University of Dublin

TRINITY COLLEGE

Isometric Hack and Slash Game Engine

Tom Mason

B.A. (Mod.) Computer Science

Final Year Project April 2014

Supervisor: Dr. Mads Haahr

School of Computer Science and Statistics

O'Reilly Institute, Trinity College, Dublin 2, Ireland

Declaration

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

Tom Mason

Date

Permission to lend

I agree that the Library and other agents of the College may lend or copy this report upon request.

Tom Mason

Date

Abstract

The purpose of the project is to create a modern open source reimplementation of the engine used in the 1996 video game Diablo.

It will use the original data files, so as to avoid issues with copyright, but will also support modern file formats, and be a generic engine for games of that style.

The original game is an isometric top down hack and slash game, which features some roguelike elements, such as random items and dungeons. These will be a focus of the project.

Acknowledgements

u - for the excellent blizzconv[1], which was a wonderful reference, and also for being extremely helpful and knowledgeable about Diablo.

Ladislav Zezula - for creating StormLib[14], without which this project could not have begun.

Pedro Faria / Jarulf - for writing Jarulf's Guide to Diablo and Hellfire[9].

The Dark Mod team - for their incredibly useful tables of Diablo.exe hex addresses[15].

The people of Blizzard North - for creating the game in the first place.

Table of Contents

Chapter 1

Introduction

1.1 Background and Motivation

Diablo is an isometric hack and slash roguelike game published by Blizzard Entertainment in 1996[2].

Hack and slash is a genre of video game, that focuses on combat. Generally, hack and slash games will emphasize traditional weapons like swords and bows over more modern ones like guns. The name is drawn from the fact that the player spends most of their time running through dungeons, slaying all in their path.

Roguelike games are another genre. The name comes from the game Rogue[7], which spawned the genre. Classic features of roguelike games are the use of a random number generator to create procedurally generated levels, items and enemies, and permanent death (although Diablo does not have this).

As it was published in 1996, the game engine that Diablo runs on has become quite outdated. It was published on old versions of Mac OS and Windows, and is difficult to run even on modern versions of these operating systems. In addition to this, the original engine has only two output resolutions, 640x480, and 800x600, both of which are insufficiently small by today's standards, requiring either stretching or letterboxing to be used on a modern monitor.

Another deficiency in the original engine is its total lack of mod support. Almost all the file formats that were used for the games content were proprietary, with no documentation or tools provided for editing them, and the game logic was completely inaccessible by being locked down into a compiled binary. This however, did not stop the modding community, who managed to reverse engineer much of the game code, and to decode the proprietary content file formats as well.

1.2 Goals

The greater aim of this project is to create a modern engine that addresses the issues mentioned above (unmoddable, unusable on moder platforms, etc.) More specifically, the aim is to create an engine using modern libraries, which is portable across platforms, is friendly to modifications without having to be recompiled, and is released under a permissive license (GPLv3 was chosen for this purpose). The long term aim is to attract contributors from the community to help build the engine as it is envisioned. For the duration of this final year project however, the goal is set a little lower, as a fully feature complete engine would simply take too long. The goal therefore, is to create a working base for the engine.
To do this, the main tasks are:

- to reverse engineer the main important file formats for levels and images,
- implement a C++ library for decoding said formats,
- build a game engine to render these decoded assets as levels and characters,
- create the random dungeon generator, which is the hallmark of Diablo, and the roguelike genre.

The game engine itself should use a modern cross platform api for rendering, to allow it to be cross platform, and also to allow it to make use of today's specialised graphics hardware. It should be as modular as possible, to allow it to be used in the future to build isometric games in general, and especially the Diablo specific code should be factored out for easy removal / replacement. The engine should be well designed, with readable code in order to attract developers in the future.

1.3 Report Roadmap

In the remainder of this report, an attempt shall be made to document the design architecture of the engine implemented, including the choices that were made, and the reasons for them. The level generation algorithm will be outlined. Finally, the various file formats that were reverse engineered in the course of the project shall be documented.

Chapter 2 will introduce some similar efforts, and evaluate them with respect to this project.

Chapter 3 will give a brief overview of the architecture of the engine.

Chapter 4 will contain details about the implementation, algorithms used, etc.

Chapter 5 is devoted to the file formats that are used by the engine. This section is deigned to be usable as a reference by someone who wishes to implement decoders for these formats themselves, and is intended to provide enough information that they would be able to do so without further research of their own.

Chapter 6 will give an evaluation of the project as it stands, with respect to the original goals stated at the start of the report.

Chapter 7 will be the conclusion, and discuss future work.

Chapter 2

Related work

In this chapter I will attempt to review a number of projects similar to this one. They will be evaluated based on their perceived usefulness to this project, which is comprised of: licensing, embedded scripting language, networking support and code quality.

2.1 Flare Isometric Engine

Flare[5] is an open source isometric hack and slash game engine. It uses the SDL library for displaying graphics, and simple text based file formats. It is in active development at time of writing, and is available under the GPLV3. It does not appear to have any embedded scripting language.

2.2 Holyspirit

Holyspirit[8] claims to be in alpha. It uses the SFML library. Does not appear to support networking. Developed in French. Unlike the above, it is an actual game in the hack and slash genre, as opposed to a game engine (although an engine is, of course a part of it). It is also available under a permissive license.

2.3 Fifengine

Fifengine[4] (Flexible Isometric Free Engine) is a FOSS generic isometric game engine. It supports python scripting, and the UI is skinnable with xml. It uses SDL and opengl. It does not support networking. Like Flare, it is a game engine, not a game, but it is more generic, in that is is intended for all kinds of isometric games.

2.4 ProjectDDT

ProjectDDT[12] is an existing attempt to create a modern FOSS engine for Diablo. It has been abandoned now since 2011. Extending this project was considered over creating a new one, but this was decided against as the existing code appears unmaintainable and quite hard to follow. It is however, very useful as a reference, as it is under the GPL. It contains code for loading and interpreting several diablo file formats which proved useful.

2.5 Diablo 1 HD Mod

This[3] is another recreation of the diablo engine that is far more advanced than ProjectDDT. The game appears to be fully playable. However, the source code is not available, and there does not appear to be any plans for it to be made available at any point.

Ultimately, I chose to make my own new engine, as I wanted to have the experience of designing a game engine from scratch.

Chapter 3

Design

The engine should support some kind of scripting in the future, to allow extension of the engine, and of games created for the engine. I would be biased here towards python, because of personal preference, but LUA, or something else entirely could be a valid choice here. File formats used by the game should be simple text formats, like the formats used in Fifengine. The engine should be divided into a number of modules.

3.1 Architecture

The architecture of the engine has been based on the OpenMW[11] engine, with which I have some experience. The project produces a number of executables (currently the main engine executable, an image viewer, and a test program for the IO library), each having its own subdirectory in the apps/ folder in the root of the project.

Code common to multiple "apps" is placed in the components/ subdirectory in the root of the project, and external libraries that have to be shipped as source along with the engine source are placed in the extern/ folder.

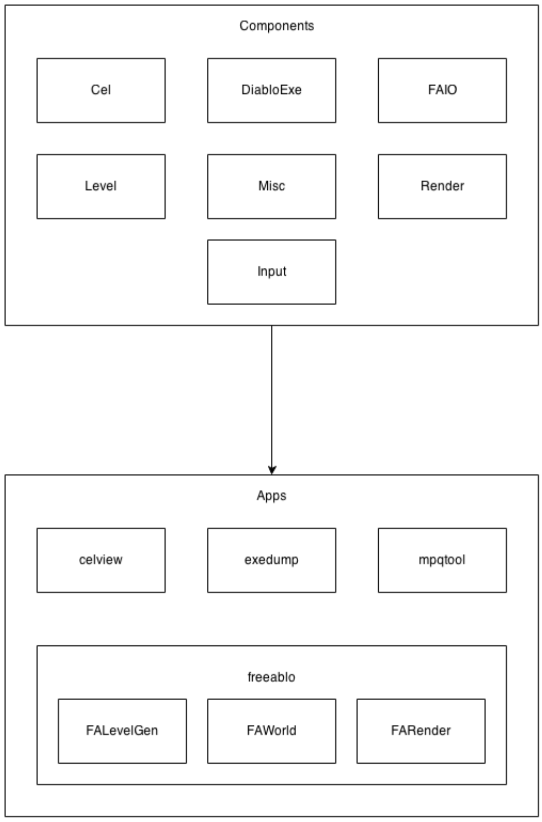

Figure 3.1: Engine Architecture

3.2 Engine Architecture

Code within the main engine folder is split into components prefixed with FA for freeablo. Again, this convention is borrowed from OpenMW[11]. The main important components so far are:

- FAWorld - a container object for the state of the current level. Holds all the objects on the level, and is responsible for updating them (i.e moving them around in response to input etc.)
- FALevelGen - responsible for generating random dungeons
- FARender - controls rendering to screen

3.3 Interaction

Render and Input are tightly coupled, due to both being supplied by SDL. A number of the components use other components (for example, Cel uses FAIO to load the unprocessed image data from the MPQ file). Apps are allowed to depend on Components, but not vice-versa.

Chapter 4

Implementation

4.1 Rendering

Rendering code is split into two parts, in different places. There is a rendering ”component” in the components/render folder. This component exports basic rendering functions for loading and drawing sprites etc, but does not deal with the rendering loop, it has a draw() function which will swap the buffers, and must be called manually. It is essentially a wrapper for a low level rendering library, with some application specific logic (it has the ability to draw ”levels”, ie Level::Level objects representing an isometric level of the game, and also load the proprietary CEL and CL2 formats). This code is placed in a component because it is common to both the freeablo game engine and the image viewer.

The rendering component can be backed by either SDL1 or 2, and this can be set at compile time using the USE_SDL2 cmake variable. SDL2 provides better support for hardware acceleration, and as a result, normally produces much higher framerates. SDL1 was retained as most linux distributions don’t ship SDL2 yet, and also some older platforms are no longer supported in SDL2.

The second part is the code that controls the actual rendering for the game.

This is located in apps/freeablo/farender. Essentially, this contains a class FARender::Renderer, that manages sprite loading and render looping for the game engine.

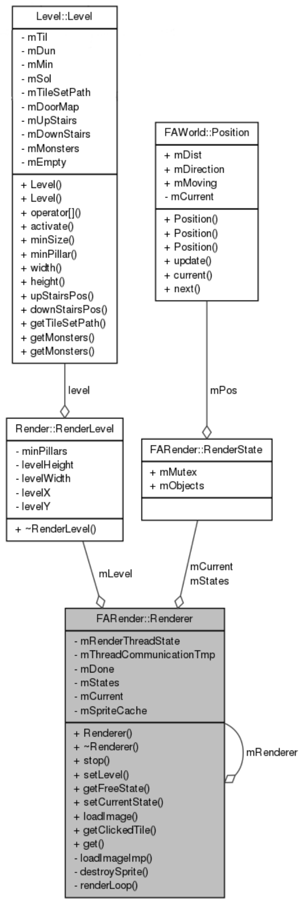

Figure 4.1: Renderer UML Diagram

When created, the Renderer class starts up a separate thread, which then loops until the object is destroyed. Each iteration, the renderer will draw the level, and a list of objects, which are essentially just sprites and locations. The game engine communicates with the renderer through a triple buffered system.

4.1.1 Triple Buffering

The Renderer creates three RenderState objects, each of which is just a container for a number of sprites and their corresponding locations, and a location on which to centre the camera. Each iteration of the game loop, after processing the game logic for the current tick, the engine will "fill" a render state, and pass it off to the renderer. This filling is basically just a flattening of game state, removing all information about objects other than sprite and location, and dumping it into the state. Three states are used, as at any given point the renderer can be drawing a state, and the game loop can be filling one, so with three we are always guaranteed to have one free. Locks are used when rendering and filling a state to ensure that we are never reading and writing the same state at the same time. As the game and render loops can (and probably are) iterating at different rates, when the render loop is going faster, some render states will never be drawn to screen, but this is acceptable, as whatever is on screen at any given moment is an accurate portrayal of game state to the granularity allowed by the iteration speed of the renderer, which is determined by the speed of the processor and GPU (no framelimit is set on the renderer).

4.1.2 Threading Issues

It is a requirement of the library that all SDL calls occur in the main thread, so there are some synchronisation issues with various actions such as loading sprites and changing level. The restriction that it be the main thread is the reason that rendering takes place in the main thread, with the game loop occurring in a separate one, which at first seems counterintuitive. Each action that must take place in the render thread, but is called from the game thread is given an entry in the RenderThreadState enum. The Renderer class has a member, mRenderThreadState, which is an atomic instance of the RenderThreadState enum type (atomicity achieved using boost::atomic). On each iteration of the render loop, the value of this synchronisation variable is checked.

When it is set to running, the game can render the frame, otherwise, it executes the action corresponding to the current value, then resets itself to allow the render to continue, and the caller to know that it has completed execution. Passing values between threads is done via a void* member called mThreadCommunticationTmp.

For example, when loading a sprite, the loadImage function (which is called from the game thread) will save the image path into the void* then set mRenderThreadState to loadSprite, and enter a busy wait for mRenderThreadState == 0. When the render thread encounters mRenderThreadState == loadSprite, it will create a new Sprite object on the heap. It will then pull the path out from the void*, then call loadImageImp with that as the parameter (the function that implements the actual image loading), saving the result into the heap variable it just created. When that function returns, it will assign mThreadCommunticationTmp the pointer to the new heap Sprite, and set mRenderThreadState back to running. The game thread will now see that it has completed, and return the sprite as required.

4.2 Input

Input is handled in the game loop thread, with the Input::InputManager singleton class. Like rendering, it is done using SDL, so it is also abstracted away in the Input component. The input component consists of an object to which one binds callbacks. These callbacks are then executed when the processInput() method is called, if the corresponding input actions have occurred.

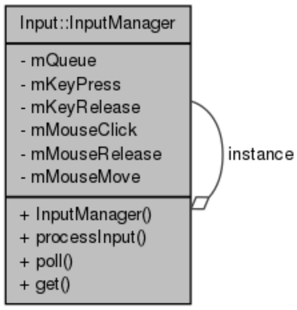

Figure 4.2: InputManager UML Diagram

Unfortunately, while the input is used in the game loop thread, the specifics of SDL require that the SDL event polling occur on the main thread. As such, the raw input polling is done inside the render loop, by calling poll(). For each SDL event generated in the render loop, an event is added onto a concurrent queue. The events on this queue are contained in a structure that very much resembles the parts of the SDL_Event union that we actually use. The game loop then calls processInput, which pops events off the queue, and it is here that the callbacks are executed.

4.3 Game Loop

As explained above, the game loop occurs in a secondary thread. The game loop is essentially a huge while loop located (for now) in the realmain function in the main.cpp file in apps/freeablo. It executes at a fixed rate of 120 time per second, and is responsible for applying user inputs and advancing game state. The fixed execution rate is required for the engine to be deterministic.

Game state is stored in the FAWorld::World object. This object is essentially a container for all object in the world. A call to World::update() will update the positions and animations of these objects. The World class is also responsible for filling RenderState objects, via the fillRenderState method.

4.4 Diablo.exe

Unfortunately, some of the information required for the game is hardcoded into the original executable. The exact extent to which this issue will occur has not yet been fully determined, but so far the freeablo engine is extracting monster and npc data.

To deal with this problem, the DiabloExe component was created. In order for this to be version independent, the DiabloExe class uses a number of ini files to specify data locations. First the file is hashed to determine which version we are using, then the corresponding ini file is loaded according to that hash. These ini files contain the addresses of the relevant data within the file, which can then be loaded.

By abstracting this process out into a generic loader class, we avoid having all sorts of nasty hex addresses cluttering up the source files where the information is actually needed.

Of course, in order to load this information, one first needs to know where it is, which can be a nontrivial task in itself. Fortunately, there has been extensive reverse engineering work done on diablo in the past. Much of the information needed has addresses documented on the website for The Dark Mod[15]. Currently, freeablo only has an ini file for version 1.09 of Diablo, as that is the version documented in the above location (this is the second most recent patch for Diablo).

However, not all the necessary information is documented there, for example the locations of the NPCs in the town had to be figured out unassisted. For this, the Hex Rays IDA[6] decompiler proved fantastically useful.

4.5 FAIO

The FAIO component is responsible for file IO in freeablo (or, more accurately, just input, as it does not allow file writing). The game data files for Diablo are all stored in the DIABDAT.MPQ file. MPQ is a proprietary archive format originally developed by Blizzard for Diablo, and used in all their games since.

Thankfully, a library exists for using these archives, named StormLib[14]. It provides an fopen / fread etc style api for mpq files. FAIO is essentially a wrapper around this library, that allows the user to override files by placing them on the local filesystem. This means that when requested to open a file, it will search for the file in the local filesystem, and if it finds it there, use that, otherwise look in the MPQ file. This behaviour is based on that of the BSA files in the Elder Scrolls game series.

4.6 Level Generation

Level generation in freeablo is performed in a number of stages. The first stage is the creation of a flat map. This is the part with interesting algorithms. After that, the map is turned isometric, and then has monsters place + random variance introduced into the tileset, but neither of these merit discussion.

The level generation algorithm used in freeablo is borrowed from a game called TinyKeep[17], the author of which has published the algorithm he developed[16]. The algorithm is designed to create rooms connected by corridors on a grid. There are a number of steps which are executed in sequence to produce this map.

- The first step is to place a number of rooms in the centre of the grid, keeping them within a small circle placed there. The rooms can overlap within this circle, and indeed are expected to. The number of rooms, and the radius of the circle in which they are placed should be related in some way to the size of the map being generated. The width, height, and position within the circle of the rooms is randomly generated, with the randomness for width and height biased so we receive more small rooms than large ones.
- After this, we use separation steering to move the rooms away from each other until none of them overlap.
- At this point, we split the rooms into two groups, by thresholding on size. Those over the threshold value (area of 30 was used in the freeablo engine) are said to be real rooms, and the rest are said to be corridor rooms. The bias when generating levels mentioned above ensures that most rooms are chosen to be corridor rooms.
- We construct a graph of real rooms, where each room is connected to each other room. We then calculate the minimum spanning tree of this graph. Now we

know that if we apply corridors corresponding to the edges on this graph, each room will be accessible from each other one.

- Because the graph we constructed above is a tree, there will be no cycles, however a small number of cycles is desirable in a dungeon crawler, so we add in a number of random edges to create some.
- For every edge on the graph, we create an L-shaped corridor on the map, joining the two rooms that correspond to that edges vertices. This is where the corridor rooms come into effect. For each corridor room that the corridors intersect, we add the shape of that room onto the corridor. In this way, we end up with lumpy corridors that can resemble large rooms themselves, and do not just look like simple L shapes.

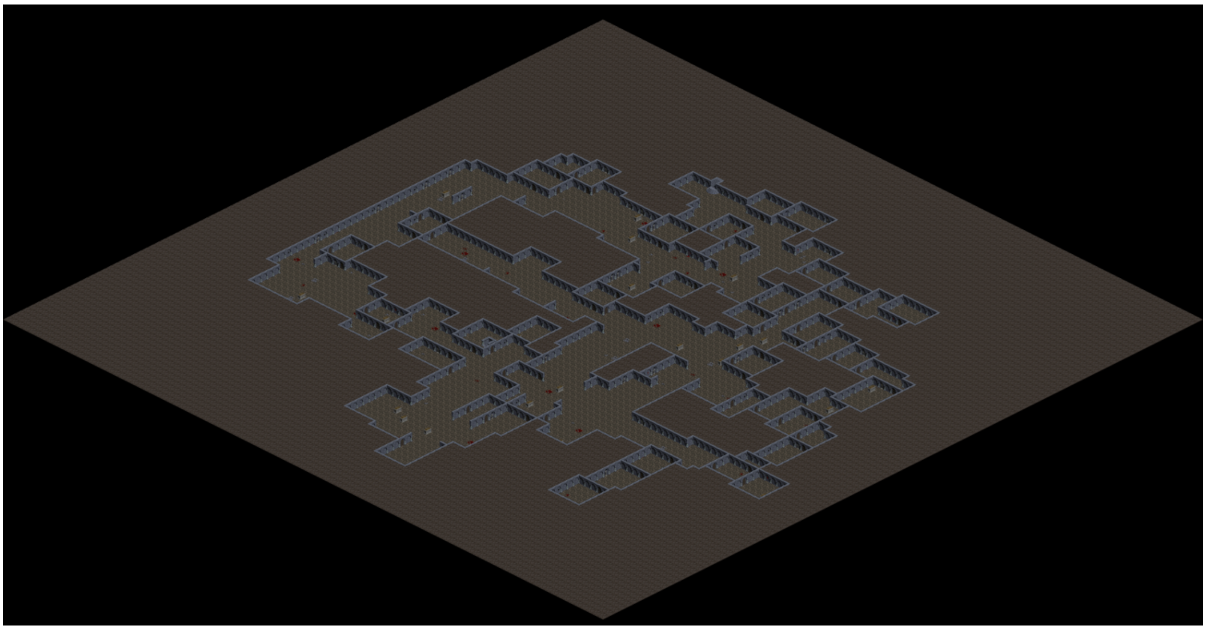

Figure 4.3: Example of generated level

4.7 Libraries

4.7.1 2d graphics libraries

There seems to be 3 different options for 2d graphics in C++:

- SDL
- Allegro
- SFML

Of the above, all are written in plain C, except Allegro, which is C++. I have decided to use SDL for this project, as I am already familiar with it. As described above, both SDL 1 and 2 were used, with the version being configurable at compile time through a cmake varaible.

4.7.2 Cross Platform

The Boost C++ library addresses many of the problems with writing portable C++ code today. Heavy use has been made of various parts of boost, mostly boost::thread and boost::filesystem.

4.7.3 Audio

SDL has a module for audio, SDL_sound[13], but it has not been updated since 2008. FFMPEG's library, libavcodec[10] supports a large number of formats. OpenAL seems to be popular also, but is no longer FOSS. Audio has not yet been implemented, but when it is I would lean towards FFMPEG, as it can also be used to decode the intro videos.

4.7.4 StormLib

StormLib[14] is a library for accessing files in Blizzard MPQ format archives. It was chosen because it is the only viable option. Others do exist, but are no longer maintained.

Chapter 5

File Formats

In the following section, I will use stdint.h style names for naming data types with exact bit width. We will be looking at the file formats used by Diablo to describe image data (CEL, CL2), and levels (DUN, MIN, TIL, SOL). The image file decoding was originally implemented according to an incomplete spec[18], that I found on the internet, coupled with trial and error. Later, when working on the level files, I found blizzconv[1], which contained a pretty much complete implementation in go of the image and level formats that I could work off.

Heuristics for determining image width were implemented by trial and error, with some pointers from the spec mentioned above.

The below specifications were produced by looking at the code that I had written, once I had finished implementing them.

5.1 PAL files

PAL files are colour palettes used by the image formats in diablo. They always contain 256 colours, and each colour is 3 bytes long (r, g, and b bytes), so they are always 768 bytes long. Image files refer to them by index into the file (thus, a two would represent the third colour, or the third group of three bytes).

5.2 CEL image files

CEL image files use the CEL and CL2 file extensions. There are some minor differences between the two, but they are fundamentally the same. The basic capabilities of the format are run length encoding, and transparency (but only total transparency, not partial). Each file can contain multiple frames that can represent parts of an object, frames in an animation, or even tilesets for levels.

5.2.1 File Header

The file header is composed of a series of uint32_t. The first is the number of frames. This is followed by an offset from the start of the file for each frame, and finally, an offset to the end of the file. Illustrated below is a pseudo-C struct representing its structure.

```
struct fileHeader

        uint32_t numFrames;
        uint32_t frameOffsets[numFrames];
        uint32_t endOffset;
};
```

This header is common to both CEL and CL2 files.

5.2.2 Frame Headers

Some CEL frames contain headers at the start of the frame. It is 5 uint16_t (10 bytes) long. Entries appear to be pointers to positions in the file, which when reached during decoding will leave us with a specific number of lines created, but I only understand the second entry (and it is the only one of use to us). This entry gives us a position in the file, that when we reach it, we will have processed 32 lines of pixels in the image. By checking how many pixels have been generated by the time we get to that point, we can divide this number by 32 to get the image width. The first entry is always 10, as it points to the start of the image data. The third entry may point to the end of the 64th line (if it exists) and so on, but I have not investigated this as it is of no use to me.

5.2.3 CEL Frames

There are two kinds of plain CEL frame. One is the "normal" kind, which contains animations of objects. Examples of these can be found in the items directory in DIABDAT.MPQ. The other is tileset cel frames. As the name implies, these contain the tilesets for levels. These are found only in levels/*/*.cel. A given CEL file will only contain one of these types, not both. A colour in a CEL frame will always be a single byte index into a palette.

5.2.3.1 Normal CEL Frames

Normal Frames are composed of a series of command and data blocks. Each block is a uint8_t. The command blocks contain instructions about what to do next during decoding. The data blocks contain indices into a palette to obtain a colour value.

Decoding is performed by starting at the start of the file (the first block will always be a control block), and executing the command there.
One advances by the number of blocks specified by the current block, which brings one to the next control block, and so on until the entire frame has been decoded. There are two kinds of control block: Regular and Transparency.

Regular blocks are denoted by values ≤ 127. When encountering a regular block, its value indicates how many pixels it contains. For example, if one encounters a Regular block with value 10, the next 10 blocks are data blocks, one pixel each, and the 11th block after is the next control block.

Transparency blocks are denoted by values > 127. When a transparency block is encountered, it indicates $256-$block value transparent pixels (for example, 250 would represent 6 transparent pixels). Transparency blocks do not use any data blocks, and so the immediate next block is the next control block.

Below is a sample implementation of decoding a frame:

```
// Frame is the raw frame from the file, pal is a palette
// raw_image is the destination for decoded pixels
void CelFile::normal_decode(vector<uint8_t>& frame, Pal pal, vector<colour>&
    raw_image)
{
    size_t i = 0;

    for(; i < frame.size(); i++)
    {
        // Regular command
        if(frame[i] <= 127)
        {
            size_t j;
            // Just push the number of pixels specified by the command
            for(j = 1; j < frame[i]+1 && i+j < frame.size(); j++)
            {
                int index = i+j;
                uint8_t f = frame[index];
                colour col = pal[f];
                raw_image.push_back(col);
            }

            i+= frame[i];
        }

        // Transparency command
        else // >= 128
        {
            // Push (256 - command value) transparent pixels
            for(size_t j = 0; j < 256-frame[i]; j++)
                raw_image.push_back(transparentColour());
        }
    }
}
```

5.2.3.2 Tileset CEL frames

These CEL files have the same format as normal CEL files, but the data in the frames is different. There are a number of possible ”types” of frame within tileset CEL files. All of them are always of width and height 32.

- Raw: Raw frames are just that, 32*32=1024 bytes of raw colours, with no transparency. A heuristic to identify this kind of frame is to look for frames whose size is exactly 1024 bytes, which works in most cases, but is not completely reliable. blizzconv[1] deals with this issue by storing the frame type for each frame in each tileset CEL file in an ini file written in advance.

- Normal: Some frames are normal frames as described in the previous section. These never have headers when contained in tileset CEL files.
- Greater/Less than frames: These are the most interesting frame type in cel files, as they are the most specialised. They are the tiny triangles which make up half of an isometric block on the map. The name greater/less than is borrowed from ProjectDDT[12].

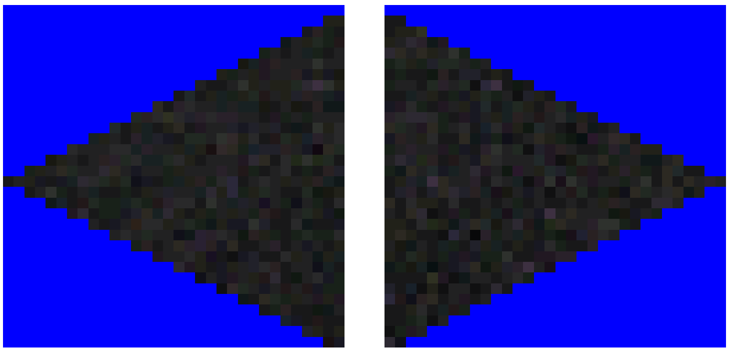

Figure 5.1: Less / Greater than example frames

Above is an example of a less than and greater than frame respectively. It is possible to see, when placed together, they make up a 64*32 pixel isometric block.
You can tell if a frame is a less than or greater than frame by looking at the contents. A certain set of bytes will be zeroed in both cases.
Less Than: bytes 0,1,8,9,24,25,48,49,80,81,120,121,168,169,224,225
Greater Than: bytes 2,3,14,15,34,35,62,63,98,99,142,143,194,195

These bytes are clearly in pairs. I got these indices from the source of ProjectDDT[12]. Each pair marks the end of two rows of colour, as shown in the figure below:

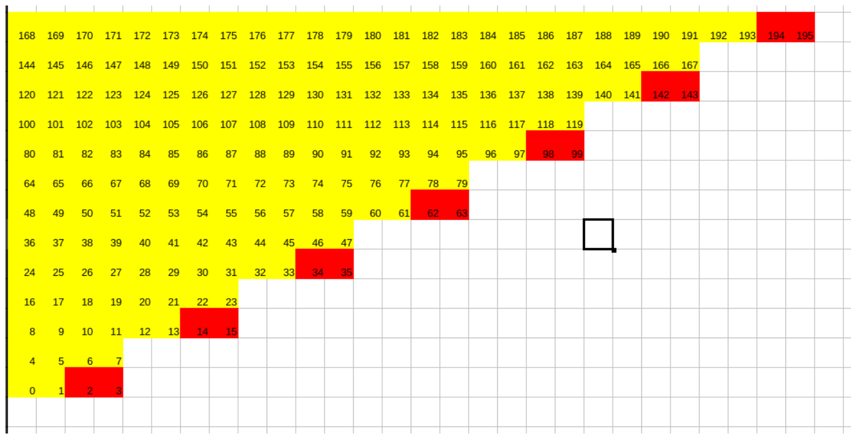

Figure 5.2: Less / Greater than markers example

The yellow blocks are the bytes in between the markers, which contain colour indices, the red are the markers themselves. When rendering, these are ignored, so all non-yellow blocks are transparent.

For a given less/greater than frame, the first half will always conform to the scheme described above, but this only shows half the image. From there, there is variation. Some frames will have another half encoded the same way, with the following markers:
Less Than Second Half: bytes 288,289,348,349,400,401,444,445,480,481,508,509,528,529
Greater Than Second Half: bytes 245,255,318,319,374,375,422,423,462,463,494,495,518, 519,534,535

If these markers are not present, however, the second half is raw with no transparency, so we can just pull it out directly, eg:

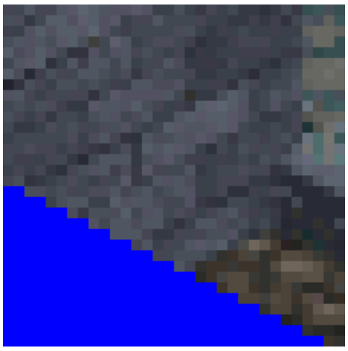

Figure 5.3: Example frame with a raw top

5.2.4 CL2 Frames

CL2 Frames are very similar to CEL frames. The main difference is the use of run-length encoding for colours as well as transparency. They also always have frame headers. In addition to the regular and transparency blocks used in normal CEL frames, they also have RLE blocks, which indicate the number of times to repeat the colour indicated by the next block. Below is some C++ code that illustrates this:

```
void cl2Decode(const std::vector<uint8_t>& frame,
   const Pal& pal, std::vector<Colour>& rawImage)
{
    size_t i = 10; // CL2 frames always have headers

    for(; i < frame.size(); i++)
    {
        // Color command
        if(frame[i] > 127)
        {
            uint8_t val = 256 - frame[i];

            // Regular command
            if(val <= 65)
            {
                size_t j;
                // Just push the number of pixels specified by the command
                for(j = 1; j < val+1 && i+j < frame.size(); j++)
                {
                    int index = i+j;
                    uint8_t f = frame[index];

                    Colour col = pal[f];

```

```
                    rawImage.push_back(col);
                }

                i+= val;
            }

            // RLE (run length encoded) Colour command
            else
            {
                for(int j = 0; j < val-65; j++)
                    rawImage.push_back(pal[frame[i+1]]);

                i += 1;
            }
        }

        // Transparency command
        else
        {
            // Push transparent pixels
            for(size_t j = 0; j < frame[i]; j++)
                rawImage.push_back(Colour(255, 0, 255, false));
        }
    }
}
```

As can be seen above, the blocks use different values, but the basic structure is the same as CEL frames.

5.2.5 Frame Width

Frame width determination is not as simple as it might sound. None of the frame formats have image dimensions built in, but there are a number of heuristics to find them. For images with a frame header, the technique described in section 5.2.2 can be used. For tileset frames, the width is always 32. For all others, there is another technique, which will work so long as the image width is not a multiple of 127, on images with no transparency (which headerless images seem to be).

The maximum stretch of a Regular block is 127. A block will never straddle two lines, so if for example a frame were of width 130, there would be a series of 127 blocks followed by 3 blocks, one pair for each line.

We can abuse this fact, by starting at the start of the frame, and adding together each command block until we find one that is not 127. At that point the sum of the previous 127s + the current block is the width of the image, as the current block has to exist to split on a line.

5.2.6 CEL Archives

Some CEL and CL2 files are in fact archives of multiple CEL/CL2 files, respectively. These are used to store multiple rotations of an animation (e.g. walk animation in all 8 possible directions). These files have headers at the start, which consist of a number of uint32_t s, each one pointing to a file contained in the archive.

As there are always 8 images in such files, the first pointer will always be 32, because it will always point to the first byte after the headers, which are 8*4=32 bytes long, so it is possible to tell which files are archives by checking the first uint32_t against 32.

For CEL files, that's all there is to it, but for CL2, it's a little more complicated. The archive header on CL2 archives points not to the data, but to the individual file headers (described in section 5.2.1), which then point to the frames, relative to their own position.

5.3 Level Files

Levels in diablo are stored in a number of files. To begin with, there is the hierarchy of DUN, TIL and MIN files. DUN files are the top level map file, which contain blocks that refer to the corresponding TIL file. Each entry in the TIL file is for tiles on the map, and each of those tiles is defined in the MIN file. The MIN file defines the sprites that make up the tile (total of 16). This is illustrated in the image below:

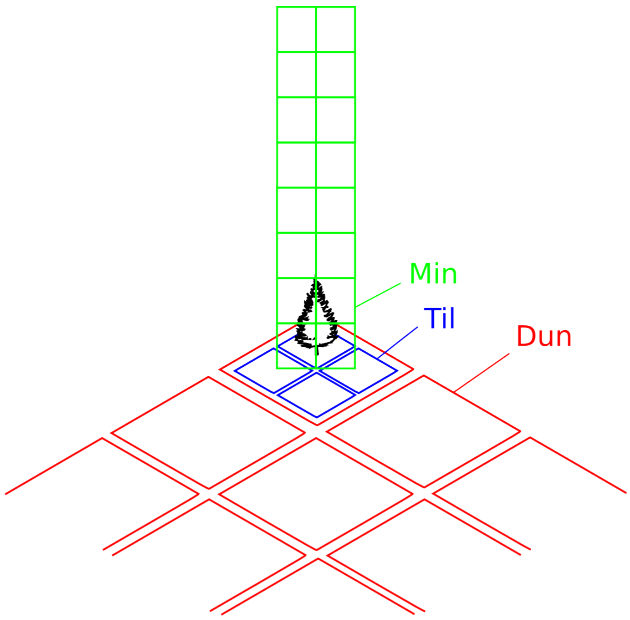

Figure 5.4: Level File Hierarchy

The properties of each tile is defined in the SOL file.

5.3.1 DUN files

DUN files are quite simple. They are essentially a giant array of int16_t s. The first two numbers are the width and height of the level (divided by four, as each block in the dun represents four actual level tiles). The remaining numbers are indices into the TIL file for each group of four tiles. Below is a c-style struct representing the structure of a dun file.

```
struct Dun
{
    int16_t width;
    int16_t height;
    int16_t blocks[width][height];
};
```

5.3.2 TIL files

TIL files are also quite simple; they are just a massive array of int16_t s, where each group of four is a block that can be referred to by the DUN file.

```
struct TilBlock
{
    int16_t top;
    int16_t left;
    int16_t right;
    int16_t bottom;
};

struct Til
{
    TilBlock blocks[FILESIZE/sizeof(TilBlock)];
};
```

5.3.3 MIN files

MIN files are slightly awkward in that their size is not set. In l4.min and town.min, each entry is of size 16, but for all others they are 10. MIN files essentially are a list of blocks, recording the cel frame indices used for each. They are each a pillar with two images on each level, allowing a block to have things up above it (e.g. a tree). They start at the top and work down, as illustrated in the figure below:

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

Figure 5.5: Min Pillar format

5.3.4 SOL files

SOL files have not been fully figured out, however they are used because we can get some useful information out of them. Each byte in the SOL file is a bit field corresponding to an entry in the MIN file. Currently the only known value is the least significant bit, which indicates if a block is "passable" by the player and npcs (i.e. ground is passable, a wall isn't). A 0 in this position indicates that the block is passable, a 1 that it is not.

Chapter 6

Evaluation

The features implemented in the current version of the freeablo engine are as follows:

- Isometric tile rendering
- CEL/CL2 file loading
- Mouse movement
- Level loading
- Level switching
- NPC placement
- Level Generation
- Enemy placement
- Basic collision detection
- Player character display
- Animation

The main file formats have been reverse engineered, and the Level and Cel namespaces contain code for decoding them. The freeablo game engine successfully takes the assets decoded by these, and implements an isometric renderer capable of displaying them as a game world. The FALevelGen component takes care of the last major concern, which is of course level generation.
I think that this should be a good starting point, and that the engine is ready now to start having gameplay elements added in.

Chapter 7

Conclusion

I intend to turn the existing codebase into a proper open source project, drawing outside contributors after the fashion of OpenMW[11]. The large remaining tasks are the implementation of a GUI system, and combat. Once a basic GUI and combat are present, work can begin on porting in all the various bits of game mechanics, such as the appropriate formulae for damage, chance to hit, etc. The excellent work of Pedro Faria / Jarulf on Jarulf's Guide to Diablo and Hellfire[9] will be invaluable in this. Audio and lighting are some more aesthetic features that will have to be carried out in the future.

Level generation currently only generates levels from the first section of the game - the catacombs. The level generator will have to be tweaked to produce levels for the rest of the game as well.

The goal initially would be simply to create a feature complete implementation capable of nothing more than the original engine. Once that goal is accomplished fully, work can begin on extensions, such as an interface for mods, and non-Diablo games to be played using the engine. This could be accomplished by the integration of a scripting language into the engine, along with support for modern file formats such as png. Further to this, it would be desirable to have all of the diablo-specific code factored out of the main engine, and loaded in at runtime as a module. This would mean porting all the image decoding and game mechanics into the scripting language chosen.

Having game code factored out from engine code, and having a system where games can be loaded as modules is of course only useful if there are games created using the engine. For this purpose, and the purpose of making modifications to existing games, it would be desirable if a mod creation and packaging tool could be developed. This could include a level editor, a sprite editor, and tools for managing various media files that may be used by the game.

Further work will have to be done in order to create ini files containing the relevant hex addresses for all the released versions of DIABLO.EXE. Support for hellfire (an expansion pack for the original game) would also be useful, once the base game is fully supported.

Bibliography

[1] blizzconv github page. https://github.com/mewrnd/blizzconv. [Online; accessed Apr 13 14].

[2] Diablo 1. http://www.diablowiki.net/Diablo_I. [Online; accessed Apr 8 14].

[3] Diablo 1 HD Mod. http://mod.diablo1.eu.org/. [Online; accessed Apr 9 14].

[4] Fife wiki features page. https://github.com/fifengine/fifengine/wiki/Features. [Online; accessed Oct 26 13].

[5] Flare engine github page. https://github.com/clintbellanger/flare-engine/. [Online; accessed Oct 26 13].

[6] Hex Rays IDA. https://www.hex-rays.com/products/ida/index.shtml. [Online; accessed Apr 10 14].

[7] History of Rogue. http://www.gamasutra.com/view/feature/4013/the_history_of_rogue_have__you_.php. [Online; accessed Apr 8 14].

[8] Holyspirit sourceforge page. http://sourceforge.net/projects/lechemindeladam/. [Online; accessed Oct 26 13].

[9] Jarulf's guide to Diablo and Hellfire, version 1.62. http://www.lurkerlounge.com/diablo/jarulf/jarulf162.pdf. [Online; accessed Apr 10 14].

[10] libavcodec wikipedia, supported formats. http://en.wikipedia.org/wiki/Libavcodec#Implemented_audio_codecs. [Online; accessed Oct 26 13].

[11] OpenMW website. https://openmw.org/en/. [Online; accessed Mar 24 14].

[12] ProjectDDT sourceforge page. http://sourceforge.net/projects/projectddt/?source=dlp. [Online; accessed Oct 26 13].

[13] SDL_sound homepage. http://www.icculus.org/SDL_sound/. [Online; accessed Oct 26 13].

[14] StormLib Library page. http://www.zezula.net/en/mpq/stormlib.html. [Online; accessed Apr 10 14].

[15] The Dark Mod hex reference. http://www.thedark5.com/info/mod.html. [Online; accessed Apr 10 14].

[16] TinyKeep Dungeon Generation Algorithm. http://www.reddit.com/r/gamedev/comments/1dlwc4/procedural_dungeon_generation_algorithm_explained/. [Online; accessed Mar 25 14].

[17] TinyKeep website. http://tinykeep.com/. [Online; accessed Mar 25 14].

[18] Unofficial Diablo CEL Image Specification. http://www.geocities.ws/fantasydiablo/cel_spec.html. [Online; accessed Apr 18 14].