libpgf

Christoph Stamm
Version 6.12.24
6/15/2012 10:06:00 AM

Table of Contents

Class 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, bool skipUserData) THROW_
- **~CDecoder** ()
- void **Partition** (**CSubband** *band, int quantParam, int width, int height, int startPos, int pitch) THROW_
- void **DecodeInterleaved** (**CWaveletTransform** *wtChannel, int level, int quantParam) THROW_
- UINT32 **GetEncodedHeaderLength** () const
- void **SetStreamPosToStart** () THROW_
- void **SetStreamPosToData** () THROW_
- void **Skip** (UINT64 offset) THROW_
- void **DequantizeValue** (**CSubband** *band, UINT32 bandPos, int quantParam) THROW_
- UINT32 **ReadEncodedData** (UINT8 *target, UINT32 len) const THROW_
- void **DecodeBuffer** () THROW_
- **CPGFStream** * **GetStream** ()
- bool **MacroBlocksAvailable** () const
- void **DecodeTileBuffer** () THROW_
- void **SkipTileBuffer** () THROW_
- void **SetROI** ()

Private Member Functions

- void **ReadMacroBlock** (**CMacroBlock** *block) THROW_
  *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)
- bool **m_roi**
  true: ensures region of interest (ROI) decoding

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*, bool*skipUserData*)

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
skipUserData	If true, then user data is not read. In case of available user data, the file position is still returned in userDataPos.

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
skipUserData	If true, then user data is not read. In case of available user data, the file position is still returned in userDataPos.

Definition at line 73 of file Decoder.cpp.

```
: m_stream(stream)
```

```
, m_startPos(0)
, m_streamSizeEstimation(0)
, m_encodedHeaderLength(0)
, m_currentBlockIndex(0)
, m_macroBlocksAvailable(0)
#ifdef __PGFROISUPPORT__
, m_roi(false)
#endif
{
        ASSERT(m_stream);

        int count, expected;

        // set number of threads
#ifdef LIBPGF_USE_OPENMP
        m_macroBlockLen = omp_get_num_procs();
#else
        m_macroBlockLen = 1;
#endif

        if (useOMP && m_macroBlockLen > 1) {
#ifdef LIBPGF_USE_OPENMP
                omp_set_num_threads(m_macroBlockLen);
#endif

                // create macro block array
                m_macroBlocks = new(std::nothrow) CMacroBlock*[m_macroBlockLen];
                if (!m_macroBlocks) ReturnWithError(InsufficientMemory);
                for (int i=0; i < m_macroBlockLen; i++) m_macroBlocks[i] = new CMacroBlock(this);
                m_currentBlock = m_macroBlocks[m_currentBlockIndex];
        } else {
                m_macroBlocks = 0;
                m_macroBlockLen = 1; // there is only one macro block
                m_currentBlock = new CMacroBlock(this);
        }

        // store current stream position
        m_startPos = m_stream->GetPos();

        // read magic and version
        count = expected = MagicVersionSize;
        m_stream->Read(&count, &preHeader);
        if (count != expected) ReturnWithError(MissingData);

        // read header size
        if (preHeader.version & Version6) {
                // 32 bit header size since version 6
                count = expected = 4;
        } else {
                count = expected = 2;
        }
        m_stream->Read(&count, ((UINT8*)&preHeader) + MagicVersionSize);
        if (count != expected) ReturnWithError(MissingData);

        // make sure the values are correct read
        preHeader.hSize = __VAL(preHeader.hSize);

        // check magic number
        if (memcmp(preHeader.magic, Magic, 3) != 0) {
                // error condition: wrong Magic number
                ReturnWithError(FormatCannotRead);
        }

        // read file header
        count = expected = (preHeader.hSize < HeaderSize) ? preHeader.hSize : HeaderSize;
        m_stream->Read(&count, &header);
        if (count != expected) ReturnWithError(MissingData);

        // make sure the values are correct read
        header.height = __VAL(UINT32(header.height));
        header.width = __VAL(UINT32(header.width));

        // be ready to read all versions including version 0
        if (preHeader.version > 0) {
#ifndef __PGFROISUPPORT__
                // check ROI usage
                if (preHeader.version & PGFROI) ReturnWithError(FormatCannotRead);
#endif

                int size = preHeader.hSize - HeaderSize;

                if (size > 0) {
                        // read post-header
                        if (header.mode == ImageModeIndexedColor) {
                                ASSERT((size_t)size >= ColorTableSize);
                                // read color table
                                count = expected = ColorTableSize;
                                m_stream->Read(&count, postHeader.clut);
                                if (count != expected) ReturnWithError(MissingData);
                                size -= count;
                        }

                        if (size > 0) {
                                userDataPos = m_stream->GetPos();
                                postHeader.userDataLen = size;
                                if (skipUserData) {
                                        Skip(size);
                                } else {
                                        // create user data memory block
                                        postHeader.userData = new(std::nothrow)
UINT8[postHeader.userDataLen];
                                        if (!postHeader.userData)
ReturnWithError(InsufficientMemory);

                                        // read user data
                                        count = expected = postHeader.userDataLen;
                                        m_stream->Read(&count, postHeader.userData);
                                        if (count != expected) ReturnWithError(MissingData);
                                }
                        }
                }

                // create levelLength
                levelLength = new(std::nothrow) UINT32[header.nLevels];
                if (!levelLength) ReturnWithError(InsufficientMemory);

                // read levelLength
                count = expected = header.nLevels*WordBytes;
                m_stream->Read(&count, levelLength);
                if (count != expected) ReturnWithError(MissingData);

#ifdef PGF_USE_BIG_ENDIAN
                // make sure the values are correct read
                for (int i=0; i < header.nLevels; i++) {
                        levelLength[i] = __VAL(levelLength[i]);
                }
#endif

                // compute the total size in bytes; keep attention: level length information is
optional
                for (int i=0; i < header.nLevels; i++) {
                        m_streamSizeEstimation += levelLength[i];
                }

        }

        // store current stream position
        m_encodedHeaderLength = UINT32(m_stream->GetPos() - m_startPos);
}
```

CDecoder::~CDecoder ()

Destructor

Definition at line 216 of file Decoder.cpp.

```
                    {
        if (m_macroBlocks) {
                for (int i=0; i < m_macroBlockLen; i++) delete m_macroBlocks[i];
                delete[] m_macroBlocks;
        } else {
                delete m_currentBlock;
        }
}
```

Member Function Documentation

void CDecoder::DecodeBuffer ()

Reads stream and decodes tile buffer It might throw an **IOException**.

Definition at line 479 of file Decoder.cpp.

```
                                   {
        ASSERT(m_macroBlocksAvailable <= 0);

        // macro block management
        if (m_macroBlockLen == 1) {
                ASSERT(m_currentBlock);
                ReadMacroBlock(m_currentBlock);
                m_currentBlock->BitplaneDecode();
                m_macroBlocksAvailable = 1;
        } else {
                m_macroBlocksAvailable = 0;
                for (int i=0; i < m_macroBlockLen; i++) {
                        // read sequentially several blocks
                        try {
                                ReadMacroBlock(m_macroBlocks[i]);
                                m_macroBlocksAvailable++;
                        } catch(IOException& ex) {
                                if (ex.error == MissingData) {
                                        break; // no further data available
                                } else {
                                        throw ex;
                                }
                        }
                }

                // decode in parallel
                #pragma omp parallel for default(shared) //no declared exceptions in next block
                for (int i=0; i < m_macroBlocksAvailable; i++) {
                        m_macroBlocks[i]->BitplaneDecode();
                }

                // prepare current macro block
                m_currentBlockIndex = 0;
                m_currentBlock = m_macroBlocks[m_currentBlockIndex];
        }
}
```

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 318 of file Decoder.cpp.

```

{
        CSubband* hlBand = wtChannel->GetSubband(level, HL);
        CSubband* lhBand = wtChannel->GetSubband(level, LH);
        const div_t lhH = div(lhBand->GetHeight(), InterBlockSize);
        const div_t hlW = div(hlBand->GetWidth(), InterBlockSize);
        const int hlws = hlBand->GetWidth() - InterBlockSize;
        const int hlwr = hlBand->GetWidth() - hlW.rem;
        const int lhws = lhBand->GetWidth() - InterBlockSize;
        const int lhwr = lhBand->GetWidth() - hlW.rem;
        int hlPos, lhPos;
        int hlBase = 0, lhBase = 0, hlBase2, lhBase2;

        ASSERT(lhBand->GetWidth() >= hlBand->GetWidth());
        ASSERT(hlBand->GetHeight() >= lhBand->GetHeight());

        if (!hlBand->AllocMemory()) ReturnWithError(InsufficientMemory);
        if (!lhBand->AllocMemory()) ReturnWithError(InsufficientMemory);

        // correct quantParam with normalization factor
        quantParam -= level;
        if (quantParam < 0) quantParam = 0;

        // main height
        for (int i=0; i < lhH.quot; i++) {
                // main width
                hlBase2 = hlBase;
                lhBase2 = lhBase;
                for (int j=0; j < hlW.quot; j++) {
                        hlPos = hlBase2;
                        lhPos = lhBase2;
                        for (int y=0; y < InterBlockSize; y++) {
                                for (int x=0; x < InterBlockSize; x++) {
                                        DequantizeValue(hlBand, hlPos, quantParam);
                                        DequantizeValue(lhBand, lhPos, quantParam);
                                        hlPos++;
                                        lhPos++;
                                }
                                hlPos += hlws;
                                lhPos += lhws;
                        }
                        hlBase2 += InterBlockSize;
                        lhBase2 += InterBlockSize;
                }
                // rest of width
                hlPos = hlBase2;
                lhPos = lhBase2;
                for (int y=0; y < InterBlockSize; y++) {
                        for (int x=0; x < hlW.rem; x++) {
                                DequantizeValue(hlBand, hlPos, quantParam);
                                DequantizeValue(lhBand, lhPos, quantParam);
                                hlPos++;
                                lhPos++;
                        }
                        // width difference between HL and LH
                        if (lhBand->GetWidth() > hlBand->GetWidth()) {
                                DequantizeValue(lhBand, lhPos, quantParam);
                        }
                        hlPos += hlwr;
                        lhPos += lhwr;
                        hlBase += hlBand->GetWidth();
                        lhBase += lhBand->GetWidth();
                }
        }
        // main width
        hlBase2 = hlBase;
        lhBase2 = lhBase;
        for (int j=0; j < hlW.quot; j++) {
                // rest of height
                hlPos = hlBase2;
                lhPos = lhBase2;
                for (int y=0; y < lhH.rem; y++) {
                        for (int x=0; x < InterBlockSize; x++) {
                                DequantizeValue(hlBand, hlPos, quantParam);
                                DequantizeValue(lhBand, lhPos, quantParam);
                                hlPos++;
                                lhPos++;
                        }
                        hlPos += hlws;
                        lhPos += lhws;
                }
                hlBase2 += InterBlockSize;
                lhBase2 += InterBlockSize;
        }
        // rest of height
        hlPos = hlBase2;
        lhPos = lhBase2;
        for (int y=0; y < lhH.rem; y++) {
                // rest of width
                for (int x=0; x < hlW.rem; x++) {
                        DequantizeValue(hlBand, hlPos, quantParam);
                        DequantizeValue(lhBand, lhPos, quantParam);
                        hlPos++;
                        lhPos++;
                }
                // width difference between HL and LH
                if (lhBand->GetWidth() > hlBand->GetWidth()) {
                        DequantizeValue(lhBand, lhPos, quantParam);
                }
                hlPos += hlwr;
                lhPos += lhwr;
                hlBase += hlBand->GetWidth();
        }
        // height difference between HL and LH
        if (hlBand->GetHeight() > lhBand->GetHeight()) {
                // total width
                hlPos = hlBase;
                for (int j=0; j < hlBand->GetWidth(); j++) {
                        DequantizeValue(hlBand, hlPos, quantParam);
                        hlPos++;
                }
        }
}
```

void CDecoder::DecodeTileBuffer ()

Reads stream and decodes tile buffer It might throw an **IOException**.

Definition at line 462 of file Decoder.cpp.

```
                                       {
        // current block has been read --> prepare next current block
        m_macroBlocksAvailable--;

        if (m_macroBlocksAvailable > 0) {
                m_currentBlock = m_macroBlocks[++m_currentBlockIndex];
        } else {
                DecodeBuffer();
        }
        ASSERT(m_currentBlock);
}
```

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 447 of file Decoder.cpp.

```
                                                                                   {
        ASSERT(m_currentBlock);

        if (m_currentBlock->IsCompletelyRead()) {
                // all data of current macro block has been read --> prepare next macro block
                DecodeTileBuffer();
        }

        band->SetData(bandPos, m_currentBlock->m_value[m_currentBlock->m_valuePos] <<
quantParam);
        m_currentBlock->m_valuePos++;
}
```

UINT32 CDecoder::GetEncodedHeaderLength () const `[inline]`

Return 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.

```
{ return m_encodedHeaderLength; }
```

CPGFStream* CDecoder::GetStream () `[inline]`

Returns:

Stream

Definition at line 174 of file Decoder.h.

```
{ return m_stream; }
```

bool CDecoder::MacroBlocksAvailable () const `[inline]`

Returns:

True if decoded macro blocks are available for processing

Definition at line 178 of file Decoder.h.

```
{ return m_macroBlocksAvailable > 1; }
```

void CDecoder::Partition (CSubband *band, intquantParam, intwidth, intheight, intstartPos, intpitch)

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 251 of file Decoder.cpp.

```
{
        ASSERT(band);

        const div_t ww = div(width, LinBlockSize);
        const div_t hh = div(height, LinBlockSize);
        const int ws = pitch - LinBlockSize;
        const int wr = pitch - ww.rem;
        int pos, base = startPos, base2;

        // main height
        for (int i=0; i < hh.quot; i++) {
                // main width
                base2 = base;
                for (int j=0; j < ww.quot; j++) {
                        pos = base2;
                        for (int y=0; y < LinBlockSize; y++) {
                                for (int x=0; x < LinBlockSize; x++) {
                                        DequantizeValue(band, pos, quantParam);
                                        pos++;
                                }
                                pos += ws;
                        }
                        base2 += LinBlockSize;
                }
                // rest of width
                pos = base2;
                for (int y=0; y < LinBlockSize; y++) {
                        for (int x=0; x < ww.rem; x++) {
                                DequantizeValue(band, pos, quantParam);
                                pos++;
                        }
                        pos += wr;
                        base += pitch;
                }
        }
        // main width
        base2 = base;
        for (int j=0; j < ww.quot; j++) {
                // rest of height
                pos = base2;
                for (int y=0; y < hh.rem; y++) {
                        for (int x=0; x < LinBlockSize; x++) {
                                DequantizeValue(band, pos, quantParam);
                                pos++;
                        }
                        pos += ws;
                }
                base2 += LinBlockSize;
        }
        // rest of height
        pos = base2;
        for (int y=0; y < hh.rem; y++) {
                // rest of width
                for (int x=0; x < ww.rem; x++) {
                        DequantizeValue(band, pos, quantParam);
                        pos++;
                }
                pos += wr;
        }
}
```

UINT32 CDecoder::ReadEncodedData (UINT8 *target, UINT32len) 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 231 of file Decoder.cpp.

```
                                                                       {
        ASSERT(m_stream);

        int count = len;
        m_stream->Read(&count, target);

        return count;
}
```

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

throws **IOException**

Definition at line 519 of file Decoder.cpp.

```
                                                       {
        ASSERT(block);

        UINT16 wordLen;
        ROIBlockHeader h(BufferSize);
        int count, expected;

#ifdef TRACE
        //UINT32 filePos = (UINT32)m_stream->GetPos();
        //printf("DecodeBuffer: %d\n", filePos);
#endif

        // read wordLen
        count = expected = sizeof(UINT16);
        m_stream->Read(&count, &wordLen);
        if (count != expected) ReturnWithError(MissingData);
        wordLen = __VAL(wordLen);
        if (wordLen > BufferSize)
                ReturnWithError(FormatCannotRead);

#ifdef __PGFROISUPPORT__
        // read ROIBlockHeader
        if (m_roi) {
                m_stream->Read(&count, &h.val);
                if (count != expected) ReturnWithError(MissingData);

                // convert ROIBlockHeader
                h.val = __VAL(h.val);
        }
#endif
        // save header
        block->m_header = h;

        // read data
        count = expected = wordLen*WordBytes;
        m_stream->Read(&count, block->m_codeBuffer);
        if (count != expected) ReturnWithError(MissingData);

#ifdef PGF_USE_BIG_ENDIAN
        // convert data
        count /= WordBytes;
        for (int i=0; i < count; i++) {
                block->m_codeBuffer[i] = __VAL(block->m_codeBuffer[i]);
        }
#endif

#ifdef __PGFROISUPPORT__
        ASSERT(m_roi && h.rbh.bufferSize <= BufferSize || h.rbh.bufferSize == BufferSize);
#else
        ASSERT(h.rbh.bufferSize == BufferSize);
#endif
}
```

void CDecoder::SetROI () `[inline]`

Enables region of interest (ROI) status.

Definition at line 193 of file Decoder.h.

```
{ m_roi = true; }
```

void CDecoder::SetStreamPosToData () `[inline]`

Reset stream position to beginning of data block

Definition at line 144 of file Decoder.h.

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

void CDecoder::SetStreamPosToStart () `[inline]`

Reset stream position to beginning of PGF pre-header

Definition at line 140 of file Decoder.h.

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

void CDecoder::Skip (UINT64*offset*)

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

Definition at line 434 of file Decoder.cpp.

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

void CDecoder::SkipTileBuffer ()

Resets stream position to next tile. It might throw an **IOException**.

Definition at line 577 of file Decoder.cpp.

```
                                     {
        // current block is not used
        m_macroBlocksAvailable--;

        // check if pre-decoded data is available
        if (m_macroBlocksAvailable > 0) {
                m_currentBlock = m_macroBlocks[++m_currentBlockIndex];
                return;
        }

        UINT16 wordLen;
        int count, expected;

        // read wordLen
        count = expected = sizeof(wordLen);
        m_stream->Read(&count, &wordLen);
        if (count != expected) ReturnWithError(MissingData);
        wordLen = __VAL(wordLen);
        ASSERT(wordLen <= BufferSize);

#ifdef __PGFROISUPPORT__
        if (m_roi) {
                // skip ROIBlockHeader
                m_stream->SetPos(FSFromCurrent, sizeof(ROIBlockHeader));
        }
#endif

        // skip data
        m_stream->SetPos(FSFromCurrent, wordLen*WordBytes);
}
```

Member Data Documentation

CMacroBlock* CDecoder::m_currentBlock `[private]`

current macro block (used by main thread)

Definition at line 212 of file Decoder.h.

int CDecoder::m_currentBlockIndex `[private]`

index of current macro block

Definition at line 209 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 206 of file Decoder.h.

int CDecoder::m_macroBlockLen `[private]`

array length

Definition at line 210 of file Decoder.h.

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

array of macroblocks

Definition at line 208 of file Decoder.h.

int CDecoder::m_macroBlocksAvailable `[private]`

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

Definition at line 211 of file Decoder.h.

bool CDecoder::m_roi `[private]`

true: ensures region of interest (ROI) decoding

Definition at line 215 of file Decoder.h.

UINT64 CDecoder::m_startPos `[private]`

stream position at the beginning of the PGF pre-header

Definition at line 204 of file Decoder.h.

CPGFStream* CDecoder::m_stream `[private]`

input PGF stream

Definition at line 203 of file Decoder.h.

UINT64 CDecoder::m_streamSizeEstimation `[private]`

estimation of stream size

Definition at line 205 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) THROW_
- **~CEncoder** ()
- void **FavorSpeedOverSize** ()
- void **Flush** () THROW_
- void **UpdatePostHeaderSize** (**PGFPreHeader** preHeader) THROW_
- UINT32 **WriteLevelLength** (UINT32 *&levelLength) THROW_
- UINT32 **UpdateLevelLength** () THROW_
- void **Partition** (**CSubband** *band, int width, int height, int startPos, int pitch) THROW_
- void **SetEncodedLevel** (int currentLevel)
- void **WriteValue** (**CSubband** *band, int bandPos) THROW_
- INT64 **ComputeHeaderLength** () const
- INT64 **ComputeBufferLength** () const
- INT64 **ComputeOffset** () const
- void **SetBufferStartPos** ()
- void **EncodeTileBuffer** () THROW_
- void **SetROI** ()

Private Member Functions

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

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
- bool **m_roi**
  true: ensures region of interest (ROI) encoding

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.

```
: m_stream(stream)
, m_bufferStartPos(0)
, m_currLevelIndex(0)
, m_nLevels(header.nLevels)
, m_favorSpeed(false)
, m_forceWriting(false)
#ifdef __PGFROISUPPORT__
, m_roi(false)
#endif
{
        ASSERT(m_stream);

        int count;

        // set number of threads
#ifdef LIBPGF_USE_OPENMP
        m_macroBlockLen = omp_get_num_procs();
#else
        m_macroBlockLen = 1;
#endif

        if (useOMP && m_macroBlockLen > 1) {
#ifdef LIBPGF_USE_OPENMP
                omp_set_num_threads(m_macroBlockLen);
#endif
                // create macro block array
                m_macroBlocks = new(std::nothrow) CMacroBlock*[m_macroBlockLen];
                if (!m_macroBlocks) ReturnWithError(InsufficientMemory);
                for (int i=0; i < m_macroBlockLen; i++) m_macroBlocks[i] = new CMacroBlock(this);
                m_lastMacroBlock = 0;
                m_currentBlock = m_macroBlocks[m_lastMacroBlock++];
        } else {
                m_macroBlocks = 0;
                m_macroBlockLen = 1;
                m_currentBlock = new CMacroBlock(this);
        }

        // save file position
        m_startPosition = m_stream->GetPos();

        // write preHeader
        preHeader.hSize = __VAL(preHeader.hSize);
        count = PreHeaderSize;
        m_stream->Write(&count, &preHeader);

        // write file header
        header.height = __VAL(header.height);
        header.width = __VAL(header.width);
        count = HeaderSize;
        m_stream->Write(&count, &header);

        // write postHeader
        if (header.mode == ImageModeIndexedColor) {
                // write color table
                count = ColorTableSize;
                m_stream->Write(&count, (void *)postHeader.clut);
        }
        // save user data file position
        userDataPos = m_stream->GetPos();
        if (postHeader.userDataLen) {
                if (postHeader.userData) {
                        // write user data
                        count = postHeader.userDataLen;
                        m_stream->Write(&count, postHeader.userData);
                } else {
                        m_stream->SetPos(FSFromCurrent, count);
                }
        }

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

CEncoder::~CEncoder ()

Destructor

Definition at line 146 of file Encoder.cpp.

```
                    {
        delete m_currentBlock;
        delete[] m_macroBlocks;
}
```

Member Function Documentation

INT64 CEncoder::ComputeBufferLength () const `[inline]`

Compute stream length of encoded buffer.

Returns:

encoded buffer length

Definition at line 175 of file Encoder.h.

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

INT64 CEncoder::ComputeHeaderLength () const `[inline]`

Compute stream length of header.

Returns:

header length

Definition at line 170 of file Encoder.h.

```
{ 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 180 of file Encoder.h.

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

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

Definition at line 336 of file Encoder.cpp.

```
                                                   {
        ASSERT(m_currentBlock);
#ifdef __PGFROISUPPORT__
        ASSERT(m_roi && h.rbh.bufferSize <= BufferSize || h.rbh.bufferSize == BufferSize);
#else
        ASSERT(h.rbh.bufferSize == BufferSize);
#endif
        m_currentBlock->m_header = h;

        // macro block management
        if (m_macroBlockLen == 1) {
                m_currentBlock->BitplaneEncode();
                WriteMacroBlock(m_currentBlock);
        } else {
                // save last level index
                int lastLevelIndex = m_currentBlock->m_lastLevelIndex;

                if (m_forceWriting || m_lastMacroBlock == m_macroBlockLen) {
                        // encode macro blocks
                        /*
                        volatile OSError error = NoError;
                        #pragma omp parallel for ordered default(shared)
                        for (int i=0; i < m_lastMacroBlock; i++) {
                                if (error == NoError) {
                                        m_macroBlocks[i]->BitplaneEncode();
                                        #pragma omp ordered
                                        {
                                                try {
                                                        WriteMacroBlock(m_macroBlocks[i]);
                                                } catch (IOException& e) {
                                                        error = e.error;
                                                }
                                                delete m_macroBlocks[i]; m_macroBlocks[i] = 0;
                                        }
                                }
                        }
                        if (error != NoError) ReturnWithError(error);
                        */
                        #pragma omp parallel for default(shared) //no declared exceptions in next
block
                        for (int i=0; i < m_lastMacroBlock; i++) {
                                m_macroBlocks[i]->BitplaneEncode();
                        }
                        for (int i=0; i < m_lastMacroBlock; i++) {
                                WriteMacroBlock(m_macroBlocks[i]);
                        }

                        // prepare for next round
                        m_forceWriting = false;
                        m_lastMacroBlock = 0;
                }
                // re-initialize macro block
                m_currentBlock = m_macroBlocks[m_lastMacroBlock++];
                m_currentBlock->Init(lastLevelIndex);
        }
}
```

void CEncoder::EncodeTileBuffer () `[inline]`

Encodes tile buffer and writes it into stream It might throw an **IOException**.

Definition at line 190 of file Encoder.h.

```
{ ASSERT(m_currentBlock && m_currentBlock->m_valuePos >= 0 && m_currentBlock->m_valuePos <=
BufferSize); EncodeBuffer(ROIBlockHeader(m_currentBlock->m_valuePos, true)); }
```

void CEncoder::FavorSpeedOverSize () `[inline]`

Encoder favors speed over compression size

Definition at line 117 of file Encoder.h.

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

void CEncoder::Flush ()

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

Definition at line 305 of file Encoder.cpp.

```
                            {
        if (m_currentBlock->m_valuePos > 0) {
                // pad buffer with zeros
                memset(&(m_currentBlock->m_value[m_currentBlock->m_valuePos]), 0, (BufferSize -
m_currentBlock->m_valuePos)*DataTSize);
                m_currentBlock->m_valuePos = BufferSize;

                // encode buffer
                m_forceWriting = true;  // makes sure that the following EncodeBuffer is really
written into the stream
                EncodeBuffer(ROIBlockHeader(m_currentBlock->m_valuePos, true));
        }
}
```

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 241 of file Encoder.cpp.

```
{
        ASSERT(band);

        const div_t hh = div(height, LinBlockSize);
        const div_t ww = div(width, LinBlockSize);
        const int ws = pitch - LinBlockSize;
        const int wr = pitch - ww.rem;
        int pos, base = startPos, base2;

        // main height
        for (int i=0; i < hh.quot; i++) {
                // main width
                base2 = base;
                for (int j=0; j < ww.quot; j++) {
                        pos = base2;
                        for (int y=0; y < LinBlockSize; y++) {
                                for (int x=0; x < LinBlockSize; x++) {
                                        WriteValue(band, pos);
                                        pos++;
                                }
                                pos += ws;
                        }
                        base2 += LinBlockSize;
                }
                // rest of width
                pos = base2;
                for (int y=0; y < LinBlockSize; y++) {
                        for (int x=0; x < ww.rem; x++) {
                                WriteValue(band, pos);
                                pos++;
                        }
                        pos += wr;
                        base += pitch;
                }
        }
        // main width
        base2 = base;
        for (int j=0; j < ww.quot; j++) {
                // rest of height
                pos = base2;
                for (int y=0; y < hh.rem; y++) {
                        for (int x=0; x < LinBlockSize; x++) {
                                WriteValue(band, pos);
                                pos++;
                        }
                        pos += ws;
                }
                base2 += LinBlockSize;
        }
        // rest of height
        pos = base2;
        for (int y=0; y < hh.rem; y++) {
                // rest of width
                for (int x=0; x < ww.rem; x++) {
                        WriteValue(band, pos);
                        pos++;
                }
                pos += wr;
        }
}
```

void CEncoder::SetBufferStartPos () `[inline]`

Save current stream position as beginning of current level.

Definition at line 184 of file Encoder.h.

```
{ 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 158 of file Encoder.h.

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

void CEncoder::SetROI () `[inline]`

Enables region of interest (ROI) status.

Definition at line 194 of file Encoder.h.

```
{ m_roi = true; }
```

UINT32 CEncoder::UpdateLevelLength ()

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

Returns:

Written image bytes.

Definition at line 197 of file Encoder.cpp.

```
                                          {
        UINT64 curPos = m_stream->GetPos(); // end of image

        // set file pos to levelLength
        m_stream->SetPos(FSFromStart, m_levelLengthPos);

        if (m_levelLength) {
        #ifdef PGF_USE_BIG_ENDIAN
                UINT32 levelLength;
                int count = WordBytes;

                for (int i=0; i < m_currLevelIndex; i++) {
                        levelLength = __VAL(UINT32(m_levelLength[i]));
                        m_stream->Write(&count, &levelLength);
                }
        #else
                int count = m_currLevelIndex*WordBytes;

                m_stream->Write(&count, m_levelLength);
        #endif //PGF_USE_BIG_ENDIAN
        } else {
                int count = m_currLevelIndex*WordBytes;
                m_stream->SetPos(FSFromCurrent, count);
        }

        // begin of image
        UINT32 retValue = UINT32(curPos - m_stream->GetPos());

        // restore file position
        m_stream->SetPos(FSFromStart, curPos);
        return retValue;
}
```

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 155 of file Encoder.cpp.

```
                                                                 {
        UINT64 curPos = m_stream->GetPos(); // end of user data
        int count = PreHeaderSize;

        // write preHeader
        m_stream->SetPos(FSFromStart, m_startPosition);
        preHeader.hSize = __VAL(preHeader.hSize);
        m_stream->Write(&count, &preHeader);

        m_stream->SetPos(FSFromStart, curPos);
}
```

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 172 of file Encoder.cpp.

```
                                                             {
        // renew levelLength
        delete[] levelLength;
        levelLength = new(std::nothrow) UINT32[m_nLevels];
        if (!levelLength) ReturnWithError(InsufficientMemory);
        for (UINT8 l = 0; l < m_nLevels; l++) levelLength[l] = 0;
        m_levelLength = levelLength;

        // save level length file position
        m_levelLengthPos = m_stream->GetPos();

        // write dummy levelLength
        int count = m_nLevels*WordBytes;
        m_stream->Write(&count, m_levelLength);

        // save current file position
        SetBufferStartPos();

        return count;
}
```

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

Definition at line 395 of file Encoder.cpp.

```
                                                        {
        ASSERT(block);

        ROIBlockHeader h = block->m_header;
        UINT16 wordLen = UINT16(NumberOfWords(block->m_codePos)); ASSERT(wordLen <=
CodeBufferLen);
        int count = sizeof(UINT16);
#ifdef TRACE
        //UINT32 filePos = (UINT32)m_stream->GetPos();
        //printf("EncodeBuffer: %d\n", filePos);
#endif

#ifdef PGF_USE_BIG_ENDIAN
        // write wordLen
        UINT16 wl = __VAL(wordLen);
        m_stream->Write(&count, &wl); ASSERT(count == sizeof(UINT16));

#ifdef __PGFROISUPPORT__
        // write ROIBlockHeader
        if (m_roi) {
                h.val = __VAL(h.val);
                m_stream->Write(&count, &h.val); ASSERT(count == sizeof(UINT16));
        }
#endif // __PGFROISUPPORT__

        // convert data
        for (int i=0; i < wordLen; i++) {
                block->m_codeBuffer[i] = __VAL(block->m_codeBuffer[i]);
        }
#else
        // write wordLen
        m_stream->Write(&count, &wordLen); ASSERT(count == sizeof(UINT16));

#ifdef __PGFROISUPPORT__
        // write ROIBlockHeader
        if (m_roi) {
                m_stream->Write(&count, &h.val); ASSERT(count == sizeof(UINT16));
        }
#endif // __PGFROISUPPORT__
#endif // PGF_USE_BIG_ENDIAN

        // write encoded data into stream
        count = wordLen*WordBytes;
        m_stream->Write(&count, block->m_codeBuffer);

        // store levelLength
        if (m_levelLength) {
                // store level length
                // EncodeBuffer has been called after m_lastLevelIndex has been updated
                ASSERT(m_currLevelIndex < m_nLevels);
                m_levelLength[m_currLevelIndex] += (UINT32)ComputeBufferLength();
                m_currLevelIndex = block->m_lastLevelIndex + 1;

        }

        // prepare for next buffer
        SetBufferStartPos();

        // reset values
        block->m_valuePos = 0;
        block->m_maxAbsValue = 0;
}
```

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 321 of file Encoder.cpp.

```
                                                            {
        if (m_currentBlock->m_valuePos == BufferSize) {
                EncodeBuffer(ROIBlockHeader(BufferSize, false));
        }
        DataT val = m_currentBlock->m_value[m_currentBlock->m_valuePos++] =
band->GetData(bandPos);
        UINT32 v = abs(val);
        if (v > m_currentBlock->m_maxAbsValue) m_currentBlock->m_maxAbsValue = v;
}
```

Member Data Documentation

UINT64 CEncoder::m_bufferStartPos [private]

stream position of encoded buffer

Definition at line 208 of file Encoder.h.

CMacroBlock* CEncoder::m_currentBlock [private]

current macro block (used by main thread)

Definition at line 213 of file Encoder.h.

int CEncoder::m_currLevelIndex [private]

counts where (=index) to save next value

Definition at line 216 of file Encoder.h.

bool CEncoder::m_favorSpeed [private]

favor speed over size

Definition at line 218 of file Encoder.h.

bool CEncoder::m_forceWriting [private]

all macro blocks have to be written into the stream

Definition at line 219 of file Encoder.h.

int CEncoder::m_lastMacroBlock [private]

array index of the last created macro block

Definition at line 212 of file Encoder.h.

UINT32* CEncoder::m_levelLength [private]

temporary saves the level index

Definition at line 215 of file Encoder.h.

UINT64 CEncoder::m_levelLengthPos [private]

stream position of Metadata

Definition at line 207 of file Encoder.h.

int CEncoder::m_macroBlockLen `[private]`

array length

Definition at line 211 of file Encoder.h.

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

array of macroblocks

Definition at line 210 of file Encoder.h.

UINT8 CEncoder::m_nLevels `[private]`

number of levels

Definition at line 217 of file Encoder.h.

bool CEncoder::m_roi `[private]`

true: ensures region of interest (ROI) encoding

Definition at line 221 of file Encoder.h.

UINT64 CEncoder::m_startPosition `[private]`

stream position of PGF start (PreHeader)

Definition at line 206 of file Encoder.h.

CPGFStream* CEncoder::m_stream `[private]`

output PMF stream

Definition at line 205 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.

```
                : m_header(0)
                , m_encoder(encoder)
                {
                        ASSERT(m_encoder);
                        Init(-1);
                }
```

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 468 of file Encoder.cpp.

```
                                        {
        UINT8   nPlanes;
        UINT32  sigLen, codeLen = 0, wordPos, refLen, signLen;
        UINT32  sigBits[BufferLen] = { 0 };
        UINT32  refBits[BufferLen] = { 0 };
        UINT32  signBits[BufferLen] = { 0 };
        UINT32  planeMask;
        UINT32  bufferSize = m_header.rbh.bufferSize; ASSERT(bufferSize <= BufferSize);
        bool    useRL;

#ifdef TRACE
        //printf("which thread: %d\n", omp_get_thread_num());
#endif

        // clear significance vector
        for (UINT32 k=0; k < bufferSize; k++) {
                m_sigFlagVector[k] = false;
        }
        m_sigFlagVector[bufferSize] = true; // sentinel

        // clear output buffer
        for (UINT32 k=0; k < bufferSize; k++) {
                m_codeBuffer[k] = 0;
        }
        m_codePos = 0;

        // compute number of bit planes and split buffer into separate bit planes
        nPlanes = NumberOfBitplanes();

        // write number of bit planes to m_codeBuffer
        // <nPlanes>
        SetValueBlock(m_codeBuffer, 0, nPlanes, MaxBitPlanesLog);
        m_codePos += MaxBitPlanesLog;

        // loop through all bit planes
        if (nPlanes == 0) nPlanes = MaxBitPlanes + 1;
        planeMask = 1 << (nPlanes - 1);

        for (int plane = nPlanes - 1; plane >= 0; plane--) {
                // clear significant bitset
                for (UINT32 k=0; k < BufferLen; k++) {
                        sigBits[k] = 0;
                }

                // split bitplane in significant bitset and refinement bitset
                sigLen = DecomposeBitplane(bufferSize, planeMask, m_codePos + RLblockSizeLen + 1,
sigBits, refBits, signBits, signLen, codeLen);

                if (sigLen > 0 && codeLen <= MaxCodeLen && codeLen < AlignWordPos(sigLen) +
AlignWordPos(signLen) + 2*RLblockSizeLen) {
                        // set RL code bit
                        // <1><codeLen>
                        SetBit(m_codeBuffer, m_codePos++);

                        // write length codeLen to m_codeBuffer
                        SetValueBlock(m_codeBuffer, m_codePos, codeLen, RLblockSizeLen);
                        m_codePos += RLblockSizeLen + codeLen;
                } else {
                #ifdef TRACE
                        //printf("new\n");
                        //for (UINT32 i=0; i < bufferSize; i++) {
                        //      printf("%s", (GetBit(sigBits, i)) ? "1" : "_");
                        //      if (i%120 == 119) printf("\n");
                        //}
                        //printf("\n");
                #endif // TRACE

                        // run-length coding wasn't efficient enough
                        // we don't use RL coding for sigBits
                        // <0><sigLen>
                        ClearBit(m_codeBuffer, m_codePos++);

                        // write length sigLen to m_codeBuffer
                        ASSERT(sigLen <= MaxCodeLen);
                        SetValueBlock(m_codeBuffer, m_codePos, sigLen, RLblockSizeLen);
                        m_codePos += RLblockSizeLen;

                        if (m_encoder->m_favorSpeed || signLen == 0) {
                                useRL = false;
                        } else {
                                // overwrite m_codeBuffer
                                useRL = true;
                                // run-length encode m_sign and append them to the m_codeBuffer
                                codeLen = RLESigns(m_codePos + RLblockSizeLen + 1, signBits,
signLen);
                        }

                        if (useRL && codeLen <= MaxCodeLen && codeLen < signLen) {
                                // RL encoding of m_sign was efficient
                                // <1><codeLen><codedSignBits>_
                                // write RL code bit
                                SetBit(m_codeBuffer, m_codePos++);

                                // write codeLen to m_codeBuffer
                                SetValueBlock(m_codeBuffer, m_codePos, codeLen,
RLblockSizeLen);

                                // compute position of sigBits
                                wordPos = NumberOfWords(m_codePos + RLblockSizeLen + codeLen);
                                ASSERT(0 <= wordPos && wordPos < CodeBufferLen);
                        } else {
                                // RL encoding of signBits wasn't efficient
                                // <0><signLen>_<signBits>_
                                // clear RL code bit
                                ClearBit(m_codeBuffer, m_codePos++);

                                // write signLen to m_codeBuffer
                                ASSERT(signLen <= MaxCodeLen);
                                SetValueBlock(m_codeBuffer, m_codePos, signLen,
RLblockSizeLen);

                                // write signBits to m_codeBuffer
                                wordPos = NumberOfWords(m_codePos + RLblockSizeLen);
                                ASSERT(0 <= wordPos && wordPos < CodeBufferLen);
                                codeLen = NumberOfWords(signLen);
```

```
                                        for (UINT32 k=0; k < codeLen; k++) {
                                                m_codeBuffer[wordPos++] = signBits[k];
                                        }
                                }

                                // write sigBits
                                // <sigBits>_
                                ASSERT(0 <= wordPos && wordPos < CodeBufferLen);
                                refLen = NumberOfWords(sigLen);

                                for (UINT32 k=0; k < refLen; k++) {
                                        m_codeBuffer[wordPos++] = sigBits[k];
                                }
                                m_codePos = wordPos << WordWidthLog;
                        }

                        // append refinement bitset (aligned to word boundary)
                        // _<refBits>
                        wordPos = NumberOfWords(m_codePos);
                        ASSERT(0 <= wordPos && wordPos < CodeBufferLen);
                        refLen = NumberOfWords(bufferSize - sigLen);

                        for (UINT32 k=0; k < refLen; k++) {
                                m_codeBuffer[wordPos++] = refBits[k];
                        }
                        m_codePos = wordPos << WordWidthLog;
                        planeMask >>= 1;
                }
        ASSERT(0 <= m_codePos && m_codePos <= CodeBufferBitLen);
}
```

UINT32 CEncoder::CMacroBlock::DecomposeBitplane (UINT32*bufferSize*, UINT32*planeMask*, UINT32*codePos*, UINT32 *_sigBits_, UINT32 *_refBits_, UINT32 *_signBits_, UINT32 &_signLen_, UINT32 &_codeLen_) `[private]`

Definition at line 620 of file Encoder.cpp.

```
{
        ASSERT(sigBits);
        ASSERT(refBits);
        ASSERT(signBits);
        ASSERT(codePos < CodeBufferBitLen);

        UINT32 sigPos = 0;
        UINT32 valuePos = 0, valueEnd;
        UINT32 refPos = 0;

        // set output value
        signLen = 0;

        // prepare RLE of Sigs and Signs
        const UINT32 outStartPos = codePos;
        UINT32 k = 3;
        UINT32 runlen = 1 << k; // = 2^k
        UINT32 count = 0;

        while (valuePos < bufferSize) {
                // search next 1 in m_sigFlagVector using searching with sentinel
                valueEnd = valuePos;
                while(!m_sigFlagVector[valueEnd]) { valueEnd++; }

                // search 1's in m_value[plane][valuePos..valueEnd)
                // these 1's are significant bits
                while (valuePos < valueEnd) {
                        if (GetBitAtPos(valuePos, planeMask)) {
                                // RLE encoding
                                // encode run of count 0's followed by a 1
                                // with codeword: 1<count>(signBits[signPos])
                                SetBit(m_codeBuffer, codePos++);
                                if (k > 0) {
                                        SetValueBlock(m_codeBuffer, codePos, count, k);
                                        codePos += k;

                                        // adapt k (half the zero run-length)
                                        k--;
                                        runlen >>= 1;
                                }

                                // copy and write sign bit
                                if (m_value[valuePos] < 0) {
                                        SetBit(signBits, signLen++);
                                        SetBit(m_codeBuffer, codePos++);
                                } else {
                                        ClearBit(signBits, signLen++);
                                        ClearBit(m_codeBuffer, codePos++);
                                }

                                // write a 1 to sigBits
                                SetBit(sigBits, sigPos++);

                                // update m_sigFlagVector
                                m_sigFlagVector[valuePos] = true;

                                // prepare for next run
                                count = 0;
                        } else {
                                // RLE encoding
                                count++;
                                if (count == runlen) {
                                        // encode run of 2^k zeros by a single 0
                                        ClearBit(m_codeBuffer, codePos++);
                                        // adapt k (double the zero run-length)
                                        if (k < WordWidth) {
                                                k++;
                                                runlen <<= 1;
                                        }

                                        // prepare for next run
                                        count = 0;
                                }

                                // write 0 to sigBits
                                sigPos++;
                        }
                        valuePos++;
                }
                // refinement bit
                if (valuePos < bufferSize) {
                        // write one refinement bit
                        if (GetBitAtPos(valuePos++, planeMask)) {
                                SetBit(refBits, refPos);
                        } else {
                                ClearBit(refBits, refPos);
                        }
                        refPos++;
                }
        }
        // RLE encoding of the rest of the plane
        // encode run of count 0's followed by a 1
        // with codeword: 1<count>(signBits[signPos])
        SetBit(m_codeBuffer, codePos++);
        if (k > 0) {
                SetValueBlock(m_codeBuffer, codePos, count, k);
                codePos += k;
        }
        // write dmmy sign bit
        SetBit(m_codeBuffer, codePos++);

        // write word filler zeros

        ASSERT(sigPos <= bufferSize);
        ASSERT(refPos <= bufferSize);
        ASSERT(signLen <= bufferSize);
        ASSERT(valuePos == bufferSize);
        ASSERT(codePos >= outStartPos && codePos < CodeBufferBitLen);
        codeLen = codePos - outStartPos;

        return sigPos;
}
```

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

Definition at line 92 of file Encoder.h.

```
{ 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 67 of file Encoder.h.

```
                                               {                         // initialize for
reusage
                        m_valuePos = 0;
                        m_maxAbsValue = 0;
                        m_codePos = 0;
                        m_lastLevelIndex = lastLevelIndex;
                }
```

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

Definition at line 736 of file Encoder.cpp.

```
                                              {
        UINT8 cnt = 0;

        // determine number of bitplanes for max value
        if (m_maxAbsValue > 0) {
                while (m_maxAbsValue > 0) {
                        m_maxAbsValue >>= 1; cnt++;
                }
                if (cnt == MaxBitPlanes + 1) cnt = 0;
                // end cs
                ASSERT(cnt <= MaxBitPlanes);
                ASSERT((cnt >> MaxBitPlanesLog) == 0);
                return cnt;
        } else {
                return 1;
        }
}
```

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

Definition at line 760 of file Encoder.cpp.

```
                                                                                      {
        ASSERT(signBits);
        ASSERT(0 <= codePos && codePos < CodeBufferBitLen);
        ASSERT(0 < signLen && signLen <= BufferSize);

        const UINT32  outStartPos = codePos;
        UINT32 k = 0;
        UINT32 runlen = 1 << k; // = 2^k
        UINT32 count = 0;
        UINT32 signPos = 0;

        while (signPos < signLen) {
                // search next 0 in signBits starting at position signPos
                count = SeekBit1Range(signBits, signPos, __min(runlen, signLen - signPos));
                // count 1's found
                if (count == runlen) {
                        // encode run of 2^k ones by a single 1
                        signPos += count;
                        SetBit(m_codeBuffer, codePos++);
                        // adapt k (double the 1's run-length)
                        if (k < WordWidth) {
                                k++;
                                runlen <<= 1;
                        }
                } else {
                        // encode run of count 1's followed by a 0
                        // with codeword: 0(count)
                        signPos += count + 1;
                        ClearBit(m_codeBuffer, codePos++);
                        if (k > 0) {
                                SetValueBlock(m_codeBuffer, codePos, count, k);
                                codePos += k;
                        }
                        // adapt k (half the 1's run-length)
                        if (k > 0) {
                                k--;
                                runlen >>= 1;
                        }
                }
        }
        ASSERT(signPos == signLen || signPos == signLen + 1);
        ASSERT(codePos >= outStartPos && codePos < CodeBufferBitLen);
        return codePos - outStartPos;
}
```

Member Data Documentation

UINT32 CEncoder::CMacroBlock::m_codeBuffer[CodeBufferLen]

output buffer for encoded bitstream

Definition at line 81 of file Encoder.h.

UINT32 CEncoder::CMacroBlock::m_codePos

current position in encoded bitstream

Definition at line 85 of file Encoder.h.

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

Definition at line 94 of file Encoder.h.

ROIBlockHeader CEncoder::CMacroBlock::m_header

block header

Definition at line 82 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 86 of file Encoder.h.

UINT32 CEncoder::CMacroBlock::m_maxAbsValue

maximum absolute coefficient in each buffer

Definition at line 84 of file Encoder.h.

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

Definition at line 95 of file Encoder.h.

DataT CEncoder::CMacroBlock::m_value[BufferSize]

input buffer of values with index m_valuePos

Definition at line 80 of file Encoder.h.

UINT32 CEncoder::CMacroBlock::m_valuePos

current buffer position

Definition at line 83 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** (**CDecoder** *decoder)
- 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

- **CDecoder** * **m_decoder**
- 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 (CDecoder **decoder*) `[inline]`

Constructor: Initializes new macro block.

Parameters:

decoder	Pointer to outer class.

Definition at line 56 of file Decoder.h.

```
                : m_header(0)                                                   //
makes sure that IsCompletelyRead() returns true for an empty macro block
                , m_valuePos(0)
                , m_decoder(decoder)
                {
                        ASSERT(m_decoder);
                }
```

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 618 of file Decoder.cpp.

```
                                         {
        UINT32 bufferSize = m_header.rbh.bufferSize; ASSERT(bufferSize <= BufferSize);

        UINT32 nPlanes;
        UINT32 codePos = 0, codeLen, sigLen, sigPos, signLen, signPos;
        DataT planeMask;

        // clear significance vector
        for (UINT32 k=0; k < bufferSize; k++) {
                m_sigFlagVector[k] = false;
        }
        m_sigFlagVector[bufferSize] = true; // sentinel

        // clear output buffer
        for (UINT32 k=0; k < BufferSize; k++) {
                m_value[k] = 0;
        }

        // read number of bit planes
        // <nPlanes>
        nPlanes = GetValueBlock(m_codeBuffer, 0, MaxBitPlanesLog);
        codePos += MaxBitPlanesLog;

        // loop through all bit planes
        if (nPlanes == 0) nPlanes = MaxBitPlanes + 1;
        ASSERT(0 < nPlanes && nPlanes <= MaxBitPlanes + 1);
        planeMask = 1 << (nPlanes - 1);

        for (int plane = nPlanes - 1; plane >= 0; plane--) {
                // read RL code
                if (GetBit(m_codeBuffer, codePos)) {
                        // RL coding of sigBits is used
                        // <1><codeLen><codedSigAndSignBits>_<refBits>
                        codePos++;

                        // read codeLen
                        codeLen = GetValueBlock(m_codeBuffer, codePos, RLblockSizeLen);
ASSERT(codeLen <= MaxCodeLen);

                        // position of encoded sigBits and signBits
                        sigPos = codePos + RLblockSizeLen; ASSERT(sigPos < CodeBufferBitLen);

                        // refinement bits
                        codePos = AlignWordPos(sigPos + codeLen); ASSERT(codePos <
CodeBufferBitLen);

                        // run-length decode significant bits and signs from m_codeBuffer and
                        // read refinement bits from m_codeBuffer and compose bit plane
                        sigLen = ComposeBitplaneRLD(bufferSize, planeMask, sigPos,
&m_codeBuffer[codePos >> WordWidthLog]);

                } else {
                        // no RL coding is used for sigBits and signBits together
                        // <0><sigLen>
                        codePos++;

                        // read sigLen
                        sigLen = GetValueBlock(m_codeBuffer, codePos, RLblockSizeLen);
ASSERT(sigLen <= MaxCodeLen);
                        codePos += RLblockSizeLen; ASSERT(codePos < CodeBufferBitLen);

                        // read RL code for signBits
                        if (GetBit(m_codeBuffer, codePos)) {
                                // RL coding is used just for signBits
                                // <1><codeLen><codedSignBits>_<sigBits>_<refBits>
                                codePos++;

                                // read codeLen
                                codeLen = GetValueBlock(m_codeBuffer, codePos, RLblockSizeLen);
ASSERT(codeLen <= MaxCodeLen);

                                // sign bits
                                signPos = codePos + RLblockSizeLen; ASSERT(signPos <
CodeBufferBitLen);

                                // significant bits
                                sigPos = AlignWordPos(signPos + codeLen); ASSERT(sigPos <
CodeBufferBitLen);

                                // refinement bits
                                codePos = AlignWordPos(sigPos + sigLen); ASSERT(codePos <
CodeBufferBitLen);

                                // read significant and refinement bitset from m_codeBuffer
                                sigLen = ComposeBitplaneRLD(bufferSize, planeMask,
&m_codeBuffer[sigPos >> WordWidthLog], &m_codeBuffer[codePos >> WordWidthLog], signPos);

                        } else {
                                // RL coding of signBits was not efficient and therefore not used
                                // <0><signLen>_<signBits>_<sigBits>_<refBits>
                                codePos++;

                                // read signLen
                                signLen = GetValueBlock(m_codeBuffer, codePos, RLblockSizeLen);
ASSERT(signLen <= MaxCodeLen);

                                // sign bits
                                signPos = AlignWordPos(codePos + RLblockSizeLen); ASSERT(signPos
< CodeBufferBitLen);

                                // significant bits
                                sigPos = AlignWordPos(signPos + signLen); ASSERT(sigPos <
CodeBufferBitLen);

                                // refinement bits
                                codePos = AlignWordPos(sigPos + sigLen); ASSERT(codePos <
CodeBufferBitLen);

                                // read significant and refinement bitset from m_codeBuffer
                                sigLen = ComposeBitplane(bufferSize, planeMask,
&m_codeBuffer[sigPos >> WordWidthLog], &m_codeBuffer[codePos >> WordWidthLog],
&m_codeBuffer[signPos >> WordWidthLog]);
                        }
                }

                // start of next chunk
                codePos = AlignWordPos(codePos + bufferSize - sigLen); ASSERT(codePos <
CodeBufferBitLen);

                // next plane
                planeMask >>= 1;
```

```
        }

        m_valuePos = 0;
}
```

UINT32 CDecoder::CMacroBlock::ComposeBitplane (UINT32bufferSize, DataTplaneMask, UINT32 *sigBits, UINT32 *refBits, UINT32 *signBits) `[private]`

Definition at line 733 of file Decoder.cpp.

```
{
        ASSERT(sigBits);
        ASSERT(refBits);
        ASSERT(signBits);

        UINT32 valPos = 0, signPos = 0, refPos = 0;
        UINT32 sigPos = 0, sigEnd;
        UINT32 zerocnt;

        while (valPos < bufferSize) {
                // search next 1 in m_sigFlagVector using searching with sentinel
                sigEnd = valPos;
                while(!m_sigFlagVector[sigEnd]) { sigEnd++; }
                sigEnd -= valPos;
                sigEnd += sigPos;

                // search 1's in sigBits[sigPos..sigEnd)
                // these 1's are significant bits
                while (sigPos < sigEnd) {
                        // search 0's
                        zerocnt = SeekBitRange(sigBits, sigPos, sigEnd - sigPos);
                        sigPos += zerocnt;
                        valPos += zerocnt;
                        if (sigPos < sigEnd) {
                                // write bit to m_value
                                SetBitAtPos(valPos, planeMask);

                                // copy sign bit
                                SetSign(valPos, GetBit(signBits, signPos++));

                                // update significance flag vector
                                m_sigFlagVector[valPos++] = true;
                                sigPos++;
                        }
                }
                // refinement bit
                if (valPos < bufferSize) {
                        // write one refinement bit
                        if (GetBit(refBits, refPos)) {
                                SetBitAtPos(valPos, planeMask);
                        }
                        refPos++;
                        valPos++;
                }
        }
        ASSERT(sigPos <= bufferSize);
        ASSERT(refPos <= bufferSize);
        ASSERT(signPos <= bufferSize);
        ASSERT(valPos == bufferSize);

        return sigPos;
}
```

UINT32 CDecoder::CMacroBlock::ComposeBitplaneRLD (UINT32bufferSize, DataTplaneMask, UINT32sigPos, UINT32 *refBits) `[private]`

Definition at line 796 of file Decoder.cpp.

```
{
        ASSERT(refBits);

        UINT32 valPos = 0, refPos = 0;
        UINT32 sigPos = 0, sigEnd;
        UINT32 k = 3;
        UINT32 runlen = 1 << k; // = 2^k
        UINT32 count = 0, rest = 0;
        bool set1 = false;

        while (valPos < bufferSize) {
                // search next 1 in m_sigFlagVector using searching with sentinel
                sigEnd = valPos;
                while(!m_sigFlagVector[sigEnd]) { sigEnd++; }
                sigEnd -= valPos;
                sigEnd += sigPos;

                while (sigPos < sigEnd) {
                        if (rest || set1) {
                                // rest of last run
                                sigPos += rest;
                                valPos += rest;
                                rest = 0;
                        } else {
                                // decode significant bits
                                if (GetBit(m_codeBuffer, codePos++)) {
                                        // extract counter and generate zero run of length count
                                        if (k > 0) {
                                                // extract counter
                                                count = GetValueBlock(m_codeBuffer, codePos, k);
                                                codePos += k;
                                                if (count > 0) {
                                                        sigPos += count;
                                                        valPos += count;
                                                }

                                                // adapt k (half run-length interval)
                                                k--;
                                                runlen >>= 1;
                                        }

                                        set1 = true;

                                } else {
                                        // generate zero run of length 2^k
                                        sigPos += runlen;
                                        valPos += runlen;

                                        // adapt k (double run-length interval)
                                        if (k < WordWidth) {
                                                k++;
                                                runlen <<= 1;
                                        }
                                }
                        }

                        if (sigPos < sigEnd) {
                                if (set1) {
                                        set1 = false;

                                        // write 1 bit
                                        SetBitAtPos(valPos, planeMask);

                                        // set sign bit
                                        SetSign(valPos, GetBit(m_codeBuffer, codePos++));

                                        // update significance flag vector
                                        m_sigFlagVector[valPos++] = true;
                                        sigPos++;
                                }
                        } else {
                                rest = sigPos - sigEnd;
                                sigPos = sigEnd;
                                valPos -= rest;
                        }

                }

                // refinement bit
                if (valPos < bufferSize) {
                        // write one refinement bit
                        if (GetBit(refBits, refPos)) {
                                SetBitAtPos(valPos, planeMask);
                        }
                        refPos++;
                        valPos++;
                }
        }
        ASSERT(sigPos <= bufferSize);
        ASSERT(refPos <= bufferSize);
        ASSERT(valPos == bufferSize);

        return sigPos;
}
```

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

Definition at line 899 of file Decoder.cpp.

```
{
        ASSERT(sigBits);
        ASSERT(refBits);

        UINT32 valPos = 0, refPos = 0;
        UINT32 sigPos = 0, sigEnd;
        UINT32 zerocnt, count = 0;
        UINT32 k = 0;
        UINT32 runlen = 1 << k; // = 2^k
        bool signBit = false;
        bool zeroAfterRun = false;

        while (valPos < bufferSize) {
                // search next 1 in m_sigFlagVector using searching with sentinel
                sigEnd = valPos;
                while(!m_sigFlagVector[sigEnd]) { sigEnd++; }
                sigEnd -= valPos;
                sigEnd += sigPos;

                // search 1's in sigBits[sigPos..sigEnd)
                // these 1's are significant bits
                while (sigPos < sigEnd) {
                        // search 0's
                        zerocnt = SeekBitRange(sigBits, sigPos, sigEnd - sigPos);
                        sigPos += zerocnt;
                        valPos += zerocnt;
                        if (sigPos < sigEnd) {
                                // write bit to m_value
                                SetBitAtPos(valPos, planeMask);

                                // check sign bit
                                if (count == 0) {
                                        // all 1's have been set
                                        if (zeroAfterRun) {
                                                // finish the run with a 0
                                                signBit = false;
                                                zeroAfterRun = false;
                                        } else {
                                                // decode next sign bit
                                                if (GetBit(m_codeBuffer, signPos++)) {
                                                        // generate 1's run of length 2^k
                                                        count = runlen - 1;
                                                        signBit = true;

                                                        // adapt k (double run-length interval)
                                                        if (k < WordWidth) {
                                                                k++;
                                                                runlen <<= 1;
                                                        }
                                                } else {
                                                        // extract counter and generate 1's run
of length count
                                                        if (k > 0) {
                                                                // extract counter
                                                                count =
GetValueBlock(m_codeBuffer, signPos, k);
                                                                signPos += k;

                                                                // adapt k (half run-length
interval)
                                                                k--;
                                                                runlen >>= 1;
                                                        }
                                                        if (count > 0) {
                                                                count--;
                                                                signBit = true;
                                                                zeroAfterRun = true;
                                                        } else {
                                                                signBit = false;
                                                        }
                                                }
                                        }
                                } else {
                                        ASSERT(count > 0);
                                        ASSERT(signBit);
                                        count--;
                                }

                                // copy sign bit
                                SetSign(valPos, signBit);

                                // update significance flag vector
                                m_sigFlagVector[valPos++] = true;
                                sigPos++;
                        }
                }

                // refinement bit
                if (valPos < bufferSize) {
                        // write one refinement bit
                        if (GetBit(refBits, refPos)) {
                                SetBitAtPos(valPos, planeMask);
                        }
                        refPos++;
                        valPos++;
                }
        }
        ASSERT(sigPos <= bufferSize);
        ASSERT(refPos <= bufferSize);
        ASSERT(valPos == bufferSize);

        return sigPos;
}
```

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 67 of file Decoder.h.

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

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

Definition at line 84 of file Decoder.h.

```
{ (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 85 of file Decoder.h.

```
{ 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 77 of file Decoder.h.

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

Definition at line 87 of file Decoder.h.

ROIBlockHeader CDecoder::CMacroBlock::m_header

block header

Definition at line 75 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 76 of file Decoder.h.

UINT32 CDecoder::CMacroBlock::m_valuePos

current position in m_value

Definition at line 78 of file Decoder.h.

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

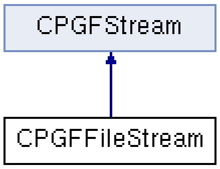
- **Decoder.h**
- **Decoder.cpp**

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) THROW_
- virtual void **Read** (int *count, void *buffer) THROW_
- virtual void **SetPos** (short posMode, INT64 posOff) THROW_
- virtual UINT64 **GetPos** () const THROW_
- 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.

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

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

Constructor

Parameters:

hFile	File handle

Definition at line 90 of file PGFstream.h.

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

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

Definition at line 94 of file PGFstream.h.

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

Member Function Documentation

HANDLE CPGFFileStream::GetHandle () `[inline]`

Returns:

File handle

Definition at line 92 of file PGFstream.h.

```
{ return m_hFile; }
```

UINT64 CPGFFileStream::GetPos () const `[virtual]`

Get current stream position.

Returns:

Current stream position

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

Definition at line 64 of file PGFstream.cpp.

```
                                       {
        ASSERT(IsValid());
        OSError err;
        UINT64 pos = 0;
        if ((err = GetFPos(m_hFile, &pos)) != NoError) ReturnWithError2(err, pos);
        return pos;
}
```

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

Check stream validity.

Returns:

True if stream and current position is valid

Implements **CPGFStream** (*p.110*).

Definition at line 99 of file PGFstream.h.

```
{ 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.110*).

Definition at line 48 of file PGFstream.cpp.

```
                                                       {
        ASSERT(count);
        ASSERT(buffPtr);
        ASSERT(IsValid());
        OSError err;
        if ((err = FileRead(m_hFile, count, buffPtr)) != NoError) ReturnWithError(err);
}
```

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.110*).

Definition at line 57 of file PGFstream.cpp.

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

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.110*).

Definition at line 38 of file PGFstream.cpp.

```
                                                                         {
        ASSERT(count);
        ASSERT(buffPtr);
        ASSERT(IsValid());
        OSError err;
        if ((err = FileWrite(m_hFile, count, buffPtr)) != NoError) ReturnWithError(err);

}
```

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** ()
- virtual **~CPGFImage** ()
- virtual void **Close** ()
- virtual void **Destroy** ()
- void **Open** (**CPGFStream** *stream) THROW_
- bool **IsOpen** () const
- void **Read** (int level=0, CallbackPtr cb=NULL, void *data=NULL) THROW_
- void **Read** (**PGFRect** &rect, int level=0, CallbackPtr cb=NULL, void *data=NULL) THROW_
- void **ReadPreview** () THROW_
- void **Reconstruct** (int level=0) THROW_
- void **GetBitmap** (int pitch, UINT8 *buff, BYTE bpp, int channelMap[]=NULL, CallbackPtr cb=NULL, void *data=NULL) const THROW_
- void **GetYUV** (int pitch, **DataT** *buff, BYTE bpp, int channelMap[]=NULL, CallbackPtr cb=NULL, void *data=NULL) const THROW_
- void **ImportBitmap** (int pitch, UINT8 *buff, BYTE bpp, int channelMap[]=NULL, CallbackPtr cb=NULL, void *data=NULL) THROW_
- void **ImportYUV** (int pitch, **DataT** *buff, BYTE bpp, int channelMap[]=NULL, CallbackPtr cb=NULL, void *data=NULL) THROW_
- void **Write** (**CPGFStream** *stream, UINT32 *nWrittenBytes=NULL, CallbackPtr cb=NULL, void *data=NULL) THROW_
- UINT32 **WriteHeader** (**CPGFStream** *stream) THROW_
- UINT32 **WriteImage** (**CPGFStream** *stream, CallbackPtr cb=NULL, void *data=NULL) THROW_
- UINT32 **Write** (int level, CallbackPtr cb=NULL, void *data=NULL) THROW_
- void **ConfigureEncoder** (bool useOMP=true, bool favorSpeedOverSize=false)
- void **ConfigureDecoder** (bool useOMP=true, bool skipUserData=false)
- void **ResetStreamPos** () THROW_
- void **SetChannel** (**DataT** *channel, int c=0)
- void **SetHeader** (const **PGFHeader** &header, BYTE flags=0, UINT8 *userData=0, UINT32 userDataLength=0) THROW_
- void **SetMaxValue** (UINT32 maxValue)
- void **SetProgressMode** (**ProgressMode** pm)
- void **SetRefreshCallback** (**RefreshCB** callback, void *arg)
- void **SetColorTable** (UINT32 iFirstColor, UINT32 nColors, const RGBQUAD *prgbColors) THROW_
- **DataT** * **GetChannel** (int c=0)
- void **GetColorTable** (UINT32 iFirstColor, UINT32 nColors, RGBQUAD *prgbColors) const THROW_
- const RGBQUAD * **GetColorTable** () const
- const **PGFHeader** * **GetHeader** () const
- UINT32 **GetMaxValue** () const
- UINT64 **GetUserDataPos** () const
- const UINT8 * **GetUserData** (UINT32 &size) const
- UINT32 **GetEncodedHeaderLength** () const
- UINT32 **GetEncodedLevelLength** (int level) const
- UINT32 **ReadEncodedHeader** (UINT8 *target, UINT32 targetLen) const THROW_
- UINT32 **ReadEncodedData** (int level, UINT8 *target, UINT32 targetLen) const THROW_
- 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
- BYTE **Quality** () const
- BYTE **Channels** () const
- BYTE **Mode** () const
- BYTE **BPP** () const
- bool **ROIisSupported** () const
- BYTE **UsedBitsPerChannel** () const
- BYTE **Version** () const

Static Public Member Functions

- static bool **ImportIsSupported** (BYTE mode)
- static UINT32 **LevelWidth** (UINT32 width, int level)
- static UINT32 **LevelHeight** (UINT32 height, int level)
- static BYTE **CurrentVersion** (BYTE version=PGFVersion)
- static BYTE **CurrentChannelDepth** (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
- 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_skipUserData**
  skip user data (metadata) during open
- bool **m_streamReinitialized**
  stream has been reinitialized
- **PGFRect m_roi**
  region of interest

Private Member Functions

- void **ComputeLevels** ()
- void **CompleteHeader** ()
- void **RgbToYuv** (int pitch, UINT8 *rgbBuff, BYTE bpp, int channelMap[], CallbackPtr cb, void *data) THROW_
- void **Downsample** (int nChannel)
- UINT32 **UpdatePostHeaderSize** () THROW_
- void **WriteLevel** () THROW_
- 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: pgf.Open(...) pgf.Read(...) pgf.GetBitmap(...) Encoding: pgf.SetHeader(...) pgf.ImportBitmap(...) pgf.Write(...)

Author:

C. Stamm, R. Spuler

Definition at line 57 of file PGFimage.h.

Constructor & Destructor Documentation

CPGFImage::CPGFImage ()

Standard constructor: It is used to create a PGF instance for opening and reading.

Definition at line 55 of file PGFimage.cpp.

```
: m_decoder(0)
, m_encoder(0)
, m_levelLength(0)
, m_quant(0)
, m_userDataPos(0)
, m_downsample(false)
, m_favorSpeedOverSize(false)
, m_useOMPinEncoder(true)
, m_useOMPinDecoder(true)
, m_skipUserData(false)
#ifdef __PGFROISUPPORT__
, m_streamReinitialized(false)
#endif
, m_cb(0)
, m_cbArg(0)
, m_progressMode(PM_Relative)
, m_percent(0)
{

        // init preHeader
        memcpy(m_preHeader.magic, Magic, 3);
        m_preHeader.version = PGFVersion;
        m_preHeader.hSize = 0;

        // init postHeader
        m_postHeader.userData = 0;
        m_postHeader.userDataLen = 0;

        // init channels
        for (int i=0; i < MaxChannels; i++) {
                m_channel[i] = 0;
                m_wtChannel[i] = 0;
        }

        // set image width and height
        m_width[0] = 0;
        m_height[0] = 0;
}
```

CPGFImage::~CPGFImage () `[virtual]`

Destructor: Destroy internal data structures.

Definition at line 97 of file PGFimage.cpp.

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

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 460 of file PGFimage.h.

```
{ 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 409 of file PGFimage.h.

```
{ return CurrentChannelDepth(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 404 of file PGFimage.h.

```
{ 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 447 of file PGFimage.h.

```
{ 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 397 of file PGFimage.h.

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

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

Definition at line 561 of file PGFimage.h.

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

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

Definition at line 564 of file PGFimage.h.

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

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

Definition at line 551 of file PGFimage.h.

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

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

Definition at line 554 of file PGFimage.h.

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

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

Definition at line 557 of file PGFimage.h.

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

void CPGFImage::Close () `[virtual]`

Close PGF image after opening and reading. Destructor calls this method during destruction.

Definition at line 121 of file PGFimage.cpp.

```
                      {
        delete m_decoder; m_decoder = 0;
}
```

void CPGFImage::CompleteHeader () `[private]`

Definition at line 207 of file PGFimage.cpp.

```
                               {
        if (m_header.mode == ImageModeUnknown) {
                // undefined mode
                switch(m_header.bpp) {
                case 1: m_header.mode = ImageModeBitmap; break;
                case 8: m_header.mode = ImageModeGrayScale; break;
                case 12: m_header.mode = ImageModeRGB12; break;
                case 16: m_header.mode = ImageModeRGB16; break;
                case 24: m_header.mode = ImageModeRGBColor; break;
                case 32: m_header.mode = ImageModeRGBA; break;
                case 48: m_header.mode = ImageModeRGB48; break;
                default: m_header.mode = ImageModeRGBColor; break;
                }
        }
        if (!m_header.bpp) {
                // undefined bpp
                switch(m_header.mode) {
                case ImageModeBitmap:
                        m_header.bpp = 1;
                        break;
                case ImageModeIndexedColor:
                case ImageModeGrayScale:
                        m_header.bpp = 8;
                        break;
                case ImageModeRGB12:
                        m_header.bpp = 12;
                        break;
                case ImageModeRGB16:
```

```
                case ImageModeGray16:
                        m_header.bpp = 16;
                        break;
                case ImageModeRGBColor:
                case ImageModeLabColor:
                        m_header.bpp = 24;
                        break;
                case ImageModeRGBA:
                case ImageModeCMYKColor:
                case ImageModeGray32:
                        m_header.bpp = 32;
                        break;
                case ImageModeRGB48:
                case ImageModeLab48:
                        m_header.bpp = 48;
                        break;
                case ImageModeCMYK64:
                        m_header.bpp = 64;
                        break;
                default:
                        ASSERT(false);
                        m_header.bpp = 24;
                }
        }
        if (m_header.mode == ImageModeRGBColor && m_header.bpp == 32) {
                // change mode
                m_header.mode = ImageModeRGBA;
        }
        ASSERT(m_header.mode != ImageModeBitmap || m_header.bpp == 1);
        ASSERT(m_header.mode != ImageModeIndexedColor || m_header.bpp == 8);
        ASSERT(m_header.mode != ImageModeGrayScale || m_header.bpp == 8);
        ASSERT(m_header.mode != ImageModeGray16 || m_header.bpp == 16);
        ASSERT(m_header.mode != ImageModeGray32 || m_header.bpp == 32);
        ASSERT(m_header.mode != ImageModeRGBColor || m_header.bpp == 24);
        ASSERT(m_header.mode != ImageModeRGBA || m_header.bpp == 32);
        ASSERT(m_header.mode != ImageModeRGB12 || m_header.bpp == 12);
        ASSERT(m_header.mode != ImageModeRGB16 || m_header.bpp == 16);
        ASSERT(m_header.mode != ImageModeRGB48 || m_header.bpp == 48);
        ASSERT(m_header.mode != ImageModeLabColor || m_header.bpp == 24);
        ASSERT(m_header.mode != ImageModeLab48 || m_header.bpp == 48);
        ASSERT(m_header.mode != ImageModeCMYKColor || m_header.bpp == 32);
        ASSERT(m_header.mode != ImageModeCMYK64 || m_header.bpp == 64);

        // set number of channels
        if (!m_header.channels) {
                switch(m_header.mode) {
                case ImageModeBitmap:
                case ImageModeIndexedColor:
                case ImageModeGrayScale:
                case ImageModeGray16:
                case ImageModeGray32:
                        m_header.channels = 1;
                        break;
                case ImageModeRGBColor:
                case ImageModeRGB12:
                case ImageModeRGB16:
                case ImageModeRGB48:
                case ImageModeLabColor:
                case ImageModeLab48:
                        m_header.channels = 3;
                        break;
                case ImageModeRGBA:
                case ImageModeCMYKColor:
                case ImageModeCMYK64:
                        m_header.channels = 4;
                        break;
                default:
                        ASSERT(false);
                        m_header.channels = 3;
                }
        }

        // store used bits per channel
        UINT8 bpc = m_header.bpp/m_header.channels;
        if (bpc > 31) bpc = 31;
        if (!m_header.usedBitsPerChannel || m_header.usedBitsPerChannel > bpc) {
                m_header.usedBitsPerChannel = bpc;
        }
}
```

void CPGFImage::ComputeLevels () `[private]`

Definition at line 798 of file PGFimage.cpp.

```
                      {
        const int maxThumbnailWidth = 20*FilterWidth;
        const int m = __min(m_header.width, m_header.height);
        int s = m;

        if (m_header.nLevels < 1 || m_header.nLevels > MaxLevel) {
                m_header.nLevels = 1;
                // compute a good value depending on the size of the image
                while (s > maxThumbnailWidth) {
                        m_header.nLevels++;
                        s = s/2;
                }
        }

        int levels = m_header.nLevels; // we need a signed value during level reduction

        // reduce number of levels if the image size is smaller than FilterWidth*2^levels
        s = FilterWidth*(1 << levels);  // must be at least the double filter size because of
subsampling
        while (m < s) {
                levels--;
                s = s/2;
        }
        if (levels > MaxLevel) m_header.nLevels = MaxLevel;
        else if (levels < 0) m_header.nLevels = 0;
        else m_header.nLevels = (UINT8)levels;

        // used in Write when PM_Absolute
        m_percent = pow(0.25, m_header.nLevels);

        ASSERT(0 <= m_header.nLevels && m_header.nLevels <= MaxLevel);
}
```

void CPGFImage::ConfigureDecoder (bool*useOMP* = `true`, bool*skipUserData* = `false`) `[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.
skipUserData	The file might contain user data (metadata). User data ist usually read during Open and stored in memory. Set this flag to false when storing in memory is not needed.

Definition at line 266 of file PGFimage.h.

```
{ m_useOMPinDecoder = useOMP; m_skipUserData = skipUserData; }
```

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 260 of file PGFimage.h.

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

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

Compute and return codec version.

Returns:

current PGF codec version

Definition at line 508 of file PGFimage.h.

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

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

Compute and return codec version.

Returns:

current PGF codec version

Return version

Definition at line 714 of file PGFimage.cpp.

```
                                           {
        if (version & Version6) return 6;
        if (version & Version5) return 5;
        if (version & Version2) return 2;
        return 1;
}
```

void CPGFImage::Destroy () `[virtual]`

Destroy internal data structures. Destructor calls this method during destruction.

Definition at line 104 of file PGFimage.cpp.

```
                        {
        Close();

        for (int i=0; i < m_header.channels; i++) {
                delete m_wtChannel[i]; m_wtChannel[i]=0; // also deletes m_channel
                m_channel[i] = 0;
        }
        delete[] m_postHeader.userData; m_postHeader.userData = 0; m_postHeader.userDataLen = 0;
        delete[] m_levelLength; m_levelLength = 0;
        delete m_encoder; m_encoder = NULL;

        m_userDataPos = 0;
}
```

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

Definition at line 754 of file PGFimage.cpp.

```
                                 {
        ASSERT(ch > 0);

        const int w = m_width[0];
        const int w2 = w/2;
        const int h2 = m_height[0]/2;
        const int oddW = w%2;                              // don't use bool -> problems with MaxSpeed
optimization
        const int oddH = m_height[0]%2;          // "
        int loPos = 0;
        int hiPos = w;
        int sampledPos = 0;
        DataT* buff = m_channel[ch]; ASSERT(buff);

        for (int i=0; i < h2; i++) {
                for (int j=0; j < w2; j++) {
                        // compute average of pixel block
                        buff[sampledPos] = (buff[loPos] + buff[loPos + 1] + buff[hiPos] +
buff[hiPos + 1]) >> 2;
                        loPos += 2; hiPos += 2;
                        sampledPos++;
                }
                if (oddW) {
                        buff[sampledPos] = (buff[loPos] + buff[hiPos]) >> 1;
                        loPos++; hiPos++;
                        sampledPos++;
                }
                loPos += w; hiPos += w;
        }
        if (oddH) {
                for (int j=0; j < w2; j++) {
                        buff[sampledPos] = (buff[loPos] + buff[loPos+1]) >> 1;
                        loPos += 2; hiPos += 2;
                        sampledPos++;
                }
                if (oddW) {
                        buff[sampledPos] = buff[loPos];
                }
        }

        // downsampled image has half width and half height
        m_width[ch] = (m_width[ch] + 1)/2;
        m_height[ch] = (m_height[ch] + 1)/2;
}
```

void CPGFImage::GetBitmap (int*pitch*, UINT8 **buff*, BYTE*bpp*, int*channelMap*[] = `NULL`, CallbackPtr*cb* = `NULL`, void **data* = `NULL`) 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 1697 of file PGFimage.cpp.

```
{
        ASSERT(buff);
        UINT32 w = m_width[0];
        UINT32 h = m_height[0];
        UINT8* targetBuff = 0;  // used if ROI is used
        UINT8* buffStart = 0;   // used if ROI is used
        int targetPitch = 0;    // used if ROI is used

#ifdef __PGFROISUPPORT__
        const PGFRect& roi = (ROIisSupported()) ? m_wtChannel[0]->GetROI(m_currentLevel) :
PGFRect(0, 0, w, h); // roi is usually larger than m_roi
        const PGFRect levelRoi(LevelWidth(m_roi.left, m_currentLevel), LevelHeight(m_roi.top,
m_currentLevel), LevelWidth(m_roi.Width(), m_currentLevel), LevelHeight(m_roi.Height(),
m_currentLevel));
        ASSERT(w <= roi.Width() && h <= roi.Height());
        ASSERT(roi.left <= levelRoi.left && levelRoi.right <= roi.right);
        ASSERT(roi.top <= levelRoi.top && levelRoi.bottom <= roi.bottom);

        if (ROIisSupported() && (levelRoi.Width() < w || levelRoi.Height() < h)) {
                // ROI is used -> create a temporary image buffer for roi
                // compute pitch
                targetPitch = pitch;
                pitch = AlignWordPos(w*bpp)/8;

                // create temporary output buffer
                targetBuff = buff;
                buff = buffStart = new(std::nothrow) UINT8[pitch*h];
                if (!buff) ReturnWithError(InsufficientMemory);
        }
#endif

        const bool wOdd = (1 == w%2);

        const double dP = 1.0/h;
        int defMap[] = { 0, 1, 2, 3, 4, 5, 6, 7 }; ASSERT(sizeof(defMap)/sizeof(defMap[0]) ==
MaxChannels);
        if (channelMap == NULL) channelMap = defMap;
        int sampledPos = 0, yPos = 0;
        DataT uAvg, vAvg;
        double percent = 0;
        UINT32 i, j;

        switch(m_header.mode) {
        case ImageModeBitmap:
                {
                        ASSERT(m_header.channels == 1);
                        ASSERT(m_header.bpp == 1);
                        ASSERT(bpp == 1);

                        const UINT32 w2 = (w + 7)/8;
                        DataT* y = m_channel[0]; ASSERT(y);

                        for (i=0; i < h; i++) {

                                for (j=0; j < w2; j++) {
                                        buff[j] = Clamp8(y[yPos++] + YUVoffset8);
                                }
                                yPos += w - w2;

                                //UINT32 cnt = w;
                                //for (j=0; j < w2; j++) {
                                //      buff[j] = 0;
                                //      for (int k=0; k < 8; k++) {
                                //              if (cnt) {
                                //                      buff[j] <<= 1;
                                //                      buff[j] |= (1 & (y[yPos++] -
YUVoffset8));
                                //                      cnt--;
                                //              }
                                //      }
                                //}
                                buff += pitch;

                                if (cb) {
                                        percent += dP;
                                        if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
                                }
                        }
                        break;
                }
        case ImageModeIndexedColor:
        case ImageModeGrayScale:
        case ImageModeHSLColor:
        case ImageModeHSBColor:
                {
                        ASSERT(m_header.channels >= 1);
                        ASSERT(m_header.bpp == m_header.channels*8);
                        ASSERT(bpp%8 == 0);

                        int cnt, channels = bpp/8; ASSERT(channels >= m_header.channels);

                        for (i=0; i < h; i++) {
                                cnt = 0;
                                for (j=0; j < w; j++) {
                                        for (int c=0; c < m_header.channels; c++) {
                                                buff[cnt + channelMap[c]] =
Clamp8(m_channel[c][yPos] + YUVoffset8);
                                        }
                                        cnt += channels;
                                        yPos++;
                                }
                                buff += pitch;

                                if (cb) {
                                        percent += dP;
                                        if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
                                }
                        }
                        break;
                }
        case ImageModeGray16:
                {
                        ASSERT(m_header.channels >= 1);
                        ASSERT(m_header.bpp == m_header.channels*16);

                        const DataT yuvOffset16 = 1 << (UsedBitsPerChannel() - 1);
                        int cnt, channels;

                        if (bpp%16 == 0) {
                                const int shift = 16 - UsedBitsPerChannel(); ASSERT(shift >= 0);
                                UINT16 *buff16 = (UINT16 *)buff;
                                int pitch16 = pitch/2;
                                channels = bpp/16; ASSERT(channels >= m_header.channels);

                                for (i=0; i < h; i++) {
                                        cnt = 0;
                                        for (j=0; j < w; j++) {
                                                for (int c=0; c < m_header.channels; c++) {
                                                        buff16[cnt + channelMap[c]] =
Clamp16((m_channel[c][yPos] + yuvOffset16) << shift);
                                                }
                                                cnt += channels;
                                                yPos++;
                                        }
                                        buff16 += pitch16;

                                        if (cb) {
                                                percent += dP;
                                                if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
                                        }
                                }
                        } else {
                                ASSERT(bpp%8 == 0);
                                const int shift = __max(0, UsedBitsPerChannel() - 8);
                                channels = bpp/8; ASSERT(channels >= m_header.channels);

                                for (i=0; i < h; i++) {
                                        cnt = 0;
                                        for (j=0; j < w; j++) {
                                                for (int c=0; c < m_header.channels; c++) {
                                                        buff[cnt + channelMap[c]] = 
Clamp8((m_channel[c][yPos] + yuvOffset16) >> shift);
                                                }
                                                cnt += channels;
                                                yPos++;
                                        }
                                        buff += pitch;

                                        if (cb) {
                                                percent += dP;
                                                if ((*cb)(percent, true, data)) 
ReturnWithError(EscapePressed);
                                        }
                                }
                        }
                        break;
                }
        case ImageModeRGBColor:
                {
                        ASSERT(m_header.channels == 3);
                        ASSERT(m_header.bpp == m_header.channels*8);
                        ASSERT(bpp%8 == 0);
                        ASSERT(bpp >= m_header.bpp);

                        DataT* y = m_channel[0]; ASSERT(y);
                        DataT* u = m_channel[1]; ASSERT(u);
                        DataT* v = m_channel[2]; ASSERT(v);
                        UINT8 *buffg = &buff[channelMap[1]],
                                  *buffr = &buff[channelMap[2]],
                                  *buffb = &buff[channelMap[0]];
                        UINT8 g;
                        int cnt, channels = bpp/8;
                        if(m_downsample){
                                for (i=0; i < h; i++) {
                                        if (i%2) sampledPos -= (w + 1)/2;
                                        cnt = 0;
                                        for (j=0; j < w; j++) {
                                                // image was downsampled
                                                uAvg = u[sampledPos];
                                                vAvg = v[sampledPos];
                                                // Yuv
                                                buffg[cnt] = g = Clamp8(y[yPos] + YUVoffset8 - 
((uAvg + vAvg ) >> 2)); // must be logical shift operator
                                                buffr[cnt] = Clamp8(uAvg + g);
                                                buffb[cnt] = Clamp8(vAvg + g);
                                                yPos++;
                                                cnt += channels;
                                                if (j%2) sampledPos++;
                                        }
                                        buffb += pitch;
                                        buffg += pitch;
                                        buffr += pitch;
                                        if (wOdd) sampledPos++;
                                        if (cb) {
                                                percent += dP;
                                                if ((*cb)(percent, true, data)) 
ReturnWithError(EscapePressed);
                                        }
                                }
                        }else{
                                for (i=0; i < h; i++) {
                                        cnt = 0;
                                        for (j = 0; j < w; j++) {
                                                uAvg = u[yPos];
                                                vAvg = v[yPos];
                                                // Yuv
                                                buffg[cnt] = g = Clamp8(y[yPos] + YUVoffset8 - 
((uAvg + vAvg ) >> 2)); // must be logical shift operator
                                                buffr[cnt] = Clamp8(uAvg + g);
                                                buffb[cnt] = Clamp8(vAvg + g);
                                                yPos++;
                                                cnt += channels;
                                        }
                                        buffb += pitch;
                                        buffg += pitch;
                                        buffr += pitch;

                                        if (cb) {
                                                percent += dP;
                                                if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
                                        }
                                }
                        }
                        break;
                }
        case ImageModeRGB48:
                {
                        ASSERT(m_header.channels == 3);
                        ASSERT(m_header.bpp == 48);

                        const DataT yuvOffset16 = 1 << (UsedBitsPerChannel() - 1);

                        DataT* y = m_channel[0]; ASSERT(y);
                        DataT* u = m_channel[1]; ASSERT(u);
                        DataT* v = m_channel[2]; ASSERT(v);
                        int cnt, channels;
                        DataT g;

                        if (bpp >= 48 && bpp%16 == 0) {
                                const int shift = 16 - UsedBitsPerChannel(); ASSERT(shift >= 0);
                                UINT16 *buff16 = (UINT16 *)buff;
                                int pitch16 = pitch/2;
                                channels = bpp/16; ASSERT(channels >= m_header.channels);

                                for (i=0; i < h; i++) {
                                        if (i%2) sampledPos -= (w + 1)/2;
                                        cnt = 0;
                                        for (j=0; j < w; j++) {
                                                if (m_downsample) {
                                                        // image was downsampled
                                                        uAvg = u[sampledPos];
                                                        vAvg = v[sampledPos];
                                                } else {
                                                        uAvg = u[yPos];
                                                        vAvg = v[yPos];
                                                }
                                                // Yuv
                                                g = y[yPos] + yuvOffset16 - ((uAvg + vAvg ) >> 2);
// must be logical shift operator
                                                buff16[cnt + channelMap[1]] = Clamp16(g <<
shift);
                                                buff16[cnt + channelMap[2]] = Clamp16((uAvg + g)
<< shift);
                                                buff16[cnt + channelMap[0]] = Clamp16((vAvg + g)
<< shift);
                                                yPos++;
                                                cnt += channels;
                                                if (j%2) sampledPos++;
                                        }
                                        buff16 += pitch16;
                                        if (wOdd) sampledPos++;

                                        if (cb) {
                                                percent += dP;
                                                if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
                                        }
                                }
                        } else {
                                ASSERT(bpp%8 == 0);
                                const int shift = __max(0, UsedBitsPerChannel() - 8);
                                channels = bpp/8; ASSERT(channels >= m_header.channels);

                                for (i=0; i < h; i++) {
                                        if (i%2) sampledPos -= (w + 1)/2;
                                        cnt = 0;
                                        for (j=0; j < w; j++) {
                                                if (m_downsample) {
                                                        // image was downsampled
                                                        uAvg = u[sampledPos];
                                                        vAvg = v[sampledPos];
                                                } else {
                                                        uAvg = u[yPos];
                                                        vAvg = v[yPos];
                                                }
                                                // Yuv
                                                g = y[yPos] + yuvOffset16 - ((uAvg + vAvg ) >> 2);
// must be logical shift operator
                                                buff[cnt + channelMap[1]] = Clamp8(g >> shift);
                                                buff[cnt + channelMap[2]] = Clamp8((uAvg + g) >>
shift);
                                                buff[cnt + channelMap[0]] = Clamp8((vAvg + g) >>
shift);
                                                yPos++;
                                                cnt += channels;
                                                if (j%2) sampledPos++;
                                        }
                                        buff += pitch;
                                        if (wOdd) sampledPos++;

                                        if (cb) {
                                                percent += dP;
                                                if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
                                        }
                                }
                        }
                        break;
                }
        case ImageModeLabColor:
                {
                        ASSERT(m_header.channels == 3);
                        ASSERT(m_header.bpp == m_header.channels*8);
                        ASSERT(bpp%8 == 0);

                        DataT* l = m_channel[0]; ASSERT(l);
                        DataT* a = m_channel[1]; ASSERT(a);
                        DataT* b = m_channel[2]; ASSERT(b);
                        int cnt, channels = bpp/8; ASSERT(channels >= m_header.channels);

                        for (i=0; i < h; i++) {
                                if (i%2) sampledPos -= (w + 1)/2;
                                cnt = 0;
                                for (j=0; j < w; j++) {
                                        if (m_downsample) {
                                                // image was downsampled
                                                uAvg = a[sampledPos];
                                                vAvg = b[sampledPos];
                                        } else {
                                                uAvg = a[yPos];
                                                vAvg = b[yPos];
                                        }
                                        buff[cnt + channelMap[0]] = Clamp8(l[yPos] + YUVoffset8);
                                        buff[cnt + channelMap[1]] = Clamp8(uAvg + YUVoffset8);
                                        buff[cnt + channelMap[2]] = Clamp8(vAvg + YUVoffset8);
                                        cnt += channels;
                                        yPos++;
                                        if (j%2) sampledPos++;
                                }
                                buff += pitch;
                                if (wOdd) sampledPos++;

                                if (cb) {
                                        percent += dP;
                                        if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
                                }
                        }
                        break;
                }
        case ImageModeLab48:
                {
                        ASSERT(m_header.channels == 3);
                        ASSERT(m_header.bpp == m_header.channels*16);

                        const DataT yuvOffset16 = 1 << (UsedBitsPerChannel() - 1);

                        DataT* l = m_channel[0]; ASSERT(l);
                        DataT* a = m_channel[1]; ASSERT(a);
                        DataT* b = m_channel[2]; ASSERT(b);
                        int cnt, channels;

                        if (bpp%16 == 0) {
                                const int shift = 16 - UsedBitsPerChannel(); ASSERT(shift >= 0);
                                UINT16 *buff16 = (UINT16 *)buff;
                                int pitch16 = pitch/2;
                                channels = bpp/16; ASSERT(channels >= m_header.channels);

                                for (i=0; i < h; i++) {
                                        if (i%2) sampledPos -= (w + 1)/2;
                                        cnt = 0;
                                        for (j=0; j < w; j++) {
                                                if (m_downsample) {
                                                        // image was downsampled
                                                        uAvg = a[sampledPos];
                                                        vAvg = b[sampledPos];
                                                } else {
                                                        uAvg = a[yPos];
                                                        vAvg = b[yPos];
                                                }
                                                buff16[cnt + channelMap[0]] = Clamp16((l[yPos] +
yuvOffset16) << shift);
                                                buff16[cnt + channelMap[1]] = Clamp16((uAvg +
yuvOffset16) << shift);
                                                buff16[cnt + channelMap[2]] = Clamp16((vAvg +
yuvOffset16) << shift);
                                                cnt += channels;
                                                yPos++;
                                                if (j%2) sampledPos++;
                                        }
                                        buff16 += pitch16;
                                        if (wOdd) sampledPos++;

                                        if (cb) {
                                                percent += dP;
                                                if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
                                        }
                                }
                        } else {
                                ASSERT(bpp%8 == 0);
                                const int shift = __max(0, UsedBitsPerChannel() - 8);
                                channels = bpp/8; ASSERT(channels >= m_header.channels);

                                for (i=0; i < h; i++) {
                                        if (i%2) sampledPos -= (w + 1)/2;
                                        cnt = 0;
                                        for (j=0; j < w; j++) {
                                                if (m_downsample) {
                                                        // image was downsampled
                                                        uAvg = a[sampledPos];
                                                        vAvg = b[sampledPos];
                                                } else {
                                                        uAvg = a[yPos];
                                                        vAvg = b[yPos];
                                                }
                                                buff[cnt + channelMap[0]] = Clamp8((l[yPos] +
yuvOffset16) >> shift);
                                                buff[cnt + channelMap[1]] = Clamp8((uAvg +
yuvOffset16) >> shift);
```

```
                                                buff[cnt + channelMap[2]] = Clamp8((vAvg +
yuvOffset16) >> shift);
                                                cnt += channels;
                                                yPos++;
                                                if (j%2) sampledPos++;
                                        }
                                        buff += pitch;
                                        if (wOdd) sampledPos++;

                                        if (cb) {
                                                percent += dP;
                                                if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
                                        }
                                }
                        }
                        break;
                }
        case ImageModeRGBA:
        case ImageModeCMYKColor:
                {
                        ASSERT(m_header.channels == 4);
                        ASSERT(m_header.bpp == m_header.channels*8);
                        ASSERT(bpp%8 == 0);

                        DataT* y = m_channel[0]; ASSERT(y);
                        DataT* u = m_channel[1]; ASSERT(u);
                        DataT* v = m_channel[2]; ASSERT(v);
                        DataT* a = m_channel[3]; ASSERT(a);
                        UINT8 g, aAvg;
                        int cnt, channels = bpp/8; ASSERT(channels >= m_header.channels);

                        for (i=0; i < h; i++) {
                                if (i%2) sampledPos -= (w + 1)/2;
                                cnt = 0;
                                for (j=0; j < w; j++) {
                                        if (m_downsample) {
                                                // image was downsampled
                                                uAvg = u[sampledPos];
                                                vAvg = v[sampledPos];
                                                aAvg = Clamp8(a[sampledPos] + YUVoffset8);
                                        } else {
                                                uAvg = u[yPos];
                                                vAvg = v[yPos];
                                                aAvg = Clamp8(a[yPos] + YUVoffset8);
                                        }
                                        // Yuv
                                        buff[cnt + channelMap[1]] = g = Clamp8(y[yPos] +
YUVoffset8 - ((uAvg + vAvg ) >> 2)); // must be logical shift operator
                                        buff[cnt + channelMap[2]] = Clamp8(uAvg + g);
                                        buff[cnt + channelMap[0]] = Clamp8(vAvg + g);
                                        buff[cnt + channelMap[3]] = aAvg;
                                        yPos++;
                                        cnt += channels;
                                        if (j%2) sampledPos++;
                                }
                                buff += pitch;
                                if (wOdd) sampledPos++;

                                if (cb) {
                                        percent += dP;
                                        if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
                                }
                        }
                        break;
                }
        case ImageModeCMYK64:
                {
                        ASSERT(m_header.channels == 4);
                        ASSERT(m_header.bpp == 64);

                        const DataT yuvOffset16 = 1 << (UsedBitsPerChannel() - 1);
                        DataT* y = m_channel[0]; ASSERT(y);
                        DataT* u = m_channel[1]; ASSERT(u);
                        DataT* v = m_channel[2]; ASSERT(v);
                        DataT* a = m_channel[3]; ASSERT(a);
                        DataT g, aAvg;
                        int cnt, channels;

                        if (bpp%16 == 0) {
                                const int shift = 16 - UsedBitsPerChannel(); ASSERT(shift >= 0);
                                UINT16 *buff16 = (UINT16 *)buff;
                                int pitch16 = pitch/2;
                                channels = bpp/16; ASSERT(channels >= m_header.channels);

                                for (i=0; i < h; i++) {
                                        if (i%2) sampledPos -= (w + 1)/2;
                                        cnt = 0;
                                        for (j=0; j < w; j++) {
                                                if (m_downsample) {
                                                        // image was downsampled
                                                        uAvg = u[sampledPos];
                                                        vAvg = v[sampledPos];
                                                        aAvg = a[sampledPos] + yuvOffset16;
                                                } else {
                                                        uAvg = u[yPos];
                                                        vAvg = v[yPos];
                                                        aAvg = a[yPos] + yuvOffset16;
                                                }
                                                // Yuv
                                                g = y[yPos] + yuvOffset16 - ((uAvg + vAvg ) >> 2);
// must be logical shift operator
                                                buff16[cnt + channelMap[1]] = Clamp16(g <<
shift);
                                                buff16[cnt + channelMap[2]] = Clamp16((uAvg + g)
<< shift);
                                                buff16[cnt + channelMap[0]] = Clamp16((vAvg + g)
<< shift);
                                                buff16[cnt + channelMap[3]] = Clamp16(aAvg <<
shift);
                                                yPos++;
                                                cnt += channels;
                                                if (j%2) sampledPos++;
                                        }
                                        buff16 += pitch16;
                                        if (wOdd) sampledPos++;

                                        if (cb) {
                                                percent += dP;
                                                if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
                                        }
                                }
                        } else {
                                ASSERT(bpp%8 == 0);
                                const int shift = __max(0, UsedBitsPerChannel() - 8);
                                channels = bpp/8; ASSERT(channels >= m_header.channels);

                                for (i=0; i < h; i++) {
                                        if (i%2) sampledPos -= (w + 1)/2;
                                        cnt = 0;
                                        for (j=0; j < w; j++) {
                                                if (m_downsample) {
                                                        // image was downsampled
                                                        uAvg = u[sampledPos];
                                                        vAvg = v[sampledPos];
                                                        aAvg = a[sampledPos] + yuvOffset16;
                                                } else {
                                                        uAvg = u[yPos];
                                                        vAvg = v[yPos];
                                                        aAvg = a[yPos] + yuvOffset16;
                                                }
                                                // Yuv
                                                g = y[yPos] + yuvOffset16 - ((uAvg + vAvg ) >> 2);
// must be logical shift operator
                                                buff[cnt + channelMap[1]] = Clamp8(g >> shift);
                                                buff[cnt + channelMap[2]] = Clamp8((uAvg + g) >>
shift);
                                                buff[cnt + channelMap[0]] = Clamp8((vAvg + g) >>
shift);
                                                buff[cnt + channelMap[3]] = Clamp8(aAvg >>
shift);
                                                yPos++;
                                                cnt += channels;
                                                if (j%2) sampledPos++;
                                        }
                                        buff += pitch;
                                        if (wOdd) sampledPos++;

                                        if (cb) {
                                                percent += dP;
                                                if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
                                        }
                                }
                        }
                        break;
                }
#ifdef __PGF32SUPPORT__
        case ImageModeGray32:
                {
                        ASSERT(m_header.channels == 1);
                        ASSERT(m_header.bpp == 32);

                        const int yuvOffset31 = 1 << (UsedBitsPerChannel() - 1);

                        DataT* y = m_channel[0]; ASSERT(y);

                        if (bpp == 32) {
                                const int shift = 31 - UsedBitsPerChannel(); ASSERT(shift >= 0);
                                UINT32 *buff32 = (UINT32 *)buff;
                                int pitch32 = pitch/4;

                                for (i=0; i < h; i++) {
                                        for (j=0; j < w; j++) {
                                                buff32[j] = Clamp31((y[yPos++] + yuvOffset31) <<
shift);
                                        }
                                        buff32 += pitch32;

                                        if (cb) {
                                                percent += dP;
                                                if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
                                        }
                                }
                        } else if (bpp == 16) {
                                const int usedBits = UsedBitsPerChannel();
                                UINT16 *buff16 = (UINT16 *)buff;
                                int pitch16 = pitch/2;

                                if (usedBits < 16) {
                                        const int shift = 16 - usedBits;
                                        for (i=0; i < h; i++) {
                                                for (j=0; j < w; j++) {
                                                        buff16[j] = Clamp16((y[yPos++] +
yuvOffset31) << shift);
                                                }
                                                buff16 += pitch16;

                                                if (cb) {
                                                        percent += dP;
                                                        if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
                                                }
                                        }
```

```
				} else {
					const int shift = __max(0, usedBits - 16);
					for (i=0; i < h; i++) {
						for (j=0; j < w; j++) {
							buff16[j] = Clamp16((y[yPos++] + 
yuvOffset31) >> shift);
						}
						buff16 += pitch16;

						if (cb) {
							percent += dP;
							if ((*cb)(percent, true, data)) 
ReturnWithError(EscapePressed);
						}
					}
				}
			} else {
				ASSERT(bpp == 8);
				const int shift = __max(0, UsedBitsPerChannel() - 8);

				for (i=0; i < h; i++) {
					for (j=0; j < w; j++) {
						buff[j] = Clamp8((y[yPos++] + yuvOffset31) >> 
shift);
					}
					buff += pitch;

					if (cb) {
						percent += dP;
						if ((*cb)(percent, true, data)) 
ReturnWithError(EscapePressed);
					}
				}
			}
			break;
		}
#endif
	case ImageModeRGB12:
		{
			ASSERT(m_header.channels == 3);
			ASSERT(m_header.bpp == m_header.channels*4);
			ASSERT(bpp == m_header.channels*4);
			ASSERT(!m_downsample);

			DataT* y = m_channel[0]; ASSERT(y);
			DataT* u = m_channel[1]; ASSERT(u);
			DataT* v = m_channel[2]; ASSERT(v);
			UINT16 yval;
			int cnt;

			for (i=0; i < h; i++) {
				cnt = 0;
				for (j=0; j < w; j++) {
					// Yuv
					uAvg = u[yPos];
					vAvg = v[yPos];
					yval = Clamp4(y[yPos++] + YUVoffset4 - ((uAvg + vAvg ) >> 
2)); // must be logical shift operator
					if (j%2 == 0) {
						buff[cnt] = UINT8(Clamp4(vAvg + yval) | (yval << 
4));
						cnt++;
						buff[cnt] = Clamp4(uAvg + yval);
					} else {
						buff[cnt] |= Clamp4(vAvg + yval) << 4;
						cnt++;
						buff[cnt] = UINT8(yval | (Clamp4(uAvg + yval) << 
4));
						cnt++;
					}
				}
				buff += pitch;
```

```
                                if (cb) {
                                        percent += dP;
                                        if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
                                }
                        }
                        break;
                }
        case ImageModeRGB16:
                {
                        ASSERT(m_header.channels == 3);
                        ASSERT(m_header.bpp == 16);
                        ASSERT(bpp == 16);
                        ASSERT(!m_downsample);

                        DataT* y = m_channel[0]; ASSERT(y);
                        DataT* u = m_channel[1]; ASSERT(u);
                        DataT* v = m_channel[2]; ASSERT(v);
                        UINT16 yval;
                        UINT16 *buff16 = (UINT16 *)buff;
                        int pitch16 = pitch/2;

                        for (i=0; i < h; i++) {
                                for (j=0; j < w; j++) {
                                        // Yuv
                                        uAvg = u[yPos];
                                        vAvg = v[yPos];
                                        yval = Clamp6(y[yPos++] + YUVoffset6 - ((uAvg + vAvg ) >>
2)); // must be logical shift operator
                                        buff16[j] = (yval << 5) | ((Clamp6(uAvg + yval) >> 1) <<
11) | (Clamp6(vAvg + yval) >> 1);
                                }
                                buff16 += pitch16;

                                if (cb) {
                                        percent += dP;
                                        if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
                                }
                        }
                        break;
                }
        default:
                ASSERT(false);
        }

#ifdef __PGFROISUPPORT__
        if (targetBuff) {
                // copy valid ROI (m_roi) from temporary buffer (roi) to target buffer
                if (bpp%8 == 0) {
                        BYTE bypp = bpp/8;
                        buff = buffStart + (levelRoi.top - roi.top)*pitch + (levelRoi.left -
roi.left)*bypp;
                        w = levelRoi.Width()*bypp;
                        h = levelRoi.Height();

                        for (i=0; i < h; i++) {
                                for (j=0; j < w; j++) {
                                        targetBuff[j] = buff[j];
                                }
                                targetBuff += targetPitch;
                                buff += pitch;
                        }
                } else {
                        // to do
                }

                delete[] buffStart;
        }
#endif
}
```

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 321 of file PGFimage.h.

```
{ 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 1269 of file PGFimage.cpp.

```

{
        if (iFirstColor + nColors > ColorTableLen)      ReturnWithError(ColorTableError);

        for (UINT32 i=iFirstColor, j=0; j < nColors; i++, j++) {
                prgbColors[j] = m_postHeader.clut[i];
        }
}
```

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

Returns:

Address of color table

Definition at line 334 of file PGFimage.h.

```
{ 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 607 of file PGFimage.cpp.

```
                                                {
        ASSERT(m_decoder);
        return m_decoder->GetEncodedHeaderLength();
}
```

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 370 of file PGFimage.h.

```
{ 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 339 of file PGFimage.h.

```
{ 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 345 of file PGFimage.h.

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

const UINT8 * CPGFImage::GetUserData (UINT32 &*size*) const

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

Parameters:

size	[out] Size of user data in bytes.

Returns:

A pointer to user data or NULL if there is no user data.

Definition at line 320 of file PGFimage.cpp.

```
                                                       {
        size = m_postHeader.userDataLen;
        return m_postHeader.userData;
}
```

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 350 of file PGFimage.h.

```
{ return m_userDataPos; }
```

void CPGFImage::GetYUV (int*pitch*, DataT **buff*, BYTE*bpp*, int*channelMap*[] = `NULL`, CallbackPtr*cb* = `NULL`, void **data* = `NULL`) 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 2455 of file PGFimage.cpp.

```
{
        ASSERT(buff);
        const UINT32 w = m_width[0];
        const UINT32 h = m_height[0];
        const bool wOdd = (1 == w%2);
        const int dataBits = DataTSize*8; ASSERT(dataBits == 16 || dataBits == 32);
        const int pitch2 = pitch/DataTSize;
        const int yuvOffset = (dataBits == 16) ? YUVoffset8 : YUVoffset16;
        const double dP = 1.0/h;

        int defMap[] = { 0, 1, 2, 3, 4, 5, 6, 7 }; ASSERT(sizeof(defMap)/sizeof(defMap[0]) ==
MaxChannels);
        if (channelMap == NULL) channelMap = defMap;
        int sampledPos = 0, yPos = 0;
        DataT uAvg, vAvg;
        double percent = 0;
        UINT32 i, j;

        if (m_header.channels == 3) {
                ASSERT(bpp%dataBits == 0);

                DataT* y = m_channel[0]; ASSERT(y);
                DataT* u = m_channel[1]; ASSERT(u);
                DataT* v = m_channel[2]; ASSERT(v);
                int cnt, channels = bpp/dataBits; ASSERT(channels >= m_header.channels);

                for (i=0; i < h; i++) {
                        if (i%2) sampledPos -= (w + 1)/2;
                        cnt = 0;
                        for (j=0; j < w; j++) {
                                if (m_downsample) {
                                        // image was downsampled
                                        uAvg = u[sampledPos];
                                        vAvg = v[sampledPos];
                                } else {
                                        uAvg = u[yPos];
                                        vAvg = v[yPos];
                                }
                                buff[cnt + channelMap[0]] = y[yPos];
                                buff[cnt + channelMap[1]] = uAvg;
                                buff[cnt + channelMap[2]] = vAvg;
                                yPos++;
                                cnt += channels;
                                if (j%2) sampledPos++;
                        }
                        buff += pitch2;
                        if (wOdd) sampledPos++;

                        if (cb) {
                                percent += dP;
                                if ((*cb)(percent, true, data)) ReturnWithError(EscapePressed);
                        }
                }
        } else if (m_header.channels == 4) {
                ASSERT(m_header.bpp == m_header.channels*8);
                ASSERT(bpp%dataBits == 0);

                DataT* y = m_channel[0]; ASSERT(y);
                DataT* u = m_channel[1]; ASSERT(u);
                DataT* v = m_channel[2]; ASSERT(v);
                DataT* a = m_channel[3]; ASSERT(a);
                UINT8 aAvg;
                int cnt, channels = bpp/dataBits; ASSERT(channels >= m_header.channels);

                for (i=0; i < h; i++) {
                        if (i%2) sampledPos -= (w + 1)/2;
                        cnt = 0;
                        for (j=0; j < w; j++) {
                                if (m_downsample) {
                                        // image was downsampled
                                        uAvg = u[sampledPos];
                                        vAvg = v[sampledPos];
                                        aAvg = Clamp8(a[sampledPos] + yuvOffset);
                                } else {
                                        uAvg = u[yPos];
                                        vAvg = v[yPos];
                                        aAvg = Clamp8(a[yPos] + yuvOffset);
                                }
                                // Yuv
                                buff[cnt + channelMap[0]] = y[yPos];
                                buff[cnt + channelMap[1]] = uAvg;
                                buff[cnt + channelMap[2]] = vAvg;
                                buff[cnt + channelMap[3]] = aAvg;
                                yPos++;
                                cnt += channels;
                                if (j%2) sampledPos++;
                        }
                        buff += pitch2;
                        if (wOdd) sampledPos++;

                        if (cb) {
                                percent += dP;
                                if ((*cb)(percent, true, data)) ReturnWithError(EscapePressed);
                        }
                }
        }
}
```

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 423 of file PGFimage.h.

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

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

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 737 of file PGFimage.cpp.

```
{
        ASSERT(buff);
        ASSERT(m_channel[0]);

        // color transform
        RgbToYuv(pitch, buff, bpp, channelMap, cb, data);

        if (m_downsample) {
                // Subsampling of the chrominance and alpha channels
                for (int i=1; i < m_header.channels; i++) {
                        Downsample(i);
                }
        }
}
```

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 1224 of file PGFimage.cpp.

```
                                   {
        size_t size = DataTSize;

        if (size >= 2) {
                switch(mode) {
                        case ImageModeBitmap:
                        case ImageModeIndexedColor:
                        case ImageModeGrayScale:
                        case ImageModeRGBColor:
                        case ImageModeCMYKColor:
                        case ImageModeHSLColor:
                        case ImageModeHSBColor:
                        //case ImageModeDuotone:
                        case ImageModeLabColor:
                        case ImageModeRGB12:
                        case ImageModeRGB16:
                        case ImageModeRGBA:
                                return true;
                }
        }
        if (size >= 3) {
                switch(mode) {
                        case ImageModeGray16:
                        case ImageModeRGB48:
                        case ImageModeLab48:
                        case ImageModeCMYK64:
                        //case ImageModeDuotone16:
                                return true;
                }
        }
        if (size >=4) {
                switch(mode) {
                        case ImageModeGray32:
                                return true;
                }
        }
        return false;
}
```

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

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 2566 of file PGFimage.cpp.

```
{
        ASSERT(buff);
        const double dP = 1.0/m_header.height;
        const int dataBits = DataTSize*8; ASSERT(dataBits == 16 || dataBits == 32);
        const int pitch2 = pitch/DataTSize;
        const int yuvOffset = (dataBits == 16) ? YUVoffset8 : YUVoffset16;

        int yPos = 0, cnt = 0;
        double percent = 0;
        int defMap[] = { 0, 1, 2, 3, 4, 5, 6, 7 }; ASSERT(sizeof(defMap)/sizeof(defMap[0]) ==
MaxChannels);

        if (channelMap == NULL) channelMap = defMap;

        if (m_header.channels == 3)     {
                ASSERT(bpp%dataBits == 0);

                DataT* y = m_channel[0]; ASSERT(y);
                DataT* u = m_channel[1]; ASSERT(u);
                DataT* v = m_channel[2]; ASSERT(v);
                const int channels = bpp/dataBits; ASSERT(channels >= m_header.channels);

                for (UINT32 h=0; h < m_header.height; h++) {
                        if (cb) {
                                if ((*cb)(percent, true, data)) ReturnWithError(EscapePressed);
                                percent += dP;
                        }

                        cnt = 0;
                        for (UINT32 w=0; w < m_header.width; w++) {
                                y[yPos] = buff[cnt + channelMap[0]];
                                u[yPos] = buff[cnt + channelMap[1]];
                                v[yPos] = buff[cnt + channelMap[2]];
                                yPos++;
                                cnt += channels;
                        }
                        buff += pitch2;
                }
        } else if (m_header.channels == 4) {
                ASSERT(bpp%dataBits == 0);

                DataT* y = m_channel[0]; ASSERT(y);
                DataT* u = m_channel[1]; ASSERT(u);
                DataT* v = m_channel[2]; ASSERT(v);
                DataT* a = m_channel[3]; ASSERT(a);
                const int channels = bpp/dataBits; ASSERT(channels >= m_header.channels);

                for (UINT32 h=0; h < m_header.height; h++) {
                        if (cb) {
                                if ((*cb)(percent, true, data)) ReturnWithError(EscapePressed);
                                percent += dP;
                        }

                        cnt = 0;
                        for (UINT32 w=0; w < m_header.width; w++) {
                                y[yPos] = buff[cnt + channelMap[0]];
                                u[yPos] = buff[cnt + channelMap[1]];
                                v[yPos] = buff[cnt + channelMap[2]];
                                a[yPos] = buff[cnt + channelMap[3]] - yuvOffset;
                                yPos++;
                                cnt += channels;
                        }
                        buff += pitch2;
                }
        }

        if (m_downsample) {
                // Subsampling of the chrominance and alpha channels
                for (int i=1; i < m_header.channels; i++) {
                        Downsample(i);
                }
        }
}
```

bool CPGFImage::IsOpen () const `[inline]`

Returns true if the PGF has been opened and not closed.

Definition at line 87 of file PGFimage.h.

```
{ return m_decoder != NULL; }
```

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 430 of file PGFimage.h.

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

static UINT32 CPGFImage::LevelHeight (UINT32*height*, int*level*) `[inline, static]`

Compute and return image height at given level.

Parameters:

height	Original image height (at level 0)
level	An image level

Returns:

Image level height in pixels

Definition at line 498 of file PGFimage.h.

```
{ ASSERT(level >= 0); UINT32 h = (height >> level); return ((h << level) == height) ? h : h + 1;
}
```

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

Return the number of image levels.

Returns:

Number of image levels

Definition at line 435 of file PGFimage.h.

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

static UINT32 CPGFImage::LevelWidth (UINT32*width*, int*level*) `[inline, static]`

Compute and return image width at given level.

Parameters:

width	Original image width (at level 0)

level	An image level

Returns:

Image level width in pixels

Definition at line 491 of file PGFimage.h.

```
{ ASSERT(level >= 0); UINT32 w = (width >> level); return ((w << level) == width) ? w : w + 1; }
```

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 454 of file PGFimage.h.

```
{ 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 130 of file PGFimage.cpp.

```
                                              {
        ASSERT(stream);

        // create decoder and read PGFPreHeader PGFHeader PGFPostHeader LevelLengths
        m_decoder = new CDecoder(stream, m_preHeader, m_header, m_postHeader, m_levelLength,
                m_userDataPos, m_useOMPinDecoder, m_skipUserData);

        if (m_header.nLevels > MaxLevel) ReturnWithError(FormatCannotRead);

        // set current level
        m_currentLevel = m_header.nLevels;

        // set image width and height
        m_width[0] = m_header.width;
        m_height[0] = m_header.height;

        // complete header
        CompleteHeader();

        // interpret quant parameter
        if (m_header.quality > DownsampleThreshold &&
                (m_header.mode == ImageModeRGBColor ||
                 m_header.mode == ImageModeRGBA ||
                 m_header.mode == ImageModeRGB48 ||
                 m_header.mode == ImageModeCMYKColor ||
                 m_header.mode == ImageModeCMYK64 ||
                 m_header.mode == ImageModeLabColor ||
                 m_header.mode == ImageModeLab48)) {
                m_downsample = true;
                m_quant = m_header.quality - 1;
        } else {
                m_downsample = false;
                m_quant = m_header.quality;
        }

        // set channel dimensions (chrominance is subsampled by factor 2)
        if (m_downsample) {
                for (int i=1; i < m_header.channels; i++) {
                        m_width[i] = (m_width[0] + 1)/2;
                        m_height[i] = (m_height[0] + 1)/2;
                }
        } else {
                for (int i=1; i < m_header.channels; i++) {
                        m_width[i] = m_width[0];
                        m_height[i] = m_height[0];
                }
        }

        if (m_header.nLevels > 0) {
                // init wavelet subbands
                for (int i=0; i < m_header.channels; i++) {
                        m_wtChannel[i] = new CWaveletTransform(m_width[i], m_height[i],
m_header.nLevels);
                }

                // used in Read when PM_Absolute
                m_percent = pow(0.25, m_header.nLevels);

        } else {
                // very small image: we don't use DWT and encoding

                // read channels
                for (int c=0; c < m_header.channels; c++) {
                        const UINT32 size = m_width[c]*m_height[c];
                        m_channel[c] = new(std::nothrow) DataT[size];
                        if (!m_channel[c]) ReturnWithError(InsufficientMemory);

                        // read channel data from stream
                        for (UINT32 i=0; i < size; i++) {
                                int count = DataTSize;
                                stream->Read(&count, &m_channel[c][i]);
                                if (count != DataTSize) ReturnWithError(MissingData);
                        }
                }
        }
}
```

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 441 of file PGFimage.h.

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

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

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 382 of file PGFimage.cpp.

```
                                                                {
        ASSERT((level >= 0 && level < m_header.nLevels) || m_header.nLevels == 0); //
m_header.nLevels == 0: image didn't use wavelet transform
        ASSERT(m_decoder);
```

```
#ifdef __PGFROISUPPORT__
        if (ROIisSupported() && m_header.nLevels > 0) {
                // new encoding scheme supporting ROI
                PGFRect rect(0, 0, m_header.width, m_header.height);
                Read(rect, level, cb, data);
                return;
        }
#endif

        if (m_header.nLevels == 0) {
                if (level == 0) {
                        // the data has already been read during open
                        // now update progress
                        if (cb) {
                                if ((*cb)(1.0, true, data)) ReturnWithError(EscapePressed);
                        }
                }
        } else {
                const int levelDiff = m_currentLevel - level;
                double percent = (m_progressMode == PM_Relative) ? pow(0.25, levelDiff) :
m_percent;

                // encoding scheme without ROI
                while (m_currentLevel > level) {
                        for (int i=0; i < m_header.channels; i++) {
                                ASSERT(m_wtChannel[i]);
                                // decode file and write stream to m_wtChannel
                                if (m_currentLevel == m_header.nLevels) {
                                        // last level also has LL band
                                        m_wtChannel[i]->GetSubband(m_currentLevel,
LL)->PlaceTile(*m_decoder, m_quant);
                                }
                                if (m_preHeader.version & Version5) {
                                        // since version 5
                                        m_wtChannel[i]->GetSubband(m_currentLevel,
HL)->PlaceTile(*m_decoder, m_quant);
                                        m_wtChannel[i]->GetSubband(m_currentLevel,
LH)->PlaceTile(*m_decoder, m_quant);
                                } else {
                                        // until version 4
                                        m_decoder->DecodeInterleaved(m_wtChannel[i],
m_currentLevel, m_quant);
                                }
                                m_wtChannel[i]->GetSubband(m_currentLevel,
HH)->PlaceTile(*m_decoder, m_quant);
                        }

                        volatile OSError error = NoError; // volatile prevents optimizations
                        #pragma omp parallel for default(shared)
                        for (int i=0; i < m_header.channels; i++) {
                                // inverse transform from m_wtChannel to m_channel
                                if (error == NoError) {
                                        OSError err =
m_wtChannel[i]->InverseTransform(m_currentLevel, &m_width[i], &m_height[i], &m_channel[i]);
                                        if (err != NoError) error = err;
                                }
                                ASSERT(m_channel[i]);
                        }
                        if (error != NoError) ReturnWithError(error);

                        // set new level: must be done before refresh callback
                        m_currentLevel--;

                        // now we have to refresh the display
                        if (m_cb) m_cb(m_cbArg);

                        // now update progress
                        if (cb) {
                                percent *= 4;
                                if (m_progressMode == PM_Absolute) m_percent = percent;
                                if ((*cb)(percent, true, data)) ReturnWithError(EscapePressed);
                        }
                }
        }

        // automatically closing
        if (m_currentLevel == 0) Close();
}
```

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

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). 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.

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). The rect might be cropped.
level	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 468 of file PGFimage.cpp.

```
                                                                                                    {
        ASSERT((level >= 0 && level < m_header.nLevels) || m_header.nLevels == 0); //
m_header.nLevels == 0: image didn't use wavelet transform
        ASSERT(m_decoder);

        if (m_header.nLevels == 0 || !ROIisSupported()) {
                rect.left = rect.top = 0;
                rect.right = m_header.width; rect.bottom = m_header.height;
                Read(level, cb, data);
        } else {
                ASSERT(ROIisSupported());
                // new encoding scheme supporting ROI
                ASSERT(rect.left < m_header.width && rect.top < m_header.height);

                const int levelDiff = m_currentLevel - level;
                double percent = (m_progressMode == PM_Relative) ? pow(0.25, levelDiff) :
m_percent;

                // check level difference
                if (levelDiff <= 0) {
                        // it is a new read call, probably with a new ROI
                        m_currentLevel = m_header.nLevels;
                        m_decoder->SetStreamPosToData();
                }

                // check rectangle
                if (rect.right == 0 || rect.right > m_header.width) rect.right = m_header.width;
                if (rect.bottom == 0 || rect.bottom > m_header.height) rect.bottom =
m_header.height;

                // enable ROI decoding and reading
                SetROI(rect);

                while (m_currentLevel > level) {
                        for (int i=0; i < m_header.channels; i++) {
                                ASSERT(m_wtChannel[i]);

                                // get number of tiles and tile indices
                                const UINT32 nTiles =
m_wtChannel[i]->GetNofTiles(m_currentLevel);
                                const PGFRect& tileIndices =
m_wtChannel[i]->GetTileIndices(m_currentLevel);

                                // decode file and write stream to m_wtChannel
                                if (m_currentLevel == m_header.nLevels) { // last level also has
LL band
                                        ASSERT(nTiles == 1);
                                        m_decoder->DecodeTileBuffer();
                                        m_wtChannel[i]->GetSubband(m_currentLevel,
LL)->PlaceTile(*m_decoder, m_quant);
                                }
                                for (UINT32 tileY=0; tileY < nTiles; tileY++) {
                                        for (UINT32 tileX=0; tileX < nTiles; tileX++) {
                                                // check relevance of tile
                                                if (tileIndices.IsInside(tileX, tileY)) {
                                                        m_decoder->DecodeTileBuffer();

m_wtChannel[i]->GetSubband(m_currentLevel, HL)->PlaceTile(*m_decoder, m_quant, true, tileX,
tileY);

m_wtChannel[i]->GetSubband(m_currentLevel, LH)->PlaceTile(*m_decoder, m_quant, true, tileX,
tileY);

m_wtChannel[i]->GetSubband(m_currentLevel, HH)->PlaceTile(*m_decoder, m_quant, true, tileX,
tileY);
                                                } else {
                                                        // skip tile
                                                        m_decoder->SkipTileBuffer();
                                                }
                                        }
                                }
                        }

                        volatile OSError error = NoError; // volatile prevents optimizations
                        #pragma omp parallel for default(shared)
                        for (int i=0; i < m_header.channels; i++) {
                                // inverse transform from m_wtChannel to m_channel
                                if (error == NoError) {
                                        OSError err =
m_wtChannel[i]->InverseTransform(m_currentLevel, &m_width[i], &m_height[i], &m_channel[i]);
                                        if (err != NoError) error = err;
                                }
                                ASSERT(m_channel[i]);
                        }
                        if (error != NoError) ReturnWithError(error);

                        // set new level: must be done before refresh callback
                        m_currentLevel--;

                        // now we have to refresh the display
                        if (m_cb) m_cb(m_cbArg);

                        // now update progress
                        if (cb) {
                                percent *= 4;
                                if (m_progressMode == PM_Absolute) m_percent = percent;
                                if ((*cb)(percent, true, data)) ReturnWithError(EscapePressed);
                        }
                }
        }

        // automatically closing
        if (m_currentLevel == 0) Close();
}
```

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 653 of file PGFimage.cpp.

```
                                                                                 {
        ASSERT(level >= 0 && level < m_header.nLevels);
        ASSERT(target);
        ASSERT(targetLen > 0);
        ASSERT(m_decoder);

        // reset stream position
        m_decoder->SetStreamPosToData();

        // position stream
        UINT64 offset = 0;

        for (int i=m_header.nLevels - 1; i > level; i--) {
                offset += m_levelLength[m_header.nLevels - 1 - i];
        }
        m_decoder->Skip(offset);

        // compute number of bytes to read
        UINT32 len = __min(targetLen, GetEncodedLevelLength(level));

        // read data
        len = m_decoder->ReadEncodedData(target, len);
        ASSERT(len >= 0 && len <= targetLen);

        return len;
}
```

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

Reads the encoded PGF headers 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 619 of file PGFimage.cpp.

```
                                                                        {
        ASSERT(target);
        ASSERT(targetLen > 0);
        ASSERT(m_decoder);

        // reset stream position
        m_decoder->SetStreamPosToStart();

        // compute number of bytes to read
        UINT32 len = __min(targetLen, GetEncodedHeaderLength());

        // read data
        len = m_decoder->ReadEncodedData(target, len);
        ASSERT(len >= 0 && len <= targetLen);
        return len;
}
```

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 121 of file PGFimage.h.

```
{ 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 330 of file PGFimage.cpp.

```
                                              {
        if (m_header.nLevels == 0) {
                // image didn't use wavelet transform
                if (level == 0) {
                        for (int i=0; i < m_header.channels; i++) {
                                ASSERT(m_wtChannel[i]);
                                m_channel[i] = m_wtChannel[i]->GetSubband(0, LL)->GetBuffer();
                        }
                }
        } else {
                int currentLevel = m_header.nLevels;

                if (ROIisSupported()) {
                        // enable ROI reading
                        SetROI(PGFRect(0, 0, m_header.width, m_header.height));
                }

                while (currentLevel > level) {
                        for (int i=0; i < m_header.channels; i++) {
                                ASSERT(m_wtChannel[i]);
                                // dequantize subbands
                                if (currentLevel == m_header.nLevels) {
                                        // last level also has LL band
                                        m_wtChannel[i]->GetSubband(currentLevel,
LL)->Dequantize(m_quant);
                                }
                                m_wtChannel[i]->GetSubband(currentLevel,
HL)->Dequantize(m_quant);
                                m_wtChannel[i]->GetSubband(currentLevel,
LH)->Dequantize(m_quant);
                                m_wtChannel[i]->GetSubband(currentLevel,
HH)->Dequantize(m_quant);

                                // inverse transform from m_wtChannel to m_channel
                                OSError err = m_wtChannel[i]->InverseTransform(currentLevel,
&m_width[i], &m_height[i], &m_channel[i]);
                                if (err != NoError) ReturnWithError(err);
                                ASSERT(m_channel[i]);
                        }

                        currentLevel--;
                }
        }
}
```

void CPGFImage::ResetStreamPos ()

Reset stream position to start of PGF pre-header

Definition at line 639 of file PGFimage.cpp.

```
                                      {
        ASSERT(m_decoder);
        return m_decoder->SetStreamPosToStart();
}
```

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

Definition at line 1308 of file PGFimage.cpp.

```
{
        ASSERT(buff);
        int yPos = 0, cnt = 0;
        double percent = 0;
        const double dP = 1.0/m_header.height;
        int defMap[] = { 0, 1, 2, 3, 4, 5, 6, 7 }; ASSERT(sizeof(defMap)/sizeof(defMap[0]) ==
MaxChannels);

        if (channelMap == NULL) channelMap = defMap;

        switch(m_header.mode) {
        case ImageModeBitmap:
                {
                        ASSERT(m_header.channels == 1);
                        ASSERT(m_header.bpp == 1);
                        ASSERT(bpp == 1);

                        const UINT32 w = m_header.width;
                        const UINT32 w2 = (m_header.width + 7)/8;
                        DataT* y = m_channel[0]; ASSERT(y);

                        for (UINT32 h=0; h < m_header.height; h++) {
                                if (cb) {
                                        if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
                                        percent += dP;
                                }

                                for (UINT32 j=0; j < w2; j++) {
                                        y[yPos++] = buff[j] - YUVoffset8;
                                }
                                for (UINT32 j=w2; j < w; j++) {
                                        y[yPos++] = YUVoffset8;
                                }

                                //UINT cnt = w;
                                //for (UINT32 j=0; j < w2; j++) {
                                //      for (int k=7; k >= 0; k--) {
                                //              if (cnt) {
                                //                      y[yPos++] = YUVoffset8 + (1 & (buff[j] >>
k));
                                //                      cnt--;
                                //              }
                                //      }
                                //}
                                buff += pitch;
                        }
                }
                break;
        case ImageModeIndexedColor:
        case ImageModeGrayScale:
        case ImageModeHSLColor:
        case ImageModeHSBColor:
        case ImageModeLabColor:
                {
                        ASSERT(m_header.channels >= 1);
                        ASSERT(m_header.bpp == m_header.channels*8);
                        ASSERT(bpp%8 == 0);
                        const int channels = bpp/8; ASSERT(channels >= m_header.channels);

                        for (UINT32 h=0; h < m_header.height; h++) {
                                if (cb) {
                                        if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
                                        percent += dP;
                                }

                                cnt = 0;
                                for (UINT32 w=0; w < m_header.width; w++) {
                                        for (int c=0; c < m_header.channels; c++) {
                                                m_channel[c][yPos] = buff[cnt + channelMap[c]] -
YUVoffset8;
                                        }
                                        cnt += channels;
                                        yPos++;
                                }
                                buff += pitch;
                        }
                }
                break;
        case ImageModeGray16:
        case ImageModeLab48:
                {
                        ASSERT(m_header.channels >= 1);
                        ASSERT(m_header.bpp == m_header.channels*16);
                        ASSERT(bpp%16 == 0);

                        UINT16 *buff16 = (UINT16 *)buff;
                        const int pitch16 = pitch/2;
                        const int channels = bpp/16; ASSERT(channels >= m_header.channels);
                        const int shift = 16 - UsedBitsPerChannel(); ASSERT(shift >= 0);
                        const DataT yuvOffset16 = 1 << (UsedBitsPerChannel() - 1);

                        for (UINT32 h=0; h < m_header.height; h++) {
                                if (cb) {
                                        if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
                                        percent += dP;
                                }

                                cnt = 0;
                                for (UINT32 w=0; w < m_header.width; w++) {
                                        for (int c=0; c < m_header.channels; c++) {
                                                m_channel[c][yPos] = (buff16[cnt +
channelMap[c]] >> shift) - yuvOffset16;
                                        }
                                        cnt += channels;
                                        yPos++;
                                }
                                buff16 += pitch16;
                        }
                }
                break;
        case ImageModeRGBColor:
                {
                        ASSERT(m_header.channels == 3);
                        ASSERT(m_header.bpp == m_header.channels*8);
                        ASSERT(bpp%8 == 0);

                        DataT* y = m_channel[0]; ASSERT(y);
                        DataT* u = m_channel[1]; ASSERT(u);
                        DataT* v = m_channel[2]; ASSERT(v);
                        const int channels = bpp/8; ASSERT(channels >= m_header.channels);
                        UINT8 b, g, r;

                        for (UINT32 h=0; h < m_header.height; h++) {
                                if (cb) {
                                        if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
                                        percent += dP;
                                }

                                cnt = 0;
                                for (UINT32 w=0; w < m_header.width; w++) {
                                        b = buff[cnt + channelMap[0]];
                                        g = buff[cnt + channelMap[1]];
                                        r = buff[cnt + channelMap[2]];
                                        // Yuv
                                        y[yPos] = ((b + (g << 1) + r) >> 2) - YUVoffset8;
                                        u[yPos] = r - g;
                                        v[yPos] = b - g;
                                        yPos++;
                                        cnt += channels;
                                }
                                buff += pitch;
                        }
                }
                break;
        case ImageModeRGB48:
                {
                        ASSERT(m_header.channels == 3);
                        ASSERT(m_header.bpp == m_header.channels*16);
                        ASSERT(bpp%16 == 0);

                        UINT16 *buff16 = (UINT16 *)buff;
                        const int pitch16 = pitch/2;
                        const int channels = bpp/16; ASSERT(channels >= m_header.channels);
                        const int shift = 16 - UsedBitsPerChannel(); ASSERT(shift >= 0);
                        const DataT yuvOffset16 = 1 << (UsedBitsPerChannel() - 1);

                        DataT* y = m_channel[0]; ASSERT(y);
                        DataT* u = m_channel[1]; ASSERT(u);
                        DataT* v = m_channel[2]; ASSERT(v);
                        UINT16 b, g, r;

                        for (UINT32 h=0; h < m_header.height; h++) {
                                if (cb) {
                                        if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
                                        percent += dP;
                                }

                                cnt = 0;
                                for (UINT32 w=0; w < m_header.width; w++) {
                                        b = buff16[cnt + channelMap[0]] >> shift;
                                        g = buff16[cnt + channelMap[1]] >> shift;
                                        r = buff16[cnt + channelMap[2]] >> shift;
                                        // Yuv
                                        y[yPos] = ((b + (g << 1) + r) >> 2) - yuvOffset16;
                                        u[yPos] = r - g;
                                        v[yPos] = b - g;
                                        yPos++;
                                        cnt += channels;
                                }
                                buff16 += pitch16;
                        }
                }
                break;
        case ImageModeRGBA:
        case ImageModeCMYKColor:
                {
                        ASSERT(m_header.channels == 4);
                        ASSERT(m_header.bpp == m_header.channels*8);
                        ASSERT(bpp%8 == 0);
                        const int channels = bpp/8; ASSERT(channels >= m_header.channels);

                        DataT* y = m_channel[0]; ASSERT(y);
                        DataT* u = m_channel[1]; ASSERT(u);
                        DataT* v = m_channel[2]; ASSERT(v);
                        DataT* a = m_channel[3]; ASSERT(a);
                        UINT8 b, g, r;

                        for (UINT32 h=0; h < m_header.height; h++) {
                                if (cb) {
                                        if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
                                        percent += dP;
                                }

                                cnt = 0;
                                for (UINT32 w=0; w < m_header.width; w++) {
                                        b = buff[cnt + channelMap[0]];
                                        g = buff[cnt + channelMap[1]];
                                        r = buff[cnt + channelMap[2]];
                                        // Yuv
                                        y[yPos] = ((b + (g << 1) + r) >> 2) - YUVoffset8;
                                        u[yPos] = r - g;
                                        v[yPos] = b - g;
                                        a[yPos++] = buff[cnt + channelMap[3]] - YUVoffset8;
                                        cnt += channels;
                                }
                                buff += pitch;
                        }
                }
                break;
        case ImageModeCMYK64:
                {
                        ASSERT(m_header.channels == 4);
                        ASSERT(m_header.bpp == m_header.channels*16);
                        ASSERT(bpp%16 == 0);

                        UINT16 *buff16 = (UINT16 *)buff;
                        const int pitch16 = pitch/2;
                        const int channels = bpp/16; ASSERT(channels >= m_header.channels);
                        const int shift = 16 - UsedBitsPerChannel(); ASSERT(shift >= 0);
                        const DataT yuvOffset16 = 1 << (UsedBitsPerChannel() - 1);

                        DataT* y = m_channel[0]; ASSERT(y);
                        DataT* u = m_channel[1]; ASSERT(u);
                        DataT* v = m_channel[2]; ASSERT(v);
                        DataT* a = m_channel[3]; ASSERT(a);
                        UINT16 b, g, r;

                        for (UINT32 h=0; h < m_header.height; h++) {
                                if (cb) {
                                        if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
                                        percent += dP;
                                }

                                cnt = 0;
                                for (UINT32 w=0; w < m_header.width; w++) {
                                        b = buff16[cnt + channelMap[0]] >> shift;
                                        g = buff16[cnt + channelMap[1]] >> shift;
                                        r = buff16[cnt + channelMap[2]] >> shift;
                                        // Yuv
                                        y[yPos] = ((b + (g << 1) + r) >> 2) - yuvOffset16;
                                        u[yPos] = r - g;
                                        v[yPos] = b - g;
                                        a[yPos++] = (buff16[cnt + channelMap[3]] >> shift) -
yuvOffset16;
                                        cnt += channels;
                                }
                                buff16 += pitch16;
                        }
                }
                break;
#ifdef __PGF32SUPPORT__
        case ImageModeGray32:
                {
                        ASSERT(m_header.channels == 1);
                        ASSERT(m_header.bpp == 32);
                        ASSERT(bpp == 32);
                        ASSERT(DataTSize == sizeof(UINT32));

                        DataT* y = m_channel[0]; ASSERT(y);

                        UINT32 *buff32 = (UINT32 *)buff;
                        const int pitch32 = pitch/4;
                        const int shift = 31 - UsedBitsPerChannel(); ASSERT(shift >= 0);
                        const DataT yuvOffset31 = 1 << (UsedBitsPerChannel() - 1);

                        for (UINT32 h=0; h < m_header.height; h++) {
                                if (cb) {
                                        if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
                                        percent += dP;
                                }

                                for (UINT32 w=0; w < m_header.width; w++) {
                                        y[yPos++] = (buff32[w] >> shift) - yuvOffset31;
                                }
                                buff32 += pitch32;
                        }
                }
                break;
#endif
        case ImageModeRGB12:
                {
                        ASSERT(m_header.channels == 3);
                        ASSERT(m_header.bpp == m_header.channels*4);
                        ASSERT(bpp == m_header.channels*4);

                        DataT* y = m_channel[0]; ASSERT(y);
                        DataT* u = m_channel[1]; ASSERT(u);
                        DataT* v = m_channel[2]; ASSERT(v);

                        UINT8 rgb = 0, b, g, r;

                        for (UINT32 h=0; h < m_header.height; h++) {
                                if (cb) {
                                        if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
                                        percent += dP;
                                }

                                cnt = 0;
                                for (UINT32 w=0; w < m_header.width; w++) {
                                        if (w%2 == 0) {
                                                // even pixel position
                                                rgb = buff[cnt];
                                                b = rgb & 0x0F;
                                                g = (rgb & 0xF0) >> 4;
                                                cnt++;
                                                rgb = buff[cnt];
                                                r = rgb & 0x0F;
                                        } else {
                                                // odd pixel position
                                                b = (rgb & 0xF0) >> 4;
                                                cnt++;
                                                rgb = buff[cnt];
                                                g = rgb & 0x0F;
                                                r = (rgb & 0xF0) >> 4;
                                                cnt++;
                                        }

                                        // Yuv
                                        y[yPos] = ((b + (g << 1) + r) >> 2) - YUVoffset4;
                                        u[yPos] = r - g;
                                        v[yPos] = b - g;
                                        yPos++;
                                }
                                buff += pitch;
                        }
                }
                break;
        case ImageModeRGB16:
                {
                        ASSERT(m_header.channels == 3);
                        ASSERT(m_header.bpp == 16);
                        ASSERT(bpp == 16);

                        DataT* y = m_channel[0]; ASSERT(y);
                        DataT* u = m_channel[1]; ASSERT(u);
                        DataT* v = m_channel[2]; ASSERT(v);

                        UINT16 *buff16 = (UINT16 *)buff;
                        UINT16 rgb, b, g, r;
                        const int pitch16 = pitch/2;

                        for (UINT32 h=0; h < m_header.height; h++) {
                                if (cb) {
                                        if ((*cb)(percent, true, data))
ReturnWithError(EscapePressed);
                                        percent += dP;
                                }
                                for (UINT32 w=0; w < m_header.width; w++) {
                                        rgb = buff16[w];
                                        r = (rgb & 0xF800) >> 10;       // highest 5 bits
                                        g = (rgb & 0x07E0) >> 5;        // middle 6 bits
                                        b = (rgb & 0x001F) << 1;        // lowest 5 bits
                                        // Yuv
                                        y[yPos] = ((b + (g << 1) + r) >> 2) - YUVoffset6;
                                        u[yPos] = r - g;
                                        v[yPos] = b - g;
                                        yPos++;
                                }

                                buff16 += pitch16;
                        }
                }
                break;
        default:
                ASSERT(false);
        }
}
```

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 465 of file PGFimage.h.

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

void CPGFImage::SetChannel (DataT **channel*, intc = 0) `[inline]`

Set internal PGF image buffer channel.

Parameters:

channel	A YUV data channel
c	A channel index

Definition at line 276 of file PGFimage.h.

```
{ 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 1283 of file PGFimage.cpp.

```
{
        if (iFirstColor + nColors > ColorTableLen)      ReturnWithError(ColorTableError);

        for (UINT32 i=iFirstColor, j=0; j < nColors; i++, j++) {
                m_postHeader.clut[i] = prgbColors[j];
        }
}
```

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

Set PGF header and user data. Precondition: The PGF image has been closed with Close(...) or 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 838 of file PGFimage.cpp.

```
{
        ASSERT(!m_decoder);     // current image must be closed
        ASSERT(header.quality <= MaxQuality);

        // init state
#ifdef __PGFROISUPPORT__
        m_streamReinitialized = false;
#endif

        // init preHeader
        memcpy(m_preHeader.magic, Magic, 3);
        m_preHeader.version = PGFVersion | flags;
        m_preHeader.hSize = HeaderSize;

        // copy header
        memcpy(&m_header, &header, HeaderSize);

        // complete header
        CompleteHeader();

        // check and set number of levels
        ComputeLevels();

        // check for downsample
        if (m_header.quality > DownsampleThreshold &&  (m_header.mode == ImageModeRGBColor ||

m_header.mode == ImageModeRGBA ||

m_header.mode == ImageModeRGB48 ||

m_header.mode == ImageModeCMYKColor ||

m_header.mode == ImageModeCMYK64 ||

m_header.mode == ImageModeLabColor ||

m_header.mode == ImageModeLab48)) {
                m_downsample = true;
                m_quant = m_header.quality - 1;
        } else {
                m_downsample = false;
                m_quant = m_header.quality;
        }

        // update header size and copy user data
        if (m_header.mode == ImageModeIndexedColor) {
                // update header size
                m_preHeader.hSize += ColorTableSize;
        }
        if (userDataLength && userData) {
                m_postHeader.userData = new(std::nothrow) UINT8[userDataLength];
                if (!m_postHeader.userData) ReturnWithError(InsufficientMemory);
                m_postHeader.userDataLen = userDataLength;
                memcpy(m_postHeader.userData, userData, userDataLength);
                // update header size
                m_preHeader.hSize += userDataLength;
        }

        // allocate channels
        for (int i=0; i < m_header.channels; i++) {
                // set current width and height
                m_width[i] = m_header.width;
                m_height[i] = m_header.height;

                // allocate channels
                ASSERT(!m_channel[i]);
                m_channel[i] = new(std::nothrow) DataT[m_header.width*m_header.height];
                if (!m_channel[i]) {
                        if (i) i--;
                        while(i) {
                                delete[] m_channel[i]; m_channel[i] = 0;
                                i--;
                        }
                        ReturnWithError(InsufficientMemory);
                }
        }
}
```

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 684 of file PGFimage.cpp.

```
                                   {
        const BYTE bpc = m_header.bpp/m_header.channels;
        BYTE pot = 0;

        while(maxValue > 0) {
                pot++;
                maxValue >>= 1;
        }
        // store bits per channel
        if (pot > bpc) pot = bpc;
        if (pot > 31) pot = 31;
        m_header.usedBitsPerChannel = pot;
}
```

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 300 of file PGFimage.h.

```
{ 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 307 of file PGFimage.h.

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

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

Compute ROIs for each channel and each level

Parameters:

rect	rectangular region of interest (ROI)

Definition at line 562 of file PGFimage.cpp.

```
                                   {
        ASSERT(m_decoder);
        ASSERT(ROIisSupported());

        // store ROI for a later call of GetBitmap
        m_roi = rect;

        // enable ROI decoding
        m_decoder->SetROI();

        // enlarge ROI because of border artefacts
        const UINT32 dx = FilterWidth/2*(1 << m_currentLevel);
        const UINT32 dy = FilterHeight/2*(1 << m_currentLevel);

        if (rect.left < dx) rect.left = 0;
        else rect.left -= dx;
        if (rect.top < dy) rect.top = 0;
        else rect.top -= dy;
        rect.right += dx;
        if (rect.right > m_header.width) rect.right = m_header.width;
        rect.bottom += dy;
        if (rect.bottom > m_header.height) rect.bottom = m_header.height;

        // prepare wavelet channels for using ROI
        ASSERT(m_wtChannel[0]);
        m_wtChannel[0]->SetROI(rect);
        if (m_downsample && m_header.channels > 1) {
                // all further channels are downsampled, therefore downsample ROI
                rect.left >>= 1;
                rect.top >>= 1;
                rect.right >>= 1;
                rect.bottom >>= 1;
        }
        for (int i=1; i < m_header.channels; i++) {
                ASSERT(m_wtChannel[i]);
                m_wtChannel[i]->SetROI(rect);
        }
}
```

UINT32 CPGFImage::UpdatePostHeaderSize () `[private]`

Definition at line 1044 of file PGFimage.cpp.

```
                                              {
        ASSERT(m_encoder);

        INT64 offset = m_encoder->ComputeOffset(); ASSERT(offset >= 0);

        if (offset > 0) {
                // update post-header size and rewrite pre-header
                m_preHeader.hSize += (UINT32)offset;
                m_encoder->UpdatePostHeaderSize(m_preHeader);
        }

        // write dummy levelLength into stream
        return m_encoder->WriteLevelLength(m_levelLength);
}
```

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 702 of file PGFimage.cpp.

```
                                         {
        const BYTE bpc = m_header.bpp/m_header.channels;

        if (bpc > 8) {
                return m_header.usedBitsPerChannel;
        } else {
                return bpc;
        }
}
```

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

Returns images' PGF version

Returns:

PGF codec version of the image

Definition at line 476 of file PGFimage.h.

```
{ return CurrentVersion(m_preHeader.version); }
```

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 416 of file PGFimage.h.

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

void CPGFImage::Write (CPGFStream *stream, UINT32 *nWrittenBytes = NULL, CallbackPtrcb = NULL, void *data = NULL)

Encode and write a 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 1140 of file PGFimage.cpp.

```
{
        ASSERT(stream);
        ASSERT(m_preHeader.hSize);

        // create wavelet transform channels and encoder
        UINT32 nBytes = WriteHeader(stream);

        // write image
        nBytes += WriteImage(stream, cb, data);

        // return written bytes
        if (nWrittenBytes) *nWrittenBytes += nBytes;
}
```

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

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.

Definition at line 1169 of file PGFimage.cpp.

```
                                                                        {
        ASSERT(m_header.nLevels > 0);
        ASSERT(0 <= level && level < m_header.nLevels);
        ASSERT(m_encoder);
        ASSERT(ROIisSupported());

        const int levelDiff = m_currentLevel - level;
        double percent = (m_progressMode == PM_Relative) ? pow(0.25, levelDiff) : m_percent;
        UINT32 nWrittenBytes = 0;

        if (m_currentLevel == m_header.nLevels) {
                // update post-header size, rewrite pre-header, and write dummy levelLength
                nWrittenBytes = UpdatePostHeaderSize();
        } else {
                // prepare for next level: save current file position, because the stream might
have been reinitialized
                if (m_encoder->ComputeBufferLength()) {
                        m_streamReinitialized = true;
                }
        }

        // encoding scheme with ROI
        while (m_currentLevel > level) {
                WriteLevel();   // decrements m_currentLevel

                if (m_levelLength) {
                        nWrittenBytes += m_levelLength[m_header.nLevels - m_currentLevel - 1];
                }
                // now update progress
                if (cb) {
                        percent *= 4;
                        if (m_progressMode == PM_Absolute) m_percent = percent;
                        if ((*cb)(percent, true, data)) ReturnWithError(EscapePressed);
                }
        }

        // automatically closing
        if (m_currentLevel == 0) {
                if (!m_streamReinitialized) {
                        // don't write level lengths, if the stream position changed inbetween two 
Write operations
                        m_encoder->UpdateLevelLength();
                }
                // delete encoder
                delete m_encoder; m_encoder = NULL;
        }

        return nWrittenBytes;
}
```

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.

Definition at line 917 of file PGFimage.cpp.

```
                                                       {
        ASSERT(m_header.nLevels <= MaxLevel);
        ASSERT(m_header.quality <= MaxQuality); // quality is already initialized

        if (m_header.nLevels > 0) {
                volatile OSError error = NoError; // volatile prevents optimizations
                // create new wt channels
                #pragma omp parallel for default(shared)
                for (int i=0; i < m_header.channels; i++) {
                        DataT *temp = NULL;
                        if (error == NoError) {
                                if (m_wtChannel[i]) {
                                        ASSERT(m_channel[i]);
                                        // copy m_channel to temp
                                        int size = m_height[i]*m_width[i];
                                        temp = new(std::nothrow) DataT[size];
                                        if (temp) {
                                                memcpy(temp, m_channel[i], size*DataTSize);
                                                delete m_wtChannel[i];  // also deletes 
m_channel
                                        } else {
                                                error = InsufficientMemory;
                                        }
                                }
                                if (error == NoError) {
                                        if (temp) m_channel[i] = temp;
                                        m_wtChannel[i] = new CWaveletTransform(m_width[i], 
m_height[i], m_header.nLevels, m_channel[i]);
                                #ifdef __PGFROISUPPORT__
                                        m_wtChannel[i]->SetROI(PGFRect(0, 0, m_header.width, 
m_header.height));
                                #endif

                                        // wavelet subband decomposition
                                        for (int l=0; error == NoError && l < m_header.nLevels;
l++) {
                                                OSError err =
m_wtChannel[i]->ForwardTransform(l, m_quant);
                                                if (err != NoError) error = err;
                                        }
                                }
                        }
                }
                if (error != NoError) ReturnWithError(error);

                m_currentLevel = m_header.nLevels;

                // create encoder and eventually write headers and levelLength
                m_encoder = new CEncoder(stream, m_preHeader, m_header, m_postHeader,
m_userDataPos, m_useOMPinEncoder);
                if (m_favorSpeedOverSize) m_encoder->FavorSpeedOverSize();

        #ifdef __PGFROISUPPORT__
                if (ROIisSupported()) {
                        // new encoding scheme supporting ROI
                        m_encoder->SetROI();
                }
        #endif

        } else {
                // very small image: we don't use DWT and encoding

                // create encoder and eventually write headers and levelLength
                m_encoder = new CEncoder(stream, m_preHeader, m_header, m_postHeader,
m_userDataPos, m_useOMPinEncoder);
        }

        INT64 nBytes = m_encoder->ComputeHeaderLength();
        return (nBytes > 0) ? (UINT32)nBytes : 0;
}
```

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

Encode and write the one and only 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 1069 of file PGFimage.cpp.

```
                                                                                      {
        ASSERT(stream);
        ASSERT(m_preHeader.hSize);

        int levels = m_header.nLevels;
        double percent = pow(0.25, levels);

        // update post-header size, rewrite pre-header, and write dummy levelLength
        UINT32 nWrittenBytes = UpdatePostHeaderSize();

        if (levels == 0) {
                // write channels
                for (int c=0; c < m_header.channels; c++) {
                        const UINT32 size = m_width[c]*m_height[c];

                        // write channel data into stream
                        for (UINT32 i=0; i < size; i++) {
                                int count = DataTSize;
                                stream->Write(&count, &m_channel[c][i]);
                        }
                }

                // now update progress
                if (cb) {
                        if ((*cb)(1, true, data)) ReturnWithError(EscapePressed);
                }

        } else {
                // encode quantized wavelet coefficients and write to PGF file
                // encode subbands, higher levels first
                // color channels are interleaved

                // encode all levels
                for (m_currentLevel = levels; m_currentLevel > 0; ) {
                        WriteLevel(); // decrements m_currentLevel

                        // now update progress
                        if (cb) {
                                percent *= 4;
                                if ((*cb)(percent, true, data)) ReturnWithError(EscapePressed);
                        }
                }

                // flush encoder and write level lengths
                m_encoder->Flush();
        }

        // update level lengths
        nWrittenBytes = m_encoder->UpdateLevelLength(); // return written image bytes

        // delete encoder
        delete m_encoder; m_encoder = NULL;

        ASSERT(!m_encoder);

        return nWrittenBytes;
}
```

void CPGFImage::WriteLevel () `[private]`

Definition at line 989 of file PGFimage.cpp.

```
                                  {
        ASSERT(m_encoder);
        ASSERT(m_currentLevel > 0);
        ASSERT(m_header.nLevels > 0);

#ifdef __PGFROISUPPORT__
        if (ROIisSupported()) {
                const int lastChannel = m_header.channels - 1;

                for (int i=0; i < m_header.channels; i++) {
                        // get number of tiles and tile indices
                        const UINT32 nTiles = m_wtChannel[i]->GetNofTiles(m_currentLevel);
                        const UINT32 lastTile = nTiles - 1;

                        if (m_currentLevel == m_header.nLevels) {
                                // last level also has LL band
                                ASSERT(nTiles == 1);
                                m_wtChannel[i]->GetSubband(m_currentLevel,
LL)->ExtractTile(*m_encoder);
                                m_encoder->EncodeTileBuffer();
                        }
                        for (UINT32 tileY=0; tileY < nTiles; tileY++) {
                                for (UINT32 tileX=0; tileX < nTiles; tileX++) {
                                        m_wtChannel[i]->GetSubband(m_currentLevel,
HL)->ExtractTile(*m_encoder, true, tileX, tileY);
```

```
                                        m_wtChannel[i]->GetSubband(m_currentLevel,
LH)->ExtractTile(*m_encoder, true, tileX, tileY);
                                        m_wtChannel[i]->GetSubband(m_currentLevel,
HH)->ExtractTile(*m_encoder, true, tileX, tileY);
                                        if (i == lastChannel && tileY == lastTile && tileX ==
lastTile) {
                                                // all necessary data are buffered. next call of
EncodeBuffer will write the last piece of data of the current level.
                                                m_encoder->SetEncodedLevel(--m_currentLevel);
                                        }
                                        m_encoder->EncodeTileBuffer();
                                }
                        }
                }
        } else
#endif
        {
                for (int i=0; i < m_header.channels; i++) {
                        ASSERT(m_wtChannel[i]);
                        if (m_currentLevel == m_header.nLevels) {
                                // last level also has LL band
                                m_wtChannel[i]->GetSubband(m_currentLevel,
LL)->ExtractTile(*m_encoder);
                        }
                        //encoder.EncodeInterleaved(m_wtChannel[i], m_currentLevel, m_quant);
// until version 4
                        m_wtChannel[i]->GetSubband(m_currentLevel,
HL)->ExtractTile(*m_encoder); // since version 5
                        m_wtChannel[i]->GetSubband(m_currentLevel,
LH)->ExtractTile(*m_encoder); // since version 5
                        m_wtChannel[i]->GetSubband(m_currentLevel,
HH)->ExtractTile(*m_encoder);
                }

                // all necessary data are buffered. next call of EncodeBuffer will write the last
piece of data of the current level.
                m_encoder->SetEncodedLevel(--m_currentLevel);
        }
}
```

Member Data Documentation

RefreshCB CPGFImage::m_cb [private]

pointer to refresh callback procedure

Definition at line 535 of file PGFimage.h.

void* CPGFImage::m_cbArg [private]

refresh callback argument

Definition at line 536 of file PGFimage.h.

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

untransformed channels in YUV format

Definition at line 512 of file PGFimage.h.

int CPGFImage::m_currentLevel [protected]

transform level of current image

Definition at line 522 of file PGFimage.h.

CDecoder* CPGFImage::m_decoder `[protected]`

PGF decoder.

Definition at line 513 of file PGFimage.h.

bool CPGFImage::m_downsample `[protected]`

chrominance channels are downsampled

Definition at line 524 of file PGFimage.h.

CEncoder* CPGFImage::m_encoder `[protected]`

PGF encoder.

Definition at line 514 of file PGFimage.h.

bool CPGFImage::m_favorSpeedOverSize `[protected]`

favor encoding speed over compression ratio

Definition at line 525 of file PGFimage.h.

PGFHeader CPGFImage::m_header `[protected]`

PGF file header.

Definition at line 519 of file PGFimage.h.

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

height of each channel at current level

Definition at line 517 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 515 of file PGFimage.h.

double CPGFImage::m_percent `[private]`

progress [0..1]

Definition at line 537 of file PGFimage.h.

PGFPostHeader CPGFImage::m_postHeader `[protected]`

PGF post-header.

Definition at line 520 of file PGFimage.h.

PGFPreHeader CPGFImage::m_preHeader `[protected]`

PGF pre-header.

Definition at line 518 of file PGFimage.h.

ProgressMode CPGFImage::m_progressMode `[private]`

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

Definition at line 538 of file PGFimage.h.

BYTE CPGFImage::m_quant `[protected]`

quantization parameter

Definition at line 523 of file PGFimage.h.

PGFRect CPGFImage::m_roi `[protected]`

region of interest

Definition at line 531 of file PGFimage.h.

bool CPGFImage::m_skipUserData `[protected]`

skip user data (metadata) during open

Definition at line 528 of file PGFimage.h.

bool CPGFImage::m_streamReinitialized `[protected]`

stream has been reinitialized

Definition at line 530 of file PGFimage.h.

bool CPGFImage::m_useOMPinDecoder `[protected]`

use Open MP in decoder

Definition at line 527 of file PGFimage.h.

bool CPGFImage::m_useOMPinEncoder `[protected]`

use Open MP in encoder

Definition at line 526 of file PGFimage.h.

UINT64 CPGFImage::m_userDataPos `[protected]`

stream position of user data

Definition at line 521 of file PGFimage.h.

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

width of each channel at current level

Definition at line 516 of file PGFimage.h.

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

wavelet transformed color channels

Definition at line 511 of file PGFimage.h.

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

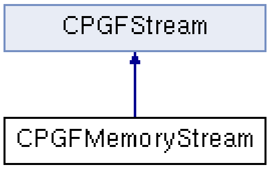
- **PGFimage.h**
- **PGFimage.cpp**

CPGFMemoryStream Class Reference

Memory stream class.

`#include <PGFstream.h>`

Inheritance diagram for CPGFMemoryStream:



Public Member Functions

- **CPGFMemoryStream** (size_t size) THROW_
- **CPGFMemoryStream** (UINT8 *pBuffer, size_t size) THROW_
- void **Reinitialize** (UINT8 *pBuffer, size_t size) THROW_
- virtual **~CPGFMemoryStream** ()
- virtual void **Write** (int *count, void *buffer) THROW_
- virtual void **Read** (int *count, void *buffer)
- virtual void **SetPos** (short posMode, INT64 posOff) THROW_
- 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.

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

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.

```
: m_buffer(pBuffer)
, m_pos(pBuffer)
, m_eos(pBuffer + size)
, m_size(size)
, m_allocated(false) {
        ASSERT(IsValid());
}
```

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

Definition at line 126 of file PGFstream.h.

```
                                    {
                m_pos = 0;
                if (m_allocated) {
                        // the memory buffer has been allocated inside of CPMFmemoryStream
constructor
                        delete[] m_buffer; m_buffer = 0;
                }
        }
```

Member Function Documentation

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

Returns:

Memory buffer

Definition at line 143 of file PGFstream.h.

```
{ return m_buffer; }
```

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

Returns:

Memory buffer

Definition at line 145 of file PGFstream.h.

```
{ return m_buffer; }
```

UINT64 CPGFMemoryStream::GetEOS () const `[inline]`

Returns:

relative position of end of stream (= stream length)

Definition at line 147 of file PGFstream.h.

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

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

Get current stream position.

Returns:

Current stream position

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

Definition at line 137 of file PGFstream.h.

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

size_t CPGFMemoryStream::GetSize () const `[inline]`

Returns:

Memory size

Definition at line 141 of file PGFstream.h.

```
{ 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.110*).

Definition at line 138 of file PGFstream.h.

```
{ 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.110*).

Definition at line 148 of file PGFstream.cpp.

```
                                                          {
        ASSERT(IsValid());
        ASSERT(count);
        ASSERT(buffPtr);
        ASSERT(m_buffer + m_size >= m_eos);
        ASSERT(m_pos <= m_eos);

        if (m_pos + *count <= m_eos) {
                memcpy(buffPtr, m_pos, *count);
                m_pos += *count;
        } else {
                // end of memory block reached -> read only until end
                *count = (int)__max(0, m_eos - m_pos);
                memcpy(buffPtr, m_pos, *count);
                m_pos += *count;
        }
        ASSERT(m_pos <= m_eos);
}
```

void CPGFMemoryStream::Reinitialize (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 102 of file PGFstream.cpp.

```
                                                                           {
        if (!m_allocated) {
                m_buffer = m_pos = pBuffer;
                m_size = size;
                m_eos = m_buffer + size;
        }
}
```

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

Parameters:

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

Definition at line 149 of file PGFstream.h.

```
{ 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.110*).

Definition at line 168 of file PGFstream.cpp.

```
                                                                {
        ASSERT(IsValid());
        switch(posMode) {
        case FSFromStart:
                m_pos = m_buffer + posOff;
                break;
        case FSFromCurrent:
                m_pos += posOff;
                break;
        case FSFromEnd:
                m_pos = m_eos + posOff;
                break;
        default:
                ASSERT(false);
        }
        if (m_pos > m_eos)
                ReturnWithError(InvalidStreamPos);
}
```

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.110*).

Definition at line 111 of file PGFstream.cpp.

```
                                                             {
        ASSERT(count);
        ASSERT(buffPtr);
        ASSERT(IsValid());
        const size_t deltaSize = 0x4000 + *count;

        if (m_pos + *count <= m_buffer + m_size) {
                memcpy(m_pos, buffPtr, *count);
                m_pos += *count;
                if (m_pos > m_eos) m_eos = m_pos;
        } else if (m_allocated) {
                // memory block is too small -> reallocate a deltaSize larger block
                size_t offset = m_pos - m_buffer;
                UINT8 *buf_tmp = (UINT8 *)realloc(m_buffer, m_size + deltaSize);
                if (!buf_tmp) {
                        delete[] m_buffer;
                        m_buffer = 0;
                        ReturnWithError(InsufficientMemory);
                } else {
                        m_buffer = buf_tmp;
                }
                m_size += deltaSize;

                // reposition m_pos
                m_pos = m_buffer + offset;

                // write block
                memcpy(m_pos, buffPtr, *count);
                m_pos += *count;
                if (m_pos > m_eos) m_eos = m_pos;
        } else {
                ReturnWithError(InsufficientMemory);
        }
        ASSERT(m_pos <= m_eos);
}
```

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:

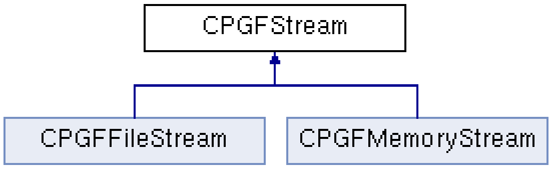
- **PGFstream.h**
- **PGFstream.cpp**

CPGFStream Class Reference

Abstract stream base class.

`#include <PGFstream.h>`

Inheritance diagram for CPGFStream:



Public Member Functions

- **CPGFStream** ()
- virtual **~CPGFStream** ()
- 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.

```
{}
```

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

Standard destructor.

Definition at line 47 of file PGFstream.h.

```
{}
```

Member Function Documentation

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

Get current stream position.

Returns:

Current stream position

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

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

Check stream validity.

Returns:

True if stream and current position is valid

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

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

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

Parameters:

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

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

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

Set stream position either absolute or relative.

Parameters:

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

Implemented in **CPGFFileStream** (*p.48*), and **CPGFMemoryStream** (*p.106*).

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

Write some bytes out of a buffer into this stream.

Parameters:

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

Implemented in **CPGFFileStream** (*p.48*), and **CPGFMemoryStream** (*p.107*).

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

- **PGFstream.h**

CRoiIndices Class Reference

ROI indices.

`#include <WaveletTransform.h>`

Public Member Functions

- UINT32 **GetNofTiles** (int level) const

Private Member Functions

- **CRoiIndices** ()
- **~CRoiIndices** ()
- void **Destroy** ()
- void **CreateIndices** ()
- void **ComputeIndices** (UINT32 width, UINT32 height, const **PGFRect** &rect)
- const **PGFRect** & **GetIndices** (int level) const
- void **SetLevels** (int levels)
- void **ComputeTileIndex** (UINT32 width, UINT32 height, UINT32 pos, bool horizontal, bool isMin)

Private Attributes

- int **m_nLevels**
  number of levels of the image
- **PGFRect** * **m_indices**
  array of tile indices (index is level)

Friends

- class **CWaveletTransform**

Detailed Description

ROI indices.

PGF ROI and tile support. This is a helper class for **CWaveletTransform**.

Author:

C. Stamm

Definition at line 45 of file WaveletTransform.h.

Constructor & Destructor Documentation

CRoiIndices::CRoiIndices () `[inline, private]`

Constructor: Creates a ROI helper object

Definition at line 50 of file WaveletTransform.h.

```
        : m_nLevels(0)
        , m_indices(0)
        {}
```

CRoiIndices::~CRoiIndices () `[inline, private]`

Destructor

Definition at line 57 of file WaveletTransform.h.

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

Member Function Documentation

void CRoiIndices::ComputeIndices (UINT32*width*, UINT32*height*, const PGFRect &*rect*) `[private]`

Compute tile indices for given rectangle (ROI)

Parameters:

width	PGF image width
height	PGF image height
rect	ROI

Definition at line 552 of file WaveletTransform.cpp.

```
                                                                                 {
        ComputeTileIndex(width, height, rect.left, true, true);
        ComputeTileIndex(width, height, rect.top, false, true);
        ComputeTileIndex(width, height, rect.right, true, false);
        ComputeTileIndex(width, height, rect.bottom, false, false);
}
```

void CRoiIndices::ComputeTileIndex (UINT32*width*, UINT32*height*, UINT32*pos*, bool*horizontal*, bool*isMin*) `[private]`

Computes a tile index either in x- or y-direction for a given image position.

Parameters:

width	PGF image width
height	PGF image height
pos	A valid image position: (0 <= pos < width) or (0 <= pos < height)
horizontal	If true, then pos must be a x-value, otherwise a y-value
isMin	If true, then pos is left/top, else pos right/bottom

Definition at line 510 of file WaveletTransform.cpp.

```
{
        ASSERT(m_indices);

        UINT32 m;
        UINT32 tileIndex = 0;
        UINT32 tileMin = 0, tileMax = (horizontal) ? width : height;
        ASSERT(pos <= tileMax);

        // compute tile index with binary search
        for (int i=m_nLevels - 1; i >= 0; i--) {
                // store values
                if (horizontal) {
                        if (isMin) {
                                m_indices[i].left = tileIndex;
                        } else {
                                m_indices[i].right = tileIndex + 1;
                        }
                } else {
                        if (isMin) {
                                m_indices[i].top = tileIndex;
                        } else {
                                m_indices[i].bottom = tileIndex + 1;
                        }
                }

                // compute values
                tileIndex <<= 1;
                m = (tileMin + tileMax)/2;
                if (pos >= m) {
                        tileMin = m;
                        tileIndex++;
                } else {
                        tileMax = m;
                }
        }
}
```

void CRoiIndices::CreateIndices () `[private]`

Definition at line 496 of file WaveletTransform.cpp.

```
                                {
        if (!m_indices) {
                // create tile indices
                m_indices = new PGFRect[m_nLevels];
        }
}
```

void CRoiIndices::Destroy () `[inline, private]`

Definition at line 59 of file WaveletTransform.h.

```
{ delete[] m_indices; m_indices = 0; }
```

const PGFRect& CRoiIndices::GetIndices (int*level*) const `[inline, private]`

Definition at line 62 of file WaveletTransform.h.

```
{ ASSERT(m_indices); ASSERT(level >= 0 && level < m_nLevels); return m_indices[level]; }
```

UINT32 CRoiIndices::GetNofTiles (int*level*) const `[inline]`

Returns the number of tiles in one dimension at given level.

Parameters:

level	A wavelet transform pyramid level (>= 0 && < Levels())

Definition at line 70 of file WaveletTransform.h.

```
{ ASSERT(level >= 0 && level < m_nLevels); return 1 << (m_nLevels - level - 1); }
```

void CRoiIndices::SetLevels (int*levels*) `[inline, private]`

Definition at line 63 of file WaveletTransform.h.

```
{ ASSERT(levels > 0); m_nLevels = levels; }
```

Friends And Related Function Documentation

friend class CWaveletTransform `[friend]`

Definition at line 46 of file WaveletTransform.h.

Member Data Documentation

PGFRect* CRoiIndices::m_indices `[private]`

array of tile indices (index is level)

Definition at line 74 of file WaveletTransform.h.

int CRoiIndices::m_nLevels `[private]`

number of levels of the image

Definition at line 73 of file WaveletTransform.h.

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

- **WaveletTransform.h**
- **WaveletTransform.cpp**

CSubband Class Reference

Wavelet channel class.

```
#include <Subband.h>
```

Public Member Functions

- **CSubband** ()
- **~CSubband** ()
- bool **AllocMemory** ()
- void **FreeMemory** ()
- void **ExtractTile** (**CEncoder** &encoder, bool tile=false, UINT32 tileX=0, UINT32 tileY=0) THROW_
- void **PlaceTile** (**CDecoder** &decoder, int quantParam, bool tile=false, UINT32 tileX=0, UINT32 tileY=0) THROW_
- 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
- void **IncBuffRow** (UINT32 pos)

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
- UINT32 **BufferWidth** () const
- void **TilePosition** (UINT32 tileX, UINT32 tileY, UINT32 &left, UINT32 &top, UINT32 &w, UINT32 &h) const
- const **PGFRect** & **GetROI** () const
- void **SetNTiles** (UINT32 nTiles)
- void **SetROI** (const **PGFRect** &roi)
- void **InitBuffPos** (UINT32 left=0, UINT32 top=0)

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
- **PGFRect m_ROI**
  region of interest
- UINT32 **m_nTiles**
  number of tiles in one dimension in this subband

Friends

- class **CWaveletTransform**

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.

```
: m_size(0)
, m_data(0)
#ifdef __PGFROISUPPORT__
, m_nTiles(0)
#endif
{
}
```

CSubband::~CSubband ()

Destructor.

Definition at line 46 of file Subband.cpp.

```
                    {
        FreeMemory();
}
```

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 72 of file Subband.cpp.

```
                           {
```

```
        UINT32 oldSize = m_size;

#ifdef __PGFROISUPPORT__
        m_size = BufferWidth()*m_ROI.Height();
#endif
        ASSERT(m_size > 0);

        if (m_data) {
                if (oldSize >= m_size) {
                        return true;
                } else {
                        delete[] m_data;
                        m_data = new(std::nothrow) DataT[m_size];
                        return (m_data != 0);
                }
        } else {
                m_data = new(std::nothrow) DataT[m_size];
                return (m_data != 0);
        }
}
```

UINT32 CSubband::BufferWidth () const `[inline, private]`

Definition at line 153 of file Subband.h.

```
{ return m_ROI.Width(); }
```

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 149 of file Subband.cpp.

```
                                        {
        if (m_orientation == LL) {
                quantParam -= m_level + 1;
        } else if (m_orientation == HH) {
                quantParam -= m_level - 1;
        } else {
                quantParam -= m_level;
        }
        if (quantParam > 0) {
                for (UINT32 i=0; i < m_size; i++) {
                        m_data[i] <<= quantParam;
                }
        }
}
```

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 172 of file Subband.cpp.

```
{
#ifdef __PGFROISUPPORT__
        if (tile) {
                // compute tile position and size
                UINT32 xPos, yPos, w, h;
                TilePosition(tileX, tileY, xPos, yPos, w, h);

                // write values into buffer using partitiong scheme
                encoder.Partition(this, w, h, xPos + yPos*m_width, m_width);
        } else
#endif
        {
                // write values into buffer using partitiong scheme
                encoder.Partition(this, m_width, m_height, 0, m_width);
        }
}
```

void CSubband::FreeMemory ()

Delete the memory buffer of this subband.

Definition at line 96 of file Subband.cpp.

```
                          {
        if (m_data) {
                delete[] m_data; m_data = 0;
        }
}
```

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 106 of file Subband.h.

```
{ return m_data; }
```

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

Definition at line 150 of file Subband.h.

```
{ 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 112 of file Subband.h.

```
{ 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 122 of file Subband.h.

```
{ return m_height; }
```

int CSubband::GetLevel () const `[inline]`

Return level of this subband.

Returns:

Level of this subband

Definition at line 117 of file Subband.h.

```
{ 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 134 of file Subband.h.

```
{ return m_orientation; }
```

const PGFRect& CSubband::GetROI () const `[inline, private]`

Definition at line 155 of file Subband.h.

```
{ return m_ROI; }
```

int CSubband::GetWidth () const `[inline]`

Return width of this subband.

Returns:

Width of this subband (in pixels)

Definition at line 127 of file Subband.h.

```
{ return m_width; }
```

void CSubband::IncBuffRow (UINT32*pos*) `[inline]`

Set data buffer position to given position + one row.

Parameters:

pos	Given position

Definition at line 140 of file Subband.h.

```
{ m_dataPos = pos + BufferWidth(); }
```

void CSubband::InitBuffPos (UINT32*left* = 0, UINT32*top* = 0) `[inline, private]`

Definition at line 158 of file Subband.h.

```
{ m_dataPos = top*BufferWidth() + left; ASSERT(m_dataPos < m_size); }
```

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

Definition at line 52 of file Subband.cpp.

```
                                                                                {
        m_width = width;
        m_height = height;
        m_size = m_width*m_height;
        m_level = level;
        m_orientation = orient;
        m_data = 0;
        m_dataPos = 0;
#ifdef __PGFROISUPPORT__
```

```
        m_ROI.left = 0;
        m_ROI.top = 0;
        m_ROI.right = m_width;
        m_ROI.bottom = m_height;
        m_nTiles = 0;
#endif
}
```

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 197 of file Subband.cpp.

```

{
        // allocate memory
        if (!AllocMemory()) ReturnWithError(InsufficientMemory);

        // correct quantParam with normalization factor
        if (m_orientation == LL) {
                quantParam -= m_level + 1;
        } else if (m_orientation == HH) {
                quantParam -= m_level - 1;
        } else {
                quantParam -= m_level;
        }
        if (quantParam < 0) quantParam = 0;

#ifdef __PGFROISUPPORT__
        if (tile) {
                UINT32 xPos, yPos, w, h;

                // compute tile position and size
                TilePosition(tileX, tileY, xPos, yPos, w, h);

                ASSERT(xPos >= m_ROI.left && yPos >= m_ROI.top);
                decoder.Partition(this, quantParam, w, h, (xPos - m_ROI.left) + (yPos -
m_ROI.top)*BufferWidth(), BufferWidth());
        } else
#endif
        {
                // read values into buffer using partitiong scheme
                decoder.Partition(this, quantParam, m_width, m_height, 0, m_width);
        }
}
```

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 107 of file Subband.cpp.

```
                                      {
        if (m_orientation == LL) {
                quantParam -= (m_level + 1);
                // uniform rounding quantization
                if (quantParam > 0) {
                        quantParam--;
                        for (UINT32 i=0; i < m_size; i++) {
                                if (m_data[i] < 0) {
                                        m_data[i] = -(((-m_data[i] >> quantParam) + 1) >> 1);
                                } else {
                                        m_data[i] = ((m_data[i] >> quantParam) + 1) >> 1;
                                }
                        }
                }
        } else {
                if (m_orientation == HH) {
                        quantParam -= (m_level - 1);
                } else {
                        quantParam -= m_level;
                }
                // uniform deadzone quantization
                if (quantParam > 0) {
                        int threshold = ((1 << quantParam) * 7)/5;      // good value
                        quantParam--;
                        for (UINT32 i=0; i < m_size; i++) {
                                if (m_data[i] < -threshold) {
                                        m_data[i] = -(((-m_data[i] >> quantParam) + 1) >> 1);
                                } else if (m_data[i] > threshold) {
                                        m_data[i] = ((m_data[i] >> quantParam) + 1) >> 1;
                                } else {
                                        m_data[i] = 0;
                                }
                        }
                }
        }
}
```

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

Definition at line 148 of file Subband.h.

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

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

Definition at line 147 of file Subband.h.

```
{ 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 101 of file Subband.h.

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

void CSubband::SetNTiles (UINT32*nTiles*) `[inline, private]`

Definition at line 156 of file Subband.h.

```
{ m_nTiles = nTiles; }
```

void CSubband::SetROI (const PGFRect &*roi*) `[inline, private]`

Definition at line 157 of file Subband.h.

```
{ ASSERT(roi.right <= m_width); ASSERT(roi.bottom <= m_height); m_ROI = roi; }
```

void CSubband::TilePosition (UINT32*tileX*, UINT32*tileY*, UINT32 &*xPos*, UINT32 &*yPos*, UINT32 &*w*, UINT32 &*h*) const `[private]`

Compute tile position and size.

Parameters:

tileX	Tile index in x-direction
tileY	Tile index in y-direction
xPos	[out] Offset to left
yPos	[out] Offset to top
w	[out] Tile width
h	[out] Tile height

Definition at line 239 of file Subband.cpp.

```
{
        // example
        // band = HH, w = 30, ldTiles = 2 -> 4 tiles in a row/column
        // --> tile widths
        // 8 7 8 7
        //
        // tile partitioning scheme
        // 0 1 2 3
        // 4 5 6 7
        // 8 9 A B
        // C D E F

        UINT32 nTiles = m_nTiles;
        ASSERT(tileX < nTiles); ASSERT(tileY < nTiles);
        UINT32 m;
        UINT32 left = 0, right = nTiles;
        UINT32 top = 0, bottom = nTiles;

        xPos = 0;
        yPos = 0;
        w = m_width;
        h = m_height;

        while (nTiles > 1) {
                // compute xPos and w with binary search
                m = (left + right) >> 1;
                if (tileX >= m) {
                        xPos += (w + 1) >> 1;
                        w >>= 1;
                        left = m;
                } else {
                        w = (w + 1) >> 1;
                        right = m;
                }
                // compute yPos and h with binary search
                m = (top + bottom) >> 1;
                if (tileY >= m) {
                        yPos += (h + 1) >> 1;
                        h >>= 1;
                        top = m;
                } else {
                        h = (h + 1) >> 1;
                        bottom = m;
                }
                nTiles >>= 1;
        }
        ASSERT(xPos < m_width && (xPos + w <= m_width));
        ASSERT(yPos < m_height && (yPos + h <= m_height));
}
```

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

Definition at line 146 of file Subband.h.

```
{ ASSERT(m_dataPos < m_size); m_data[m_dataPos++] = val; }
```

Friends And Related Function Documentation

friend class CWaveletTransform `[friend]`

Definition at line 43 of file Subband.h.

Member Data Documentation

DataT* CSubband::m_data `[private]`

buffer

Definition at line 170 of file Subband.h.

UINT32 CSubband::m_dataPos `[private]`

current position in m_data

Definition at line 169 of file Subband.h.

UINT32 CSubband::m_height `[private]`

height in pixels

Definition at line 165 of file Subband.h.

int CSubband::m_level `[private]`

recursion level

Definition at line 167 of file Subband.h.

UINT32 CSubband::m_nTiles `[private]`

number of tiles in one dimension in this subband

Definition at line 174 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 168 of file Subband.h.

PGFRect CSubband::m_ROI `[private]`

region of interest

Definition at line 173 of file Subband.h.

UINT32 CSubband::m_size `[private]`

size of data buffer m_data

Definition at line 166 of file Subband.h.

UINT32 CSubband::m_width `[private]`

width in pixels

Definition at line 164 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=NULL)
- **~CWaveletTransform** ()
- OSError **ForwardTransform** (int level, int quant)
- OSError **InverseTransform** (int level, UINT32 *width, UINT32 *height, **DataT** **data)
- **CSubband** * **GetSubband** (int level, **Orientation** orientation)
- void **SetROI** (const **PGFRect** &rect)
- const **PGFRect** & **GetTileIndices** (int level) const
- UINT32 **GetNofTiles** (int level) const
- const **PGFRect** & **GetROI** (int level) const

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 **LinearToMallat** (int destLevel, **DataT** *loRow, **DataT** *hiRow, UINT32 width)
- void **MallatToLinear** (int srcLevel, **DataT** *loRow, **DataT** *hiRow, UINT32 width)

Private Attributes

- **CRoiIndices m_ROIindices**
  ROI indices.
- int **m_nLevels**
  number of transform levels: one more than the number of level 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 84 of file WaveletTransform.h.

Constructor & Destructor Documentation

CWaveletTransform::CWaveletTransform (UINT32*width*, UINT32*height*, int*levels*, DataT **data* = NULL)

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.

```
: m_nLevels(levels + 1)
, m_subband(0)
{
        ASSERT(m_nLevels > 0 && m_nLevels <= MaxLevel + 1);
        InitSubbands(width, height, data);
#ifdef __PGFROISUPPORT__
        m_ROIindices.SetLevels(levels + 1);
#endif
}
```

CWaveletTransform::~CWaveletTransform () [inline]

Destructor

Definition at line 98 of file WaveletTransform.h.

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

Member Function Documentation

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

Definition at line 151 of file WaveletTransform.h.

```
                        {
                delete[] m_subband; m_subband = 0;
        #ifdef __PGFROISUPPORT__
                m_ROIindices.Destroy();
        #endif
        }
```

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

Definition at line 181 of file WaveletTransform.cpp.

```
                                                           {
        if (width >= FilterWidth) {
                UINT32 i = 3;

                // left border handling
                src[1] -= ((src[0] + src[2] + c1) >> 1);
                src[0] += ((src[1] + c1) >> 1);

                // middle part
                for (; i < width-1; i += 2) {
                        src[i] -= ((src[i-1] + src[i+1] + c1) >> 1);
                        src[i-1] += ((src[i-2] + src[i] + c2) >> 2);
                }

                // right border handling
                if (width & 1) {
                        src[i-1] += ((src[i-2] + c1) >> 1);
                } else {
                        src[i] -= src[i-1];
                        src[i-1] += ((src[i-2] + src[i] + c2) >> 2);
                }
        }
}
```

OSError CWaveletTransform::ForwardTransform (int*level*, int*quant*)

Compute fast forward wavelet transform of LL subband at given level and stores result on all 4 subbands of level + 1.

Parameters:

level	A wavelet transform pyramid level (>= 0 && < Levels())
quant	A quantization value (linear scalar quantization)

Returns:

error in case of a memory allocation problem

Definition at line 88 of file WaveletTransform.cpp.

```
                                                                  {
        ASSERT(level >= 0 && level < m_nLevels - 1);
        const int destLevel = level + 1;
        ASSERT(m_subband[destLevel]);
        CSubband* srcBand = &m_subband[level][LL]; ASSERT(srcBand);
        const UINT32 width = srcBand->GetWidth();
        const UINT32 height = srcBand->GetHeight();
        DataT* src = srcBand->GetBuffer(); ASSERT(src);
        DataT *row0, *row1, *row2, *row3;

        // Allocate memory for next transform level
        for (int i=0; i < NSubbands; i++) {
                if (!m_subband[destLevel][i].AllocMemory()) return InsufficientMemory;
        }

        if (height >= FilterHeight) {
                // transform LL subband
                // top border handling
                row0 = src; row1 = row0 + width; row2 = row1 + width;
                ForwardRow(row0, width);
                ForwardRow(row1, width);
                ForwardRow(row2, width);
                for (UINT32 k=0; k < width; k++) {
                        row1[k] -= ((row0[k] + row2[k] + c1) >> 1);
                        row0[k] += ((row1[k] + c1) >> 1);
                }
                LinearToMallat(destLevel, row0, row1, width);
                row0 = row1; row1 = row2; row2 += width; row3 = row2 + width;

                // middle part
                for (UINT32 i=3; i < height-1; i += 2) {
                        ForwardRow(row2, width);
                        ForwardRow(row3, width);
                        for (UINT32 k=0; k < width; k++) {
                                row2[k] -= ((row1[k] + row3[k] + c1) >> 1);
                                row1[k] += ((row0[k] + row2[k] + c2) >> 2);
                        }
                        LinearToMallat(destLevel, row1, row2, width);
                        row0 = row2; row1 = row3; row2 = row3 + width; row3 = row2 + width;
                }

                // bottom border handling
                if (height & 1) {
                        for (UINT32 k=0; k < width; k++) {
                                row1[k] += ((row0[k] + c1) >> 1);
                        }
                        LinearToMallat(destLevel, row1, NULL, width);
                        row0 = row1; row1 += width;
                } else {
                        ForwardRow(row2, width);
                        for (UINT32 k=0; k < width; k++) {
                                row2[k] -= row1[k];
                                row1[k] += ((row0[k] + row2[k] + c2) >> 2);
                        }
                        LinearToMallat(destLevel, row1, row2, width);
                        row0 = row1; row1 = row2; row2 += width;
                }
        } else {
                // if height is too small
                row0 = src; row1 = row0 + width;
                // first part
                for (UINT32 k=0; k < height; k += 2) {
                        ForwardRow(row0, width);
                        ForwardRow(row1, width);
                        LinearToMallat(destLevel, row0, row1, width);
                        row0 += width << 1; row1 += width << 1;
                }
                // bottom
                if (height & 1) {
                        LinearToMallat(destLevel, row0, NULL, width);
                }
        }

        if (quant > 0) {
                // subband quantization (without LL)
                for (int i=1; i < NSubbands; i++) {
                        m_subband[destLevel][i].Quantize(quant);
                }
                // LL subband quantization
                if (destLevel == m_nLevels - 1) {
                        m_subband[destLevel][LL].Quantize(quant);
                }
        }

        // free source band
        srcBand->FreeMemory();
        return NoError;
}
```

UINT32 CWaveletTransform::GetNofTiles (int*level*) const `[inline]`

Get number of tiles in x- or y-direction at given level.

Parameters:

level	A valid subband level.

Definition at line 141 of file WaveletTransform.h.

```
{ return m_ROIindices.GetNofTiles(level); }
```

const PGFRect& CWaveletTransform::GetROI (int*level*) const `[inline]`

Return ROI at given level.

Parameters:

level	A valid subband level.

Definition at line 146 of file WaveletTransform.h.

```
{ return m_subband[level][LL].GetROI(); }
```

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 122 of file WaveletTransform.h.

```
                                                                  {
```

```
                ASSERT(level >= 0 && level < m_nLevels);
                return &m_subband[level][orientation];
        }
```

const PGFRect& CWaveletTransform::GetTileIndices (int*level*) const `[inline]`

Get tile indices of a ROI at given level.

Parameters:

level	A valid subband level.

Definition at line 136 of file WaveletTransform.h.

```
{ return m_ROIindices.GetIndices(level); }
```

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

Definition at line 53 of file WaveletTransform.cpp.

```
                                                                               {
        if (m_subband) Destroy();

        // create subbands
        m_subband = new CSubband[m_nLevels][NSubbands];

        // init subbands
        UINT32 loWidth = width;
        UINT32 hiWidth = width;
        UINT32 loHeight = height;
        UINT32 hiHeight = height;

        for (int level = 0; level < m_nLevels; level++) {
                m_subband[level][LL].Initialize(loWidth, loHeight, level, LL);  // LL
                m_subband[level][HL].Initialize(hiWidth, loHeight, level, HL);  //    HL
                m_subband[level][LH].Initialize(loWidth, hiHeight, level, LH);  // LH
                m_subband[level][HH].Initialize(hiWidth, hiHeight, level, HH);  //    HH
                hiWidth = loWidth >> 1;                 hiHeight = loHeight >> 1;
                loWidth = (loWidth + 1) >> 1;   loHeight = (loHeight + 1) >> 1;
        }
        if (data) {
                m_subband[0][LL].SetBuffer(data);
        }
}
```

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

Definition at line 367 of file WaveletTransform.cpp.

```
                                                             {
        if (width >= FilterWidth) {
                UINT32 i = 2;

                // left border handling
                dest[0] -= ((dest[1] + c1) >> 1);

                // middle part
                for (; i < width - 1; i += 2) {
                        dest[i] -= ((dest[i-1] + dest[i+1] + c2) >> 2);
                        dest[i-1] += ((dest[i-2] + dest[i] + c1) >> 1);
                }

                // right border handling
                if (width & 1) {
                        dest[i] -= ((dest[i-1] + c1) >> 1);
                        dest[i-1] += ((dest[i-2] + dest[i] + c1) >> 1);
                } else {
                        dest[i-1] += dest[i-2];
                }
        }
}
```

OSError CWaveletTransform::InverseTransform (int*level*, UINT32 **width*, UINT32 **height*, DataT ***data*)

Compute fast inverse wavelet transform of all 4 subbands of given level and stores result in LL subband of level - 1.

Parameters:

level	A wavelet transform pyramid level (> 0 && <= Levels())
width	A pointer to the returned width of subband LL (in pixels)
height	A pointer to the returned height of subband LL (in pixels)
data	A pointer to the returned array of image data

Returns:

error in case of a memory allocation problem

Definition at line 245 of file WaveletTransform.cpp.

```
                                                                                        {
        ASSERT(srcLevel > 0 && srcLevel < m_nLevels);
        const int destLevel = srcLevel - 1;
        ASSERT(m_subband[destLevel]);
        CSubband* destBand = &m_subband[destLevel][LL];
        UINT32 width, height;

        // allocate memory for the results of the inverse transform
        if (!destBand->AllocMemory()) return InsufficientMemory;
        DataT *dest = destBand->GetBuffer(), *origin = dest, *row0, *row1, *row2, *row3;

#ifdef __PGFROISUPPORT__
        PGFRect destROI = destBand->GetROI();   // is valid only after AllocMemory
        width = destROI.Width();
        height = destROI.Height();
        const UINT32 destWidth = width; // destination buffer width
        const UINT32 destHeight = height; // destination buffer height

        // update destination ROI
        if (destROI.top & 1) {
                destROI.top++;
                origin += destWidth;
                height--;
        }
        if (destROI.left & 1) {
                destROI.left++;
                origin++;
                width--;
        }

        // init source buffer position
        for (int i=0; i < NSubbands; i++) {
                UINT32 left = (destROI.left >> 1) - m_subband[srcLevel][i].GetROI().left;
                UINT32 top = (destROI.top >> 1) - m_subband[srcLevel][i].GetROI().top;
                m_subband[srcLevel][i].InitBuffPos(left, top);
        }
#else
        width = destBand->GetWidth();
        height = destBand->GetHeight();
        PGFRect destROI(0, 0, width, height);
        const UINT32 destWidth = width; // destination buffer width
        const UINT32 destHeight = height; // destination buffer height

        // init source buffer position
        for (int i=0; i < NSubbands; i++) {
                m_subband[srcLevel][i].InitBuffPos();
        }
#endif

        if (destHeight >= FilterHeight) {
                // top border handling
                row0 = origin; row1 = row0 + destWidth;
                MallatToLinear(srcLevel, row0, row1, width);
                for (UINT32 k=0; k < width; k++) {
                        row0[k] -= ((row1[k] + c1) >> 1);
                }

                // middle part
                row2 = row1 + destWidth; row3 = row2 + destWidth;
                for (UINT32 i=destROI.top + 2; i < destROI.bottom - 1; i += 2) {
                        MallatToLinear(srcLevel, row2, row3, width);
                        for (UINT32 k=0; k < width; k++) {
                                row2[k] -= ((row1[k] + row3[k] + c2) >> 2);
                                row1[k] += ((row0[k] + row2[k] + c1) >> 1);
                        }
                        InverseRow(row0, width);
                        InverseRow(row1, width);
                        row0 = row2; row1 = row3; row2 = row1 + destWidth; row3 = row2 + destWidth;
                }

                // bottom border handling
                if (height & 1) {
                        MallatToLinear(srcLevel, row2, NULL, width);
                        for (UINT32 k=0; k < width; k++) {
                                row2[k] -= ((row1[k] + c1) >> 1);
                                row1[k] += ((row0[k] + row2[k] + c1) >> 1);
                        }
                        InverseRow(row0, width);
                        InverseRow(row1, width);
                        InverseRow(row2, width);
                        row0 = row1; row1 = row2; row2 += destWidth;
                } else {
                        for (UINT32 k=0; k < width; k++) {
                                row1[k] += row0[k];
                        }
                        InverseRow(row0, width);
                        InverseRow(row1, width);
                        row0 = row1; row1 += destWidth;
                }
        } else {
                // height is too small
                row0 = origin; row1 = row0 + destWidth;
                // first part
                for (UINT32 k=0; k < height; k += 2) {
                        MallatToLinear(srcLevel, row0, row1, width);
                        InverseRow(row0, width);
                        InverseRow(row1, width);
                        row0 += destWidth << 1; row1 += destWidth << 1;
                }
                // bottom
                if (height & 1) {
                        MallatToLinear(srcLevel, row0, NULL, width);
                        InverseRow(row0, width);
                }
        }

        // free memory of the current srcLevel
        for (int i=0; i < NSubbands; i++) {
                m_subband[srcLevel][i].FreeMemory();
        }

        // return info
        *w = destWidth;
        *h = height;
        *data = dest;
        return NoError;
}
```

void CWaveletTransform::LinearToMallat (intdestLevel, DataT *loRow, DataT *hiRow, UINT32width) [private]

Definition at line 207 of file WaveletTransform.cpp.

```
{
        const UINT32 wquot = width >> 1;
        const bool wrem = width & 1;
        CSubband &ll = m_subband[destLevel][LL], &hl = m_subband[destLevel][HL];
        CSubband &lh = m_subband[destLevel][LH], &hh = m_subband[destLevel][HH];

        if (hiRow) {
                for (UINT32 i=0; i < wquot; i++) {
                        ll.WriteBuffer(*loRow++);       // first access, than increment
                        hl.WriteBuffer(*loRow++);
                        lh.WriteBuffer(*hiRow++);       // first access, than increment
                        hh.WriteBuffer(*hiRow++);
                }
                if (wrem) {
                        ll.WriteBuffer(*loRow);
                        lh.WriteBuffer(*hiRow);
                }
        } else {
                for (UINT32 i=0; i < wquot; i++) {
                        ll.WriteBuffer(*loRow++);       // first access, than increment
                        hl.WriteBuffer(*loRow++);
                }
                if (wrem) ll.WriteBuffer(*loRow);
        }
}
```

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

Definition at line 392 of file WaveletTransform.cpp.

```
                                                                                                    {
        const UINT32 wquot = width >> 1;
        const bool wrem = width & 1;
        CSubband &ll = m_subband[srcLevel][LL], &hl = m_subband[srcLevel][HL];
        CSubband &lh = m_subband[srcLevel][LH], &hh = m_subband[srcLevel][HH];

        if (hiRow) {
        #ifdef __PGFROISUPPORT__
                const bool storePos = wquot < ll.BufferWidth();
                UINT32 llPos = 0, hlPos = 0, lhPos = 0, hhPos = 0;

                if (storePos) {
                        // save current src buffer positions
                        llPos = ll.GetBuffPos();
                        hlPos = hl.GetBuffPos();
                        lhPos = lh.GetBuffPos();
                        hhPos = hh.GetBuffPos();
                }
        #endif

                for (UINT32 i=0; i < wquot; i++) {
                        *loRow++ = ll.ReadBuffer();// first access, than increment
                        *loRow++ = hl.ReadBuffer();// first access, than increment
                        *hiRow++ = lh.ReadBuffer();// first access, than increment
                        *hiRow++ = hh.ReadBuffer();// first access, than increment
                }

                if (wrem) {
                        *loRow++ = ll.ReadBuffer();// first access, than increment
                        *hiRow++ = lh.ReadBuffer();// first access, than increment
                }

        #ifdef __PGFROISUPPORT__
                if (storePos) {
                        // increment src buffer positions
                        ll.IncBuffRow(llPos);
                        hl.IncBuffRow(hlPos);
                        lh.IncBuffRow(lhPos);
                        hh.IncBuffRow(hhPos);
                }
        #endif
        } else {
        #ifdef __PGFROISUPPORT__
                const bool storePos = wquot < ll.BufferWidth();
                UINT32 llPos = 0, hlPos = 0;

                if (storePos) {
                        // save current src buffer positions
                        llPos = ll.GetBuffPos();
                        hlPos = hl.GetBuffPos();
                }
        #endif

                for (UINT32 i=0; i < wquot; i++) {
                        *loRow++ = ll.ReadBuffer();// first access, than increment
                        *loRow++ = hl.ReadBuffer();// first access, than increment
                }
                if (wrem) *loRow++ = ll.ReadBuffer();

        #ifdef __PGFROISUPPORT__
                if (storePos) {
                        // increment src buffer positions
                        ll.IncBuffRow(llPos);
                        hl.IncBuffRow(hlPos);
                }
        #endif
        }
}
```

void CWaveletTransform::SetROI (const PGFRect &*rect*)

Compute and store ROIs for each level

Parameters:

rect	rectangular region of interest (ROI)

Definition at line 466 of file WaveletTransform.cpp.

```
                                                  {
        // create tile indices
        m_ROIindices.CreateIndices();

        // compute tile indices
        m_ROIindices.ComputeIndices(m_subband[0][LL].GetWidth(), m_subband[0][LL].GetHeight(),
rect);

        // compute ROIs
        UINT32 w, h;
        PGFRect r;

        for (int i=0; i < m_nLevels; i++) {
                const PGFRect& indices = m_ROIindices.GetIndices(i);

                for (int o=0; o < NSubbands; o++) {
                        CSubband& subband = m_subband[i][o];

                        subband.SetNTiles(m_ROIindices.GetNofTiles(i)); // must be called before
TilePosition()
                        subband.TilePosition(indices.left, indices.top, r.left, r.top, w, h);
                        subband.TilePosition(indices.right - 1, indices.bottom - 1, r.right,
r.bottom, w, h);
                        r.right += w;
                        r.bottom += h;
                        subband.SetROI(r);
                }
        }
}
```

Friends And Related Function Documentation

friend class CSubband `[friend]`

Definition at line 85 of file WaveletTransform.h.

Member Data Documentation

int CWaveletTransform::m_nLevels `[private]`

number of transform levels: one more than the number of level in PGFimage

Definition at line 167 of file WaveletTransform.h.

CRoiIndices CWaveletTransform::m_ROIindices `[private]`

ROI indices.

Definition at line 164 of file WaveletTransform.h.

CSubband(* CWaveletTransform::m_subband)[NSubbands] `[private]`

quadtree of subbands: LL HL LH HH

Definition at line 168 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 180 of file PGFtypes.h.

Constructor & Destructor Documentation

IOException::IOException () `[inline]`

Standard constructor.

Definition at line 182 of file PGFtypes.h.

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

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

Constructor

Parameters:

err	Run-time error

Definition at line 185 of file PGFtypes.h.

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

Member Data Documentation

OSError IOException::error

operating system error code

Definition at line 187 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 DWT levels
- 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
- UINT8 **reserved1**
- UINT8 **reserved2**
  not used

Detailed Description

PGF header.

PGF header contains image information

Author:

C. Stamm

Definition at line 123 of file PGFtypes.h.

Constructor & Destructor Documentation

PGFHeader::PGFHeader () `[inline]`

Definition at line 124 of file PGFtypes.h.

```
: width(0), height(0), nLevels(0), quality(0), bpp(0), channels(0), mode(ImageModeUnknown),
usedBitsPerChannel(0), reserved1(0), reserved2(0) {}
```

Member Data Documentation

UINT8 PGFHeader::bpp

bits per pixel

Definition at line 129 of file PGFtypes.h.

UINT8 PGFHeader::channels

number of channels

Definition at line 130 of file PGFtypes.h.

UINT32 PGFHeader::height

image height in pixels

Definition at line 126 of file PGFtypes.h.

UINT8 PGFHeader::mode

image mode according to Adobe's image modes

Definition at line 131 of file PGFtypes.h.

UINT8 PGFHeader::nLevels

number of DWT levels

Definition at line 127 of file PGFtypes.h.

UINT8 PGFHeader::quality

quantization parameter: 0=lossless, 4=standard, 6=poor quality

Definition at line 128 of file PGFtypes.h.

UINT8 PGFHeader::reserved1

Definition at line 133 of file PGFtypes.h.

UINT8 PGFHeader::reserved2

not used

Definition at line 133 of file PGFtypes.h.

UINT8 PGFHeader::usedBitsPerChannel

number of used bits per channel in 16- and 32-bit per channel modes

Definition at line 132 of file PGFtypes.h.

UINT32 PGFHeader::width

image width in pixels

Definition at line 125 of file PGFtypes.h.

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

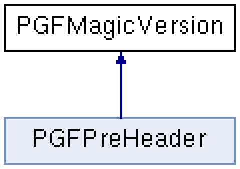
- **PGFtypes.h**

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 PGFPreHeaderV6 **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 104 of file PGFtypes.h.

Member Data Documentation

char PGFMagicVersion::magic[3]

PGF identification = "PGF".

Definition at line 105 of file PGFtypes.h.

UINT8 PGFMagicVersion::version

PGF version.

Definition at line 106 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
- UINT8 * **userData**
  user data of size userDataLen
- UINT32 **userDataLen**
  user data size in bytes

Detailed Description

Optional PGF post-header.

PGF post-header is optional. It contains color table and user data

Author:

C. Stamm

Definition at line 141 of file PGFtypes.h.

Member Data Documentation

RGBQUAD PGFPostHeader::clut[ColorTableLen]

color table for indexed color images

Definition at line 142 of file PGFtypes.h.

UINT8* PGFPostHeader::userData

user data of size userDataLen

Definition at line 143 of file PGFtypes.h.

UINT32 PGFPostHeader::userDataLen

user data size in bytes

Definition at line 144 of file PGFtypes.h.

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

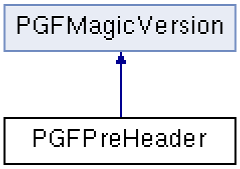
- **PGFtypes.h**

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*
- 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 114 of file PGFtypes.h.

Member Data Documentation

UINT32 PGFPreHeader::hSize

total size of **PGFHeader**, [ColorTable], and [UserData] in bytes

Definition at line 115 of file PGFtypes.h.

char PGFMagicVersion::magic[3] `[inherited]`

PGF identification = "PGF".

Definition at line 105 of file PGFtypes.h.

UINT8 PGFMagicVersion::version `[inherited]`

PGF version.

Definition at line 106 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 194 of file PGFtypes.h.

Constructor & Destructor Documentation

PGFRect::PGFRect () `[inline]`

Standard constructor.

Definition at line 196 of file PGFtypes.h.

```
: 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 202 of file PGFtypes.h.

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

Member Function Documentation

UINT32 PGFRect::Height () const `[inline]`

Returns:

Rectangle height

Definition at line 207 of file PGFtypes.h.

```
{ return bottom - top; }
```

bool PGFRect::IsInside (UINT32*x*, UINT32*y*) const `[inline]`

Test if point (x,y) is inside this rectangle

Parameters:

x	Point coordinate x
y	Point coordinate y

Returns:

True if point (x,y) is inside this rectangle

Definition at line 213 of file PGFtypes.h.

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

UINT32 PGFRect::Width () const `[inline]`

Returns:

Rectangle width

Definition at line 205 of file PGFtypes.h.

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

Member Data Documentation

UINT32 PGFRect::bottom

Definition at line 215 of file PGFtypes.h.

UINT32 PGFRect::left

Definition at line 215 of file PGFtypes.h.

UINT32 PGFRect::right

Definition at line 215 of file PGFtypes.h.

UINT32 PGFRect::top

Definition at line 215 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 162 of file PGFtypes.h.

Member Data Documentation

UINT16 ROIBlockHeader::RBH::bufferSize

number of uncoded UINT32 values in a block

Definition at line 167 of file PGFtypes.h.

UINT16 ROIBlockHeader::RBH::tileEnd

1: last part of a tile

Definition at line 168 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 151 of file PGFtypes.h.

Constructor & Destructor Documentation

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

Constructor

Parameters:

v	Buffer size

Definition at line 154 of file PGFtypes.h.

```
{ 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 158 of file PGFtypes.h.

```
{ 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 160 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"
```

Defines

- #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

Definition in file **BitStream.h**.

Define 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 40 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 260 of file BitStream.h.

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

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 56 of file BitStream.h.

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

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 155 of file BitStream.h.

```
                                                                  {
        ASSERT(len > 0);
        const UINT32 iFirstInt = pos >> WordWidthLog;
        const UINT32 iLastInt = (pos + len - 1) >> WordWidthLog;

        const UINT32 startMask = Filled << (pos%WordWidth);
//      const UINT32 endMask=Filled>>(WordWidth-1-((pos+len-1)%WordWidth));

        if (iFirstInt == iLastInt) {
                stream[iFirstInt] &= ~(startMask /*& endMask*/);
        } else {
                stream[iFirstInt] &= ~startMask;
                for (UINT32 i = iFirstInt + 1; i <= iLastInt; i++) { // changed <=
                        stream[i] = 0;
                }
                //stream[iLastInt] &= ~endMask;
        }
}
```

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 77 of file BitStream.h.

```
                                                                                {
        const UINT32 iLoInt = pos >> WordWidthLog;
        const UINT32 iHiInt = (pos + k - 1) >> WordWidthLog;
        ASSERT(iLoInt <= iHiInt);
        const UINT32 mask = (Filled >> (WordWidth - k));

        if (iLoInt == iHiInt) {
                // fits into one integer
                val &= mask;
                val <<= (pos%WordWidth);
                return (stream[iLoInt] & val) == val;
        } else {
                // must be splitted over integer boundary
                UINT64 v1 = MAKEU64(stream[iLoInt], stream[iHiInt]);
                UINT64 v2 = UINT64(val & mask) << (pos%WordWidth);
                return (v1 & v2) == v2;
        }
}
```

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 65 of file BitStream.h.

```
                                                    {
        return (stream[pos >> WordWidthLog] & (1 << (pos%WordWidth))) > 0;

}
```

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 128 of file BitStream.h.

```
                                                                  {
        UINT32 count, hiCount;
        const UINT32 iLoInt = pos >> WordWidthLog;                              // integer of
first bit
        const UINT32 iHiInt = (pos + k - 1) >> WordWidthLog;            // integer of last bit
        const UINT32 loMask = Filled << (pos%WordWidth);
        const UINT32 hiMask = Filled >> (WordWidth - 1 - ((pos + k - 1)%WordWidth));

        if (iLoInt == iHiInt) {
                // inside integer boundary
                count = stream[iLoInt] & (loMask & hiMask);
                count >>= pos%WordWidth;
        } else {
                // overlapping integer boundary
                count = stream[iLoInt] & loMask;
                count >>= pos%WordWidth;
                hiCount = stream[iHiInt] & hiMask;
                hiCount <<= WordWidth - (pos%WordWidth);
                count |= hiCount;
        }
        return count;
}
```

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 269 of file BitStream.h.

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

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 235 of file BitStream.h.

```
                                                                    {
        UINT32 count = 0;
        UINT32 testMask = 1 << (pos%WordWidth);
        UINT32* word = stream + (pos >> WordWidthLog);

        while (((*word & testMask) != 0) && (count < len)) {
                count++;
                testMask <<= 1;
                if (!testMask) {
                        word++; testMask = 1;

                        // fast steps if all bits in a word are one
                        while ((count + WordWidth <= len) && (*word == Filled)) {
                                word++;
                                count += WordWidth;
                        }
                }
        }
        return count;
}
```

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 206 of file BitStream.h.

```
                                                                    {
        UINT32 count = 0;
        UINT32 testMask = 1 << (pos%WordWidth);
        UINT32* word = stream + (pos >> WordWidthLog);

        while (((*word & testMask) == 0) && (count < len)) {
                count++;
                testMask <<= 1;
                if (!testMask) {
                        word++; testMask = 1;

                        // fast steps if all bits in a word are zero
                        while ((count + WordWidth <= len) && (*word == 0)) {
                                word++;
                                count += WordWidth;
                        }
                }
        }

        return count;
}
```

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 48 of file BitStream.h.

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

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 179 of file BitStream.h.

```
                                                                 {
        ASSERT(len > 0);

        const UINT32 iFirstInt = pos >> WordWidthLog;
        const UINT32 iLastInt = (pos + len - 1) >> WordWidthLog;

        const UINT32 startMask = Filled << (pos%WordWidth);
//      const UINT32 endMask=Filled>>(WordWidth-1-((pos+len-1)%WordWidth));

        if (iFirstInt == iLastInt) {
                stream[iFirstInt] |= (startMask /*& endMask*/);
        } else {
                stream[iFirstInt] |= startMask;
                for (UINT32 i = iFirstInt + 1; i <= iLastInt; i++) { // changed <=
                        stream[i] = Filled;
                }
                //stream[iLastInt] &= ~endMask;
        }
}
```

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 102 of file BitStream.h.

```
                                                                            {
        const UINT32 offset = pos%WordWidth;
        const UINT32 iLoInt = pos >> WordWidthLog;
        const UINT32 iHiInt = (pos + k - 1) >> WordWidthLog;
        ASSERT(iLoInt <= iHiInt);
        const UINT32 loMask = Filled << offset;
        const UINT32 hiMask = Filled >> (WordWidth - 1 - ((pos + k - 1)%WordWidth));

        if (iLoInt == iHiInt) {
                // fits into one integer
                stream[iLoInt] &= ~(loMask & hiMask); // clear bits
                stream[iLoInt] |= val << offset; // write value
        } else {
                // must be splitted over integer boundary
                stream[iLoInt] &= ~loMask; // clear bits
                stream[iLoInt] |= val << offset; // write lower part of value
                stream[iHiInt] &= ~hiMask; // clear bits
                stream[iHiInt] |= val >> (WordWidth - offset); // write higher part of value
        }
}
```

Variable Documentation

const UINT32 Filled = 0xFFFFFFFF [static]

Definition at line 37 of file BitStream.h.

Decoder.cpp File Reference

PGF decoder class implementation.

```
#include "Decoder.h"
```

Defines

- #define **CodeBufferBitLen**   (CodeBufferLen*WordWidth)
  max number of bits in m_codeBuffer
- #define **MaxCodeLen**   ((1 << RLblockSizeLen) - 1)
  max length of RL encoded block

Detailed Description

PGF decoder class implementation.

Author:

C. Stamm, R. Spuler

Definition in file **Decoder.cpp**.

Define 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) Defines

- #define **BufferLen**   (BufferSize/WordWidth)
  number of words per buffer
- #define **CodeBufferLen**   BufferSize
  number of words in code buffer (CodeBufferLen > BufferLen)

Detailed Description

PGF decoder class.

Author:

C. Stamm, R. Spuler

Definition in file **Decoder.h**.

Define 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"
```

Defines

- #define **CodeBufferBitLen**   (CodeBufferLen*WordWidth)
  max number of bits in m_codeBuffer
- #define **MaxCodeLen**   ((1 << RLblockSizeLen) - 1)
  max length of RL encoded block

Detailed Description

PGF encoder class implementation.

Author:

C. Stamm, R. Spuler

Definition in file **Encoder.cpp**.

Define 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) Defines

- #define **BufferLen** (BufferSize/WordWidth)
  number of words per buffer
- #define **CodeBufferLen** BufferSize
  number of words in code buffer (CodeBufferLen > BufferLen)

Detailed Description

PGF encoder class.

Author:

C. Stamm, R. Spuler

Definition in file **Encoder.h**.

Define 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 <cmath>
#include <cstring>
```

Defines

- #define **YUVoffset4**   8
- #define **YUVoffset6**   32
- #define **YUVoffset8**   128
- #define **YUVoffset16**   32768

Detailed Description

PGF image class implementation.

Author:

C. Stamm

Definition in file **PGFimage.cpp**.

Define Documentation

#define YUVoffset16   32768

Definition at line 38 of file PGFimage.cpp.

#define YUVoffset4   8

Definition at line 35 of file PGFimage.cpp.

#define YUVoffset6   32

Definition at line 36 of file PGFimage.cpp.

#define YUVoffset8   128

Definition at line 37 of file PGFimage.cpp.

PGFimage.h File Reference

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

Classes

- class **CPGFImage**

PGF main class. Enumerations

- enum **ProgressMode** { **PM_Relative**, **PM_Absolute** }

Detailed Description

PGF image class.

Author:

C. Stamm

Definition in file **PGFimage.h**.

Enumeration Type Documentation

enum ProgressMode

Enumerator:

PM_Relative
PM_Absolute

Definition at line 36 of file PGFimage.h.

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

PGFplatform.h File Reference

PGF platform specific definitions.

```
#include <cassert>
#include <cmath>
#include <cstdlib>
```

Defines

- #define **__PGFROISUPPORT__**
- #define **__PGF32SUPPORT__**
- #define **WordWidth** 32
  *WordBytes*8.*
- #define **WordWidthLog** 5
  ld of WordWidth
- #define **WordMask** 0xFFFFFFE0
  least WordWidthLog bits are zero
- #define **WordBytes** 4
  sizeof(UINT32)
- #define **WordBytesMask** 0xFFFFFFFC
  least WordBytesLog bits are zero
- #define **WordBytesLog** 2
  ld of WordBytes
- #define **DWWIDTHBITS**(bits) (((bits) + WordWidth - 1) & WordMask)
  aligns scanline width in bits to DWORD value
- #define **DWWIDTH**(bytes) (((bytes) + WordBytes - 1) & WordBytesMask)
  aligns scanline width in bytes to DWORD value
- #define **DWWIDTHREST**(bytes) ((WordBytes - (bytes)%WordBytes)%WordBytes)
  ***DWWIDTH(bytes)** - bytes.*
- #define **__min**(x, y) ((x) <= (y) ? (x) : (y))
- #define **__max**(x, y) ((x) >= (y) ? (x) : (y))
- #define **ImageModeBitmap** 0
- #define **ImageModeGrayScale** 1
- #define **ImageModeIndexedColor** 2
- #define **ImageModeRGBColor** 3
- #define **ImageModeCMYKColor** 4
- #define **ImageModeHSLColor** 5
- #define **ImageModeHSBColor** 6
- #define **ImageModeMultichannel** 7
- #define **ImageModeDuotone** 8
- #define **ImageModeLabColor** 9
- #define **ImageModeGray16** 10
- #define **ImageModeRGB48** 11
- #define **ImageModeLab48** 12
- #define **ImageModeCMYK64** 13
- #define **ImageModeDeepMultichannel** 14
- #define **ImageModeDuotone16** 15
- #define **ImageModeRGBA** 17
- #define **ImageModeGray32** 18
- #define **ImageModeRGB12** 19

- #define **ImageModeRGB16**   20
- #define **ImageModeUnknown**   255
- #define **__VAL**(x)   (x)

Detailed Description

PGF platform specific definitions.

Author:

C. Stamm

Definition in file **PGFplatform.h**.

Define 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 604 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

Definition in file **PGFstream.cpp**.

PGFstream.h File Reference

PGF stream class.

```
#include "PGFtypes.h"
#include <new>
```

Classes

- class **CPGFStream**
- *Abstract stream base class.* class **CPGFFileStream**
- *File stream class.* class **CPGFMemoryStream**

Memory stream class.

Detailed Description

PGF stream class.

Author:

C. Stamm

Definition in file **PGFstream.h**.

PGFtypes.h File Reference

PGF definitions.

```
#include "PGFplatform.h"
```

Classes

- struct **PGFMagicVersion**
- *PGF identification and version.* struct **PGFPreHeader**
- *PGF pre-header.* 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. Defines

- #define **PGFCodecVersion** "6.12.24"
  Minor number: Year (2) Week (2)
- #define **PGFCodecVersionID** 0x061224
  Codec version ID to use for API check in client implementation.
- #define **Magic** "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
  new HeaderSize: 32 bits instead of 16 bits
- #define **PGFVersion** (Version2 | PGF32 | Version5 | Version6)
  current standard version
- #define **BufferSize** 16384
  must be a multiple of WordWidth

- #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**)

Typedefs

- typedef INT32 **DataT**
- typedef void(* **RefreshCB**)(void *p)

Enumerations

- enum **Orientation** { **LL** = 0, **HL** = 1, **LH** = 2, **HH** = 3 }

Detailed Description

PGF definitions.

Author:

C. Stamm

Definition in file **PGFtypes.h**.

Define Documentation

#define BufferSize 16384

must be a multiple of WordWidth

Definition at line 77 of file PGFtypes.h.

#define ColorTableLen 256

size of color lookup table (clut)

Definition at line 60 of file PGFtypes.h.

#define ColorTableSize   ColorTableLen*sizeof(RGBQUAD)

Definition at line 232 of file PGFtypes.h.

#define DataTSize   sizeof(DataT)

Definition at line 233 of file PGFtypes.h.

#define DownsampleThreshold   3

if quality is larger than this threshold than downsampling is used

Definition at line 59 of file PGFtypes.h.

#define HeaderSize   sizeof(PGFHeader)

Definition at line 231 of file PGFtypes.h.

#define InterBlockSize   4

side length of a coefficient block in a HL or LH subband

Definition at line 80 of file PGFtypes.h.

#define LinBlockSize   8

side length of a coefficient block in a HH or LL subband

Definition at line 79 of file PGFtypes.h.

#define Magic   "PGF"

PGF identification.

Definition at line 55 of file PGFtypes.h.

#define MagicVersionSize   sizeof(PGFMagicVersion)

Definition at line 229 of file PGFtypes.h.

#define MaxBitPlanes   31

maximum number of bit planes of m_value: 32 minus sign bit

Definition at line 82 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 86 of file PGFtypes.h.

#define MaxChannels   8

maximum number of (color) channels

Definition at line 58 of file PGFtypes.h.

#define MaxLevel   30

maximum number of transform levels

Definition at line 56 of file PGFtypes.h.

#define MaxQuality   MaxBitPlanes

maximum quality

Definition at line 87 of file PGFtypes.h.

#define NSubbands   4

number of subbands per level

Definition at line 57 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 63 of file PGFtypes.h.

#define PGFCodecVersion   "6.12.24"

Minor number: Year (2) Week (2)

Major number

Definition at line 48 of file PGFtypes.h.

#define PGFCodecVersionID   0x061224

Codec version ID to use for API check in client implementation.

Definition at line 50 of file PGFtypes.h.

#define PGFROI   8

supports Regions Of Interest

Definition at line 64 of file PGFtypes.h.

#define PGFVersion (Version2 | PGF32 | Version5 | Version6)

current standard version

Definition at line 69 of file PGFtypes.h.

#define PreHeaderSize sizeof(PGFPreHeader)

Definition at line 230 of file PGFtypes.h.

#define RLblockSizeLen 15

block size length (< 16): ld(BufferSize) < RLblockSizeLen <= 2*ld(BufferSize)

Definition at line 78 of file PGFtypes.h.

#define Version2 2

data structure **PGFHeader** of major version 2

Definition at line 62 of file PGFtypes.h.

#define Version5 16

new coding scheme since major version 5

Definition at line 65 of file PGFtypes.h.

#define Version6 32

new HeaderSize: 32 bits instead of 16 bits

Definition at line 66 of file PGFtypes.h.

Typedef Documentation

typedef INT32 DataT

Definition at line 219 of file PGFtypes.h.

typedef void(* RefreshCB)(void *p)

Definition at line 224 of file PGFtypes.h.

Enumeration Type Documentation

enum Orientation

Enumerator:

- ***LL***
- ***HL***
- ***LH***
- ***HH***

Definition at line 92 of file PGFtypes.h.

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

Subband.cpp File Reference

PGF wavelet subband class implementation.

```
#include "Subband.h"
#include "Encoder.h"
#include "Decoder.h"
```

Detailed Description

PGF wavelet subband class implementation.

Author:

C. Stamm

Definition in file **Subband.cpp**.

Subband.h File Reference

PGF wavelet subband class.

```
#include "PGFtypes.h"
```

Classes

- class **CSubband**

Wavelet channel class.

Detailed Description

PGF wavelet subband class.

Author:

C. Stamm

Definition in file **Subband.h**.

WaveletTransform.cpp File Reference

PGF wavelet transform class implementation.

```
#include "WaveletTransform.h"
```

Defines

- #define **c1** 1
- #define **c2** 2

Detailed Description

PGF wavelet transform class implementation.

Author:

C. Stamm

Definition in file **WaveletTransform.cpp**.

Define 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 **CRoiIndices**
- *ROI indices.* class **CWaveletTransform**

PGF wavelet transform. Defines

- #define **FilterWidth** 5
  number of coefficients of the row wavelet filter
- #define **FilterHeight** 3
  number of coefficients of the column wavelet filter

Detailed Description

PGF wavelet transform class.

Author:

C. Stamm

Definition in file **WaveletTransform.h**.

Define Documentation

#define FilterHeight 3

number of coefficients of the column wavelet filter

Definition at line 38 of file WaveletTransform.h.

#define FilterWidth 5

number of coefficients of the row wavelet filter

Definition at line 37 of file WaveletTransform.h.

Index