libpgf

Christoph Stamm
Version 7.15.32
8/6/2015 8:12:00 PM

Table of Contents

Hierarchical Index

Class Hierarchy

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

Class Index

Class List

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

File Index

File List

Here is a list of all files with brief descriptions:

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.

```
76 : m_stream(stream)
```

```
   77 , m_startPos(0)
   78 , m_streamSizeEstimation(0)
   79 , m_encodedHeaderLength(0)
   80 , m_currentBlockIndex(0)
   81 , m_macroBlocksAvailable(0)
   82 #ifdef __PGFROISUPPORT__
   83 , m_roi(false)
   84 #endif
   85 {
   86         ASSERT(m_stream);
   87 
   88         int count, expected;
   89 
   90         // set number of threads
   91 #ifdef LIBPGF_USE_OPENMP
   92         m_macroBlockLen = omp_get_num_procs();
   93 #else
   94         m_macroBlockLen = 1;
   95 #endif
   96 
   97         if (useOMP && m_macroBlockLen > 1) {
   98 #ifdef LIBPGF_USE_OPENMP
   99                 omp_set_num_threads(m_macroBlockLen);
  100 #endif
  101 
  102                 // create macro block array
  103                 m_macroBlocks = new(std::nothrow) CMacroBlock*[m_macroBlockLen];
  104                 if (!m_macroBlocks) ReturnWithError(InsufficientMemory);
  105                 for (int i=0; i < m_macroBlockLen; i++) m_macroBlocks[i] = new
CMacroBlock();
  106                 m_currentBlock = m_macroBlocks[m_currentBlockIndex];
  107         } else {
  108                 m_macroBlocks = 0;
  109                 m_macroBlockLen = 1; // there is only one macro block
  110                 m_currentBlock = new CMacroBlock();
  111         }
  112 
  113         // store current stream position
  114         m_startPos = m_stream->GetPos();
  115 
  116         // read magic and version
  117         count = expected = MagicVersionSize;
  118         m_stream->Read(&count, &preHeader);
  119         if (count != expected) ReturnWithError(MissingData);
  120 
  121         // read header size
  122         if (preHeader.version & Version6) {
  123                 // 32 bit header size since version 6
  124                 count = expected = 4;
  125         } else {
  126                 count = expected = 2;
  127         }
  128         m_stream->Read(&count, ((UINT8*)&preHeader) + MagicVersionSize);
  129         if (count != expected) ReturnWithError(MissingData);
  130 
  131         // make sure the values are correct read
  132         preHeader.hSize = __VAL(preHeader.hSize);
  133 
  134         // check magic number
  135         if (memcmp(preHeader.magic, PGFMagic, 3) != 0) {
  136                 // error condition: wrong Magic number
  137                 ReturnWithError(FormatCannotRead);
  138         }
  139 
  140         // read file header
  141         count = expected = (preHeader.hSize < HeaderSize) ? preHeader.hSize : HeaderSize;
  142         m_stream->Read(&count, &header);
  143         if (count != expected) ReturnWithError(MissingData);
  144 
  145         // make sure the values are correct read
  146         header.height = __VAL(UINT32(header.height));
```

```
  147         header.width = __VAL(UINT32(header.width));
  148
  149         // be ready to read all versions including version 0
  150         if (preHeader.version > 0) {
  151 #ifndef __PGFROISUPPORT__
  152                 // check ROI usage
  153                 if (preHeader.version & PGFROI) ReturnWithError(FormatCannotRead);
  154 #endif
  155
  156                 UINT32 size = preHeader.hSize;
  157
  158                 if (size > HeaderSize) {
  159                         size -= HeaderSize;
  160                         count = 0;
  161
  162                         // read post-header
  163                         if (header.mode == ImageModeIndexedColor) {
  164                                 ASSERT((size_t)size >= ColorTableSize);
  165                                 // read color table
  166                                 count = expected = ColorTableSize;
  167                                 m_stream->Read(&count, postHeader.clut);
  168                                 if (count != expected) ReturnWithError(MissingData);
  169                         }
  170
  171                         if (size > (UINT32)count) {
  172                                 size -= count;
  173
  174                                 // read/skip user data
  175                                 UserdataPolicy policy = (UserdataPolicy)((userDataPolicy
<= MaxUserDataSize) ? UP_CachePrefix : 0xFFFFFFFF - userDataPolicy);
  176                                 userDataPos = m_stream->GetPos();
  177                                 postHeader.userDataLen = size;
  178
  179                                 if (policy == UP_Skip) {
  180                                         postHeader.cachedUserDataLen = 0;
  181                                         postHeader.userData = nullptr;
  182                                         Skip(size);
  183                                 } else {
  184                                         postHeader.cachedUserDataLen = (policy ==
UP_CachePrefix) ? __min(size, userDataPolicy) : size;
  185
  186                                         // create user data memory block
  187                                         postHeader.userData = new(std::nothrow)
UINT8[postHeader.cachedUserDataLen];
  188                                         if (!postHeader.userData)
ReturnWithError(InsufficientMemory);
  189
  190                                         // read user data
  191                                         count = expected = postHeader.cachedUserDataLen;
  192                                         m_stream->Read(&count, postHeader.userData);
  193                                         if (count != expected)
ReturnWithError(MissingData);
  194
  195                                         // skip remaining user data
  196                                         if (postHeader.cachedUserDataLen < size)
Skip(size - postHeader.cachedUserDataLen);
  197                                 }
  198                         }
  199                 }
  200
  201                 // create levelLength
  202                 levelLength = new(std::nothrow) UINT32[header.nLevels];
  203                 if (!levelLength) ReturnWithError(InsufficientMemory);
  204
  205                 // read levelLength
  206                 count = expected = header.nLevels*WordBytes;
  207                 m_stream->Read(&count, levelLength);
  208                 if (count != expected) ReturnWithError(MissingData);
  209
  210 #ifdef PGF_USE_BIG_ENDIAN
  211                 // make sure the values are correct read
```

```
  212                 for (int i=0; i < header.nLevels; i++) {
  213                         levelLength[i] = __VAL(levelLength[i]);
  214                 }
  215 #endif
  216
  217                 // compute the total size in bytes; keep attention: level length information
is optional
  218                 for (int i=0; i < header.nLevels; i++) {
  219                         m_streamSizeEstimation += levelLength[i];
  220                 }
  221
  222         }
  223
  224         // store current stream position
  225         m_encodedHeaderLength = UINT32(m_stream->GetPos() - m_startPos);
  226 }
```

CDecoder::~CDecoder ()

Destructor.

Definition at line 230 of file Decoder.cpp.

```
  230                     {
  231         if (m_macroBlocks) {
  232                 for (int i=0; i < m_macroBlockLen; i++) delete m_macroBlocks[i];
  233                 delete[] m_macroBlocks;
  234         } else {
  235                 delete m_currentBlock;
  236         }
  237 }
```

Member Function Documentation

void CDecoder::DecodeBuffer ()

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

Definition at line 493 of file Decoder.cpp.

```
  493                             {
  494         ASSERT(m_macroBlocksAvailable <= 0);
  495
  496         // macro block management
  497         if (m_macroBlockLen == 1) {
  498                 ASSERT(m_currentBlock);
  499                 ReadMacroBlock(m_currentBlock);
  500                 m_currentBlock->BitplaneDecode();
  501                 m_macroBlocksAvailable = 1;
  502         } else {
  503                 m_macroBlocksAvailable = 0;
  504                 for (int i=0; i < m_macroBlockLen; i++) {
  505                         // read sequentially several blocks
  506                         try {
  507                                 ReadMacroBlock(m_macroBlocks[i]);
  508                                 m_macroBlocksAvailable++;
  509                         } catch(IOException& ex) {
  510                                 if (ex.error == MissingData || ex.error ==
FormatCannotRead) {
  511                                         break; // no further data available or the data
isn't valid PGF data (might occur in streaming or PPPExt)
  512                                 } else {
  513                                         throw;
  514                                 }
  515                         }
  516                 }
  517 #ifdef LIBPGF_USE_OPENMP
  518                 // decode in parallel
```

```
  519                 #pragma omp parallel for default(shared) //no declared exceptions in next
block
  520 #endif
  521                 for (int i=0; i < m_macroBlocksAvailable; i++) {
  522                         m_macroBlocks[i]->BitplaneDecode();
  523                 }
  524
  525                 // prepare current macro block
  526                 m_currentBlockIndex = 0;
  527                 m_currentBlock = m_macroBlocks[m_currentBlockIndex];
  528         }
  529 }
```

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 332 of file Decoder.cpp.

```
  332
{
  333         CSubband* hlBand = wtChannel->GetSubband(level, HL);
  334         CSubband* lhBand = wtChannel->GetSubband(level, LH);
  335         const div_t lhH = div(lhBand->GetHeight(), InterBlockSize);
  336         const div_t hlW = div(hlBand->GetWidth(), InterBlockSize);
  337         const int hlws = hlBand->GetWidth() - InterBlockSize;
  338         const int hlwr = hlBand->GetWidth() - hlW.rem;
  339         const int lhws = lhBand->GetWidth() - InterBlockSize;
  340         const int lhwr = lhBand->GetWidth() - hlW.rem;
  341         int hlPos, lhPos;
  342         int hlBase = 0, lhBase = 0, hlBase2, lhBase2;
  343
  344         ASSERT(lhBand->GetWidth() >= hlBand->GetWidth());
  345         ASSERT(hlBand->GetHeight() >= lhBand->GetHeight());
  346
  347         if (!hlBand->AllocMemory()) ReturnWithError(InsufficientMemory);
  348         if (!lhBand->AllocMemory()) ReturnWithError(InsufficientMemory);
  349
  350         // correct quantParam with normalization factor
  351         quantParam -= level;
  352         if (quantParam < 0) quantParam = 0;
  353
  354         // main height
  355         for (int i=0; i < lhH.quot; i++) {
  356                 // main width
  357                 hlBase2 = hlBase;
  358                 lhBase2 = lhBase;
  359                 for (int j=0; j < hlW.quot; j++) {
  360                         hlPos = hlBase2;
  361                         lhPos = lhBase2;
  362                         for (int y=0; y < InterBlockSize; y++) {
  363                                 for (int x=0; x < InterBlockSize; x++) {
  364                                         DequantizeValue(hlBand, hlPos, quantParam);
  365                                         DequantizeValue(lhBand, lhPos, quantParam);
  366                                         hlPos++;
  367                                         lhPos++;
  368                                 }
  369                                 hlPos += hlws;
  370                                 lhPos += lhws;
  371                         }
  372                         hlBase2 += InterBlockSize;
```

```
373                         lhBase2 += InterBlockSize;
374                 }
375                 // rest of width
376                 hlPos = hlBase2;
377                 lhPos = lhBase2;
378                 for (int y=0; y < InterBlockSize; y++) {
379                         for (int x=0; x < hlW.rem; x++) {
380                                 DequantizeValue(hlBand, hlPos, quantParam);
381                                 DequantizeValue(lhBand, lhPos, quantParam);
382                                 hlPos++;
383                                 lhPos++;
384                         }
385                         // width difference between HL and LH
386                         if (lhBand->GetWidth() > hlBand->GetWidth()) {
387                                 DequantizeValue(lhBand, lhPos, quantParam);
388                         }
389                         hlPos += hlwr;
390                         lhPos += lhwr;
391                         hlBase += hlBand->GetWidth();
392                         lhBase += lhBand->GetWidth();
393                 }
394         }
395         // main width
396         hlBase2 = hlBase;
397         lhBase2 = lhBase;
398         for (int j=0; j < hlW.quot; j++) {
399                 // rest of height
400                 hlPos = hlBase2;
401                 lhPos = lhBase2;
402                 for (int y=0; y < lhH.rem; y++) {
403                         for (int x=0; x < InterBlockSize; x++) {
404                                 DequantizeValue(hlBand, hlPos, quantParam);
405                                 DequantizeValue(lhBand, lhPos, quantParam);
406                                 hlPos++;
407                                 lhPos++;
408                         }
409                         hlPos += hlws;
410                         lhPos += lhws;
411                 }
412                 hlBase2 += InterBlockSize;
413                 lhBase2 += InterBlockSize;
414         }
415         // rest of height
416         hlPos = hlBase2;
417         lhPos = lhBase2;
418         for (int y=0; y < lhH.rem; y++) {
419                 // rest of width
420                 for (int x=0; x < hlW.rem; x++) {
421                         DequantizeValue(hlBand, hlPos, quantParam);
422                         DequantizeValue(lhBand, lhPos, quantParam);
423                         hlPos++;
424                         lhPos++;
425                 }
426                 // width difference between HL and LH
427                 if (lhBand->GetWidth() > hlBand->GetWidth()) {
428                         DequantizeValue(lhBand, lhPos, quantParam);
429                 }
430                 hlPos += hlwr;
431                 lhPos += lhwr;
432                 hlBase += hlBand->GetWidth();
433         }
434         // height difference between HL and LH
435         if (hlBand->GetHeight() > lhBand->GetHeight()) {
436                 // total width
437                 hlPos = hlBase;
438                 for (int j=0; j < hlBand->GetWidth(); j++) {
439                         DequantizeValue(hlBand, hlPos, quantParam);
440                         hlPos++;
441                 }
442         }
443 }
```

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 461 of file Decoder.cpp.

```
461                                                                     {
462         ASSERT(m_currentBlock);
463
464         if (m_currentBlock->IsCompletelyRead()) {
465                 // all data of current macro block has been read --> prepare next macro block
466                 GetNextMacroBlock();
467         }
468
469         band->SetData(bandPos, m_currentBlock->m_value[m_currentBlock->m_valuePos] <<
quantParam);
470         m_currentBlock->m_valuePos++;
471 }
```

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.

```
136 { return m_encodedHeaderLength; }
```

void CDecoder::GetNextMacroBlock ()

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

Definition at line 476 of file Decoder.cpp.

```
476                                 {
477         // current block has been read --> prepare next current block
478         m_macroBlocksAvailable--;
479
480         if (m_macroBlocksAvailable > 0) {
481                 m_currentBlock = m_macroBlocks[++m_currentBlockIndex];
482         } else {
483                 DecodeBuffer();
484         }
485         ASSERT(m_currentBlock);
486 }
```

CPGFStream* CDecoder::GetStream ()[inline]

Returns:

Stream

Definition at line 174 of file Decoder.h.

```
174 { 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 265 of file Decoder.cpp.

```
265
{
266         ASSERT(band);
267
268         const div_t ww = div(width, LinBlockSize);
269         const div_t hh = div(height, LinBlockSize);
270         const int ws = pitch - LinBlockSize;
271         const int wr = pitch - ww.rem;
272         int pos, base = startPos, base2;
273
274         // main height
275         for (int i=0; i < hh.quot; i++) {
276                 // main width
277                 base2 = base;
278                 for (int j=0; j < ww.quot; j++) {
279                         pos = base2;
280                         for (int y=0; y < LinBlockSize; y++) {
281                                 for (int x=0; x < LinBlockSize; x++) {
282                                         DequantizeValue(band, pos, quantParam);
283                                         pos++;
284                                 }
285                                 pos += ws;
286                         }
287                         base2 += LinBlockSize;
288                 }
289                 // rest of width
290                 pos = base2;
291                 for (int y=0; y < LinBlockSize; y++) {
292                         for (int x=0; x < ww.rem; x++) {
293                                 DequantizeValue(band, pos, quantParam);
294                                 pos++;
295                         }
296                         pos += wr;
297                         base += pitch;
298                 }
299         }
300         // main width
301         base2 = base;
302         for (int j=0; j < ww.quot; j++) {
303                 // rest of height
304                 pos = base2;
305                 for (int y=0; y < hh.rem; y++) {
306                         for (int x=0; x < LinBlockSize; x++) {
307                                 DequantizeValue(band, pos, quantParam);
308                                 pos++;
309                         }
310                         pos += ws;
311                 }
```

```
312                 base2 += LinBlockSize;
313         }
314         // rest of height
315         pos = base2;
316         for (int y=0; y < hh.rem; y++) {
317                 // rest of width
318                 for (int x=0; x < ww.rem; x++) {
319                         DequantizeValue(band, pos, quantParam);
320                         pos++;
321                 }
322                 pos += wr;
323         }
324 }
```

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 245 of file Decoder.cpp.

```
245                                                                     {
246         ASSERT(m_stream);
247
248         int count = len;
249         m_stream->Read(&count, target);
250
251         return count;
252 }
```

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

throws **IOException**

Definition at line 534 of file Decoder.cpp.

```
534                                                 {
535         ASSERT(block);
536
537         UINT16 wordLen;
538         ROIBlockHeader h(BufferSize);
539         int count, expected;
540
541 #ifdef TRACE
542         //UINT32 filePos = (UINT32)m_stream->GetPos();
543         //printf("DecodeBuffer: %d\n", filePos);
544 #endif
545
546         // read wordLen
547         count = expected = sizeof(UINT16);
548         m_stream->Read(&count, &wordLen);
549         if (count != expected) ReturnWithError(MissingData);
550         wordLen = __VAL(wordLen); // convert wordLen
551         if (wordLen > BufferSize) ReturnWithError(FormatCannotRead);
552
553 #ifdef __PGFROISUPPORT__
554         // read ROIBlockHeader
555         if (m_roi) {
556                 count = expected = sizeof(ROIBlockHeader);
557                 m_stream->Read(&count, &h.val);
558                 if (count != expected) ReturnWithError(MissingData);
559                 h.val = __VAL(h.val); // convert ROIBlockHeader
560         }
```

```
561 #endif
562         // save header
563         block->m_header = h;
564 
565         // read data
566         count = expected = wordLen*WordBytes;
567         m_stream->Read(&count, block->m_codeBuffer);
568         if (count != expected) ReturnWithError(MissingData);
569 
570 #ifdef PGF_USE_BIG_ENDIAN
571         // convert data
572         count /= WordBytes;
573         for (int i=0; i < count; i++) {
574                 block->m_codeBuffer[i] = __VAL(block->m_codeBuffer[i]);
575         }
576 #endif
577 
578 #ifdef __PGFROISUPPORT__
579         ASSERT(m_roi && h.rbh.bufferSize <= BufferSize || h.rbh.bufferSize == BufferSize);
580 #else
581         ASSERT(h.rbh.bufferSize == BufferSize);
582 #endif
583 }
```

void CDecoder::SetStreamPosToData ()[inline]

Resets stream position to beginning of data block.

Definition at line 144 of file Decoder.h.

```
144 { 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.

```
140 { 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 448 of file Decoder.cpp.

```
448                                     {
449         m_stream->SetPos(FSFromCurrent, offset);
450 }
```

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.

```
   71 : m_stream(stream)
   72 , m_bufferStartPos(0)
   73 , m_currLevelIndex(0)
   74 , m_nLevels(header.nLevels)
   75 , m_favorSpeed(false)
   76 , m_forceWriting(false)
   77 #ifdef __PGFROISUPPORT__
   78 , m_roi(false)
   79 #endif
   80 {
   81         ASSERT(m_stream);
   82 
   83         int count;
   84         m_lastMacroBlock = 0;
   85         m_levelLength = nullptr;
   86 
   87         // set number of threads
   88 #ifdef LIBPGF_USE_OPENMP
   89         m_macroBlockLen = omp_get_num_procs();
   90 #else
   91         m_macroBlockLen = 1;
   92 #endif
   93 
   94         if (useOMP && m_macroBlockLen > 1) {
   95 #ifdef LIBPGF_USE_OPENMP
   96                 omp_set_num_threads(m_macroBlockLen);
   97 #endif
   98                 // create macro block array
   99                 m_macroBlocks = new(std::nothrow) CMacroBlock*[m_macroBlockLen];
  100                 if (!m_macroBlocks) ReturnWithError(InsufficientMemory);
  101                 for (int i=0; i < m_macroBlockLen; i++) m_macroBlocks[i] = new
CMacroBlock(this);
  102                 m_currentBlock = m_macroBlocks[m_lastMacroBlock++];
  103         } else {
  104                 m_macroBlocks = 0;
  105                 m_macroBlockLen = 1;
  106                 m_currentBlock = new CMacroBlock(this);
  107         }
  108 
  109         // save file position
  110         m_startPosition = m_stream->GetPos();
  111 
  112         // write preHeader
  113         preHeader.hSize = __VAL(preHeader.hSize);
  114         count = PreHeaderSize;
  115         m_stream->Write(&count, &preHeader);
  116 
  117         // write file header
  118         header.height = __VAL(header.height);
  119         header.width = __VAL(header.width);
  120         count = HeaderSize;
  121         m_stream->Write(&count, &header);
  122 
  123         // write postHeader
  124         if (header.mode == ImageModeIndexedColor) {
  125                 // write color table
  126                 count = ColorTableSize;
  127                 m_stream->Write(&count, (void *)postHeader.clut);
  128         }
  129         // save user data file position
  130         userDataPos = m_stream->GetPos();
  131         if (postHeader.userDataLen) {
  132                 if (postHeader.userData) {
  133                         // write user data
  134                         count = postHeader.userDataLen;
  135                         m_stream->Write(&count, postHeader.userData);
  136                 } else {
  137                         m_stream->SetPos(FSFromCurrent, count);
  138                 }
  139         }
  140 
```

```
141         // save level length file position
142         m_levelLengthPos = m_stream->GetPos();
143 }
```

CEncoder::~CEncoder ()

Destructor.

Definition at line 147 of file Encoder.cpp.

```
147                     {
148         if (m_macroBlocks) {
149                 for (int i=0; i < m_macroBlockLen; i++) delete m_macroBlocks[i];
150                 delete[] m_macroBlocks;
151         } else {
152                 delete m_currentBlock;
153         }
154 }
```

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.

```
179 { 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.

```
174 { 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.

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

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

Definition at line 341 of file Encoder.cpp.

```
341                                             {
342         ASSERT(m_currentBlock);
343 #ifdef __PGFROISUPPORT__
344         ASSERT(m_roi && h.rbh.bufferSize <= BufferSize || h.rbh.bufferSize == BufferSize);
345 #else
346         ASSERT(h.rbh.bufferSize == BufferSize);
347 #endif
348         m_currentBlock->m_header = h;
```

```
  349
  350         // macro block management
  351         if (m_macroBlockLen == 1) {
  352                 m_currentBlock->BitplaneEncode();
  353                 WriteMacroBlock(m_currentBlock);
  354         } else {
  355                 // save last level index
  356                 int lastLevelIndex = m_currentBlock->m_lastLevelIndex;
  357
  358                 if (m_forceWriting || m_lastMacroBlock == m_macroBlockLen) {
  359                         // encode macro blocks
  360                         /*
  361                         volatile OSError error = NoError;
  362                         #ifdef LIBPGF_USE_OPENMP
  363                         #pragma omp parallel for ordered default(shared)
  364                         #endif
  365                         for (int i=0; i < m_lastMacroBlock; i++) {
  366                                 if (error == NoError) {
  367                                         m_macroBlocks[i]->BitplaneEncode();
  368                                         #ifdef LIBPGF_USE_OPENMP
  369                                         #pragma omp ordered
  370                                         #endif
  371                                         {
  372                                                 try {
  373
WriteMacroBlock(m_macroBlocks[i]);
  374                                                 } catch (IOException& e) {
  375                                                         error = e.error;
  376                                                 }
  377                                                 delete m_macroBlocks[i];
m_macroBlocks[i] = 0;
  378                                         }
  379                                 }
  380                         }
  381                         if (error != NoError) ReturnWithError(error);
  382                         */
  383 #ifdef LIBPGF_USE_OPENMP
  384                         #pragma omp parallel for default(shared) //no declared exceptions
in next block
  385 #endif
  386                         for (int i=0; i < m_lastMacroBlock; i++) {
  387                                 m_macroBlocks[i]->BitplaneEncode();
  388                         }
  389                         for (int i=0; i < m_lastMacroBlock; i++) {
  390                                 WriteMacroBlock(m_macroBlocks[i]);
  391                         }
  392
  393                         // prepare for next round
  394                         m_forceWriting = false;
  395                         m_lastMacroBlock = 0;
  396                 }
  397                 // re-initialize macro block
  398                 m_currentBlock = m_macroBlocks[m_lastMacroBlock++];
  399                 m_currentBlock->Init(lastLevelIndex);
  400         }
  401 }
```

void CEncoder::FavorSpeedOverSize ()[inline]

Encoder favors speed over compression size.

Definition at line 121 of file Encoder.h.

```
  121 { 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.

```
 310                     {
 311         if (m_currentBlock->m_valuePos > 0) {
 312                 // pad buffer with zeros
 313                 memset(&(m_currentBlock->m_value[m_currentBlock->m_valuePos]), 0,
(BufferSize - m_currentBlock->m_valuePos)*DataTSize);
 314                 m_currentBlock->m_valuePos = BufferSize;
 315
 316                 // encode buffer
 317                 m_forceWriting = true;  // makes sure that the following EncodeBuffer is
really written into the stream
 318                 EncodeBuffer(ROIBlockHeader(m_currentBlock->m_valuePos, true));
 319         }
 320 }
```

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.

```
 246
{
 247         ASSERT(band);
 248
 249         const div_t hh = div(height, LinBlockSize);
 250         const div_t ww = div(width, LinBlockSize);
 251         const int ws = pitch - LinBlockSize;
 252         const int wr = pitch - ww.rem;
 253         int pos, base = startPos, base2;
 254
 255         // main height
 256         for (int i=0; i < hh.quot; i++) {
 257                 // main width
 258                 base2 = base;
 259                 for (int j=0; j < ww.quot; j++) {
 260                         pos = base2;
 261                         for (int y=0; y < LinBlockSize; y++) {
 262                                 for (int x=0; x < LinBlockSize; x++) {
 263                                         WriteValue(band, pos);
 264                                         pos++;
 265                                 }
 266                                 pos += ws;
 267                         }
 268                         base2 += LinBlockSize;
 269                 }
 270                 // rest of width
 271                 pos = base2;
 272                 for (int y=0; y < LinBlockSize; y++) {
 273                         for (int x=0; x < ww.rem; x++) {
 274                                 WriteValue(band, pos);
 275                                 pos++;
 276                         }
 277                         pos += wr;
 278                         base += pitch;
 279                 }
 280         }
 281         // main width
 282         base2 = base;
```

```
283         for (int j=0; j < ww.quot; j++) {
284                 // rest of height
285                 pos = base2;
286                 for (int y=0; y < hh.rem; y++) {
287                         for (int x=0; x < LinBlockSize; x++) {
288                                 WriteValue(band, pos);
289                                 pos++;
290                         }
291                         pos += ws;
292                 }
293                 base2 += LinBlockSize;
294         }
295         // rest of height
296         pos = base2;
297         for (int y=0; y < hh.rem; y++) {
298                 // rest of width
299                 for (int x=0; x < ww.rem; x++) {
300                         WriteValue(band, pos);
301                         pos++;
302                 }
303                 pos += wr;
304         }
305 }
```

void CEncoder::SetBufferStartPos ()[inline]

Save current stream position as beginning of current level.

Definition at line 192 of file Encoder.h.

```
192 { 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.

```
162 { 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.

```
188 { 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.

```
202                                     {
203         UINT64 curPos = m_stream->GetPos(); // end of image
204
205         // set file pos to levelLength
206         m_stream->SetPos(FSFromStart, m_levelLengthPos);
207
208         if (m_levelLength) {
209         #ifdef PGF_USE_BIG_ENDIAN
```

```
210                 UINT32 levelLength;
211                 int count = WordBytes;
212
213                 for (int i=0; i < m_currLevelIndex; i++) {
214                         levelLength = __VAL(UINT32(m_levelLength[i]));
215                         m_stream->Write(&count, &levelLength);
216                 }
217         #else
218                 int count = m_currLevelIndex*WordBytes;
219
220                 m_stream->Write(&count, m_levelLength);
221         #endif //PGF_USE_BIG_ENDIAN
222         } else {
223                 int count = m_currLevelIndex*WordBytes;
224                 m_stream->SetPos(FSFromCurrent, count);
225         }
226
227         // begin of image
228         UINT32 retValue = UINT32(curPos - m_stream->GetPos());
229
230         // restore file position
231         m_stream->SetPos(FSFromStart, curPos);
232
233         return retValue;
234 }
```

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.

```
160                                                             {
161         UINT64 curPos = m_stream->GetPos(); // end of user data
162         int count = PreHeaderSize;
163
164         // write preHeader
165         SetStreamPosToStart();
166         preHeader.hSize = __VAL(preHeader.hSize);
167         m_stream->Write(&count, &preHeader);
168
169         m_stream->SetPos(FSFromStart, curPos);
170 }
```

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.

```
177                                                     {
178         // renew levelLength
179         delete[] levelLength;
180         levelLength = new(std::nothrow) UINT32[m_nLevels];
181         if (!levelLength) ReturnWithError(InsufficientMemory);
182         for (UINT8 l = 0; l < m_nLevels; l++) levelLength[l] = 0;
183         m_levelLength = levelLength;
184
185         // save level length file position
```

```
  186         m_levelLengthPos = m_stream->GetPos();
  187
  188         // write dummy levelLength
  189         int count = m_nLevels*WordBytes;
  190         m_stream->Write(&count, m_levelLength);
  191
  192         // save current file position
  193         SetBufferStartPos();
  194
  195         return count;
  196 }
```

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

Definition at line 406 of file Encoder.cpp.

```
  406                                                 {
  407         ASSERT(block);
  408 #ifdef __PGFROISUPPORT__
  409         ROIBlockHeader h = block->m_header;
  410 #endif
  411         UINT16 wordLen = UINT16(NumberOfWords(block->m_codePos)); ASSERT(wordLen <=
CodeBufferLen);
  412         int count = sizeof(UINT16);
  413
  414 #ifdef TRACE
  415         //UINT32 filePos = (UINT32)m_stream->GetPos();
  416         //printf("EncodeBuffer: %d\n", filePos);
  417 #endif
  418
  419 #ifdef PGF_USE_BIG_ENDIAN
  420         // write wordLen
  421         UINT16 wl = __VAL(wordLen);
  422         m_stream->Write(&count, &wl); ASSERT(count == sizeof(UINT16));
  423
  424 #ifdef __PGFROISUPPORT__
  425         // write ROIBlockHeader
  426         if (m_roi) {
  427                 count = sizeof(ROIBlockHeader);
  428                 h.val = __VAL(h.val);
  429                 m_stream->Write(&count, &h.val); ASSERT(count == sizeof(ROIBlockHeader));
  430         }
  431 #endif // __PGFROISUPPORT__
  432
  433         // convert data
  434         for (int i=0; i < wordLen; i++) {
  435                 block->m_codeBuffer[i] = __VAL(block->m_codeBuffer[i]);
  436         }
  437 #else
  438         // write wordLen
  439         m_stream->Write(&count, &wordLen); ASSERT(count == sizeof(UINT16));
  440
  441 #ifdef __PGFROISUPPORT__
  442         // write ROIBlockHeader
  443         if (m_roi) {
  444                 count = sizeof(ROIBlockHeader);
  445                 m_stream->Write(&count, &h.val); ASSERT(count == sizeof(ROIBlockHeader));
  446         }
  447 #endif // __PGFROISUPPORT__
  448 #endif // PGF_USE_BIG_ENDIAN
  449
  450         // write encoded data into stream
  451         count = wordLen*WordBytes;
  452         m_stream->Write(&count, block->m_codeBuffer);
  453
  454         // store levelLength
  455         if (m_levelLength) {
  456                 // store level length
  457                 // EncodeBuffer has been called after m_lastLevelIndex has been updated
```

```
458                 ASSERT(m_currLevelIndex < m_nLevels);
459                 m_levelLength[m_currLevelIndex] += (UINT32)ComputeBufferLength();
460                 m_currLevelIndex = block->m_lastLevelIndex + 1;
461
462         }
463
464         // prepare for next buffer
465         SetBufferStartPos();
466
467         // reset values
468         block->m_valuePos = 0;
469         block->m_maxAbsValue = 0;
470 }
```

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.

```
326                                                     {
327         if (m_currentBlock->m_valuePos == BufferSize) {
328                 EncodeBuffer(ROIBlockHeader(BufferSize, false));
329         }
330         DataT val = m_currentBlock->m_value[m_currentBlock->m_valuePos++] =
band->GetData(bandPos);
331         UINT32 v = abs(val);
332         if (v > m_currentBlock->m_maxAbsValue) m_currentBlock->m_maxAbsValue = v;
333 }
```

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**

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.

```
 57                             : 4351 )
 58                 : m_value()
 59                 , m_codeBuffer()
 60                 , m_header(0)
 61                 , m_encoder(encoder)
 62                 , m_sigFlagVector()
 63                 {
 64                         ASSERT(m_encoder);
 65                         Init(-1);
 66                 }
```

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.

```
482                                             {
483         UINT8   nPlanes;
484         UINT32  sigLen, codeLen = 0, wordPos, refLen, signLen;
485         UINT32  sigBits[BufferLen] = { 0 };
486         UINT32  refBits[BufferLen] = { 0 };
487         UINT32  signBits[BufferLen] = { 0 };
488         UINT32  planeMask;
489         UINT32  bufferSize = m_header.rbh.bufferSize; ASSERT(bufferSize <= BufferSize);
490         bool    useRL;
491
492 #ifdef TRACE
493         //printf("which thread: %d\n", omp_get_thread_num());
494 #endif
495
496         // clear significance vector
497         for (UINT32 k=0; k < bufferSize; k++) {
498                 m_sigFlagVector[k] = false;
499         }
500         m_sigFlagVector[bufferSize] = true; // sentinel
501
502         // clear output buffer
503         for (UINT32 k=0; k < bufferSize; k++) {
504                 m_codeBuffer[k] = 0;
505         }
506         m_codePos = 0;
507
508         // compute number of bit planes and split buffer into separate bit planes
509         nPlanes = NumberOfBitplanes();
510
511         // write number of bit planes to m_codeBuffer
512         // <nPlanes>
513         SetValueBlock(m_codeBuffer, 0, nPlanes, MaxBitPlanesLog);
514         m_codePos += MaxBitPlanesLog;
515
516         // loop through all bit planes
517         if (nPlanes == 0) nPlanes = MaxBitPlanes + 1;
518         planeMask = 1 << (nPlanes - 1);
```

```
  519
  520         for (int plane = nPlanes - 1; plane >= 0; plane--) {
  521                 // clear significant bitset
  522                 for (UINT32 k=0; k < BufferLen; k++) {
  523                         sigBits[k] = 0;
  524                 }
  525
  526                 // split bitplane in significant bitset and refinement bitset
  527                 sigLen = DecomposeBitplane(bufferSize, planeMask, m_codePos +
RLblockSizeLen + 1, sigBits, refBits, signBits, signLen, codeLen);
  528
  529                 if (sigLen > 0 && codeLen <= MaxCodeLen && codeLen < AlignWordPos(sigLen)
+ AlignWordPos(signLen) + 2*RLblockSizeLen) {
  530                         // set RL code bit
  531                         // <1><codeLen>
  532                         SetBit(m_codeBuffer, m_codePos++);
  533
  534                         // write length codeLen to m_codeBuffer
  535                         SetValueBlock(m_codeBuffer, m_codePos, codeLen, RLblockSizeLen);
  536                         m_codePos += RLblockSizeLen + codeLen;
  537                 } else {
  538                 #ifdef TRACE
  539                         //printf("new\n");
  540                         //for (UINT32 i=0; i < bufferSize; i++) {
  541                         //      printf("%s", (GetBit(sigBits, i)) ? "1" : "_");
  542                         //      if (i%120 == 119) printf("\n");
  543                         //}
  544                         //printf("\n");
  545                 #endif // TRACE
  546
  547                         // run-length coding wasn't efficient enough
  548                         // we don't use RL coding for sigBits
  549                         // <0><sigLen>
  550                         ClearBit(m_codeBuffer, m_codePos++);
  551
  552                         // write length sigLen to m_codeBuffer
  553                         ASSERT(sigLen <= MaxCodeLen);
  554                         SetValueBlock(m_codeBuffer, m_codePos, sigLen, RLblockSizeLen);
  555                         m_codePos += RLblockSizeLen;
  556
  557                         if (m_encoder->m_favorSpeed || signLen == 0) {
  558                                 useRL = false;
  559                         } else {
  560                                 // overwrite m_codeBuffer
  561                                 useRL = true;
  562                                 // run-length encode m_sign and append them to the
m_codeBuffer
  563                                 codeLen = RLESigns(m_codePos + RLblockSizeLen + 1,
signBits, signLen);
  564                         }
  565
  566                         if (useRL && codeLen <= MaxCodeLen && codeLen < signLen) {
  567                                 // RL encoding of m_sign was efficient
  568                                 // <1><codeLen><codedSignBits>_
  569                                 // write RL code bit
  570                                 SetBit(m_codeBuffer, m_codePos++);
  571
  572                                 // write codeLen to m_codeBuffer
  573                                 SetValueBlock(m_codeBuffer, m_codePos, codeLen,
RLblockSizeLen);
  574
  575                                 // compute position of sigBits
  576                                 wordPos = NumberOfWords(m_codePos + RLblockSizeLen +
codeLen);
  577                                 ASSERT(0 <= wordPos && wordPos < CodeBufferLen);
  578                         } else {
  579                                 // RL encoding of signBits wasn't efficient
  580                                 // <0><signLen>_<signBits>_
  581                                 // clear RL code bit
  582                                 ClearBit(m_codeBuffer, m_codePos++);
  583
```

```
  584                                   // write signLen to m_codeBuffer
  585                                   ASSERT(signLen <= MaxCodeLen);
  586                                   SetValueBlock(m_codeBuffer, m_codePos, signLen,
RLblockSizeLen);
  587
  588                                   // write signBits to m_codeBuffer
  589                                   wordPos = NumberOfWords(m_codePos + RLblockSizeLen);
  590                                   ASSERT(0 <= wordPos && wordPos < CodeBufferLen);
  591                                   codeLen = NumberOfWords(signLen);
  592
  593                                   for (UINT32 k=0; k < codeLen; k++) {
  594                                           m_codeBuffer[wordPos++] = signBits[k];
  595                                   }
  596                           }
  597
  598                           // write sigBits
  599                           // <sigBits>_
  600                           ASSERT(0 <= wordPos && wordPos < CodeBufferLen);
  601                           refLen = NumberOfWords(sigLen);
  602
  603                           for (UINT32 k=0; k < refLen; k++) {
  604                                   m_codeBuffer[wordPos++] = sigBits[k];
  605                           }
  606                           m_codePos = wordPos << WordWidthLog;
  607                   }
  608
  609                   // append refinement bitset (aligned to word boundary)
  610                   // _<refBits>
  611                   wordPos = NumberOfWords(m_codePos);
  612                   ASSERT(0 <= wordPos && wordPos < CodeBufferLen);
  613                   refLen = NumberOfWords(bufferSize - sigLen);
  614
  615                   for (UINT32 k=0; k < refLen; k++) {
  616                           m_codeBuffer[wordPos++] = refBits[k];
  617                   }
  618                   m_codePos = wordPos << WordWidthLog;
  619                   planeMask >>= 1;
  620           }
  621           ASSERT(0 <= m_codePos && m_codePos <= CodeBufferBitLen);
  622 }
```

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.

```
  634
{
  635           ASSERT(sigBits);
  636           ASSERT(refBits);
  637           ASSERT(signBits);
  638           ASSERT(codePos < CodeBufferBitLen);
  639
  640           UINT32 sigPos = 0;
  641           UINT32 valuePos = 0, valueEnd;
  642           UINT32 refPos = 0;
  643
  644           // set output value
  645           signLen = 0;
  646
  647           // prepare RLE of Sigs and Signs
  648           const UINT32 outStartPos = codePos;
  649           UINT32 k = 3;
  650           UINT32 runlen = 1 << k; // = 2^k
  651           UINT32 count = 0;
  652
  653           while (valuePos < bufferSize) {
  654                   // search next 1 in m_sigFlagVector using searching with sentinel
```

```
655                 valueEnd = valuePos;
656                 while(!m_sigFlagVector[valueEnd]) { valueEnd++; }
657
658                 // search 1's in m_value[plane][valuePos..valueEnd)
659                 // these 1's are significant bits
660                 while (valuePos < valueEnd) {
661                         if (GetBitAtPos(valuePos, planeMask)) {
662                                 // RLE encoding
663                                 // encode run of count 0's followed by a 1
664                                 // with codeword: 1<count>(signBits[signPos])
665                                 SetBit(m_codeBuffer, codePos++);
666                                 if (k > 0) {
667                                         SetValueBlock(m_codeBuffer, codePos, count, k);
668                                         codePos += k;
669
670                                         // adapt k (half the zero run-length)
671                                         k--;
672                                         runlen >>= 1;
673                                 }
674
675                                 // copy and write sign bit
676                                 if (m_value[valuePos] < 0) {
677                                         SetBit(signBits, signLen++);
678                                         SetBit(m_codeBuffer, codePos++);
679                                 } else {
680                                         ClearBit(signBits, signLen++);
681                                         ClearBit(m_codeBuffer, codePos++);
682                                 }
683
684                                 // write a 1 to sigBits
685                                 SetBit(sigBits, sigPos++);
686
687                                 // update m_sigFlagVector
688                                 m_sigFlagVector[valuePos] = true;
689
690                                 // prepare for next run
691                                 count = 0;
692                         } else {
693                                 // RLE encoding
694                                 count++;
695                                 if (count == runlen) {
696                                         // encode run of 2^k zeros by a single 0
697                                         ClearBit(m_codeBuffer, codePos++);
698                                         // adapt k (double the zero run-length)
699                                         if (k < WordWidth) {
700                                                 k++;
701                                                 runlen <<= 1;
702                                         }
703
704                                         // prepare for next run
705                                         count = 0;
706                                 }
707
708                                 // write 0 to sigBits
709                                 sigPos++;
710                         }
711                         valuePos++;
712                 }
713                 // refinement bit
714                 if (valuePos < bufferSize) {
715                         // write one refinement bit
716                         if (GetBitAtPos(valuePos++, planeMask)) {
717                                 SetBit(refBits, refPos);
718                         } else {
719                                 ClearBit(refBits, refPos);
720                         }
721                         refPos++;
722                 }
723         }
724         // RLE encoding of the rest of the plane
725         // encode run of count 0's followed by a 1
```

```
726         // with codeword: 1<count>(signBits[signPos])
727         SetBit(m_codeBuffer, codePos++);
728         if (k > 0) {
729                 SetValueBlock(m_codeBuffer, codePos, count, k);
730                 codePos += k;
731         }
732         // write dmmy sign bit
733         SetBit(m_codeBuffer, codePos++);
734
735         // write word filler zeros
736
737         ASSERT(sigPos <= bufferSize);
738         ASSERT(refPos <= bufferSize);
739         ASSERT(signLen <= bufferSize);
740         ASSERT(valuePos == bufferSize);
741         ASSERT(codePos >= outStartPos && codePos < CodeBufferBitLen);
742         codeLen = codePos - outStartPos;
743
744         return sigPos;
745 }
```

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

Definition at line 96 of file Encoder.h.

```
96 { 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.

```
71                                                  {                           // initialize
for reusage
72                          m_valuePos = 0;
73                          m_maxAbsValue = 0;
74                          m_codePos = 0;
75                          m_lastLevelIndex = lastLevelIndex;
76                  }
```

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

Definition at line 750 of file Encoder.cpp.

```
750                                                 {
751         UINT8 cnt = 0;
752
753         // determine number of bitplanes for max value
754         if (m_maxAbsValue > 0) {
755                 while (m_maxAbsValue > 0) {
756                         m_maxAbsValue >>= 1; cnt++;
757                 }
758                 if (cnt == MaxBitPlanes + 1) cnt = 0;
759                 // end cs
760                 ASSERT(cnt <= MaxBitPlanes);
761                 ASSERT((cnt >> MaxBitPlanesLog) == 0);
762                 return cnt;
763         } else {
764                 return 1;
765         }
766 }
```

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

Definition at line 774 of file Encoder.cpp.

```
774                                                                                                 {
775         ASSERT(signBits);
776         ASSERT(0 <= codePos && codePos < CodeBufferBitLen);
777         ASSERT(0 < signLen && signLen <= BufferSize);
778 
779         const UINT32  outStartPos = codePos;
780         UINT32 k = 0;
781         UINT32 runlen = 1 << k; // = 2^k
782         UINT32 count = 0;
783         UINT32 signPos = 0;
784 
785         while (signPos < signLen) {
786                 // search next 0 in signBits starting at position signPos
787                 count = SeekBit1Range(signBits, signPos, __min(runlen, signLen - signPos));
788                 // count 1's found
789                 if (count == runlen) {
790                         // encode run of 2^k ones by a single 1
791                         signPos += count;
792                         SetBit(m_codeBuffer, codePos++);
793                         // adapt k (double the 1's run-length)
794                         if (k < WordWidth) {
795                                 k++;
796                                 runlen <<= 1;
797                         }
798                 } else {
799                         // encode run of count 1's followed by a 0
800                         // with codeword: 0(count)
801                         signPos += count + 1;
802                         ClearBit(m_codeBuffer, codePos++);
803                         if (k > 0) {
804                                 SetValueBlock(m_codeBuffer, codePos, count, k);
805                                 codePos += k;
806                         }
807                         // adapt k (half the 1's run-length)
808                         if (k > 0) {
809                                 k--;
810                                 runlen >>= 1;
811                         }
812                 }
813         }
814         ASSERT(signPos == signLen || signPos == signLen + 1);
815         ASSERT(codePos >= outStartPos && codePos < CodeBufferBitLen);
816         return codePos - outStartPos;
817 }
```

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**

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.

```
   56                 : m_header(0)
// makes sure that IsCompletelyRead() returns true for an empty macro block
   57 #pragma warning( suppress : 4351 )
   58                 , m_value()
   59                 , m_codeBuffer()
   60                 , m_valuePos(0)
   61                 , m_sigFlagVector()
   62                 {
   63                 }
```

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 649 of file Decoder.cpp.

```
  649                                         {
  650         UINT32 bufferSize = m_header.rbh.bufferSize; ASSERT(bufferSize <= BufferSize);
  651
  652         // clear significance vector
  653         for (UINT32 k=0; k < bufferSize; k++) {
  654                 m_sigFlagVector[k] = false;
  655         }
  656         m_sigFlagVector[bufferSize] = true; // sentinel
  657
  658         // clear output buffer
  659         for (UINT32 k=0; k < BufferSize; k++) {
  660                 m_value[k] = 0;
  661         }
  662
  663         // read number of bit planes
  664         // <nPlanes>
  665         UINT32 nPlanes = GetValueBlock(m_codeBuffer, 0, MaxBitPlanesLog);
  666         UINT32 codePos = MaxBitPlanesLog;
  667
  668         // loop through all bit planes
  669         if (nPlanes == 0) nPlanes = MaxBitPlanes + 1;
  670         ASSERT(0 < nPlanes && nPlanes <= MaxBitPlanes + 1);
  671         DataT planeMask = 1 << (nPlanes - 1);
  672
  673         for (int plane = nPlanes - 1; plane >= 0; plane--) {
  674                 UINT32 sigLen = 0;
  675
  676                 // read RL code
  677                 if (GetBit(m_codeBuffer, codePos)) {
  678                         // RL coding of sigBits is used
  679                         // <1><codeLen><codedSigAndSignBits>_<refBits>
  680                         codePos++;
  681
  682                         // read codeLen
  683                         UINT32 codeLen = GetValueBlock(m_codeBuffer, codePos,
RLblockSizeLen); ASSERT(codeLen <= MaxCodeLen);
  684
  685                         // position of encoded sigBits and signBits
  686                         UINT32 sigPos = codePos + RLblockSizeLen; ASSERT(sigPos <
CodeBufferBitLen);
  687
  688                         // refinement bits
  689                         codePos = AlignWordPos(sigPos + codeLen); ASSERT(codePos <
CodeBufferBitLen);
  690
```

```
  691                           // run-length decode significant bits and signs from m_codeBuffer
and
  692                           // read refinement bits from m_codeBuffer and compose bit plane
  693                           sigLen = ComposeBitplaneRLD(bufferSize, planeMask, sigPos,
&m_codeBuffer[codePos >> WordWidthLog]);
  694
  695                   } else {
  696                           // no RL coding is used for sigBits and signBits together
  697                           // <0><sigLen>
  698                           codePos++;
  699
  700                           // read sigLen
  701                           sigLen = GetValueBlock(m_codeBuffer, codePos, RLblockSizeLen);
ASSERT(sigLen <= MaxCodeLen);
  702                           codePos += RLblockSizeLen; ASSERT(codePos < CodeBufferBitLen);
  703
  704                           // read RL code for signBits
  705                           if (GetBit(m_codeBuffer, codePos)) {
  706                                   // RL coding is used just for signBits
  707                                   // <1><codeLen><codedSignBits>_<sigBits>_<refBits>
  708                                   codePos++;
  709
  710                                   // read codeLen
  711                                   UINT32 codeLen = GetValueBlock(m_codeBuffer, codePos,
RLblockSizeLen); ASSERT(codeLen <= MaxCodeLen);
  712
  713                                   // sign bits
  714                                   UINT32 signPos = codePos + RLblockSizeLen; ASSERT(signPos
< CodeBufferBitLen);
  715
  716                                   // significant bits
  717                                   UINT32 sigPos = AlignWordPos(signPos + codeLen);
ASSERT(sigPos < CodeBufferBitLen);
  718
  719                                   // refinement bits
  720                                   codePos = AlignWordPos(sigPos + sigLen); ASSERT(codePos <
CodeBufferBitLen);
  721
  722                                   // read significant and refinement bitset from m_codeBuffer
  723                                   sigLen = ComposeBitplaneRLD(bufferSize, planeMask,
&m_codeBuffer[sigPos >> WordWidthLog], &m_codeBuffer[codePos >> WordWidthLog], signPos);
  724
  725                           } else {
  726                                   // RL coding of signBits was not efficient and therefore
not used
  727                                   // <0><signLen>_<signBits>_<sigBits>_<refBits>
  728                                   codePos++;
  729
  730                                   // read signLen
  731                                   UINT32 signLen = GetValueBlock(m_codeBuffer, codePos,
RLblockSizeLen); ASSERT(signLen <= MaxCodeLen);
  732
  733                                   // sign bits
  734                                   UINT32 signPos = AlignWordPos(codePos + RLblockSizeLen);
ASSERT(signPos < CodeBufferBitLen);
  735
  736                                   // significant bits
  737                                   UINT32 sigPos = AlignWordPos(signPos + signLen);
ASSERT(sigPos < CodeBufferBitLen);
  738
  739                                   // refinement bits
  740                                   codePos = AlignWordPos(sigPos + sigLen); ASSERT(codePos <
CodeBufferBitLen);
  741
  742                                   // read significant and refinement bitset from m_codeBuffer
  743                                   sigLen = ComposeBitplane(bufferSize, planeMask,
&m_codeBuffer[sigPos >> WordWidthLog], &m_codeBuffer[codePos >> WordWidthLog],
&m_codeBuffer[signPos >> WordWidthLog]);
  744                           }
  745                   }
  746
```

```
  747                 // start of next chunk
  748                 codePos = AlignWordPos(codePos + bufferSize - sigLen); ASSERT(codePos <
CodeBufferBitLen);
  749
  750                 // next plane
  751                 planeMask >>= 1;
  752         }
  753
  754         m_valuePos = 0;
  755 }
```

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

Definition at line 762 of file Decoder.cpp.

```
  762
{
  763         ASSERT(sigBits);
  764         ASSERT(refBits);
  765         ASSERT(signBits);
  766
  767         UINT32 valPos = 0, signPos = 0, refPos = 0, sigPos = 0;
  768
  769         while (valPos < bufferSize) {
  770                 // search next 1 in m_sigFlagVector using searching with sentinel
  771                 UINT32 sigEnd = valPos;
  772                 while(!m_sigFlagVector[sigEnd]) { sigEnd++; }
  773                 sigEnd -= valPos;
  774                 sigEnd += sigPos;
  775
  776                 // search 1's in sigBits[sigPos..sigEnd)
  777                 // these 1's are significant bits
  778                 while (sigPos < sigEnd) {
  779                         // search 0's
  780                         UINT32 zerocnt = SeekBitRange(sigBits, sigPos, sigEnd - sigPos);
  781                         sigPos += zerocnt;
  782                         valPos += zerocnt;
  783                         if (sigPos < sigEnd) {
  784                                 // write bit to m_value
  785                                 SetBitAtPos(valPos, planeMask);
  786
  787                                 // copy sign bit
  788                                 SetSign(valPos, GetBit(signBits, signPos++));
  789
  790                                 // update significance flag vector
  791                                 m_sigFlagVector[valPos++] = true;
  792                                 sigPos++;
  793                         }
  794                 }
  795                 // refinement bit
  796                 if (valPos < bufferSize) {
  797                         // write one refinement bit
  798                         if (GetBit(refBits, refPos)) {
  799                                 SetBitAtPos(valPos, planeMask);
  800                         }
  801                         refPos++;
  802                         valPos++;
  803                 }
  804         }
  805         ASSERT(sigPos <= bufferSize);
  806         ASSERT(refPos <= bufferSize);
  807         ASSERT(signPos <= bufferSize);
  808         ASSERT(valPos == bufferSize);
  809
  810         return sigPos;
  811 }
```

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

Definition at line 823 of file Decoder.cpp.

```
  823
{
  824         ASSERT(refBits);
  825
  826         UINT32 valPos = 0, refPos = 0;
  827         UINT32 sigPos = 0, sigEnd;
  828         UINT32 k = 3;
  829         UINT32 runlen = 1 << k; // = 2^k
  830         UINT32 count = 0, rest = 0;
  831         bool set1 = false;
  832
  833         while (valPos < bufferSize) {
  834                 // search next 1 in m_sigFlagVector using searching with sentinel
  835                 sigEnd = valPos;
  836                 while(!m_sigFlagVector[sigEnd]) { sigEnd++; }
  837                 sigEnd -= valPos;
  838                 sigEnd += sigPos;
  839
  840                 while (sigPos < sigEnd) {
  841                         if (rest || set1) {
  842                                 // rest of last run
  843                                 sigPos += rest;
  844                                 valPos += rest;
  845                                 rest = 0;
  846                         } else {
  847                                 // decode significant bits
  848                                 if (GetBit(m_codeBuffer, codePos++)) {
  849                                         // extract counter and generate zero run of length
count
  850                                         if (k > 0) {
  851                                                 // extract counter
  852                                                 count = GetValueBlock(m_codeBuffer,
codePos, k);
  853                                                 codePos += k;
  854                                                 if (count > 0) {
  855                                                         sigPos += count;
  856                                                         valPos += count;
  857                                                 }
  858
  859                                                 // adapt k (half run-length interval)
  860                                                 k--;
  861                                                 runlen >>= 1;
  862                                         }
  863
  864                                         set1 = true;
  865
  866                                 } else {
  867                                         // generate zero run of length 2^k
  868                                         sigPos += runlen;
  869                                         valPos += runlen;
  870
  871                                         // adapt k (double run-length interval)
  872                                         if (k < WordWidth) {
  873                                                 k++;
  874                                                 runlen <<= 1;
  875                                         }
  876                                 }
  877                         }
  878
  879                         if (sigPos < sigEnd) {
  880                                 if (set1) {
  881                                         set1 = false;
  882
  883                                         // write 1 bit
  884                                         SetBitAtPos(valPos, planeMask);
```

```
  885
  886                                         // set sign bit
  887                                         SetSign(valPos, GetBit(m_codeBuffer,
codePos++));
  888
  889                                         // update significance flag vector
  890                                         m_sigFlagVector[valPos++] = true;
  891                                         sigPos++;
  892                                 }
  893                         } else {
  894                                 rest = sigPos - sigEnd;
  895                                 sigPos = sigEnd;
  896                                 valPos -= rest;
  897                         }
  898
  899                 }
  900
  901                 // refinement bit
  902                 if (valPos < bufferSize) {
  903                         // write one refinement bit
  904                         if (GetBit(refBits, refPos)) {
  905                                 SetBitAtPos(valPos, planeMask);
  906                         }
  907                         refPos++;
  908                         valPos++;
  909                 }
  910         }
  911         ASSERT(sigPos <= bufferSize);
  912         ASSERT(refPos <= bufferSize);
  913         ASSERT(valPos == bufferSize);
  914
  915         return sigPos;
  916 }
```

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

Definition at line 926 of file Decoder.cpp.

```
  926
{
  927         ASSERT(sigBits);
  928         ASSERT(refBits);
  929
  930         UINT32 valPos = 0, refPos = 0;
  931         UINT32 sigPos = 0, sigEnd;
  932         UINT32 zerocnt, count = 0;
  933         UINT32 k = 0;
  934         UINT32 runlen = 1 << k; // = 2^k
  935         bool signBit = false;
  936         bool zeroAfterRun = false;
  937
  938         while (valPos < bufferSize) {
  939                 // search next 1 in m_sigFlagVector using searching with sentinel
  940                 sigEnd = valPos;
  941                 while(!m_sigFlagVector[sigEnd]) { sigEnd++; }
  942                 sigEnd -= valPos;
  943                 sigEnd += sigPos;
  944
  945                 // search 1's in sigBits[sigPos..sigEnd)
  946                 // these 1's are significant bits
  947                 while (sigPos < sigEnd) {
  948                         // search 0's
  949                         zerocnt = SeekBitRange(sigBits, sigPos, sigEnd - sigPos);
  950                         sigPos += zerocnt;
  951                         valPos += zerocnt;
  952                         if (sigPos < sigEnd) {
  953                                 // write bit to m_value
  954                                 SetBitAtPos(valPos, planeMask);
```

```
 955
 956                                // check sign bit
 957                                if (count == 0) {
 958                                        // all 1's have been set
 959                                        if (zeroAfterRun) {
 960                                                // finish the run with a 0
 961                                                signBit = false;
 962                                                zeroAfterRun = false;
 963                                        } else {
 964                                                // decode next sign bit
 965                                                if (GetBit(m_codeBuffer, signPos++)) {
 966                                                        // generate 1's run of length 2^k
 967                                                        count = runlen - 1;
 968                                                        signBit = true;
 969
 970                                                        // adapt k (double run-length
interval)
 971                                                        if (k < WordWidth) {
 972                                                                k++;
 973                                                                runlen <<= 1;
 974                                                        }
 975                                                } else {
 976                                                        // extract counter and generate
1's run of length count
 977                                                        if (k > 0) {
 978                                                                // extract counter
 979                                                                count =
GetValueBlock(m_codeBuffer, signPos, k);
 980                                                                signPos += k;
 981
 982                                                                // adapt k (half
run-length interval)
 983                                                                k--;
 984                                                                runlen >>= 1;
 985                                                        }
 986                                                        if (count > 0) {
 987                                                                count--;
 988                                                                signBit = true;
 989                                                                zeroAfterRun = true;
 990                                                        } else {
 991                                                                signBit = false;
 992                                                        }
 993                                                }
 994                                        }
 995                                } else {
 996                                        ASSERT(count > 0);
 997                                        ASSERT(signBit);
 998                                        count--;
 999                                }
1000
1001                                // copy sign bit
1002                                SetSign(valPos, signBit);
1003
1004                                // update significance flag vector
1005                                m_sigFlagVector[valPos++] = true;
1006                                sigPos++;
1007                        }
1008                }
1009
1010                // refinement bit
1011                if (valPos < bufferSize) {
1012                        // write one refinement bit
1013                        if (GetBit(refBits, refPos)) {
1014                                SetBitAtPos(valPos, planeMask);
1015                        }
1016                        refPos++;
1017                        valPos++;
1018                }
1019        }
1020        ASSERT(sigPos <= bufferSize);
1021        ASSERT(refPos <= bufferSize);
```

```
1022         ASSERT(valPos == bufferSize);
1023
1024         return sigPos;
1025 }
```

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.

```
68 { 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.

```
85 { (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.

```
86 { 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:

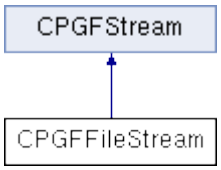
- **Decoder.h**
- **Decoder.cpp**

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.

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

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

Constructor

Parameters:

hFile	File handle

Definition at line 90 of file PGFstream.h.

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

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

Definition at line 94 of file PGFstream.h.

```
94 { m_hFile = 0; }
```

Member Function Documentation

HANDLE CPGFFileStream::GetHandle ()[inline]

Returns:

File handle

Definition at line 92 of file PGFstream.h.

```
92 { return m_hFile; }
```

UINT64 CPGFFileStream::GetPos () const[virtual]

Get current stream position.

Returns:

Current stream position

Implements **CPGFStream** (*p.112*).

Definition at line 64 of file PGFstream.cpp.

```
64                                      {
65          ASSERT(IsValid());
66          OSError err;
67          UINT64 pos = 0;
68          if ((err = GetFPos(m_hFile, &pos)) != NoError) ReturnWithError2(err, pos);
69          return pos;
70 }
```

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

Check stream validity.

Returns:

True if stream and current position is valid

Implements **CPGFStream** (*p.112*).

Definition at line 99 of file PGFstream.h.

```
99 { 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.112*).

Definition at line 48 of file PGFstream.cpp.

```
48                                                      {
49          ASSERT(count);
50          ASSERT(buffPtr);
51          ASSERT(IsValid());
52          OSError err;
53          if ((err = FileRead(m_hFile, count, buffPtr)) != NoError) ReturnWithError(err);
54 }
```

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.112*).

Definition at line 57 of file PGFstream.cpp.

```
57                                                          {
58          ASSERT(IsValid());
59          OSError err;
60          if ((err = SetFPos(m_hFile, posMode, posOff)) != NoError) ReturnWithError(err);
61 }
```

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.112*).

Definition at line 38 of file PGFstream.cpp.

```
38                                                      {
39          ASSERT(count);
40          ASSERT(buffPtr);
41          ASSERT(IsValid());
42          OSError err;
43          if ((err = FileWrite(m_hFile, count, buffPtr)) != NoError) ReturnWithError(err);
44
45 }
```

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** ()
- void **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.

```
64                          {
65          Init();
66 }
```

CPGFImage::~CPGFImage ()[virtual]

Destructor.

Definition at line 117 of file PGFimage.cpp.

```
117                          {
118         m_currentLevel = -100; // unusual value used as marker in Destroy()
119         Destroy();
120 }
```

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.

```
461 { 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.

```
406 { 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.

```
401 { 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.

```
448 { 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.

```
394 { 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.

```
573                                         {
574                 if (v & 0xFFFF0000) return (v < 0) ? (UINT16)0: (UINT16)65535; else return
(UINT16)v;
575         }
```

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

Definition at line 576 of file PGFimage.h.

```
576                                         {
577                 return (v < 0) ? 0 : (UINT32)v;
578         }
```

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

Definition at line 563 of file PGFimage.h.

```
563                                       {
564                 if (v & 0xFFFFFFF0) return (v < 0) ? (UINT8)0: (UINT8)15; else return
(UINT8)v;
565         }
```

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

Definition at line 566 of file PGFimage.h.

```
  566                                     {
  567                 if (v & 0xFFFFFFC0) return (v < 0) ? (UINT16)0: (UINT16)63; else return
(UINT16)v;
  568         }
```

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

Definition at line 569 of file PGFimage.h.

```
  569                                     {
  570                 // needs only one test in the normal case
  571                 if (v & 0xFFFFFF00) return (v < 0) ? (UINT8)0 : (UINT8)255; else return
(UINT8)v;
  572         }
```

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 766 of file PGFimage.cpp.

```
  766                                                 {
  767         if (version & Version7) return 7;
  768         if (version & Version6) return 6;
  769         if (version & Version5) return 5;
  770         if (version & Version2) return 2;
  771         return 1;
  772 }
```

void CPGFImage::CompleteHeader ()[private]

Definition at line 218 of file PGFimage.cpp.

```
  218                                 {
  219         // set current codec version
  220         m_header.version = PGFVersionNumber(PGFMajorNumber, PGFYear, PGFWeek);
  221
  222         if (m_header.mode == ImageModeUnknown) {
  223                 // undefined mode
  224                 switch(m_header.bpp) {
  225                 case 1: m_header.mode = ImageModeBitmap; break;
  226                 case 8: m_header.mode = ImageModeGrayScale; break;
  227                 case 12: m_header.mode = ImageModeRGB12; break;
  228                 case 16: m_header.mode = ImageModeRGB16; break;
  229                 case 24: m_header.mode = ImageModeRGBColor; break;
  230                 case 32: m_header.mode = ImageModeRGBA; break;
  231                 case 48: m_header.mode = ImageModeRGB48; break;
  232                 default: m_header.mode = ImageModeRGBColor; break;
  233                 }
  234         }
  235         if (!m_header.bpp) {
  236                 // undefined bpp
  237                 switch(m_header.mode) {
```

```
238                 case ImageModeBitmap:
239                         m_header.bpp = 1;
240                         break;
241                 case ImageModeIndexedColor:
242                 case ImageModeGrayScale:
243                         m_header.bpp = 8;
244                         break;
245                 case ImageModeRGB12:
246                         m_header.bpp = 12;
247                         break;
248                 case ImageModeRGB16:
249                 case ImageModeGray16:
250                         m_header.bpp = 16;
251                         break;
252                 case ImageModeRGBColor:
253                 case ImageModeLabColor:
254                         m_header.bpp = 24;
255                         break;
256                 case ImageModeRGBA:
257                 case ImageModeCMYKColor:
258                 case ImageModeGray32:
259                         m_header.bpp = 32;
260                         break;
261                 case ImageModeRGB48:
262                 case ImageModeLab48:
263                         m_header.bpp = 48;
264                         break;
265                 case ImageModeCMYK64:
266                         m_header.bpp = 64;
267                         break;
268                 default:
269                         ASSERT(false);
270                         m_header.bpp = 24;
271                 }
272         }
273         if (m_header.mode == ImageModeRGBColor && m_header.bpp == 32) {
274                 // change mode
275                 m_header.mode = ImageModeRGBA;
276         }
277         ASSERT(m_header.mode != ImageModeBitmap || m_header.bpp == 1);
278         ASSERT(m_header.mode != ImageModeIndexedColor || m_header.bpp == 8);
279         ASSERT(m_header.mode != ImageModeGrayScale || m_header.bpp == 8);
280         ASSERT(m_header.mode != ImageModeGray16 || m_header.bpp == 16);
281         ASSERT(m_header.mode != ImageModeGray32 || m_header.bpp == 32);
282         ASSERT(m_header.mode != ImageModeRGBColor || m_header.bpp == 24);
283         ASSERT(m_header.mode != ImageModeRGBA || m_header.bpp == 32);
284         ASSERT(m_header.mode != ImageModeRGB12 || m_header.bpp == 12);
285         ASSERT(m_header.mode != ImageModeRGB16 || m_header.bpp == 16);
286         ASSERT(m_header.mode != ImageModeRGB48 || m_header.bpp == 48);
287         ASSERT(m_header.mode != ImageModeLabColor || m_header.bpp == 24);
288         ASSERT(m_header.mode != ImageModeLab48 || m_header.bpp == 48);
289         ASSERT(m_header.mode != ImageModeCMYKColor || m_header.bpp == 32);
290         ASSERT(m_header.mode != ImageModeCMYK64 || m_header.bpp == 64);
291
292         // set number of channels
293         if (!m_header.channels) {
294                 switch(m_header.mode) {
295                 case ImageModeBitmap:
296                 case ImageModeIndexedColor:
297                 case ImageModeGrayScale:
298                 case ImageModeGray16:
299                 case ImageModeGray32:
300                         m_header.channels = 1;
301                         break;
302                 case ImageModeRGBColor:
303                 case ImageModeRGB12:
304                 case ImageModeRGB16:
305                 case ImageModeRGB48:
306                 case ImageModeLabColor:
307                 case ImageModeLab48:
308                         m_header.channels = 3;
```

```
  309                         break;
  310                 case ImageModeRGBA:
  311                 case ImageModeCMYKColor:
  312                 case ImageModeCMYK64:
  313                         m_header.channels = 4;
  314                         break;
  315                 default:
  316                         ASSERT(false);
  317                         m_header.channels = 3;
  318                 }
  319         }
  320
  321         // store used bits per channel
  322         UINT8 bpc = m_header.bpp/m_header.channels;
  323         if (bpc > 31) bpc = 31;
  324         if (!m_header.usedBitsPerChannel || m_header.usedBitsPerChannel > bpc) {
  325                 m_header.usedBitsPerChannel = bpc;
  326         }
  327 }
```

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 852 of file PGFimage.cpp.

```
  852                                 {
  853         const int maxThumbnailWidth = 20*FilterSize;
  854         const int m = __min(m_header.width, m_header.height);
  855         int s = m;
  856
  857         if (m_header.nLevels < 1 || m_header.nLevels > MaxLevel) {
  858                 m_header.nLevels = 1;
  859                 // compute a good value depending on the size of the image
  860                 while (s > maxThumbnailWidth) {
  861                         m_header.nLevels++;
  862                         s >>= 1;
  863                 }
  864         }
  865
  866         int levels = m_header.nLevels; // we need a signed value during level reduction
  867
  868         // reduce number of levels if the image size is smaller than FilterSize*(2^levels)
  869         s = FilterSize*(1 << levels);   // must be at least the double filter size because
of subsampling
  870         while (m < s) {
  871                 levels--;
  872                 s >>= 1;
  873         }
  874         if (levels > MaxLevel) m_header.nLevels = MaxLevel;
  875         else if (levels < 0) m_header.nLevels = 0;
  876         else m_header.nLevels = (UINT8)levels;
  877
  878         // used in Write when PM_Absolute
  879         m_percent = pow(0.25, m_header.nLevels);
  880
  881         ASSERT(0 <= m_header.nLevels && m_header.nLevels <= MaxLevel);
  882 }
```

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.

```
  260 { 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.

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

void CPGFImage::Destroy ()

Definition at line 124 of file PGFimage.cpp.

```
  124                         {
  125         for (int i = 0; i < m_header.channels; i++) {
  126                 delete m_wtChannel[i]; // also deletes m_channel
  127         }
  128         delete[] m_postHeader.userData;
  129         delete[] m_levelLength;
  130         delete m_decoder;
  131         delete m_encoder;
  132
  133         if (m_currentLevel != -100) Init();
  134 }
```

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

Definition at line 808 of file PGFimage.cpp.

```
  808                                     {
  809         ASSERT(ch > 0);
  810
  811         const int w = m_width[0];
  812         const int w2 = w/2;
```

```
  813         const int h2 = m_height[0]/2;
  814         const int oddW = w%2;                           // don't use bool -> problems with
MaxSpeed optimization
  815         const int oddH = m_height[0]%2;         // "
  816         int loPos = 0;
  817         int hiPos = w;
  818         int sampledPos = 0;
  819         DataT* buff = m_channel[ch]; ASSERT(buff);
  820
  821         for (int i=0; i < h2; i++) {
  822                 for (int j=0; j < w2; j++) {
  823                         // compute average of pixel block
  824                         buff[sampledPos] = (buff[loPos] + buff[loPos + 1] + buff[hiPos] +
buff[hiPos + 1]) >> 2;
  825                         loPos += 2; hiPos += 2;
  826                         sampledPos++;
  827                 }
  828                 if (oddW) {
  829                         buff[sampledPos] = (buff[loPos] + buff[hiPos]) >> 1;
  830                         loPos++; hiPos++;
  831                         sampledPos++;
  832                 }
  833                 loPos += w; hiPos += w;
  834         }
  835         if (oddH) {
  836                 for (int j=0; j < w2; j++) {
  837                         buff[sampledPos] = (buff[loPos] + buff[loPos+1]) >> 1;
  838                         loPos += 2; hiPos += 2;
  839                         sampledPos++;
  840                 }
  841                 if (oddW) {
  842                         buff[sampledPos] = buff[loPos];
  843                 }
  844         }
  845
  846         // downsampled image has half width and half height
  847         m_width[ch] = (m_width[ch] + 1)/2;
  848         m_height[ch] = (m_height[ch] + 1)/2;
  849 }
```

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 1787 of file PGFimage.cpp.

```
 1787
{
 1788         ASSERT(buff);
 1789         UINT32 w = m_width[0];  // width of decoded image
 1790         UINT32 h = m_height[0]; // height of decoded image
 1791         UINT32 yw = w;                  // y-channel width
 1792         UINT32 uw = m_width[1]; // u-channel width
 1793         UINT32 roiOffsetX = 0;
 1794         UINT32 roiOffsetY = 0;
 1795         UINT32 yOffset = 0;
 1796         UINT32 uOffset = 0;
 1797
 1798 #ifdef __PGFROISUPPORT__
 1799         const PGFRect& roi = GetAlignedROI(); // in pixels, roi is usually larger than
levelRoi
 1800         ASSERT(w == roi.Width() && h == roi.Height());
 1801         const PGFRect levelRoi = ComputeLevelROI();
 1802         ASSERT(roi.left <= levelRoi.left && levelRoi.right <= roi.right);
 1803         ASSERT(roi.top <= levelRoi.top && levelRoi.bottom <= roi.bottom);
 1804
 1805         if (ROIisSupported() && (levelRoi.Width() < w || levelRoi.Height() < h)) {
 1806                 // ROI is used
 1807                 w = levelRoi.Width();
 1808                 h = levelRoi.Height();
 1809                 roiOffsetX = levelRoi.left - roi.left;
 1810                 roiOffsetY = levelRoi.top - roi.top;
 1811                 yOffset = roiOffsetX + roiOffsetY*yw;
 1812
 1813                 if (m_downsample) {
 1814                         const PGFRect& downsampledRoi = GetAlignedROI(1);
 1815                         uOffset = levelRoi.left/2 - downsampledRoi.left + (levelRoi.top/2
- downsampledRoi.top)*m_width[1];
 1816                 } else {
 1817                         uOffset = yOffset;
 1818                 }
 1819         }
 1820 #endif
 1821
 1822         const double dP = 1.0/h;
 1823         int defMap[] = { 0, 1, 2, 3, 4, 5, 6, 7 }; ASSERT(sizeof(defMap)/sizeof(defMap[0])
== MaxChannels);
 1824         if (channelMap == nullptr) channelMap = defMap;
 1825         DataT uAvg, vAvg;
 1826         double percent = 0;
 1827         UINT32 i, j;
 1828
 1829         switch(m_header.mode) {
 1830         case ImageModeBitmap:
 1831                 {
 1832                         ASSERT(m_header.channels == 1);
 1833                         ASSERT(m_header.bpp == 1);
 1834                         ASSERT(bpp == 1);
 1835
 1836                         const UINT32 w2 = (w + 7)/8;
 1837                         DataT* y = m_channel[0]; ASSERT(y);
 1838
 1839                         if (m_preHeader.version & Version7) {
 1840                                 // new unpacked version has a little better compression
ratio
 1841                                 // since version 7
 1842                                 for (i = 0; i < h; i++) {
 1843                                         UINT32 cnt = 0;
 1844                                         for (j = 0; j < w2; j++) {
 1845                                                 UINT8 byte = 0;
 1846                                                 for (int k = 0; k < 8; k++) {
 1847                                                         byte <<= 1;
 1848                                                         UINT8 bit = 0;
```

```
 1849                                                    if (cnt < w) {
 1850                                                            bit = y[yOffset + cnt] &
1;
 1851                                                    }
 1852                                                    byte |= bit;
 1853                                                    cnt++;
 1854                                            }
 1855                                            buff[j] = byte;
 1856                                    }
 1857                                    yOffset += yw;
 1858                                    buff += pitch;
 1859
 1860                                    if (cb) {
 1861                                            percent += dP;
 1862                                            if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
 1863                                    }
 1864                            }
 1865                    } else {
 1866                            // old versions
 1867                            // packed pixels: 8 pixel in 1 byte of channel[0]
 1868                            if (!(m_preHeader.version & Version5)) yw = w2; // not
version 5 or 6
 1869                            yOffset = roiOffsetX/8 + roiOffsetY*yw; // 1 byte in y
contains 8 pixel values
 1870                            for (i = 0; i < h; i++) {
 1871                                    for (j = 0; j < w2; j++) {
 1872                                            buff[j] = Clamp8(y[yOffset + j] +
YUVoffset8);
 1873                                    }
 1874                                    yOffset += yw;
 1875                                    buff += pitch;
 1876
 1877                                    if (cb) {
 1878                                            percent += dP;
 1879                                            if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
 1880                                    }
 1881                            }
 1882                    }
 1883                    break;
 1884            }
 1885        case ImageModeIndexedColor:
 1886        case ImageModeGrayScale:
 1887        case ImageModeHSLColor:
 1888        case ImageModeHSBColor:
 1889                {
 1890                        ASSERT(m_header.channels >= 1);
 1891                        ASSERT(m_header.bpp == m_header.channels*8);
 1892                        ASSERT(bpp%8 == 0);
 1893
 1894                        UINT32 cnt, channels = bpp/8; ASSERT(channels >=
m_header.channels);
 1895
 1896                        for (i=0; i < h; i++) {
 1897                                UINT32 yPos = yOffset;
 1898                                cnt = 0;
 1899                                for (j=0; j < w; j++) {
 1900                                        for (UINT32 c=0; c < m_header.channels; c++) {
 1901                                                buff[cnt + channelMap[c]] =
Clamp8(m_channel[c][yPos] + YUVoffset8);
 1902                                        }
 1903                                        cnt += channels;
 1904                                        yPos++;
 1905                                }
 1906                                yOffset += yw;
 1907                                buff += pitch;
 1908
 1909                                if (cb) {
 1910                                        percent += dP;
```

```
 1911                                            if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
 1912                                    }
 1913                            }
 1914                            break;
 1915                    }
 1916            case ImageModeGray16:
 1917                    {
 1918                            ASSERT(m_header.channels >= 1);
 1919                            ASSERT(m_header.bpp == m_header.channels*16);
 1920
 1921                            const DataT yuvOffset16 = 1 << (UsedBitsPerChannel() - 1);
 1922                            UINT32 cnt, channels;
 1923
 1924                            if (bpp%16 == 0) {
 1925                                    const int shift = 16 - UsedBitsPerChannel(); ASSERT(shift
>= 0);
 1926                                    UINT16 *buff16 = (UINT16 *)buff;
 1927                                    int pitch16 = pitch/2;
 1928                                    channels = bpp/16; ASSERT(channels >= m_header.channels);
 1929
 1930                                    for (i=0; i < h; i++) {
 1931                                            UINT32 yPos = yOffset;
 1932                                            cnt = 0;
 1933                                            for (j=0; j < w; j++) {
 1934                                                    for (UINT32 c=0; c < m_header.channels;
c++) {
 1935                                                            buff16[cnt + channelMap[c]] =
Clamp16((m_channel[c][yPos] + yuvOffset16) << shift);
 1936                                                    }
 1937                                                    cnt += channels;
 1938                                                    yPos++;
 1939                                            }
 1940                                            yOffset += yw;
 1941                                            buff16 += pitch16;
 1942
 1943                                            if (cb) {
 1944                                                    percent += dP;
 1945                                                    if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
 1946                                            }
 1947                                    }
 1948                            } else {
 1949                                    ASSERT(bpp%8 == 0);
 1950                                    const int shift = __max(0, UsedBitsPerChannel() - 8);
 1951                                    channels = bpp/8; ASSERT(channels >= m_header.channels);
 1952
 1953                                    for (i=0; i < h; i++) {
 1954                                            UINT32 yPos = yOffset;
 1955                                            cnt = 0;
 1956                                            for (j=0; j < w; j++) {
 1957                                                    for (UINT32 c=0; c < m_header.channels;
c++) {
 1958                                                            buff[cnt + channelMap[c]] =
Clamp8((m_channel[c][yPos] + yuvOffset16) >> shift);
 1959                                                    }
 1960                                                    cnt += channels;
 1961                                                    yPos++;
 1962                                            }
 1963                                            yOffset += yw;
 1964                                            buff += pitch;
 1965
 1966                                            if (cb) {
 1967                                                    percent += dP;
 1968                                                    if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
 1969                                            }
 1970                                    }
 1971                            }
 1972                            break;
 1973                    }
```

```
 1974         case ImageModeRGBColor:
 1975                 {
 1976                         ASSERT(m_header.channels == 3);
 1977                         ASSERT(m_header.bpp == m_header.channels*8);
 1978                         ASSERT(bpp%8 == 0);
 1979                         ASSERT(bpp >= m_header.bpp);
 1980
 1981                         DataT* y = m_channel[0]; ASSERT(y);
 1982                         DataT* u = m_channel[1]; ASSERT(u);
 1983                         DataT* v = m_channel[2]; ASSERT(v);
 1984                         UINT8 *buffg = &buff[channelMap[1]],
 1985                                  *buffr = &buff[channelMap[2]],
 1986                                  *buffb = &buff[channelMap[0]];
 1987                         UINT8 g;
 1988                         UINT32 cnt, channels = bpp/8;
 1989
 1990                         if (m_downsample) {
 1991                                 for (i=0; i < h; i++) {
 1992                                         UINT32 uPos = uOffset;
 1993                                         UINT32 yPos = yOffset;
 1994                                         cnt = 0;
 1995                                         for (j=0; j < w; j++) {
 1996                                                 // u and v are downsampled
 1997                                                 uAvg = u[uPos];
 1998                                                 vAvg = v[uPos];
 1999                                                 // Yuv
 2000                                                 buffg[cnt] = g = Clamp8(y[yPos] +
YUVoffset8 - ((uAvg + vAvg ) >> 2)); // must be logical shift operator
 2001                                                 buffr[cnt] = Clamp8(uAvg + g);
 2002                                                 buffb[cnt] = Clamp8(vAvg + g);
 2003                                                 cnt += channels;
 2004                                                 if (j & 1) uPos++;
 2005                                                 yPos++;
 2006                                         }
 2007                                         if (i & 1) uOffset += uw;
 2008                                         yOffset += yw;
 2009                                         buffb += pitch;
 2010                                         buffg += pitch;
 2011                                         buffr += pitch;
 2012
 2013                                         if (cb) {
 2014                                                 percent += dP;
 2015                                                 if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
 2016                                         }
 2017                                 }
 2018
 2019                         } else {
 2020                                 for (i=0; i < h; i++) {
 2021                                         cnt = 0;
 2022                                         UINT32 yPos = yOffset;
 2023                                         for (j = 0; j < w; j++) {
 2024                                                 uAvg = u[yPos];
 2025                                                 vAvg = v[yPos];
 2026                                                 // Yuv
 2027                                                 buffg[cnt] = g = Clamp8(y[yPos] +
YUVoffset8 - ((uAvg + vAvg ) >> 2)); // must be logical shift operator
 2028                                                 buffr[cnt] = Clamp8(uAvg + g);
 2029                                                 buffb[cnt] = Clamp8(vAvg + g);
 2030                                                 cnt += channels;
 2031                                                 yPos++;
 2032                                         }
 2033                                         yOffset += yw;
 2034                                         buffb += pitch;
 2035                                         buffg += pitch;
 2036                                         buffr += pitch;
 2037
 2038                                         if (cb) {
 2039                                                 percent += dP;
 2040                                                 if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
```

```
 2041                                            }
 2042                                    }
 2043                            }
 2044                            break;
 2045                    }
 2046            case ImageModeRGB48:
 2047                    {
 2048                            ASSERT(m_header.channels == 3);
 2049                            ASSERT(m_header.bpp == 48);
 2050
 2051                            const DataT yuvOffset16 = 1 << (UsedBitsPerChannel() - 1);
 2052
 2053                            DataT* y = m_channel[0]; ASSERT(y);
 2054                            DataT* u = m_channel[1]; ASSERT(u);
 2055                            DataT* v = m_channel[2]; ASSERT(v);
 2056                            UINT32 cnt, channels;
 2057                            DataT g;
 2058
 2059                            if (bpp >= 48 && bpp%16 == 0) {
 2060                                    const int shift = 16 - UsedBitsPerChannel(); ASSERT(shift
>= 0);
 2061                                    UINT16 *buff16 = (UINT16 *)buff;
 2062                                    int pitch16 = pitch/2;
 2063                                    channels = bpp/16; ASSERT(channels >= m_header.channels);
 2064
 2065                                    for (i=0; i < h; i++) {
 2066                                            UINT32 uPos = uOffset;
 2067                                            UINT32 yPos = yOffset;
 2068                                            cnt = 0;
 2069                                            for (j=0; j < w; j++) {
 2070                                                    uAvg = u[uPos];
 2071                                                    vAvg = v[uPos];
 2072                                                    // Yuv
 2073                                                    g = y[yPos] + yuvOffset16 - ((uAvg + vAvg
) >> 2); // must be logical shift operator
 2074                                                    buff16[cnt + channelMap[1]] = Clamp16(g <<
shift);
 2075                                                    buff16[cnt + channelMap[2]] =
Clamp16((uAvg + g) << shift);
 2076                                                    buff16[cnt + channelMap[0]] =
Clamp16((vAvg + g) << shift);
 2077                                                    cnt += channels;
 2078                                                    if (!m_downsample || (j & 1)) uPos++;
 2079                                                    yPos++;
 2080                                            }
 2081                                            if (!m_downsample || (i & 1)) uOffset += uw;
 2082                                            yOffset += yw;
 2083                                            buff16 += pitch16;
 2084
 2085                                            if (cb) {
 2086                                                    percent += dP;
 2087                                                    if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
 2088                                            }
 2089                                    }
 2090                            } else {
 2091                                    ASSERT(bpp%8 == 0);
 2092                                    const int shift = __max(0, UsedBitsPerChannel() - 8);
 2093                                    channels = bpp/8; ASSERT(channels >= m_header.channels);
 2094
 2095                                    for (i=0; i < h; i++) {
 2096                                            UINT32 uPos = uOffset;
 2097                                            UINT32 yPos = yOffset;
 2098                                            cnt = 0;
 2099                                            for (j=0; j < w; j++) {
 2100                                                    uAvg = u[uPos];
 2101                                                    vAvg = v[uPos];
 2102                                                    // Yuv
 2103                                                    g = y[yPos] + yuvOffset16 - ((uAvg + vAvg
) >> 2); // must be logical shift operator
```

```
 2104                                                    buff[cnt + channelMap[1]] = Clamp8(g >>
shift);
 2105                                                    buff[cnt + channelMap[2]] = Clamp8((uAvg
+ g) >> shift);
 2106                                                    buff[cnt + channelMap[0]] = Clamp8((vAvg
+ g) >> shift);
 2107                                                    cnt += channels;
 2108                                                    if (!m_downsample || (j & 1)) uPos++;
 2109                                                    yPos++;
 2110                                            }
 2111                                            if (!m_downsample || (i & 1)) uOffset += uw;
 2112                                            yOffset += yw;
 2113                                            buff += pitch;
 2114
 2115                                            if (cb) {
 2116                                                    percent += dP;
 2117                                                    if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
 2118                                            }
 2119                                    }
 2120                            }
 2121                            break;
 2122                    }
 2123            case ImageModeLabColor:
 2124                    {
 2125                            ASSERT(m_header.channels == 3);
 2126                            ASSERT(m_header.bpp == m_header.channels*8);
 2127                            ASSERT(bpp%8 == 0);
 2128
 2129                            DataT* l = m_channel[0]; ASSERT(l);
 2130                            DataT* a = m_channel[1]; ASSERT(a);
 2131                            DataT* b = m_channel[2]; ASSERT(b);
 2132                            UINT32 cnt, channels = bpp/8; ASSERT(channels >=
m_header.channels);
 2133
 2134                            for (i=0; i < h; i++) {
 2135                                    UINT32 uPos = uOffset;
 2136                                    UINT32 yPos = yOffset;
 2137                                    cnt = 0;
 2138                                    for (j=0; j < w; j++) {
 2139                                            uAvg = a[uPos];
 2140                                            vAvg = b[uPos];
 2141                                            buff[cnt + channelMap[0]] = Clamp8(l[yPos] +
YUVoffset8);
 2142                                            buff[cnt + channelMap[1]] = Clamp8(uAvg +
YUVoffset8);
 2143                                            buff[cnt + channelMap[2]] = Clamp8(vAvg +
YUVoffset8);
 2144                                            cnt += channels;
 2145                                            if (!m_downsample || (j & 1)) uPos++;
 2146                                            yPos++;
 2147                                    }
 2148                                    if (!m_downsample || (i & 1)) uOffset += uw;
 2149                                    yOffset += yw;
 2150                                    buff += pitch;
 2151
 2152                                    if (cb) {
 2153                                            percent += dP;
 2154                                            if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
 2155                                    }
 2156                            }
 2157                            break;
 2158                    }
 2159            case ImageModeLab48:
 2160                    {
 2161                            ASSERT(m_header.channels == 3);
 2162                            ASSERT(m_header.bpp == m_header.channels*16);
 2163
 2164                            const DataT yuvOffset16 = 1 << (UsedBitsPerChannel() - 1);
 2165
```

```
 2166                           DataT* l = m_channel[0]; ASSERT(l);
 2167                           DataT* a = m_channel[1]; ASSERT(a);
 2168                           DataT* b = m_channel[2]; ASSERT(b);
 2169                           UINT32 cnt, channels;
 2170
 2171                           if (bpp%16 == 0) {
 2172                                   const int shift = 16 - UsedBitsPerChannel(); ASSERT(shift
>= 0);
 2173                                   UINT16 *buff16 = (UINT16 *)buff;
 2174                                   int pitch16 = pitch/2;
 2175                                   channels = bpp/16; ASSERT(channels >= m_header.channels);
 2176
 2177                                   for (i=0; i < h; i++) {
 2178                                           UINT32 uPos = uOffset;
 2179                                           UINT32 yPos = yOffset;
 2180                                           cnt = 0;
 2181                                           for (j=0; j < w; j++) {
 2182                                                   uAvg = a[uPos];
 2183                                                   vAvg = b[uPos];
 2184                                                   buff16[cnt + channelMap[0]] =
Clamp16((l[yPos] + yuvOffset16) << shift);
 2185                                                   buff16[cnt + channelMap[1]] =
Clamp16((uAvg + yuvOffset16) << shift);
 2186                                                   buff16[cnt + channelMap[2]] =
Clamp16((vAvg + yuvOffset16) << shift);
 2187                                                   cnt += channels;
 2188                                                   if (!m_downsample || (j & 1)) uPos++;
 2189                                                   yPos++;
 2190                                           }
 2191                                           if (!m_downsample || (i & 1)) uOffset += uw;
 2192                                           yOffset += yw;
 2193                                           buff16 += pitch16;
 2194
 2195                                           if (cb) {
 2196                                                   percent += dP;
 2197                                                   if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
 2198                                           }
 2199                                   }
 2200                           } else {
 2201                                   ASSERT(bpp%8 == 0);
 2202                                   const int shift = __max(0, UsedBitsPerChannel() - 8);
 2203                                   channels = bpp/8; ASSERT(channels >= m_header.channels);
 2204
 2205                                   for (i=0; i < h; i++) {
 2206                                           UINT32 uPos = uOffset;
 2207                                           UINT32 yPos = yOffset;
 2208                                           cnt = 0;
 2209                                           for (j=0; j < w; j++) {
 2210                                                   uAvg = a[uPos];
 2211                                                   vAvg = b[uPos];
 2212                                                   buff[cnt + channelMap[0]] =
Clamp8((l[yPos] + yuvOffset16) >> shift);
 2213                                                   buff[cnt + channelMap[1]] = Clamp8((uAvg
+ yuvOffset16) >> shift);
 2214                                                   buff[cnt + channelMap[2]] = Clamp8((vAvg
+ yuvOffset16) >> shift);
 2215                                                   cnt += channels;
 2216                                                   if (!m_downsample || (j & 1)) uPos++;
 2217                                                   yPos++;
 2218                                           }
 2219                                           if (!m_downsample || (i & 1)) uOffset += uw;
 2220                                           yOffset += yw;
 2221                                           buff += pitch;
 2222
 2223                                           if (cb) {
 2224                                                   percent += dP;
 2225                                                   if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
 2226                                           }
 2227                                   }
```

```
 2228                        }
 2229                        break;
 2230                }
 2231        case ImageModeRGBA:
 2232        case ImageModeCMYKColor:
 2233                {
 2234                        ASSERT(m_header.channels == 4);
 2235                        ASSERT(m_header.bpp == m_header.channels*8);
 2236                        ASSERT(bpp%8 == 0);
 2237
 2238                        DataT* y = m_channel[0]; ASSERT(y);
 2239                        DataT* u = m_channel[1]; ASSERT(u);
 2240                        DataT* v = m_channel[2]; ASSERT(v);
 2241                        DataT* a = m_channel[3]; ASSERT(a);
 2242                        UINT8 g, aAvg;
 2243                        UINT32 cnt, channels = bpp/8; ASSERT(channels >=
m_header.channels);
 2244
 2245                        for (i=0; i < h; i++) {
 2246                                UINT32 uPos = uOffset;
 2247                                UINT32 yPos = yOffset;
 2248                                cnt = 0;
 2249                                for (j=0; j < w; j++) {
 2250                                        uAvg = u[uPos];
 2251                                        vAvg = v[uPos];
 2252                                        aAvg = Clamp8(a[uPos] + YUVoffset8);
 2253                                        // Yuv
 2254                                        buff[cnt + channelMap[1]] = g = Clamp8(y[yPos] +
YUVoffset8 - ((uAvg + vAvg ) >> 2)); // must be logical shift operator
 2255                                        buff[cnt + channelMap[2]] = Clamp8(uAvg + g);
 2256                                        buff[cnt + channelMap[0]] = Clamp8(vAvg + g);
 2257                                        buff[cnt + channelMap[3]] = aAvg;
 2258                                        cnt += channels;
 2259                                        if (!m_downsample || (j & 1)) uPos++;
 2260                                        yPos++;
 2261                                }
 2262                                if (!m_downsample || (i & 1)) uOffset += uw;
 2263                                yOffset += yw;
 2264                                buff += pitch;
 2265
 2266                                if (cb) {
 2267                                        percent += dP;
 2268                                        if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
 2269                                }
 2270                        }
 2271                        break;
 2272                }
 2273        case ImageModeCMYK64:
 2274                {
 2275                        ASSERT(m_header.channels == 4);
 2276                        ASSERT(m_header.bpp == 64);
 2277
 2278                        const DataT yuvOffset16 = 1 << (UsedBitsPerChannel() - 1);
 2279
 2280                        DataT* y = m_channel[0]; ASSERT(y);
 2281                        DataT* u = m_channel[1]; ASSERT(u);
 2282                        DataT* v = m_channel[2]; ASSERT(v);
 2283                        DataT* a = m_channel[3]; ASSERT(a);
 2284                        DataT g, aAvg;
 2285                        UINT32 cnt, channels;
 2286
 2287                        if (bpp%16 == 0) {
 2288                                const int shift = 16 - UsedBitsPerChannel(); ASSERT(shift
>= 0);
 2289                                UINT16 *buff16 = (UINT16 *)buff;
 2290                                int pitch16 = pitch/2;
 2291                                channels = bpp/16; ASSERT(channels >= m_header.channels);
 2292
 2293                                for (i=0; i < h; i++) {
 2294                                        UINT32 uPos = uOffset;
```

```
 2295                                           UINT32 yPos = yOffset;
 2296                                           cnt = 0;
 2297                                           for (j=0; j < w; j++) {
 2298                                                   uAvg = u[uPos];
 2299                                                   vAvg = v[uPos];
 2300                                                   aAvg = a[uPos] + yuvOffset16;
 2301                                                   // Yuv
 2302                                                   g = y[yPos] + yuvOffset16 - ((uAvg + vAvg
) >> 2); // must be logical shift operator
 2303                                                   buff16[cnt + channelMap[1]] = Clamp16(g <<
shift);
 2304                                                   buff16[cnt + channelMap[2]] =
Clamp16((uAvg + g) << shift);
 2305                                                   buff16[cnt + channelMap[0]] =
Clamp16((vAvg + g) << shift);
 2306                                                   buff16[cnt + channelMap[3]] =
Clamp16(aAvg << shift);
 2307                                                   cnt += channels;
 2308                                                   if (!m_downsample || (j & 1)) uPos++;
 2309                                                   yPos++;
 2310                                           }
 2311                                           if (!m_downsample || (i & 1)) uOffset += uw;
 2312                                           yOffset += yw;
 2313                                           buff16 += pitch16;
 2314
 2315                                           if (cb) {
 2316                                                   percent += dP;
 2317                                                   if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
 2318                                           }
 2319                                   }
 2320                           } else {
 2321                                   ASSERT(bpp%8 == 0);
 2322                                   const int shift = __max(0, UsedBitsPerChannel() - 8);
 2323                                   channels = bpp/8; ASSERT(channels >= m_header.channels);
 2324
 2325                                   for (i=0; i < h; i++) {
 2326                                           UINT32 uPos = uOffset;
 2327                                           UINT32 yPos = yOffset;
 2328                                           cnt = 0;
 2329                                           for (j=0; j < w; j++) {
 2330                                                   uAvg = u[uPos];
 2331                                                   vAvg = v[uPos];
 2332                                                   aAvg = a[uPos] + yuvOffset16;
 2333                                                   // Yuv
 2334                                                   g = y[yPos] + yuvOffset16 - ((uAvg + vAvg
) >> 2); // must be logical shift operator
 2335                                                   buff[cnt + channelMap[1]] = Clamp8(g >>
shift);
 2336                                                   buff[cnt + channelMap[2]] = Clamp8((uAvg
+ g) >> shift);
 2337                                                   buff[cnt + channelMap[0]] = Clamp8((vAvg
+ g) >> shift);
 2338                                                   buff[cnt + channelMap[3]] = Clamp8(aAvg >>
shift);
 2339                                                   cnt += channels;
 2340                                                   if (!m_downsample || (j & 1)) uPos++;
 2341                                                   yPos++;
 2342                                           }
 2343                                           if (!m_downsample || (i & 1)) uOffset += uw;
 2344                                           yOffset += yw;
 2345                                           buff += pitch;
 2346
 2347                                           if (cb) {
 2348                                                   percent += dP;
 2349                                                   if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
 2350                                           }
 2351                                   }
 2352                           }
 2353                           break;
```

```
 2354                 }
 2355 #ifdef __PGF32SUPPORT__
 2356         case ImageModeGray32:
 2357                 {
 2358                         ASSERT(m_header.channels == 1);
 2359                         ASSERT(m_header.bpp == 32);
 2360
 2361                         const int yuvOffset31 = 1 << (UsedBitsPerChannel() - 1);
 2362                         DataT* y = m_channel[0]; ASSERT(y);
 2363
 2364                         if (bpp == 32) {
 2365                                 const int shift = 31 - UsedBitsPerChannel(); ASSERT(shift
>= 0);
 2366                                 UINT32 *buff32 = (UINT32 *)buff;
 2367                                 int pitch32 = pitch/4;
 2368
 2369                                 for (i=0; i < h; i++) {
 2370                                         UINT32 yPos = yOffset;
 2371                                         for (j = 0; j < w; j++) {
 2372                                                 buff32[j] = Clamp31((y[yPos++] +
yuvOffset31) << shift);
 2373                                         }
 2374                                         yOffset += yw;
 2375                                         buff32 += pitch32;
 2376
 2377                                         if (cb) {
 2378                                                 percent += dP;
 2379                                                 if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
 2380                                         }
 2381                                 }
 2382                         } else if (bpp == 16) {
 2383                                 const int usedBits = UsedBitsPerChannel();
 2384                                 UINT16 *buff16 = (UINT16 *)buff;
 2385                                 int pitch16 = pitch/2;
 2386
 2387                                 if (usedBits < 16) {
 2388                                         const int shift = 16 - usedBits;
 2389                                         for (i=0; i < h; i++) {
 2390                                                 UINT32 yPos = yOffset;
 2391                                                 for (j = 0; j < w; j++) {
 2392                                                         buff16[j] = Clamp16((y[yPos++] +
yuvOffset31) << shift);
 2393                                                 }
 2394                                                 yOffset += yw;
 2395                                                 buff16 += pitch16;
 2396
 2397                                                 if (cb) {
 2398                                                         percent += dP;
 2399                                                         if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
 2400                                                 }
 2401                                         }
 2402                                 } else {
 2403                                         const int shift = __max(0, usedBits - 16);
 2404                                         for (i=0; i < h; i++) {
 2405                                                 UINT32 yPos = yOffset;
 2406                                                 for (j = 0; j < w; j++) {
 2407                                                         buff16[j] = Clamp16((y[yPos++] +
yuvOffset31) >> shift);
 2408                                                 }
 2409                                                 yOffset += yw;
 2410                                                 buff16 += pitch16;
 2411
 2412                                                 if (cb) {
 2413                                                         percent += dP;
 2414                                                         if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
 2415                                                 }
 2416                                         }
 2417                                 }
```

```
 2418                         } else {
 2419                                 ASSERT(bpp == 8);
 2420                                 const int shift = __max(0, UsedBitsPerChannel() - 8);
 2421
 2422                                 for (i=0; i < h; i++) {
 2423                                         UINT32 yPos = yOffset;
 2424                                         for (j = 0; j < w; j++) {
 2425                                                 buff[j] = Clamp8((y[yPos++] +
yuvOffset31) >> shift);
 2426                                         }
 2427                                         yOffset += yw;
 2428                                         buff += pitch;
 2429
 2430                                         if (cb) {
 2431                                                 percent += dP;
 2432                                                 if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
 2433                                         }
 2434                                 }
 2435                         }
 2436                         break;
 2437                 }
 2438 #endif
 2439         case ImageModeRGB12:
 2440                 {
 2441                         ASSERT(m_header.channels == 3);
 2442                         ASSERT(m_header.bpp == m_header.channels*4);
 2443                         ASSERT(bpp == m_header.channels*4);
 2444                         ASSERT(!m_downsample);
 2445
 2446                         DataT* y = m_channel[0]; ASSERT(y);
 2447                         DataT* u = m_channel[1]; ASSERT(u);
 2448                         DataT* v = m_channel[2]; ASSERT(v);
 2449                         UINT16 yval;
 2450                         UINT32 cnt;
 2451
 2452                         for (i=0; i < h; i++) {
 2453                                 UINT32 yPos = yOffset;
 2454                                 cnt = 0;
 2455                                 for (j=0; j < w; j++) {
 2456                                         // Yuv
 2457                                         uAvg = u[yPos];
 2458                                         vAvg = v[yPos];
 2459                                         yval = Clamp4(y[yPos] + YUVoffset4 - ((uAvg + vAvg
) >> 2)); // must be logical shift operator
 2460                                         if (j%2 == 0) {
 2461                                                 buff[cnt] = UINT8(Clamp4(vAvg + yval) |
(yval << 4));
 2462                                                 cnt++;
 2463                                                 buff[cnt] = Clamp4(uAvg + yval);
 2464                                         } else {
 2465                                                 buff[cnt] |= Clamp4(vAvg + yval) << 4;
 2466                                                 cnt++;
 2467                                                 buff[cnt] = UINT8(yval | (Clamp4(uAvg +
yval) << 4));
 2468                                                 cnt++;
 2469                                         }
 2470                                         yPos++;
 2471                                 }
 2472                                 yOffset += yw;
 2473                                 buff += pitch;
 2474
 2475                                 if (cb) {
 2476                                         percent += dP;
 2477                                         if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
 2478                                 }
 2479                         }
 2480                         break;
 2481                 }
 2482         case ImageModeRGB16:
```

```
 2483                 {
 2484                         ASSERT(m_header.channels == 3);
 2485                         ASSERT(m_header.bpp == 16);
 2486                         ASSERT(bpp == 16);
 2487                         ASSERT(!m_downsample);
 2488
 2489                         DataT* y = m_channel[0]; ASSERT(y);
 2490                         DataT* u = m_channel[1]; ASSERT(u);
 2491                         DataT* v = m_channel[2]; ASSERT(v);
 2492                         UINT16 yval;
 2493                         UINT16 *buff16 = (UINT16 *)buff;
 2494                         int pitch16 = pitch/2;
 2495
 2496                         for (i=0; i < h; i++) {
 2497                                 UINT32 yPos = yOffset;
 2498                                 for (j = 0; j < w; j++) {
 2499                                         // Yuv
 2500                                         uAvg = u[yPos];
 2501                                         vAvg = v[yPos];
 2502                                         yval = Clamp6(y[yPos++] + YUVoffset6 - ((uAvg +
vAvg ) >> 2)); // must be logical shift operator
 2503                                         buff16[j] = (yval << 5) | ((Clamp6(uAvg + yval) >>
1) << 11) | (Clamp6(vAvg + yval) >> 1);
 2504                                 }
 2505                                 yOffset += yw;
 2506                                 buff16 += pitch16;
 2507
 2508                                 if (cb) {
 2509                                         percent += dP;
 2510                                         if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
 2511                                 }
 2512                         }
 2513                         break;
 2514                 }
 2515         default:
 2516                 ASSERT(false);
 2517         }
 2518
 2519 #ifdef _DEBUG
 2520         // display ROI (RGB) in debugger
 2521         roiimage.width = w;
 2522         roiimage.height = h;
 2523         if (pitch > 0) {
 2524                 roiimage.pitch = pitch;
 2525                 roiimage.data = buff;
 2526         } else {
 2527                 roiimage.pitch = -pitch;
 2528                 roiimage.data = buff + (h - 1)*pitch;
 2529         }
 2530 #endif
 2531
 2532 }
```

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.

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

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 1348 of file PGFimage.cpp.

```
 1348
{
 1349         if (iFirstColor + nColors > ColorTableLen)      ReturnWithError(ColorTableError);
 1350
 1351         for (UINT32 i=iFirstColor, j=0; j < nColors; i++, j++) {
 1352                 prgbColors[j] = m_postHeader.clut[i];
 1353         }
 1354 }
```

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

Returns:

Address of color table

Definition at line 330 of file PGFimage.h.

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

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 647 of file PGFimage.cpp.

```
  647                                                 {
  648         ASSERT(m_decoder);
  649         return m_decoder->GetEncodedHeaderLength();
  650 }
```

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.

```
  367 { 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.

```
335 { 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.

```
341 { 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 336 of file PGFimage.cpp.

```
336                                                                                 {
337         cachedSize = m_postHeader.cachedUserDataLen;
338         if (pTotalSize) *pTotalSize = m_postHeader.userDataLen;
339         return m_postHeader.userData;
340 }
```

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.

```
346 { 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 2548 of file PGFimage.cpp.

```
 2548
{
 2549         ASSERT(buff);
 2550         const UINT32 w = m_width[0];
 2551         const UINT32 h = m_height[0];
 2552         const bool wOdd = (1 == w%2);
 2553         const int dataBits = DataTSize*8; ASSERT(dataBits == 16 || dataBits == 32);
 2554         const int pitch2 = pitch/DataTSize;
 2555         const int yuvOffset = (dataBits == 16) ? YUVoffset8 : YUVoffset16;
 2556         const double dP = 1.0/h;
 2557
 2558         int defMap[] = { 0, 1, 2, 3, 4, 5, 6, 7 }; ASSERT(sizeof(defMap)/sizeof(defMap[0])
== MaxChannels);
 2559         if (channelMap == nullptr) channelMap = defMap;
 2560         int sampledPos = 0, yPos = 0;
 2561         DataT uAvg, vAvg;
 2562         double percent = 0;
 2563         UINT32 i, j;
 2564
 2565         if (m_header.channels == 3) {
 2566                 ASSERT(bpp%dataBits == 0);
 2567
 2568                 DataT* y = m_channel[0]; ASSERT(y);
 2569                 DataT* u = m_channel[1]; ASSERT(u);
```

```
 2570                 DataT* v = m_channel[2]; ASSERT(v);
 2571                 int cnt, channels = bpp/dataBits; ASSERT(channels >= m_header.channels);
 2572
 2573                 for (i=0; i < h; i++) {
 2574                         if (i%2) sampledPos -= (w + 1)/2;
 2575                         cnt = 0;
 2576                         for (j=0; j < w; j++) {
 2577                                 if (m_downsample) {
 2578                                         // image was downsampled
 2579                                         uAvg = u[sampledPos];
 2580                                         vAvg = v[sampledPos];
 2581                                 } else {
 2582                                         uAvg = u[yPos];
 2583                                         vAvg = v[yPos];
 2584                                 }
 2585                                 buff[cnt + channelMap[0]] = y[yPos];
 2586                                 buff[cnt + channelMap[1]] = uAvg;
 2587                                 buff[cnt + channelMap[2]] = vAvg;
 2588                                 yPos++;
 2589                                 cnt += channels;
 2590                                 if (j%2) sampledPos++;
 2591                         }
 2592                         buff += pitch2;
 2593                         if (wOdd) sampledPos++;
 2594
 2595                         if (cb) {
 2596                                 percent += dP;
 2597                                 if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
 2598                         }
 2599                 }
 2600         } else if (m_header.channels == 4) {
 2601                 ASSERT(m_header.bpp == m_header.channels*8);
 2602                 ASSERT(bpp%dataBits == 0);
 2603
 2604                 DataT* y = m_channel[0]; ASSERT(y);
 2605                 DataT* u = m_channel[1]; ASSERT(u);
 2606                 DataT* v = m_channel[2]; ASSERT(v);
 2607                 DataT* a = m_channel[3]; ASSERT(a);
 2608                 UINT8 aAvg;
 2609                 int cnt, channels = bpp/dataBits; ASSERT(channels >= m_header.channels);
 2610
 2611                 for (i=0; i < h; i++) {
 2612                         if (i%2) sampledPos -= (w + 1)/2;
 2613                         cnt = 0;
 2614                         for (j=0; j < w; j++) {
 2615                                 if (m_downsample) {
 2616                                         // image was downsampled
 2617                                         uAvg = u[sampledPos];
 2618                                         vAvg = v[sampledPos];
 2619                                         aAvg = Clamp8(a[sampledPos] + yuvOffset);
 2620                                 } else {
 2621                                         uAvg = u[yPos];
 2622                                         vAvg = v[yPos];
 2623                                         aAvg = Clamp8(a[yPos] + yuvOffset);
 2624                                 }
 2625                                 // Yuv
 2626                                 buff[cnt + channelMap[0]] = y[yPos];
 2627                                 buff[cnt + channelMap[1]] = uAvg;
 2628                                 buff[cnt + channelMap[2]] = vAvg;
 2629                                 buff[cnt + channelMap[3]] = aAvg;
 2630                                 yPos++;
 2631                                 cnt += channels;
 2632                                 if (j%2) sampledPos++;
 2633                         }
 2634                         buff += pitch2;
 2635                         if (wOdd) sampledPos++;
 2636
 2637                         if (cb) {
 2638                                 percent += dP;
```

```
 2639                                 if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
 2640                                 }
 2641                 }
 2642         }
 2643 }
```

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.

```
  420 { 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 790 of file PGFimage.cpp.

```
  790
{
  791         ASSERT(buff);
  792         ASSERT(m_channel[0]);
  793
  794         // color transform
  795         RgbToYuv(pitch, buff, bpp, channelMap, cb, data);
  796
  797         if (m_downsample) {
  798                 // Subsampling of the chrominance and alpha channels
  799                 for (int i=1; i < m_header.channels; i++) {
  800                         Downsample(i);
  801                 }
  802         }
  803 }
```

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 1303 of file PGFimage.cpp.

```
1303                                            {
1304         size_t size = DataTSize;
1305
1306         if (size >= 2) {
1307                 switch(mode) {
1308                         case ImageModeBitmap:
1309                         case ImageModeIndexedColor:
1310                         case ImageModeGrayScale:
1311                         case ImageModeRGBColor:
1312                         case ImageModeCMYKColor:
1313                         case ImageModeHSLColor:
1314                         case ImageModeHSBColor:
1315                         //case ImageModeDuotone:
1316                         case ImageModeLabColor:
1317                         case ImageModeRGB12:
1318                         case ImageModeRGB16:
1319                         case ImageModeRGBA:
1320                                 return true;
1321                 }
1322         }
1323         if (size >= 3) {
1324                 switch(mode) {
1325                         case ImageModeGray16:
1326                         case ImageModeRGB48:
1327                         case ImageModeLab48:
1328                         case ImageModeCMYK64:
1329                         //case ImageModeDuotone16:
1330                                 return true;
1331                 }
1332         }
1333         if (size >=4) {
1334                 switch(mode) {
1335                         case ImageModeGray32:
1336                                 return true;
1337                 }
1338         }
1339         return false;
1340 }
```

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 2659 of file PGFimage.cpp.

```
 2659
{
 2660         ASSERT(buff);
 2661         const double dP = 1.0/m_header.height;
 2662         const int dataBits = DataTSize*8; ASSERT(dataBits == 16 || dataBits == 32);
 2663         const int pitch2 = pitch/DataTSize;
 2664         const int yuvOffset = (dataBits == 16) ? YUVoffset8 : YUVoffset16;
 2665
 2666         int yPos = 0, cnt = 0;
 2667         double percent = 0;
 2668         int defMap[] = { 0, 1, 2, 3, 4, 5, 6, 7 }; ASSERT(sizeof(defMap)/sizeof(defMap[0])
== MaxChannels);
 2669
 2670         if (channelMap == nullptr) channelMap = defMap;
 2671
 2672         if (m_header.channels == 3)     {
 2673                 ASSERT(bpp%dataBits == 0);
 2674
 2675                 DataT* y = m_channel[0]; ASSERT(y);
 2676                 DataT* u = m_channel[1]; ASSERT(u);
 2677                 DataT* v = m_channel[2]; ASSERT(v);
 2678                 const int channels = bpp/dataBits; ASSERT(channels >= m_header.channels);
 2679
 2680                 for (UINT32 h=0; h < m_header.height; h++) {
 2681                         if (cb) {
 2682                                 if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
 2683                                 percent += dP;
 2684                         }
 2685
 2686                         cnt = 0;
 2687                         for (UINT32 w=0; w < m_header.width; w++) {
 2688                                 y[yPos] = buff[cnt + channelMap[0]];
 2689                                 u[yPos] = buff[cnt + channelMap[1]];
 2690                                 v[yPos] = buff[cnt + channelMap[2]];
 2691                                 yPos++;
 2692                                 cnt += channels;
 2693                         }
 2694                         buff += pitch2;
```

```
 2695                 }
 2696         } else if (m_header.channels == 4) {
 2697                 ASSERT(bpp%dataBits == 0);
 2698
 2699                 DataT* y = m_channel[0]; ASSERT(y);
 2700                 DataT* u = m_channel[1]; ASSERT(u);
 2701                 DataT* v = m_channel[2]; ASSERT(v);
 2702                 DataT* a = m_channel[3]; ASSERT(a);
 2703                 const int channels = bpp/dataBits; ASSERT(channels >= m_header.channels);
 2704
 2705                 for (UINT32 h=0; h < m_header.height; h++) {
 2706                         if (cb) {
 2707                                 if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
 2708                                 percent += dP;
 2709                         }
 2710
 2711                         cnt = 0;
 2712                         for (UINT32 w=0; w < m_header.width; w++) {
 2713                                 y[yPos] = buff[cnt + channelMap[0]];
 2714                                 u[yPos] = buff[cnt + channelMap[1]];
 2715                                 v[yPos] = buff[cnt + channelMap[2]];
 2716                                 a[yPos] = buff[cnt + channelMap[3]] - yuvOffset;
 2717                                 yPos++;
 2718                                 cnt += channels;
 2719                         }
 2720                         buff += pitch2;
 2721                 }
 2722         }
 2723
 2724         if (m_downsample) {
 2725                 // Subsampling of the chrominance and alpha channels
 2726                 for (int i=1; i < m_header.channels; i++) {
 2727                         Downsample(i);
 2728                 }
 2729         }
 2730 }
```

void CPGFImage::Init () [private]

Definition at line 69 of file PGFimage.cpp.

```
   69                       {
   70         // init pointers
   71         m_decoder = nullptr;
   72         m_encoder = nullptr;
   73         m_levelLength = nullptr;
   74
   75         // init members
   76 #ifdef __PGFROISUPPORT__
   77         m_streamReinitialized = false;
   78 #endif
   79         m_currentLevel = 0;
   80         m_quant = 0;
   81         m_userDataPos = 0;
   82         m_downsample = false;
   83         m_favorSpeedOverSize = false;
   84         m_useOMPinEncoder = true;
   85         m_useOMPinDecoder = true;
   86         m_cb = nullptr;
   87         m_cbArg = nullptr;
   88         m_progressMode = PM_Relative;
   89         m_percent = 0;
   90         m_userDataPolicy = UP_CacheAll;
   91
   92         // init preHeader
   93         memcpy(m_preHeader.magic, PGFMagic, 3);
   94         m_preHeader.version = PGFVersion;
   95         m_preHeader.hSize = 0;
```

```
 96
 97         // init postHeader
 98         m_postHeader.userData = nullptr;
 99         m_postHeader.userDataLen = 0;
100         m_postHeader.cachedUserDataLen = 0;
101
102         // init channels
103         for (int i = 0; i < MaxChannels; i++) {
104                 m_channel[i] = nullptr;
105                 m_wtChannel[i] = nullptr;
106         }
107
108         // set image width and height
109         for (int i = 0; i < MaxChannels; i++) {
110                 m_width[0] = 0;
111                 m_height[0] = 0;
112         }
113 }
```

bool CPGFImage::IsFullyRead () const[inline]

Return true if all levels have been read.

Definition at line 436 of file PGFimage.h.

```
436 { 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.

```
 77 { 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.

```
427 { 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.

```
432 { 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.

```
506 { 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.

```
499 { 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.

```
518 { 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.

```
455 { 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.

```
141                                         {
142         ASSERT(stream);
143
144         // create decoder and read PGFPreHeader PGFHeader PGFPostHeader LevelLengths
145         m_decoder = new CDecoder(stream, m_preHeader, m_header, m_postHeader,
m_levelLength,
146                 m_userDataPos, m_useOMPinDecoder, m_userDataPolicy);
147
148         if (m_header.nLevels > MaxLevel) ReturnWithError(FormatCannotRead);
149
150         // set current level
151         m_currentLevel = m_header.nLevels;
```

```
  152
  153         // set image width and height
  154         m_width[0] = m_header.width;
  155         m_height[0] = m_header.height;
  156
  157         // complete header
  158         CompleteHeader();
  159
  160         // interpret quant parameter
  161         if (m_header.quality > DownsampleThreshold &&
  162                 (m_header.mode == ImageModeRGBColor ||
  163                  m_header.mode == ImageModeRGBA ||
  164                  m_header.mode == ImageModeRGB48 ||
  165                  m_header.mode == ImageModeCMYKColor ||
  166                  m_header.mode == ImageModeCMYK64 ||
  167                  m_header.mode == ImageModeLabColor ||
  168                  m_header.mode == ImageModeLab48)) {
  169                 m_downsample = true;
  170                 m_quant = m_header.quality - 1;
  171         } else {
  172                 m_downsample = false;
  173                 m_quant = m_header.quality;
  174         }
  175
  176         // set channel dimensions (chrominance is subsampled by factor 2)
  177         if (m_downsample) {
  178                 for (int i=1; i < m_header.channels; i++) {
  179                         m_width[i] = (m_width[0] + 1) >> 1;
  180                         m_height[i] = (m_height[0] + 1) >> 1;
  181                 }
  182         } else {
  183                 for (int i=1; i < m_header.channels; i++) {
  184                         m_width[i] = m_width[0];
  185                         m_height[i] = m_height[0];
  186                 }
  187         }
  188
  189         if (m_header.nLevels > 0) {
  190                 // init wavelet subbands
  191                 for (int i=0; i < m_header.channels; i++) {
  192                         m_wtChannel[i] = new CWaveletTransform(m_width[i], m_height[i],
m_header.nLevels);
  193                 }
  194
  195                 // used in Read when PM_Absolute
  196                 m_percent = pow(0.25, m_header.nLevels);
  197
  198         } else {
  199                 // very small image: we don't use DWT and encoding
  200
  201                 // read channels
  202                 for (int c=0; c < m_header.channels; c++) {
  203                         const UINT32 size = m_width[c]*m_height[c];
  204                         m_channel[c] = new(std::nothrow) DataT[size];
  205                         if (!m_channel[c]) ReturnWithError(InsufficientMemory);
  206
  207                         // read channel data from stream
  208                         for (UINT32 i=0; i < size; i++) {
  209                                 int count = DataTSize;
  210                                 stream->Read(&count, &m_channel[c][i]);
  211                                 if (count != DataTSize) ReturnWithError(MissingData);
  212                         }
  213                 }
  214         }
  215 }
```

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.

```
442 { 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 401 of file PGFimage.cpp.

```
401                                                                     {
402         ASSERT((level >= 0 && level < m_header.nLevels) || m_header.nLevels == 0); //
m_header.nLevels == 0: image didn't use wavelet transform
403         ASSERT(m_decoder);
404
405 #ifdef __PGFROISUPPORT__
406         if (ROIisSupported() && m_header.nLevels > 0) {
407                 // new encoding scheme supporting ROI
408                 PGFRect rect(0, 0, m_header.width, m_header.height);
409                 Read(rect, level, cb, data);
410                 return;
411         }
412 #endif
413
414         if (m_header.nLevels == 0) {
415                 if (level == 0) {
416                         // the data has already been read during open
417                         // now update progress
418                         if (cb) {
419                                 if ((*cb)(1.0, true, data))
ReturnWithError(EscapePressed);
420                         }
421                 }
422         } else {
423                 const int levelDiff = m_currentLevel - level;
424                 double percent = (m_progressMode == PM_Relative) ? pow(0.25, levelDiff) :
m_percent;
425
426                 // encoding scheme without ROI
427                 while (m_currentLevel > level) {
428                         for (int i=0; i < m_header.channels; i++) {
429                                 CWaveletTransform* wtChannel = m_wtChannel[i];
430                                 ASSERT(wtChannel);
431
432                                 // decode file and write stream to m_wtChannel
433                                 if (m_currentLevel == m_header.nLevels) {
434                                         // last level also has LL band
435                                         wtChannel->GetSubband(m_currentLevel,
LL)->PlaceTile(*m_decoder, m_quant);
436                                 }
437                                 if (m_preHeader.version & Version5) {
438                                         // since version 5
```

```
  439                                           wtChannel->GetSubband(m_currentLevel,
HL)->PlaceTile(*m_decoder, m_quant);
  440                                           wtChannel->GetSubband(m_currentLevel,
LH)->PlaceTile(*m_decoder, m_quant);
  441                                   } else {
  442                                           // until version 4
  443                                           m_decoder->DecodeInterleaved(wtChannel,
m_currentLevel, m_quant);
  444                                   }
  445                                   wtChannel->GetSubband(m_currentLevel,
HH)->PlaceTile(*m_decoder, m_quant);
  446                           }
  447
  448                           volatile OSError error = NoError; // volatile prevents
optimizations
  449 #ifdef LIBPGF_USE_OPENMP
  450                           #pragma omp parallel for default(shared)
  451 #endif
  452                           for (int i=0; i < m_header.channels; i++) {
  453                                   // inverse transform from m_wtChannel to m_channel
  454                                   if (error == NoError) {
  455                                           OSError err =
m_wtChannel[i]->InverseTransform(m_currentLevel, &m_width[i], &m_height[i], &m_channel[i]);
  456                                           if (err != NoError) error = err;
  457                                   }
  458                                   ASSERT(m_channel[i]);
  459                           }
  460                           if (error != NoError) ReturnWithError(error);
  461
  462                           // set new level: must be done before refresh callback
  463                           m_currentLevel--;
  464
  465                           // now we have to refresh the display
  466                           if (m_cb) m_cb(m_cbArg);
  467
  468                           // now update progress
  469                           if (cb) {
  470                                   percent *= 4;
  471                                   if (m_progressMode == PM_Absolute) m_percent = percent;
  472                                   if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
  473                           }
  474                   }
  475           }
  476 }
```

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 705 of file PGFimage.cpp.

```
705                                                                                 {
706         ASSERT(level >= 0 && level < m_header.nLevels);
707         ASSERT(target);
708         ASSERT(targetLen > 0);
709         ASSERT(m_decoder);
710
711         // reset stream position
712         m_decoder->SetStreamPosToData();
713
714         // position stream
715         UINT64 offset = 0;
716
717         for (int i=m_header.nLevels - 1; i > level; i--) {
718                 offset += m_levelLength[m_header.nLevels - 1 - i];
719         }
720         m_decoder->Skip(offset);
721
722         // compute number of bytes to read
723         UINT32 len = __min(targetLen, GetEncodedLevelLength(level));
724
725         // read data
726         len = m_decoder->ReadEncodedData(target, len);
727         ASSERT(len >= 0 && len <= targetLen);
728
729         return len;
730 }
```

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 659 of file PGFimage.cpp.

```
659                                                                             {
660         ASSERT(target);
661         ASSERT(targetLen > 0);
662         ASSERT(m_decoder);
663
664         // reset stream position
665         m_decoder->SetStreamPosToStart();
666
667         // compute number of bytes to read
668         UINT32 len = __min(targetLen, GetEncodedHeaderLength());
669
670         // read data
```

```
  671         len = m_decoder->ReadEncodedData(target, len);
  672         ASSERT(len >= 0 && len <= targetLen);
  673
  674         return len;
  675 }
```

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.

```
  111 { 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 347 of file PGFimage.cpp.

```
  347                                         {
  348         if (m_header.nLevels == 0) {
  349                 // image didn't use wavelet transform
  350                 if (level == 0) {
  351                         for (int i=0; i < m_header.channels; i++) {
  352                                 ASSERT(m_wtChannel[i]);
  353                                 m_channel[i] = m_wtChannel[i]->GetSubband(0,
LL)->GetBuffer();
  354                         }
  355                 }
  356         } else {
  357                 int currentLevel = m_header.nLevels;
  358
  359         #ifdef __PGFROISUPPORT__
  360                 if (ROIisSupported()) {
  361                         // enable ROI reading
  362                         SetROI(PGFRect(0, 0, m_header.width, m_header.height));
  363                 }
  364         #endif
  365
  366                 while (currentLevel > level) {
  367                         for (int i=0; i < m_header.channels; i++) {
  368                                 ASSERT(m_wtChannel[i]);
  369                                 // dequantize subbands
  370                                 if (currentLevel == m_header.nLevels) {
  371                                         // last level also has LL band
  372                                         m_wtChannel[i]->GetSubband(currentLevel,
LL)->Dequantize(m_quant);
  373                                 }
  374                                 m_wtChannel[i]->GetSubband(currentLevel,
HL)->Dequantize(m_quant);
  375                                 m_wtChannel[i]->GetSubband(currentLevel,
LH)->Dequantize(m_quant);
  376                                 m_wtChannel[i]->GetSubband(currentLevel,
HH)->Dequantize(m_quant);
  377
  378                                 // inverse transform from m_wtChannel to m_channel
  379                                 OSError err =
m_wtChannel[i]->InverseTransform(currentLevel, &m_width[i], &m_height[i], &m_channel[i]);
  380                                 if (err != NoError) ReturnWithError(err);
  381                                 ASSERT(m_channel[i]);
  382                         }
  383
  384                         currentLevel--;
```

```
 385                 }
 386         }
 387 }
```

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 681 of file PGFimage.cpp.

```
 681                                                 {
 682         if (startOfData) {
 683                 ASSERT(m_decoder);
 684                 m_decoder->SetStreamPosToData();
 685         } else {
 686                 if (m_decoder) {
 687                         m_decoder->SetStreamPosToStart();
 688                 } else if (m_encoder) {
 689                         m_encoder->SetStreamPosToStart();
 690                 } else {
 691                         ASSERT(false);
 692                 }
 693         }
 694 }
```

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

Definition at line 1387 of file PGFimage.cpp.

```
 1387
{
 1388         ASSERT(buff);
 1389         UINT32 yPos = 0, cnt = 0;
 1390         double percent = 0;
 1391         const double dP = 1.0/m_header.height;
 1392         int defMap[] = { 0, 1, 2, 3, 4, 5, 6, 7 }; ASSERT(sizeof(defMap)/sizeof(defMap[0])
== MaxChannels);
 1393
 1394         if (channelMap == nullptr) channelMap = defMap;
 1395
 1396         switch(m_header.mode) {
 1397         case ImageModeBitmap:
 1398                 {
 1399                         ASSERT(m_header.channels == 1);
 1400                         ASSERT(m_header.bpp == 1);
 1401                         ASSERT(bpp == 1);
 1402
 1403                         const UINT32 w = m_header.width;
 1404                         const UINT32 w2 = (m_header.width + 7)/8;
 1405                         DataT* y = m_channel[0]; ASSERT(y);
 1406
 1407                         // new unpacked version since version 7
 1408                         for (UINT32 h = 0; h < m_header.height; h++) {
 1409                                 if (cb) {
 1410                                         if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
 1411                                         percent += dP;
 1412                                 }
 1413                                 cnt = 0;
 1414                                 for (UINT32 j = 0; j < w2; j++) {
 1415                                         UINT8 byte = buff[j];
```

```
 1416                                        for (int k = 0; k < 8; k++) {
 1417                                                UINT8 bit = (byte & 0x80) >> 7;
 1418                                                if (cnt < w) y[yPos++] = bit;
 1419                                                byte <<= 1;
 1420                                                cnt++;
 1421                                        }
 1422                                }
 1423                                buff += pitch;
 1424                        }
 1425                        /* old version: packed values: 8 pixels in 1 byte
 1426                        for (UINT32 h = 0; h < m_header.height; h++) {
 1427                                if (cb) {
 1428                                        if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
 1429                                        percent += dP;
 1430                                }
 1431
 1432                                for (UINT32 j = 0; j < w2; j++) {
 1433                                        y[yPos++] = buff[j] - YUVoffset8;
 1434                                }
 1435                                // version 5 and 6
 1436                                // for (UINT32 j = w2; j < w; j++) {
 1437                                //      y[yPos++] = YUVoffset8;
 1438                                //}
 1439                                buff += pitch;
 1440                        }
 1441                        */
 1442                }
 1443                break;
 1444        case ImageModeIndexedColor:
 1445        case ImageModeGrayScale:
 1446        case ImageModeHSLColor:
 1447        case ImageModeHSBColor:
 1448        case ImageModeLabColor:
 1449                {
 1450                        ASSERT(m_header.channels >= 1);
 1451                        ASSERT(m_header.bpp == m_header.channels*8);
 1452                        ASSERT(bpp%8 == 0);
 1453                        const int channels = bpp/8; ASSERT(channels >= m_header.channels);
 1454
 1455                        for (UINT32 h=0; h < m_header.height; h++) {
 1456                                if (cb) {
 1457                                        if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
 1458                                        percent += dP;
 1459                                }
 1460
 1461                                cnt = 0;
 1462                                for (UINT32 w=0; w < m_header.width; w++) {
 1463                                        for (int c=0; c < m_header.channels; c++) {
 1464                                                m_channel[c][yPos] = buff[cnt +
channelMap[c]] - YUVoffset8;
 1465                                        }
 1466                                        cnt += channels;
 1467                                        yPos++;
 1468                                }
 1469                                buff += pitch;
 1470                        }
 1471                }
 1472                break;
 1473        case ImageModeGray16:
 1474        case ImageModeLab48:
 1475                {
 1476                        ASSERT(m_header.channels >= 1);
 1477                        ASSERT(m_header.bpp == m_header.channels*16);
 1478                        ASSERT(bpp%16 == 0);
 1479
 1480                        UINT16 *buff16 = (UINT16 *)buff;
 1481                        const int pitch16 = pitch/2;
 1482                        const int channels = bpp/16; ASSERT(channels >= m_header.channels);
 1483                        const int shift = 16 - UsedBitsPerChannel(); ASSERT(shift >= 0);
```

```
 1484                           const DataT yuvOffset16 = 1 << (UsedBitsPerChannel() - 1);
 1485
 1486                           for (UINT32 h=0; h < m_header.height; h++) {
 1487                                   if (cb) {
 1488                                           if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
 1489                                           percent += dP;
 1490                                   }
 1491
 1492                                   cnt = 0;
 1493                                   for (UINT32 w=0; w < m_header.width; w++) {
 1494                                           for (int c=0; c < m_header.channels; c++) {
 1495                                                   m_channel[c][yPos] = (buff16[cnt +
channelMap[c]] >> shift) - yuvOffset16;
 1496                                           }
 1497                                           cnt += channels;
 1498                                           yPos++;
 1499                                   }
 1500                                   buff16 += pitch16;
 1501                           }
 1502                   }
 1503                   break;
 1504           case ImageModeRGBColor:
 1505                   {
 1506                           ASSERT(m_header.channels == 3);
 1507                           ASSERT(m_header.bpp == m_header.channels*8);
 1508                           ASSERT(bpp%8 == 0);
 1509
 1510                           DataT* y = m_channel[0]; ASSERT(y);
 1511                           DataT* u = m_channel[1]; ASSERT(u);
 1512                           DataT* v = m_channel[2]; ASSERT(v);
 1513                           const int channels = bpp/8; ASSERT(channels >= m_header.channels);
 1514                           UINT8 b, g, r;
 1515
 1516                           for (UINT32 h=0; h < m_header.height; h++) {
 1517                                   if (cb) {
 1518                                           if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
 1519                                           percent += dP;
 1520                                   }
 1521
 1522                                   cnt = 0;
 1523                                   for (UINT32 w=0; w < m_header.width; w++) {
 1524                                           b = buff[cnt + channelMap[0]];
 1525                                           g = buff[cnt + channelMap[1]];
 1526                                           r = buff[cnt + channelMap[2]];
 1527                                           // Yuv
 1528                                           y[yPos] = ((b + (g << 1) + r) >> 2) - YUVoffset8;
 1529                                           u[yPos] = r - g;
 1530                                           v[yPos] = b - g;
 1531                                           yPos++;
 1532                                           cnt += channels;
 1533                                   }
 1534                                   buff += pitch;
 1535                           }
 1536                   }
 1537                   break;
 1538           case ImageModeRGB48:
 1539                   {
 1540                           ASSERT(m_header.channels == 3);
 1541                           ASSERT(m_header.bpp == m_header.channels*16);
 1542                           ASSERT(bpp%16 == 0);
 1543
 1544                           UINT16 *buff16 = (UINT16 *)buff;
 1545                           const int pitch16 = pitch/2;
 1546                           const int channels = bpp/16; ASSERT(channels >= m_header.channels);
 1547                           const int shift = 16 - UsedBitsPerChannel(); ASSERT(shift >= 0);
 1548                           const DataT yuvOffset16 = 1 << (UsedBitsPerChannel() - 1);
 1549
 1550                           DataT* y = m_channel[0]; ASSERT(y);
 1551                           DataT* u = m_channel[1]; ASSERT(u);
```

```
 1552                           DataT* v = m_channel[2]; ASSERT(v);
 1553                           UINT16 b, g, r;
 1554
 1555                           for (UINT32 h=0; h < m_header.height; h++) {
 1556                                   if (cb) {
 1557                                           if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
 1558                                           percent += dP;
 1559                                   }
 1560
 1561                                   cnt = 0;
 1562                                   for (UINT32 w=0; w < m_header.width; w++) {
 1563                                           b = buff16[cnt + channelMap[0]] >> shift;
 1564                                           g = buff16[cnt + channelMap[1]] >> shift;
 1565                                           r = buff16[cnt + channelMap[2]] >> shift;
 1566                                           // Yuv
 1567                                           y[yPos] = ((b + (g << 1) + r) >> 2) - yuvOffset16;
 1568                                           u[yPos] = r - g;
 1569                                           v[yPos] = b - g;
 1570                                           yPos++;
 1571                                           cnt += channels;
 1572                                   }
 1573                                   buff16 += pitch16;
 1574                           }
 1575                   }
 1576                   break;
 1577           case ImageModeRGBA:
 1578           case ImageModeCMYKColor:
 1579                   {
 1580                           ASSERT(m_header.channels == 4);
 1581                           ASSERT(m_header.bpp == m_header.channels*8);
 1582                           ASSERT(bpp%8 == 0);
 1583                           const int channels = bpp/8; ASSERT(channels >= m_header.channels);
 1584
 1585                           DataT* y = m_channel[0]; ASSERT(y);
 1586                           DataT* u = m_channel[1]; ASSERT(u);
 1587                           DataT* v = m_channel[2]; ASSERT(v);
 1588                           DataT* a = m_channel[3]; ASSERT(a);
 1589                           UINT8 b, g, r;
 1590
 1591                           for (UINT32 h=0; h < m_header.height; h++) {
 1592                                   if (cb) {
 1593                                           if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
 1594                                           percent += dP;
 1595                                   }
 1596
 1597                                   cnt = 0;
 1598                                   for (UINT32 w=0; w < m_header.width; w++) {
 1599                                           b = buff[cnt + channelMap[0]];
 1600                                           g = buff[cnt + channelMap[1]];
 1601                                           r = buff[cnt + channelMap[2]];
 1602                                           // Yuv
 1603                                           y[yPos] = ((b + (g << 1) + r) >> 2) - YUVoffset8;
 1604                                           u[yPos] = r - g;
 1605                                           v[yPos] = b - g;
 1606                                           a[yPos++] = buff[cnt + channelMap[3]] -
YUVoffset8;
 1607                                           cnt += channels;
 1608                                   }
 1609                                   buff += pitch;
 1610                           }
 1611                   }
 1612                   break;
 1613           case ImageModeCMYK64:
 1614                   {
 1615                           ASSERT(m_header.channels == 4);
 1616                           ASSERT(m_header.bpp == m_header.channels*16);
 1617                           ASSERT(bpp%16 == 0);
 1618
 1619                           UINT16 *buff16 = (UINT16 *)buff;
```

```
 1620                           const int pitch16 = pitch/2;
 1621                           const int channels = bpp/16; ASSERT(channels >= m_header.channels);
 1622                           const int shift = 16 - UsedBitsPerChannel(); ASSERT(shift >= 0);
 1623                           const DataT yuvOffset16 = 1 << (UsedBitsPerChannel() - 1);
 1624
 1625                           DataT* y = m_channel[0]; ASSERT(y);
 1626                           DataT* u = m_channel[1]; ASSERT(u);
 1627                           DataT* v = m_channel[2]; ASSERT(v);
 1628                           DataT* a = m_channel[3]; ASSERT(a);
 1629                           UINT16 b, g, r;
 1630
 1631                           for (UINT32 h=0; h < m_header.height; h++) {
 1632                                   if (cb) {
 1633                                           if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
 1634                                           percent += dP;
 1635                                   }
 1636
 1637                                   cnt = 0;
 1638                                   for (UINT32 w=0; w < m_header.width; w++) {
 1639                                           b = buff16[cnt + channelMap[0]] >> shift;
 1640                                           g = buff16[cnt + channelMap[1]] >> shift;
 1641                                           r = buff16[cnt + channelMap[2]] >> shift;
 1642                                           // Yuv
 1643                                           y[yPos] = ((b + (g << 1) + r) >> 2) - yuvOffset16;
 1644                                           u[yPos] = r - g;
 1645                                           v[yPos] = b - g;
 1646                                           a[yPos++] = (buff16[cnt + channelMap[3]] >> shift)
- yuvOffset16;
 1647                                           cnt += channels;
 1648                                   }
 1649                                   buff16 += pitch16;
 1650                           }
 1651                   }
 1652                   break;
 1653 #ifdef __PGF32SUPPORT__
 1654           case ImageModeGray32:
 1655                   {
 1656                           ASSERT(m_header.channels == 1);
 1657                           ASSERT(m_header.bpp == 32);
 1658                           ASSERT(bpp == 32);
 1659                           ASSERT(DataTSize == sizeof(UINT32));
 1660
 1661                           DataT* y = m_channel[0]; ASSERT(y);
 1662
 1663                           UINT32 *buff32 = (UINT32 *)buff;
 1664                           const int pitch32 = pitch/4;
 1665                           const int shift = 31 - UsedBitsPerChannel(); ASSERT(shift >= 0);
 1666                           const DataT yuvOffset31 = 1 << (UsedBitsPerChannel() - 1);
 1667
 1668                           for (UINT32 h=0; h < m_header.height; h++) {
 1669                                   if (cb) {
 1670                                           if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
 1671                                           percent += dP;
 1672                                   }
 1673
 1674                                   for (UINT32 w=0; w < m_header.width; w++) {
 1675                                           y[yPos++] = (buff32[w] >> shift) - yuvOffset31;
 1676                                   }
 1677                                   buff32 += pitch32;
 1678                           }
 1679                   }
 1680                   break;
 1681 #endif
 1682           case ImageModeRGB12:
 1683                   {
 1684                           ASSERT(m_header.channels == 3);
 1685                           ASSERT(m_header.bpp == m_header.channels*4);
 1686                           ASSERT(bpp == m_header.channels*4);
 1687
```

```
1688                        DataT* y = m_channel[0]; ASSERT(y);
1689                        DataT* u = m_channel[1]; ASSERT(u);
1690                        DataT* v = m_channel[2]; ASSERT(v);
1691
1692                        UINT8 rgb = 0, b, g, r;
1693
1694                        for (UINT32 h=0; h < m_header.height; h++) {
1695                                if (cb) {
1696                                        if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
1697                                        percent += dP;
1698                                }
1699
1700                                cnt = 0;
1701                                for (UINT32 w=0; w < m_header.width; w++) {
1702                                        if (w%2 == 0) {
1703                                                // even pixel position
1704                                                rgb = buff[cnt];
1705                                                b = rgb & 0x0F;
1706                                                g = (rgb & 0xF0) >> 4;
1707                                                cnt++;
1708                                                rgb = buff[cnt];
1709                                                r = rgb & 0x0F;
1710                                        } else {
1711                                                // odd pixel position
1712                                                b = (rgb & 0xF0) >> 4;
1713                                                cnt++;
1714                                                rgb = buff[cnt];
1715                                                g = rgb & 0x0F;
1716                                                r = (rgb & 0xF0) >> 4;
1717                                                cnt++;
1718                                        }
1719
1720                                        // Yuv
1721                                        y[yPos] = ((b + (g << 1) + r) >> 2) - YUVoffset4;
1722                                        u[yPos] = r - g;
1723                                        v[yPos] = b - g;
1724                                        yPos++;
1725                                }
1726                                buff += pitch;
1727                        }
1728                }
1729                break;
1730        case ImageModeRGB16:
1731                {
1732                        ASSERT(m_header.channels == 3);
1733                        ASSERT(m_header.bpp == 16);
1734                        ASSERT(bpp == 16);
1735
1736                        DataT* y = m_channel[0]; ASSERT(y);
1737                        DataT* u = m_channel[1]; ASSERT(u);
1738                        DataT* v = m_channel[2]; ASSERT(v);
1739
1740                        UINT16 *buff16 = (UINT16 *)buff;
1741                        UINT16 rgb, b, g, r;
1742                        const int pitch16 = pitch/2;
1743
1744                        for (UINT32 h=0; h < m_header.height; h++) {
1745                                if (cb) {
1746                                        if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
1747                                        percent += dP;
1748                                }
1749                                for (UINT32 w=0; w < m_header.width; w++) {
1750                                        rgb = buff16[w];
1751                                        r = (rgb & 0xF800) >> 10;       // highest 5 bits
1752                                        g = (rgb & 0x07E0) >> 5;        // middle 6 bits
1753                                        b = (rgb & 0x001F) << 1;        // lowest 5 bits
1754                                        // Yuv
1755                                        y[yPos] = ((b + (g << 1) + r) >> 2) - YUVoffset6;
1756                                        u[yPos] = r - g;
```

```
1757                                            v[yPos] = b - g;
1758                                            yPos++;
1759                                    }
1760
1761                                    buff16 += pitch16;
1762                            }
1763                    }
1764                    break;
1765            default:
1766                    ASSERT(false);
1767            }
1768 }
```

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.

```
466 { 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.

```
272 { 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 1362 of file PGFimage.cpp.

```
1362
{
1363            if (iFirstColor + nColors > ColorTableLen)      ReturnWithError(ColorTableError);
1364
1365            for (UINT32 i=iFirstColor, j=0; j < nColors; i++, j++) {
1366                    m_postHeader.clut[i] = prgbColors[j];
1367            }
1368 }
```

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 892 of file PGFimage.cpp.

```
  892
{
  893         ASSERT(!m_decoder);     // current image must be closed
  894         ASSERT(header.quality <= MaxQuality);
  895         ASSERT(userDataLength <= MaxUserDataSize);
  896
  897         // init state
  898 #ifdef __PGFROISUPPORT__
  899         m_streamReinitialized = false;
  900 #endif
  901
  902         // init preHeader
  903         memcpy(m_preHeader.magic, PGFMagic, 3);
  904         m_preHeader.version = PGFVersion | flags;
  905         m_preHeader.hSize = HeaderSize;
  906
  907         // copy header
  908         memcpy(&m_header, &header, HeaderSize);
  909
  910         // check quality
  911         if (m_header.quality > MaxQuality) m_header.quality = MaxQuality;
  912
  913         // complete header
  914         CompleteHeader();
  915
  916         // check and set number of levels
  917         ComputeLevels();
  918
  919         // check for downsample
  920         if (m_header.quality > DownsampleThreshold &&  (m_header.mode == ImageModeRGBColor
||
  921
m_header.mode == ImageModeRGBA ||
  922
m_header.mode == ImageModeRGB48 ||
  923
m_header.mode == ImageModeCMYKColor ||
  924
m_header.mode == ImageModeCMYK64 ||
  925
m_header.mode == ImageModeLabColor ||
  926
m_header.mode == ImageModeLab48)) {
  927                 m_downsample = true;
  928                 m_quant = m_header.quality - 1;
  929         } else {
  930                 m_downsample = false;
  931                 m_quant = m_header.quality;
  932         }
  933
  934         // update header size and copy user data
  935         if (m_header.mode == ImageModeIndexedColor) {
  936                 // update header size
  937                 m_preHeader.hSize += ColorTableSize;
  938         }
  939         if (userDataLength && userData) {
  940                 if (userDataLength > MaxUserDataSize) userDataLength = MaxUserDataSize;
  941                 m_postHeader.userData = new(std::nothrow) UINT8[userDataLength];
  942                 if (!m_postHeader.userData) ReturnWithError(InsufficientMemory);
  943                 m_postHeader.userDataLen = m_postHeader.cachedUserDataLen =
userDataLength;
  944                 memcpy(m_postHeader.userData, userData, userDataLength);
  945                 // update header size
  946                 m_preHeader.hSize += userDataLength;
  947         }
```

```
948
949         // allocate channels
950         for (int i=0; i < m_header.channels; i++) {
951                 // set current width and height
952                 m_width[i] = m_header.width;
953                 m_height[i] = m_header.height;
954
955                 // allocate channels
956                 ASSERT(!m_channel[i]);
957                 m_channel[i] = new(std::nothrow) DataT[m_header.width*m_header.height];
958                 if (!m_channel[i]) {
959                         if (i) i--;
960                         while(i) {
961                                 delete[] m_channel[i]; m_channel[i] = 0;
962                                 i--;
963                         }
964                         ReturnWithError(InsufficientMemory);
965                 }
966         }
967 }
```

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 736 of file PGFimage.cpp.

```
736                                             {
737         const BYTE bpc = m_header.bpp/m_header.channels;
738         BYTE pot = 0;
739
740         while(maxValue > 0) {
741                 pot++;
742                 maxValue >>= 1;
743         }
744         // store bits per channel
745         if (pot > bpc) pot = bpc;
746         if (pot > 31) pot = 31;
747         m_header.usedBitsPerChannel = pot;
748 }
```

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.

```
296 { 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.

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

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

UINT32 CPGFImage::UpdatePostHeaderSize ()[private]

Definition at line 1122 of file PGFimage.cpp.

```
1122                                        {
1123         ASSERT(m_encoder);
1124
1125         INT64 offset = m_encoder->ComputeOffset(); ASSERT(offset >= 0);
1126
1127         if (offset > 0) {
1128                 // update post-header size and rewrite pre-header
1129                 m_preHeader.hSize += (UINT32)offset;
1130                 m_encoder->UpdatePostHeaderSize(m_preHeader);
1131         }
1132
1133         // write dummy levelLength into stream
1134         return m_encoder->WriteLevelLength(m_levelLength);
1135 }
```

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 754 of file PGFimage.cpp.

```
 754                                          {
 755         const BYTE bpc = m_header.bpp/m_header.channels;
 756
 757         if (bpc > 8) {
 758                 return m_header.usedBitsPerChannel;
 759         } else {
 760                 return bpc;
 761         }
 762 }
```

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.

```
  484 { 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.

```
  413 { 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 1219 of file PGFimage.cpp.

```
 1219
{
 1220         ASSERT(stream);
 1221         ASSERT(m_preHeader.hSize);
 1222
 1223         // create wavelet transform channels and encoder
 1224         UINT32 nBytes = WriteHeader(stream);
 1225
 1226         // write image
 1227         nBytes += WriteImage(stream, cb, data);
 1228
 1229         // return written bytes
 1230         if (nWrittenBytes) *nWrittenBytes += nBytes;
 1231 }
```

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 977 of file PGFimage.cpp.

```
  977                                                 {
  978         ASSERT(m_header.nLevels <= MaxLevel);
  979         ASSERT(m_header.quality <= MaxQuality); // quality is already initialized
  980
  981         if (m_header.nLevels > 0) {
  982                 volatile OSError error = NoError; // volatile prevents optimizations
  983                 // create new wt channels
  984 #ifdef LIBPGF_USE_OPENMP
  985                 #pragma omp parallel for default(shared)
  986 #endif
  987                 for (int i=0; i < m_header.channels; i++) {
  988                         DataT *temp = nullptr;
  989                         if (error == NoError) {
  990                                 if (m_wtChannel[i]) {
  991                                         ASSERT(m_channel[i]);
  992                                         // copy m_channel to temp
  993                                         int size = m_height[i]*m_width[i];
  994                                         temp = new(std::nothrow) DataT[size];
  995                                         if (temp) {
  996                                                 memcpy(temp, m_channel[i],
size*DataTSize);
  997                                                 delete m_wtChannel[i];  // also deletes
m_channel
  998                                                 m_channel[i] = nullptr;
  999                                         } else {
 1000                                                 error = InsufficientMemory;
 1001                                         }
 1002                                 }
 1003                                 if (error == NoError) {
 1004                                         if (temp) {
 1005                                                 ASSERT(!m_channel[i]);
 1006                                                 m_channel[i] = temp;
 1007                                         }
 1008                                         m_wtChannel[i] = new
CWaveletTransform(m_width[i], m_height[i], m_header.nLevels, m_channel[i]);
 1009                                         if (m_wtChannel[i]) {
 1010                                         #ifdef __PGFROISUPPORT__
 1011                                                 m_wtChannel[i]->SetROI(PGFRect(0, 0,
m_width[i], m_height[i]));
 1012                                         #endif
 1013
 1014                                                 // wavelet subband decomposition
 1015                                                 for (int l=0; error == NoError && l <
m_header.nLevels; l++) {
 1016                                                         OSError err =
m_wtChannel[i]->ForwardTransform(l, m_quant);
 1017                                                         if (err != NoError) error = err;
 1018                                                 }
 1019                                         } else {
 1020                                                 delete[] m_channel[i];
 1021                                                 error = InsufficientMemory;
 1022                                         }
```

```
1023                                    }
1024                            }
1025                    }
1026                    if (error != NoError) {
1027                            // free already allocated memory
1028                            for (int i=0; i < m_header.channels; i++) {
1029                                    delete m_wtChannel[i];
1030                            }
1031                            ReturnWithError(error);
1032                    }
1033
1034                    m_currentLevel = m_header.nLevels;
1035
1036                    // create encoder, write headers and user data, but not level-length area
1037                    m_encoder = new CEncoder(stream, m_preHeader, m_header, m_postHeader,
m_userDataPos, m_useOMPinEncoder);
1038                    if (m_favorSpeedOverSize) m_encoder->FavorSpeedOverSize();
1039
1040            #ifdef __PGFROISUPPORT__
1041                    if (ROIisSupported()) {
1042                            // new encoding scheme supporting ROI
1043                            m_encoder->SetROI();
1044                    }
1045            #endif
1046
1047            } else {
1048                    // very small image: we don't use DWT and encoding
1049
1050                    // create encoder, write headers and user data, but not level-length area
1051                    m_encoder = new CEncoder(stream, m_preHeader, m_header, m_postHeader,
m_userDataPos, m_useOMPinEncoder);
1052            }
1053
1054            INT64 nBytes = m_encoder->ComputeHeaderLength();
1055            return (nBytes > 0) ? (UINT32)nBytes : 0;
1056 }
```

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 1148 of file PGFimage.cpp.

```
1148                                                                                    {
1149            ASSERT(stream);
1150            ASSERT(m_preHeader.hSize);
1151
1152            int levels = m_header.nLevels;
1153            double percent = pow(0.25, levels);
1154
1155            // update post-header size, rewrite pre-header, and write dummy levelLength
1156            UINT32 nWrittenBytes = UpdatePostHeaderSize();
1157
1158            if (levels == 0) {
1159                    // for very small images: write channels uncoded
1160                    for (int c=0; c < m_header.channels; c++) {
1161                            const UINT32 size = m_width[c]*m_height[c];
```

```
 1162
 1163                           // write channel data into stream
 1164                           for (UINT32 i=0; i < size; i++) {
 1165                                   int count = DataTSize;
 1166                                   stream->Write(&count, &m_channel[c][i]);
 1167                           }
 1168                   }
 1169
 1170                   // now update progress
 1171                   if (cb) {
 1172                           if ((*cb)(1, true, data)) ReturnWithError(EscapePressed);
 1173                   }
 1174
 1175           } else {
 1176                   // encode quantized wavelet coefficients and write to PGF file
 1177                   // encode subbands, higher levels first
 1178                   // color channels are interleaved
 1179
 1180                   // encode all levels
 1181                   for (m_currentLevel = levels; m_currentLevel > 0; ) {
 1182                           WriteLevel(); // decrements m_currentLevel
 1183
 1184                           // now update progress
 1185                           if (cb) {
 1186                                   percent *= 4;
 1187                                   if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
 1188                           }
 1189                   }
 1190
 1191                   // flush encoder and write level lengths
 1192                   m_encoder->Flush();
 1193           }
 1194
 1195           // update level lengths
 1196           nWrittenBytes += m_encoder->UpdateLevelLength(); // return written image bytes
 1197
 1198           // delete encoder
 1199           delete m_encoder; m_encoder = nullptr;
 1200
 1201           ASSERT(!m_encoder);
 1202
 1203           return nWrittenBytes;
 1204 }
```

void CPGFImage::WriteLevel ()[private]

Definition at line 1066 of file PGFimage.cpp.

```
 1066                             {
 1067           ASSERT(m_encoder);
 1068           ASSERT(m_currentLevel > 0);
 1069           ASSERT(m_header.nLevels > 0);
 1070
 1071 #ifdef __PGFROISUPPORT__
 1072           if (ROIisSupported()) {
 1073                   const int lastChannel = m_header.channels - 1;
 1074
 1075                   for (int i=0; i < m_header.channels; i++) {
 1076                           // get number of tiles and tile indices
 1077                           const UINT32 nTiles =
m_wtChannel[i]->GetNofTiles(m_currentLevel);
 1078                           const UINT32 lastTile = nTiles - 1;
 1079
 1080                           if (m_currentLevel == m_header.nLevels) {
 1081                                   // last level also has LL band
 1082                                   ASSERT(nTiles == 1);
 1083                                   m_wtChannel[i]->GetSubband(m_currentLevel,
LL)->ExtractTile(*m_encoder);
```

```
 1084                                    m_encoder->EncodeTileBuffer(); // encode macro block with
tile-end = true
 1085                            }
 1086                            for (UINT32 tileY=0; tileY < nTiles; tileY++) {
 1087                                    for (UINT32 tileX=0; tileX < nTiles; tileX++) {
 1088                                            // extract tile to macro block and encode already
filled macro blocks with tile-end = false
 1089                                            m_wtChannel[i]->GetSubband(m_currentLevel,
HL)->ExtractTile(*m_encoder, true, tileX, tileY);
 1090                                            m_wtChannel[i]->GetSubband(m_currentLevel,
LH)->ExtractTile(*m_encoder, true, tileX, tileY);
 1091                                            m_wtChannel[i]->GetSubband(m_currentLevel,
HH)->ExtractTile(*m_encoder, true, tileX, tileY);
 1092                                            if (i == lastChannel && tileY == lastTile && tileX
== lastTile) {
 1093                                                    // all necessary data are buffered. next
call of EncodeTileBuffer will write the last piece of data of the current level.
 1094
m_encoder->SetEncodedLevel(--m_currentLevel);
 1095                                            }
 1096                                            m_encoder->EncodeTileBuffer(); // encode last
macro block with tile-end = true
 1097                                    }
 1098                            }
 1099                    }
 1100            } else
 1101 #endif
 1102            {
 1103                    for (int i=0; i < m_header.channels; i++) {
 1104                            ASSERT(m_wtChannel[i]);
 1105                            if (m_currentLevel == m_header.nLevels) {
 1106                                    // last level also has LL band
 1107                                    m_wtChannel[i]->GetSubband(m_currentLevel,
LL)->ExtractTile(*m_encoder);
 1108                            }
 1109                            //encoder.EncodeInterleaved(m_wtChannel[i], m_currentLevel,
m_quant); // until version 4
 1110                            m_wtChannel[i]->GetSubband(m_currentLevel,
HL)->ExtractTile(*m_encoder); // since version 5
 1111                            m_wtChannel[i]->GetSubband(m_currentLevel,
LH)->ExtractTile(*m_encoder); // since version 5
 1112                            m_wtChannel[i]->GetSubband(m_currentLevel,
HH)->ExtractTile(*m_encoder);
 1113                    }
 1114
 1115                    // all necessary data are buffered. next call of EncodeBuffer will write
the last piece of data of the current level.
 1116                    m_encoder->SetEncodedLevel(--m_currentLevel);
 1117            }
 1118 }
```

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:

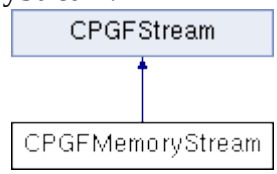
- **PGFimage.h**
- **PGFimage.cpp**

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.

```
  79 : m_size(size)
  80 , m_allocated(true) {
  81         m_buffer = m_pos = m_eos = new(std::nothrow) UINT8[m_size];
  82         if (!m_buffer) ReturnWithError(InsufficientMemory);
  83 }
```

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.

```
  90 : m_buffer(pBuffer)
  91 , m_pos(pBuffer)
  92 , m_eos(pBuffer + size)
  93 , m_size(size)
  94 , m_allocated(false) {
  95         ASSERT(IsValid());
  96 }
```

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

Definition at line 128 of file PGFstream.h.

```
 128                                     {
 129                 m_pos = 0;
 130                 if (m_allocated) {
 131                         // the memory buffer has been allocated inside of CPMFmemoryStream
constructor
 132                         delete[] m_buffer; m_buffer = 0;
 133                 }
 134         }
```

Member Function Documentation

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

Returns:

Memory buffer

Definition at line 145 of file PGFstream.h.

```
145 { return m_buffer; }
```

UINT8* CPGFMemoryStream::GetBuffer () [inline]

Returns:

Memory buffer

Definition at line 147 of file PGFstream.h.

```
147 { 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.

```
149 { 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.112*).

Definition at line 139 of file PGFstream.h.

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

size_t CPGFMemoryStream::GetSize () const [inline]

Returns:

Memory size

Definition at line 143 of file PGFstream.h.

```
143 { 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.112*).

Definition at line 140 of file PGFstream.h.

```
140 { 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.112*).

Definition at line 148 of file PGFstream.cpp.

```
148                                                         {
149         ASSERT(IsValid());
150         ASSERT(count);
151         ASSERT(buffPtr);
152         ASSERT(m_buffer + m_size >= m_eos);
153         ASSERT(m_pos <= m_eos);
154 
155         if (m_pos + *count <= m_eos) {
156                 memcpy(buffPtr, m_pos, *count);
157                 m_pos += *count;
158         } else {
159                 // end of memory block reached -> read only until end
160                 *count = (int)__max(0, m_eos - m_pos);
161                 memcpy(buffPtr, m_pos, *count);
162                 m_pos += *count;
163         }
164         ASSERT(m_pos <= m_eos);
165 }
```

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.

```
102                                                                 {
103         if (!m_allocated) {
104                 m_buffer = m_pos = pBuffer;
105                 m_size = size;
106                 m_eos = m_buffer + size;
107         }
108 }
```

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

Parameters:

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

Definition at line 151 of file PGFstream.h.

```
151 { 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.112*).

Definition at line 168 of file PGFstream.cpp.

```
168                                                         {
```

```
169         ASSERT(IsValid());
170         switch(posMode) {
171         case FSFromStart:
172                 m_pos = m_buffer + posOff;
173                 break;
174         case FSFromCurrent:
175                 m_pos += posOff;
176                 break;
177         case FSFromEnd:
178                 m_pos = m_eos + posOff;
179                 break;
180         default:
181                 ASSERT(false);
182         }
183         if (m_pos > m_eos)
184                 ReturnWithError(InvalidStreamPos);
185 }
```

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.112*).

Definition at line 111 of file PGFstream.cpp.

```
111                                                     {
112         ASSERT(count);
113         ASSERT(buffPtr);
114         ASSERT(IsValid());
115         const size_t deltaSize = 0x4000 + *count;
116 
117         if (m_pos + *count <= m_buffer + m_size) {
118                 memcpy(m_pos, buffPtr, *count);
119                 m_pos += *count;
120                 if (m_pos > m_eos) m_eos = m_pos;
121         } else if (m_allocated) {
122                 // memory block is too small -> reallocate a deltaSize larger block
123                 size_t offset = m_pos - m_buffer;
124                 UINT8 *buf_tmp = (UINT8 *)realloc(m_buffer, m_size + deltaSize);
125                 if (!buf_tmp) {
126                         delete[] m_buffer;
127                         m_buffer = 0;
128                         ReturnWithError(InsufficientMemory);
129                 } else {
130                         m_buffer = buf_tmp;
131                 }
132                 m_size += deltaSize;
133 
134                 // reposition m_pos
135                 m_pos = m_buffer + offset;
136 
137                 // write block
138                 memcpy(m_pos, buffPtr, *count);
139                 m_pos += *count;
140                 if (m_pos > m_eos) m_eos = m_pos;
141         } else {
142                 ReturnWithError(InsufficientMemory);
143         }
144         ASSERT(m_pos <= m_eos);
145 }
```

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:

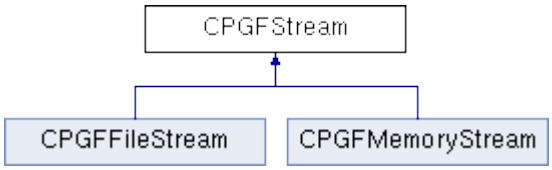
- **PGFstream.h**
- **PGFstream.cpp**

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.

```
43 {}
```

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

Standard destructor.

Definition at line 47 of file PGFstream.h.

```
47 {}
```

Member Function Documentation

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

Get current stream position.

Returns:

Current stream position

Implemented in **CPGFMemoryStream** (*p.107*), and **CPGFFileStream** (*p.47*).

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

Check stream validity.

Returns:

True if stream and current position is valid

Implemented in **CPGFMemoryStream** (*p.107*), and **CPGFFileStream** (*p.47*).

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 **CPGFMemoryStream** (*p.107*), and **CPGFFileStream** (*p.47*).

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 **CPGFMemoryStream** (*p.108*), and **CPGFFileStream** (*p.48*).

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 **CPGFMemoryStream** (*p.109*), and **CPGFFileStream** (*p.48*).

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.

```
36 : m_width(0)
37 , m_height(0)
38 , m_size(0)
39 , m_level(0)
40 , m_orientation(LL)
41 , m_data(0)
42 , m_dataPos(0)
43 #ifdef __PGFROISUPPORT__
44 , m_nTiles(0)
45 #endif
46 {
47 }
```

CSubband::~CSubband ()

Destructor.

Definition at line 51 of file Subband.cpp.

```
51                      {
52         FreeMemory();
53 }
```

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.

```
  77                              {
  78         UINT32 oldSize = m_size;
  79
  80 #ifdef __PGFROISUPPORT__
  81         m_size = BufferWidth()*m_ROI.Height();
  82 #endif
  83         ASSERT(m_size > 0);
  84
  85         if (m_data) {
  86                 if (oldSize >= m_size) {
  87                         return true;
  88                 } else {
  89                         delete[] m_data;
  90                         m_data = new(std::nothrow) DataT[m_size];
  91                         return (m_data != 0);
  92                 }
  93         } else {
  94                 m_data = new(std::nothrow) DataT[m_size];
  95                 return (m_data != 0);
  96         }
  97 }
```

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.

```
 154                                     {
 155         if (m_orientation == LL) {
 156                 quantParam -= m_level + 1;
 157         } else if (m_orientation == HH) {
 158                 quantParam -= m_level - 1;
 159         } else {
 160                 quantParam -= m_level;
 161         }
 162         if (quantParam > 0) {
 163                 for (UINT32 i=0; i < m_size; i++) {
 164                         m_data[i] <<= quantParam;
 165                 }
 166         }
 167 }
```

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.

```
 177
{
 178 #ifdef __PGFROISUPPORT__
 179         if (tile) {
 180                 // compute tile position and size
```

```
181                 UINT32 xPos, yPos, w, h;
182                 TilePosition(tileX, tileY, xPos, yPos, w, h);
183
184                 // write values into buffer using partitiong scheme
185                 encoder.Partition(this, w, h, xPos + yPos*m_width, m_width);
186         } else
187 #endif
188         {
189                 tileX; tileY; tile; // prevents from unreferenced formal parameter warning
190                 // write values into buffer using partitiong scheme
191                 encoder.Partition(this, m_width, m_height, 0, m_width);
192         }
193 }
```

void CSubband::FreeMemory ()

Delete the memory buffer of this subband.

Definition at line 101 of file Subband.cpp.

```
101                             {
102         if (m_data) {
103                 delete[] m_data; m_data = 0;
104         }
105 }
```

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.

```
107 { return m_data; }
```

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

Definition at line 151 of file Subband.h.

```
151 { 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.

```
113 { 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.

```
123 { 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.

```
118 { 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.

```
135 { 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.

```
128 { return m_width; }
```

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

Definition at line 162 of file Subband.h.

```
162 { m_dataPos = 0; }
```

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

Definition at line 57 of file Subband.cpp.

```
 57                                                                      {
 58         m_width = width;
 59         m_height = height;
 60         m_size = m_width*m_height;
 61         m_level = level;
 62         m_orientation = orient;
 63         m_data = 0;
 64         m_dataPos = 0;
 65 #ifdef __PGFROISUPPORT__
 66         m_ROI.left = 0;
 67         m_ROI.top = 0;
 68         m_ROI.right = m_width;
 69         m_ROI.bottom = m_height;
 70         m_nTiles = 0;
 71 #endif
 72 }
```

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.

```
  203
{
  204         // allocate memory
  205         if (!AllocMemory()) ReturnWithError(InsufficientMemory);
  206
  207         // correct quantParam with normalization factor
  208         if (m_orientation == LL) {
  209                 quantParam -= m_level + 1;
  210         } else if (m_orientation == HH) {
  211                 quantParam -= m_level - 1;
  212         } else {
  213                 quantParam -= m_level;
  214         }
  215         if (quantParam < 0) quantParam = 0;
  216
  217 #ifdef __PGFROISUPPORT__
  218         if (tile) {
  219                 UINT32 xPos, yPos, w, h;
  220
  221                 // compute tile position and size
  222                 TilePosition(tileX, tileY, xPos, yPos, w, h);
  223
  224                 ASSERT(xPos >= m_ROI.left && yPos >= m_ROI.top);
  225                 decoder.Partition(this, quantParam, w, h, (xPos - m_ROI.left) + (yPos -
m_ROI.top)*BufferWidth(), BufferWidth());
  226         } else
  227 #endif
  228         {
  229                 tileX; tileY; tile; // prevents from unreferenced formal parameter warning
  230                 // read values into buffer using partitiong scheme
  231                 decoder.Partition(this, quantParam, m_width, m_height, 0, m_width);
  232         }
  233 }
```

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.

```
  112                                     {
  113         if (m_orientation == LL) {
  114                 quantParam -= (m_level + 1);
  115                 // uniform rounding quantization
  116                 if (quantParam > 0) {
  117                         quantParam--;
  118                         for (UINT32 i=0; i < m_size; i++) {
  119                                 if (m_data[i] < 0) {
  120                                         m_data[i] = -(((-m_data[i] >> quantParam) + 1) >>
1);
  121                                 } else {
  122                                         m_data[i] = ((m_data[i] >> quantParam) + 1) >> 1;
  123                                 }
  124                         }
  125                 }
```

```
126         } else {
127                 if (m_orientation == HH) {
128                         quantParam -= (m_level - 1);
129                 } else {
130                         quantParam -= m_level;
131                 }
132                 // uniform deadzone quantization
133                 if (quantParam > 0) {
134                         int threshold = ((1 << quantParam) * 7)/5;      // good value
135                         quantParam--;
136                         for (UINT32 i=0; i < m_size; i++) {
137                                 if (m_data[i] < -threshold) {
138                                         m_data[i] = -(((-m_data[i] >> quantParam) + 1) >>
1);
139                                 } else if (m_data[i] > threshold) {
140                                         m_data[i] = ((m_data[i] >> quantParam) + 1) >> 1;
141                                 } else {
142                                         m_data[i] = 0;
143                                 }
144                         }
145                 }
146         }
147 }
```

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

Definition at line 149 of file Subband.h.

```
149 { 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.

```
148 { 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.

```
102 { ASSERT(pos < m_size); m_data[pos] = v; }
```

void CSubband::WriteBuffer (DataT *val*)[inline], [private]

Definition at line 147 of file Subband.h.

```
147 { 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.

```
   41 : m_nLevels(levels + 1) // m_nLevels in CPGFImage determines the number of FWT steps;
this.m_nLevels determines the number subband-planes
   42 , m_subband(nullptr)
   43 , m_indices(nullptr)
   44 {
   45         ASSERT(m_nLevels > 0 && m_nLevels <= MaxLevel + 1);
   46         InitSubbands(width, height, data);
   47 }
```

CWaveletTransform::~CWaveletTransform ()[inline]

Destructor.

Definition at line 69 of file WaveletTransform.h.

```
   69 { Destroy(); }
```

Member Function Documentation

void CWaveletTransform::Destroy ()[inline], [private]

Definition at line 125 of file WaveletTransform.h.

```
  125                         {
  126                 delete[] m_subband; m_subband = nullptr;
  127         #ifdef __PGFROISUPPORT__
  128                 delete[] m_indices; m_indices = nullptr;
  129         #endif
  130         }
```

void CWaveletTransform::ForwardRow (DataT * *buff*, UINT32 *width*)[private]

Definition at line 178 of file WaveletTransform.cpp.

```
  178                                                             {
  179         if (width >= FilterSize) {
  180                 UINT32 i = 3;
  181
  182                 // left border handling
  183                 src[1] -= ((src[0] + src[2] + c1) >> 1); // high pass
  184                 src[0] += ((src[1] + c1) >> 1); // low pass
  185
  186                 // middle part
  187                 for (; i < width-1; i += 2) {
  188                         src[i] -= ((src[i-1] + src[i+1] + c1) >> 1); // high pass
  189                         src[i-1] += ((src[i-2] + src[i] + c2) >> 2); // low pass
  190                 }
  191
  192                 // right border handling
  193                 if (width & 1) {
  194                         src[i-1] += ((src[i-2] + c1) >> 1); // low pass
  195                 } else {
  196                         src[i] -= src[i-1]; // high pass
  197                         src[i-1] += ((src[i-2] + src[i] + c2) >> 2); // low pass
  198                 }
```

```
  199         }
  200 }
```

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 86 of file WaveletTransform.cpp.

```
   86                                                                      {
   87         ASSERT(level >= 0 && level < m_nLevels - 1);
   88         const int destLevel = level + 1;
   89         ASSERT(m_subband[destLevel]);
   90         CSubband* srcBand = &m_subband[level][LL]; ASSERT(srcBand);
   91         const UINT32 width = srcBand->GetWidth();
   92         const UINT32 height = srcBand->GetHeight();
   93         DataT* src = srcBand->GetBuffer(); ASSERT(src);
   94         DataT *row0, *row1, *row2, *row3;
   95
   96         // Allocate memory for next transform level
   97         for (int i=0; i < NSubbands; i++) {
   98                 if (!m_subband[destLevel][i].AllocMemory()) return InsufficientMemory;
   99         }
  100
  101         if (height >= FilterSize) { // changed from FilterSizeH to FilterSize
  102                 // top border handling
  103                 row0 = src; row1 = row0 + width; row2 = row1 + width;
  104                 ForwardRow(row0, width);
  105                 ForwardRow(row1, width);
  106                 ForwardRow(row2, width);
  107                 for (UINT32 k=0; k < width; k++) {
  108                         row1[k] -= ((row0[k] + row2[k] + c1) >> 1); // high pass
  109                         row0[k] += ((row1[k] + c1) >> 1); // low pass
  110                 }
  111                 InterleavedToSubbands(destLevel, row0, row1, width);
  112                 row0 = row1; row1 = row2; row2 += width; row3 = row2 + width;
  113
  114                 // middle part
  115                 for (UINT32 i=3; i < height-1; i += 2) {
  116                         ForwardRow(row2, width);
  117                         ForwardRow(row3, width);
  118                         for (UINT32 k=0; k < width; k++) {
  119                                 row2[k] -= ((row1[k] + row3[k] + c1) >> 1); // high pass
filter
  120                                 row1[k] += ((row0[k] + row2[k] + c2) >> 2); // low pass
filter
  121                         }
  122                         InterleavedToSubbands(destLevel, row1, row2, width);
  123                         row0 = row2; row1 = row3; row2 = row3 + width; row3 = row2 + width;
  124                 }
  125
  126                 // bottom border handling
  127                 if (height & 1) {
  128                         for (UINT32 k=0; k < width; k++) {
  129                                 row1[k] += ((row0[k] + c1) >> 1); // low pass
  130                         }
  131                         InterleavedToSubbands(destLevel, row1, nullptr, width);
  132                         row0 = row1; row1 += width;
  133                 } else {
  134                         ForwardRow(row2, width);
  135                         for (UINT32 k=0; k < width; k++) {
  136                                 row2[k] -= row1[k]; // high pass
```

```
137                                 row1[k] += ((row0[k] + row2[k] + c2) >> 2); // low pass
138                         }
139                         InterleavedToSubbands(destLevel, row1, row2, width);
140                         row0 = row1; row1 = row2; row2 += width;
141                 }
142         } else {
143                 // if height is too small
144                 row0 = src; row1 = row0 + width;
145                 // first part
146                 for (UINT32 k=0; k < height; k += 2) {
147                         ForwardRow(row0, width);
148                         ForwardRow(row1, width);
149                         InterleavedToSubbands(destLevel, row0, row1, width);
150                         row0 += width << 1; row1 += width << 1;
151                 }
152                 // bottom
153                 if (height & 1) {
154                         InterleavedToSubbands(destLevel, row0, nullptr, width);
155                 }
156         }
157
158         if (quant > 0) {
159                 // subband quantization (without LL)
160                 for (int i=1; i < NSubbands; i++) {
161                         m_subband[destLevel][i].Quantize(quant);
162                 }
163                 // LL subband quantization
164                 if (destLevel == m_nLevels - 1) {
165                         m_subband[destLevel][LL].Quantize(quant);
166                 }
167         }
168
169         // free source band
170         srcBand->FreeMemory();
171         return NoError;
172 }
```

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.

```
93                                                                      {
94                 ASSERT(level >= 0 && level < m_nLevels);
95                 return &m_subband[level][orientation];
96         }
```

void CWaveletTransform::InitSubbands (UINT32 *width*, UINT32 *height*, DataT * *data*)`[private]`

Definition at line 51 of file WaveletTransform.cpp.

```
51                                                                                      {
52         if (m_subband) Destroy();
53
54         // create subbands
55         m_subband = new CSubband[m_nLevels][NSubbands];
56
57         // init subbands
58         UINT32 loWidth = width;
59         UINT32 hiWidth = width;
60         UINT32 loHeight = height;
61         UINT32 hiHeight = height;
62
```

```
  63         for (int level = 0; level < m_nLevels; level++) {
  64                 m_subband[level][LL].Initialize(loWidth, loHeight, level, LL);  // LL
  65                 m_subband[level][HL].Initialize(hiWidth, loHeight, level, HL);  //    HL
  66                 m_subband[level][LH].Initialize(loWidth, hiHeight, level, LH);  // LH
  67                 m_subband[level][HH].Initialize(hiWidth, hiHeight, level, HH);  //    HH
  68                 hiWidth = loWidth >> 1;                 hiHeight = loHeight >> 1;
  69                 loWidth = (loWidth + 1) >> 1;   loHeight = (loHeight + 1) >> 1;
  70         }
  71         if (data) {
  72                 m_subband[0][LL].SetBuffer(data);
  73         }
  74 }
```

void CWaveletTransform::InterleavedToSubbands (int *destLevel*, DataT * *loRow*, DataT * *hiRow*, UINT32 *width*)[private]

Definition at line 204 of file WaveletTransform.cpp.

```
  204
{
  205         const UINT32 wquot = width >> 1;
  206         const bool wrem = (width & 1);
  207         CSubband &ll = m_subband[destLevel][LL], &hl = m_subband[destLevel][HL];
  208         CSubband &lh = m_subband[destLevel][LH], &hh = m_subband[destLevel][HH];
  209
  210         if (hiRow) {
  211                 for (UINT32 i=0; i < wquot; i++) {
  212                         ll.WriteBuffer(*loRow++);       // first access, than increment
  213                         hl.WriteBuffer(*loRow++);
  214                         lh.WriteBuffer(*hiRow++);       // first access, than increment
  215                         hh.WriteBuffer(*hiRow++);
  216                 }
  217                 if (wrem) {
  218                         ll.WriteBuffer(*loRow);
  219                         lh.WriteBuffer(*hiRow);
  220                 }
  221         } else {
  222                 for (UINT32 i=0; i < wquot; i++) {
  223                         ll.WriteBuffer(*loRow++);       // first access, than increment
  224                         hl.WriteBuffer(*loRow++);
  225                 }
  226                 if (wrem) ll.WriteBuffer(*loRow);
  227         }
  228 }
```

void CWaveletTransform::InverseRow (DataT * *buff*, UINT32 *width*)[private]

Definition at line 417 of file WaveletTransform.cpp.

```
  417                                                     {
  418         if (width >= FilterSize) {
  419                 UINT32 i = 2;
  420
  421                 // left border handling
  422                 dest[0] -= ((dest[1] + c1) >> 1); // even
  423
  424                 // middle part
  425                 for (; i < width - 1; i += 2) {
  426                         dest[i] -= ((dest[i-1] + dest[i+1] + c2) >> 2); // even
  427                         dest[i-1] += ((dest[i-2] + dest[i] + c1) >> 1); // odd
  428                 }
  429
  430                 // right border handling
  431                 if (width & 1) {
  432                         dest[i] -= ((dest[i-1] + c1) >> 1); // even
  433                         dest[i-1] += ((dest[i-2] + dest[i] + c1) >> 1); // odd
  434                 } else {
  435                         dest[i-1] += dest[i-2]; // odd
```

```
436                 }
437         }
438 }
```

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 243 of file WaveletTransform.cpp.

```
243
{
244         ASSERT(srcLevel > 0 && srcLevel < m_nLevels);
245         const int destLevel = srcLevel - 1;
246         ASSERT(m_subband[destLevel]);
247         CSubband* destBand = &m_subband[destLevel][LL];
248         UINT32 width, height;
249
250         // allocate memory for the results of the inverse transform
251         if (!destBand->AllocMemory()) return InsufficientMemory;
252         DataT *origin = destBand->GetBuffer(), *row0, *row1, *row2, *row3;
253
254 #ifdef __PGFROISUPPORT__
255         PGFRect destROI = destBand->GetAlignedROI();
256         const UINT32 destWidth  = destROI.Width();  // destination buffer width
257         const UINT32 destHeight = destROI.Height(); // destination buffer height
258         width = destWidth;              // destination working width
259         height = destHeight;    // destination working height
260
261         // update destination ROI
262         if (destROI.top & 1) {
263                 destROI.top++;
264                 origin += destWidth;
265                 height--;
266         }
267         if (destROI.left & 1) {
268                 destROI.left++;
269                 origin++;
270                 width--;
271         }
272
273         // init source buffer position
274         const UINT32 leftD = destROI.left >> 1;
275         const UINT32 left0 = m_subband[srcLevel][LL].GetAlignedROI().left;
276         const UINT32 left1 = m_subband[srcLevel][HL].GetAlignedROI().left;
277         const UINT32 topD = destROI.top >> 1;
278         const UINT32 top0 = m_subband[srcLevel][LL].GetAlignedROI().top;
279         const UINT32 top1 = m_subband[srcLevel][LH].GetAlignedROI().top;
280         ASSERT(m_subband[srcLevel][LH].GetAlignedROI().left == left0);
281         ASSERT(m_subband[srcLevel][HH].GetAlignedROI().left == left1);
282         ASSERT(m_subband[srcLevel][HL].GetAlignedROI().top == top0);
283         ASSERT(m_subband[srcLevel][HH].GetAlignedROI().top == top1);
284
285         UINT32 srcOffsetX[2] = { 0, 0 };
286         UINT32 srcOffsetY[2] = { 0, 0 };
287
288         if (leftD >= __max(left0, left1)) {
289                 srcOffsetX[0] = leftD - left0;
```

```
290                 srcOffsetX[1] = leftD - left1;
291         } else {
292                 if (left0 <= left1) {
293                         const UINT32 dx = (left1 - leftD) << 1;
294                         destROI.left += dx;
295                         origin += dx;
296                         width -= dx;
297                         srcOffsetX[0] = left1 - left0;
298                 } else {
299                         const UINT32 dx = (left0 - leftD) << 1;
300                         destROI.left += dx;
301                         origin += dx;
302                         width -= dx;
303                         srcOffsetX[1] = left0 - left1;
304                 }
305         }
306         if (topD >= __max(top0, top1)) {
307                 srcOffsetY[0] = topD - top0;
308                 srcOffsetY[1] = topD - top1;
309         } else {
310                 if (top0 <= top1) {
311                         const UINT32 dy = (top1 - topD) << 1;
312                         destROI.top += dy;
313                         origin += dy*destWidth;
314                         height -= dy;
315                         srcOffsetY[0] = top1 - top0;
316                 } else {
317                         const UINT32 dy = (top0 - topD) << 1;
318                         destROI.top += dy;
319                         origin += dy*destWidth;
320                         height -= dy;
321                         srcOffsetY[1] = top0 - top1;
322                 }
323         }
324
325         m_subband[srcLevel][LL].InitBuffPos(srcOffsetX[0], srcOffsetY[0]);
326         m_subband[srcLevel][HL].InitBuffPos(srcOffsetX[1], srcOffsetY[0]);
327         m_subband[srcLevel][LH].InitBuffPos(srcOffsetX[0], srcOffsetY[1]);
328         m_subband[srcLevel][HH].InitBuffPos(srcOffsetX[1], srcOffsetY[1]);
329
330 #else
331         width = destBand->GetWidth();
332         height = destBand->GetHeight();
333         PGFRect destROI(0, 0, width, height);
334         const UINT32 destWidth = width; // destination buffer width
335         const UINT32 destHeight = height; // destination buffer height
336
337         // init source buffer position
338         for (int i = 0; i < NSubbands; i++) {
339                 m_subband[srcLevel][i].InitBuffPos();
340         }
341 #endif
342
343         if (destHeight >= FilterSize) { // changed from FilterSizeH to FilterSize
344                 // top border handling
345                 row0 = origin; row1 = row0 + destWidth;
346                 SubbandsToInterleaved(srcLevel, row0, row1, width);
347                 for (UINT32 k = 0; k < width; k++) {
348                         row0[k] -= ((row1[k] + c1) >> 1); // even
349                 }
350
351                 // middle part
352                 row2 = row1 + destWidth; row3 = row2 + destWidth;
353                 for (UINT32 i = destROI.top + 2; i < destROI.bottom - 1; i += 2) {
354                         SubbandsToInterleaved(srcLevel, row2, row3, width);
355                         for (UINT32 k = 0; k < width; k++) {
356                                 row2[k] -= ((row1[k] + row3[k] + c2) >> 2); // even
357                                 row1[k] += ((row0[k] + row2[k] + c1) >> 1); // odd
358                         }
359                         InverseRow(row0, width);
360                         InverseRow(row1, width);
```

```
  361                         row0 = row2; row1 = row3; row2 = row1 + destWidth; row3 = row2 +
destWidth;
  362                 }
  363
  364                 // bottom border handling
  365                 if (height & 1) {
  366                         SubbandsToInterleaved(srcLevel, row2, nullptr, width);
  367                         for (UINT32 k = 0; k < width; k++) {
  368                                 row2[k] -= ((row1[k] + c1) >> 1); // even
  369                                 row1[k] += ((row0[k] + row2[k] + c1) >> 1); // odd
  370                         }
  371                         InverseRow(row0, width);
  372                         InverseRow(row1, width);
  373                         InverseRow(row2, width);
  374                         row0 = row1; row1 = row2; row2 += destWidth;
  375                 } else {
  376                         for (UINT32 k = 0; k < width; k++) {
  377                                 row1[k] += row0[k];
  378                         }
  379                         InverseRow(row0, width);
  380                         InverseRow(row1, width);
  381                         row0 = row1; row1 += destWidth;
  382                 }
  383         } else {
  384                 // height is too small
  385                 row0 = origin; row1 = row0 + destWidth;
  386                 // first part
  387                 for (UINT32 k = 0; k < height; k += 2) {
  388                         SubbandsToInterleaved(srcLevel, row0, row1, width);
  389                         InverseRow(row0, width);
  390                         InverseRow(row1, width);
  391                         row0 += destWidth << 1; row1 += destWidth << 1;
  392                 }
  393                 // bottom
  394                 if (height & 1) {
  395                         SubbandsToInterleaved(srcLevel, row0, nullptr, width);
  396                         InverseRow(row0, width);
  397                 }
  398         }
  399
  400         // free memory of the current srcLevel
  401         for (int i = 0; i < NSubbands; i++) {
  402                 m_subband[srcLevel][i].FreeMemory();
  403         }
  404
  405         // return info
  406         *w = destWidth;
  407         *h = destHeight;
  408         *data = destBand->GetBuffer();
  409         return NoError;
  410 }
```

void CWaveletTransform::SubbandsToInterleaved (int *srcLevel*, DataT * *loRow*, DataT * *hiRow*, UINT32 *width*)`[private]`

Definition at line 442 of file WaveletTransform.cpp.

```
  442
{
  443         const UINT32 wquot = width >> 1;
  444         const bool wrem = (width & 1);
  445         CSubband &ll = m_subband[srcLevel][LL], &hl = m_subband[srcLevel][HL];
  446         CSubband &lh = m_subband[srcLevel][LH], &hh = m_subband[srcLevel][HH];
  447
  448         if (hiRow) {
  449         #ifdef __PGFROISUPPORT__
  450                 const bool storePos = wquot < ll.BufferWidth();
  451                 UINT32 llPos = 0, hlPos = 0, lhPos = 0, hhPos = 0;
  452
```

```
453                 if (storePos) {
454                         // save current src buffer positions
455                         llPos = ll.GetBuffPos();
456                         hlPos = hl.GetBuffPos();
457                         lhPos = lh.GetBuffPos();
458                         hhPos = hh.GetBuffPos();
459                 }
460         #endif
461
462                 for (UINT32 i=0; i < wquot; i++) {
463                         *loRow++ = ll.ReadBuffer();// first access, than increment
464                         *loRow++ = hl.ReadBuffer();// first access, than increment
465                         *hiRow++ = lh.ReadBuffer();// first access, than increment
466                         *hiRow++ = hh.ReadBuffer();// first access, than increment
467                 }
468
469                 if (wrem) {
470                         *loRow++ = ll.ReadBuffer();// first access, than increment
471                         *hiRow++ = lh.ReadBuffer();// first access, than increment
472                 }
473
474         #ifdef __PGFROISUPPORT__
475                 if (storePos) {
476                         // increment src buffer positions
477                         ll.IncBuffRow(llPos);
478                         hl.IncBuffRow(hlPos);
479                         lh.IncBuffRow(lhPos);
480                         hh.IncBuffRow(hhPos);
481                 }
482         #endif
483
484         } else {
485         #ifdef __PGFROISUPPORT__
486                 const bool storePos = wquot < ll.BufferWidth();
487                 UINT32 llPos = 0, hlPos = 0;
488
489                 if (storePos) {
490                         // save current src buffer positions
491                         llPos = ll.GetBuffPos();
492                         hlPos = hl.GetBuffPos();
493                 }
494         #endif
495
496                 for (UINT32 i=0; i < wquot; i++) {
497                         *loRow++ = ll.ReadBuffer();// first access, than increment
498                         *loRow++ = hl.ReadBuffer();// first access, than increment
499                 }
500                 if (wrem) *loRow++ = ll.ReadBuffer();
501
502         #ifdef __PGFROISUPPORT__
503                 if (storePos) {
504                         // increment src buffer positions
505                         ll.IncBuffRow(llPos);
506                         hl.IncBuffRow(hlPos);
507                 }
508         #endif
509         }
510 }
```

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 208 of file PGFtypes.h.

Constructor & Destructor Documentation

IOException::IOException ()[inline]

Standard constructor.

Definition at line 210 of file PGFtypes.h.

```
210 : error(NoError) {}
```

IOException::IOException (OSError *err*)[inline]

Constructor

Parameters:

err	Run-time error

Definition at line 213 of file PGFtypes.h.

```
213 : error(err) {}
```

Member Data Documentation

OSError IOException::error

operating system error code

Definition at line 215 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 150 of file PGFtypes.h.

Constructor & Destructor Documentation

PGFHeader::PGFHeader ()[inline]

Definition at line 151 of file PGFtypes.h.

```
  151 : 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 156 of file PGFtypes.h.

UINT8 PGFHeader::channels

number of channels
Definition at line 157 of file PGFtypes.h.

UINT32 PGFHeader::height

image height in pixels
Definition at line 153 of file PGFtypes.h.

UINT8 PGFHeader::mode

image mode according to Adobe's image modes
Definition at line 158 of file PGFtypes.h.

UINT8 PGFHeader::nLevels

number of FWT transforms
Definition at line 154 of file PGFtypes.h.

UINT8 PGFHeader::quality

quantization parameter: 0=lossless, 4=standard, 6=poor quality
Definition at line 155 of file PGFtypes.h.

UINT8 PGFHeader::usedBitsPerChannel

number of used bits per channel in 16- and 32-bit per channel modes
Definition at line 159 of file PGFtypes.h.

PGFVersionNumber PGFHeader::version

codec version number: (since Version 7)
Definition at line 160 of file PGFtypes.h.

UINT32 PGFHeader::width

image width in pixels

Definition at line 152 of file PGFtypes.h.

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

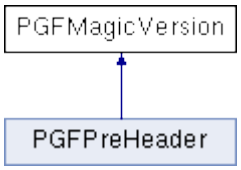
- **PGFtypes.h**

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 168 of file PGFtypes.h.

Member Data Documentation

UINT32 PGFPostHeader::cachedUserDataLen

cached user data size in bytes (not part of file header)

Definition at line 172 of file PGFtypes.h.

RGBQUAD PGFPostHeader::clut[ColorTableLen]

color table for indexed color images (optional part of file header)

Definition at line 169 of file PGFtypes.h.

UINT8* PGFPostHeader::userData

user data of size userDataLen (optional part of file header)

Definition at line 170 of file PGFtypes.h.

UINT32 PGFPostHeader::userDataLen

user data size in bytes (not part of file header)

Definition at line 171 of file PGFtypes.h.

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

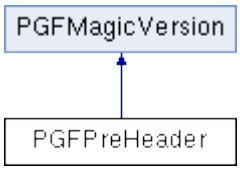
- **PGFtypes.h**

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 222 of file PGFtypes.h.

Constructor & Destructor Documentation

PGFRect::PGFRect ()[inline]

Standard constructor.

Definition at line 224 of file PGFtypes.h.

```
224 : 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 230 of file PGFtypes.h.

```
230 : left(x), top(y), right(x + width), bottom(y + height) {}
```

Member Function Documentation

UINT32 PGFRect::Height () const[inline]

Returns:

Rectangle height

Definition at line 250 of file PGFtypes.h.

```
250 { 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 256 of file PGFtypes.h.

```
256 { return (x >= left && x < right && y >= top && y < bottom); }
```

UINT32 PGFRect::Width () const[inline]

Returns:

Rectangle width

Definition at line 248 of file PGFtypes.h.

```
248 { return right - left; }
```

Member Data Documentation

UINT32 PGFRect::bottom

Definition at line 258 of file PGFtypes.h.

UINT32 PGFRect::left

Definition at line 258 of file PGFtypes.h.

UINT32 PGFRect::right

Definition at line 258 of file PGFtypes.h.

UINT32 PGFRect::top

Definition at line 258 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.

```
133 : 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 190 of file PGFtypes.h.

Member Data Documentation

UINT16 ROIBlockHeader::RBH::bufferSize

number of uncoded UINT32 values in a block

Definition at line 195 of file PGFtypes.h.

UINT16 ROIBlockHeader::RBH::tileEnd

1: last part of a tile

Definition at line 196 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 179 of file PGFtypes.h.

Constructor & Destructor Documentation

ROIBlockHeader::ROIBlockHeader (UINT16 *v*)[inline]

Constructor

Parameters:

v	Buffer size

Definition at line 182 of file PGFtypes.h.

```
182 { 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 186 of file PGFtypes.h.

```
186 { 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 188 of file PGFtypes.h.

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

- **PGFtypes.h**

File Documentation

BitStream.h File Reference

PGF bit-stream operations.
`#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

Detailed Description

PGF bit-stream operations.

Author:
C. Stamm

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.

```
328                                         {
329 //      return ((pos + WordWidth - 1) >> WordWidthLog) << WordWidthLog;
330         return DWWIDTHBITS(pos);
331 }
```

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.

```
 70                                                  {
 71         stream[pos >> WordWidthLog] &= ~(1 << (pos%WordWidth));
 72 }
```

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.

```
169                                                                 {
170         ASSERT(len > 0);
171         const UINT32 iFirstInt = pos >> WordWidthLog;
172         const UINT32 iLastInt = (pos + len - 1) >> WordWidthLog;
173 
174         const UINT32 startMask = Filled << (pos%WordWidth);
175 //      const UINT32 endMask=Filled>>(WordWidth-1-((pos+len-1)%WordWidth));
176 
177         if (iFirstInt == iLastInt) {
178                 stream[iFirstInt] &= ~(startMask /*& endMask*/);
179         } else {
180                 stream[iFirstInt] &= ~startMask;
181                 for (UINT32 i = iFirstInt + 1; i <= iLastInt; i++) { // changed <=
182                         stream[i] = 0;
183                 }
184                 //stream[iLastInt] &= ~endMask;
185         }
186 }
```

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.

```
  91                                                                                {
  92         const UINT32 iLoInt = pos >> WordWidthLog;
  93         const UINT32 iHiInt = (pos + k - 1) >> WordWidthLog;
  94         ASSERT(iLoInt <= iHiInt);
  95         const UINT32 mask = (Filled >> (WordWidth - k));
  96
  97         if (iLoInt == iHiInt) {
  98                 // fits into one integer
  99                 val &= mask;
 100                 val <<= (pos%WordWidth);
 101                 return (stream[iLoInt] & val) == val;
 102         } else {
 103                 // must be splitted over integer boundary
 104                 UINT64 v1 = MAKEU64(stream[iLoInt], stream[iHiInt]);
 105                 UINT64 v2 = UINT64(val & mask) << (pos%WordWidth);
 106                 return (v1 & v2) == v2;
 107         }
 108 }
```

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.

```
  79                                                {
  80         return (stream[pos >> WordWidthLog] & (1 << (pos%WordWidth))) > 0;
  81
  82 }
```

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.

```
 142                                                                {
 143         UINT32 count, hiCount;
 144         const UINT32 iLoInt = pos >> WordWidthLog;                              // integer
of first bit
 145         const UINT32 iHiInt = (pos + k - 1) >> WordWidthLog;            // integer of last
bit
 146         const UINT32 loMask = Filled << (pos%WordWidth);
 147         const UINT32 hiMask = Filled >> (WordWidth - 1 - ((pos + k - 1)%WordWidth));
 148
 149         if (iLoInt == iHiInt) {
```

```
150                 // inside integer boundary
151                 count = stream[iLoInt] & (loMask & hiMask);
152                 count >>= pos%WordWidth;
153         } else {
154                 // overlapping integer boundary
155                 count = stream[iLoInt] & loMask;
156                 count >>= pos%WordWidth;
157                 hiCount = stream[iHiInt] & hiMask;
158                 hiCount <<= WordWidth - (pos%WordWidth);
159                 count |= hiCount;
160         }
161         return count;
162 }
```

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.

```
337                                         {
338         return (pos + WordWidth - 1) >> WordWidthLog;
339 }
```

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.

```
249                                                                         {
250         UINT32 count = 0;
251         UINT32 testMask = 1 << (pos%WordWidth);
252         UINT32* word = stream + (pos >> WordWidthLog);
253
254         while (((*word & testMask) != 0) && (count < len)) {
255                 count++;
256                 testMask <<= 1;
257                 if (!testMask) {
258                         word++; testMask = 1;
259
260                         // fast steps if all bits in a word are one
261                         while ((count + WordWidth <= len) && (*word == Filled)) {
262                                 word++;
263                                 count += WordWidth;
264                         }
265                 }
266         }
267         return count;
268 }
```

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.

```
220                                                                     {
221         UINT32 count = 0;
222         UINT32 testMask = 1 << (pos%WordWidth);
223         UINT32* word = stream + (pos >> WordWidthLog);
224
225         while (((*word & testMask) == 0) && (count < len)) {
226                 count++;
227                 testMask <<= 1;
228                 if (!testMask) {
229                         word++; testMask = 1;
230
231                         // fast steps if all bits in a word are zero
232                         while ((count + WordWidth <= len) && (*word == 0)) {
233                                 word++;
234                                 count += WordWidth;
235                         }
236                 }
237         }
238
239         return count;
240 }
```

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.

```
 62                                                 {
 63         stream[pos >> WordWidthLog] |= (1 << (pos%WordWidth));
 64 }
```

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.

```
193                                                                     {
194         ASSERT(len > 0);
195
196         const UINT32 iFirstInt = pos >> WordWidthLog;
197         const UINT32 iLastInt = (pos + len - 1) >> WordWidthLog;
198
199         const UINT32 startMask = Filled << (pos%WordWidth);
200 //      const UINT32 endMask=Filled>>(WordWidth-1-((pos+len-1)%WordWidth));
201
202         if (iFirstInt == iLastInt) {
203                 stream[iFirstInt] |= (startMask /*& endMask*/);
204         } else {
205                 stream[iFirstInt] |= startMask;
206                 for (UINT32 i = iFirstInt + 1; i <= iLastInt; i++) { // changed <=
207                         stream[i] = Filled;
```

```
208                 }
209                 //stream[iLastInt] &= ~endMask;
210         }
211 }
```

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.

```
116                                                                                 {
117         const UINT32 offset = pos%WordWidth;
118         const UINT32 iLoInt = pos >> WordWidthLog;
119         const UINT32 iHiInt = (pos + k - 1) >> WordWidthLog;
120         ASSERT(iLoInt <= iHiInt);
121         const UINT32 loMask = Filled << offset;
122         const UINT32 hiMask = Filled >> (WordWidth - 1 - ((pos + k - 1)%WordWidth));
123
124         if (iLoInt == iHiInt) {
125                 // fits into one integer
126                 stream[iLoInt] &= ~(loMask & hiMask); // clear bits
127                 stream[iLoInt] |= val << offset; // write value
128         } else {
129                 // must be splitted over integer boundary
130                 stream[iLoInt] &= ~loMask; // clear bits
131                 stream[iLoInt] |= val << offset; // write lower part of value
132                 stream[iHiInt] &= ~hiMask; // clear bits
133                 stream[iHiInt] |= val >> (WordWidth - offset); // write higher part of value
134         }
135 }
```

Variable Documentation

const UINT32 Filled = 0xFFFFFFFF[static]

Definition at line 38 of file 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

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.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

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.

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

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.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

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.

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

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.h File Reference

PGF image class.
`#include "PGFstream.h"`

Classes

- class **CPGFImage**

PGF main class.

Detailed Description

PGF image class.

Author:

C. Stamm

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

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 601 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.

PGFstream.cpp File Reference

PGF stream class implementation.
`#include "PGFstream.h"`

Detailed Description

PGF stream class implementation.

Author:
C. Stamm

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

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** 15
- #define **PGFWeek** 32
- #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 31 bits, otherwise 16 bit values are used -> allows at maximum 15 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

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 275 of file PGFtypes.h.

#define DataTSize sizeof(DataT)

Definition at line 276 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 274 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 272 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 277 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 31 bits, otherwise 16 bit values are used -> allows at maximum 15 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 32

Definition at line 46 of file PGFtypes.h.

#define PGFYear 15

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 273 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 262 of file PGFtypes.h.

typedef void(* RefreshCB) (void *p)

Definition at line 267 of file PGFtypes.h.

Enumeration Type Documentation

enum Orientation

Enumerator

LL
HL
LH
HH

Definition at line 99 of file PGFtypes.h.

```
99 { LL = 0, HL = 1, LH = 2, HH = 3 };
```

enum ProgressMode

Enumerator

PM_Relative
PM_Absolute

Definition at line 100 of file PGFtypes.h.

```
100 { PM_Relative, PM_Absolute };
```

enum UserdataPolicy

Enumerator

UP_Skip
UP_CachePrefix
UP_CacheAll

Definition at line 101 of file PGFtypes.h.

```
101 { UP_Skip = 0, UP_CachePrefix = 1, UP_CacheAll = 2 };
```

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

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

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

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.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

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.

Index