Guildhall Convex Scene (GHCS)

File Format Specification (SMU Guildhall, SD4, Cohort 28)

*Prof. Eiserloh & Cohort 28 Coders, Feb 23, 2020*

# Overview

.ghcs is a custom flexible binary file format used to save and load 2D convex object “scenes” (levels). The format is uniform and compatible/interchangeable amongst all students, such that everyone can save & load each other’s scenes. However, it also supports optional “chunks” of binary data for extended scene metadata, such as the structure of a BSP tree built and saved by one student’s demo (but perhaps ignored by another student’s demo, which uses symmetric quadtrees instead).

Throughout this document, red text is recently added or changed (and approved), and magenta text is tentative / work-in-progress (not yet set in stone). Blue text is used in a variety of context-specific ways (e.g. chunk header data vs. chunk private data).

# Conventions

### Structure

* File Header (12 bytes)
* Chunk: ??? (optional) // at least one “canonical” chunk type required; all canonical chunks are first
* Chunk: ??? (optional)
* Chunk: ??? (optional) // any number of optional chunks may follow
* Chunk: ??? (optional)
* Table of Contents

The File Header, Table of Contents, and each Sub-Header (at the start of each chunk) all have formats prescribed precisely below; the internal data contents of each different chunk type vary but are also provided below (at the end of this document).

Maximum total file/buffer size is 4GB (0xFFFFFFFF == 4,294,967,295 bytes). The maximum size of any chunk is also 4GB, less space taken by required components (Header, Table of Contents), although in practice chunks (and the file in total) will typically be much smaller.

### Dirtiness

All chunks in the file are assumed to be valid (unless marked as type 0xFF=invalid). All chunks must describe the same convex scene. They must always agree in terms of the scene objects themselves, which are given in a consistent order, and without gaps. Thus object [4] in the ConvexPolys chunk must always reference the same object as object [4] in the ConvexHulls chunk, etc.

Objects do not have explicit IDs, but metadata (non-canonical) chunks may assume the above paragraph is true and therefore refer to objects by their ordered index number.

When writing a .ghcs file, unless you can guarantee a chunk is valid (either you are writing that chunk, or you haven’t changed the scene at all since that chunk was originally created), you should not write it out.

### Endianness

This file format requires support for both little and big endian data, in two important ways:

1. First, the file’s overall “structural endianness” which is used for the File Header, Table of Contents, and all chunk Sub-Headers (including each one’s “chunk data size”). This can be either big or little, and is specified in the File Header (just after the major/minor version bytes - see below).
2. Second, each chunk’s own private data (but NOT its chunk Sub-Header!) can be stored in its own endianness. This can be either big endian or little endian (or “n/a” for chunks whose data consists only of endian-agnostic bytes), and is determined by the “chunk data endianness” byte in that chunk’s Sub-Header (see below).

*We will never use or support any other endian format besides big or little (e.g. middle-endian, PDP, Honeywell).*

### Data Types

All data in this file - including all chunk data - consists only of the following primitive data elements:

* char (8 bit signed / int8\_t)
* byte (8 bit unsigned char / uint8\_t)
* short (16 bit signed / int16\_t)
* unsigned short (32 bit signed / uint16\_t)
* int (32 bit signed / int32\_t)
* unsigned int (32 bit / uint32\_t)
* int64\_t (64 bit signed int)
* uint64\_t (64 bit unsigned int)
* float (32 bit IEEE 754 standard)
* double (64 bit IEEE 754 standard)

Notable exceptions and omissions:

* Single booleans are always stored as bytes (each byte either 0=false or 1=true; other values illegal).

Compound PoD (non-virtual “Plain Old Data”) arrays & structs made up of these types are of course easily supported. Examples used in this file format by common convention include:

* Vec2 (float x, float y)
* Vec3 (float x, float y, float z)
* Rgba8 (byte r, byte g, byte b, byte a)
* VertexPCU (Vec3 pos, Rgba color, Vec2 uv)
* Explicit-length UTF-8 String (uint32 length, followed by array of that many bytes, **no** zero-terminator)
* Zero-terminated UTF-8 String (array of unknown number of bytes, ending in a zero byte terminator)

*(note: for both of the above string types, UTF-8 encoding is presumed; ASCII is a perfect subset of this)*

# File Header

***File Header****: 12 bytes total*

* (4 byte array) **FourCC** byte sequence: ‘G’ ‘H’ ‘C’ ‘S’ (hex: 0x47, 0x48, 0x43, 0x53) (dec: 71, 72, 67, 83)
* (1 byte) reserved: **zero** (0x00)
* (1 byte) **Major file version** number: currently 1 (increases only)
* (1 byte) **Minor file version** number: currently 0 (increases, resets within each Major version number)
* (1 byte) **Endian** mode (1=little, 2=big) for **overall file structure**; this is used for all chunk Sub-Headers, the Table of Contents, and the ToC offset following.
* (4 bytes, uint32) **ToC offset**; byte-offset location of the Table of Contents (from start of file)

# Chunk Structure

***Chunk Sub-Header****: 10 bytes (at the start of each Chunk)*

* (4 byte array) **FourCC** byte sequence: ‘\0’ ‘C’ ‘H’ ‘K’ (hex: 0x00, 0x43, 0x48, 0x4B) (dec: 0, 67, 72,75)
* (1 byte) **Chunk type** (see below for a list of chunk types)
* (1 byte) **Chunk data endianness** (1=little, 2=big, 0=n/a) used for the chunk’s private data segment
  + *Note: Chunk data endianness can be 0 (n/a) if the chunk’s endian-agnostic single bytes only.*
* (4 bytes, uint32) **Chunk data size** (excluding sub-header), in endianness specified in main File Header
  + *NOTE: “chunk data size” uses the overall file structure endian setting, NOT the chunk’s setting!*

***Chunk Data Segment****: X bytes (given in the chunk’s Sub-Header)*

* *Chunk private data format varies by chunk type; see Chunk Formats, below*

# Table of Contents

At the end of the file, after the final chunk’s data segment, is the Table of Contents (ToC).

The ToC’s position (byte offset) in the buffer/file was given in the main File Header.

The ToC’s endianness is consistent throughout, and was given in the main File Header.

***Table of Contents****: ? bytes*

* (4 byte array) **FourCC** byte sequence: ‘\0’ ‘T’ ‘O’ ‘C’ (hex: 0x00, 0x54, 0x4F, 0x43) (dec: 0, 84, 79, 67)
* (1 byte) **Number of chunks** present in file (and ToC), including SceneInfo chunk

*...and then one entry in the ToC* ***for each chunk*** *in the file:*

* + (1 byte) **Chunk type** (see below for a list of chunk types)
  + (4 bytes, uint32) **Chunk location** (byte-offset within file of the start of the chunk’s Sub-Header)
  + (4 bytes, uint32) **Chunk data size** (excluding sub-header); agrees with # in chunk Sub-Header

# Chunk Types

Chunk types < 128 (i.e. with no high bit set) are considered **canonical** scene chunks, i.e. any one of which is sufficient to rebuild the original “scene” (set of convex objects), and at least one of which is required to define a non-empty scene. Canonical chunks may not be repeated (e.g. you cannot have two ConvexPolys chunks).

All Canonical chunks must precede all optional chunks.

Optional chunks - e.g. those with metadata such as Bounding Discs or Quadtrees - always have chunk type numbers of 128+ (i.e. with high bit set). Chunk type 255 (0xFF) is reserved to mean “unknown/invalid chunk type”. *Any other chunk type - including chunk type 0x00 - is an error (illegal/corrupted file). It is also legal to write a scene file that has NO chunks in it, which indicates the scene is empty.*

Canonical chunk types (at least one of which is required):

* 0x01: **ConvexPoly**s Chunk (canonical)
* 0x02: **ConvexHull**s Chunk (canonical)
* 0x03: **BSP**2 Tree Chunk (canonical)
* 0x04: ?

Optional chunk types:

* 0x80: **SceneInfo** Chunk
* 0x81: **Bounding Disc**s Chunk
* 0x82: **Bounding AABB**s Chunk
* 0x83: **AABB2 Tree** (BVH) Chunk
* 0x84: **OBB2 Tree** (BVH) Chunk
* 0x85: **ConvexHull Tree** (BVH) Chunk
* 0x86: **Asymmetric Quadtree** Chunk
* 0x87: **Symmetric Quadtree** Chunk
* 0x88: **Tiled Bit Regions** Chunk
* 0x89: **ColumnRow Bit Regions** Chunk
* 0x8A: **Disc2 Tree** (BVH) Chunk
* 0xFF: **Ignore** (invalid) chunk - but chunk data size must still be valid in chunk’s Sub-Header

# Chunk Formats

0x01: **ConvexPolys Chunk** (canonical)

*Blue text is required chunk sub-header:*

* (4 byte array) **FourCC** byte sequence: ‘\0’ ‘C’ ‘H’ ‘K’ (hex: 0x00, 0x43, 0x48, 0x4B) (dec: 0, 67, 72,75)
* (1 byte) **Chunk type** = 0x01
* (1 byte) **Chunk data endianness** = 1 (always little endian)
* (4 bytes, uint32) **Chunk data size** = ?
* (4 bytes, uint32) **Number of convex objects** in scene
* *...and then the following for each convex poly in the scene:*
  + (2 bytes, uint16) **Number of vertexes** / sides for this convex polygon
  + *...and then the following for each vertex in the convex poly; vertexes are always given in “+X axis turning positive toward +Y axis” order (e.g. counter-clockwise if origin is bottom-left):*
    - (8 bytes, Vec2) **Vertex Position**

0x02: **ConvexHulls Chunk** (canonical)

* (4 byte array) **FourCC** byte sequence: ‘\0’ ‘C’ ‘H’ ‘K’ (hex: 0x00, 0x43, 0x48, 0x4B) (dec: 0, 67, 72,75)
* (1 byte) **Chunk type** = 0x02
* (1 byte) **Chunk data endianness** = ?
* (4 bytes, uint32) **Chunk data size** = ?
* (4 bytes, uint32) **Number of convex objects** in scene
* *...and then the following for each convex object in the scene:*
  + (2 bytes, uint16) **Number of planes** for this convex hull (note: may include “useless” planes)
  + *...and then the following for each plane in the convex poly:*
    - (8 bytes, Vec2) **Plane Normal Vector** (must be normalized)
    - (4 bytes, float) **Distance to plane** from origin along normal (d>0 means origin is behind)
      * ...where Dot( N, P ) = d, for plane normal N and any point on plane P, distance along normal from origin of **d**

0x03: **BSP**2 Tree Chunk (canonical)

* (4 byte array) **FourCC** byte sequence: ‘\0’ ‘C’ ‘H’ ‘K’ (hex: 0x00, 0x43, 0x48, 0x4B) (dec: 0, 67, 72,75)
* (1 byte) **Chunk type** = 0x08
* (1 byte) **Chunk data endianness** = ?
* (4 bytes, uint32) **Chunk data size** = ?
* (1 byte) BSP2 Tree Chunk version number (current version: 1); *will increment if the following changes:*
* (4 bytes, uint32) Number of nodes in BSP Tree (including leaves) (for example, 122)
* ...and for each node in the BSP Tree (for example, 122 nodes):
  + (8 bytes, Vec2) Plane2D normal x,y; normalized for non-leaf nodes, or (0,0) for leaf nodes
  + (4 bytes, float) Signed distance of split from origin along normal (e.g. + if origin is behind plane)
  + (4 bytes, int32) Parent node index (-1 for root)
  + (4 bytes, int32) Back child index (-1 for leaf nodes)
  + (4 bytes, int32) Front child index (-1 for leaf nodes)
  + (4 bytes, int32) Node contents metadata, e.g. enum type (-1 for “mixed” contents below)
* (4 bytes, uint32) Number of convex objects in the original scene (for example, 76 original objects)
* ...and for each convex object in the original scene:
  + (4 bytes, uint32) Number of BSP tree leaves which contain this convex object (where it went)
  + For each tree leaf expected (the array count just given) (76 of them):
    - (4 bytes, int32) BSP Tree node index (one of 1+ that this object was cut up into)
    - *Note: Each of these ^^^ is a BSP Tree node index (e.g. in [0,121]) that MUST be a leaf*

0x80: **SceneInfo Chunk**

* (4 byte array) **FourCC** byte sequence: ‘\0’ ‘C’ ‘H’ ‘K’ (hex: 0x00, 0x43, 0x48, 0x4B) (dec: 0, 67, 72,75)
* (1 byte) **Chunk type** = 0x00
* (1 byte) **Chunk data endianness** = 1 (always little endian)
* (4 bytes, uint32) **Chunk data size** = 16
* (16 bytes, AABB2) **Camera ortho** (float minX, float minY, float maxX, float maxY)

0x81: **Bounding Disc**s Chunk

* (4 byte array) **FourCC** byte sequence: ‘\0’ ‘C’ ‘H’ ‘K’ (hex: 0x00, 0x43, 0x48, 0x4B) (dec: 0, 67, 72,75)
* (1 byte) **Chunk type** = 0x03
* (1 byte) **Chunk data endianness** = 1 or 2 (can be either little or big endian)
* (4 bytes, uint32) **Chunk data size** = 4 + (12 \* N) bytes, where N is number of scene objects
* (4 bytes, uint32) Number of convex objects in scene (must agree with canonical chunk’s count)
* For each convex object, in scene order:
  + (8 bytes, Vec2) Bounding disc center x,y
  + (4 bytes, float) Bounding disc radius
* Each disc is guaranteed to be big enough to fully encompass its object, but discs may not be perfectly optimized for best/tightest fit

0x82: **Bounding AABB**s Chunk

* (4 byte array) **FourCC** byte sequence: ‘\0’ ‘C’ ‘H’ ‘K’ (hex: 0x00, 0x43, 0x48, 0x4B) (dec: 0, 67, 72,75)
* (1 byte) **Chunk type** = 0x04
* (1 byte) **Chunk data endianness** = ?
* (4 bytes, uint32) **Chunk data size** = ?
* ??? (TBD)

0x83: **AABB2 Tree** (BVH) Chunk

* (4 byte array) **FourCC** byte sequence: ‘\0’ ‘C’ ‘H’ ‘K’ (hex: 0x00, 0x43, 0x48, 0x4B) (dec: 0, 67, 72,75)
* (1 byte) **Chunk type** = 0x05
* (1 byte) **Chunk data endianness** = ?
* (4 bytes, uint32) **Chunk data size** = ?
* ??? (TBD)

0x84: **OBB2 Tree** (BVH) Chunk

* (4 byte array) **FourCC** byte sequence: ‘\0’ ‘C’ ‘H’ ‘K’ (hex: 0x00, 0x43, 0x48, 0x4B) (dec: 0, 67, 72,75)
* (1 byte) **Chunk type** = 0x06
* (1 byte) **Chunk data endianness** = ?
* (4 bytes, uint32) **Chunk data size** = ?
* ??? (TBD)

0x85: **ConvexHull Tree** (BVH) Chunk

* (4 byte array) **FourCC** byte sequence: ‘\0’ ‘C’ ‘H’ ‘K’ (hex: 0x00, 0x43, 0x48, 0x4B) (dec: 0, 67, 72,75)
* (1 byte) **Chunk type** = 0x07
* (1 byte) **Chunk data endianness** = ?
* (4 bytes, uint32) **Chunk data size** = ?
* ??? (TBD)

0x86: **Asymmetric Quadtree** Chunk

* (4 byte array) **FourCC** byte sequence: ‘\0’ ‘C’ ‘H’ ‘K’ (hex: 0x00, 0x43, 0x48, 0x4B) (dec: 0, 67, 72,75)
* (1 byte) **Chunk type** = 0x09
* (1 byte) **Chunk data endianness** = ?
* (4 bytes, uint32) **Chunk data size** = ?
* ??? (TBD)

0x87: **Symmetric Quadtree** Chunk

* (4 byte array) **FourCC** byte sequence: ‘\0’ ‘C’ ‘H’ ‘K’ (hex: 0x00, 0x43, 0x48, 0x4B) (dec: 0, 67, 72,75)
* (1 byte) **Chunk type** = 0x0A
* (1 byte) **Chunk data endianness** = ?
* (4 bytes, uint32) **Chunk data size** = ?
* ??? (TBD) - KZ, Jon, Adam
* Nodes are described in tree traversal pre-order. (Root then four children nodes, recursively)
* Layout
  + Subheader
  + Node(s)
    - This will recursively describe the tree
* Elements
  + Sub Header
    - (16 bytes, AABB2) **Boundary** (float minX, float minY, float maxX, float maxY)
    - 1 byte
      * 0x00 - root is a leaf node
      * 0x01 - root is not a leaf (i.e. a branch node)
  + A node representing the root of the tree. (Leaf/branch nodes)
  + Branch Node
    - (1 byte, char)
      * Structure is 0b0000\*\*\*\*
      * The lower four bits represent whether that quadrant is a leaf
      * The order is bottom left, top left, bottom right, top right, from lower bits to higher bits
      * 0 means leaf
      * 1 means non-leaf
    - Followed by 4 other nodes (leaf nodes and/or branch nodes), in the same order mentioned before.
  + Leaf Node
    - 4 bytes - unsigned int
      * Number of convex object index
    - Array of 4 byte - unsigned int
      * List of convex object indices

0x88: **Tiled Bit Regions** Chunk

* (4 byte array) **FourCC** byte sequence: ‘\0’ ‘C’ ‘H’ ‘K’ (hex: 0x00, 0x43, 0x48, 0x4B) (dec: 0, 67, 72,75)
* (1 byte) **Chunk type** = 0x0B
* (1 byte) **Chunk data endianness** = ?
* (4 bytes, uint32) **Chunk data size** = ?
* (8 bytes, Vec2) ~~Scene bounds~~ Tiled area bounds mins: float minX, float minY
* (8 bytes, Vec2) ~~Scene bounds~~ Tiled area bounds maxs: float maxX, float maxY
* For each object:
  + In object order specified by canonical chunk
  + (8 bytes, uint64) Object bit flags
* If multiple TiledBitRegion chunks exist in the file, each chunk MUST list every object in the scene in the specified order.
* Chunk expects an 8x8 grid, with a bottom-left x-minor tile indexing
  + i.e. bits 0 through 63 correspond to tile 0 in bottom-left, tile 1 to its immediate right, etc.

0x89: **ColumnRow Bit Regions** Chunk

* (4 byte array) **FourCC** byte sequence: ‘\0’ ‘C’ ‘H’ ‘K’ (hex: 0x00, 0x43, 0x48, 0x4B) (dec: 0, 67, 72,75)
* (1 byte) **Chunk type** = 0x0C
* (1 byte) **Chunk data endianness** = ?
* (4 bytes, uint32) **Chunk data size** = ?
* (8 bytes, Vec2) Tiled area bounds min: float minX, float minY
* (8 bytes, Vec2) Tiled area bounds max: float maxX, float maxY
* For each object:
  + In order of objects specified by canonical chunk
  + (4 bytes, uint32\_t) Object row bitflags (along x axis)
  + (4 bytes, uint32\_t) Object column bitflags (along y axis)
* If multiple ColumnRowBitRegion chunks exist in the file, each chunk MUST list every object in the scene in the specified order.

0x8A: **Disc2 Tree** (BVH) Chunk

* (4 byte array) **FourCC** byte sequence: ‘\0’ ‘C’ ‘H’ ‘K’ (hex: 0x00, 0x43, 0x48, 0x4B) (dec: 0, 67, 72,75)
* (1 byte) **Chunk type** = 0x05
* (1 byte) **Chunk data endianness** = ?
* (4 bytes, uint32) **Chunk data size** = ?
* (4 bytes unsigned int) Discs in scene count
* For each object:
  + (8 bytes, Vec2) Disc Center
  + (4 bytes, float) Disc radius
  + (4 bytes, unsigned int) Parent Disc index (0x00 for null parent)
  + (4 bytes, unsigned int) Left Disc index (0x00 for null child)
  + (4 bytes, unsigned int) Right Disc index (0x00 for null child)
  + (4 bytes, unsigned int) Convex Polys Count that are inside me
  + X amount of (4 bytes, unsigned int) Convex Poly Index of poly that is inside me

# Version & Revision History

* Version 01.00 (major.minor): Initial file specification