GNOSIS Map Tiles

Specifications Draft version 0.2

Contributors

Jérôme Jacovella-St-Louis, Ecere Corporation *jerome@ecere.com*

Alexis Naveros, Ecere Corporation *alexis@ecere.com*

Layout for binary representation of tiled geospatial data
(including vector tiles, embedded and referenced 3D models, point clouds, imagery & coverages)

Note	• Offsets and sizes are specified in decimal bytes. • Despite MSB being network byte ordering, values are encoded as little-endian (Least Significant Bit first) to avoid a very significant amount of byte swapping, accommodating today's most common architectures.

The tile data is prefixed by a 24 bytes header:

GNOSIS Map Tile Header

Offset	Type	Size	Name
0	char	3	Signature: The 3 characters GMT
3	uint8	1	Major (Currently 1)
4	uint8	1	Minor (Currently 0)
5	uint8	1	Type *(See Types table below)*
6	uint16	2	Flags *(See Flags table below)*
8	uint64	8	Tile Key (tiling scheme specific) GNOSIS Global Grid layout *(from high to low bits)*: level (5 bits: 0..28), latitude index (29 bits), longitude index (30 bits)
16	uint32	4	Size of uncompressed data excluding header
20	uint8	1	Encoding *(See Encodings table below)*
21	uint	3	Compressed size (note: 3 bytes only)
24	**Total size of header**		

If the tile is not flagged as empty or full, the actual tile data follows, based on geospatial data type.

The following ***Types*** are currently defined:

Type	**Value**	**Notes**
Vector types		
vectorPoints	0x10	Points vector type
vector3DPoints	0x11	This implies a 16-bit Z value per point
vector3DPoints32	0x12	This implies a 32-bit Z value per point
vectorLines	0x14	Lines vector type
vector3DLine	0x15	This implies a 16-bit Z value per point
vector3DLines32	0x16	This implies a 32-bit Z value per point

vectorPolygons	0x18	Polygons are stored a list of triangles (CDT)
vector3DPolygons	0x19	This implies a 16-bit Z value per point
vector3DPolygons32	0x1A	This implies a 32-bit Z value per point
vectorContours	0x1C	Polygons are stored as contours
vector3DContours	0x1D	This implies a 16-bit Z value per point
vector3DContours32	0x1E	This implies a 32-bit Z value per point
vectorTopoContours	0x20	Polygons are stored as contours with shared segments
vector3DTopoContours	0x21	This implies a 16-bit Z value per point
vector3DTopoContours32	0x22	This implies a 32-bit Z value per point
Imagery types		
rasterARGB	0x30	Alpha, Red, Green, Blue (the alpha in high order bit). 4 bytes per pixel (262,144 bytes per 256x256 tile).
raster16Bit	0x31	signed 16-bit integer (131,072 bytes per 256x256 tile)
raster8Bit	0x32	1 unsigned byte per pixel (65,536 per 256x256 tile)
Gridded coverage types		
coverage8Bit	0x50	unsigned, 1 byte per pixel (67,081 total)
coverage16Bit	0x51	signed, 2 bytes per pixel (134,162 total)
coverageInt32	0x52	signed, 4 bytes per pixel (268,324 total)
coverageFloat32	0x53	floating-point, 4 bytes per pixel (536,648 total)
coverageDouble64	0x54	floating-point, 8 bytes per pixel (1,073,296 total)
coverageQuantized16	0x70	- 2x 64-bit double to specify range (min, max) -- Paeth filter, image encoding e.g. PNG does not apply to range - signed 16-bit integer per pixel (134,178 bytes per tile) representing grid values quantized to the min-max range 0 represents (min+max)/2.
3D environment types		
The geometry of pointCloud and models3D(Ground) is still technically vector/points:		
pointCloud	0x90	For e.g. LAS files whereas ESRI Shapefiles pointZ/ S57 Sounding Points will be vector3DPoints
models3D(Ground) reference or embed 3D models (e.g. E3D, glTF, COLLADA, FLT, 3DS, OBJ...)		
models3D	0xA0	This references external 3D models -- 16-bit altitude per point, 32-bit LevelModelID in point data
models3DGround	0xA1	This references external 3D models -- dropped to ground (no Z), 32-bit LevelModelID in point data
embedded3DModel	0xB0	This embeds a single 3D model and is purely 3D geometry (treated similarly to vector3DPolygons)

The following ***Flags*** are currently defined:

Flag	**Position** *(from lsb)*	**Description**
All types		
full	0	The tile is full(finer zoom levels within the pyramid need not be defined).
empty	1	The tile is empty (finer zoom levels within the pyramid need not be defined).
Points, Referenced 3D Models & Point Clouds		
pointsDataId	2	Separate ID instead of elements list (more efficient when most points are individual). Duplicates are allowed (quantization can bring to same point).
rgb	3	Color information
alpha	4	Opacity information
All vector types, Point Clouds (for intensity)		
measure	5	Measurement information (e.g. PointM)
measure32	6	Measurement information (32 bit)
3D Polygons and Embedded 3D Model		
texCoords	8	Texture coordinates
normals	9	Normals information
tangents	10	Tangents information
Referenced 3D Models		
yaw	8	Orientation information
yawPitchRoll	9	Separate yaw, pitch, roll orientation information
scale	10	Scaling information
xyzScale	11	Separate x, y, z scaling information
Point clouds *(note: id is used for classification)*		
scanInfo	8	Scan information

The following ***Encodings*** are currently defined:

Encoding	**Value**	**Description**
uncompressed	0x00	The raw data without any special encoding.
deflate	0x01	The data is compressed with deflate (zlib) algorithm.
lzma	0x02	The data is compressed with LZMA.
jpeg2000	0x80	The data is compressed as a JPEG-2000 image.
png	0x81	The data is compressed as a PNG image.
paethLZMA	0x82	A ***Paeth filter**** is applied; then the data is compresed with LZMA.

Compact Vector Tiles representation

Compact storage as localized vertices with accuracy proportional to scale

- Coordinates are specified as two 16-bit signed integer per vertex, the first integer representing the latitude, and the second the longitude — like ISO 6709:1983. The full range (-32,767..32,767) of these integers are linearly mapped to the geospatial extent of the tile.
- Preserving proper topology with varying accuracy was a major challenge which has been solved in the GNOSIS vector pipeline.
- All points used by the tile are specified in one single array.

Pre-triangulated for high performance GPU rendering and optimal service-to-display processing

- Polygons are described as triangles since tessellation is a required step for hardware accelerated rendering of polygons which are either concave or feature inner holes. The tessellation process can add to the initial loading/processing time before incoming geometry can be visualized on the screen, and therefore this delay is minimized.
- Constrained Delaunay Triangulation is performed to produce an optimal tessellation which maximizes the fill rate.

Enforced topologically correct representation (shared vertex indices)

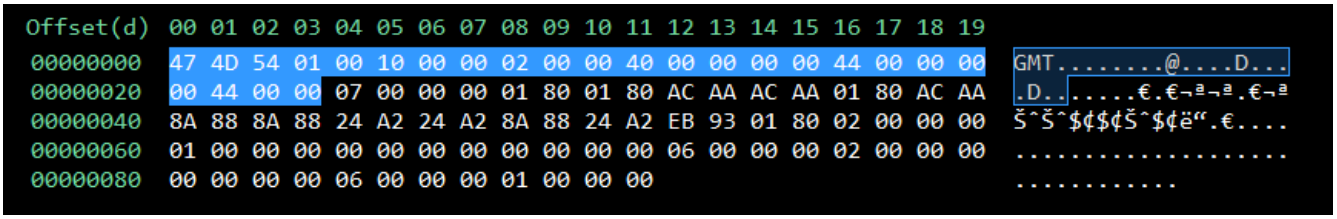
- Lines and polygons provide a list of 16-bit indices into the array of vertices to be re-used by multiple elements sharing the same edges, or by multiple pieces of the same element connecting. This ensures proper topology as common edges and the spatial relationship between different elements are preserved, and makes the representation suitable for both high performance visualization as well as analysis.
- For lines, the indices of one single element make up a single line string
- For polygons, the indices of one single element make up a list of triangles (3 indices per triangle).
- The polygon indices making up triangles are always specified in a **counter-clockwise** manner.

Elements listing indices making up a given feature uniquely identified by a 64-bit ID

Elements are specified by the ID, the start index (in the list of indices for lines and polygons; in the list of points for points) and the count of indices/points used.

Center lines for curved area labels

Since computing the center lines of curved polygons is better done in regard to the overall shapes before tiling occurs, this information can optionally be included together with polygon geometry. This is useful for example to render labels following the curve of those areas, such as typically seen on lakes and large rivers.

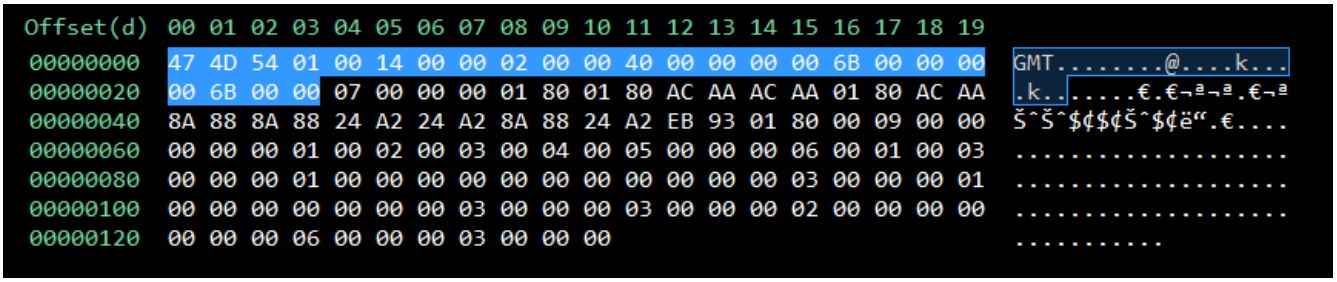


Binary layout for Points, Point Clouds and Positioned & Oriented 3D Models tiles

Offset	Type	Size	Name	Description
0	int	4	numPoints	The number of vertices in the tile
Vertices *(numPoints occurrences)*				
4+n*12	int64	8	id	*(If pointsDataId flag is set)* ID identifying the feature the following point is part of (in the data store's geometry table).
pointsDataId set: 4+n*12 *not set:* 4+n*4	int16	2	latitude	Latitude mapped from the tile's latitude extent to -32,767 to 32,767, with the bottom (south) edge being at -32,767
pointsDataId set: 6+n*12 *not set:* 6+n*4	int16	2	longitude	Longitude mapped from the tile's longitude extent to -32,767 to 32,767, with the left (west) edge being at -32,767
(end of vertices data)				
(If pointsDataId flag is not set)				
4+numPoints*4	int	4	numElements	The number of elements in the tile
Elements *(numElements occurrences)*				
8+numPoints*4 +n*16	int64	8	id	ID identifying the feature the points within this element are part of (in the data store's geometry table).
16+numPoints*4 +n*16	int	4	start	Index to the first vertices for this element
20+numPoints*4 +n*16	int	4	count	Number of consecutive vertices making up element

Offset	Type	Size	Name	Description
If altitude data is available				
	double	8	loAlt	Minimum altitude.
	double	8	hiAlt	Maximum altitude.
	int16 / int32	(2 or 4) * numPoints	altitudes	*min..max as -32767..32767 (32 bit if vector3DPoints32)*
If measurement data is available				
	double	8	loMeasure	Minimum measurement.
	double	8	hiMeasure	Maximum measurement.
	int16 / int32	(2 or 4) * numPoints	measures	*min..max as -32767..32767 (32 bit if measure32 flag set)*
If color and/or alpha information is available (repeats numPoints times)				
	byte	1	alpha	Opacity value *(if alpha flag is set)*
	byte	3	r, g, b	Red, Green and Blue *(if color flag is set)*

Offset	Type	Size	Name	Description
If model flag is set, this points layer references 3D models				
	uint32	4 * numPoints	modelIDs	High 5 bits: model level Low 27 bits: model
If model flag is set and orientation information is available (repeats numPoints times)				
	uint16	2	yaw	-32767(-360°)..32767 (360°)
	uint16	2	pitch	*(if separate ypr flags is set)*
	uint16	2	roll	*(if separate ypr flags is set)*
If model flag is set and scaling information is available (repeats numPoints times)				
	int16	2	sx or scale	*(mul'ed by 256, e.g. 512=2x)*
	int16	2	sy	*(if xyz scaling flag is set)*
	int16	2	sz	*(if xyz scaling flag is set)*
If model flag is set				
	double	8 * 6	extent	Overall extent of referenced models (not including models from other tiles spilling into this one) loLat, loLon, loAlt, hiLat, hiLon, hiAlt
	int	4	numSpillTiles	The number of tiles whose models extend over this tile
For each tile whose models spill onto this tile:				
	uint64	8	spillTileKey	
	*	*	spillData	All information for those points whose models spill onto this tile (i.e. all fields from 'numPoints' to 'extent' inclusively).
If point cloud flag is set and scan information is available				
	uint16	2 * numPoints	scanInfo	*(from lsb to msb)* returnNumber (3 bits) numberOfReturns (3 bits) scanDirection (1 bit) edgeOfFlight (1 bit) angle (8 bit)

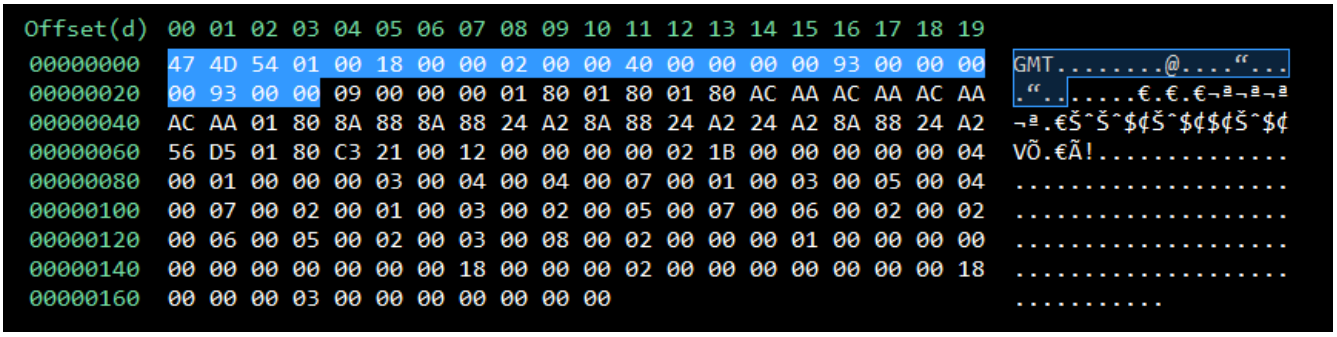


Binary layout for Lines tiles

Offset	Type	Size	Name	Description
0	int	4	numPoints	The number of vertices in the tile
Vertices *(numPoints occurrences)*				
4+n*4	int16	2	latitude	Latitude mapped from the tile's latitude extent to -32,767 to 32,767, with the bottom (south) edge being at -32,767
6+n*4	int16	2	longitude	Longitude mapped from the tile's longitude extent to -32,767 to 32,767, with the left (west) edge being at -32,767
(end of vertices data)				
4+numPoints*4	uint8	(numPoints+7) /8	flags	A compact bits array of flags (1 bit per vertex) set to 1 if the vertex is artificial (i.e not present in source data). The least significant bit represents the first of the up to 8 vertices mapped to each byte of flags.
4+numPoints*4 +(numPoints+7)/8	int	4	numIndices	The number of indices in the tile
8+numPoints*4 +(numPoints+7)/8	uint16	numIndices * 2	indices	16-bit indices into the vertex table to be referenced by elements
8+numPoints*4 +(numPoints+7)/8 +numIndices*2	int	4	numElements	The number of elements in the tile

Offset	Type	Size	Name	Description
Elements *(numElements occurrences)* Each element defines a line string as a series of indices.				
12+numPoints*4 +(numPoints+7)/8 +numIndices*2 +n*16	int64	8	id	ID identifying the feature the lines within this element are part of (in the data store's geometry table).
20+numPoints*4 +(numPoints+7)/8 +numIndices*2 +n*16	int	4	start	Index to the first index making up the lines for this element
24+numPoints*4 +(numPoints+7)/8 +numIndices*2 +n*16	int	4	count	Number of consecutive indices making up the lines for this element
12+numPoints*4 +(numPoints+7)/8 +numIndices*2 +numElements*16	**Total Size**			

NOTE: This table does not yet describe the layout for altitude and measurement values.



Binary layout for Polygons tiles

Offset	Type	Size	Name	Description
0	int	4	numPoints	The number of vertices in the tile
Vertices *(numPoints occurrences)*				
4+n*4	int16	2	latitude	Latitude mapped from the tile's latitude extent to -32,767 to 32,767, with the bottom (south) edge being at -32,767
6+n*4	int16	2	longitude	Longitude mapped from the tile's longitude extent to -32,767 to 32,767, with the left (west) edge being at -32,767
(end of vertices data)				
Polygon Vertex Flags *(numPoints occurrences)* Each vertex has an associated flag indicating whether it lies on the tile boundary and whether edges stemming from it were in the source data (see section explaining vertex flags).				
4+numPoints*4+n (& 0x01)	bit	single bit	onBottomEdge	Set if this vertex lies on the **bottom** tile boundary
4+numPoints*4+n (& 0x02)	bit	single bit	onLeftEdge	Set if this vertex lies on the **left** tile boundary
4+numPoints*4+n (& 0x04)	bit	single bit	onTopEdge	Set if this vertex lies on the **top** tile boundary
4+numPoints*4+n (& 0x08)	bit	single bit	onRightEdge	Set if this vertex lies on the **right** tile boundary
4+numPoints*4+n (& 0x10)	bit	single bit	downIn	Set if an edge from this vertex going **down**originates from source data
4+numPoints*4+n (& 0x20)	bit	single bit	leftIn	Set if an edge from this vertex going **left** originates from source data

Offset	Type	Size	Name	Description
4+numPoints*4+n (& 0x40)	bit	single bit	upIn	Set if an edge from this vertex going **up** originates from source data
4+numPoints*4+n (& 0x80)	bit	single bit	rightIn	Set if an edge from this vertex going **right** originates from source data
(end of vertex flags)				
4+numPoints*5	int	4	numIndices	The number of indices in the tile
8+numPoints*5	uint16	numIndices*2	indices	16-bit indices into the vertex table to be referenced by elements
8+numPoints*5 +numIndices*2	int	4	numElements	The number of elements in the tile
Elements *(numElements occurrences)* Elements define polygons as a series of triplets of indices, defining counter-clockwise triangles.				
12+numPoints*5 +numIndices*2 +n*16	int64	8	id	ID identifying the feature the polygons within this element are part of (in the data store's geometry table).
20+numPoints*5 +numIndices*2 +n*16	int	4	start	Index to the first index making up the polygons for this element
24+numPoints*5 +numIndices*2 +n*16	int	4	count	Number of consecutive indices making up the polygons for this element
(end of elements data)				

Offset	Type	Size	Name	Description
28+numPoints*5 +numIndices*2 +numElements*16	int	4	numCLVertices	The number of vertices describing all centerlines.
Center Lines Vertices				
28+numPoints*5 +numIndices*2 +numElements*16 +n*4	int16	2	latitude	Latitude mapped from the tile's latitude extent to -32,767 to 32,767, with the bottom (south) edge being at -32,767
30+numPoints*5 +numIndices*2 +numElements*16 +n*4	int16	2	longitude	Longitude mapped from the tile's longitude extent to -32,767 to 32,767, with the left (west) edge being at -32,767
(end of center lines vertices)				
30+numPoints*5 +numIndices*2 +numElements*16 +numCLVertices*4	int	4	numCenterLines	The number of center lines defined for the tile. 0 if center lines are not defined.
Center Lines *(numCenterLines occurrences)* Each center line defines a line string as a series of vertices.				
34+numPoints*5 +numIndices*2 +numElements*16 +numCLVertices*4 +n*16	int64	8	id	ID identifying the feature for which a center line is being defined (in the data store's geometry table).
42+numPoints*5 +numIndices*2 +numElements*16 +numCLVertices*4 +n*16	int	4	start	Index to the first vertex making up this center line
46+numPoints*5 +numIndices*2 +numElements*16 +numCLVertices*4 +n*16	int	4	count	Number of consecutive vertices making up this center line
34+numPoints*5 +numIndices*2 +numElements*16 +numCLVertices*4 +numCenterLines*16	**Total Size**			

NOTE: This table does not yet describe the layout for altitude and measurement values. Contours and Topological Contours representation remain to be defined as well.

Vertex flags for identifying tile boundaries and artificial edges

- In order to avoid rendering unwanted edges at the tile boundaries of polygons, flags are marked at each vertex.
- This feature is also used to avoid similar edges problems at the dateline with global datasets
- Each vertex actually has two set of flags, represented in the Tiles API by the PolygonVertexFlags class.
- The first set of flags indicates whether a vertex is on any of a tile's 4 boundaries (top, left, bottom, right). These flags are also useful for recombining tiles, by identifying vertices at a tile's border. If an edge links two vertices flagged as being on the same edge, it is deemed to be an artificial edge, unless explicitly marked as being an actual edge by the second set of flags.
- The other direction flags as they are named in the Tiles API indicate whether there is actually a real edge (i.e. a segment of a polygon contour not introduced by tiling or by wrapping around the dateline) leaving from the flagged vertex going into each 4 directions (up, left, down, right). These flags should only set or inspected in relation to the corresponding set of edge flags:
    - For on the right edge and on the left edge flags, the up and/or down edge is not artificial flags can be set.
    - For on the top edge and on the bottom edge flags, the left and/or right edge is not artificial flags can be set.
- The PolygonVertexFlags provides a simple draw() method to determine whether an edge from one point to another should be drawn or not. The ordering of the vertices matter: the method should be called with a point counter-clockwise to the object on which it is invoked. This is because the flags mark whether an actual edge from the source data passed through each vertex coming from a certain direction.
- A sample implementation of a PolygonVertexFlags class follows:

```
public class PolygonVertexFlags : byte
{
public:
    EdgeFlags onEdge:4;
    DirFlags d:4;
    bool draw(PolygonVertexFlags b)
    {
        bool drawEdge = true;
        EdgeFlags cf = onEdge & b.onEdge;
        if(cf && (
            (cf.right  && (!d.upIn    && !b.d.downIn )) ||
            (cf.top    && (!d.leftIn  && !b.d.rightIn)) ||
            (cf.left   && (!d.downIn  && !b.d.upIn   )) ||
            (cf.bottom && (!d.rightIn && !b.d.leftIn ))))
            drawEdge = false;
        return drawEdge;
    }
};
```

NOTE	When encoded using an image compression format such as PNG or JPEG-2000, which already defines a way to encode the image and dimensions, only the geospatial mapping and geometry of the image is relevant from what is described below.

Binary layout for Imagery tiles

Offset	Type	Size	Name	Description
0	uint16	2	width	Width of the data (typically 256)
2	uint16	2	height	Height of the data (typically 256)
4	*(based on format)*	width * height * sizeof(type) (typically 262,144)	data	The first pixel has its upper-left corner at the upper-left (north-west) corner of the tile, and the next pixels fill a scanline to the East. The next scanline is south of the first one, and so on. Each pixel represents a color for the entire pixel sampled from the center or average, with the 256 x 256 squares to be entirely within the tiles

Binary layout for gridded Coverage tiles

Offset	Type	Size	Name	Description
0	uint16	2	width	Width of the data (typically 259)
2	uint16	2	height	Height of the data (typically 259)
4	*(based on format)*	width * height * sizeof(type) (typically 67,081)	data	The first value reflects a sample 1/256th of the tile's latitude difference (height) and longitude difference (width) away towards the north-west direction from the upper-left (north-west) corner. The next values fill a scanline to the East, going 1/256th past the tile to the East, for a total of 259 samples across. The next scanline is south of the first one, and so on for a total of 259 scanlines, with the last scanline 1/256th of the tile's latitude difference south of the bottom (south) edge. The value are expected to be sampled at exact location (e.g. at the corners of the imagery 'pixels'). The values in different cells for the same geospatial location (e.g. on the tile boundary, as well as for the 1 value buffer around each tile) should match exactly, and facilitate dealing with partial data during visualization or analysis (e.g. to dynamically create a 3D terrain mesh from elevation grids).

For coverages, NODATA values are encoded as -32,767.

*** Paeth filtering**

Imagery and coverages can be filtered using a Paeth filter before being compressed with LZMA. For 16 bit rasters such as the *coverageQuantized16* format used for encoding terrain elevation data, this achieves significantly better compression than the Paeth filter within the PNG format because it treats the 16 bit integers as a whole rather than as individual bytes. It also seems to compress ARGB rasters better than PNG.

The following is an eC reference implementation for the Paeth filter encoding:

```
// a: left, b: above, c: upper left
static inline int paethPredictor(int a, int b, int c)
{
   int p = a + b - c;
   int pa = Abs(p - a);
   int pb = Abs(p - b);
   int pc = Abs(p - c);

   return pa <= pb && pa <= pc ? a : pb <= pc ? b : c;
}

static inline uint16 intToUint16(int x)
{
   x = (short)(uint16)(((uint) x) & 65535);
   return x < 0 ? ((-x-1)*2 + 1) & 65535 : (x * 2) & 65535;
}

static inline int uint16ToInt(uint16 x)
{
   return (x & 1) ? -((int)x-1)/2-1 : (int)x/2;
}

static inline byte intToByte(int x)
{
   return (byte)(((uint) x) & 255);
}

static inline int byteToInt(uint16 x)
{
   return (byte)(((uint) x) & 255);
}
```

```
static void encodePaeth(void * src, uint16 width, uint16 height, Format format)
{
   int x, y;

   if(format == raster16)
   {
      short * in = src;
      uint16 * temp = new uint16[width * height];
      uint16 * out = temp;

      for(y = 0; y < height; y++)
      {
         short a = 0, c = 0;

         for(x = 0; x < width; x++, out++, in++)
         {
            short d = *in, b = (y > 0) ? in[-width] : 0;
            int r = (int)d - paethPredictor(a, b, c);

            *out = intToUint16(r);
            a = d, c = b;
         }
      }
      memcpy(src, temp, width*height*2);
      delete temp;
   }
   else if(format == rasterARGB)
   {
      char * in = src;
      byte * temp = new byte[width * height * 4];
      byte * out = temp;

      for(y = 0; y < height; y++)
      {
         byte a = 0, c = 0;
         for(x = 0; x < width*4; x++, out++, in++)
         {
            byte d = *in, b = (y > 0) ? in[-width*4] : 0;
            int r = (int)d - paethPredictor(a, b, c);

            *out = intToByte(r);
            // NOTE: These are the values for the next iteration
            if(x >= 3) { a = in[-3]; if(y > 0) c = in[-width*4-3]; }
         }
      }
      memcpy(src, temp, (int)width*height*4);
      delete temp;
   }
}
```

```
static void decodePaeth(void * src,  uint16 width, uint16 height, Format format)
{
   int x, y;

   if(format == raster16)
   {
      uint16 * in = src;
      short * temp = new short[width * height];
      short * out = temp;

      for(y = 0; y < height; y++)
      {
         short a = 0, c = 0;

         for(x = 0; x < width; x++, out++, in++)
         {
            short b = (y > 0) ? out[-width] : 0;
            int r = (int)uint16ToInt(*in) + paethPredictor(a, b, c);
            short d = (short)r;

            *out = d;
            a = d, c = b;
         }
      }
      memcpy(src, temp, width*height*2);
      delete temp;
   }
   else if(format == rasterARGB)
   {
      byte * in = src;
      byte * temp = new byte[width * height * 4];
      byte * out = temp;

      for(y = 0; y < height; y++)
      {
         byte a = 0, c = 0;
         for(x = 0; x < width*4; x++, out++, in++)
         {
            byte b = (y > 0) ? out[-width*4] : 0;
            int r = (int)byteToInt(*in) + paethPredictor(a, b, c);
            byte d = (byte)(char)r;

            *out = d;
            // NOTE: These are the values for the next iteration
            if(x >= 3) { a = out[-3]; if(y > 0) c = out[-width*4-3]; }
         }
      }
      memcpy(src, temp, (int)width*height*4);
      delete temp;
   }
}
```