.BDAE File Format Parser and Viewer

Project overview

This project implements a 3D model viewer for .bdae file format, based on the .bdae parser from Gameloft's game engine, and compiled into a standalone application. The parser loads the file sections into memory and processes file data, applying offset correction and extracting the strings that store the core information for rendering the 3D model. Originally borrowed from the source code of another Gameloft title, it has been modified to support the 64-bit .bdae file version 0.0.0.779 used in Order and Chaos Online v4.2.5. After reading the below documentation, you may clearly understand what is the .bdae file format, which is a result of my combined reverse-engineering and game-engine source code research.

Main information

All game 3D models are stored in binary files of the `.bdae` format. They include **data for mesh, material, textures, nodes, bones, SFX, and animations**. BDAE stands for **Binary Digital Access Exchange** and is a binary version of the DAE format, which itself is written in the XML language common for 3D models. The binary nature of .bdae files makes them superior in terms of size, runtime efficiency, and protection. When developers introduce new game assets, they run a dae2bdae script. **The developer of .bdae is Gameloft — it is used in their games and natively supported by their Glitch Engine** (which itself is based on the Irrlicht Engine). .bdae file format is not unified — each game uses a different version of it. Furthermore, each version has 4 subversions. The subversion is likely generated or updated automatically based on the configuration of the Glitch Engine.

Known information about .bdae file format is limited and located across a few forums. This is because **for reading and understanding these files you would have to do reverse-engineering**. For OaC .bdae files, this has been done in Game Engine Filetypes Reverse Engineering. That project provides a file-parsing template to view the structure of a .bdae file in a binary file editor like 010 Editor. However, because this is a pure individual-file-based reverse engineering approach without having the game engine's source code as a reference, some 3D model data interpretations remain incomplete or incorrect.

Another problem is that not only can you not easily read and understand .bdae files, but more importantly, there is no convenient software to render the 3D models they contain. Several available tools you might find are based on custom-written plugins that are unstable; they may work on one .bdae version and fail on others. For the OaC .bdae file version, the only reliable option is Ultimate Unwrap 3D Pro, which does support the format, yet can only display a mesh, without any textures applied.

BDAE file parser

The .bdae parser consists of:

- `resFile.cpp` – parser's core implementation (explained below).
- `resFile.h` – parser's header file that declares the in-memory layout of the .bdae File object and its header structure.
- `access.h` – utility header that provides an interface for accessing loaded data either as a file-relative offset or as a direct pointer.
- `io` – input / output library that provides an interface for reading any game resource files from various sources (disk, memory, Gameloft's custom packed resource format, ZIP archives) with efficient memory management and reference counting. It is a part of the Glitch Engine, but has no dependencies on other engine modules.

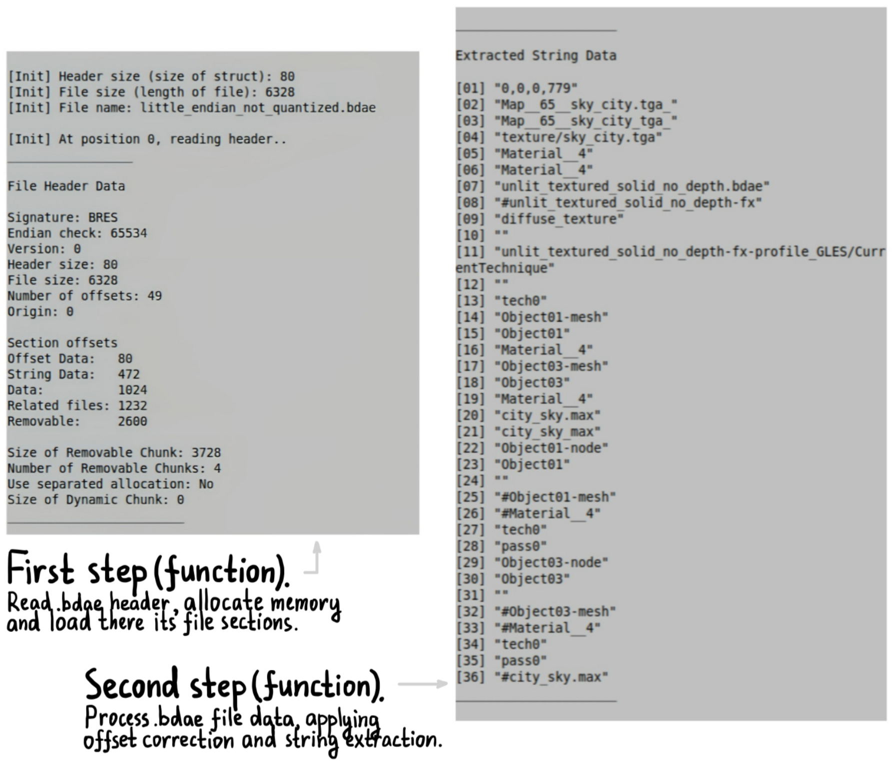
These files were taken from the Heroes of Order and Chaos game source code and reworked. Their .bdae parser was implemented as a utility module of the Glitch Engine, accessible under the `glitch::res` namespace. It is the absolute **entry point for a .bdae file in the game, performing its in-memory initialization**. When the world map loads, the very first step is to correctly load all game resources, and for .bdae files, this parser is responsible for that.

My target was to build a parser independent of the Glitch Engine that would correctly parse .bdae files from the latest OaC version 4.2.5. In order to achieve this, the **parser had to be hardly modified: handled .bdae version difference (OaC uses v0.0.0.779 against v0.0.0.884 in HoC) and architecture difference (old OaC v1.0.3 and HoC .bdae files are designed to be parsed by a 32-bit game engine, while newer OaC 4.2.5 files expect a 64-bit game engine), Glitch Engine dependency removed, refactored and highly annotated**. Advanced explanation – it appears that inside a .bdae version there are 4 possible subversions / architecture configurations: big-endian 32-bit, big-endian 64-bit, little-endian 32-bit, and little-endian 64-bit. Attempting to parse a .bdae file of the wrong architecture would lead to undefined behavior, as 32-bit systems are written for a pointer size of 4 bytes, and in 64-bit systems it is 8 bytes, resulting in incorrect offsets. OaC v1.0.3 and HoC both use little-endian 32-bit .bdae files, while OaC v4.2.5 uses little-endian 64-bit (both of the .bdae version 0.0.0.79), i.e., the old parser is incompatible with the latest OaC .bdae files. This issue has been resolved.

How does the .bdae parser work?

Assume we opened the outer `some_model.bdae` archive file and there is a file `little_endian_not_quantized.bdae` inside it, which is the real file storing the 3D model data (see `main.cpp`), and so we opened this inner file as well. Now we call the initialization function `Init()`, which is split into 2 separate functions with the same name. **In the first function, we read the raw binary data from the .bdae file and load its sections into memory.** Basically, it is the preparation step for the main initialization, since we don't do any parsing and just allocate memory and load raw data based on the values read from the .bdae header. **In the second function, we resolve all relative offsets in the loaded .bdae file, converting them to direct pointers to the data while handling internal vs. external data references, string extraction, and removable chunks.** This is the main initialization step, after which we can quickly access any data of the 3D model.

Some explanation may be required here.. See the code annotation for more detail.



```
[Init] Header size (size of struct): 80
[Init] File size (length of file): 6328
[Init] File name: little_endian_not_quantized.bdae

[Init] At position 0, reading header..
________________

File Header Data

Signature: BRES
Endian check: 65534
Version: 0
Header size: 80
File size: 6328
Number of offsets: 49
Origin: 0

Section offsets
Offset Data:   80
String Data:   472
Data:          1024
Related files: 1232
Removable:     2600

Size of Removable Chunk: 3728
Number of Removable Chunks: 4
Use separated allocation: No
Size of Dynamic Chunk: 0
______________________
```

First step (function).
Read .bdae header, allocate memory and load there its file sections.

```
____________________

Extracted String Data

[01] "0,0,0,779"
[02] "Map__65__sky_city.tga_"
[03] "Map__65__sky_city_tga_"
[04] "texture/sky_city.tga"
[05] "Material__4"
[06] "Material__4"
[07] "unlit_textured_solid_no_depth.bdae"
[08] "#unlit_textured_solid_no_depth-fx"
[09] "diffuse_texture"
[10] ""
[11] "unlit_textured_solid_no_depth-fx-profile_GLES/Curr
entTechnique"
[12] ""
[13] "tech0"
[14] "Object01-mesh"
[15] "Object01"
[16] "Material__4"
[17] "Object03-mesh"
[18] "Object03"
[19] "Material__4"
[20] "city_sky.max"
[21] "city_sky_max"
[22] "Object01-node"
[23] "Object01"
[24] ""
[25] "#Object01-mesh"
[26] "#Material__4"
[27] "tech0"
[28] "pass0"
[29] "Object03-node"
[30] "Object03"
[31] ""
[32] "#Object03-mesh"
[33] "#Material__4"
[34] "tech0"
[35] "pass0"
[36] "#city_sky.max"
____________________
```

Second step (function).
Process .bdae file data, applying offset correction and string extraction.

Figure 1: result