

Isometric Hack and Slash Game Engine

Tom Mason - 10329877
Supervisor: Mads Haahr

March 25, 2014

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.

Contents

1	Background Research	3
1.1	Relevant existing FOSS Isometric Engines	3
1.1.1	Flare Isometric Engine	3
1.1.2	Holyspirit	3
1.1.3	Fifengine	3
1.1.4	ProjectDDT	3
2	Design Choices	4
2.1	Architecture	4
2.2	Renderer	4
2.3	Libraries	5
2.3.1	2d graphics libraries	5
2.3.2	Cross Platform	5
2.3.3	Audio	5
3	File Formats	6
3.1	CEL image files	6
3.1.1	File Header	6
3.1.2	Frame Headers	6
3.1.3	CEL Frames	6
3.1.3.1	Normal CEL Frames	7
3.1.3.2	Tileset CEL frames	8
3.1.4	CL2 Frames	10
3.1.5	Frame Width	11
3.1.6	CEL Archives	11

1 Background Research

1.1 Relevant existing FOSS Isometric Engines

1.1.1 Flare Isometric Engine

Flare[2] 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 does not appear to have any embedded scripting language.

1.1.2 Holyspirit

Holyspirit[3] claims to be in alpha. It uses the SFML library. Does not appear to support networking. Developed in French.

1.1.3 Fifengine

Fifengine[1] (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.

1.1.4 ProjectDDT

ProjectDDT[6] is an existing attempt to create a modern FOSS engine for Diablo. It has been abandoned now since 2011. I have decided to create my own project, in favour of continuing work on this, because it is extremely badly engineered. For example: the creator of this work decided not to use header files, but simply placed function declarations at the start of files where he needed them. The code is also just generally not very readable, and he has a tendency to not create classes where they would have been appropriate. 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 while hard to follow, is still extremely useful.

2 Design Choices

The engine should support python scripting, to allow extension of the engine, and of games created for the engine. 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.

2.1 Architecture

The architecture of the engine has been based on the OpenMW[5] 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.

2.2 Renderer

The current rendering library being used is SDL 1.2. However, all SDL specific code has been confined to two files, with all other parts of the code using functions exported by those two files. This is done with the intention of easing the transition of switching to SDL 2 in the future.

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 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. 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.

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 ok 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 your processor and GPU (no framelimit is set on the renderer).

2.3 Libraries

2.3.1 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. More specifically, I have decided to use SDL 1. SDL 2 has been released, but is not yet packaged in most distros. The intent is to write an SDL backend, which will eventually support either SDL1 or 2.

2.3.2 Cross Platform

The Boost C++ library addressess many of the problems with writing portable C++ code today. Specifically, I intend to make use of the `boost::filesystem` and `boost::threads` modules to provide platform-agnostic access to threads and files. Even with `boost::filesystem`, I shall have to take care to use case insensitive file loading, as the original game was written for windows, so filename cases may not be consistent.

2.3.3 Audio

SDL has a module for audio, `SDL_sound`[7], but it has not been updated since 2008. FFmpeg's library, `libavcodec`[4] supports a large number of formats. OpenAL seems to be popular also, but is no longer FOSS.

3 File Formats

In the following section, I will use `stdint.h` style names for naming datatypes with exact bit width.

3.1 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.

3.1.1 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 it's structure.

```
struct fileHeader
{
    uint32_t numFrames;
    uint32_t frameOffsets[numFrames];
    uint32_t endOffset;
};
```

This header is common to both CEL and CL2 files.

3.1.2 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.

3.1.3 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.

3.1.3.1 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.

Then you advance by the number of blocks specified by the current block, which brings you to the next control block, and so on until you have decoded the entire frame. There are two kinds of control block: Regular and Transparency.

Regular blocks are denoted by values ≤ 127 . When you encounter a regular block, its value indicates how many pixels it contains. For example, if you encounter 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. 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
```

```

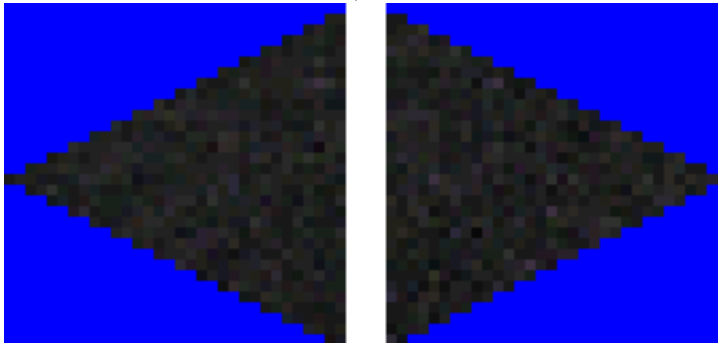
    {
        // Push (256 - command value) transparent
        pixels
        for (size_t j = 0; j < 256-frame[i]; j++)
            raw_image.push_back(transparentColour())
        ;
    }
}

```

3.1.3.2 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.
- 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. 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[6].



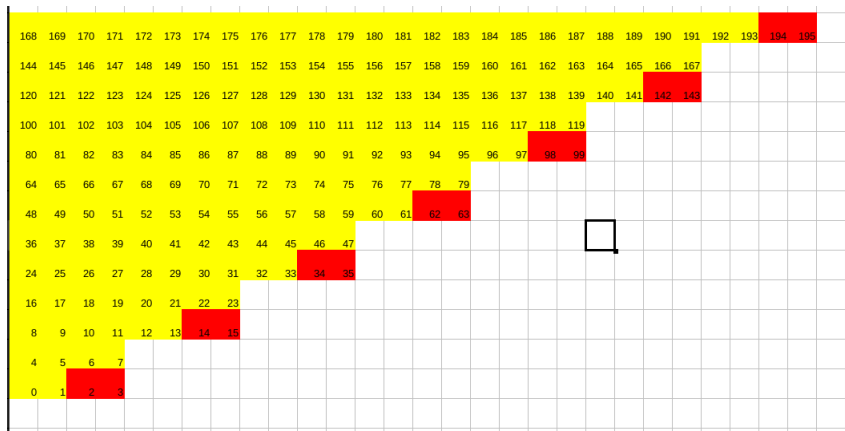
Above is an example of a less than and greater than frame respectively. As you can see, when placed together, they make up a 64*32 pixel isometric block.

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. Each pair marks the end of two rows of colour, as shown in the image below:



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:



3.1.4 CL2 Frames

CL2 Frames are very similar to CEL frames, with the main difference that they use 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.

3.1.5 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 3.1.2 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.

3.1.6 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 (eg 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 is always 8 images in such files, the first pointer will always be 32, as it will always point to the first byte after the headers, which are $8 \times 4 = 32$ 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 3.1.1), which then point to the frames, relative to their own position.

References

- [1] Fife wiki features page. <https://github.com/fifengine/fifengine/wiki/Features>. [Online; accessed Oct 26 13].
- [2] Flare engine github page. <https://github.com/clintbellanger/flare-engine/>. [Online; accessed Oct 26 13].
- [3] Holyspirit sourceforge page. <http://sourceforge.net/projects/lechmindeladam/>. [Online; accessed Oct 26 13].
- [4] libavcodec wikipedia, supported formats. http://en.wikipedia.org/wiki/Libavcodec#Implemented_audio_codecs. [Online; accessed Oct 26 13].
- [5] OpenMW website. <https://openmw.org/en/>. [Online; accessed Mar 24 14].
- [6] ProjectDDT sourceforge page. <http://sourceforge.net/projects/projectddt/?source=dlp>. [Online; accessed Oct 26 13].
- [7] SDL_sound homepage. http://www.icculus.org/SDL_sound/. [Online; accessed Oct 26 13].