BGL File Format

From FSDeveloper Wiki

This page is a work-in-progress.
Generic message - Please note some detail may possibly be missing or incorrect.

Applicable
LM P3D 2
LM P3D
MS Flight
FSXA
FSX
FS2004
FS2002

Contents

- 1 Introduction
- 2 BGL Common Format
  - 2.1 File Header
    - 2.1.1 Example
  - 2.2 Sections
    - 2.2.1 Example
  - 2.3 Subsections
    - 2.3.1 Example
- 3 AIRPORT
  - 3.1 Name
  - 3.2 Included Tower Scenery Object
  - 3.3 Runway
    - 3.3.1 OffsetThreshold
    - 3.3.2 BlastPad
    - 3.3.3 Overrun
    - 3.3.4 Vasi
    - 3.3.5 Approach Lights
  - 3.4 Helipad
  - 3.5 Start
  - 3.6 Com
  - 3.7 Delete Airport
    - 3.7.1 for Runways
    - 3.7.2 for Starts
    - 3.7.3 for Frequencies
  - 3.8 Apron
  - 3.9 Apron Edge Lights
  - 3.10 Taxiway Point
  - 3.11 Taxiway Parking
  - 3.12 Taxiway Path
  - 3.13 TaxiName
  - 3.14 Taxiway Sign
  - 3.15 Jetway
  - 3.16 Approach
    - 3.16.1 ApproachLegs
    - 3.16.2 MissedApproachLegs
    - 3.16.3 Transition
    - 3.16.4 TransitionLegs
  - 3.17 WayPoint
  - 3.18 Fences
  - 3.19 Unknown Record 0x3B
- 4 AIRPORTSUMMARY
- 5 ILS/VOR
  - 5.1 Localizer
  - 5.2 GlideSlope
  - 5.3 Dme
  - 5.4 Name
- 6 TACAN
- 7 NDB
- 8 SCENERYOBJECT
  - 8.1 TaxiwaySign
  - 8.2 LibraryObject
    - 8.2.1 AttachedObject
  - 8.3 Effect
  - 8.4 GenericBuilding
  - 8.5 Windsock
  - 8.6 Extrusion Bridge
  - 8.7 Trigger
- 9 MARKER
- 10 BOUNDARY
- 11 GEOPOL
- 12 MODEL DATA

- 13 EXCLUSIONRECTANGLE
- 14 NAMELIST
  - 14.1 Example
- 15 TERRAIN_VECTOR_DB
  - 15.1 Subsection Header
    - 15.1.1 Attributes Buffer Example
  - 15.2 Entity Structure
  - 15.3 Segment Structure
    - 15.3.1 Method 1
    - 15.3.2 Method 2
      - 15.3.2.1 Algorithm
    - 15.3.3 Example
- 16 TERRAIN SECTIONS
  - 16.1 SubSection with TRQ1 Records
    - 16.1.1 TRQ1 Record
      - 16.1.1.1 Delta Compressed Segment
      - 16.1.1.2 LZ1 (LZ77) Compressed Segment
      - 16.1.1.3 LZ2 (LZ78) Compressed Segment
      - 16.1.1.4 Bitpack Compressed Segment
    - 16.1.2 RCS1 Record
- 17 INDEXES
- 18 Annexe A
  - 18.1 Computing the bounding coordinates from a DWORD value
  - 18.2 Computing Longitude and Latitude from a DWORD value
  - 18.3 Pitch, bank and heading
  - 18.4 ICAO Identifiers and region codes
  - 18.5 Computing QMID u and v based on level and coordinates
  - 18.6 Getting DWORD values from QMID
  - 18.7 Getting QMID from DWORD values
  - 18.8 How are the header QMIDs computed?
  - 18.9 Delta Decompression Algorithm
  - 18.10 LZ1 (LZ77) Decompression Algorithm
  - 18.11 LZ2 (LZ78) Decompression Algorithm
  - 18.12 Bitpack Decompression Algorithm

Introduction

The information contained in this wiki comes from different sources:

- "FSX File structure" by **Winfried Orthmann**

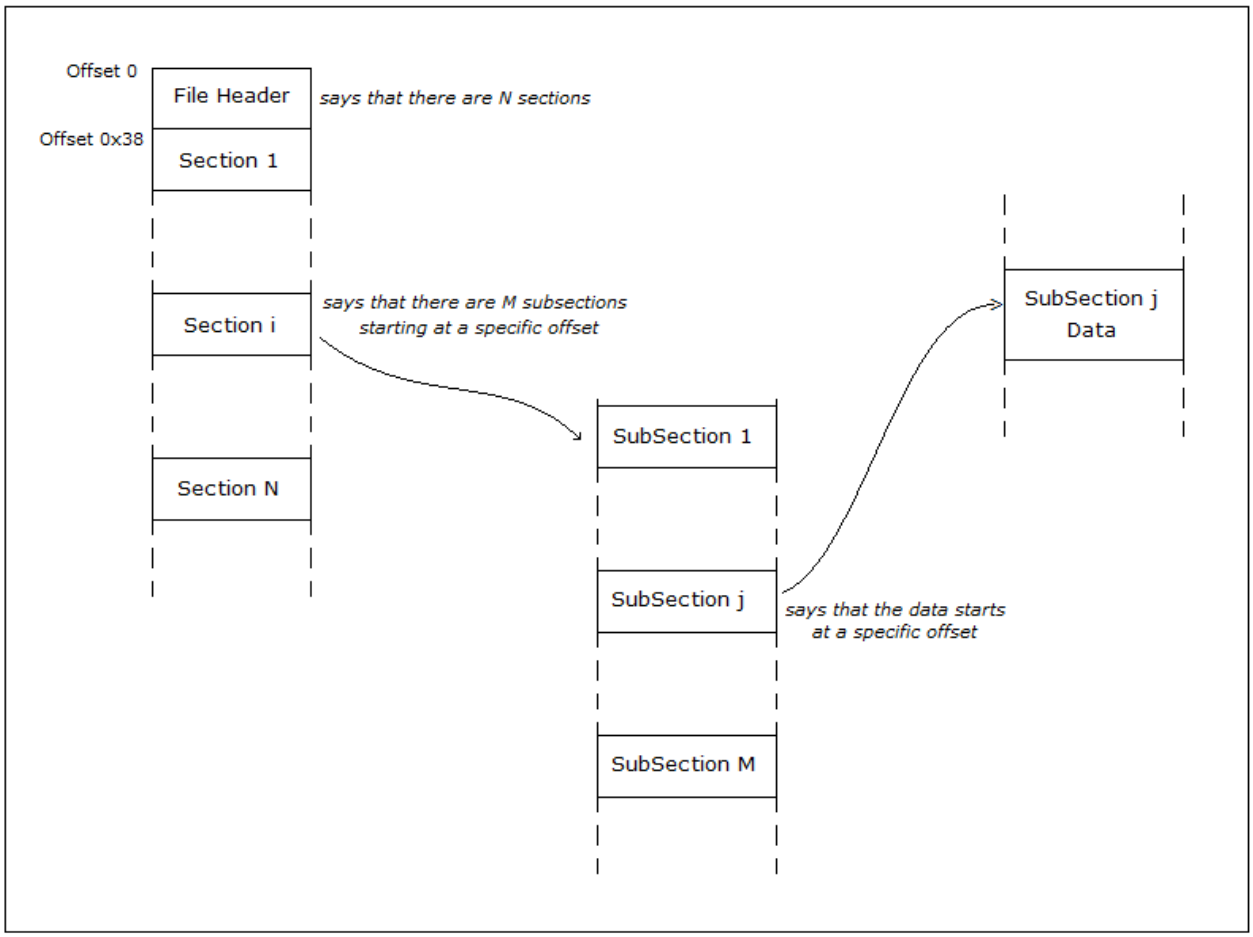
  This is the first document about BGL file format. It describes the generic format of a BGL file and the content of some important sections like Airport, Scenery Objects,etc.

- "FS9 FSX_BGL_Format.doc" by **Jon Masterson** (a.k.a. ScruffyDuck (http://www.fsdeveloper.com/forum/members/scruffyduck.851))
- Some reverse engineering work on the TmfViewer and BglComp applications by **Patrick Germain** (http://www.fsdeveloper.com/forum/members/patrick-germain.14030)

BGL Common Format

All BGL files share the same generic format. A BGL file is made of a header, sections, subsections and subsection data. Subsections are children of sections. The sections are here to help us locate the subsections in the file. Data specific information is contained in the subsections data and their format is dependent on the section type.
A BGL file is really a big container where all kind of information can be stored (in the subsection).The meaning of the data contained in the subsections is known only by the application using it. Th only contraint is for the file to comply to the generic format described below.

A BGL file always start with a header (Size = 0x38 bytes), followed by a list of section pointers. The number of section pointers is defined in the header.



File Header

The header consists of 0x38 (56) bytes. It contains the number of sections defined in the file as well as the bounding geographical coordinates of the covered squared area.

Offset	Number of bytes	Description
0x00	4 - DWORD	Magic Number #1 – Must be 0x01, 0x02, 0x92, 0x19 (all FS versions from FS9 to P3Dv4)
0x04	4 - DWORD	Header size : 0x38
0x08	4 - DWORD	dwLowDateTime of the FILETIME structure. Date and Time the file was created The FILETIME structure represents the number of 100-nanosecond intervals since January 1, 1601 See Working with the FILETIME Structure (http://support.microsoft.com/kb/188768).
0x0C	4 - DWORD	dwHighDateTime of the FILETIME structure
0x10	4 - DWORD	Magic Number #2 – Must be 0x03, 0x18, 0x05, 0x08 (all FS versions from FS9 to P3Dv4)
0x14	4 - DWORD	The number of sections following this header.
0x18	32	Array[8] of unsigned integers (DWORD). Each value derives from a QMID(l,u,v). See the algorithm in Annexe A to retrieve l,u,v values from these dwords. In that case, the second DWORD needed by the algorithm is zero. The QMIDs (up to 8) define the area covered in this BGL file. Even if 8 slots are provided, it is not necessary to have all 8 values filled. The list stops at the first null (0x00000000) value. You can also get the upper-left corner and lower-right corner coordinates of each QMID using Computing the bounding coordinates from a DWORD value. To get the bounding coordinates of the area covered by the file, just keep the minimal and maximal values from each bounding coordinates. See also How are the header QMIDs computed? in Annexe A.

Example

For example, in CVX2815.bgl :

Offset	Values	Description
0x00	01 02 92 19	Magic Number #1
0x04	38 00 00 00	Header size
0x08	EF 82 DF E2	Low = 3806298863
0x0C	E8 C7 C6 01	High = 29804520 => February 27, 2007
0x10	03 18 15 08	Magic Number #2
0x14	01 00 00 00	1 section following this header
0x18	E8 07 02 00	MinLatitude(Deg) = 46.40625 MaxLatitude(Deg) = 47.8125 MinLongitude(Deg) = -75.0 MaxLongitude(Deg) = -73.125 QMID (u=56, v=30, l=8) , using 0x000207E8 as A and 0 as B in this algorithm.
0x1C	E9 07 02 00	MinLatitude(Deg) = 46.40625 MaxLatitude(Deg) = 47.8125 MinLongitude(Deg) = - 73.125 MaxLongitude(Deg) = -71.25 QMID (u=57, v=30, l=8)
0x20	EA 07 02 00	MinLatitude(Deg) = 45.0 MaxLatitude(Deg) = 46.40625 MinLongitude(Deg) = -75.0 MaxLongitude(Deg) = -73.125 QMID (u=56, v=31, l=8)
0x24	EB 07 02 00	MinLatitude(Deg) = 45.0 MaxLatitude(Deg) = 46.40625 MinLongitude(Deg) = -73.124 MaxLongitude(Deg) = -71.25 QMID (u=57, v=31, l=8)
0x28	00 00 00 00	
0x2C	00 00 00 00	
0x30	00 00 00 00	
0x34	00 00 00 00	

You'll notice that only 4 subareas are defined (on a possibility of 8) and the last 4 available slots are empty (Value = 0)
So the bounding coordinates of the area covered by cvx2815.bgl are:

MinLatitude(Deg) = 45.0

MaxLatitude(Deg) = 47.8125
MinLongitude(Deg) = -75.0
MaxLongitude(Deg) = -71.25

Sections

Following the header, at offset 0x38, there are as many sections as defined at offset 0x14 of the header. A same file may contain sections of different type. Each section has a size of 20 bytes and has the following structure:

Relative Offset	Number of bytes	Description
0x00	4 - DWORD	Section type (as defined by Microsoft Flight Simulator): one of the following values: • None = 0x0 • Copyright = 0x1 • Guid = 0x2 • Airport = 0x3 • Ils/Vor = 0x13 • Ndb = 0x17 • Marker = 0x18 • Boundary = 0x20 • Waypoint = 0x22 • Geopol = 0x23 • SceneryObject = 0x25 • NameList = 0x27 • VorIlsIcaoIndex = 0x28 • NdbIcaoIndex = 0x29 • WaypointIcaoIndex = 0x2A • ModelData = 02B • AirportSummary = 0x2C • Exclusion = 0x2E • TimeZone = 0x2F • TerrainVectorDb = 0x65 • TerrainElevation = 0x67 • TerrainLandClass = 0x68 • TerrainWaterClass = 0x69 • TerrainRegion = 0x6A • PopulationDensity = 0x6C • AutogenAnnotation = 0x6D • TerrainIndex = 0x6E • TerrainTextureLookup = 0x6F • TerrainSeasonJan = 0x78 • TerrainSeasonFeb = 0x79 • TerrainSeasonMar = 0x7A • TerrainSeasonApr = 0x7B • TerrainSeasonMay = 0x7C • TerrainSeasonJun = 0x7D • TerrainSeasonJul = 0x7E • TerrainSeasonAug = 0x7F • TerrainSeasonSep = 0x80 • TerrainSeasonOct = 0x81 • TerrainSeasonNov = 0x82 • TerrainSeasonDec = 0x83 • TerrainPhotoJan = 0x8C • TerrainPhotoFeb = 0x8D • TerrainPhotoMar = 0x8E • TerrainPhotoApr = 0x8F • TerrainPhotoMay = 0x90 • TerrainPhotoJun = 0x91 • TerrainPhotoJul = 0x92 • TerrainPhotoAug = 0x93 • TerrainPhotoSep = 0x94 • TerrainPhotoOct = 0x95 • TerrainPhotoNov = 0x96 • TerrainPhotoDec = 0x97 • TerrainPhotoNight = 0x98 • Tacan = 0xA0 • TacanIndex = 0xA1 • FakeTypes = 0x2710 • IcaoRunway = 0x2711
0x04	4 - DWORD	Value used to compute the size of each subsection. `subSection Size = ((value & 0x10000) | 0x40000) >> 0x0E` Most subsections have a size of 16 bytes. However, subsections for TerrainSeasonXXXX have a size of 20 bytes.
0x08	4 - DWORD	Number of subsections in the section.
0x0C	4 - DWORD	File offset = position in the file where the first subsection starts.
0x10	4 - DWORD	Total Size (in bytes) of all the subsections. This value should be equal to : nbSubSections x (size computed at offset 0x04)

Example

For example, in CVX2815.bgl, the only one section, at offset 0x38, has this:

Offset	Values	Description
0x38	`65 00 00 00`	TerrainVectorDb = 101 = 0x65
0x3C	`01 00 00 00`	1 => subsection size = 16
0x40	`8D 07 00 00`	0x078D = 1933 subsections in the section
0x44	`01 CD 1F 00`	0x1FCD01 = File offset = position in the file where the first subsection starts.
0x48	`D0 78 00 00`	Size (in bytes) =0x78D0 = 30928 bytes = 16 * 1933

Subsections

A subsection contains information about the geographical area it covers. If also contains the file offset of the subsection's data. All subsections of a same section are contiguous meaning that they are following each other in the file.
A subsection has a size of 16 bytes, except for the following sections that have a size of 20 bytes

- TerrainElevation
- TerrainLandClass
- TerrainWaterClass
- TerrainRegion
- PopulationDensity
- TerrainIndex
- TerrainSeasonXXX
- TerrainPhotoXXX
- TerrainPhotoXXX

The subsection has the following structure:

Relative Offset	Number of bytes	Description
0x00	4 - DWORD	QMID Dword A. See this algorithm to retrieve the QMID values. ***Note that all subsections inside a section are sorted on this value, meaning that subsections with the lowest QMID level (bigger squares) are followed by subsections with higher QMID levels (smaller squares).***
0x04	4 - DWORD	QMID Dword B - This field is present only for subsections having a size of 20 bytes)
	4 - DWORD	Number of records contained in the subsection's data. It is 0 for TerrainVectorDb. For **NameList**, there is only one record. This value is actually the number of ICAO entries defined in the record data.
0x08(16bytes)/0x0C(20bytes)	4 - DWORD	File Offset = Position in the file of the subsection's data
0x0C(16bytes)/0x10(20bytes)	4 - DWORD	Size of the subsection's data

The interpretation of the section data depends on the section type.

Example

In CVX2815.bgl:

Offset	Values	Description
0x1FCD01	00 FA 81 00	Bounding coordinates: ▪ MinLatitude(Deg) = 47.63671875 ▪ MaxLatitude(Deg) = 47.8125 ▪ MinLongitude(Deg) =- 75.0 ▪ MaxLongitude(Deg) = -74.765625
0x1FCD05	00 00 00 00	Number of records = 0
0x1FCD09	4C 00 00 00	0x4C = File Offset = Position in the file of the subsection's data.
0x1FCD0D	DD 00 00 00	Size of the subsection's data = 0xDD = 221 bytes.

AIRPORT

Each airport record consists of a fixed part with the length of 0x38 bytes, followed by a variable part with 0..n subrecords of different types. The structure of the fixed part is as follows

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x003C**)
0x02	4	DWORD	Size of the airport record
0x06	1	BYTE	Number of RUNWAY subrecords
0x07	1	BYTE	Number of COM subrecords
0x08	1	BYTE	Number of START subrecords
0x09	1	BYTE	Number of APPROACH subrecords (?)
0x0A	1	BYTE	Bit 0-6 : Number of aprons (?) Bit 7: flag for deleteAirport record
0x0B	1	BYTE	Number of HELIPAD subrecords
0x0C	4	DWORD	Longitude
0x10	4	DWORD	Latitude
0x14	4	DWORD	Altitude (in meters)
0x18	4	DWORD	Tower Longitude (if present)
0x1C	4	DWORD	Tower Latitude (if present)
0x20	4	DWORD	Tower Altitude (in meters)
0x24	4	FLOAT	Magnetic Variation (Deg)
0x28	4	DWORD	ICAO Ident (Special Format)
0x2C	4	DWORD	Region Ident (Special Format)
0x30	4	DWORD	Type of fuel present at the airport and its availability. Each availability is coded on 2 bits: 0 = No, 1 = Unknown, 2 = Prior Request, 3 = Yes ▪ Bits 0-1 : Availability for 73 octane ▪ Bits 2-3 : Availability for 87 octane ▪ Bits 4-5 : Availability for 100 octane ▪ Bits 6-7 : Availability for 130 octane ▪ Bits 8-9 : Availability for 145 octane ▪ Bits 10-11 : Availability for MOGAS ▪ Bits 12-13 : Availability for JET ▪ Bits 14-15 : Availability for JETA ▪ Bits 16-17 : Availability for JETA1 ▪ Bits 18-19 : Availability for JETAP ▪ Bits 20-21 : Availability for JETB ▪ Bits 22-23 : Availability for JET4 ▪ Bits 24-22 : Availability for JET5 ▪ Bit 30 : Airport has avgas ▪ Bit 31 : Airport has jet fuel
0x34	1	BYTE	Unknown (always 0x00) - FSX only
0x35	1	BYTE	INT(Traffic Scalar * 255) - FSX only
0x36	2	WORD	Unknown (always 0x00) - FSX only

The following subrecords can be present after the main airport record:

Name

This subrecord seems to be present in every airport record, and is usually the first one immediately after the fixed part. However a DELETE subsection may precede the Name subsection.

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x0019**)
0x02	4	DWORD	Size of the Name subrecord
0x06		STRING	AirportName (padded with NUL (0x000 bytes)

Included Tower Scenery Object

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x0066**)
0x02	4	DWORD	Size of the subrecord
0x06	4	DWORD	Size of the included scenery object

After this record we find an included scenery object with an internal structure identical to that of other scenery objects (see below) and including possible attachments. The BglComp compiler allows only one scenery object to be included at this point, but in some FS X scenery files we find more than one objects included here. If present, the subrecords of this type appear immediately after the Name subrecord.

Runway

The runway subrecord consists of a fixed part with a length of 0x34 bytes and a variable number of sub-subrecords. The fixed part has the following structure:

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x0004**)
0x02	4	DWORD	Size of the runway subrecord
0x06	2	WORD	Type of surface: ▪ 0x0000 CONCRETE ▪ 0x0001 GRASS ▪ 0x0002 WATER ▪ 0x0004 ASPHALT ▪ 0x0007 CLAY ▪ 0x0008 SNOW ▪ 0x0009 ICE ▪ 0x000C DIRT ▪ 0x000D CORAL ▪ 0x000E GRAVEL ▪ 0x000F OIL_TREATED ▪ 0x0010 STEEL_MATS ▪ 0x0011 BITUMINOUS ▪ 0x0012 BRICK ▪ 0x0013 MACADAM ▪ 0x0014 PLANKS ▪ 0x0015 SAND ▪ 0x0016 SHALE ▪ 0x0017 TARMAC ▪ 0x00FE UNKNOWN
0x08	1	BYTE	Primary runway number (01 - 36) or ▪ 37 = n ▪ 38 = ne ▪ 39 = e ▪ 40 = se ▪ 41 = s ▪ 42 = sw ▪ 43 = w ▪ 44 = nw
0x09	1	BYTE	Primary runway designator: ▪ 0 = NONE ▪ 1 = LEFT ▪ 2 = RIGHT ▪ 3 = CENTER ▪ 4 = WATER ▪ 5 = A ▪ 6 = B
0x0A	1	BYTE	Secondary runway number
0x0B	1	BYTE	Secondary runway designator
0x0C	4	DWORD	ICAO Ident. for primary ILS (Special Format), 0x0000 if none.
0x10	4	DWORD	ICAO Ident. for seconday ILS (Special Format)
0x14	4	DWORD	Longitude
0x18	4	DWORD	Latitude
0x1C	4	DWORD	Elevation x 1000 (in meters)
0x20	4	FLOAT	Length (in meters)
0x24	4	FLOAT	Width (in meters)
0x28	4	FLOAT	Heading (Deg)
0x2C	4	FLOAT	Pattern Altitude (in meters)
0x30	2	WORD	Marking flags: BIT 0: edges BIT 1: threshold BIT 2: fixedDistance BIT 3: touchdown BIT 4: dashes BIT 5: ident BIT 6: precision BIT 7: edgePavement BIT 8: singleEnd BIT 9: primaryClosed BIT 10: secondaryClosed

			BIT 11: primaryStol BIT 12: secondaryStol BIT 13: alternate threshold BIT 14: alternate fixedDistance BIT 15: alternate touchdown
0x32	1	BYTE	Lights flags: BIT 0-1: edge ▪ 0 = None ▪ 1 = Low ▪ 2 = Medium ▪ 3 = High BIT 2-3: center (as with edge) BIT 4: centerRed flag BIT 5: alternatePrecision BIT 6: leadingZeroIdent BIT 7: noThresholdEndArrows
0x33	1	BYTE	Pattern flags: BIT 0: primaryTakeoff (0 = YES) BIT 1: primaryLanding (0 = YES) BIT 2: primaryPattern (0 = LEFT) BIT 3: secondaryTakeoff BIT 4: secondaryLanding BIT 5: secondaryPattern BIT 6-7: unused (?)

The following sub-subreports can be present within a runway subrecord:

OffsetThreshold

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id Primary (**0x0005**) Id Secondary (**0x0006**)
0x02	4	DWORD	Size of sub-subrecord (0x10)
0x06	2	WORD	Surface (same as in runway)
0x08	4	FLOAT	Length in meters
0x0C	4	FLOAT	Width in meters

BlastPad

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id Primary (**0x0007**) Id Secondary (**0x0008**)
0x02	4	DWORD	Size of sub-subrecord (0x10)
0x06	2	WORD	Surface (same as in runway)
0x08	4	FLOAT	Length in meters
0x0C	4	FLOAT	Width in meters

Overrun

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id Primary (**0x0009**) Id Secondary (**0x000A**)
0x02	4	DWORD	Size of sub-subrecord (0x10)
0x06	2	WORD	Surface (same as in runway)
0x08	4	FLOAT	Length in meters
0x0C	4	FLOAT	Width in meters

Vasi

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id Primary Left (**0x000B**) Id Primary Right (**0x000C**) Id Secondary Left (**0x000D**) Id Secondary Right (**0x000E**)
0x02	4	DWORD	Size of sub-subrecord (0x18)
0x06	2	WORD	Type: ▪ 0x01 = VASI21 ▪ 0x02 = VASI31 ▪ 0x03 = VASI22 ▪ 0x04 = VASI32 ▪ 0x05 = VASI23 ▪ 0x06 = VASI33 ▪ 0x07 = PAPI2 ▪ 0x08 = PAPI4 ▪ 0x09 = TRICOLOR ▪ 0x0a = PVASI ▪ 0x0b = TVASI ▪ 0x0c = BALL ▪ 0x0d = APAP/PANELS
0x08	4	FLOAT	BiasX
0x0C	4	FLOAT	BiasZ
0x10	4	FLOAT	Spacing
0x14	4	FLOAT	Pitch

Approach Lights

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id Primary (**0x000F**) Id Secondary (**0x0010**)
0x02	4	DWORD	Size of sub-subrecord (0x08)
0x06	1	BYTE	Bits [0-5]: System ▪ 0x00 = NONE ▪ 0x01 = ODALS ▪ 0x02 = MALSF ▪ 0x03 = MALSR ▪ 0x04 = SSALF ▪ 0x05 = SSALR ▪ 0x06 = ALSF1 ▪ 0x07 = ALSF2 ▪ 0x08 = RAIL ▪ 0x09 = CALVERT ▪ 0x0a = CALVERT2 ▪ 0x0b = MALS ▪ 0x0c = SALS ▪ 0x0e = SSALS Bit 5: Endlights (0 = false, 1 = true) Bit 6: Reil (0 = false, 1 = true) Bit 7: Touchdown (0 = false, 1 = true)
0x07	1	BYTE	Number of strobes

Helipad

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x0026**)
0x02	4	DWORD	Size of subrecord (0x24)
0x06	1	BYTE	Surface (as in Runway)
0x07	1	BYTE	bit 0-3: Type ▪ 0 = NONE ▪ 1 = H ▪ 2 = SQUARE ▪ 3 = CIRCLE ▪ 4 = MEDICAL bit 4: transparent bit 5: closed bit 6-7: unused
0x08	4	BYTE[4]	color (cannot be set with bglcomp because of error kin compiler)
0x0C	4	DWORD	Longitude
0x10	4	DWORD	Latitude
0x14	4	DWORD	Altitude x 1000 (in meters)
0x18	4	FLOAT	Length
0x1C	4	FLOAT	Width
0x18	4	FLOAT	Heading

Start

(the keywords “Start” and “RunwayStart” produce identical subrecords)

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x0011**)
0x02	4	DWORD	Size of subrecord (0x18)
0x06	1	BYTE	Runway Number
0x07	1	BYTE	bit 0-3: Runway Designator (as with runway subrecord) bit 4-7: Start Type ▪ 1 = RUNWAY ▪ 2 = WATER ▪ 2 = HELIPAD
0x08	4	DWORD	Longitude
0x0C	4	DWORD	Latitude
0x10	4	DWORD	Elevation x 1000 (in meters)
0x14	4	FLOAT	Heading

Com

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x0012**)
0x02	4	DWORD	Size of subrecord: variable
0x06	2	WORD	Type. The following numbers have been identified: ▪ 0x0001 ATIS ▪ 0x0002 MULTICOM ▪ 0x0003 UNICOM ▪ 0x0004 CTAF ▪ 0x0005 GROUND ▪ 0x0006 TOWER ▪ 0x0007 CLEARANCE ▪ 0x0008 APPROACH ▪ 0x0009 DEPARTURE ▪ 0x000A CENTER ▪ 0x000B FSS ▪ 0x000C AWOS ▪ 0x000D ASOS ▪ 0x000E CLEARANCE_PRE_TAXI ▪ 0x000F REMOTE_CLEARANCE_DELIVERY
0x08	4	DWORD	Frequency x 1000000
0x0C	Variable	STRINGZ	Name. Maximum Length = 48 (0x30)

Delete Airport

The DeleteAirport subrecord has a fixed and a variable part. The fixed part has the following structure:

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x0033**)
0x02	4	DWORD	Size of subrecord: variable
0x06	2	WORD	delete flags ▪ BIT 0: allApproaches ▪ BIT 1: allApronLights Note: in the bglcomp.xsd this keyword is written allApronlights, but the compiler accepts only allApronLights. You have to edit bglcomp.xsd, if you want to use this feature. ▪ BIT 2: allAprons ▪ BIT 3: allFrequencies ▪ BIT 4: allHelipads ▪ BIT 5: allRunways ▪ BIT 6: allStarts ▪ BIT 7: allTaxiways
0x08	1	BYTE	Number of individual runways to delete
0x09	1	BYTE	Number of individual starts to delete
0x0A	1	BYTE	Number of individual frequencies to delete
0x0B	1	BYTE	Unused (?)

according to the number of individual features to delete there are the following parts of the record added:

for Runways

Offset	Number of bytes	Format	Description
0x00	1	BYTE	Surface (as in runway subrecord)
0x01	1	BYTE	Runway number primary
0x02	1	BYTE	Runway number secondary
0x03	1	BYTE	bit 0-3: Runway designator primary bit 4-7: Runway designator secondary

for Starts

Offset	Number of bytes	Format	Description
0x00	1	BYTE	Runway number
0x01	1	BYTE	Runway designator
0x02	1	BYTE	Type of start ▪ 1 = RUNWAY, ▪ 2 = WATER, ▪ 3 = HELIPAD
0x03	1	BYTE	Unused (?)

for Frequencies

Offset	Number of bytes	Format	Description
0x00	4	DWORD	bit 28-31: type (as with COM records) bit 0-27: frequency * 1000000

Apron

There are 2 subrecords for each apron which follow each other. Both have variable length.
First record

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x0037**)
0x02	4	DWORD	Size
0x06	1	BYTE	surface (as with runway subrecord)
0x07	2	WORD	Number of vertices
and then for each vertex:			
	4	DWORD	Longitude
	4	DWORD	Latitude
and then			
			zero-fill to next DWORD boundary

Second record

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x0030**)
0x02	4	DWORD	Size
0x06	1	BYTE	surface (as with runway subrecord)
0x07	1	BYTE	flags: ▪ bit 0: drawSurface ▪ bit 1: drawDetail
0x08	2	WORD	Number of vertices
0x0A	2	WORD	Number of triangles to draw
and then for each vertex:			
	4	DWORD	Longitude
	4	DWORD	Latitude
and then for each triangle to draw:			
	2	WORD	Index of first point
	2	WORD	Index of second point
	2	WORD	Index of third point

Apron Edge Lights

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x0031**)
0x02	4	DWORD	Size
0x06	2	WORD	Unknown
0x08	2	WORD	Number of vertices
0x0A	2	WORD	Number of edges
0x0C	4	DWORD	0xFF0000FF: Unknown, probably color of lights
0x10	4	FLOAT	0x3F800000: Unknown (value 1)
0x14	4	FLOAT	0x44480000: Unknown (value 800)
and then for each vertex			
	4	DWORD	Longitude
	4	DWORD	Latitude
and then for each edge			
	4	FLOAT	Unknown (value 60.96)
	2	WORD	Index of start vertex
	2	WORD	Index of end vertex

Taxiway Point

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x001A**)
0x02	4	DWORD	Size: variable
0x06	2	WORD	Number of taxiway points present
and for each taxipoint:			
	1	BYTE	Type ▪ 1 = NORMAL ▪ 2 = HOLD_SHORT ▪ 4 = ILS_HOLD_SHORT
	1	BYTE	Flag ▪ 0 = FORWARD ▪ 1 = REVERSE
	2	WORD	Unknown
	4	DWORD	Longitude
	4	DWORD	Latitude

Taxiway Parking

This record type has a short fixed part for all TaxiwayParking records together and a longer variable part with sections for each TaxiwayParking. The fixed part is 8 bytes long:

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x003D**)
0x02	4	DWORD	Size: variable
0x06	2	WORD	Number of taxiway parking records present

The record sections for each TaxiwayParking are again of variable length, depending on the number of airlineCodes present:

Offset	Number of bytes	Format	Description
0x00	4	DWORD	bit 31-24: count of airlineCodes present bit 23-12: number bit 11-8: type ▪ 0x1 = RAMP_GA ▪ 0x2 = RAMP_GA_SMALL ▪ 0x3 = RAMP_GA_MEDIUM ▪ 0x4 = RAMP_GA_LARGE ▪ 0x5 = RAMP_CARGO ▪ 0x6 = RAMP_MIL_CARGO ▪ 0x7 = RAMP_MIL_COMBAT ▪ 0x8 = GATE_SMALL ▪ 0x9 = GATE_MEDIUM ▪ 0xa = GATE_HEAVY ▪ 0xb = DOCK_GA ▪ 0xc = FUEL ▪ 0xd = VEHICLES bit 7-6: pushback ▪ 00 = none ▪ 01 = left ▪ 10 = right ▪ 11 = both bit 5-0: name ▪ 0x00 = NONE ▪ 0x01 = PARKING ▪ 0x02 = N_PARKING ▪ 0x03 = NE_PARKING ▪ 0x04 = E_PARKING ▪ 0x05 = SE_PARKING ▪ 0x06 = S_PARKING ▪ 0x07 = SW_PARKING ▪ 0x08 = W_PARKING ▪ 0x09 = NW_PARKING ▪ 0x0a = GATE ▪ 0x0b = DOCK ▪ 0x0c = GATE_A ▪ 0x0d = GATE_B ▪ 0x0e = GATE_C ▪ 0x0f = GATE_D ▪ 0x10 = GATE_E ▪ .. ▪ .. ▪ 0x25 = GATE_Z
0x04	4	FLOAT	Radius
0x08	4	FLOAT	Heading
0x0C	4	FLOAT	teeOffset1
0x10	4	FLOAT	teeOffset1
0x14	4	FLOAT	teeOffset3
0x18	4	FLOAT	teeOffset4
0x1C	4	DWORD	Longitude
0x20	4	DWORD	Latitude
...	4	STRING	Airline Designator (0..n times repeated)

Taxiway Path

This record has a fixed length of 8 bytes and a variable part with records for each path. It has the following structure:

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x001C**)
0x02	4	DWORD	Size
0x06	2	WORD	Number of paths defined
and for each path:			
0x00	2	WORD	index of start point. For type TAXI, the index of the start and of the end must both refer to a TaxiPoint. For type PARKING the start index must refer to a TaxiPoint, the end index must refer to a TaxiwayParking. Indexes are zero-based.
0x02	2	WORD	Bit 0-11: index of end point Bit 12-15: runway designator
0x04	1	BYTE	BIT 0-4: Type ▪ 0x1 = TAXI ▪ 0x2 = RUNWAY ▪ 0x3 = PARKING ▪ 0x4 = PATH ▪ 0x5 = CLOSED ▪ 0x6 = VEHICLE BIT 5: "drawSurface flag" (TRUE=1) BIT 6: "drawDetail flag" (TRUE=1) BIT 7: unused (?)
0x05	1	BYTE	runway number / index into TaxiName
0x06	1	BYTE	bitfield BIT 0: centerline BIT 1: centerLineLighted BIT 2-3: leftEdge ▪ 00 = NONE ▪ 01 = SOLID ▪ 10 = DASHED ▪ 11 = SOLID_DASHED BIT 4: leftEdgeLighted BIT 5-6: rightEdge BIT 7: rightEdgeLighted
0x07	1	BYTE	Surface
0x08	4	FLOAT	Width
0x0C	4	FLOAT	Weight Limit
0x10	4	DWORD	??

TaxiName

This record has variable length, it consist of 8 bytes as a fixed part and then 8 bytes for each Name.

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x001D**)
0x02	4	DWORD	Size: variable
0x06	2	WORD	Number of name entries
			and for each name:
		STRING	Taxiname

Taxiway Sign

These records are coded in the section for scenery objects (0x25) with a separate type of entry. Apparently all Taxiway signs for one airport are coded together on one record. There seems to be no coordination of this record with the airport record to which it belongs! I guess that for historical reasons, the TaxiwaySign fixed part looks like the LibraryObject record and that is why the records are stored in the Scenery object section.

Jetway

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x003A**) 0x05 for FS9
0x04	2	WORD	Size: variable
0x06	2	WORD	Parking number (refers to an existing parking)
0x08	2	WORD	Gate Name
0x0A	4	DWORD	Size of the scenery object data to follow (0x40)
0x0E	64		LibraryObject record

Approach

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x0024**)
0x02	4	DWORD	Size: variable
0x06	1	BYTE	Suffix
0x07	1	BYTE	Runway Number
0x08	1	BYTE	bit 0-3: Type ▪ 0x01 = GPS ▪ 0x02 = VOR ▪ 0x03 = NDB ▪ 0x04 = ILS ▪ 0x05 = LOCALIZER ▪ 0x06 = SDF ▪ 0x07 = LDA ▪ 0x08 = VORDME ▪ 0x09 = NDBDME ▪ 0x0a = RNAV ▪ 0x0b = LOCALIZER_BACKCOURSE bit 4-6: Runway designator bit 7: gpsOverlay flag
0x09	1	BYTE	Number of transitions (?)
0x0A	1	BYTE	Number of approach legs
0x0B	1	BYTE	Number of missedApproach legs ?
0x0C	4	DWORD	fixIdent BIT 0-4: fixType ▪ 02 = VOR ▪ 03 = NDB ▪ 04 = TERMINAL_NDB ▪ 05 = WAYPOINT ▪ 06 = TERMINAL_WAYPOINT ▪ 09 = RUNWAY BIT 5-31: fixIdent
0x10	4	DWORD	bit 0-10: fixRegion bit 11-31: ICAO Id of relevant airport
0x14	4	FLOAT	Altitude
0x18	4	FLOAT	Heading
0x1C	4	FLOAT	missedAltitude

after this the following record can occur:

ApproachLegs

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x002D**)
0x02	4	DWORD	Size: variable
0x06	2	WORD	Number of legs to follow

each leg is a structure with a fixed length of 44 bytes

Offset	Number of bytes	Format	Description
0x00	1	BYTE	ID of the leg types found: ▪ 0x01 = AF ▪ 0x02 = CA ▪ 0x03 = CD ▪ 0x04 = CF ▪ 0x05 = CI ▪ 0x06 = CR ▪ 0x07 = DF ▪ 0x08 = FA ▪ 0x09 = FC ▪ 0x0a = FD ▪ 0x0b = FM ▪ 0x0c = HA ▪ 0x0d = HF ▪ 0x0e = HM ▪ 0x0f = IF ▪ 0x10 = PI ▪ 0x11 = RF ▪ 0x12 = TF ▪ 0x13 = VA ▪ 0x14 = VD ▪ 0x15 = VI ▪ 0x16 = VM ▪ 0x17 = VR
0x01	1	BYTE	Altitude Descriptor ▪ 01 = A ▪ 02 = + ▪ 03 = - ▪ 04 = B
0x02	2	WORD	Flags ▪ bit 0: turnDirection = L ▪ bit 1: turnDirection = R ▪ bit 8: magneticCourse (0)* trueCourse (1) ▪ bit 9: distance (0) or time (1) ▪ bit 10: flyover false (0) true (1)
0x04	4	DWORD	bit 5-31: fixIdent bit 0-4: fixType
0x08	4	DWORD	bit 0-10: fixRegion bit 11-32: ICAO Id of relevant airport
0x0C	4	DWORD	bit 5-31: recommendedIdent bit 0-4: recommendedType
0x10	4	DWORD	Recommended region
0x14	4	FLOAT	Theta
0x18	4	FLOAT	Rho
0x1C	4	FLOAT	trueCourse / magneticCourse (depending on flag)
0x20	4	FLOAT	Distance / Time
0x24	4	FLOAT	Altitude 1
0x28	4	FLOAT	Altitude 2

MissedApproachLegs

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x002E**)
0x02	4	DWORD	Size: variable
0x06	2	WORD	Number of legs to follow

Transition

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x002C**)
0x02	4	DWORD	Size: variable
0x06	1	BYTE	transitionType 1 = FULL, 2 = DME
0x07	1	BYTE	Number of TransitionLegs
0x08	4	DWORD	bit 0-4: fixType ▪ 2 = VOR ▪ 3 = NDB ▪ 4 = TERMINAL_NDB ▪ 5 = WAYPOINT ▪ 6 = TERMINAL_WAYPOINT bit 5-31: fixIdent (Special Format)
0x0C	4	DWORD	bit 0-10: fixRegion bit 11-31 : airportID of relevant airport
0x10	4	FLOAT	Altitude
			If transitionType = DME and DmeArc record exists, then the following 16 bytes are present
0x14	4	DWORD	dmeIndent
0x18	4	DWORD	bit 0-10: dmeRegion bit 11-31: airportID of relevant airport
0x1C	4	DWORD	Radial
0x20	4	FLOAT	Distance

TransitionLegs

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x002F**)
0x02	4	DWORD	Size: variable
0x06	2	WORD	Number of legs to follow.

WayPoint

The waypoint record can be part of the Airport group or can be entered independently. In both cases the output for the BGL is the same but for the DWORD at offset 0x18.

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x0022**)
0x02	4	DWORD	Size: variable
0x06	1	BYTE	Type ▪ 1 = NAMED ▪ 2 = UNNAMED ▪ 3 = VOR ▪ 4 = NDB ▪ 5 = OFF_ROUTE ▪ 6 = IAF ▪ 7 = FAF
0x07	1	BYTE	Number of Route entries to follow
0x08	4	DWORD	Longitude
0x0C	4	DWORD	Latitude
0x10	4	FLOAT	Magnetic Variation
0x14	4	DWORD	WayPoint Ident (Special Format)
0x18	4	DWORD	bit 0-10: waypointRegion (Special Format) bit 11-31: ICAO ident of the relevant airport, if it is a terminal waypoint, defined within an airport record.
Optional, if Route is given			
0x1C	1	BYTE	Route type ▪ 1 = VICTOR ▪ 2 = JET ▪ 3 = BOTH
0x1D	8	char[8]	Name (zero padded), name cannot be longer than 8 characters
for Next:			
0x25	4	DWORD	BIT 0-2: Type ▪ 1 = NDB ▪ 2 = VOR ▪ 5 = all other BIT 5-31: waypointIdent (Special Format)
0x29	4	DWORD	Bit 0-10 waypointRegion (Special Format) Bit 11-31 airportId if terminal waypoint
0x2D	4	FLOAT	Altitude Minimum
for Previous:			
0x31	4	DWORD	BIT 0-2: Type ▪ 1 = NDB ▪ 2 = VOR ▪ 5 = all other BIT 5-31: waypointIdent (Special Format)
0x35	4	DWORD	Bit 0-10 waypointRegion (Special Format) Bit 11-31 airportId if terminal waypoint
0x39	4	FLOAT	Altitude Minimum

Note: it is not necessary for any route to have both previous and next defined, in that case the fields for this part of the record are all zero.

Fences

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id **0x0038** BlastFence **0x0039** BoundaryFence
0x02	4	DWORD	Size: variable
0x06	2	WORD	Number of vertices
0x08	16	GUID	Instance Id
0x18	16	GUID	Profile
and then for each vertex:			
	4	DWORD	Longitude
	4	DWORD	Latitude

Unknown Record 0x3B

Every (?) airport in the FS X scenery files contains a subrecord with the ID of 0x3B. This record contains as usual a DWORD length field at offset 0x02.
It cannot be reproduced with the BglComp compiler, and it has no apparent function.
It concists of a long list of vertices along the perimeter of the airport and a list of indices for triangles to be drawn (similar to the second Apron record), but in fact the sim apparently does not use this list for drawing.
It has the following structure:

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id : **0x003B**
0x02	4	DWORD	Size (variable)
0x06	2	WORD	Unknown
0x08	2	WORD	Number of vertices
0x0A	2	WORD	Number of triangles
and then for each vertex:			
	4	DWORD	Longitude
	4	DWORD	Latitude
and then for each triangle:			
	2	WORD	Index of vertex #1
	2	WORD	Index of vertex #2
	2	WORD	Index of vertex #3

AIRPORTSUMMARY

The record has a fixed size of 44 bytes.

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (<b>0x0032)
0x02	4	DWORD	Size (0x2C)
0x06	2	WORD	Type of COM present at the airport, runway surfaces and approach availability. Each availability is coded on 1 bit: 0 = No, 1 = Yes ▪ Bit 0 : Availability of at least one COM TOWER station ▪ Bits 1-2 : (1=At least one ASPHALT or CONCRETE runway, 2=Only WATER runway(s), 0=All other combinations) ▪ Bits 3-4 : Unknown (0) ▪ Bit 5 : Availability for GPS approach ▪ Bit 6 : Availability for VOR approach ▪ Bit 7 : Availability for NDB approach ▪ Bit 8 : Availability for ILS approach ▪ Bit 9 : Availability for LOC approach ▪ Bit 10 : Availability for SDF approach ▪ Bit 11 : Availability for LDA approach ▪ Bit 12 : Availability for VORDME approach ▪ Bit 13 : Availability for NDBDME approach ▪ Bit 14 : Availability for RNAV approach ▪ Bit 15 : Availability for LOCBC approach
0x08	4	DWORD	Longitude
0x0C	4	DWORD	Latitude
0x10	4	DWORD	Altitude x 1000 (in meters)
0x14	4	DWORD	ICAO Ident.
0x18	4	DWORD	Region
0x1C	4	FLOAT	Magnetic Variation
0x20	4	FLOAT	Length of the longest runway
0x24	4	FLOAT	Heading of the longest runway
0x28	4	DWORD	Type of fuel present at the airport and its availability. Each availability is coded on 2 bits: 0 = No, 1 = Unknown, 2 = Prior Request, 3 = Yes ▪ Bits 0-1 : Availability for 73 octane ▪ Bits 2-3 : Availability for 87 octane ▪ Bits 4-5 : Availability for 100 octane ▪ Bits 6-7 : Availability for 130 octane ▪ Bits 8-9 : Availability for 145 octane ▪ Bits 10-11 : Availability for MOGAS ▪ Bits 12-13 : Availability for JET ▪ Bits 14-15 : Availability for JETA ▪ Bits 16-17 : Availability for JETA1 ▪ Bits 18-19 : Availability for JETAP ▪ Bits 20-21 : Availability for JETB ▪ Bits 22-23 : Availability for JET4 ▪ Bits 24-22 : Availability for JET4 ▪ Bit 30 : Airport has avgas ▪ Bit 31 : Airport has jet fuel

ILS/VOR

The records for ILS and VOR are in the same section and they are identical for the fixed section. ILS records can have an additional subrecords. The fixed part is 40 bytes long and has the following structure:

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x0013**)
0x02	4	DWORD	Size
0x06	1	BYTE	Type. The following numbers have been found: ▪ 0x0001 VOR TERMINAL ▪ 0x0002 VOR LOW ▪ 0x0003 VOR HIGH ▪ 0x0004 ILS ▪ 0x0005 VOR VOT
0x07	1	BYTE	Flags. The following bits have been recognized: ▪ bit 0: if 0 then DME only, otherwise 1 for ILS ▪ bit 2: backcourse (0 = false, 1 = true) ▪ bit 3: glideslope present ▪ bit 4: DME present ▪ bit 5: NAV true
0x08	4	DWORD	Longitude
0x0C	4	DWORD	Latitude
0x10	4	DWORD	Elevation x 1000 (in meters)
0x14	4	DWORD	Frequency
0x18	4	FLOAT	Range in meters
0x1C	4	FLOAT	Magnetic Variation
0x20	4	DWORD	ICAO ident (Special Format)
0x24	4	DWORD	bit 0-10 regionId bit 11-31 airportId (for ILS)

The following subrecords can follow:

Localizer

(For ILS)

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id Localizer (**0x0014**)
0x02	4	DWORD	Size (0x10)
0x06	1	BYTE	Runway Number
0x07	1	BYTE	Runway Designator (0=NONE, 1=LEFT, 2=RIGHT, 3=CENTER)
0x08	4	FLOAT	Heading (True,°)
0x0C	4	FLOAT	Loc Beam width (°)

GlideSlope

For (ILS)

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id GlideSlope (**0x0015**)
0x02	4	DWORD	Size (0x1C)
0x06	2	WORD	Unknown
0x08	4	DWORD	Longitude
0x0C	4	DWORD	Latitude
0x10	4	DWORD	Elevation x 1000 (in meters)
0x14	4	FLOAT	Range
0x18	4	FLOAT	Pitch

Dme

For (ILS/VOR)

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id DME(**0x0016**)
0x02	4	DWORD	Size (0x18)
0x06	2	WORD	Unknown
0x08	4	DWORD	Longitude
0x0C	4	DWORD	Latitude
0x10	4	DWORD	Elevation x 1000 (in meters)
0x14	4	FLOAT	Range

After these subsections, a name subsection is added:

Name

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x0019**)
0x02	4	DWORD	Size
0x06		STRING	Name (max. 48 characters)

If VisualModel is added in the source file, the compiler adds another section to the file with a record of type **0x0025** (SceneryxObject) with the GUID for the object referenced. The coordinates for this objects are taken from the ILS/VOR and adjusted, if BiasXYZ is added to the VisualModel.

TACAN

P3D Only

Applicable LM P3D

The TACAN record has a 63 bytes long fixed section and a name section of variable length. The fixed section has the following structure:

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x00A0**)
0x02	4	DWORD	Size
0x06	4	DWORD	Longitude
0x0A	4	DWORD	Latitude
0x0E	4	DWORD	Elevation x 1000 (in meters)
0x12	4	DWORD	Channel
0x16	1	BYTE	Type. The following numbers have been found: ▪ bit 0: 0=X type, 1=Y type ▪ bit 1: 0=DmeOnly true, 1=DmeOnly false ▪ bit 2-7: Unknown/Unused(0)
0x17	4	FLOAT	Range in meters
0x1B	4	FLOAT	Magnetic Variation
0x1F	4	DWORD	ICAO ident (Special Format)
0x23	4	DWORD	bit 0-10 regionId bit 11-31 airportId (for ILS)
0x27	24	BYTE	Unknown sub-record with an Id of 0x16

The name subsection has the following structure

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x0019**)
0x02	4	DWORD	Size
0x06		STRING	Name

NDB

The NDB records are stored in a separate section. They have a 40 bytes long fixed section and a name section of variable length. The fixed section has the following structure:

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x0017**)
0x02	4	DWORD	Size: variable
0x06	2	WORD	Type ▪ 0 = COMPASS_POINT ▪ 1 = MH ▪ 2 = H ▪ 3 = HH
0x08	4	DWORD	Frequency
0x0C	4	DWORD	Longitude
0x10	4	DWORD	Latitude
0x14	4	DWORD	Elevation x 1000 (in meters)
0x18	4	FLOAT	Range
0x1C	4	FLOAT	Magnetic Variation
0x20	4	DWORD	ICAO ident (Special Format)
0x24	4	DWORD	bit 0-10: region bit 11-31: ICAO id of airport, if it was defined with an airport (terminal NDB)

The name subsection has the following structure

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x0019**)
0x02	4	DWORD	Size
0x06		STRING	Name

SCENERYOBJECT

TaxiwaySign

These records are coded in the section for scenery objects (0x25) with a separate type of entry. Apparently all Taxiway signs for one airport are coded together on one record. There seems to be no coordination of this record with the airport record to which it belongs!

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x000E**) 0x05 for FS9
0x02	2	WORD	Size: variable
0x04	4	DWORD	Longitude
0x08	4	DWORD	Latitude
0x0C	4	DWORD	Altitude (?) cannot be coded with the compiler
0x10	2	WORD	Flags (cannot be coded) ▪ Bit 0: IsAboveAGL ▪ Bit 1: NoAutogenSuppression ▪ Bit 2: NoCrash ▪ Bit 3: NoFog ▪ Bit 4: NoShadow ▪ Bit 5: NoZWrite ▪ Bit 6: NoZTest
0x12	2	WORD	Pitch(?) - Cannot be coded
0x14	2	WORD	Bank(?) - Cannot be coded
0x16	2	WORD	Heading(?) - Cannot be coded
0x18	2	WORD	ImageComplexity (?) - Cannot be coded
0x1A	2	WORD	Unknown
0x1C	2	WORD	Number of taxiway signs for this airport
and then for each sign:			
0x00	4	FLOAT	Longitude offset from anchor value in fix part above Value = (TaxiSign Longitude - Anchor Longitude) * 40075000 * cos ((pi/180) * abs (Anchor Latitude + LatOffset /2) / 360
0x04	4	FLOAT	Latitude offset from anchor value in fix part above Value = (TaxiSign Latitude - Anchor Latitude) * 40007000 / 360
0x08	2	WORD	Heading
0x0A	1	BYTE	Size ▪ 1 = SIZE1 ▪ 2 = SIZE2 ▪ 3 = SIZE3 ▪ 4 = SIZE4 ▪ 5 = SIZE5
0x0B	1	BYTE	Justification (1 = right, 2 = left)
0x0C		STRINGZ	Label (zero filled to next WORD address)

LibraryObject

The record has a fixed length of 0x40 bytes with the following structure:

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x000B**) (0x02 for FS9)
0x02	2	WORD	Size (0x0040) (0x0030 for FS9)
0x04	4	DWORD	Longitude
0x08	4	DWORD	Latitude
0x0C	4	DWORD	Altitude
0x10	2	WORD	Flags ▪ Bit 0: IsAboveAGL ▪ Bit 1: NoAutogenSuppression ▪ Bit 2: NoCrash ▪ Bit 3: NoFog ▪ Bit 4: NoShadow ▪ Bit 5: NoZWrite ▪ Bit 6: NoZTest
0x12	2	WORD	Pitch
0x14	2	WORD	Bank
0x16	2	WORD	Heading
0x18	2	WORD	imageComplexity ▪ 0 = VERYSPARSE ▪ 1 = SPARSE ▪ 2 = NORMAL ▪ 3 = DENSE ▪ 4 = VERYDENSE
0x1A	2	WORD	Unknown
0x1C	16	GUID	FSX only: contains an empty GUID
0x2C	16	GUID	Name
0x3C	4	FLOAT	Scale

If an **AttachedObject** exists, there are 3 other records following:

AttachedObject

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x1002**)
0x02	2	WORD	Size (0x0004)
and then the 2nd record			
0x00	2	WORD	ID depending on the kind of attached object. It is possible to attach beacons, effects and other library objects
0x02	2	WORD	Size
0x04	2	WORD	Offset of attach point string
0x06	2	WORD	Pitch
0x08	2	WORD	Bank
0x0A	2	WORD	Heading
0x0C	4	DWORD	BiasX
0x10	4	DWORD	BiasY
0x14	4	DWORD	BiasZ
0x18	16	GUID	Instance Id
0x28	2	WORD	Probability
0x30	2	WORD	Randomness
0x32	1	BYTE	Type ▪ 0xF5 = CIVILIAN AIRPORT ▪ 0xF6 = CIVILIAN HELIPORT ▪ 0xF7 = CIVILIAN SEA_BASE ▪ 0xF8 = MILITARY AIRPORT ▪ 0xF9 = MILITARY HELIPORT ▪ 0xFA = MILITARY SEA_BASE
0x33	1	BYTE	Unknown, always 0x01 (?)
0x34	2	WORD	unknown , always 0x0000
0x36		STRINGZ	Name of the attachment point
and then the 3rd record			
0x00	2	WORD	Id (**0x1001**)
0x02	2	WORD	Size (0x004)

Effect

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x000D**)
0x02	2	WORD	Size: variable
0x04	4	DWORD	Longitude
0x08	4	DWORD	Latitude
0x0C	4	DWORD	Altitude
0x10	2	WORD	Flag: 1 = isAboveAGL
0x12	2	WORD	Pitch
0x14	2	WORD	Bank
0x16	2	WORD	Heading
0x18	2	WORD	ImageComplexity ▪ 0 = VERYSPARSE ▪ 1 = SPARSE ▪ 2 = NORMAL ▪ 3 = DENSE ▪ 4 = VERYDENSE
0x1A	2	WORD	Unknown
0x1C	80	STRINGZ	effectName
0x6C		STRINGZ	effectParams

GenericBuilding

NB.: BuildingBias is not implemented in the compiler.

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x000A**)
0x02	2	WORD	Size: variable
0x04	4	DWORD	Longitude
0x08	4	DWORD	Latitude
0x0C	4	DWORD	Altitude
0x10	2	WORD	flag: 1 = isAboveAGL
0x12	2	WORD	Pitch
0x14	2	WORD	Bank
0x16	2	WORD	Heading
0x18	2	WORD	imageComplexity ▪ 0 = VERYSPARSE ▪ 1 = SPARSE ▪ 2 = NORMAL ▪ 3 = DENSE ▪ 4 = VERYDENSE
0x1A	2	WORD	Unknown
0x1C	4	FLOAT	Scale
0x20	2	WORD	type: 0x00a0 generic building
0x22	2	WORD	Size fo record
0x24	2	WORD	subtype. The following numbers have been identified: ▪ 0x0004 rectangular with roofType FLAT ▪ 0x0006 rectangular with roofType RIDGE ▪ 0x0007 rectangular with roofType PEAKED ▪ 0x0008 rectangular with roofType SLANT ▪ 0x0009 pyramidal building ▪ 0x000a multisidedBuilding
for all rectangular buildings:			
0x26	2	WORD	SizeX (0)
0x28	2	WORD	SizeZ (1)
0x2A	2	WORD	bottomTexture (2)
0x2C	2	WORD	sizeBottomY (3)
0x2E	2	WORD	textureIndexBottomX (4)
0x30	2	WORD	textureIndexBottomZ (5)
0x32	2	WORD	windowTexture(6)
0x34	2	WORD	sizeWindowY(7)
0x36	2	WORD	textureIndexWindowX(8)
0x38	2	WORD	textureIndexWindowY(9)
0x3A	2	WORD	textureIndexWindowZ(10)
0x3C	2	WORD	topTexture (11)
0x3E	2	WORD	sizeTopY (12)
0x40	2	WORD	textureIndexTopX (13)
0x42	2	WORD	textureIndexTopZ (14)
0x44	2	WORD	roofTexture (15)
0x46	2	WORD	textureIndexRoofX (16)
0x48	2	WORD	textureIndexRoofX (17)
end for rectangular buildings with rooftype FLAT.			
for rectangular buildings with roofType RIDGE or SLANTED:			
0x4A	2	WORD	sizeRoofY (18)
0x4C	2	WORD	textureIndexGableY (19)
0x4E	2	WORD	gableTexture (20)

0x50	2	WORD	textureIndexGableZ (21)
for roofType SLANTED only:			
0x52	2	WORD	faceTexture (22)
0x54	2	WORD	textureIndexFaceX (23)
0x56	2	WORD	textureIndexFaceX (24)
for rectangular buildings with roofType PEAKED:			
0x4A	2	WORD	sizeRoofY(18)
0x4C	2	WORD	textureIndexRoofY (19)
for multisided buildings:			
0x26	2	WORD	buildingSides Note: The Argument for smoothing is required by the compiler, but it has no effect on the BGL-file
0x28	2	WORD	sizeX (1)
0x2A	2	WORD	sizeZ (2)
0x2C	2	WORD	bottomTexture (3)
0x2E	2	WORD	sizeBottomY (4)
0x30	2	WORD	textureIndexBottomX (5)
0x32	2	WORD	windowTexture (6)
0x34	2	WORD	sizeWindowY (7)
0x36	2	WORD	textureIndexWindowX (8)
0x38	2	WORD	textureIndexWindowY (9)
0x3A	2	WORD	topTexture (10)
0x3C	2	WORD	sizeTopY (11)
0x3E	2	WORD	textureIndexTopX (12)
0x40	2	WORD	roofTexture (13)
0x42	2	WORD	sizeRoofY (14)
0x44	2	WORD	textureIndexRoofX (15)
0x46	2	WORD	textureIndexRoofY (16)
Note: textureIndexRoofY is required by the compiler, but it has no effect on the bgl file !			
for pyramidal buildings:			
0x26	2	WORD	sizeX (0)
0x28	2	WORD	sizeZ (1)
0x2A	2	WORD	sizeTopX (2)
0x2C	2	WORD	sizeTopZ (3)
0x2E	2	WORD	bottomTexture (4)
0x30	2	WORD	sizeBottomY (5)
0x32	2	WORD	textureIndexBottomX (6)
0x34	2	WORD	textureIndexBottomZ (7)
0x36	2	WORD	windowTexture (8)
0x38	2	WORD	sizeWindowY (9)
0x3A	2	WORD	textureIndexWindowX (10)
0x3C	2	WORD	textureIndexWindowY (11)
0x3E	2	WORD	textureIndexWindowZ (12)
0x40	2	WORD	topTexture (13)
0x42	2	WORD	sizeTopY (14)
0x44	2	WORD	textureIndexTopX (15)
0x46	2	WORD	textureIndexTopZ (16)
0x48	2	WORD	roofTexture (17)
0x4A	2	WORD	textureIndexRoofX (18)

0x4C	2	WORD	textureIndexRoofZ (19)
for all			
	2	WORD	Unknown : 0x22 0x00

Windsock

Record with fixed length of 46 bytes.

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x0C**)
0x02	2	WORD	Size (0x2E)
0x04	4	DWORD	Longitude
0x08	4	DWORD	Latitude
0x0C	4	DWORD	Altitude
0x10	2	WORD	flags (unused)
0x12	2	WORD	Pitch
0x14	2	WORD	Bank
0x16	2	WORD	Heading
0x18	2	WORD	imageComplexity
0x1A	2	WORD	Unknown
0x1C	16	GUID	Instance Id
0x2C	4	FLOAT	poleheight
0x30	4	FLOAT	sockLength
0x34	1	BYTE	PoleColor: blue
0x35	1	BYTE	PoleColor: green
0x36	1	BYTE	PoleColor: red
0x37	1	BYTE	PoleColor ? (0xFF)
0x38	4	BYTE[4]	SockColor
0x3C	2	WORD	flag: lighted (TRUE = 0x0001)

Extrusion Bridge

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x12**)
0x02	2	WORD	Size
0x04	4	DWORD	Longitude
0x08	4	LONG	Latitude
0x0C	4	DWORD	Altitude
0x10	2	WORD	Flags
0x12	2	WORD	Pitch
0x14	2	WORD	Bank
0x16	2	WORD	Heading
0x18	2	WORD	imageComplexity
0x1A	2	WORD	Unknown
0x1C	16	GUID	Instance Id
0x2C	16	GUID	Profile
0x3C	16	GUID	Material Set
0x4C	12	DWORD[3]	altitude sample location 1
0x58	12	DWORD[3]	altitude sample location 2
0x64	4	FLOAT	road width
0x68	4	FLOAT	Probability
0x6C	1	BYTE	Suppress
0x6D	1	BYTE	Placement Count
0x6E	2	WORD	Point count
			and then for each polyline object placement
	16	GUID	placement id
			and then for each polyline point
	4	DWORD	Longitude
	4	DWORD	Latitude
	4	LONG	Elevation

Trigger

The record consists of a fixed part and a variable part. The fixed part is 34 bytes long and has the following structure:

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x10**)
0x02	2	WORD	Size: variable
0x04	4	DWORD	Longitude
0x08	4	DWORD	Latitude
0x0C	4	DWORD	Altitude
0x10	2	WORD	AltitudeIsAGL (0x0001 = TRUE)
0x12	2	WORD	Pitch
0x14	2	WORD	Bank
0x16	2	WORD	Heading
0x18	2	WORD	imageComplexity
0x1A	2	WORD	Unknown
0x1C	16	GUID	Instance Id
0x2C	2	WORD	Type ▪ 0x0000 = REFUEL_REPAIR ▪ 0x0001 = WEATHER
0x2E	4	FLOAT	triggerHeight
In case of **WEATHER** the variable part has the following structure:			
0x32	2	WORD	Type ▪ 0x0001 = RIDGE_LIFT ▪ 0x0002 = UNIDIRECTIONAL_TURBULENCE note: in bglcomp.xsd this keyword is spelled NONDIRECTIONAL_TURBULENCE, but the compiles does not understand it. If you change the keyword in bglcomp.xsd compilation is ok. ▪ 0x0003 = DIRECTIONAL_TURBULENCE ▪ 0x0004 = THERMAL
0x34	4	FLOAT	Heading
0x38	4	FLOAT	Scalar
0x3C	4	DWORD	Number of vertices
			and then for each vertex:
	4	FLOAT	BiasX
	4	FLOAT	BiasY
In case of **FUEL_REPAIR** the variable part has the following structure:			
0x32	4	DWORD	Fuel type and availability ▪ bit 0-1: type 73 ▪ bit 2-3: type 87 ▪ bit 4-5: type 100 ▪ bit 6-7: type 130 ▪ bit 8-9: type 145 ▪ bit 10-11: type MOGAS ▪ bit 12-13: type JET ▪ bit 14-15: type JETA ▪ bit 16-17: type JETA1 ▪ bit 18-19: type JETAP ▪ bit 20-21: type JETB ▪ bit 22-23: type JET4 ▪ bit 24-25: type JET5 ▪ bit 26-29 : unused ▪ bit 30 : piston type ▪ bit 31 : jet type for all except last two : ▪ 0 = NO ▪ 1 = UNKNOWN ▪ 2 = PRIOR_REQUEST ▪ 3 = YES when type=UNKNOWN and availability = YES then type=100 and type = JETA both are set to availability=YES

0x036	4	DWORD	Number of vertices
and then for each vertex			
0x036	4	FLOAT	BiasX
0x036	4	FLOAT	BiasZ

MARKER

The Marker record has a fixed length of 28 bytes with the following structure:
NOTE: The structure given by Winfried appears to be wrong in the first three fields

- ID 2 bytes WORD
- Size 4 bytes DWORD
- Heading 1 byte BYTE

The structure below is taken from the source code of BGLXMl.

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x18**)
0x02	4	WORD	Size (0x0000001C)
0x04	1	BYTE	Unknown
0x05	2	WORD	Heading
0x07	1	BYTE	Type - 0 = INNER - 1 = MIDDLE - 2 = OUTER - 3 = BACKCOURSE
0x08	4	DWORD	Longitude
0x0C	4	DWORD	Latitude
0x10	4	DWORD	Altitude
0x14	4	DWORD	Ident (Special Format)
0x18	2	WORD	Region (Special Format)
0x1A	2	WORD	Unknown (0x0000)

BOUNDARY

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x20**)
0x00	4	DWORD	Size: variable
0x06	1	BYTE	Type - 00 = NONE - 01 = CENTER - 02 = CLASS_A - 03 = CLASS_B - 04 = CLASS_C - 05 = CLASS_D - 06 = CLASS_E - 07 = CLASS_F - 08 = CLASS_G - 09 = TOWER - 0a = CLEARANCE - 0b = GROUND - 0c = DEPARTURE - 0d = APPROACH - 0e = MOA - 0f = RESTRICTED - 10 = PROHIBITED - 11 = WARNING - 12 = ALERT - 13 = DANGER - 14 = NATIONAL_PARK - 15 = MODEC - 16 = RADAR - 17 = TRAINING
0x07	1	BYTE	BIT 0-3: maximumAltitudeType BIT 4-7: minimumAltitudeType - 1 = MEAN_SEA_LEVEL (= UNKNOWN) - 2 = ABOVE_GROUND_LEVEL - 3 = UNLIMITED
0x08	4	DWORD	Minimum Longitude of area covered
0x12	4	DWORD	Minimum Latitude of area covered
0x16	4	DWORD	Minimum Altitude * 1000
0x20	4	DWORD	Maximum Longitude of area covered
0x24	4	DWORD	Maximum Latitude of area covered
0x28	4	DWORD	Maximum Altitude
0x32	2	WORD	Type field of name record (0x19)
0x34	4	DWORD	Size of name record
0x36	size-6	STRING	Name

on this follows a record describing the drawing of the lines

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x21**)
0x02	4	DWORD	size: variable
0x06	2	WORD	Number of points to follow
			for each point 10 bytes
0x00	2	WORD	Type of point ▪ 1 = START ▪ 2 = LINE ▪ 3 = ORIGIN ▪ 4 = ARC clockwise ▪ 5 = arc counter-clockwise ▪ 6 = circle NB: in case of circle, the entries for minimumAltitude and maximumAltitude override the values in start if both are given. the start entry is in case of circle not needed at all Note: there is a bug in the new version of bglcomp.xsd: the word BoundaryStart in grpBoundaryChildren has to be replaced by Start, otherwise the compiler does not accept it!
0x02	4	DWORD	Latitude of point (in case of circle: unknown, = 0x0000)
0x06	4	DWORD	Longitude of point (in case of circle: FLOAT: radius)

GEOPOL

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x23**)
0x02	2	DWORD	Size: Variable
0x06	2	WORD	Bit 0-13: Number of vertices number of vertices Bit 14-15: Type 0x40 = BOUNDARY 0x80 = COASTLINE)
0x08	2	DWORD	Minimum Longitude
0x0C	2	DWORD	Minimum Latitude
0x10	2	DWORD	Maximum Longitude
0x14	2	DWORD	Maximum Latitude
variable part: for each vertex			
0x02	2	DWORD	Longitude
0x04	2	DWORD	Latitude

MODEL DATA

The model data structure has a fixed length of 24 bytes.

Offset	Number of bytes	Format	Description
0x00	16	GUID	Name
0x10	4	DWORD	Mdl File offset from the start of this subsection
0x14	4	DWORD	Mdl File Length

EXCLUSIONRECTANGLE

This record has a fixed length record of 20 bytes.

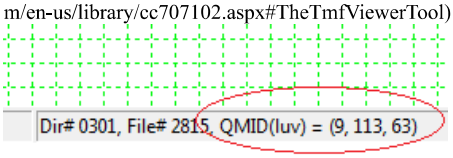
Offset	Number of bytes	Format	Description
0x00	2	WORD	exclusion type 0x0008 = exclude All Otherwise: ▪ bit 4 = Beacon Objects ▪ bit 5 = Effect Objects ▪ bit 6 = GenericBuilding Objects ▪ bit 7 = Library Objects ▪ bit 8 = TaxiwaySign Objects ▪ bit 9 = Trigger Objects ▪ bit 10 = Windsock Objects ▪ bit 11 = ExclusionBridge Objects
0x02	2	WORD	Size (unused - always 0)
0x04	4	DWORD	Longitude of NW corner
0x08	4	DWORD	Latitude of NW corner
0x0C	4	DWORD	Longitude of SE corner
0x10	4	DWORD	Latitude of SE corner

NAMELIST

The namelist contains only one record of variable length. It consists of a fixed part and a variable part. The fixed part is 42 bytes long and has the following structure:

Offset	Number of bytes	Format	Description
0x00	2	WORD	Id (**0x0027**)
0x02	4	DWORD	Size (?) seems always to be 0x00000000
0x06	2	WORD	Number of region names
0x08	2	WORD	Number of country names
0x0A	2	WORD	Number of state names
0x0C	2	WORD	Number of city names
0x0E	2	WORD	Number of airport names
0x10	2	WORD	Number of ICAO Ident
0x12	4	DWORD	Offset of region list (from start of record)
0x16	4	DWORD	Offset of country list (from start of record)
0x1A	4	DWORD	Offset of state list (from start of record)
0x1E	4	DWORD	Offset of city list (from start of record)
0x22	4	DWORD	Offset of airport list (from start of record)
0x26	4	DWORD	Offset of ICAO Ident list (from start of record)

The lists for region, country, state, city and airport names have all the same structure: an index with 1 DWORD for each entry in the list. The DWORD is an offset in the buffer of names that follows the DWORDs. Each name in that buffer is null-char terminated. Note that the offset values may not be in ascending order (see example).
The ICAO list has a different structure. It contains n entries (one for each ICAO name), each of them 20 bytes long, with the following structure:

Offset	Number of bytes	Format	Description
0x00	1	BYTE	Region name index (all indexes start with 0 for the first name in the relevant list)
0x01	1	BYTE	Country Name index
0x02	2	WORD	bit 0-3 : unknown bit 4-15 : state name index
0x04	2	WORD	City Name index
0x06	2	WORD	Airport Name index
0x08	4	DWORD	ICAO Identifier (Special Format)
0x0C	4	DWORD	Region Ident (Special Format)
0x10	2	WORD	Longitude of the upper left corner of the corresponding QMID Level 9 Square. The value stored here is the same (2nd value in the triplet) that you see in the status bar of the TmfViewer (http://msdn.microsoft.com/en-us/library/cc707102.aspx#TheTmfViewerTool) application when the QMID grid Level 9 is selected.  . The 1st value of the triplet is the level.
0x12	2	WORD	Latitude of the upper left corner of the corresponding QMID Level 9 Square The value stored here is the same (3rd value in the triplet) that you see in the status bar of the TmfViewer (http://msdn.microsoft.com/en-us/library/cc707102.aspx#TheTmfViewerTool) application when the QMID grid Level 9 is selected.

Example

Let's look at the file APX25230.bgl located in the (...)\Microsoft Flight Simulator X\Scenery\0302\scenery folder.
The NameList record starts at offset 0x77B4.

Offset	Field	Value
77B4	Id	0x0027
77B6	Size	0
77BA	NbRegionNames	1
77BC	NbCountryNames	1
77BE	NbStateNames	1
77C0	NbCityNames	3
77C2	NbAirportNames	3
77C4	NbICAOIdents	3
77C6	RegionListOffset	0x002A
77CA	CountryListOffset	0x002F
77CE	StateListOffset	0x0038
77D2	CityListOffset	0x003D
77D6	AirportListOffset	0x0070
77DA	ICAOListOffset	0x00AA

Let's focus on the **cities**. There are 3 cities defined (at per offset 0x77C0). So the cities list start at offset 0x77B4 + 0x3D = 0x77F1.
At offset 0x77F1, we find 3 DWORDs (one for each city).

Offset	Field	Value
77F1	Offset #0	7: the first name if at offset 7 in the buffer that follow at offset 0x77FD
77F5	Offset #1	0: the second name if at offset 0 in the buffer that follow at offset 0x77FD
77F9	Offset #2	14: the second third if at offset 14 in the buffer that follow at offset 0x77FD

Then follows a buffer of 38 (0x26) bytes containing the names of the cities (since the AirportList starts at 0x77B4 + 0x70 = 0x7824, we know the the buffer of cities names stops at 0x7823 so it has size of 0x26).

```
4D 61 72 69 65 6C 00 48 61 76 61 6E 61 00 53 61
6E 20 41 6E 74 6F 6E 69 6F 20 44 65 20 4C 6F 73
20 42 61 6E 6F 73
```

The first name starts at offset 7:

```
48 61 76 61 6E 61 00 : Havana
```

The second name starts at offset 0:

```
4D 61 72 69 65 6C 00 : Mariel
```

The third name starts at offset 14:

```
53 61 6E 20 41 6E 74 6F 6E 69 6F 20 44 65 20 4C 6F 73 20 42 61 6E 6F 73 : San Antonio De Los Banos
```

Now the **ICAO**.
The ICAO list starts at offset 0x77B4 + 0xAA = 0x785E. There are 3 ICA identifiers, so 3 consecutive records of 20 bytes. Let's look at the first ICAO record:

Offset	Field	Value
785E	RegionNameIndex	0
785F	CountryNameIndex	0
7860	StateNameIndex	0
7862	CityNameIndex	1 : Index = 1 so that is the 2nd name in the cities names list = Mariel
7864	AirportNameIndex	0
7866	ICAOIdent	E1 0C 9A 02 => MUML
786A	Unknown	???
786E	UpperLeft Longitude of the QMID	67 00 => -83.4375
7870	UpperLeft Latitude of the QMID	5F 00 => 23.203125

TERRAIN_VECTOR_DB

This type of subsection contains values to retrieve geographical coordinates and types of vector data, organized as segments.

The section does not contain geographical coordinates per se but **offsets** to the lower left corner (minimum latitude / longitude) of the covered area (QMID Square). The covered area is also defined in the subsection.
Not that a vector that stretches over several QMID squares is split in as many sub-vectors.

The data is organized in lists of pair values (One pair per geographical coordinate). Each pair can then be used to compute the final geographical coordinates. There are 3 ways to retrieve these lists of pairs.

The first value in a pair is longitude-related. The second value is latitude-related.

The algorithm to compute the geographical coordinates from a pair is:

```
void convertToCoordinates (double longitude_related_Value, double latitude_related_Value)
{
    var deltaLongFactor = (MaxLongitudeDeg - MinLongitudeDeg) / 0x8000;
    var deltaLatFactor = (MaxLatitudeDeg - MinLatitudeDeg) / 0x8000;

    var LongitudeDeg = MinLongitudeDeg + (longitude_related_Value * deltaLongFactor);
    var LatitudeDeg = MinLongitudeDeg + (latitude_related_Value * deltaLatFactor);
}
```

Subsection Header

Each subsection starts with the following structure:

<table>
<tr><th>Relative Offset</th><th>Number of bytes</th><th>Description</th></tr>
<tr><td>0x00</td><td>4 - DWORD</td><td>6
This is the identifier for a vector subsection.</td></tr>
<tr><td>0x04</td><td>4 - DWORD</td><td>Defines the QMID square related to this subsection (all vectors are within the square). See Getting QMID from DWORD values.</td></tr>
<tr><td>0x08</td><td>4 - DWORD</td><td>ADDTOCELLS

When the -ADDTOCELLS flag is used with the Shp2Vec tool, this value is 1, otherwise 0</td></tr>
<tr><td>0x0C</td><td>4 - DWORD</td><td>Number of entities.
An entity is a list of segments. (a segment is a list of points).</td></tr>
<tr><td>0x10</td><td>4 - DWORD</td><td>Number of bytes in the attributes buffer.
The attributes buffer contains some GUIDs identifying the segments of this subsection.

Some GUID values (like the Texture) are not used by the TmfViewer application. It also contains some extra bytes needed by some attributes (for example, 2 FLOAT32 values - SlopeX and SlopeY - are needed for WaterPolys).
The GUID values are defined at Terrain And Scenery (http://msdn.microsoft.com/en-us/library/cc707102.aspx#TheShp2VecTool)
(See Vector Attributes for the Shp2Vec Tool)
The usable primary GUID values are:

<pre>{359C73E8-06BE-4FB2-ABCB-EC942F7761D0} Airport Bounds
{91CB4A9B-9398-48E6-81DA-70AEA3295914} Parks
{EA0C44F7-01DE-4D10-97EB-FB5510EB7B72} Water Polygons (GPS)
{956A42AD-EC8A-41BE-B7CB-C68B5FF1727E} Water Polygons
{AC39CDCB-DB78-4628-9A7C-051DA7AC864A} Exclusions
{0CBC8FAD-DF73-40A1-AD2B-FE62F8004F6F} Shorelines
{714BF912-F9DF-467E-80AE-28EB27374DBD} Streams
{C7ACE4AE-871D-4938-8BDC-BB29C4BBF4E3} Utilities
{33239EB4-D2B8-46F5-98AB-47B3D0922E2A} Railways
{560FA8E6-723D-407D-B730-AE08039102A5} Roads
{54B91ED8-BC02-41B7-8C3B-2B8449FF85EC} Freeway Traffic Roads
{1B6A15BB-05FB-4401-A8D1-BB520E84904C} Water Polygons Slope</pre></td></tr>
<tr><td>0x14</td><td>4 - DWORD</td><td>Number of attribute offsets used in that subsection.</td></tr>
<tr><td>0x18</td><td>4 - DWORD</td><td>Number of points used in that subsection.</td></tr>
<tr><td>0x1C</td><td>4 - DWORD</td><td>Number of points, in that subsection, that have a different altitude value than the altitude of their siblings in the same segment.</td></tr>
<tr><td>0x20</td><td>N</td><td>Attributes buffer containing <N> bytes, where N is defined at relative offset 0x10.

The attributes buffer contains the primary GUIDs identifying the type of vectors (Roads, RailWays, Streams,...) used in the QMID square related to this subsection, as well as secondary GUIDS (Type of park for example).
The GUIDs are those defined in the XML file used by the Shp2Vec tool (http://msdn.microsoft.com/en-us/library/cc707102.aspx#TheShp2VecTool). The buffer is organized as this:

<table>
<tr><th>Description</th><th>Number of bytes</th></tr>
<tr><td>Vector Type 1 GUID</td><td>16 - GUID</td></tr>
<tr><td>Nb Additional bytes (ADD) for Type 1</td><td>4 - DWORD</td></tr>
<tr><td colspan="2">if additional bytes > 0</td></tr>
<tr><td>Additional Data

For Type Texture, the additional data is usually a GUID (ADD=16) and is one of the values defined at Vector Shape Properties GUIDs (http://msdn.microsoft.com/en-us/library/cc526968.aspx)</td><td>ADD</td></tr>
<tr><td>Vector Type 2 GUID</td><td>16 - GUID</td></tr>
<tr><td>Nb Additional bytes (ADD) for Type 2</td><td>4 - DWORD</td></tr>
<tr><td>...</td><td></td></tr>
</table></td></tr>
</table>

Attributes Buffer Example

This example is taken from the file (...)\Microsoft Flight Simulator X\Scenery\0301\scenery\cvx2815.bgl At offset 0x565, we have 160 bytes:

```
86 7D B0 CE 05 36 BE 44 B4 8A 97 F8 D0 1B 74 DE
08 00 00 00 00 00 00 00 00 00 00 00 AD 8F BC 0C
```

```
73 DF A1 40 AD 2B FE 62 F8 00 4F 6F 00 00 00 00
AD 42 6A 95 8A EC BE 41 B7 CB C6 8B 5F F1 72 7E
00 00 00 00 F7 44 0C EA DE 01 10 4D 97 EB FB 55
10 EB 7B 72 00 00 00 00 BB 15 6A 1B FB 05 01 44
A8 D1 BB 52 0E 84 90 4C 10 00 00 00 82 C1 D5 BC
8B 9C 57 4C 97 BD 27 2C F4 92 CB FF BB 15 6A 1B
FB 05 01 44 A8 D1 BB 52 0E 84 90 4C 10 00 00 00
D8 29 32 7B ED 18 4D 4F AE 22 12 66 08 62 AB A1
```

that can be interpreted as :
At offset 0
86 7D B0 CE 05 36 BE 44 B4 8A 97 F8 D0 1B 74 DE : GUID for Water Polygons Slope {CEB07D86-3605-44BE-B48A-97F8D01B74DE}
08 00 00 00 : 8 bytes of additionnal data (for slopeX and slopeY)
00 00 00 00 : slopeX = 0
00 00 00 00 : slopeY = 0
At offset 0x1C:
AD 8F BC 0C 73 DF A1 40 AD 2B FE 62 F8 00 4F 6F : GUID for Shorelines {0CBC8FAD-DF73-40A1-AD2B-FE62F8004F6F}
00 00 00 00 : No additional data
At offset 0x30:
AD 42 6A 95 8A EC BE 41 B7 CB C6 8B 5F F1 72 7E : GUID for Water Polygons {956A42AD-EC8A-41BE-B7CB-C68B5FF1727E}
00 00 00 00 : No additional data
At offset 0x44:
F7 44 0C EA DE 01 10 4D 97 EB FB 55 10 EB 7B 72 : GUID for Water Polygons - GPS {EA0C44F7-01DE-4D10-97EB-FB5510EB7B72}
00 00 00 00 : No additional data
At offset 0x58:
BB 15 6A 1B FB 05 01 44 A8 D1 BB 52 0E 84 90 4C : GUID Texture {1B6A15BB-05FB-4401-A8D1-BB520E84904C}
10 00 00 00 : 16 bytes of additional data
82 C1 D5 BC 8B 9C 57 4C 97 BD 27 2C F4 92 CB FF : GUID for Hydro_Polygons_Generic_Lake_Perennial {BCD5C182-9C8B-4C57-97BD-272CF492CBFF}
At offset 0x7C:
BB 15 6A 1B FB 05 01 44 A8 D1 BB 52 0E 84 90 4C : GUID Texture {1B6A15BB-05FB-4401-A8D1-BB520E84904C}
10 00 00 00 : 16 bytes of additional data
D8 29 32 7B ED 18 4D 4F AE 22 12 66 08 62 AB A1 : GUID for Shorelines_Generic_Lake {7B3229D8-18ED-4F4D-AE22-12660862ABA1}

Entity Structure

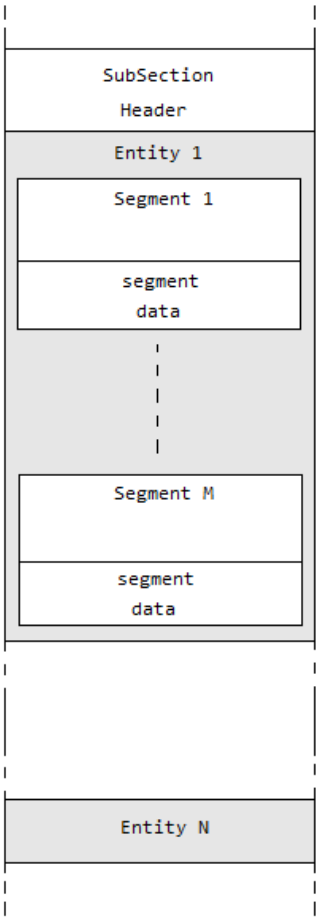
Then for each entity (the number of entities is defined at relative offset 0x0C of the subsection header (see Subsection Header above), we have the following structure:

Relative Offset	Number of bytes	Description
0x00	4 - DWORD	Number of segments in the entity.
0x04	4 - DWORD	Segment type: ▪ 1 – Points only ▪ 2 – Lines ▪ 3 - Polygons
0x08	2 - WORD	Number of signatures offsets (Must be < 0x64).
0x0A	4 * N - DWORD[N]	N = Number of signatures offsets (see above). Will contain the offsets into the signature buffer. For example, 2 signatures offsets may be defined here: one for a park and the second for the texture used for this park.

Segment Structure

Then for each segment (the number of segments is defined at relative offset 0x00 of the entity structure (see Entity Structure above), we have the following structure:

Relative Offset	Number of bytes	Description
0x00	4 - DWORD	Number of points (geographical coordinates) in this segment.
0x04	1 - BYTE	Altitude Information Flag ▪ 0 – No altitude information ▪ 1 – The points have different altitudes ▪ 2 – All points have the same altitude If 1 or 2 there will be additional bytes after the data buffer.
0x05	1 - BYTE	Method used to build the pairs list from the data buffer. Possible values are 1,2 or 3 ▪ 1 – Method1 ▪ 2 – Method2 ▪ 3 – Method3 Note: I did not find a cvx file with a value of 3 but it looks like Method3 read the pairs list directly from the data buffer without any extra processing. Methods 1 and 2 need some extra processing. See below.
0x06		Data buffer – Size may vary depending on the method used. See below.
-	4 x N - FLOAT[N]	N depends on the Altitude Information Flag defined at relative offset 0x04. ▪ If 0 then N = 0 (no bytes) ▪ If 1 then N = NumberOfPoints (as defined at relative offset 0x00). Each FLOAT is the altitude (in meters) of the corresponding point. ▪ If 2 then N = 1 then this is the altitude (in meters) common to all points.



Method 1

By far the most complex method. I have some code for it. But I still have to figure out the big picture.

The data buffer for this method has the following structure:

Relative offset	Number of bytes	Description
0x00	4 - DWORD	First Longitude Data - First value in the list of pairs.
0x04	4 - DWORD	First Latitude Data - First value in the list of pairs.
0x08	4 - DWORD	LongitudeData Increment
0x0C	4 - DWORD	LatitudeData Increment
0x10	4 - DWORD	Number of bytes
0x14	N	N is the number of bytes defined above at relative offset 0x10. These N bytes are the raw data used to build the final list of pairs.

Method 2

The data buffer for this method has the following structure:

Relative offset	Number of bytes	Description
0x00	1 - BYTE	Root Mask
0x01	N	N is computed using the value of the Root Mask. N = (RootMask * NbPoints * 2 + 7) >> 3 where ▪ *RootMask* is defined at relative offset 0x00 of this structure. ▪ *NbPoints* is the number of points as defined at relative offset 0x00 of the Segment Structure. These N bytes are the raw data used to build the final list of pairs.

Algorithm

The algorithm used by Method2 to build the list of pairs is:

```
/*
Fills an array listOfValues of DWORD
*/

PositionInPair = 0;   /* 0 = first value in pair, 1 = second value in pair */
PairIndex = 0 ;
ShiftValue = 0 ;
NbBytesLeftToRead = ((rootMask * nbPoints) * 2 + 7) >> 3;
Mask = 1 << rootMask ;    /* 2 ^ rootMask */

/* Read first value */
if (NbBytesLeftToRead > 3)
{
    valueFromFile = read (4 bytes)
    NbBytesLeftToRead -= 4 ;
}
else
{
    valueFromFile = read (NbBytesLeftToRead bytes)
    NbBytesLeftToRead = 0 ;
}

while (PairIndex < nbPoints)
{
    if (ShiftValue < 0)
    {
        /* Shift Left */
        result = (uint)((valueFromFile << (-ShiftValue)) & Mask);

        /* Add to existing pair value */
        result += listOfValues[PairIndex * 2 + PositionInPair];

        listOfValues[PairIndex * 2 + PositionInPair] = result;
    }
    else
    {
        /* Shift Right */
        result = (uint)((valueFromFile >> ShiftValue) & Mask);
        listOfValues[PairIndex * 2 + PositionInPair] = result;
    }

    if (rootMask + ShiftValue >= 32)
    {
        if (NbBytesLeftToRead > 3)
        {
            valueFromFile = read (4 bytes)
            NbBytesLeftToRead -= 4 ;
        }
        else
        {
            valueFromFile = read (NbBytesLeftToRead bytes)
            NbBytesLeftToRead = 0 ;
```

```
        }
        ShiftValue -= 32;    /* now negative */
    }
    else
    {
        /* continue processing value from file */
        ShiftValue += rootMask;
        PositionInPair++;
    }

    if (PositionInPair == 2)
    {
        PairIndex++;    /* Next entry in the list of pair values */
        PositionInPair = 0;
    }
}
```

Example

Still with the same example as in Subsections, we know that the first subsection data starts at offset 0x4C (as described in the first subsection starting at offset 0x1FCD01) . So at this file offset, we have the subsection header of the first subsection data.

Offset	Value	Description
0x4C	`06 00 00 00`	6
0x50	`00 FA 81 00`	Bounding coordinates: ▪ MinLatitude(Deg) = 47.63671875 ▪ MaxLatitude(Deg) = 47.8125 ▪ MinLongitude(Deg) =- 75.0 ▪ MaxLongitude(Deg) = -74.765625 QMID (u=448, v=240, l=11) , using 0x0081FA00 as A and 0 as B in this algorithm.
0x54	`00 00 00 00`	0 => AddToCells = False
0x58	`03 00 00 00`	Number of entities = 3
0x5C	`14 00 00 00`	Number of bytes in the signatures buffer = 20
0x60	`03 00 00 00`	Number of attribute offsets used in that subsection = 3
0x64	`1C 00 00 00`	Number of pair values in that subsection = 28
0x68	`00 00 00 00`	0
0x6C		Signatures buffer containing 20 bytes. `F7 44 0C EA DE 01 10 4D 97 EB FB 55 10 EB 7B 72 00 00 00 00` That is GUID {EA0C44F7-01DE-4D10-97EB-FB5510EB7B72} = Water Polygons (GPS), followed by 4 zeroes.

Then follows 3 entities.
The first entity (at offset 0x80 of the file) is

Offset	Value	Description
0x80	01 00 00 00	Number of segments in the entity = 1
0x84	03 00 00 00	Segment type = 3 - Polygons
0x88	01 00	Number of signatures offsets = 1
0x8A	00 00 00 00	Signatures offset = 0: points to the only one GUID

This entity is followed by only one segment, at offset 0x8E of the file

Offset	Value	Description
0x8E	0E 00 00 00	Number of points = 14
0x92	00	0 => No Altitude information
0x93	02	2 – Method2
0x94	0F	Root Mask = 0x0F So N = (15 * 14* 2 + 7) >> 3 = 53 53 bytes to follow.
0x95		53 bytes: 84 3E 56 37 65 10 74 1D 3C 44 5B 9F 11 C5 D0 05 C9 2C F3 D5 B2 3A 8D 8A 8D 9D 9B 67 8B ED 52 59 B3 BC 0B 36 35 5F 54 25 B5 33 30 6D E2 64 4B EB 36 A1 8F D5 0D

The next entity starts at file offset 0xCA.

So the algorithm for method 2 goes like this:

```
Mask = 0x7FFF ;

NbBytesLeftToRead = 53 ; shiftValue = 0; PairIndex = 0 ; PositionInPair = 0
   valueFromFile = read (4 bytes) = 0x37563E84
   result = (valueFromFile >> shiftValue) & mask = 0x3E84
   listOfValues[0] = 0x3E84

NbBytesLeftToRead = 49 ; shiftValue = 0x0F; PairIndex = 0 ; PositionInPair = 1
   result = (valueFromFile >> shiftValue) & mask = 0x6EAC
   listOfValues[1] = 0x6EAC

NbBytesLeftToRead = 49 ; shiftValue = 0x1E; PairIndex = 1 ; PositionInPair = 0
   result = (valueFromFile >> shiftValue) & mask = 0x000
   listOfValues[2] = 0x0000
   rootMask + shiftValue = 0x2D > 0x20 so
        valueFromFile = read (4 bytes) = 0x1D741065
        shiftValue = 0xFFFFFFFE

NbBytesLeftToRead = 45 ; shiftValue = 0xFFFFFFFE; PairIndex = 1 ; PositionInPair = 0
   result = (valueFromFile << (-shiftValue)) & mask = 0x4194
   result += listOfValues[2] = 0x4194 + 0x0000 = 0x4194
   listOfValues[2] = 0x4194
```

```
NbBytesLeftToRead = 45 ; shiftValue = 0x0D; PairIndex = 1 ; PositionInPair = 1
   result = (valueFromFile >> shiftValue) & mask = 0x6BA0
   listOfValues[3] = 0x6BA0

NbBytesLeftToRead = 45 ; shiftValue = 0x1C; PairIndex = 2 ; PositionInPair = 0
   result = (valueFromFile >> shiftValue) & mask = 0x0001
   listOfValues[4] = 0x0001
   rootMask + shiftValue = 0x2BD > 0x20 so
        valueFromFile = read (4 bytes) = 0x9F5B443C
        shiftValue = 0xFFFFFFFC

NbBytesLeftToRead = 41 ; shiftValue = 0xFFFFFFFC; PairIndex = 2 ; PositionInPair = 0
   result = (valueFromFile << (-shiftValue)) & mask = 0x43C0
   result += listOfValues[4] = 0x0x43C0 + 0x0001 = 0x43C1
   listOfValues[4] = 0x43C1

etc
```

Once the 53 bytes have been processed, the list contains the following 28 values, making up a list of 14 pairs.

```
3E84
6EAC
4194
6BA0
43C1
6B68
4467
6862
4905
6659
4B57
69D5
58A8
73B1
59E6
76C5
5952
7966
582E
79A9
5545
76A4
4C0C
7136
4B64
6DD6
3E84
6EAC
```

To get the coordinates of the first point, we take the first pair {0x3E84, 0x6EAC} and use the convertToCoordinates method described at Terrain_Vector_DB.
The first value in the pair is longitude-related. The second value in the pair is latitude-related.

```
double deltaLongFactor = ((-74.765625) - (- 75.0)) / 0x8000;  // = 0.000007152557373046875
double deltaLatFactor  = (47.8125 - 47.63671875) / 0x8000;    // = 0.000005364418029785156 3

double LongitudeDeg = (- 75.0) + (0x3E84 * deltaLongFactor);    // = -74.885530471801758
double LatitudeDeg = 47.63671875 + (0x6EAC * deltaLatFactor);   // = 47.788703441619873
```

TERRAIN SECTIONS

The following sections share the same structure:

- TerrainElevation
- TerrainLandClass
- TerrainWaterClass
- TerrainRegion
- PopulationDensity
- TerrainIndex
- TerrainSeasonXXX
- TerrainPhotoXXX
- TerrainPhotoXXX

The number of subsections depends on the size of the scenery area. For higher levels of details, chances are that the area will overlap across multiple QMID squares. However, there will be at least a level for which the entire scenery area is contained inside the QMID square.
So there will be a subsection for each level for which the entire scenery area is contained inside the corresponding QMID square.

For example, let's say that your photo scenery has the following coordinates:
North-West corner : Longitude = -73.388000, Latitude = 45.511400

Sout-East corner : Longitude = -73.373019, Latitude = 45.498672

At QMID level 14, the image overlaps across QMID (luv) = (14, 3639, 2024) and QMID(luv) = (14,3629, 2025).
However, at level 13, the image is entirely contained within QMID(luv) = (13,1819, 1012)
So there will be 14 subsections, the first one defining the level 0 and the last one defining the level 13.

While the relation between the QMID Level and the rank of the subsection may be true for BGL files generated by the resample tool, it appears that this is wrong with other native BGL files such as the worldlc.bgl or other files located in the BASE folder.

SubSection with TRQ1 Records

The number of subsections for a Terrain section is defined in the parent section.
A subsection has a size of 16 (0x10) bytes and has the following structure:

Relative Offset	Number of bytes	Description
0x00	8 - DWORD[2]	These 2 values are derived from the u,v,l values of the QMID containing the whole scenery for this specific level. To retrieve the corresponding QMID values, see the algorithm in Annexe.
0x08	4 - DWORD	File Offset = Position in the file of the subsection's data
0x0C	4 - DWORD	Size of the subsection's data

TRQ1 Record

The TRQ1 Record is a fixed-size header that contains information about each RasterDataLayer "Chunk" in a BGL file. Raster chunks are 2-Dimensional bitmaps of rasterized data containing 16-bit unsigned integer (32-bit for MS Flight) data points broken down into 256X256 pixel squares. The data can represent many different terrain elements including landclass, elevation, and raw aerial imagery data.

At the file offset specified at relative offset 0x08 of the subsection above, we find a TRQ1 record, followed by 1 or 2 other records.

The TRQ1 record has a size of 40 (0x28) bytes for the following structure:

Relative Offset	Number of bytes	Description
0x00	4 - DWORD	Signature = 0x31515254 = 'TRQ1'
0x04	4 - DWORD	Size of this record = 0x28
0x08	2 - WORD	Identifier of the parent section ▪ 1: TERRAIN_PHOTO_XXX ▪ 2: TERRAIN ELEVATION ▪ 3: TERRAIN_LAND_CLASS ▪ 4: TERRAIN_WATER_CLASS ▪ 5: TERRAIN_REGION ▪ 6: TERRAIN_SEASON_XXX ▪ 7: POPULATION_DENSITY ▪ 8: N/A ▪ 9: TERRAIN_INDEX
0x0A	1 - BYTE	Compression Type of 1st Section (Values) following this record (Note that only a subset of these appear to be used by the resampler) ▪ 0x0: Not_Compressed ▪ 0x1: LZ1_Compressed ▪ 0x2: Delta_Compressed ▪ 0x3: Delta_And_LZ1_Compressed ▪ 0x4: LZ2_Compressed ▪ 0x5: Delta_And_LZ2_Compressed ▪ 0x6: BitPack_Compressed ▪ 0x7: BitPack_And_LZ1 ▪ 0x8: Solid_Block ▪ 0x9: BitPack_And_LZ2 ▪ 0xA: PTC ▪ 0xB: DXT1 ▪ 0xC: DXT3 ▪ 0xD: DXT5 Note that when a double compression is involved (for example, Delta_And_LZ2_Compressed), you have to decompress in the reverse order (For example, decompress LZ2 and then Delta)
0x0B	1 - BYTE	Compression Type of 2nd Section (Mask) following this record (same as above)
0x0C	4 - DWORD[2]	Same 2 DWORDS as the ones defined at offset 0 of the subsection above.
0x14	4 - DWORD	Month Mask for TERRAIN_PHOTO and TERRAIN_SEASON (0x0000 otherwise) ▪ JAN: 0x0001 ▪ FEB: 0x0002 ▪ MAR: 0x0004 ▪ APR: 0x0008 ▪ MAY: 0x0010 ▪ JUN: 0x0020 ▪ JUL: 0x0040 ▪ AUG: 0x0080 ▪ SEP: 0x0100 ▪ OCT: 0x0200 ▪ NOV: 0x0400 ▪ DEC: 0x0800 ▪ NIGHT: 0x1000
0x18	4 - DWORD	NROWS - Number of raster rows present in chunk (including whitespace)
0x1C	4 - DWORD	NCOLS - Number of raster columns present in chunk (including whitespace)
0x20	4 - DWORD	Size of 1st data chunk (Values) that follows this record
0x24	4 - DWORD	Size of 2nd data chunk (Mask) following this record (that follows the 1st record if size > 0) Not Present for TERRAIN_INDEX sections Note: this section is automatically added by resample after relevant raster sections, does not count toward total block count displayed in the resample statistics, and seems to always be relatively short such as TERRAIN_INDEX sections. It appears to always be compressed with BitPack + LZ1.

The content of the 2 records that follows is still to be determined based off of each applicable compression scheme. Work is underway to decompress each type.

The decompression algorithms works on a square matrix covering each QMID square. The size of each matrix depends on the identifier defined at offset 8, as shown in the table below:

Identifier	Section Type	Matrix Size	Cell Size	Description
1	TERRAIN_PHOTO	256 x 256	WORD	
2	TERRAIN_ELEVATION	257 x 257	DWORD	
3	TERRAIN_LAND_CLASS	257 x 257	BYTE	
4	TERRAIN_WATER_CLASS	257 x 257	BYTE	
5	TERRAIN_REGION	257 x 257	BYTE	
6	TERRAIN_SEASON_XXX	257 x 257	BYTE	
7	POPULATION_DENSITY	257 x 257	BYTE	
9	TERRAIN INDEX	32 x 32	WORD	Each value is a mask indicating what kind of terrain the cell contains. ▪ 0x0001 TerrainVectorDb ▪ 0x0002 TerrainPhotoXXX ▪ 0x0004 TerrainElevation ▪ 0x0008 TerrainLandClass ▪ 0x0010 TerrainWaterClass ▪ 0x0020 TerrainRegion ▪ 0x0080 PopulationDensity ▪ 0x0100 AutogenAnnotation ▪ 0x0200 TerrainIndex ▪ 0x0400 TerrainSeasonXXX

Delta Compressed Segment

Adaptive delta compression is the simplest of the decompression techniques. It simply looks for one of several sentinel values, and the following bytes are read as deltas from the last value, until another sentinel value is seen.

See decompression algorithm in annexe A.

LZ1 (LZ77) Compressed Segment

See decompression algorithm in annexe A.

LZ2 (LZ78) Compressed Segment

See decompression algorithm in annexe A.

Bitpack Compressed Segment

The decompression algorithm is a recursive one. It divides the original matrix until a sufficient size (<= 8) is reached and then applies the decompression to the smaller matrix.
See decompression algorithm in Annexe A.

RCS1 Record

The RCS1 Record is a fixed-size header that contains information about terrain scaling for any section of type TERRAIN_ELEVATION.

Terrain scaling is used to convert the integer terrain values in the BGL data to the floating point values used in the terrain triangulation engine. These values correspond directly to the Fraction Bits settings in the resampler, and is always present with default values if these settings are not specified.

It is not a true subsection in that it does not have a child section, it is merely a data section that is prepended to the compressed chunk.

The RCS1 record has a size of 12 (0xC) bytes for the following structure:

Relative Offset	Number of bytes	Description
0x00	4 - DWORD	Signature = 0x31435352= 'RCS1'
0x04	4 - FLOAT	Scale Value = The scaling multiplier for the integer terrain value (Default: 1.0f) This number is halved for each fractional bit required (FractionBits=1: Scale = 0.5f, FractionBits=2: Scale=0.25)
0x08	4 - FLOAT	Base Value = The base offset to be added to the scaled value (Default: 0) This is exposed as a signed int through resample, but is cast to a float representation in the BGL

Thus, the actual terrain value = (Decompressed integer value * Scale) + Base. When default values are used, the input is simply converted to a floating point number and not modified.

For more information on Fractional Values please see the Scaled Elevation Values section in the Terrain and Scenery SDK document.

INDEXES

The section VorIlsIcaoIndex (0x28), NdbIcaoIndex (0x29), WayPointIcaoindex (0x2A) and TacanIndex (0xA1) [**P3D only**] all share the same structure of 12 bytes:

Relative Offset	Number of bytes	Description
0x00	4 - DWORD	ICAO Identification (Special Format)
0x04	4 - DWORD	Region and Airport Identification Bits 0-10 : Region (Special Format) Bits 11-31 : Airport ICA (Special Format)
0x08	2 - WORD	QMID U Value (Level 9)
0x0C	2 - WORD	QMID V value (Level 9)

Annexe A

Computing the bounding coordinates from a DWORD value

```
public static List<double> GetBoundingCoordinates(uint boundingValue)
{
    var list = new List<double>();
    var shiftValue = 15;
    var work = boundingValue;
    var latitudeData = (uint)0;
    var longitudeData = (uint)0;

    while (work < 0x80000000 && shiftValue >= 0)
    {
        shiftValue--;
        work *= 4;
    }
    work &= 0x7FFFFFFF;    // Remove negative flag, if any
    var powerOfTwo = shiftValue;

    while (shiftValue >= 0)
    {
        if ((work & 0x80000000) != 0)
        {
            latitudeData += (uint)(1 << shiftValue);
        }

        if ((work & 0x40000000) != 0)
        {
            longitudeData += (uint)(1 << shiftValue);
        }
        work *= 4;
        shiftValue--;
    }

    // factor = 1.0 / (2^i)
    var factor = 1.0 / (1 << powerOfTwo);

    // Calc bounding coordinates
    var minLatitudeDeg = 90.0 - ((latitudeData + 1.0) * factor * 360.0);
    var maxLatitudeDeg = 90.0 - (latitudeData * factor * 360.0);
    var minLongitude = (longitudeData * factor * 480.0) - 180.0;
    var maxLongitude = ((longitudeData + 1.0) * factor * 480.0) - 180.0;

    list.Add(minLatitudeDeg);
    list.Add(maxLatitudeDeg);
    list.Add(minLongitude);
```

```
    list.Add(maxLongitude);
    return list;
}
```

Computing Longitude and Latitude from a DWORD value

Latitude and longitude are no longer represented as before. Each location on the earth is fixed in the LOD grid. Longitude and latitude are each represented by a 4 byte value (DWORD). The formula for obtaining the decimal values is as follows:

```
(double) Lon = ((DWORD) Lon * (360.0 / (3 * 0x10000000))) - 180.0
(double) Lat = 90.0 - (DWORD) Lat * (180.0 / (2 * 0x10000000))
```

Pitch, bank and heading

Pitch, bank and heading are given as ANGLE16 in form of a DWORD. The formula for obtaining the decimal value is as follows: (double) Pitch = (DWORD) Pitch * 360.0 / 0x10000

ICAO Identifiers and region codes

ICAO Identifiers and region codes are coded in a special format. Each number and letter has a value from 0 .. 37:

blank	00
Digits 0-9	02-11
Letters A-Z	12-37

Encoding

The code is calculated by starting from left: the value of the first digit/letter is multiplied by 38 (0x26), then the value of the next digit/letter to the right is added, the sum s multiplied by 38 (0x26), and as long as there are more digits/letters this process is repeated. The region codes have only 2 digits/letters and the result is used as such; for the ICAO identifiers for airports, ILS, VOR, NDB and waypoints there are up to 5 digits/letters, and the result is shifted left by 5 positions, i.e. multiplied by 0x20. Bits 0 .. 4 of the resulting DWORD are frequently used for other purposes. The ICAO identifiers for primary and secondary ILS in a runway record are not shifted.

Decoding So you have a DWORD value to be translated in a ICAO string. If the value comes from an airport identifier, it first has to be shifted 5 bits to the right. The pseudo-algorithm looks like this

```
if (values from airport data)
{
   shift value 5 bits to the right
}

while (value > 37)
{
   oneCodedChar = value % 38
   prepend OneCodedChar to the list
   value = (value - oneCodedChar) / 38
   if (value < 38)
   {
      oneCodedChar = value
   }
}

// The first coded char in the list is the last one computed in the while loop

foreach (oneCodedChar in list)
{
   if (oneCodedChar == 0)
   {
      output space char
   }
   else if (oneCodedChar > 1 && oneCodedChar < 12)
   {
      // digit 0-9
      output '0' + (oneCodedChar - 2)
   }
   else
   {
      // letter
      output 'A' + (oneCodedChar - 12)
   }
}
```

Example:
The value 0x0257C221 comes from an airport record.
0x0257C221 is first shifted 5 bits to the right , which gives 0x0012BE11 = 1228305
- 1228305 is >= 38 so 1228305 % 38 = **31** and (1228305 - 31)/ 38 = 32323

- 32323 is >= 38 so 32323 % 38 = **23** and (32323 - 23) / 38 = 850
- 850 >= 38 so 850 % 38 = **14** and (850 - 14) / 38 = 22
- **22** is < 38 so that is the last value.
So we got 31,23,14 and 22
- 22 is in the range [12 - 37] so letter = 'A' + (22-12) = 'K'
- 14 is in the range [12 - 37] so letter = 'A' + (14-12) = 'C'
- 23 is in the range [12 - 37] so letter = 'A' + (14-23) = 'L'
- 31 is in the range [12 - 37] so letter = 'A' + (31-12) = 'T'
So the ICAO code is KCLT

Computing QMID u and v based on level and coordinates

Whenever you move your mouse in the TmfViewer application, the u and v values of the QMID (Quad Mesh IDentifier) are updated in the status bar at the bottom of the screen. All coordinates (longitude , latitude) inside the same QMID (or square) have the same u and v values for a specific level. The greater the level, the smaller the square. See Microsoft: Terrain and Scenery (http://msdn.microsoft.com/en-ca/library/cc707102.aspx#QMIDandLODValues).

The algorithm to compute the u and v values is the following:

```
Input:
 - Longitude in degrees
 - Latitude in degrees
 - Level (QMID Level = LOD + 2) in the range [2..29]

LongitudeData = INT(0.5 + (180 + LongitudeDeg) * (0x2000000 / 15))
LatitudeData = INT(0.5 + (90 - LatitudeDeg) * (0x8000000 / 45))

If LongitudeData > 0x30000000
        LongitudeData -= 0x30000000
If LongitudeData < 0
        LongitudeData += 0x30000000

If LatitudeData > 0x20000000
        LatitudeData -= 0x20000000
If LatitudeData < 0
        LatitudeData += 0x20000000

n = 30 - QMIDLevel
QMID.u = LongitudeData >> n
QMID.v = LatitudeData >> n
QMID.l = level
```

Getting DWORD values from QMID

The algorithm takes the QMID data (u, v and l) as input and produces 2 DWORD values as output.

The **u** (longitude-related), **v** (latitude-related) and **l** (level) values are handled as DWORD (4 bytes).

The DWORD value **u** is made of 4 bytes: U_3, U_2, U_1 and U_0 where U_3 is the most significant byte and U_0 the least significant.
The DWORD value **v** is made of 4 bytes: V_3, V_2, V_1 and V_0 where V_3 is the most significant byte and V_0 the least significant.

Let's compute U'_3, U'_2, U'_1, U'_0, V'_3, V'_2, V'_1, V'_0 where:

- $U'_i = f(U_i)$
- $V'_i = f(V_i)$

The fonction f(x) decomposes the input value x (a byte) in base-4:
$x = a_3.4^3 + a_2.4^2 + a_1.4^1 + a_0.4^0 = a_3.64 + a_2.16 + a_1.4 + a_0$

Then for each a_i, we compute the corresponding b_i:

- $a_i = 0 \Rightarrow b_i = 0$
- $a_i = 1 \Rightarrow b_i = 1$
- $a_i = 2 \Rightarrow b_i = 4$
- $a_i = 3 \Rightarrow b_i = 5$

where b_i are the coefficients of the output value in base-16.

The output value y = f(x) is computed as follows:
$y = b_3.4096 + b_2.256 + b_1.16 + b_0$

For example, f(0x4B) gives:
$a_3 = 1$, $a_2 = 0$, $a_1 = 2$, $a_0 = 3$ (0x4B = 75 = 1 x 64 + 0 x 16 + 2 x 4 + 3)
So we have $b_3 = 1$, $b_2 = 0$, $b_1 = 4$, $b_0 = 5$
and the output value is: 1 x 4096 + 0 x 256 + 4 x 16 + 5 = 4165 = 0x1045

We now know how to compute U'_i and V'_i.

Let's have:
A = (2 << (2 * Level)) + 2 * (V'_1 * 65536 + V'_0) + (U'_1 * 65536 + U'_0)
B = 2 * (V'_3 * 65536 + V'_2) + (U'_3 * 65536 + U'_2)

where **A** and **B** are the 2 output DWORD values.
The value **A** goes at offset 0 of the subsection.
The value **B** goes at offset 4 of the subsection.

Note: Because of the `(2 << (2 * Level))` part and because the A is a DWORD, the maximum QMID level is 15.

Example
Lets' take our previous example with QMID (13,1819,1012).

1819 = 0x71B and 1012 = 0x3F4.
so U_0 = 0x1B, U_1 = 0x07, V_0 = 0xF4 and V_1 = 0x03. Other values are 0x00.
The conversion gives:
U'_0 = 0x145, U'_1 = 0x15, V'_0 = 0x5510 and V'_1 = 0x05. Other values are 0x00.
2 << (2 * Level) = 2 << 26 = 0x8000000.

So **A** = 0x8000000 + 2 *(0x00055510) + 0x00150145 = **0x81FAB65**
and **B** = **0x00**

Getting QMID from DWORD values

The algorithm takes 2 DWORD values (A and B) as input and produces the QMID data (u, v and l) as output.
For the header and some subsections, the 2nd DWORD value (B) is zero.

The **level** value can be deduced from the A value since for each level there is a maximum value of u and v and hence a maximum value of A. (The minimum value is 0)
$U_{max} = 3 * 2^{Level-2} - 1$
$V_{max} = 2^{Level-1} - 1$

Level	U_{max}	V_{max}	A_{min}	A_{max}
0	0	0	0x02	0x02
1	0	0	0x08	0x08
2	2	1	0x20	0x26
3	5	3	0x80	0x9B
4	11	7	0x200	0x26F
5	23	15	0x800	0x9BF
6	47	31	0x2000	0x26FF
7	95	63	0x8000	0x9BFF
8	191	127	0x20000	0x26FFF
9	383	255	0x80000	0x9BFFF
10	767	511	0x200000	0x26FFFF
11	1535	1023	0x800000	0x9BFFFF
12	3071	2047	0x2000000	0x26FFFFF
13	6143	4095	0x8000000	0x9BFFFFF
14	12287	8191	0x20000000	0x26FFFFFF
15	24575	16383	0x80000000	0x9BFFFFFF

The level can be deduced from A by finding the range where $A_{min} \leq A \leq A_{max}$. Note that A_{min} = (2 << (2 * Level).

We have 2 equations:

- A - (2<< (2 * Level) = 2 * (V'_1 * 65536 + V'_0) + (U'_1 * 65536 + U'_0)
- B = 2 * (V'_3 * 65536 + V'_2) + U'_2 + (U'_3 * 65536)

Both equations are in the form : $\Sigma = 2 * \beta + \alpha$

So how do we differentiate α and β ?

The trick is to remember that the U' and V' values were generated using only the 0,1,4 and 5 digits.
The table below shows you what happens when you add these digits using $\Sigma = 2 * \beta + \alpha$

		α			
		0	1	4	5
β	2 * β				
0	0	0	1	4	5
1	2	2	3	6	7
4	8	8	9	12	13
5	10	10	11	14	15

So any addition in the form of $\Sigma = 2 * \beta + \alpha$ yields a unique result in the range [0..15]. Knowing the result Σ, we are able to retrieve α and β.

We have the equation A - (2<< (2 * Level) = 2 * V'+ U'. (bytes 0 and 1)
Hence if we take each byte of this value, we are able to retrieve each b_i of U' (α) and V' (β).

Same thing for the equation B = 2 * V' + U' (bytes 2 and 3).

Once we have the b_i, we can deduce the a_i values and restore the original U_i and V_i values.

Note: You may want to have an array of precomputed values if you are using this algorithm intensively.

Example
Lets' take our previous example with A = 0x81FAB65 and B = 0.

Since B = 0 we already know that $U_3 = U_2 = V_3 = V_2 = 0$.
0x8000000 ≤ 0x81FAB65 ≤ 0x9BFFFFF so **Level = 13**

A - (2<< (2 * Level) = 0x1FAB65
The hexadecimal digit A (10d) can only be obtained when α = 5 and β = 0 (see table above).

	$U'_1 + 2 * V'_0$				$U'_0 + 2 * V'_0$			
A ⇒	**0**	**0**	**1**	**F**	**A**	**B**	**6**	**5**
$u_i = \beta$	0	0	1	5	0	1	4	5
$v_i = \alpha$	0	0	0	5	5	5	1	0
					b_3	b_2	b_1	b_0

So for U'_0, we have:

- $b_3 = 0 \Rightarrow a_3 = 0$
- $b_2 = 1 \Rightarrow a_2 = 1$
- $b_1 = 4 \Rightarrow a_1 = 2$
- $b_0 = 5 \Rightarrow a_0 = 3$

$\Rightarrow U_0 = 0 * 4^3 + 1 * 4^2 + 2 * 4^1$ 0 1 $* 3^0 = 27d = 0x1B$

For U'_1, we have:

- $b_3 = 0 \Rightarrow a_3 = 0$
- $b_2 = 0 \Rightarrow a_2 = 0$

- $b_1 = 1 \Rightarrow a_1 = 1$
- $b_0 = 5 \Rightarrow a_0 = 3$

$\Rightarrow U_1 = 0 * 4^3 + 0 * 4^2 + 1 * 4^1\ 0\ 1 * 3^0 = 7d = 0x07$
so U = 0x071B

You got the idea: it's the same for V.
A simpler algorithm can be:

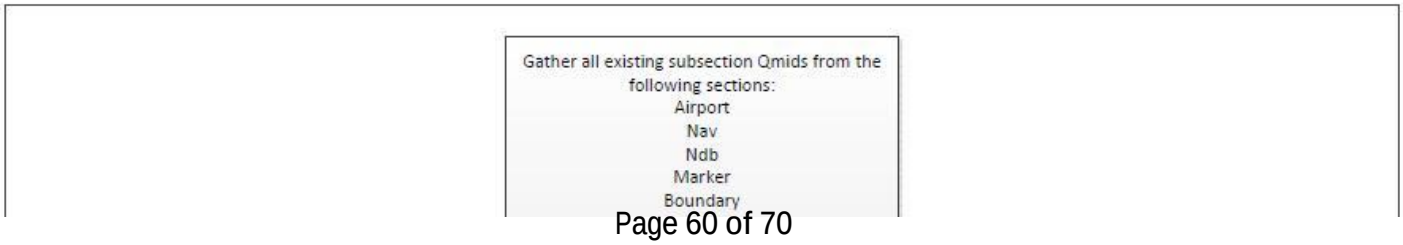
```
public static Qmid CalcQmidFromDwords(UInt32 dwordA, UInt32 dwordB)
{
        var v = 0;
        var u = 0;
        var cnt = 0x1F;
        var workDwordA = dwordA;
        var workDwordB = dwordB;


        while (cnt > 0 && (workDwordB & 0x80000000) == 0)
        {
                workDwordB <<= 2;
                workDwordB += (workDwordA & 0xC0000000) >> 30;

                workDwordA += workDwordA;
                workDwordA += workDwordA;
                cnt--;
        }

        workDwordB &= 0x7FFFFFFF;
        var level = cnt;

        while (cnt >= 0)
        {
                if ((workDwordB & 0x80000000) != 0)
                {
                        v += (1 << cnt);
                }

                if ((workDwordB & 0x40000000) != 0)
                {
                        u += (1 << cnt);
                }

                workDwordB <<= 2;
                workDwordB += (workDwordA & 0xC0000000) >> 30;
                workDwordA += workDwordA;
                workDwordA += workDwordA;
                cnt--;
        }

        return new Qmid(u, v, level);
}
```

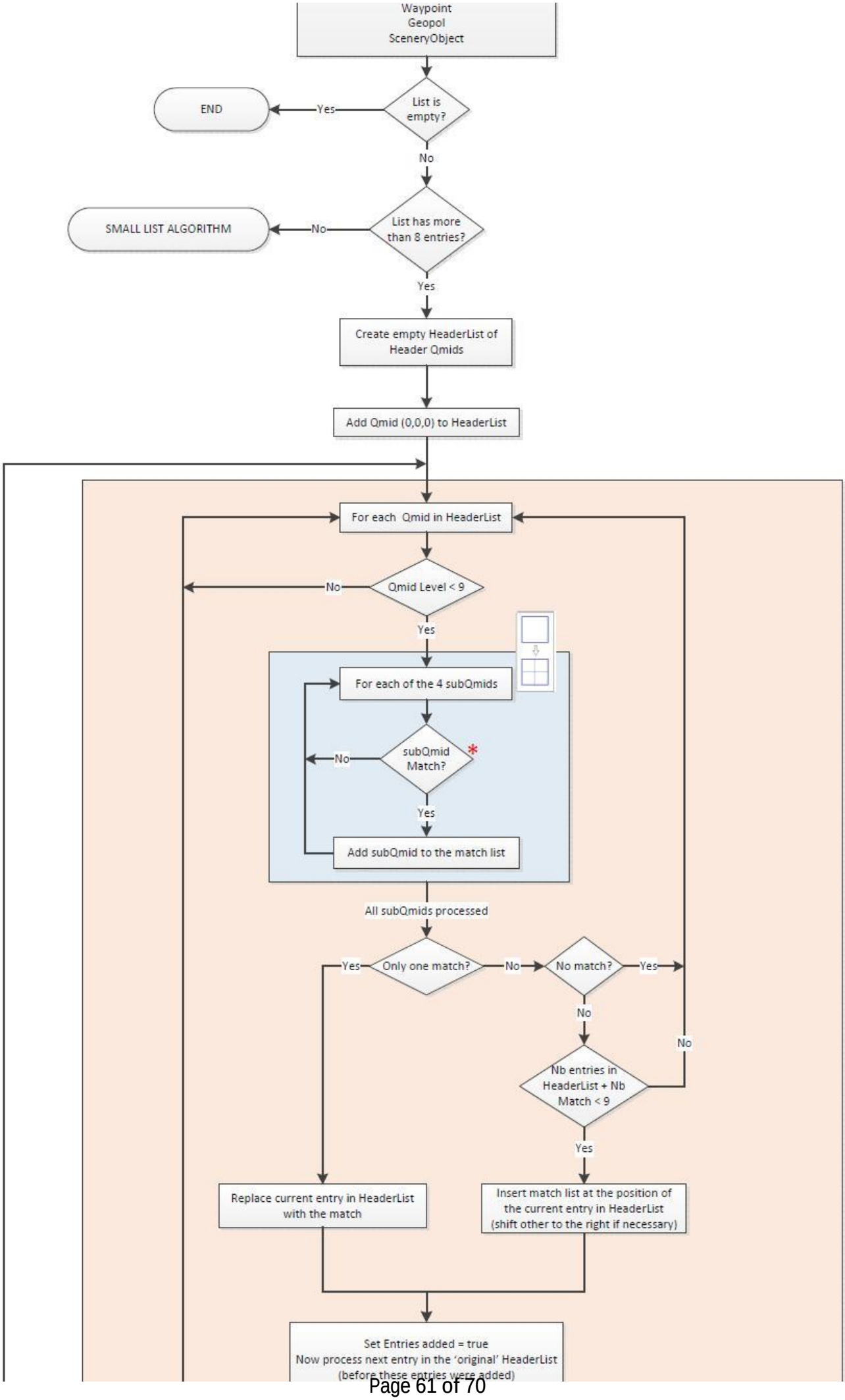
How are the header QMIDs computed?

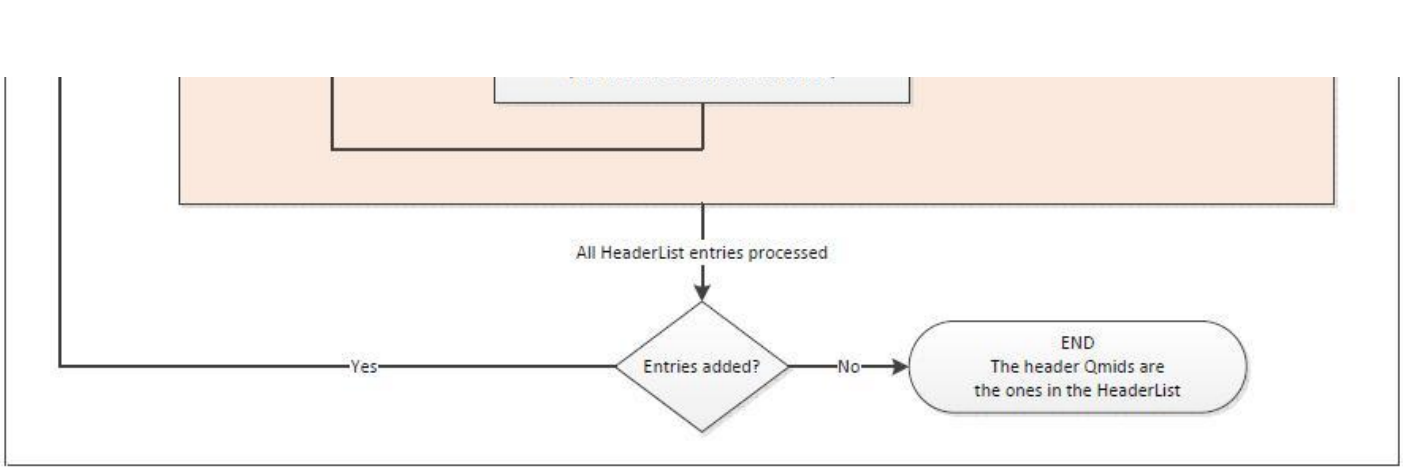
The following algorithms apply only to APX*.bgl (Airport), ATX*.bgl (Waypoint), BRX*.bgl (Extrusion Bridge), NVX*.bgl (Nav), OBX*.bgl (SceneryObject) and BNX*.bgl (Boundary) files.
For these files, the maximum level value is 9.
To compute the QMID values that will be stored in the header, one must gather all the subsection's QMID values for the following sections:

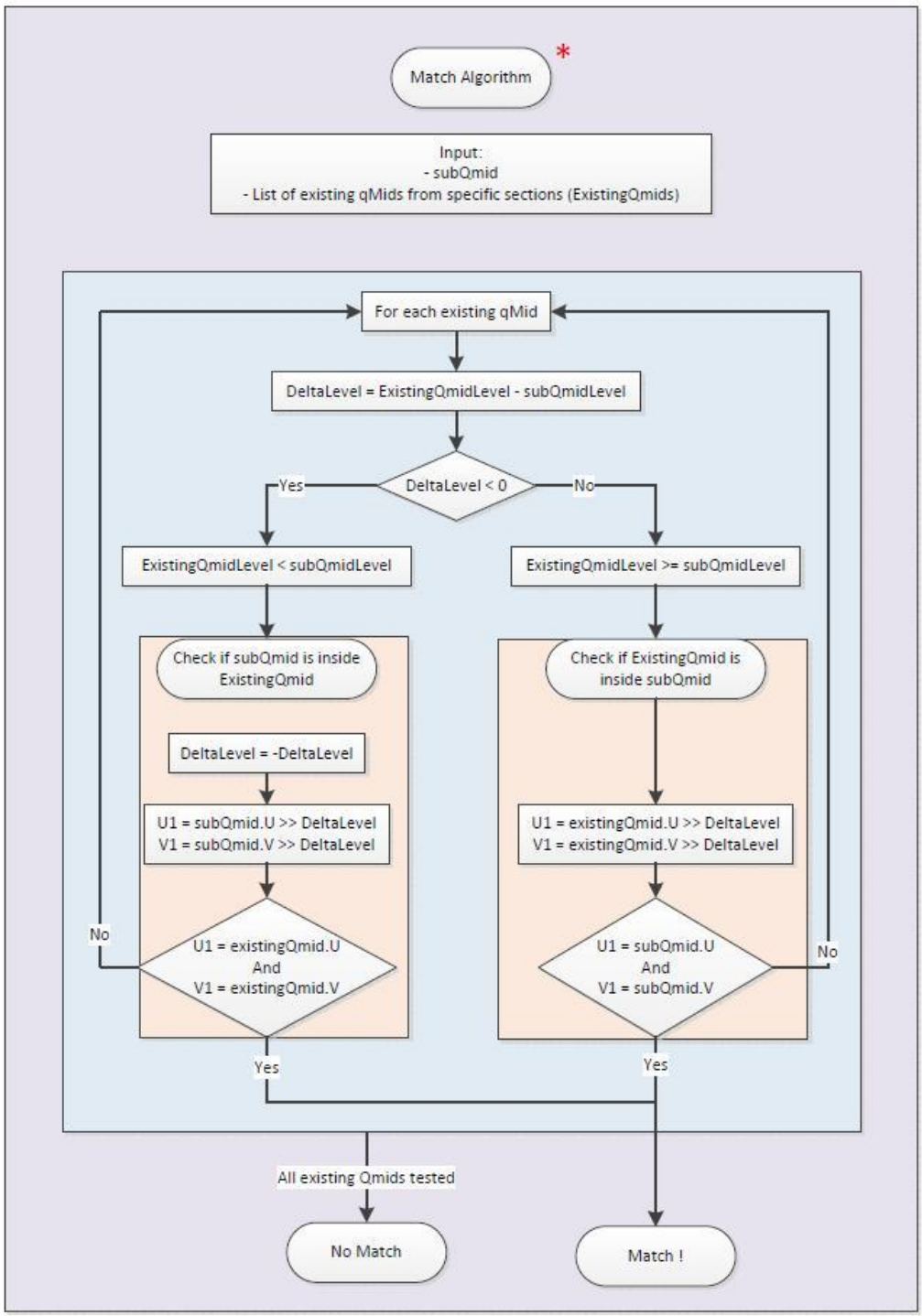
- Airport
- Nav
- Ndb
- Marker
- Boundary
- Waypoint
- Geopol
- SceneryObject

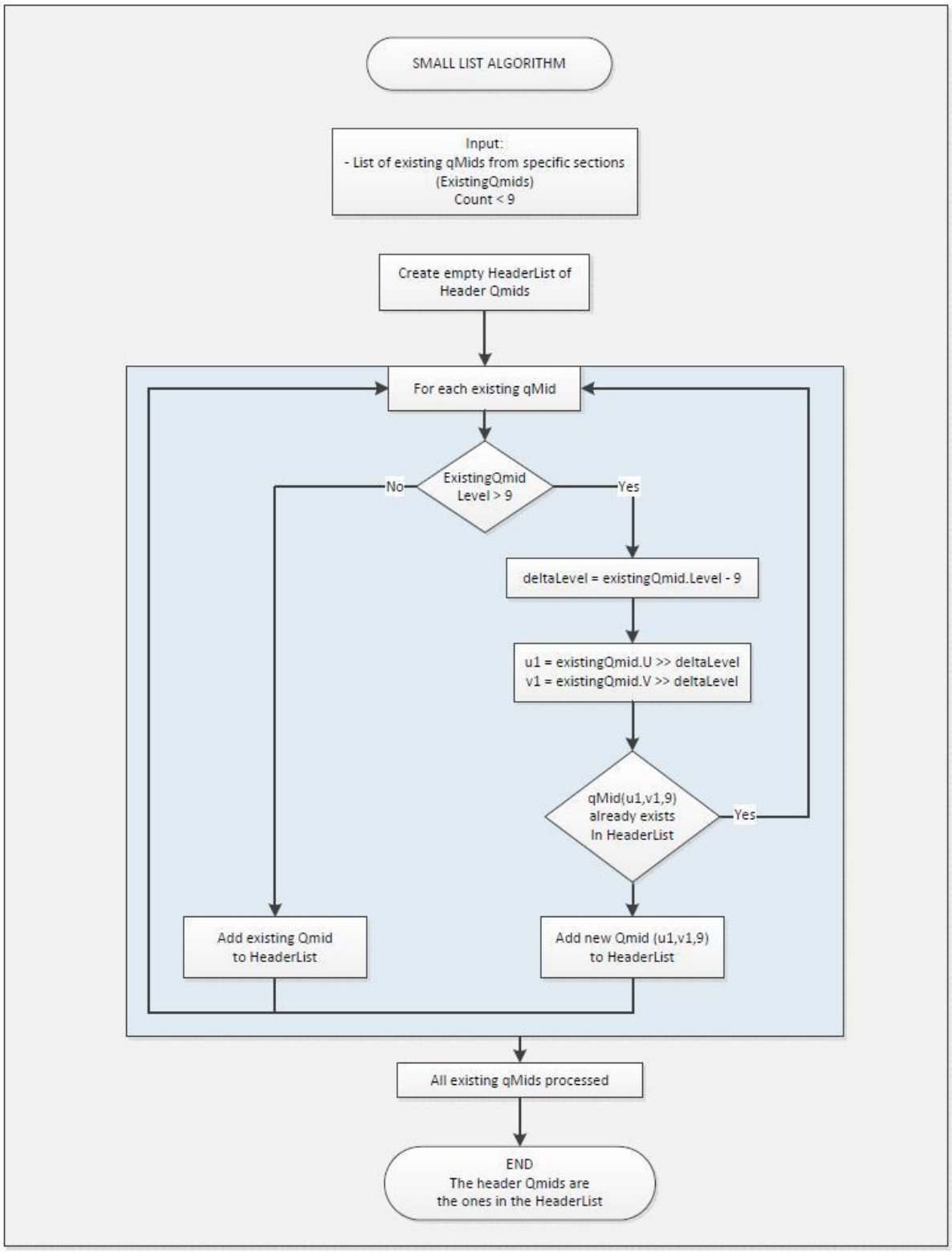
Then the computation flow is :




Waypoint
Geopol
SceneryObject
List is empty?
Yes
END
No
List has more than 8 entries?
No
SMALL LIST ALGORITHM
Yes
Create empty HeaderList of Header Qmids
Add Qmid (0,0,0) to HeaderList
For each Qmid in HeaderList
Qmid Level < 9
No
Yes
For each of the 4 subQmids
subQmid Match?
*
No
Yes
Add subQmid to the match list
All subQmids processed
Only one match?
Yes
No
No match?
Yes
No
Nb entries in HeaderList + Nb Match < 9
No
Yes
Replace current entry in HeaderList with the match
Insert match list at the position of the current entry in HeaderList (shift other to the right if necessary)
Set Entries added = true
Now process next entry in the 'original' HeaderList
(before these entries were added)


All HeaderList entries processed
Entries added?
Yes
No
END
The header Qmids are
the ones in the HeaderList


Match Algorithm
*
Input:
- subQmid
- List of existing qMids from specific sections (ExistingQmids)
For each existing qMid
DeltaLevel = ExistingQmidLevel - subQmidLevel
DeltaLevel < 0
Yes
No
ExistingQmidLevel < subQmidLevel
ExistingQmidLevel >= subQmidLevel
Check if subQmid is inside ExistingQmid
DeltaLevel = -DeltaLevel
U1 = subQmid.U >> DeltaLevel
V1 = subQmid.V >> DeltaLevel
U1 = existingQmid.U
And
V1 = existingQmid.V
No
Yes
Check if ExistingQmid is inside subQmid
U1 = existingQmid.U >> DeltaLevel
V1 = existingQmid.V >> DeltaLevel
U1 = subQmid.U
And
V1 = subQmid.V
No
Yes
All existing Qmids tested
No Match
Match !



Delta Decompression Algorithm

```
public List<byte> DeltaDecompress(byte[] source, int destinationSize)
{
    var output = new List<byte>();
    var sourceIndex = 0;

    if ((destinationSize % 2) == 1)
    {
        // Destination size is odd
        output.Add(source[0]);
        sourceIndex++;
        destinationSize--;
    }

    if (destinationSize > 0)
    {
        var curSourceValue = BitConverter.ToUInt16(source, sourceIndex);    // Read WORD from source buffer

        // Copy WORD to destination buffer
```

```
        output.Add(source[sourceIndex]);
        output.Add(source[sourceIndex + 1]);
        sourceIndex += 2;

        var cnt = (destinationSize >> 1) - 1;
        if (cnt != 0)
        {
            for (;;)
            {
                UInt16 addedValue;
                if (source[sourceIndex] == 0x80)
                {
                    addedValue = BitConverter.ToUInt16(source, sourceIndex + 1);
                    sourceIndex += 3;
                }
                else if (source[sourceIndex] == 0x81)
                {
                    addedValue = (UInt16) (curSourceValue - source[sourceIndex + 1] - 0x7E);
                    sourceIndex += 2;
                }
                else if (source[sourceIndex] == 0x82)
                {
                    addedValue = (UInt16) (curSourceValue + source[sourceIndex + 1] + 0x80);
                    sourceIndex += 2;
                }
                else
                {
                    if (source[sourceIndex] > 0x7F)
                    {
                        addedValue = (UInt16)(curSourceValue + (UInt16)(source[sourceIndex] + 0xFF00));
                    }
                    else
                    {
                        addedValue = (UInt16)(curSourceValue + source[sourceIndex]);
                    }
                    sourceIndex++;
                }
                output.AddRange(BitConverter.GetBytes(addedValue));

                curSourceValue = addedValue;
                cnt--;

                if (0 == cnt)
                {
                    return output;
                }
            }
        }
    }
    return output ;
}
```

LZ1 (LZ77) Decompression Algorithm

```
Algorithm:

while output count < expected output size
{
    flag = Read 2 bits ; // possible values : 0,1,2,3

    if (flag == 1)
    {
        output the next 7 bits (as byte)  + 0x80
    }
    else if (flag == 2)
    {
        output the next 7 bits (as byte)
    }
    else
    {
        // flag = 0 or 3
        // This is a sequence already stored in the output buffer
        // Retrieve the offset of where this sequence is stored.

        if (flag == 0)
        {
            existingSequenceOffset = Read next 6 bits
        }
        else
        {
            if (next bit == 0)
            {
                existingSequenceOffset = 0x40 + Read next 8 bits
            }
            else
            {
                existingSequenceOffset = 0x140 + Read next 12 bits
            }

            if (existingSequenceOffset != 0x113F)
            {
                // Now get existing sequence length
                var nbBitsToRead = 0;
                while (0 == read next bit)
                {
                    nbBitsToRead++;
                }
```

```
                if (nbBitsToRead == 0)
                {
                    sequenceLength = 2;
                }
                else
                {
                    sequenceLength = (read next <nbBitsToRead> bits) + 1 + (2 ^ nbBitsToRead)
                }
                backward output <sequenceLength> bytes from existingSequenceOffset
            }
        }
    }
}

Reading bits:
  Bits are read from the bits pool from right to left (<-)
  When there is not enough bits in the bits pool, data is read from the source buffer and each byte is prepended (to the left) to the bits pool

Backward output:
  nbBytesToCopy = sequenceLength
  index = output count - existingSequenceOffset

  while (nbBytesToCopy > 0)
  {
      output byte at outputBuffer[index]
      index++;
      nbBytesToCopy--;
  }
```

LZ2 (LZ78) Decompression Algorithm

```
Algorithm:

while output count < expected output size
{
  if (ReadNextBit = 0)
  {
      output the next 7 bits (as byte)
  }
  else
  {
      if (ReadNextBit != 0)
      {
          output the next 7 bits (as byte)  + 0x80
      }
      else
      {
          // Read existingSequenceOffset value
          if (ReadNextBit = 0)
          {
              existingSequenceOffset = read next 6 bits
          }
          else
          {
              if (ReadNextBit = 0)
              {
                  existingSequenceOffset = (read next 8 bits) + 0x40
              }
              else
              {
                  existingSequenceOffset = (read next 12 bits) + 0x140
              }
          }

          if (existingSequenceOffset != 0x113F)
          {
              nbBitsToGet = 0
              while (ReadNextBit = 0)
              {
                  nbBitsToGet++;
              }
              if (nbBitsToGet != 0)
              {
                  sequenceLength = (ReadNext <nbBitsToGet> bits) + (2 ^ nbBitsToGet) + 2
              }
              else
              {
                  sequenceLength = 3 ;
              }
              backward output <sequenceLength> bytes from existingSequenceOffset
          }
      }
   }
}


Reading bits:
  Bits are read from the bits pool from right to left (<-)
  When there is not enough bits in the bits pool, data is read from the source buffer and each byte is prepended (to the left) to the bits pool

Backward output:
  nbBytesToCopy = sequenceLength
  index = output count - existingSequenceOffset

  while (nbBytesToCopy > 0)
```

```
while (nbBytesToCopy > 0)
{
    output byte at outputBuffer[index]
    index++;
    nbBytesToCopy--;
}
```

Bitpack Decompression Algorithm

The decompression algorithm is a recursive one. It divides the original matrix until a sufficient size (<= 8) is reached and then applies the decompression to the smaller matrix.

Note : Bits are read from the LSB to the MSB of the byte (from right to left)

First, 5 values are read. These values are the parameters of the decompression algorithm:

Parameter	Size in bits
nbBytesForInitialAddValue	8 (1 byte)
nbShifts	8 (1 byte)
initialAddValue	8 x nbBytesForInitialAddValue (nbBytesForInitialAddValue bytes)
nbBitsToGetPerData	4
maxBitsPerValueRead	4

At this point (3 + nbBytesForInitialAddValue) have been read. Then the matrix, as defined in the TRQ1 Record is divided in 16 parts:

- the rows are divided by 4. If the number of rows is not a multiple a 4, the last part is the biggest (for example, 257 will give 64,64,64 and 65).
- the columns are divided by 4 (same as above)

```
foreach (rowpart)    /* 4 times */
{
    foreach (columnpart)   /* 4 times */
    {
        populateDecompressedBuffer(rowPart,columnPart,parameters)
    }
}
```

This piece of code may be called recursively (each time, the matrix being subdivided into 16 smaller parts) until a correct size (<= 8) is reached.

See C# code below:

```
internal List<byte> Decompress(byte[] record, int outputSize, TerrainMatrixParam matrixParams)
{
   var nbRows = matrixParams.NbRows;
   var nbColumns = matrixParams.NbColumns;
   var targetIndex = 0;

   _sourceBuffer = record;
   _sourceIndex = 0;
   _nbRead = 0;
   _nbBitsNotProcessed = 0;
   _outputBuffer = new byte[outputSize];

   if (null == record || 0 == record.Length)
   {
      return null;
   }

   _nbRemainingBits = 8 * record.Length;
   _nbBitsNotProcessed = 8;

   Int32 nbBytesForInitialAddValue;
   var b = getNext_N_SourceBits(8, out nbBytesForInitialAddValue);

   Int32 nbShifts;
   b &= getNext_N_SourceBits(8, out nbShifts);

   Int32 initialAddValue;
   b &= getNext_N_SourceBits(8 * nbBytesForInitialAddValue, out initialAddValue);

   Int32 nbBitsToGetPerData;
   b &= getNext_N_SourceBits(4, out nbBitsToGetPerData);

   Int32 maxBitsPerValueRead;
   b &= getNext_N_SourceBits(4, out maxBitsPerValueRead);

   if (maxBitsPerValueRead == 0)
   {
```

```
      maxBitsPerValueRead = 16;
   }

   var nbRowsPerRowIteration = nbRows / 4;
   var nbColumnsPerColumnIteration = nbColumns/4;
   var bytesInterval = nbRowsPerRowIteration * nbColumns;

   var nbColumnsForLastColumnIteration = (nbColumns % 4);
   var nbRowsForLastRowIteration = (nbRows % 4);

   for (var rowIteration = 0; rowIteration < 4; rowIteration++)
   {
      var nbRowsToProcess = nbRowsPerRowIteration;
      if (rowIteration == 3)
      {
         // Last row iteration
         nbRowsToProcess += nbRowsForLastRowIteration;
      }

      for (var columnIteration = 0; columnIteration < 4; columnIteration++)
      {
         var copyCountPerRow = nbColumnsPerColumnIteration;
         if (columnIteration == 3)
         {
            // Last Column Iteration
            copyCountPerRow += nbColumnsForLastColumnIteration ;
         }

         b &= populateDecompressedBuffer(targetIndex + columnIteration * (nbColumns / 4), nbColumns, initialAddValue,
                                         nbBitsToGetPerData, copyCountPerRow, nbRowsToProcess, nbShifts, maxBitsPerValueRead);
      }
      targetIndex += bytesInterval;
   }
   if (_nbBitsNotProcessed != 8)
   {
      _nbRead++;
   }
   if (_nbRead != record.Length)
   {
      return null;
   }

   return _outputBuffer.ToList();
}

private bool populateDecompressedBuffer(int startRowCopyIndex, int nbColumnsPerRow, int addValue, int nbBitsToGetPerData, int nbBytesToOutputPerRow, int nb
{
   int copyData, nbBitsPerValueRead;

   // Read copyData
   var b = getNext_N_SourceBits(Math.Min(nbBitsToGetPerData, 8), out copyData);

   // Read nbBitsPerValueRead
   // if 0, will copy value as is
   b &= getNext_N_SourceBits(4, out nbBitsPerValueRead);

   var nbAdditionalShift = (nbBitsToGetPerData <= 8) ? 0 : nbBitsToGetPerData - 8;

   var valueToCopy = (copyData << ((nbShifts + nbAdditionalShift) & 0xFF)) + addValue;

   if (nbBitsPerValueRead == 0)
   {
      //-------------------------------
      // Identical value to be repeated
      //-------------------------------
      targetSetIdenticalValue(startRowCopyIndex, (byte)valueToCopy, nbRowsToProcess, nbColumnsPerRow, nbBytesToOutputPerRow);
      return b;
   }

   // Copy will handle only blocks 8 x 8 otherwise recursive call
   if (nbBytesToOutputPerRow < 8 || nbRowsToProcess < 8)
   {
      if (nbBitsPerValueRead > maxBitsPerValueRead)
      {
         nbBitsPerValueRead = maxBitsPerValueRead;
      }
      //-------------------
      // Values Read And Set
      //-------------------
      b &= targetReadAndSetValue(startRowCopyIndex, nbRowsToProcess,nbColumnsPerRow, nbBytesToOutputPerRow, valueToCopy, nbBitsPerValueRead, nbShifts);
      return b;
   }

   //-------------------------------------------
   // RECURSIVE CALL for a square 4 times smaller
   //-------------------------------------------
   var nbRowsPerRowIteration = nbRowsToProcess / 4;
   var nbRowsForLastRowIteration = nbRowsToProcess % 4;

   var bytesInterval = nbRowsPerRowIteration * nbColumnsPerRow;

   var nbColumnsPerColumnIteration = nbBytesToOutputPerRow / 4;
   var nbColumnsForLastColumnIteration = nbBytesToOutputPerRow % 4;

   for (var rowIteration = 0; rowIteration < 4; rowIteration++)
   {
      nbRowsToProcess = nbRowsPerRowIteration;
      if (rowIteration == 3)
      {
         // Last row iteration
         nbRowsToProcess += nbRowsForLastRowIteration;
```

```
      }

      var rowStartIndex = startRowCopyIndex;
      for (var columnIteration = 0; columnIteration < 4; columnIteration++)
      {
         nbBytesToOutputPerRow = nbColumnsPerColumnIteration;
         if (columnIteration == 3)
         {
            nbBytesToOutputPerRow += nbColumnsForLastColumnIteration;
         }
         b &= populateDecompressedBuffer(rowStartIndex, nbColumnsPerRow, valueToCopy, nbBitsPerValueRead,
                                         nbBytesToOutputPerRow, nbRowsToProcess, nbShifts, maxBitsPerValueRead);

         rowStartIndex += nbColumnsPerColumnIteration;     // Next row
      }

      startRowCopyIndex += bytesInterval;
   }
   return b;
}

private void targetSetIdenticalValue(int startRowCopyIndex, byte valueToCopy, int nbRowsToProcess, int nbColumnsPerRow, int nbRepeatPerRow)
{
   if (nbRepeatPerRow > 0)
   {
      var srclist = new List<byte>(nbRepeatPerRow);
      srclist.AddRange(Enumerable.Repeat(valueToCopy, nbRepeatPerRow));
      var srcArray = srclist.ToArray();
      while (nbRowsToProcess > 0)
      {
         // copy <valueToCopy> <cnt> times starting at the current position (current row)
         Buffer.BlockCopy(srcArray, 0, _outputBuffer, startRowCopyIndex, nbRepeatPerRow);

         nbRowsToProcess--;
         startRowCopyIndex += nbColumnsPerRow;
      }
   }
}

private bool targetReadAndSetValue(int startRowCopyIndex, int nbRowsToProcess, int nbColumnsPerRow, int nbBytesToOutputPerRow, int addValue, int nbBitsPerR
{
   // nbBitsPerRead = nb bits per value to read
   // nbShiftsLeft = number of times to shift the read value to the left before adding the <addValue>
   if (nbRowsToProcess < 0)
   {
      return true;
   }

   for (var i = 0; i < nbRowsToProcess; i++, startRowCopyIndex += nbColumnsPerRow)
   {
      for (var j = 0; j < nbBytesToOutputPerRow; j++)
      {
         Int32 srcValue;
         if (false == getNext_N_SourceBits (nbBitsPerRead, out srcValue))
         {
            return false;
         }

         var byteValue = (byte)(((srcValue << nbShiftsLeft) + addValue) & 0xFF);

         // copy AL to output buffer
         _outputBuffer[startRowCopyIndex + j] = byteValue;
      }
   }
   return true;
}

private bool getNext_N_SourceBits(int nbBitsToGet, out Int32 retValue)
{
   // Read always the LSB bits first
   retValue = 0;

   if (_nbRemainingBits < nbBitsToGet)
   {
      return false;
   }

   if (nbBitsToGet > 0)
   {
      _nbRemainingBits -= nbBitsToGet;

      var nbRightShift = 8 - _nbBitsNotProcessed;

      if (_nbBitsNotProcessed <= nbBitsToGet)
      {
         retValue = _sourceBuffer[_sourceIndex] >> nbRightShift;

         nbRightShift = _nbBitsNotProcessed; // keep for later
         _sourceIndex++;
         _nbRead++;

         var nbBitsStillToGet = nbBitsToGet - _nbBitsNotProcessed;
         _nbBitsNotProcessed = 8;

         if (nbBitsStillToGet <= 0)
         {
            return true;
         }

         while (nbBitsStillToGet > 0)
         {
```

```
            var sourceByteValue = _sourceBuffer[_sourceIndex];

            if (nbBitsStillToGet < 8)
            {
               retValue |= ((sourceByteValue & (1 << nbBitsStillToGet) - 1) << nbRightShift);
               _nbBitsNotProcessed -= nbBitsStillToGet;
               return true;
            }
            sourceByteValue >>= nbRightShift;
            retValue |= sourceByteValue;
            _sourceIndex++;
            _nbRead++;
            nbBitsStillToGet -= _nbBitsNotProcessed; // ESI = nbBitsStillToGet ?
            nbRightShift += _nbBitsNotProcessed;
            _nbBitsNotProcessed = 8;
         }
         return true;
      }
      retValue = (_sourceBuffer[_sourceIndex] >> nbRightShift) & ((1 << nbBitsToGet) - 1);
      _nbBitsNotProcessed -= nbBitsToGet;
      return true;
   }
   return true;
}
```

Retrieved from ‘http://www.fsdeveloper.com/wiki/index.php?title=BGL_File_Format&oldid=10595’
