tile partitioning scheme
00265         // 0 1 2 3
00266         // 4 5 6 7
00267         // 8 9 A B
00268         // C D E F
00269
00270         UINT32 nTiles = m_nTiles;
00271         UINT32 m;
00272         UINT32 left = 0, right = nTiles;
00273         UINT32 top = 0, bottom = nTiles;
00274
00275         xPos = 0;
00276         yPos = 0;
00277         w = m_width;
00278         h = m_height;
00279
00280         while (nTiles > 1) {
00281                 // compute xPos and w with binary search
00282                 m = left + ((right - left) >> 1);
00283                 if (tileX >= m) {
00284                         xPos += (w + 1) >> 1;
00285                         w >>= 1;
00286                         left = m;
00287                 } else {
00288                         w = (w + 1) >> 1;
00289                         right = m;
00290                 }
00291                 // compute yPos and h with binary search
00292                 m = top + ((bottom - top) >> 1);
00293                 if (tileY >= m) {
00294                         yPos += (h + 1) >> 1;
00295                         h >>= 1;
00296                         top = m;
00297                 } else {
00298                         h = (h + 1) >> 1;
00299                         bottom = m;
00300                 }
00301                 nTiles >>= 1;
00302         }
00303         ASSERT(xPos < m_width && (xPos + w <= m_width));
00304         ASSERT(yPos < m_height && (yPos + h <= m_height));
00305 }
00306
00309 void CSubband::TileIndex(bool topLeft, UINT32 xPos, UINT32 yPos, UINT32& tileX,
UINT32& tileY, UINT32& x, UINT32& y) const {
00310         UINT32 m;
00311         UINT32 left = 0, right = m_width;
00312         UINT32 top = 0, bottom = m_height;
00313         UINT32 nTiles = m_nTiles;
00314
00315         if (xPos > m_width) xPos = m_width;
00316         if (yPos > m_height) yPos = m_height;
00317
00318         if (topLeft) {
00319                 // compute tileX with binary search
00320                 tileX = 0;
00321                 while (nTiles > 1) {
00322                         nTiles >>= 1;
00323                         m = left + ((right - left + 1) >> 1);
00324                         if (xPos < m) {
00325                                 // exclusive m
00326                                 right = m;
00327                         } else {
00328                                 tileX += nTiles;
00329                                 left = m;
00330                         }
00331                 }
00332                 x = left;
00333                 ASSERT(tileX >= 0 && tileX < m_nTiles);
```

```
00334
00335                   // compute tileY with binary search
00336                   nTiles = m_nTiles;
00337                   tileY = 0;
00338                   while (nTiles > 1) {
00339                           nTiles >>= 1;
00340                           m = top + ((bottom - top + 1) >> 1);
00341                           if (yPos < m) {
00342                                   // exclusive m
00343                                   bottom = m;
00344                           } else {
00345                                   tileY += nTiles;
00346                                   top = m;
00347                           }
00348                   }
00349                   y = top;
00350                   ASSERT(tileY >= 0 && tileY < m_nTiles);
00351
00352           } else {
00353                   // compute tileX with binary search
00354                   tileX = 1;
00355                   while (nTiles > 1) {
00356                           nTiles >>= 1;
00357                           m = left + ((right - left + 1) >> 1);
00358                           if (xPos <= m) {
00359                                   // inclusive m
00360                                   right = m;
00361                           } else {
00362                                   tileX += nTiles;
00363                                   left = m;
00364                           }
00365                   }
00366                   x = right;
00367                   ASSERT(tileX > 0 && tileX <= m_nTiles);
00368
00369                   // compute tileY with binary search
00370                   nTiles = m_nTiles;
00371                   tileY = 1;
00372                   while (nTiles > 1) {
00373                           nTiles >>= 1;
00374                           m = top + ((bottom - top + 1) >> 1);
00375                           if (yPos <= m) {
00376                                   // inclusive m
00377                                   bottom = m;
00378                           } else {
00379                                   tileY += nTiles;
00380                                   top = m;
00381                           }
00382                   }
00383                   y = bottom;
00384                   ASSERT(tileY > 0 && tileY <= m_nTiles);
00385           }
00386 }
00387
00388 #endif
```

Subband.h File Reference

PGF wavelet subband class.
`#include "PGFtypes.h"`

Classes

- class **CSubband**
  Wavelet channel class.

Detailed Description

PGF wavelet subband class.

Author

C. Stamm

Definition in file **Subband.h**.

Subband.h

```
Go to the documentation of this file.00001 /*
00002  * The Progressive Graphics File; http://www.libpgf.org
00003  *
00004  * $Date: 2006-06-04 22:05:59 +0200 (So, 04 Jun 2006) $
00005  * $Revision: 229 $
00006  *
00007  * This file Copyright (C) 2006 xeraina GmbH, Switzerland
00008  *
00009  * This program is free software; you can redistribute it and/or
00010  * modify it under the terms of the GNU LESSER GENERAL PUBLIC LICENSE
00011  * as published by the Free Software Foundation; either version 2.1
00012  * of the License, or (at your option) any later version.
00013  *
00014  * This program is distributed in the hope that it will be useful,
00015  * but WITHOUT ANY WARRANTY; without even the implied warranty of
00016  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00017  * GNU General Public License for more details.
00018  *
00019  * You should have received a copy of the GNU General Public License
00020  * along with this program; if not, write to the Free Software
00021  * Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
00022  */
00023
00028
00029 #ifndef PGF_SUBBAND_H
00030 #define PGF_SUBBAND_H
00031
00032 #include "PGFtypes.h"
00033
00034 class CEncoder;
00035 class CDecoder;
00036 class CRoiIndices;
00037
00042 class CSubband {
00043         friend class CWaveletTransform;
00044         friend class CRoiIndices;
00045
00046 public:
00049         CSubband();
00050
00053         ~CSubband();
00054
00058         bool AllocMemory();
00059
00062         void FreeMemory();
00063
00072         void ExtractTile(CEncoder& encoder, bool tile = false, UINT32 tileX = 0,
UINT32 tileY = 0);
00073
00082         void PlaceTile(CDecoder& decoder, int quantParam, bool tile = false, UINT32
tileX = 0, UINT32 tileY = 0);
00083
00089         void Quantize(int quantParam);
00090
00096         void Dequantize(int quantParam);
00097
00102         void SetData(UINT32 pos, DataT v)       { ASSERT(pos < m_size); m_data[pos]
= v; }
00103
00107         DataT* GetBuffer()                                  { return m_data; }
00108
00113         DataT GetData(UINT32 pos) const         { ASSERT(pos < m_size); return
m_data[pos]; }
00114
00118         int GetLevel() const                            { return m_level; }
00119
00123         int GetHeight() const                           { return m_height; }
00124
00128         int GetWidth() const                            { return m_width; }
00129
00135         Orientation GetOrientation() const      { return m_orientation; }
00136
00137 #ifdef __PGFROISUPPORT__
```

```
00141         void IncBuffRow(UINT32 pos)     { m_dataPos = pos + BufferWidth(); }
00142
00143 #endif
00144
00145 private:
00146         void Initialize(UINT32 width, UINT32 height, int level, Orientation orient);
00147         void WriteBuffer(DataT val)                         { ASSERT(m_dataPos <
m_size); m_data[m_dataPos++] = val; }
00148         void SetBuffer(DataT* b)                            { ASSERT(b); m_data = b; }
00149         DataT ReadBuffer()                                          {
ASSERT(m_dataPos < m_size); return m_data[m_dataPos++]; }
00150
00151         UINT32 GetBuffPos() const                           { return m_dataPos; }
00152
00153 #ifdef __PGFROISUPPORT__
00154         UINT32 BufferWidth() const                          { return m_ROI.Width(); }
00155         void TilePosition(UINT32 tileX, UINT32 tileY, UINT32& left, UINT32& top,
UINT32& w, UINT32& h) const;
00156         void TileIndex(bool topLeft, UINT32 xPos, UINT32 yPos, UINT32& tileX,
UINT32& tileY, UINT32& x, UINT32& y) const;
00157         const PGFRect& GetAlignedROI() const { return m_ROI; }
00158         void SetNTiles(UINT32 nTiles)           { m_nTiles = nTiles; }
00159         void SetAlignedROI(const PGFRect& roi);
00160         void InitBuffPos(UINT32 left = 0, UINT32 top = 0)       { m_dataPos =
top*BufferWidth() + left; ASSERT(m_dataPos < m_size); }
00161 #else
00162         void InitBuffPos()                                          { m_dataPos = 0; }
00163 #endif
00164
00165 private:
00166         UINT32 m_width;
00167         UINT32 m_height;
00168         UINT32 m_size;
00169         int m_level;
00170         Orientation m_orientation;
00171         UINT32 m_dataPos;
00172         DataT* m_data;
00173
00174 #ifdef __PGFROISUPPORT__
00175         PGFRect m_ROI;
00176         UINT32  m_nTiles;
00177 #endif
00178 };
00179
00180 #endif //PGF_SUBBAND_H
```

WaveletTransform.cpp File Reference

PGF wavelet transform class implementation.

```
#include "WaveletTransform.h"
```

Macros

- #define **c1** 1
- #define **c2** 2

Detailed Description

PGF wavelet transform class implementation.

Author

C. Stamm

Definition in file **WaveletTransform.cpp**.

Macro Definition Documentation

#define c1 1

Definition at line **31** of file **WaveletTransform.cpp**.

#define c2 2

Definition at line **32** of file **WaveletTransform.cpp**.

WaveletTransform.cpp

```
Go to the documentation of this file.00001 /*
00002  * The Progressive Graphics File; http://www.libpgf.org
00003  *
00004  * $Date: 2006-05-18 16:03:32 +0200 (Do, 18 Mai 2006) $
00005  * $Revision: 194 $
00006  *
00007  * This file Copyright (C) 2006 xeraina GmbH, Switzerland
00008  *
00009  * This program is free software; you can redistribute it and/or
00010  * modify it under the terms of the GNU LESSER GENERAL PUBLIC LICENSE
00011  * as published by the Free Software Foundation; either version 2.1
00012  * of the License, or (at your option) any later version.
00013  *
00014  * This program is distributed in the hope that it will be useful,
00015  * but WITHOUT ANY WARRANTY; without even the implied warranty of
00016  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00017  * GNU General Public License for more details.
00018  *
00019  * You should have received a copy of the GNU General Public License
00020  * along with this program; if not, write to the Free Software
00021  * Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
00022  */
00023
00028
00029 #include "WaveletTransform.h"
00030
00031 #define c1 1    // best value 1
00032 #define c2 2    // best value 2
00033
00035 // Constructor: Constructs a wavelet transform pyramid of given size and levels.
00036 // @param width The width of the original image (at level 0) in pixels
00037 // @param height The height of the original image (at level 0) in pixels
00038 // @param levels The number of levels (>= 0)
00039 // @param data Input data of subband LL at level 0
00040 CWaveletTransform::CWaveletTransform(UINT32 width, UINT32 height, int levels,
DataT* data)
00041 : m_nLevels(levels + 1) // m_nLevels in CPGFImage determines the number of FWT steps;
this.m_nLevels determines the number subband-planes
00042 , m_subband(nullptr)
00043 #ifdef __PGFROISUPPORT__
00044 , m_indices(nullptr)
00045 #endif
00046 {
00047         ASSERT(m_nLevels > 0 && m_nLevels <= MaxLevel + 1);
00048         InitSubbands(width, height, data);
00049 }
00050
00052 // Initialize size subbands on all levels
00053 void CWaveletTransform::InitSubbands(UINT32 width, UINT32 height, DataT* data) {
00054         if (m_subband) Destroy();
00055
00056         // create subbands
00057         m_subband = new CSubband[m_nLevels][NSubbands];
00058
00059         // init subbands
00060         UINT32 loWidth = width;
00061         UINT32 hiWidth = width;
00062         UINT32 loHeight = height;
00063         UINT32 hiHeight = height;
00064
00065         for (int level = 0; level < m_nLevels; level++) {
00066                 m_subband[level][LL].Initialize(loWidth, loHeight, level, LL);
// LL
00067                 m_subband[level][HL].Initialize(hiWidth, loHeight, level, HL);
//    HL
00068                 m_subband[level][LH].Initialize(loWidth, hiHeight, level, LH);
// LH
00069                 m_subband[level][HH].Initialize(hiWidth, hiHeight, level, HH);
//    HH
00070                 hiWidth = loWidth >> 1;                 hiHeight = loHeight >> 1;
00071                 loWidth = (loWidth + 1) >> 1;   loHeight = (loHeight + 1) >> 1;
00072         }
00073         if (data) {
```

```
00074                   m_subband[0][LL].SetBuffer(data);
00075           }
00076 }
00077
00079 // Compute fast forward wavelet transform of LL subband at given level and
00080 // stores result in all 4 subbands of level + 1.
00081 // Wavelet transform used in writing a PGF file
00082 // Forward Transform of srcBand and split and store it into subbands on destLevel
00083 // low pass filter at even positions: 1/8[-1, 2, (6), 2, -1]
00084 // high pass filter at odd positions: 1/4[-2, (4), -2]
00085 // @param level A wavelet transform pyramid level (>= 0 && < Levels())
00086 // @param quant A quantization value (linear scalar quantization)
00087 // @return error in case of a memory allocation problem
00088 OSError CWaveletTransform::ForwardTransform(int level, int quant) {
00089           ASSERT(level >= 0 && level < m_nLevels - 1);
00090           const int destLevel = level + 1;
00091           ASSERT(m_subband[destLevel]);
00092           CSubband* srcBand = &m_subband[level][LL]; ASSERT(srcBand);
00093           const UINT32 width = srcBand->GetWidth();
00094           const UINT32 height = srcBand->GetHeight();
00095           DataT* src = srcBand->GetBuffer(); ASSERT(src);
00096           DataT *row0, *row1, *row2, *row3;
00097
00098           // Allocate memory for next transform level
00099           for (int i=0; i < NSubbands; i++) {
00100                   if (!m_subband[destLevel][i].AllocMemory()) return
InsufficientMemory;
00101           }
00102
00103           if (height >= FilterSize) { // changed from FilterSizeH to FilterSize
00104                   // top border handling
00105                   row0 = src; row1 = row0 + width; row2 = row1 + width;
00106                   ForwardRow(row0, width);
00107                   ForwardRow(row1, width);
00108                   ForwardRow(row2, width);
00109                   for (UINT32 k=0; k < width; k++) {
00110                           row1[k] -= ((row0[k] + row2[k] + c1) >> 1); // high pass
00111                           row0[k] += ((row1[k] + c1) >> 1); // low pass
00112                   }
00113                   InterleavedToSubbands(destLevel, row0, row1, width);
00114                   row0 = row1; row1 = row2; row2 += width; row3 = row2 + width;
00115
00116                   // middle part
00117                   for (UINT32 i=3; i < height-1; i += 2) {
00118                           ForwardRow(row2, width);
00119                           ForwardRow(row3, width);
00120                           for (UINT32 k=0; k < width; k++) {
00121                                   row2[k] -= ((row1[k] + row3[k] + c1) >> 1); // high
pass filter
00122                                   row1[k] += ((row0[k] + row2[k] + c2) >> 2); // low
pass filter
00123                           }
00124                           InterleavedToSubbands(destLevel, row1, row2, width);
00125                           row0 = row2; row1 = row3; row2 = row3 + width; row3 = row2
+ width;
00126                   }
00127
00128                   // bottom border handling
00129                   if (height & 1) {
00130                           for (UINT32 k=0; k < width; k++) {
00131                                   row1[k] += ((row0[k] + c1) >> 1); // low pass
00132                           }
00133                           InterleavedToSubbands(destLevel, row1, nullptr, width);
00134                           row0 = row1; row1 += width;
00135                   } else {
00136                           ForwardRow(row2, width);
00137                           for (UINT32 k=0; k < width; k++) {
00138                                   row2[k] -= row1[k]; // high pass
00139                                   row1[k] += ((row0[k] + row2[k] + c2) >> 2); // low
pass
00140                           }
00141                           InterleavedToSubbands(destLevel, row1, row2, width);
00142                           row0 = row1; row1 = row2; row2 += width;
00143                   }
00144           } else {
00145                   // if height is too small
00146                   row0 = src; row1 = row0 + width;
```

```
00147                   // first part
00148                   for (UINT32 k=0; k < height; k += 2) {
00149                           ForwardRow(row0, width);
00150                           ForwardRow(row1, width);
00151                           InterleavedToSubbands(destLevel, row0, row1, width);
00152                           row0 += width << 1; row1 += width << 1;
00153                   }
00154                   // bottom
00155                   if (height & 1) {
00156                           InterleavedToSubbands(destLevel, row0, nullptr, width);
00157                   }
00158           }
00159
00160           if (quant > 0) {
00161                   // subband quantization (without LL)
00162                   for (int i=1; i < NSubbands; i++) {
00163                           m_subband[destLevel][i].Quantize(quant);
00164                   }
00165                   // LL subband quantization
00166                   if (destLevel == m_nLevels - 1) {
00167                           m_subband[destLevel][LL].Quantize(quant);
00168                   }
00169           }
00170
00171           // free source band
00172           srcBand->FreeMemory();
00173           return NoError;
00174 }
00175
00177 // Forward transform one row
00178 // low pass filter at even positions: 1/8[-1, 2, (6), 2, -1]
00179 // high pass filter at odd positions: 1/4[-2, (4), -2]
00180 void CWaveletTransform::ForwardRow(DataT* src, UINT32 width) {
00181           if (width >= FilterSize) {
00182                   UINT32 i = 3;
00183
00184                   // left border handling
00185                   src[1] -= ((src[0] + src[2] + c1) >> 1); // high pass
00186                   src[0] += ((src[1] + c1) >> 1); // low pass
00187
00188                   // middle part
00189                   for (; i < width-1; i += 2) {
00190                           src[i] -= ((src[i-1] + src[i+1] + c1) >> 1); // high pass
00191                           src[i-1] += ((src[i-2] + src[i] + c2) >> 2); // low pass
00192                   }
00193
00194                   // right border handling
00195                   if (width & 1) {
00196                           src[i-1] += ((src[i-2] + c1) >> 1); // low pass
00197                   } else {
00198                           src[i] -= src[i-1]; // high pass
00199                           src[i-1] += ((src[i-2] + src[i] + c2) >> 2); // low pass
00200                   }
00201           }
00202 }
00203
00205 // Copy transformed and interleaved (L,H,L,H,...) rows loRow and hiRow to subbands
LL,HL,LH,HH
00206 void CWaveletTransform::InterleavedToSubbands(int destLevel, DataT* loRow, DataT*
hiRow, UINT32 width) {
00207           const UINT32 wquot = width >> 1;
00208           const bool wrem = (width & 1);
00209           CSubband &ll = m_subband[destLevel][LL], &hl = m_subband[destLevel][HL];
00210           CSubband &lh = m_subband[destLevel][LH], &hh = m_subband[destLevel][HH];
00211
00212           if (hiRow) {
00213                   for (UINT32 i=0; i < wquot; i++) {
00214                           ll.WriteBuffer(*loRow++);       // first access, than
increment
00215                           hl.WriteBuffer(*loRow++);
00216                           lh.WriteBuffer(*hiRow++);       // first access, than
increment
00217                           hh.WriteBuffer(*hiRow++);
00218                   }
00219                   if (wrem) {
00220                           ll.WriteBuffer(*loRow);
00221                           lh.WriteBuffer(*hiRow);
```

```
00222                   }
00223           } else {
00224                   for (UINT32 i=0; i < wquot; i++) {
00225                           ll.WriteBuffer(*loRow++);       // first access, than
increment
00226                           hl.WriteBuffer(*loRow++);
00227                   }
00228                   if (wrem) ll.WriteBuffer(*loRow);
00229           }
00230 }
00231
00233 // Compute fast inverse wavelet transform of all 4 subbands of given level and
00234 // stores result in LL subband of level - 1.
00235 // Inverse wavelet transform used in reading a PGF file
00236 // Inverse Transform srcLevel and combine to destBand
00237 // low-pass coefficients at even positions, high-pass coefficients at odd positions
00238 // inverse filter for even positions: 1/4[-1, (4), -1]
00239 // inverse filter for odd positions: 1/8[-1, 4, (6), 4, -1]
00240 // @param srcLevel A wavelet transform pyramid level (> 0 && <= Levels())
00241 // @param w [out] A pointer to the returned width of subband LL (in pixels)
00242 // @param h [out] A pointer to the returned height of subband LL (in pixels)
00243 // @param data [out] A pointer to the returned array of image data
00244 // @return error in case of a memory allocation problem
00245 OSError CWaveletTransform::InverseTransform(int srcLevel, UINT32* w, UINT32* h,
DataT** data) {
00246           ASSERT(srcLevel > 0 && srcLevel < m_nLevels);
00247           const int destLevel = srcLevel - 1;
00248           ASSERT(m_subband[destLevel]);
00249           CSubband* destBand = &m_subband[destLevel][LL];
00250           UINT32 width, height;
00251
00252           // allocate memory for the results of the inverse transform
00253           if (!destBand->AllocMemory()) return InsufficientMemory;
00254           DataT *origin = destBand->GetBuffer(), *row0, *row1, *row2, *row3;
00255
00256 #ifdef __PGFROISUPPORT__
00257           PGFRect destROI = destBand->GetAlignedROI();
00258           const UINT32 destWidth  = destROI.Width();  // destination buffer width
00259           const UINT32 destHeight = destROI.Height(); // destination buffer height
00260           width = destWidth;              // destination working width
00261           height = destHeight;    // destination working height
00262
00263           // update destination ROI
00264           if (destROI.top & 1) {
00265                   destROI.top++;
00266                   origin += destWidth;
00267                   height--;
00268           }
00269           if (destROI.left & 1) {
00270                   destROI.left++;
00271                   origin++;
00272                   width--;
00273           }
00274
00275           // init source buffer position
00276           const UINT32 leftD = destROI.left >> 1;
00277           const UINT32 left0 = m_subband[srcLevel][LL].GetAlignedROI().left;
00278           const UINT32 left1 = m_subband[srcLevel][HL].GetAlignedROI().left;
00279           const UINT32 topD = destROI.top >> 1;
00280           const UINT32 top0 = m_subband[srcLevel][LL].GetAlignedROI().top;
00281           const UINT32 top1 = m_subband[srcLevel][LH].GetAlignedROI().top;
00282           ASSERT(m_subband[srcLevel][LH].GetAlignedROI().left == left0);
00283           ASSERT(m_subband[srcLevel][HH].GetAlignedROI().left == left1);
00284           ASSERT(m_subband[srcLevel][HL].GetAlignedROI().top == top0);
00285           ASSERT(m_subband[srcLevel][HH].GetAlignedROI().top == top1);
00286
00287           UINT32 srcOffsetX[2] = { 0, 0 };
00288           UINT32 srcOffsetY[2] = { 0, 0 };
00289
00290           if (leftD >= __max(left0, left1)) {
00291                   srcOffsetX[0] = leftD - left0;
00292                   srcOffsetX[1] = leftD - left1;
00293           } else {
00294                   if (left0 <= left1) {
00295                           const UINT32 dx = (left1 - leftD) << 1;
00296                           destROI.left += dx;
00297                           origin += dx;
```

```
00298                           width -= dx;
00299                           srcOffsetX[0] = left1 - left0;
00300                   } else {
00301                           const UINT32 dx = (left0 - leftD) << 1;
00302                           destROI.left += dx;
00303                           origin += dx;
00304                           width -= dx;
00305                           srcOffsetX[1] = left0 - left1;
00306                   }
00307           }
00308           if (topD >= __max(top0, top1)) {
00309                   srcOffsetY[0] = topD - top0;
00310                   srcOffsetY[1] = topD - top1;
00311           } else {
00312                   if (top0 <= top1) {
00313                           const UINT32 dy = (top1 - topD) << 1;
00314                           destROI.top += dy;
00315                           origin += dy*destWidth;
00316                           height -= dy;
00317                           srcOffsetY[0] = top1 - top0;
00318                   } else {
00319                           const UINT32 dy = (top0 - topD) << 1;
00320                           destROI.top += dy;
00321                           origin += dy*destWidth;
00322                           height -= dy;
00323                           srcOffsetY[1] = top0 - top1;
00324                   }
00325           }
00326
00327           m_subband[srcLevel][LL].InitBuffPos(srcOffsetX[0], srcOffsetY[0]);
00328           m_subband[srcLevel][HL].InitBuffPos(srcOffsetX[1], srcOffsetY[0]);
00329           m_subband[srcLevel][LH].InitBuffPos(srcOffsetX[0], srcOffsetY[1]);
00330           m_subband[srcLevel][HH].InitBuffPos(srcOffsetX[1], srcOffsetY[1]);
00331
00332 #else
00333           width = destBand->GetWidth();
00334           height = destBand->GetHeight();
00335           PGFRect destROI(0, 0, width, height);
00336           const UINT32 destWidth = width; // destination buffer width
00337           const UINT32 destHeight = height; // destination buffer height
00338
00339           // init source buffer position
00340           for (int i = 0; i < NSubbands; i++) {
00341                   m_subband[srcLevel][i].InitBuffPos();
00342           }
00343 #endif
00344
00345           if (destHeight >= FilterSize) { // changed from FilterSizeH to FilterSize
00346                   // top border handling
00347                   row0 = origin; row1 = row0 + destWidth;
00348                   SubbandsToInterleaved(srcLevel, row0, row1, width);
00349                   for (UINT32 k = 0; k < width; k++) {
00350                           row0[k] -= ((row1[k] + c1) >> 1); // even
00351                   }
00352
00353                   // middle part
00354                   row2 = row1 + destWidth; row3 = row2 + destWidth;
00355                   for (UINT32 i = destROI.top + 2; i < destROI.bottom - 1; i += 2) {
00356                           SubbandsToInterleaved(srcLevel, row2, row3, width);
00357                           for (UINT32 k = 0; k < width; k++) {
00358                                   row2[k] -= ((row1[k] + row3[k] + c2) >> 2); // even
00359                                   row1[k] += ((row0[k] + row2[k] + c1) >> 1); // odd
00360                           }
00361                           InverseRow(row0, width);
00362                           InverseRow(row1, width);
00363                           row0 = row2; row1 = row3; row2 = row1 + destWidth; row3 =
row2 + destWidth;
00364                   }
00365
00366                   // bottom border handling
00367                   if (height & 1) {
00368                           SubbandsToInterleaved(srcLevel, row2, nullptr, width);
00369                           for (UINT32 k = 0; k < width; k++) {
00370                                   row2[k] -= ((row1[k] + c1) >> 1); // even
00371                                   row1[k] += ((row0[k] + row2[k] + c1) >> 1); // odd
00372                           }
00373                           InverseRow(row0, width);
```

```
00374                           InverseRow(row1, width);
00375                           InverseRow(row2, width);
00376                           row0 = row1; row1 = row2; row2 += destWidth;
00377                   } else {
00378                           for (UINT32 k = 0; k < width; k++) {
00379                                   row1[k] += row0[k];
00380                           }
00381                           InverseRow(row0, width);
00382                           InverseRow(row1, width);
00383                           row0 = row1; row1 += destWidth;
00384                   }
00385           } else {
00386                   // height is too small
00387                   row0 = origin; row1 = row0 + destWidth;
00388                   // first part
00389                   for (UINT32 k = 0; k < height; k += 2) {
00390                           SubbandsToInterleaved(srcLevel, row0, row1, width);
00391                           InverseRow(row0, width);
00392                           InverseRow(row1, width);
00393                           row0 += destWidth << 1; row1 += destWidth << 1;
00394                   }
00395                   // bottom
00396                   if (height & 1) {
00397                           SubbandsToInterleaved(srcLevel, row0, nullptr, width);
00398                           InverseRow(row0, width);
00399                   }
00400           }
00401
00402           // free memory of the current srcLevel
00403           for (int i = 0; i < NSubbands; i++) {
00404                   m_subband[srcLevel][i].FreeMemory();
00405           }
00406
00407           // return info
00408           *w = destWidth;
00409           *h = destHeight;
00410           *data = destBand->GetBuffer();
00411           return NoError;
00412 }
00413
00415 // Inverse Wavelet Transform of one row
00416 // low-pass coefficients at even positions, high-pass coefficients at odd positions
00417 // inverse filter for even positions: 1/4[-1, (4), -1]
00418 // inverse filter for odd positions: 1/8[-1, 4, (6), 4, -1]
00419 void CWaveletTransform::InverseRow(DataT* dest, UINT32 width) {
00420           if (width >= FilterSize) {
00421                   UINT32 i = 2;
00422
00423                   // left border handling
00424                   dest[0] -= ((dest[1] + c1) >> 1); // even
00425
00426                   // middle part
00427                   for (; i < width - 1; i += 2) {
00428                           dest[i] -= ((dest[i-1] + dest[i+1] + c2) >> 2); // even
00429                           dest[i-1] += ((dest[i-2] + dest[i] + c1) >> 1); // odd
00430                   }
00431
00432                   // right border handling
00433                   if (width & 1) {
00434                           dest[i] -= ((dest[i-1] + c1) >> 1); // even
00435                           dest[i-1] += ((dest[i-2] + dest[i] + c1) >> 1); // odd
00436                   } else {
00437                           dest[i-1] += dest[i-2]; // odd
00438                   }
00439           }
00440 }
00441
00443 // Copy transformed coefficients from subbands LL,HL,LH,HH to interleaved format
(L,H,L,H,...)
00444 void CWaveletTransform::SubbandsToInterleaved(int srcLevel, DataT* loRow, DataT*
hiRow, UINT32 width) {
00445           const UINT32 wquot = width >> 1;
00446           const bool wrem = (width & 1);
00447           CSubband &ll = m_subband[srcLevel][LL], &hl = m_subband[srcLevel][HL];
00448           CSubband &lh = m_subband[srcLevel][LH], &hh = m_subband[srcLevel][HH];
00449
00450           if (hiRow) {
```

```
00451         #ifdef __PGFROISUPPORT__
00452                 const bool storePos = wquot < ll.BufferWidth();
00453                 UINT32 llPos = 0, hlPos = 0, lhPos = 0, hhPos = 0;
00454
00455                 if (storePos) {
00456                         // save current src buffer positions
00457                         llPos = ll.GetBuffPos();
00458                         hlPos = hl.GetBuffPos();
00459                         lhPos = lh.GetBuffPos();
00460                         hhPos = hh.GetBuffPos();
00461                 }
00462         #endif
00463
00464                 for (UINT32 i=0; i < wquot; i++) {
00465                         *loRow++ = ll.ReadBuffer();// first access, than increment
00466                         *loRow++ = hl.ReadBuffer();// first access, than increment
00467                         *hiRow++ = lh.ReadBuffer();// first access, than increment
00468                         *hiRow++ = hh.ReadBuffer();// first access, than increment
00469                 }
00470
00471                 if (wrem) {
00472                         *loRow++ = ll.ReadBuffer();// first access, than increment
00473                         *hiRow++ = lh.ReadBuffer();// first access, than increment
00474                 }
00475
00476         #ifdef __PGFROISUPPORT__
00477                 if (storePos) {
00478                         // increment src buffer positions
00479                         ll.IncBuffRow(llPos);
00480                         hl.IncBuffRow(hlPos);
00481                         lh.IncBuffRow(lhPos);
00482                         hh.IncBuffRow(hhPos);
00483                 }
00484         #endif
00485
00486         } else {
00487         #ifdef __PGFROISUPPORT__
00488                 const bool storePos = wquot < ll.BufferWidth();
00489                 UINT32 llPos = 0, hlPos = 0;
00490
00491                 if (storePos) {
00492                         // save current src buffer positions
00493                         llPos = ll.GetBuffPos();
00494                         hlPos = hl.GetBuffPos();
00495                 }
00496         #endif
00497
00498                 for (UINT32 i=0; i < wquot; i++) {
00499                         *loRow++ = ll.ReadBuffer();// first access, than increment
00500                         *loRow++ = hl.ReadBuffer();// first access, than increment
00501                 }
00502                 if (wrem) *loRow++ = ll.ReadBuffer();
00503
00504         #ifdef __PGFROISUPPORT__
00505                 if (storePos) {
00506                         // increment src buffer positions
00507                         ll.IncBuffRow(llPos);
00508                         hl.IncBuffRow(hlPos);
00509                 }
00510         #endif
00511         }
00512 }
00513
00514 #ifdef __PGFROISUPPORT__
00518 void CWaveletTransform::SetROI(PGFRect roi) {
00519         const UINT32 delta = (FilterSize >> 1) << m_nLevels;
00520
00521         // create tile indices
00522         delete[] m_indices;
00523         m_indices = new PGFRect[m_nLevels];
00524
00525         // enlarge rect: add margin
00526         roi.left = (roi.left > delta) ? roi.left - delta : 0;
00527         roi.top  = (roi.top  > delta) ? roi.top  - delta : 0;
00528         roi.right += delta;
00529         roi.bottom += delta;
00530
```

```
00531         for (int l = 0; l < m_nLevels; l++) {
00532                 PGFRect alignedROI;
00533                 PGFRect& indices = m_indices[l];
00534                 UINT32 nTiles = GetNofTiles(l);
00535                 CSubband& subband = m_subband[l][LL];
00536
00537                 // use roi to determine the necessary tile indices (for all subbands
the same) and aligned ROI for LL subband
00538                 subband.SetNTiles(nTiles); // must be called before TileIndex()
00539                 subband.TileIndex(true, roi.left, roi.top, indices.left,
indices.top, alignedROI.left, alignedROI.top);
00540                 subband.TileIndex(false, roi.right, roi.bottom, indices.right,
indices.bottom, alignedROI.right, alignedROI.bottom);
00541                 subband.SetAlignedROI(alignedROI);
00542                 ASSERT(l == 0 ||
00543                         (m_indices[l-1].left >= 2*m_indices[l].left &&
00544                         m_indices[l-1].top >= 2*m_indices[l].top &&
00545                         m_indices[l-1].right <= 2*m_indices[l].right &&
00546                         m_indices[l-1].bottom <= 2*m_indices[l].bottom));
00547
00548                 // determine aligned ROI of other three subbands
00549                 PGFRect aroi;
00550                 UINT32 w, h;
00551                 for (int b = 1; b < NSubbands; b++) {
00552                         CSubband& sb = m_subband[l][b];
00553                         sb.SetNTiles(nTiles); // must be called before
TilePosition()
00554                         sb.TilePosition(indices.left, indices.top, aroi.left,
aroi.top, w, h);
00555                         sb.TilePosition(indices.right - 1, indices.bottom - 1,
aroi.right, aroi.bottom, w, h);
00556                         aroi.right += w;
00557                         aroi.bottom += h;
00558                         sb.SetAlignedROI(aroi);
00559                 }
00560
00561                 // use aligned ROI of LL subband for next level
00562                 roi.left = alignedROI.left >> 1;
00563                 roi.top = alignedROI.top >> 1;
00564                 roi.right = (alignedROI.right + 1) >> 1;
00565                 roi.bottom = (alignedROI.bottom + 1) >> 1;
00566         }
00567 }
00568
00569 #endif // __PGFROISUPPORT__
```

WaveletTransform.h File Reference

PGF wavelet transform class.

```
#include "PGFtypes.h"
#include "Subband.h"
```

Classes

- class **CWaveletTransform**
  PGF wavelet transform.

Variables

- const UINT32 **FilterSizeL** = 5
  number of coefficients of the low pass filter
- const UINT32 **FilterSizeH** = 3
  number of coefficients of the high pass filter
- const UINT32 **FilterSize** = **__max**(**FilterSizeL**, **FilterSizeH**)

Detailed Description

PGF wavelet transform class.

Author

C. Stamm

Definition in file **WaveletTransform.h**.

Variable Documentation

const UINT32 FilterSize = __max(FilterSizeL, FilterSizeH)

Definition at line **39** of file **WaveletTransform.h**.

const UINT32 FilterSizeH = 3

number of coefficients of the high pass filter

Definition at line **38** of file **WaveletTransform.h**.

const UINT32 FilterSizeL = 5

number of coefficients of the low pass filter

Definition at line **37** of file **WaveletTransform.h**.

WaveletTransform.h

```
Go to the documentation of this file.00001 /*
00002  * The Progressive Graphics File; http://www.libpgf.org
00003  *
00004  * $Date: 2006-05-18 16:03:32 +0200 (Do, 18 Mai 2006) $
00005  * $Revision: 194 $
00006  *
00007  * This file Copyright (C) 2006 xeraina GmbH, Switzerland
00008  *
00009  * This program is free software; you can redistribute it and/or
00010  * modify it under the terms of the GNU LESSER GENERAL PUBLIC LICENSE
00011  * as published by the Free Software Foundation; either version 2.1
00012  * of the License, or (at your option) any later version.
00013  *
00014  * This program is distributed in the hope that it will be useful,
00015  * but WITHOUT ANY WARRANTY; without even the implied warranty of
00016  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00017  * GNU General Public License for more details.
00018  *
00019  * You should have received a copy of the GNU General Public License
00020  * along with this program; if not, write to the Free Software
00021  * Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
00022  */
00023
00028
00029 #ifndef PGF_WAVELETTRANSFORM_H
00030 #define PGF_WAVELETTRANSFORM_H
00031
00032 #include "PGFtypes.h"
00033 #include "Subband.h"
00034
00036 // Constants
00037 const UINT32 FilterSizeL = 5;
00038 const UINT32 FilterSizeH = 3;
00039 const UINT32 FilterSize = __max(FilterSizeL, FilterSizeH);
00040
00041 #ifdef __PGFROISUPPORT__
00046 class CRoiIndices {
00047 };
00048 #endif //__PGFROISUPPORT__
00049
00050
00055 class CWaveletTransform {
00056         friend class CSubband;
00057
00058 public:
00065         CWaveletTransform(UINT32 width, UINT32 height, int levels, DataT* data = 
nullptr);
00066
00069         ~CWaveletTransform() { Destroy(); }
00070
00077         OSError ForwardTransform(int level, int quant);
00078
00087         OSError InverseTransform(int level, UINT32* width, UINT32* height, DataT** 
data);
00088
00093         CSubband* GetSubband(int level, Orientation orientation) {
00094                 ASSERT(level >= 0 && level < m_nLevels);
00095                 return &m_subband[level][orientation];
00096         }
00097
00098 #ifdef __PGFROISUPPORT__
00102         void SetROI(PGFRect rect);
00103
00109         const bool TileIsRelevant(int level, UINT32 tileX, UINT32 tileY) const { 
ASSERT(m_indices); ASSERT(level >= 0 && level < m_nLevels); return 
m_indices[level].IsInside(tileX, tileY); }
00110
00115         UINT32 GetNofTiles(int level) const { ASSERT(level >= 0 && level < 
m_nLevels); return 1 << (m_nLevels - level - 1); }
00116
00120         const PGFRect& GetAlignedROI(int level) const           { return 
m_subband[level][LL].GetAlignedROI(); }
00121
```

```
00122 #endif // __PGFROISUPPORT__
00123
00124 private:
00125         void Destroy() {
00126                 delete[] m_subband; m_subband = nullptr;
00127         #ifdef __PGFROISUPPORT__
00128                 delete[] m_indices; m_indices = nullptr;
00129         #endif
00130         }
00131         void InitSubbands(UINT32 width, UINT32 height, DataT* data);
00132         void ForwardRow(DataT* buff, UINT32 width);
00133         void InverseRow(DataT* buff, UINT32 width);
00134         void InterleavedToSubbands(int destLevel, DataT* loRow, DataT* hiRow,
UINT32 width);
00135         void SubbandsToInterleaved(int srcLevel, DataT* loRow, DataT* hiRow, UINT32
width);
00136
00137 #ifdef __PGFROISUPPORT__
00138         PGFRect *m_indices;
00139 #endif //__PGFROISUPPORT__
00140
00141         int                     m_nLevels;
00142         CSubband        (*m_subband)[NSubbands];
00143 };
00144
00145 #endif //PGF_WAVELETTRANSFORM_H
```

Index

INDEX