NEA: Font Rasterizer

Jake Irvine

March 27, 2021

Chapter 1

Analysis

Text is a key aspect of how we interact with machines. Virtually everything that is done on a computer requires some form of reading on a screen. The program that converts the binary text data to pixels on a screen is a 'font engine' or 'font rasterizer'. These take some text or a single character and output a texture (a collection of pixels) which will get copied into the screen buffer (which is then copied onto the display itself to be shown to the user). There is another component of font rendering, which arranges individual characters output by a font rasterizer into words or paragraphs, called a layout engine.

Creating a full font engine is a very large task - the TrueType format (the industry standard for font files) is a very large specification, and can be further enhanced by vendor-specific extensions. For example, some fonts contain small programs inside them that adjust the actual glyphs to better fit the pixel grid of the screen. These are implemented in a custom virtual machine that is very complex and time consuming to create. However, when rendering characters at a large size, the impact of these features is far less, yet still comes at a substantial performance cost.

To show a font on the screen, we must accomplish two tasks: **Parsing**, and **Rendering**. Parsing is the process of taking the raw font file (which is just a set of bits) and converting it into data structures that make it possible to access specific attributes or structures stored in the file. For example, a key step in parsing the font will be to divide the file up into a number of Glyph tables. Once this is done, the created data structures will be passed to the renderer, which will turn these structures into a bitmap (set of pixels) that is suitable for display on the screen.

The parser will take a TrueType font file and output the bezier curves of the selected character, by reading the binary file. This is non-trivial, because the TrueType format is relatively complex, and contains a lot of optional features (which this project will largely ignore, unless I have time left over). I will

investigate parser-combinator systems and either use that or create my own custom parser architecture.

TrueType fonts are composed of straight line segments and bezier curves. These are primitives that are relatively easy to draw to the screen quickly, using simple mathematical techniques. There are a number of additional options that can be implemented in the renderer, such as anti-aliasing, which smooths the sharp pixel edges of the shapes to make them appear smoother to the eye.

1.1 Project Background

1.1.1 Domain Terminology

Font Family: a collection of typefaces, at different weights, or styles such as italics.

Typeface: commonly referred to as a font, a typeface is a style of lettering that can be displayed on the screen or in print.

Font: an instance of a typeface at a specfic size.

Glyph: the shape corresponding to a specific character in a specific font.

TrueType: TrueType is the industry standard for font files, and the specific file format that is being targeted by this project. It contains all of the data needed to describe a typeface (or sometimes multiple typefaces).

TTF file: A file following the TrueType format.

Table: a table in a TTF file is a section of data following a specfic format that is indexed using the table directory at the start of the font file. For example, the `glyf` table stores the actual glyph data of the font, and the `cmap` table stores the mappings between character and glyph.

Anti-Aliasing: the act of smoothing a line, curve or other bitmap with lighter shades of it's colours to make it appear smoother to the viewier.

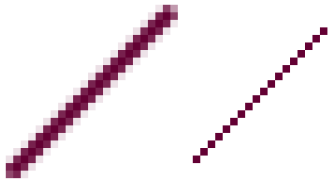


Figure 1.1: *An anti-aliased line on the left, and a non-anti-aliased line on the right.*

Bezier Curve: A curve that is defined by a set of equation. The TTF specification uses a variant of these called **Quadratic Bezier Curves**, which have a single control point. For more details see the section on Bezier Curves below.

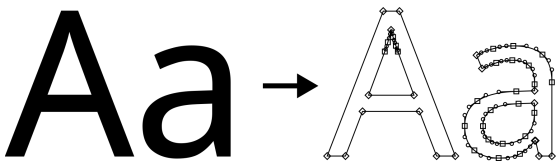


Figure 1.3: *Letters are made up of line segments and bezier curves*

Raster Image: A set of pixels that describe an image

1.1.2 Parsing

As described above, parsing is the act of taking a text or binary sequence and transforming it into data structures that can be used later. There are several ways of doing this. Note that the process of parsing a font file is slightly different to that of parsing a programming language or other free-form format. Programming languages consist of a number of lines of code, whereas a font file is highly structured, with every section of the file having a specific role depending on it's position, (hereafter known as *offset*).

1.1.3 Bezier Curves

Individual glyphs of a font are built up using *outlines*, which are themselves built of several segments of line and curve. More specifically, an outline consists of a set of line segments and quadratic bezier curves, which create a closed loop that forms all or part of the glyph. Luckily, both line segments and bezier curves are fairly easy for computers to render, and have been used in computer graphics since it's inception.

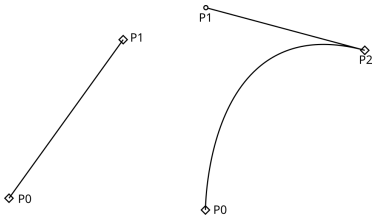


Figure 1.2: *A line segment and bezier curve, with control point*

A quadratic bezier curve is described by the equation

$$\mathbf{B}(t) = (1-t)^2\mathbf{P}_0 + 2(1-t)t\mathbf{P}_1 + t^2\mathbf{P}_2$$

where $\mathbf{P}_0$, $\mathbf{P}_1$, $\mathbf{P}_2$ are the position vectors of the 3 control points for the curve, as t ranges from 0 to 1. We can draw this in a similar way to how we draw line segments: for a sufficient sample rate over t, we can evaluate $\mathbf{B}(t)$ and, rounding to the nearest pixel, plot that point. Similarly, we can describe a line segment (which is effectively a linear bezier curve) using the following equation:

$$\mathbf{B}(t) = \mathbf{P}_0 + t(\mathbf{P}_1 - \mathbf{P}_0)$$

where $\mathbf{P}_0$, $\mathbf{P}_1$ are the two control points, the beginning and end of the line segment.

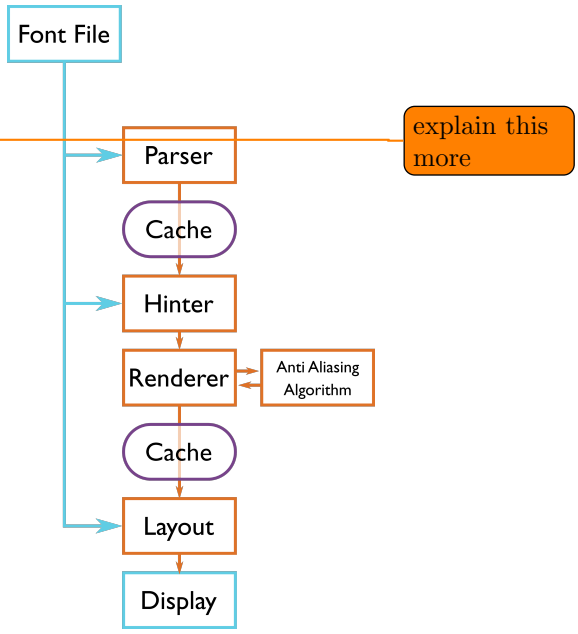
needs pictures

1.1.4 The Font Pipeline

There are several stages involved in getting text to display on the screen. First, the file containing the font is read. Often these are in a standard folder, but all font systems support reading from arbutrary files. The file is parsed (which means broken down into the data that's needed for the specific task) and various tables in the font file are read. The glyph coordinate data will then be parsed and read, usually into a cache.

Next is the hinting step. Because fonts are usually distributed as vectors, yet they are being displayed on a fine pixel grid, naievely rendering them can result in poor font quality. Hinting is an process of distorting the glyph to better fit the pixel grid and size that it will be displayed on. A properly hinted font contains a small program for each glyph, to be executced in a virtual machine, that distorts the vector data to align with the pixel grid. This is a very complex process that is responsible for most of the time rendering the font.

The hinted font is then rendered, taking the bezier curves of the font and creating a raster image. This image is then anti-aliased, probably through supersampling , which is a process that turns a black-and-white image to a greyscale one which looks sharper on the screen. Some advanced anti-aliasing techniques take advantage of the RGB format of our displays, to get a larger effective resolution to work with.



Finally, the anti-aliased image is displayed on the screen. The exact process behind where it should be placed is handled by a layout engine, which is responsible for forming individual glyphs into words and pages. A layout engine still needs to reference the original font file, as kerning information which dictates the individual spacing between characters is stored in a table there.

font file → parser → cache → hinting → rendering → anti-aliasing → layout

1.1.5 The TrueType Format

A TrueType font file is a binary file that contains all the data needed to display a font on the screen. The format, initially codenamed 'Bass', was designed by Apple in the 1980s for Mac System 7, and has continued to evolve and is now used by virtually every consumer operating system and font engine.

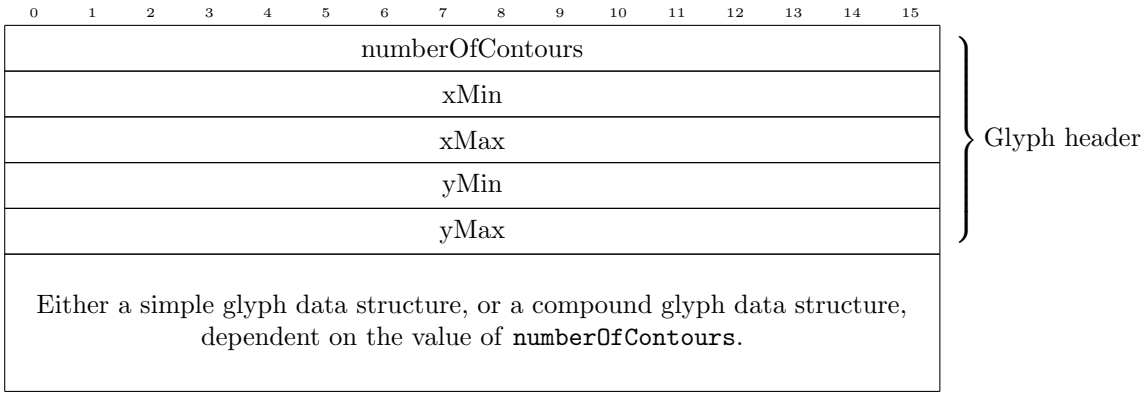
The full specification is available at `https://developer.apple.com/fonts/TrueType-Reference-Manual/`, which will be summarized here. The format is a *binary* file, which means that we look at the file in terms of bytes, rather than in terms of characters and lines of text. The file is composed of a number of *tables*, which are analoguous to chapters of a book. The file begins with the table directory, which is the table of contents for these chapters – it contains their titles, along with where in the file they can be found. For example, the table directory may contain an entry for a table called `glyf`, which points to the area of the file that stores the glyph data for the font.

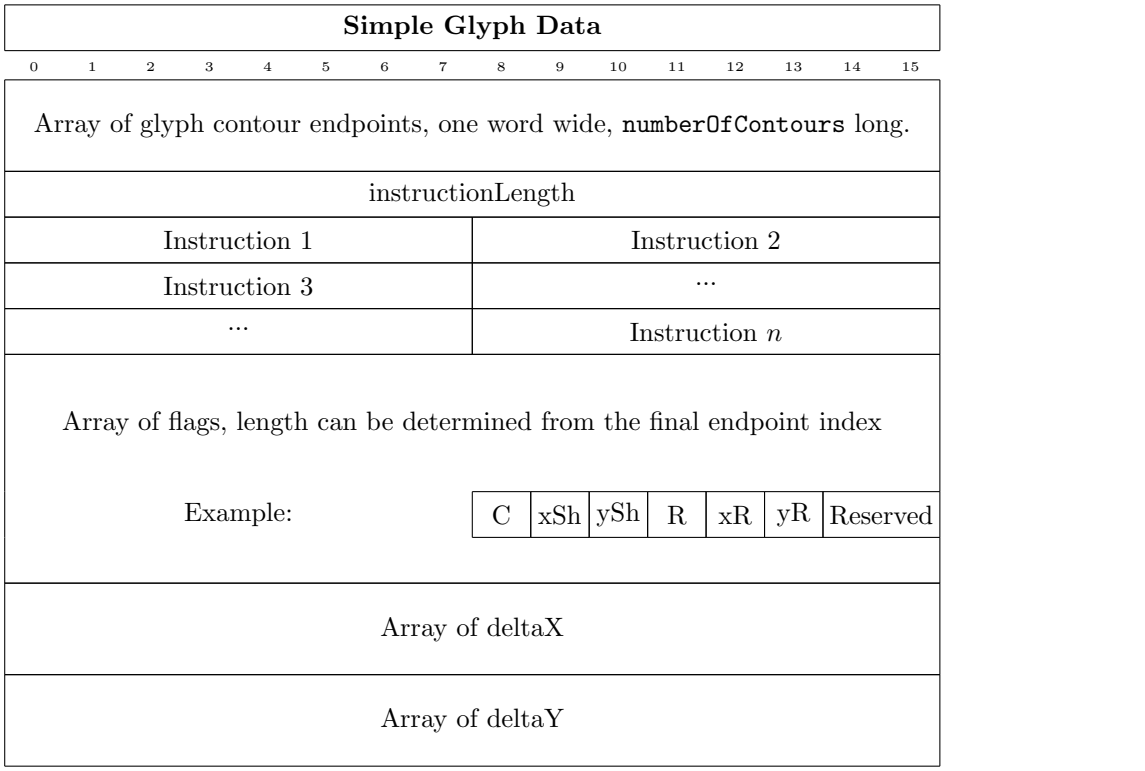
There are several mandatory tables, as shown below. More tables may exist in the file, but are optional, and may contain extensions to the format, or other data such as raster images.

Name	Function
`cmap`	Maps between characters and glyphs
`glyf`	Stores the actual glyph data
`head`	Font header, contains various data involved in the rest of the font
`hhea`	Secondary header containing specifics of horizontal font layout
`hmtx`	Data on the metrics of fonts when layed out horizontally
`loca`	Indexes glyph IDs to offsets in the file
`maxp`	Contains the maximum quantities of various features of the font
`name`	Contains the name of the font
`post`	Data for printing fonts

The tables of relevance to this project are `head`, `loca`, `cmap`, and `glyf`. A detailed explanation of table data structures can be found in the TrueType specification; a simplified version of the `glyf` table is shown below:

The `glyf` table consists of a number of glyph data structures. Each data structure is either a simple or compound glyph, and is of the form





Note: Compound glyph tables are outside of the scope of this project.

To turn a character into a set of points, we must first lookup the character in the `cmap` table. This will give us a glyph index, which we can index into the `loca` table to get a glyph offset and length. Finally, we can use the offset and

length to find the appropriate section of the `glyf` table, and get the glyph data itself.

1.1.6 Hinting

The TrueType format provides support for font hinting, which is a set of small adjustments to the generated outline before it is displayed on the screen to make it better fit the pixel grid. These are implemented through a virtual machine, which runs a set of instructions that alter the font metrics and points at runtime.

1.2 Existing Solutions

Since we can see text on a screen, there are clearly several existing solutions to font rasterization. A few prominent examples are outlined below.

1.2.1 freetype

Freetype is a free, open source font engine that is used in many large software projects, such as GNU/Linux, iOS and Postscript. It supports the vast majority of font features and formats, including little-used ones such as .FON files and X11 PCF fonts. However, due to this, it is very large, and rather slow.

Advantages:

- Can handle virtually any font format you may need.
- A full implementation of the TrueType specification, meaning it supports all of the features of the specification, as well as a number of vendor specific extensions.
- Will run on a wide variety of systems, including the 16-bit Atari and Amiga computers.
- Supports a wide range of character encodings, including the full unicode range of encodings, such as UTF-8 and UTF-16.

Disadvantages:

- Slow, due to it's support of the full specification.
- Has a large memory footprint, meaning there is less space available for other programs.
- Has a relatively complex API, that can be harder to integrate into applications already using a different font engine.
- It does not handle complex layouts, which may be useful for some users.

1.2.2 SDL_ttf

SDL_ttf is a ready to use library for rendering fonts to a texture using the SDL graphics library. Internally, it uses the freetype library as discussed above.

Advantages:

- Simple to integrate into (SDL) applications.
- Uses the freetype backend, which supports a wide range of font features.
- Can be compiled for any platform, due to the fact that it's written using crossplatform libraries and C++.

Disadvantages:

- Slower than using freetype alone, because it adds SDL bindings on top of it.
- Can only render on the CPU, failing to take advantage of any GPU that may be present.
- Since it uses SDL for rendering, it can be quite hard to use with a non SDL project.
- Due to it's simple API, it fails to expose many advanced features of freetype, that might be useful to a client.

1.2.3 Quartz

Quartz is the font renderer used on MacOS and iOS devices, for almost all software. It uses subpixel positioning, which means each glyph doesn't have to be aligned with the pixel grid, instead being allowed to be anywhere between pixels, using subpixel rendering and anti-aliasing to properly handle this case.

Whilst this can result in clearer display at high dpi resolutions, (such as Apple's retina displays), at lower dpi monitors or smaller font sizes it can result in harder to read text.

Advantages:

- Provides the closest similarity to printed text when viewed on high DPI displays.
- Is included by default with MacOS and iOS

Disadvantages:

- Will only run on MacOS / iOS devices
- The subpixel grid can make fonts look blurry at low DPI monitors.

1.3 User Feedback

I consulted with a potential user of my project, to assist in determining what my priorities should be. They use computers for programming, revision, and entertainment, so they spend a lot of time looking at computer-generated text. They also program GUI based applications that need to display text on the screen, so they are the target user for this system. Their responses are shown below in italics:

Do you notice the speed of font rendering on your computer?

Usually no, as most programs render at small resolutions. However, some programs such as my PDF viewer (Zathura) take a very long time rendering fonts when zoomed way in.

Do you notice the quality of font rendering on your computer?

Usually it is perfectly fine, but occasionally kerning issues.

What features would you want from a typeface rendering system?

Fast and accurate rendering, the ability to output SVGs, support for common modern fonts, a good API for integrating into other libraries or programs, compatibility with SDL/OpenGL libraries, debugging tools for viewing path tracing.

Is speed a concern in a font renderer?

Definitely. As soon as it becomes less-than-instant it becomes a productivity bottleneck and can make navigating an operating system frustrating.

Would you rather have increased speed of a font rendering system, or support of more advanced typographical features?

A balance of both. Performance should be considered, but not over every other feature.

I used these responses along with my analysis above to determine what, and what priority, I should give to my objectives below.

1.4 Project Objectives

#	**Priority**	**Description**	**Reasoning**
1	High	The system will accept correctly formed TTF files as input.	The TrueType format has been chosen because it is the industry standard, and almost every font available today can be obtained in TTF format. It also has a rigid specification, and support from other font rendering systems.

2	Medium	The system will reject files that do not exist, or are not correctly formed TTF files.	Cleanly handling errors is important for usability and stability.
3	High	The system will correctly parse the tables `head`, `cmap`, `loca`, and `glyf` to read the font.	This is critical functionality, required for any displaying of any character from the font.
4	Low	The system will output information about the tables on stdout.	This would be beneficial for both debugging purposes and to demonstrate how the internals of a font work.
5	High	The parser section of the system will output bezier curves that can be sent to the renderer to display the font, or exposed via an API for use in some other application.	Critical functionality that is required to display any fonts at all.
6	High	The system will accept an ASCII character to be the glyph to be parsed and eventually displayed.	This will be part of the demonstration of my renderer, which is additionally useful for testing
7	Medium	The system will render fonts at large point-sizes quickly.	This is a key factor identified by the user consultation, as they said that their current solution was unacceptably slow.
8	Medium	The rendering subsystem will render the curves given to it by the parser at a high readability and with anti-aliasing.	The readability of fonts is an area that is already covered well by existing solutions as discussed above, but simple readability improvements such as anti-aliasing are relatively simple to implement and make a large difference to the final output.

9	Low	The system will export font outlines to SVG.	This is another feature identified in the user consultation, but due to time constraints I may not be able to implement it. However, I will try to design the program in such a way that adding it later will be simple.

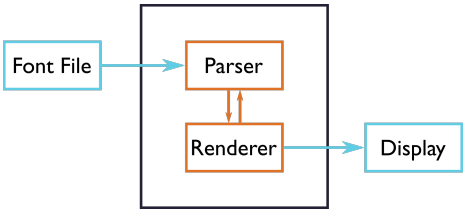
Chapter 2

Design

2.1 High Level Design

The task has 2 clear subdivisions:

1. Parsing the specified font file.
2. Displaying the parsed bezier curves on the screen.



I anticipate task 1 being the hardest, because the TrueType format is very complex and requires sophisticated parsing to extract the data needed. In contrast, rendering bezier curves to the screen is a task is moderately simple, as the bezier curve is used frequently in computer graphics applications.

I have selected C++ as the programming language for this project, because I am familiar with it, and whilst it supports high-level concepts such as object-oriented programming, it is easy to modify variables at the byte level, which is needed for some parsing stages. I use the cmake build system, because it is integrated well into my tools (Intellij CLion). This makes compiling large C++ programs easier, and means I only need to recompile sections of my program that change, which will speed up developement times.

I have also decided to use the SDL 2D graphics library to draw the curves and display the resulting image on the screen. I have used this library before, and it is both fast (it can be used in a mode that is GPU-accelerated) and easy to use. In addition, it means my library can be very easily integrated into any other SDL application. I will also use an additional library, `sdl_gfx` to assist with rendering anti-aliased curves. Note that I am not using the standard version

of the library – instead, I am using a slightly modified version that supports rendering anti-aliased bezier curves.

2.1.1 The Parser

The parser needs to take the filename of a font, and by examining the associated file, read various data about the font. The following is a loose list of the data needed; other data may be needed to properly parse this, or for other purposes.

1. The `head` table, which contains information about the rest of the file
2. The `cmap` table, which contains the mapping between character and glyph index
3. The `loca` table, which maps glyph indices to offsets from the start of the `glyf` table
4. The individual glyph data, which requires the above data points to find in the file
5. Various font metrics, contained in various tables around the font file

At the very beginning of the file is the table directory, which contains the locations and length of all the font tables. Parsing this allows us to look up the locations of the rest of the data in the file. Next we will examine the `head` table to store important metrics that will be used elsewhere in the file.

At this point, the program will prompt the user for the character that they would like displayed. This character will be looked up in the `cmap` table, to get a glyph index, and then the `loca` table, to convert the index into a glyph offset and length.

Finally, we lookup the glyph in the `glyf` table, and store the entire glyph data structure. This is quite complex to parse, and will have it's own section of the program specifically. Parsing this will result in a list of bezier curves to be rendered, which is then passed to the renderer.

The `Font` class

A key data structure in this project will be the `Font` class. This is a large structure that represents a specific font. We create one at the start of our program, and gradually fill it in over the course of parsing the font.

The `Font` class		
Type	**Name**	**Access**
`Glyph`	glyph	public
`std::string`	filename	private
`Header`	header	private
`HEADTable`	head	private
`CMAPTable`	cmap	private
`std::vector<uint8_t>*`	data	private
`int`	fileLength	private
`char`	characterToGet	private
`uint32_t`	glyphOffset	private
`uint32_t`	glyphLength	private
Signature	**Returns**	**Access**
`Font()`	Font	public
`Font(std::string filename, char characterToGet)`	Font	public
`readFont(std::string filename)`	void	private

The most important section of this class is the `readFont` method. It is called in the constructor of the font, and does the following things:

1. Opens the file speficied by `filename`.
2. Reads that file into the `data` variable in memory.
3. Parses the header of the font to get the offsets of the tables.
4. Parses the HEAD table into the the `head` variable.
5. Parses the CMAP table into the `cmap` variable.
6. Uses the `cmap` table to get the glyph index of the requested glyph.
7. Looks up this glyph index in the `loca` table.
8. Parses the glyph using that information.

This prepares the font to be passed to the renderer, where the public member `glyph` will be read by the renderer itself to draw the font to the screen.

Parsing Fonts

The TTF specification uses a number of different types of data, which must be parsed from a series of bytes. My project has a utility module that contains various functions to take a data array and an offset, and can return a requested data type from that data. For example, `parse16` will return a signed 16-bit integer parsed from the bytes after the offset given. These functions are used throughout the project, and the util module is included in nearly every file of the parsing system.

The Glyph Table

The core algorithm of the parsing section is the glyph parser. To save space, data in the glyph itself is not repeated, instead a special flag is set to tell the parser to repeat this data again. In addition, we must calculate when to switch from parsing x values to parsing y values, which requires special calculations to do. The algorithm in pseudocode is as follows:

```
Glyph::parse(data, offset, length):
  numberOfContours = parse an unsigned 16bit integer at (offset)
  xMin = parse an unsigned 16bit integer at (offset+2)
  yMin = parse an unsigned 16bit integer at (offset+4)
  xMax = parse an unsigned 16bit integer at (offset+6)
  yMax = parse an unsigned 16bit integer at (offset+8)

  if numberOfContours < 0:
    abort() (compound glyphs are outside the scope of the program)

  endPointsOfContours = empty vector
  for (i = 0; i < numberOfContours; i++):
    parse a 16bit integer at (offset+10+i*2) and append it to   \
    endPointsOfContours

  instructionOffset = offset+10+numberOfContours*2
  instructionLength = parse an unsigned 16bit integer at
                      (instructionOffset)

  dataOffset = instructionOffset + instructionLength
  currentOffset = dataOffset

  totalPoints = the last value of endPointsOfContours

  flags = empty vector
  xDeltas = empty vector
  yDeltas = empty vector

  currentOffset += 2

  while numberOfPoints < totalPoints:
    construct a PointFlag at the currentOffset and append it to  \
    the flags vector
    numberOfPoints++
    currentOffset++

  currentOffset--

```

```
39    for each flag in flags:
40      xDelta = 0
41      if xShortVector is set in flag:
42        currentOffset++
43        xDelta = parse an unsigned 8bit integer at (currentOffset)
44        if sign is not set in flag:
45          xDelta = -xDelta
46      else:
47        if xSame is not set in flag:
48          currentOffset++
49          xDelta = parse a signed 16bit integer at (currentOffset)
50          currentOffset++
51        else: (if xSame is set)
52          xDelta = 0
53      append xDelta to xDeltas
54
55    repeat the above loop for the yDeltas
56
57    for each i in range(length(xDeltas)):
58      point = (xDelta, yDelta, flag)
59      append point to points
60
61    return points
```

EXPLAIN THIS PROPERLY

2.1.2 Rendering

The rendering system is comparitively much simpler. We take the points array returned by the algorithm above, and draw it to the screen. To make this process simpler, we use a slightly modified version of the `sdl_gfx` library. This library has been modified to allow for anti-aliased bezier curves, which previously the library did not support.

First, we iterate through the list of points to convert all of the delta values into absolute coordinates. During this process, we also insert phantom points between two off-curve points, to allow the drawing routine to look at the points in sets of two or three. Finally, we add the first point again at the end of the loop, to ensure that the loop gets closed.

Now that we have prepared our final list of points, we iterate through each point, drawing either a line or a bezier curve (using the `sdl_gfx` library functions) as required. These are drawn onto the back framebuffer, which is then made the active framebuffer (and displayed on the screen) with the call to `SDL_RenderPresent` in `main()`.

Chapter 3

Technical Solution

Contents

3.1 cmakelists.txt

```
cmake_minimum_required(VERSION 3.15)
project(NEA)
list(APPEND CMAKE_MODULE_PATH ${CMAKE_CURRENT_SOURCE_DIR})
set(CMAKE_CXX_STANDARD 17)

find_package(SDL2 REQUIRED)
#find_package(SDL2_gfx REQUIRED)

```

```
include_directories(${SDL2_INCLUDE_DIRS})

add_executable(NEA SDL_gfx/SDL2_gfxPrimitives.c
    SDL_gfx/SDL2_gfxPrimitives.h src/main.cpp src/Font.cpp
    include/Font.h src/Header.cpp include/Header.h src/util.cpp
    include/util.h src/TableHeader.cpp include/TableHeader.h
    src/HEADTable.cpp include/HEADTable.h src/CMAPTable.cpp
    include/CMAPTable.h src/Glyph.cpp include/Glyph.h
    src/PointFlag.cpp include/PointFlag.h src/Point.cpp
    include/Point.h  SDL_gfx/SDL2_rotozoom.c
    SDL_gfx/SDL2_rotozoom.h src/log.cpp include/log.h)

target_link_libraries(NEA ${SDL2_LIBRARY})
```

3.2 main.cpp

```
#include <iostream>
#include "../include/Font.h"

#include <SDL2/SDL.h>
#include "../SDL_gfx/SDL2_gfxPrimitives.h"
#include <algorithm>

const int WIDTH = 400;
const int HEIGHT = 400;

const double SCALE = 0.1;

void drawFont(Font font, SDL_Renderer* renderer);

int main() {

    SDL_Window* window = nullptr;         // Create pointers for
    ↪  our window and renderer -- these will remain as null
    SDL_Renderer* renderer = nullptr;    // if window or renderer
    ↪  creation fails.

    SDL_Init(SDL_INIT_VIDEO);             // Attempt to initialize
    ↪  SDL

    window = SDL_CreateWindow( "Font Renderer",         // Attempt
    ↪  to create the window...
                               SDL_WINDOWPOS_UNDEFINED,
```

```
                                SDL_WINDOWPOS_UNDEFINED,
                                WIDTH, HEIGHT,
                                SDL_WINDOW_SHOWN );

    renderer = SDL_CreateRenderer(window, 0,
    ↪  SDL_RENDERER_ACCELERATED);  // ...and the renderer

    // Render a white background
    SDL_SetRenderDrawColor(renderer, 255, 255, 255, 255);
    SDL_RenderClear(renderer);

    char characterToDraw = 'A';

    std::string filename = "Roboto.TTF";
    Font font = Font(filename, characterToDraw);    // Create the
    ↪  font object (this will also parse the file)


    bool quit = false;
    // Do things here
    while(!quit) {
        SDL_SetRenderDrawColor(renderer, 255, 255, 255, 255);
        SDL_RenderClear(renderer);        // Clear the screen

        drawFont(font, renderer);  // Render our font to the
        ↪  buffer
        SDL_RenderPresent(renderer);    // Show the buffer on the
        ↪  screen
        SDL_StartTextInput();
        SDL_Event ev;
        while (SDL_PollEvent(&ev)) {
            if(ev.type == SDL_TEXTINPUT) {
                characterToDraw = ev.text.text[0];
                font = Font(filename, characterToDraw);
                SDL_StopTextInput();
                SDL_StartTextInput(); // Restart text input to
                ↪  avoid any weird text editing stuff
            }

            if(ev.type == SDL_QUIT) {
                quit = true;
            }
        }
    }

    SDL_DestroyRenderer(renderer);
```

```
65      SDL_DestroyWindow(window);
66      SDL_Quit();
67  }
68
69
70  void drawFont(Font font, SDL_Renderer *renderer) {
71      // Pick a sensible place to start
72      double locX = 1400;
73      double locY = -2500;
74
75      std::vector<Point> points = {};
76      std::vector<int> endpoints = {};
77
78      for(int i=0; i<font.glyph.points.size(); i++) {
79
80          // If this point would be the last point of a contour, we
            ↪  need to not draw a line here.
81          // We add it the current index to the endpoints array so
            ↪  we know when not to draw a line.
82          if (std::find(font.glyph.endPointsOfContours.begin(),
            ↪  font.glyph.endPointsOfContours.end(), i) !=
            ↪  font.glyph.endPointsOfContours.end()) {
83              endpoints.push_back(points.size());
84          }
85
86          Point* thisPoint = &font.glyph.points.at(i);
87          Point* nextPoint = nullptr;
88          try {
89              nextPoint = &font.glyph.points.at(i+1);
90          } catch (std::out_of_range &e) {
91              nextPoint = &font.glyph.points.at(0); // If we're at
                ↪  the last glyph, go back to the start
92          }
93
94          // Insert the point
95          locX += thisPoint->deltaX;
96          locY += thisPoint->deltaY;
97          points.emplace_back(locX, -locY, thisPoint->flag);
98
99
100
101         if(!thisPoint->flag.onCurvePoint &&
            ↪  !nextPoint->flag.onCurvePoint) {
102             // Insert a phantom point between the two
103             auto deltaX = nextPoint->deltaX / 2;
104             auto deltaY = nextPoint->deltaY / 2;
```

```
105
106             points.emplace_back(locX+deltaX, -(locY+deltaY),
                ↪  PointFlag(0b00000001));
107         }
108
109     }
110
111     // Sort the points into contours
112     std::vector<std::vector<Point>> contours;
113
114     int i = 0;
115     for(auto maxI : endpoints) {
116         std::cout << "Starting contour, will end at point #" <<
            ↪  maxI << std::endl;
117         std::vector<Point> currentContour = {};
118         int s = i;
119         for(; i <= maxI; i++) {
120             std::cout << "Adding point #" << i << std::endl;
121             currentContour.emplace_back(points[i]);
122         }
123         std::cout << "Readding point #" << s << std::endl;
124         currentContour.emplace_back(points[s]);
125         currentContour.emplace_back(points[s]);
126         contours.push_back(currentContour);
127     }
128
129     std::cout << "Done adding points" << std::endl;
130
131     // Draw each contour to the screen
132     for(auto contour : contours) {
133         std::cout << "Starting new contour" << std::endl;
134         for(int j=0; j < contour.size()-1; j++) {
135             Point* currentPoint = &contour[j];
136             Point* controlPoint = nullptr;
137             Point* nextPoint;
138             if(!contour[j+1].flag.onCurvePoint) {
139                 controlPoint = &contour[j+1];
140                 nextPoint = &contour[j+2];
141                 j++;
142             } else {
143                 nextPoint = &contour[j+1];
144             }
145
146             if(controlPoint) {
```

```
147                 std::cout << "Drawing curve from (" <<
                    ↪  currentPoint->deltaX << ", " <<
                    ↪  currentPoint->deltaY << ") to (" <<
                    ↪  nextPoint->deltaX << ", " <<
                    ↪  nextPoint->deltaY << ").\n";
148                 double pX[] = {
149                         static_cast<double>(currentPoint->deltaX)
                            ↪  * SCALE,
150                         static_cast<double>(controlPoint->deltaX)
                            ↪  * SCALE,
151                         static_cast<double>(nextPoint->deltaX) *
                            ↪  SCALE
152                 };
153
154                 double pY[] = {
155                         static_cast<double>(currentPoint->deltaY)
                            ↪  * SCALE,
156                         static_cast<double>(controlPoint->deltaY)
                            ↪  * SCALE,
157                         static_cast<double>(nextPoint->deltaY) *
                            ↪  SCALE
158                 };
159                 aaBezierRGBA(renderer, pX, pY, 3, 20, 1.0, 0, 0,
                    ↪  0, 255);
160             } else {
161                 std::cout << "Drawing line from (" <<
                    ↪  currentPoint->deltaX << ", " <<
                    ↪  currentPoint->deltaY << ") to (" <<
                    ↪  nextPoint->deltaX << ", " <<
                    ↪  nextPoint->deltaY << ").\n";
162                 aalineRGBA(renderer, currentPoint->deltaX *
                    ↪  SCALE, currentPoint->deltaY * SCALE,
                    ↪  nextPoint->deltaX * SCALE, nextPoint->deltaY
                    ↪  * SCALE, 0, 0, 0, 255);
163             }
164         }
165     }
166 }
```

3.3 util.h

```
1 //
2 // Created by jake on 09/11/2020.
```

```
//

#ifndef NEA_UTIL_H
#define NEA_UTIL_H

#include <cstdint>
#include <vector>
#include <string>

uint32_t parseu32(std::vector<uint8_t>* data, int offset);
uint16_t parseu16(std::vector<uint8_t>* data, int offset);
uint8_t parseu8(std::vector<uint8_t>* data, int offset);

int16_t parse16(std::vector<uint8_t>* data, int offset);

std::string parseString(std::vector<uint8_t>* data, int offset,
↪  int length);

#endif //NEA_UTIL_H
```

3.4 util.cpp

```
//
// Created by jake on 09/11/2020.
//

#include "../include/util.h"
#include <cstring>

uint32_t parseu32(std::vector<uint8_t>* data, int offset) {

    uint32_t val = (uint32_t) data->at(offset);
    val <<= 8;
    val += data->at(offset+1);
    val <<= 8;
    val += data->at(offset+2);
    val <<= 8;
    val += data->at(offset+3);

    return val;
}

uint16_t parseu16(std::vector<uint8_t>* data, int offset) {
```

```
22      uint16_t val = (uint16_t) data->at(offset);
23      val <<= 8;
24      val += data->at(offset+1);
25
26      return val;
27  }
28
29  std::string parseString(std::vector<uint8_t> *data, int offset,
    ↪  int length) {
30      std::string result = std::string("");
31      for(int i=offset; i<offset+length; i++) {
32          result += (char) data->at(i);
33      }
34      return result;
35  }
36
37  uint8_t parseu8(std::vector<uint8_t> *data, int offset) {
38      return data->at(offset);
39  }
40
41  int16_t parse16(std::vector<uint8_t> *data, int offset) {
42
43      auto higher = data->at(offset);
44      auto lower = data->at(offset+1);
45
46
47      auto val = (uint16_t) data->at(offset);
48      val <<= 8;
49      val |= (uint16_t) data->at(offset+1);
50
51      int16_t result;
52      std::memcpy(&result, &val, 2); // Copy two bytes of memory
        ↪  from val -> result
53      // We need to do this because leftshifting a signed integer
        ↪  is UB and c++ doesn't want to reinterpret it
54
55      return result;
56  }
```

3.5 Font.h

```
1   //
2   // Created by jake on 12/10/2020.
```

```
3  //
4
5  #ifndef NEA_FONT_H
6  #define NEA_FONT_H
7  #include <string>
8  #include <vector>
9  #include "Header.h"
10 #include "HEADTable.h"
11 #include "CMAPTable.h"
12 #include "Glyph.h"
13
14 class Font {
15 public:
16     Font();
17     Font(std::string filename, char characterToGet);
18     ~Font();
19
20     Glyph glyph;
21 private:
22
23     void readFont(std::string filename);
24
25     std::string filename;
26
27     int fileLength;
28     std::vector<uint8_t>* data;
29
30     Header header;
31     HEADTable head;
32     CMAPTable cmap;
33
34     char characterToGet;
35     uint32_t glyphOffset;
36     uint32_t glyphLength;
37 };
38
39 std::pair<uint32_t, uint32_t>
   ↪ getGlyphOffset(std::vector<uint8_t>* data, Header header,
   ↪ HEADTable head, uint16_t glyphIndex);
40
41 #endif //NEA_FONT_H
```

3.6 Font.cpp

```cpp
#include <fstream>
#include <filesystem>
#include <iostream>
#include "../include/Font.h"
#include "../include/util.h"
#include "../include/log.h"

Font::Font() {
}

Font::Font(std::string filename, char characterToGet) :
    filename(filename), characterToGet(characterToGet) {
    info("Opening font: ", filename);
    info("-> Looking for character: ", std::string(1,
        characterToGet));

    readFont(filename);
}

void Font::readFont(std::string filename) {


    std::ifstream file = std::ifstream(filename, std::ios::in |
        std::ios::binary);
    // Check if the file opened successfully
    if (file.fail()) {
        info("Error: Could not open font file!", "");
        std::abort();
    }

    std::filesystem::path path = std::filesystem::path(filename);
    fileLength = std::filesystem::file_size(path);
    data = new
        std::vector<uint8_t>((std::istreambuf_iterator<char>(file)),
        std::istreambuf_iterator<char>());

    info("-> Read " + std::to_string(data->size()) + " bytes.",
        "");

    // Parse the header! This is used to get the table offsets
        for the rest of the parsing.
    header = Header(data);

```

```
37      head = HEADTable(data, header);
38      cmap = CMAPTable(data, header, characterToGet);
39
40      // So now, we've got the glyph index we want, and need to use
        ↪  the loca table to convert that into a glyph offset and
        ↪  length!
41      auto glyphOffsetAndLength = getGlyphOffset(data, header,
        ↪  head, cmap.glyphIndex);
42      glyphOffset = glyphOffsetAndLength.first;
43      glyphLength = glyphOffsetAndLength.second;
44
45      // And now we need to actually parse the glyph!
46      glyph = Glyph(data, glyphOffset, glyphLength);
47
48
49  }
50
51  // Mixing levels of abstraction = bad, I know, but parsing the
    ↪  loca table is too trivial to get it's own class imo
52  std::pair<uint32_t, uint32_t> getGlyphOffset(std::vector<uint8_t>
    ↪  *data, Header header, HEADTable head, uint16_t glyphIndex) {
53      // Find the offset of the loca table
54      uint32_t offset = header.tables["loca"].tableOffset;
55
56      info("Glyph Index: ", std::to_string(glyphIndex));
57
58      uint16_t glyphPointerMultiplier = head.indexToLocFormat ? 1 :
        ↪  2; // why is the spec life this :(
59      uint16_t glyphPointerSize = head.indexToLocFormat ? 4 : 2;
60
61      uint16_t locaOffset = offset + glyphIndex*glyphPointerSize;
62
63      uint32_t glyphOffset;
64      uint32_t nextGlyphOffset;
65
66      if (glyphPointerSize == 2) {
67          glyphOffset = parseu16(data, locaOffset);
68          nextGlyphOffset = parseu16(data, locaOffset+2);
69
70      } else {
71          glyphOffset = parseu32(data, locaOffset);
72          nextGlyphOffset = parseu32(data, locaOffset+4);
73      }
74
75      glyphOffset *= glyphPointerMultiplier;
76      nextGlyphOffset *= glyphPointerMultiplier;
```

```
77
78      uint32_t length = nextGlyphOffset - glyphOffset;
79
80      glyphOffset += header.tables["glyf"].tableOffset;
81
82      // huh okay that was less trivial than I expected
83      return std::pair(glyphOffset, length);
84  }
85
86  Font::~Font() {
87
88  }
```

3.7 Header.h

```
1   //
2   // Created by jake on 09/11/2020.
3   //
4
5   #ifndef NEA_HEADER_H
6   #define NEA_HEADER_H
7   #include <vector>
8   #include <map>
9   #include <cstdint>
10  #include <string>
11  #include "TableHeader.h"
12
13  class Header {
14
15  public:
16      Header() = default;
17      Header(std::vector<uint8_t>* data);
18
19      ~Header() = default;
20
21      std::map<std::string, TableHeader> tables;
22  private:
23      uint32_t fontType;
24      uint16_t numTables;
25  };
26
27
28  #endif //NEA_HEADER_H
```

3.8 Header.cpp

```cpp
//
// Created by jake on 09/11/2020.
//

#include "../include/Header.h"
#include "../include/util.h"
#include "../include/log.h"
#include <iostream>

Header::Header(std::vector<uint8_t>* data) {
    fontType = parseu32(data, 0);
    numTables = parseu16(data, 4);
    // skipping searchRange, entrySelector, rangeShift, they're not relevant.

    tables = std::map<std::string, TableHeader>();


    if(fontType != 65536) {
        info("Error: This does not appear to be a TTF file", "");
        std::abort();
    }

    info("-> Enumerating tables...", "");

    // ok, now time to parse each table
    for (int offset=12; offset < numTables*16+12; offset += 16) {
        auto table = TableHeader(data, offset);
        tables[table.tag] = table;
    }
    // Print each table to the screen, because logging is important!

    for (auto table : tables) {
        info("->-> Found table " + table.first + " @ " + std::to_string(table.second.tableOffset), "");
    }

    // And we're done!
}
```

3.9 CMAPTable.h

```
//
// Created by jake on 16/11/2020.
//

#ifndef NEA_CMAPTABLE_H
#define NEA_CMAPTABLE_H

#include <vector>
#include <cstdint>
#include "Header.h"

class CMAPTable {
public:
    CMAPTable();

    CMAPTable(std::vector<uint8_t> *data, Header header, char
    ↪  characterToGet);

    void parse(std::vector<uint8_t> *data, int offset, int
    ↪  length, char characterToGet);

    uint16_t glyphIndex;

private:
    void parseSubtableType4(std::vector<uint8_t> *data, int
    ↪  offset, char characterToMap);
};


#endif //NEA_CMAPTABLE_H
```

3.10 CMAPTable.cpp

```
//
// Created by jake on 16/11/2020.
//

#include "../include/CMAPTable.h"
```

```cpp
#include "../include/util.h"
#include <iostream>

CMAPTable::CMAPTable() {

}

CMAPTable::CMAPTable(std::vector<uint8_t> *data, Header header, char characterToGet) {
    // Look for the table in the header
    TableHeader table = header.tables["cmap"];

    parse(data, table.tableOffset, table.length, characterToGet);
}

void CMAPTable::parse(std::vector<uint8_t> *data, int offset, int length, char characterToGet) {
    // Parsing the cmap table is a bit more complex, there are multiple possible formats it could be in.
    uint16_t numSubtables = parseu16(data, offset+2);

    std::vector<uint16_t> platformIds = {};
    std::vector<uint16_t> platformSpecificIds = {};
    std::vector<uint32_t> offsets = {};

    // Enumerate the subtables
    for (int i=offset+4; i<offset+4+(8*numSubtables); i+=8) {
        platformIds.push_back(parseu16(data, i));
        platformSpecificIds.push_back(parseu16(data, i+2));
        offsets.push_back(parseu32(data, i+4));
    }

    uint16_t subOffset = 0;

    // Look for a table with type 0/3 or 0/4 (unicode)
    for (int i=0; i<numSubtables; i++) {
        if(
                platformIds[i] == 0 &&
                (platformSpecificIds[i] == 4 || platformSpecificIds[i] == 3)
        ) {
            subOffset = offsets[i];
            break;
        }
    }

```

```
48      // If we didn't find a suitable subtable
49      if (subOffset == 0) {
50          std::abort(); // TODO: proper error handling
51      }
52
53
54
55      // So now we know which subtable to use, and we have it's
        ↪  offset
56      subOffset = subOffset + offset;
57      // Find the subtable format
58      uint16_t subtableFormat = parseu16(data, subOffset);
59
60      // Right now, my font only has a type 4 table
61      // TODO: Add support for more subtable types
62      if(subtableFormat == 4) {
63          parseSubtableType4(data, subOffset, characterToGet);
64      } else {
65          std::abort(); // TODO: add error handling
66      }
67  }
68
69  void CMAPTable::parseSubtableType4(std::vector<uint8_t> *data,
    ↪  int offset, char characterToMap) {
70      uint16_t subtableLength = parseu16(data, offset+2);
71      uint16_t languageCode = parseu16(data, offset+4);
72      uint16_t segCount = parseu16(data, offset+6) / 2; // This is
        ↪  inexplicably doubled in the file, idk why
73
74      // Next few bytes are weird old optimisation stuff, we can
        ↪  safely ignore them
75
76      std::vector<uint16_t> endCodes = {};
77      for (int i=offset+14; i<offset+14+(2*segCount); i+=2) {
78          endCodes.push_back(parseu16(data, i));
79      }
80      std::vector<uint16_t> startCodes = {};
81      for (int i=offset+16+(2*segCount); i<offset+16+(4*segCount);
        ↪  i+=2) {
82          startCodes.push_back(parseu16(data, i));
83      }
84
85      std::vector<uint16_t> idDeltas = {};
86      for (int i=offset+16+(4*segCount); i<offset+16+(6*segCount);
        ↪  i+=2) {
87          idDeltas.push_back(parseu16(data, i));
```

```
88      }
89      std::vector<uint16_t> idRangeOffsets = {};
90      for (int i=offset+16+(6*segCount); i<offset+16+(8*segCount);
        ↪  i+=2) {
91          idRangeOffsets.push_back(parseu16(data, i));
92      }
93
94      uint16_t startCode;
95      uint16_t endCode;
96      uint16_t idDelta;
97      uint16_t idRangeOffset;
98
99      // Search for the character in the data structures
100     for (int i=0; i<endCodes.size(); i++) {
101         if (endCodes[i] >= characterToMap) {
102             if(startCodes[i] <= characterToMap) {
103                 // We've found our character!
104                 startCode = startCodes[i];
105                 endCode = endCodes[i];
106                 idDelta = idDeltas[i];
107                 idRangeOffset = idRangeOffsets[i];
108                 break;
109             } else {
110                 std::abort(); // oh no
111             }
112         }
113     }
114
115     if (idRangeOffset != 0) {
116         std::cout << "Error: This font uses a nonzero
            ↪  idRangeOffset, which is not supported yet." <<
            ↪  std::endl;
117         std::abort();
118     }
119
120     glyphIndex = idDelta + characterToMap;
121
122 }
```

3.11 HEADTable.h

```
1   //
2   // Created by jake on 09/11/2020.
```

```
3  //
4
5  #ifndef NEA_HEADTABLE_H
6  #define NEA_HEADTABLE_H
7
8  #include <vector>
9  #include <cstdint>
10 #include "Header.h"
11
12 class HEADTable {
13 public:
14     HEADTable();
15
16     HEADTable(std::vector<uint8_t> *data, Header header);
17
18     void parse(std::vector<uint8_t> *data, int offset, int
       ↪  length);
19
20     uint16_t flags;
21     uint16_t unitsPerEm;
22     int16_t xMin;
23     int16_t xMax;
24     int16_t yMin;
25     int16_t yMax;
26
27     bool indexToLocFormat;
28 };
29
30
31 #endif //NEA_HEADTABLE_H
```

3.12 HEADTable.cpp

```
1  //
2  // Created by jake on 09/11/2020.
3  //
4
5  #include "../include/HEADTable.h"
6  #include "../include/util.h"
7  #include "../include/log.h"
8  #include <iostream>
9
10
```

```
void HEADTable::parse(std::vector<uint8_t> *data, int offset, int
↪  length) {

    info("-> Parsing HEAD table @ ", "");

    // Offset +16: flag word
    flags = parseu16(data, offset+16);

    // Offset +18: unitsPerEm
    unitsPerEm = parseu16(data, offset+18);

    // Offset +36: xMin
    xMin = parse16(data, offset+36);
    yMin = parse16(data, offset+38);

    xMax = parse16(data, offset+40);
    yMax = parse16(data, offset+42);

    indexToLocFormat = (bool) parseu16(data, offset+50);
    // Most of the rest of HEAD is kinda irrelevant, checksum,
    ↪  various other properties don't really matter
}

HEADTable::HEADTable() {

}

HEADTable::HEADTable(std::vector<uint8_t> *data, Header header) {
    // Look for the table in the header
    TableHeader table = header.tables["head"];

    parse(data, table.tableOffset, table.length);
}
```

3.13 Glyph.h

```
//
// Created by kernel on 16/11/2020.
//

#ifndef NEA_GLYPH_H
#define NEA_GLYPH_H

```

```cpp
#include <cstdint>
#include <vector>
#include "Point.h"

class Glyph {
public:
    Glyph();
    Glyph(std::vector<uint8_t>* data, uint32_t offset, uint32_t
    ↪  length);

    void parse(std::vector<uint8_t>* data, uint32_t offset,
    ↪  uint32_t length);

    int16_t numberOfContours;
    int16_t xMin;
    int16_t yMin;
    int16_t xMax;
    int16_t yMax;

    std::vector<uint16_t> endPointsOfContours;
    uint16_t instructionLength;

    std::vector<Point> points;

};


#endif //NEA_GLYPH_H
```

3.14 Glyph.cpp

```cpp
//
// Created by jake on 16/11/2020.
//

#include "../include/Glyph.h"
#include "../include/util.h"
#include "../include/log.h"

Glyph::Glyph(std::vector<uint8_t> *data, uint32_t offset,
↪  uint32_t length) : points() {
    parse(data, offset, length);
}
```

```
12
13  Glyph::Glyph() {
14
15  }
16
17  void Glyph::parse(std::vector<uint8_t> *data, uint32_t offset,
    ↪  uint32_t length) {
18      // The number of contours in the glyph
19      numberOfContours = parseu16(data, offset);
20      xMin = parseu16(data, offset+2);
21      yMin = parseu16(data, offset+4);
22      xMax = parseu16(data, offset+6);
23      yMax = parseu16(data, offset+8);
24
25      info("numberOfContours: ", std::to_string(numberOfContours));
26
27      if (numberOfContours < 0) {
28          // Compound glyph, out of the scope of the program
29          abort();
30      }
31
32      endPointsOfContours = {};
33      for (int i=0; i < numberOfContours; i++) {
34          endPointsOfContours.push_back(parseu16(data,
            ↪  offset+10+(i*2)));
35      }
36
37      uint32_t instructionLengthOffset = offset + 10 +
        ↪  numberOfContours*2;
38
39      instructionLength = parseu16(data, instructionLengthOffset);
40
41      // Skip the instructions, they're not used here
42      uint32_t dataOffset = instructionLengthOffset +
        ↪  instructionLength;
43      uint32_t currentOffset = dataOffset;
44
45      uint16_t totalPoints = endPointsOfContours.back() + 1;
46      uint16_t numberOfPoints = 0;
47      std::vector<PointFlag> flags = {};
48      std::vector<int16_t> xDeltas = {};
49      std::vector<int16_t> yDeltas = {};
50
51      // THE BIGGER ALGORITHM
52
53      // Attempt to read each flag
```

```
54      currentOffset+=2;
55      while (numberOfPoints < totalPoints) {
56          flags.emplace_back(PointFlag(data, currentOffset)); //
            ↪  kinda cool that we can just pass stuff like this
57          numberOfPoints += 1;
58          currentOffset++;
59      } // TODO: handle repeating?
60
61      currentOffset -= 1;
62
63      // Parse the X values
64      for(auto flag : flags) {
65          int16_t xDelta;
66          if(flag.xShortVector) {
67              currentOffset++;
68              bool sign = flag.xSame;
69              // Read the single byte of the vector
70              xDelta = parseu8(data, currentOffset);
71              if(!sign) { // If the sign is negative, invert the
                ↪  delta
72                  xDelta = -xDelta;
73              }
74
75          } else {
76              if(!flag.xSame) {
77                  currentOffset +=1;
78                  xDelta = parse16(data, currentOffset);
79                  currentOffset +=1;
80              }
81              else
82                  xDelta = 0;//lastX;
83          }
84
85          xDeltas.emplace_back(xDelta);
86      }
87      // This might just be valid!
88
89      // Parse the Y values
90      for(auto flag : flags) {
91          int16_t yDelta;
92          if(flag.yShortVector) {
93              currentOffset++;
94              bool sign = flag.ySame;
95              // Read the single byte of the vector
96              yDelta = parseu8(data, currentOffset);
```

```
97              if(!sign) { // If the sign is negative, invert the
                ↪  delta
98                  yDelta = -yDelta;
99              }
100
101         } else {
102             if(!flag.ySame) {
103                 currentOffset +=1;
104                 yDelta = parse16(data, currentOffset);
105                 currentOffset +=1;
106             }
107             else
108                 yDelta = 0;//lastY;
109         }
110
111         yDeltas.emplace_back(yDelta);
112     }
113
114     for(int i = 0; i < xDeltas.size(); i++) {
115         points.emplace_back(xDeltas[i], yDeltas[i], flags[i]);
116     }
117
118
119 }
```

3.15 Point.h

```
1  //
2  // Created by jake on 04/12/2020.
3  //
4
5  #ifndef NEA_POINT_H
6  #define NEA_POINT_H
7
8
9  #include "PointFlag.h"
10
11 class Point {
12 public:
13     Point();
14     Point(int16_t deltaX, int16_t deltaY, PointFlag flag);
15
16     int16_t deltaX;
```

```
17      int16_t deltaY;
18
19      PointFlag flag;
20  };
21
22
23  #endif //NEA_POINT_H
```

3.16 Point.cpp

```
1   //
2   // Created by jake on 04/12/2020.
3   //
4
5   #include "../include/Point.h"
6
7   Point::Point() {
8       deltaX = 0;
9       deltaY = 0;
10  }
11
12  Point::Point(int16_t deltaX, int16_t deltaY, PointFlag flag) :
    ↪  deltaX(deltaX), deltaY(deltaY), flag(flag) {}
```

3.17 PointFlag.h

```
1   //
2   // Created by jake on 25/11/2020.
3   //
4
5   #ifndef NEA_POINTFLAG_H
6   #define NEA_POINTFLAG_H
7
8   #include <vector>
9   #include <cstdint>
10
11  class PointFlag {
12  public:
13      PointFlag();
14      explicit PointFlag(uint8_t flagByte);
15      PointFlag(std::vector<uint8_t>* data, uint32_t offset);
```

```
16
17      bool onCurvePoint;
18      bool xShortVector;
19      bool yShortVector;
20      bool repeat;
21      uint8_t repeatCount;
22      bool xSame;
23      bool ySame;
24      bool overlapSimple;
25
26      uint8_t flagByte;
27  };
28
29
30  #endif //NEA_POINTFLAG_H
```

3.18 PointFlag.cpp

```
1   //
2   // Created by jake on 25/11/2020.
3   //
4
5   #include "../include/PointFlag.h"
6
7   PointFlag::PointFlag(std::vector<uint8_t> *data, uint32_t offset)
    ↪  {
8       flagByte = data->at(offset);
9
10      onCurvePoint = flagByte & 0b00000001;
11      xShortVector = flagByte & 0b00000010;
12      yShortVector = flagByte & 0b00000100;
13      repeat       = flagByte & 0b00001000;
14      xSame        = flagByte & 0b00010000;
15      ySame        = flagByte & 0b00100000;
16      overlapSimple= flagByte & 0b01000000;
17      // reserved  = flagByte & 0b10000000;
18
19      if (repeat) {
20          repeatCount = data->at(offset+1);
21      } else {
22          repeatCount = 0;
23      }
24  }
```

```
25
26  PointFlag::PointFlag(uint8_t flagByte) {
27      onCurvePoint = flagByte & 0b00000001;
28      xShortVector = flagByte & 0b00000010;
29      yShortVector = flagByte & 0b00000100;
30      repeat       = flagByte & 0b00001000;
31      xSame        = flagByte & 0b00010000;
32      ySame        = flagByte & 0b00100000;
33      overlapSimple= flagByte & 0b01000000;
34  }
35
36  PointFlag::PointFlag() {
37
38  }
```

3.19 TableHeader.h

```
1   //
2   // Created by jake on 09/11/2020.
3   //
4
5   #ifndef NEA_TABLEHEADER_H
6   #define NEA_TABLEHEADER_H
7
8   #include <string>
9   #include <vector>
10  #include <cstdint>
11
12  class TableHeader {
13  public:
14      TableHeader();
15      TableHeader(std::vector<uint8_t>* data, int offset);
16      ~TableHeader() = default;
17
18      std::string tag;
19      uint32_t checksum;
20      uint32_t tableOffset;
21      uint32_t length;
22  };
23
24
25  #endif //NEA_TABLEHEADER_H
```

3.20 TableHeader.cpp

```cpp
//
// Created by jake on 09/11/2020.
//

#include "../include/TableHeader.h"
#include "../include/util.h"


TableHeader::TableHeader(std::vector<uint8_t>* data, int offset)
    {
    tag = parseString(data, offset, 4);
    checksum = parseu32(data, offset+4);
    tableOffset = parseu32(data, offset+8);
    length = parseu32(data, offset+12);
}

TableHeader::TableHeader() {

}
```

3.21 log.h

```cpp
//
// Created by jake on 14/12/2020.
//

#ifndef NEA_LOG_H
#define NEA_LOG_H

#include <iostream>

void info(const std::string& prompt, const std::string& data);
void error(const std::string& message);

std::string itoh(long val);
#endif //NEA_LOG_H
```

3.22 log.cpp

```cpp
//
// Created by jake on 14/12/2020.
//

#include "../include/log.h"
#include <sstream>

void error(const std::string& message) {
    std::cerr << message << std::endl;
}

void info(const std::string& prompt, const std::string& data) {
    std::cout << prompt << data << std::endl;
}

std::string itoh(long val) {
    std::stringstream s;
    s << std::hex << val;
    std::string result( s.str() );
    return result;
}
```

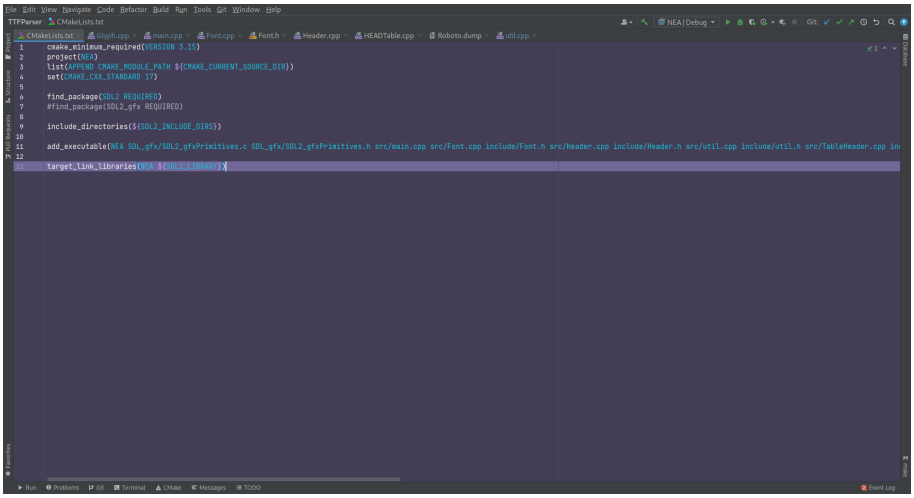
Chapter 4

Testing

To test the project I used a number of different methods to test my code. During the intial programming of the technical solution, I used a debugger to directly step through the code, along with directly modifying the code to print debugging information to the console. I also compared the values various parts of my program produced with the values produced by Freetype, another font rendering system described in the analysis section. Once the solution had been completed, I tested by running the code with a variety of inputs, and visually comparing the output to reference images.

4.0.1 Debugging

For most of the programming work on this project I used the IDE CLion.



This is a highly sophisticated IDE that has many features that make developing

C++ programs easier. The most useful of these is the visual debugger. This allows you to pause the program at any point, inspect and change the values of local and global variables, and generally get more of an understanding of how your program works.

I used debugging extensively throughout this project, for a number of purposes. When I was first writing the various modules, I used debugging as a basic sanity check that my code was running. Because I was using C++, an unsafe language, I was directly manipulating pointers at times, which meant that a Segmentation Fault (when a program tries to read from invalid memory) could occur. The debugger catches this, and I can see the exact line at which the segfault occurs, which makes it significantly easier to identify and resolve the issue.

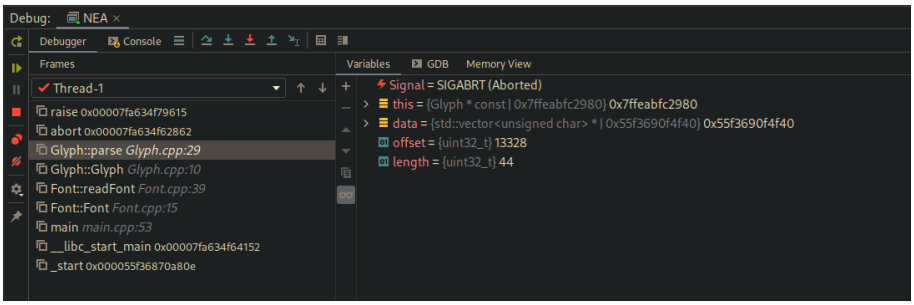


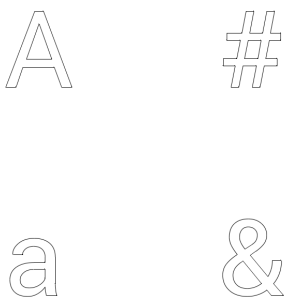
Figure 4.1: *My debugger after a segmentation fault has occured*

4.0.2 Unit Testing

Unit testing is the act of testing a program one unit at a time, so that you can be sure that if all units are working as expected, then hopefully the entire program is working as expected. To accomplish this I created my own sections of TTF files that would test specific areas of the program, and ran it explicitly with those files. This meant that if something broke, I could run my tests to determine which area the problem is, complementing my other debugging using a manual debugger.

4.1 Testing the final program

To test the final program, I used a number of methods. First, I simply observed the output, to check that it resembled the font that I was giving it. I then put the program through a set of tests that measured it's speed, as well as comparing it's output to inkscape, which internally uses freetype as it's font renderer. Finally, I tested it with fonts other than that of Roboto, which I had been using to test the font whilst I was developing the system.



4.1.1 Sample Outputs

The above samples are from my program, running with the font Roboto.ttf. The program supports all alphanumeric characters, along with most of the rest of the displayable ASCII range. The only notable exclusion is the semicolon (';') which is a compound glyph, reusing the glyph for full stop. The parsing of these proved too complicated to implement in the time given, but the architecture of the program means that it could be easily added in later.

4.1.2 Objective 1: The system will accept correctly formed TTF files as input.

I tested this by running my program with several correctly formed TTF files: Roboto, Roboto Italic, Noto Sans, and Hack Mono. The first three files loaded correctly into my program, whereas Hack Mono resulted in an error. Further investigation revealed that Hack Mono used an advanced feature of the `loca` table that my parser did not support. As a result of this testing, I added code to safely handle this error, so that files containing unsupported features are rejected with a descriptive error message rather than the program simply crashing.

4.1.3 Objective 2: The system will reject files that do not exist, or are not correctly formed TTF files.

To test this, I inputted the filenames of nonexistent files and files that are not TTF files into the generator. I tested for rejection of a nonexistent file, a image file called `test.png`, and that same image file renamed to `test.ttf`. In all cases, the program outputs an error message and exits cleanly. Even if the file appears to have a .ttf extension, the program checks the file structure rather than the filename to determine if a file is a TTF file or not.

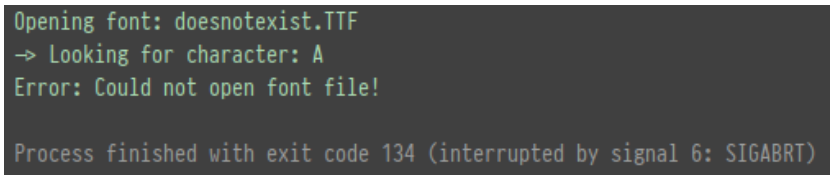


Figure 4.2: *The program rejects invalid or nonexistent files.*

4.1.4 Objective 3: The system will correctly parse the tables head, cmap, loca, and glyf to read the font.

Testing this was quite difficult, because just looking at the font file it is often difficult to determine what the correct outputs of the parser should be. To remedy this, I took advantage of the fact that freetype, one of the existing solutions described in the Analysis section, is open source, which meant I could inspect it's internals. I debugged the freetype library using the same debug tools above, to check that the values reported by freetype were the same as the ones that I was parsing. This was a great help during development, where I couldn't check that the fonts were rendering correctly because the parser and renderer was not complete.

4.1.5 Objective 4: The system will output information about the tables on stdout.

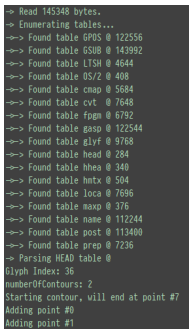
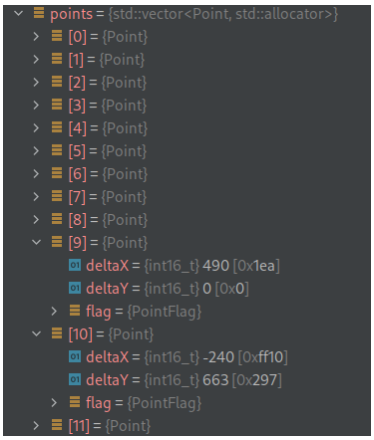


Figure 4.3: *The program outputs various information on the tables on stdout for debugging purposes.*

4.1.6 Objective 5: The parser section of the system will output bezier curves that can be sent to the renderer to display the font, or exposed via an API for use in some other application.

This was achieved by the parser system, which outputs the curves needed to then render the glyph in the rendering section. For example, the curves needed to render the character 'A' are composed of 12 points, as seen in the debug view below:



Testing that these points are correct was done by putting them into the renderer, and observing that the correct glyph was displayed:

Chapter 5

Evaluation