FONT RASTERIZATION ENGINE IN OPENGL

Project Work Report

Submitted in partial fulfillment of the requirements for the degree of

BACHELOR OF TECHNOLOGY in

COMPUTER ENGINEERING

By

GOVIND KRISHNA (06CO24)

RAJAN HARILAL BAPODRA (06CO46)

Under the Guidance of

Mr. Annappa

Assistant Professor

DEPARTMENT OF COMPUTER ENGINEERING

NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA

SURATHKAL, MANGALORE – 575 025

April, 2010

DECLARATION

We hereby declare that the Project Work Report entitled “**Font Rasterization Engine in OpenGL**” which is being submitted to the **National Institute of Technology Karnataka, Surathkal** for the award of the degree of **Bachelor of Technology in Computer Engineering** is a bonafide report of the work carried out by us. The material contained in this Project Report has not been submitted to any University or Institution for the award of any degree.

1. Govind Krishna Raghu (06CO24)

2. Rajan Harilal Bapodra (06CO46)

Place: NITK, Surathkal Date:

CERTIFICATE

This is to certify that the project entitled “**Font Rasterization Engine in OpenGL**” was carried out by **Govind Krishna Raghu (Reg.No.06CO24)** and **Rajan Harilal Bapodra (Reg.No.06CO46)** in partial fulfillment of the requirements for the award of the Degree of Bachelor of Technology in Computer Engineering of National Institute of Technology Karnataka, Surathkal during the academic year 2009-2010.

Mr. Annappa

Project Guide

Place: NITK Surathkal

Date:

ACKNOWLEDGEMENT

The successful completion of our project gives us an opportunity to convey heartfelt regard to each and everyone who has been instrumental in shaping up the final outcome of this project.

We would like to thank Mr. Annappa, Assistant Professor, Department of Computer Engineering, NITK - Surathkal for his able guidance and encouragement during our project work which was of great help in the successful completion of the project.

We would like to thank Mr. Ramachandra Acharya for suggesting the project and guiding us throughout the process. We would also like to express our sincere gratitude to Mr. Sharatchandra Bhagawat and Mr. Gururaj Bhatt of Robosoft Technolgies, Udupi. We thank them for the time spent in discussing and suggesting new solutions during the project.

Finally we are grateful to the Department of Computer Engineering, NITK for providing us this opportunity.

ABSTRACT

The primary aim of this project is to develop an API which enables programmers to use system fonts for rendering text in graphics applications. We are providing this functionality for OpenGL which is an open and cross platform API for rendering 3D graphics. OpenGL provides a lot of graphical power to the programmer for rendering 3D scenes but it is typically at a lower level i.e. it renders the scenes using simple primitives. In order to support platform independence a number of services have been left out like windowing and user interaction. For the same reason OpenGL does not provide direct font support either. Rendering of text is an important aspect in graphics and game development. However the fact that OpenGL does not support text rendering leaves the graphics programmers with limited option for displaying text. They end up using OpenGL's other features for font rendering such as bitmap rendering which usually results in rendering of pixilated low quality text. In order to render high quality text, programmers rely on font libraries. We are developing such a font library using texture mapping approach to render high quality text. Moreover we will also focus on using OpenGL extensions like Vertex Buffer Objects (VBOs) and Frame Buffer Objects (FBOs) to optimize the rendering process. This project is actually a part of a bigger project (Creating a game engine) which Robosoft Solutions Ltd is planning to develop. The font engine is a subpart of the game engine. Our task is to build an easy to use font engine which would be specific to the requirements of the game engine.

TABLE OF CONTENTS:

LIST OF FIGURES:

CHAPTER 1: INTRODUCTION

Rendering of text is an important aspect in graphics and game development. However many 3D games and other 3D applications suffer from the lack of readable text within the scenes they render. This is because the graphics API does not provide direct font support to use system fonts. Thus the programmers have to rely on bitmap rendering of character images which eventually leads to pixilated low quality text. However in some situations the programmer may require high quality text as it adds a real-world richness. Particularly in 3D games, text can convey immersive information. So how does a 3D programmer add high quality text into 3D scenes? Font libraries that interface system fonts in graphics applications is the answer for rendering high quality text. The technique used for text rendering in such libraries is not trivial.

The primary aim of this project is to develop such a font library which enables programmers to use system fonts for rendering text in graphics applications. We are providing this functionality for OpenGL which is an open and cross platform API for rendering 3D graphics. OpenGL provides a lot of graphical power to the programmer for rendering 3D scenes but it is typically at a lower level ie: it renders the scenes using simple primitives. In order to support platform independence a number of services have been left out like windowing and user interaction. For the same reason OpenGL does not provide direct font support either. A separate font engine is required to render high quality fonts.

This project is actually a part of a bigger project (Creating a game engine) which Robosoft Solutions Ltd is planning to develop. The font engine is a subpart of the game engine. Our task is to build an easy to use font engine which would be specific to the requirements of the game engine. The main objective of the API is to act as an interface between fonts on the system (ttf fonts) and the OpenGL application. Also the API has to be cross-platform and text must be of high quality (anti-aliased). High quality font textures can be created using the system's font support, any user supplied ttf fonts or customized textured fonts. Moreover we need to provide support for basic animation of text like translation, rotation, scaling and also API support for style – bold italics and color. The project should also allow users to use their own font textures. It should provide a means of specifying the font texture and also the font texture maps to use customized fonts. In addition to all this we will also focus on using OpenGL extensions like Vertex Buffer Objects (VBOs) and Frame Buffer Objects (FBOs) to optimize the rendering process.

As stated earlier text rendering is not trivial in graphics applications. However there are some common techniques used by existing font libraries for rendering text such as bitmap font rendering, outline font rendering and texture mapping. The approach we are following in the project for rendering text is the Texture Mapped approach. The idea here is to create a texture map that contains all characters of a font. To render an individual character a texture mapped quad is drawn with texture coordinates configured to select the desired individual character. This method is typically faster than bitmaps and outline fonts.

In order to dissect system font files (ttf files) we need a parser that will provide the font face information in an understandable format and also we require a rasterizer to produce bitmap images of the characters. For this purpose we will be using the Freetype engine which allows easy retrieval of glyph data from font files, independent of the format it is stored. FreeType is designed to be small, efficient, highly customizable and portable while capable of producing high-quality output (glyph images). Using these glyph image we can create a texture map of all the characters of the font and then use it to render strings.

The API must give importance to available optimization techniques provided in OpenGL. Rendering high quality text at run time is the challenge in OpenGL. The main disadvantage of existing font libraries is their slow rendering time. So we plan to reduce the rendering time by using OpenGL extensions like Vertex Buffer Objects (VBOs) and Frame Buffer Objects (FBOs). By using VBOs we can put vertex information closer to the GPU thus reducing the transfer bottleneck between the CPU and the GPU.

CHAPTER 2: LITERATURE REVIEW

2.1 OpenGL:

OpenGL is an open and cross platform graphics API for rendering 3D graphics. OpenGL was developed by Silicon Graphics Inc. (SGI) in 1992 and is widely used to create high performance and visually compelling graphics software in markets like CAD, virtual reality, game development, information visualization, and simulation.

OpenGL's basic operation is to accept primitives such as points, lines and polygons, and convert them into pixels. This is done by a graphics pipeline known as the OpenGL state machine. The interface consists of over 250 different function calls which can be used to draw complex three-dimensional scenes from simple primitives such as points, lines and polygons. In addition OpenGL supports lighting and shading, texture mapping, animation and other special effects.

OpenGL is a low-level procedural API rather than a descriptive graphics language. It doesn't provide high-level commands for describing models of three-dimensional objects. With OpenGL, you must build up your desired model from a small set of geometric primitive - points, lines, and polygons. Instead of describing the scene, the programmer describes the steps necessary to achieve a certain appearance or effect.

OpenGL was designed to be graphic output-only: it provides only rendering functions. The core API has no concept of windowing systems, audio, printing to the screen, keyboard/mouse or other input devices. While this seems restrictive at first, it allows the code that does the rendering to be completely independent of the operating system it is running on, allowing cross-platform development. However, some integration with the native windowing system is required to allow clean interaction with the host system. So there must be a separate library to provide window support and hence handing over to OpenGL the drawing control of a window. This can be provided by libraries such as WGL, GLUT, SDL etc.

OpenGL standardizes access to hardware, and pushes the development responsibility of hardware interface programs (device drivers) to hardware manufacturers and windowing functions to the underlying operating system.

2.2 OpenGL Commands:

The OpenGL interface consists of over 250 different function calls which can be used to draw complex three-dimensional scenes from simple primitives such as points, lines and polygons.

OpenGL is a state machine ie: you put it into various states (or modes) that then remain in effect until you change them. For example you can set the current color to white, red, or any other color, and thereafter every object is drawn with that color until you set the current color to something else. There are a number of functions to change the state. The state variables include colour, transformation matrix, line width, normal etc.

Most OpenGL commands either issue primitives to the graphics pipeline, or configure how the pipeline processes these primitives. Internally OpenGL uses a rendering pipeline that processes commands sequentially.

Some of the basic ideas about the OpenGL API and commands are given below:

- OpenGL commands generally follow the following format :
  <library><command name><num & type of parameters>(<parameters>) for eg: glColor3f(1.0,1.0,0.25).

- To make it easier to port OpenGL code from one platform to another, OpenGL defines its own data types for eg: GLbyte, GLshort, GLfloat, GLuint.

- OpenGL is a state machine. States can be enabled and disabled using glEnable() and glDisable(). Eg: glEnable(GL_LIGHTING).

- There are also commands for performing clearing operations eg: glClearColor(), glClearDepth(). These two commands set the values for clearing. Actual clearing is done by glClear().

- OpenGL provides commands to draw geometrical primitives like point, lines, polygons. The geometric information is represented as a set of vertices. Vertices are specified using glVertex() command. When vertex is added, the state variables are also added along with it.
- The vertex commands can only be passed within a block of glBegin() and glEnd(). Only few commands are allowed within this block. The modes are GL_LINES, GL_TRIANGLES, GL_QUADS.

```
glBegin(GL_TRIANGLES);
glColor3f(1.0, 1.0, 1.0);
glVertex3f(0.0, 0.0, 0.0);
glVertex3f(0.0, 1.0, 0.0);
glVertex3f(0.0, 1.0, 0.0);
glEnd();
```

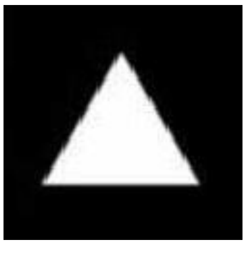


Fig 1: OpenGL commands to draw a triangle.

- The glFlush() function simply tells OpenGL that it should proceed with the drawing instructions supplied thus far before waiting for any more drawing commands.

There are a number of other commands in OpenGL. We have covered some of the basic commands to make us understand OpenGL programs and get us started programming simple applications in OpenGL.

2.3. The Graphics Pipeline:

The graphics pipeline typically accepts some representation of a three-dimensional scene as an input and results in a 2D raster image as output. The input to the graphics pipeline is in the form of vertices. These vertices then undergo transformation and per-vertex lighting. Once transformed and lit, the vertices undergo clipping and rasterization resulting in fragments. The stages of the graphics pipeline are described below:

- **Modelling transformation:** A GPU can specify each logical object in a scene in its own locally defined coordinate system. For this purpose the GPU must first transform all objects into a common coordinate system. This transformation is limited to simple transformations like rotation, translation, scaling.

- **Per-vertex lighting:** Geometry in the complete 3D scene is lit according to the defined locations of light sources and reflectance and other surface properties. Current hardware implementations of the graphics pipeline compute lighting only at the vertices of the polygons being rendered. The lighting values between vertices are then interpolated during rasterization.

- **Viewing transformation:** Objects are transformed from 3D world space coordinates into a 3D coordinate system based on the position and orientation of a virtual camera. This results in the original 3D scene as seen from the camera's point of view defined in what is called *eye space* or *camera space*.

- **Primitives generation:** After the transformation, new primitives (triangles, lines etc) are generated from the vertices that were sent to the beginning of the graphics pipeline.

- **Projection transformation:** In this stage of the graphics pipeline, geometry is transformed from the eye space of the rendering camera into 2D image space, mapping the 3D scene onto a plane as seen from the virtual camera.

- **Clipping:** Geometric primitives that now fall outside of the viewing frustum will not be visible and are discarded at this stage. Clipping accelerates the rendering process by eliminating the unneeded rasterization and post-processing on primitives that will not appear anyway.

- **Scan conversion or rasterization**: Rasterization is the process by which the 2D image space representation of the scene is converted into raster format and the correct resulting pixel values are determined.

- **Texturing, fragment shading:** At this stage of the pipeline individual fragments (or pre-pixels) are assigned a color based on values interpolated from the vertices during rasterization or from a texture in memory.

- **Display:** The final colored pixels can then be displayed on a computer monitor or other display.

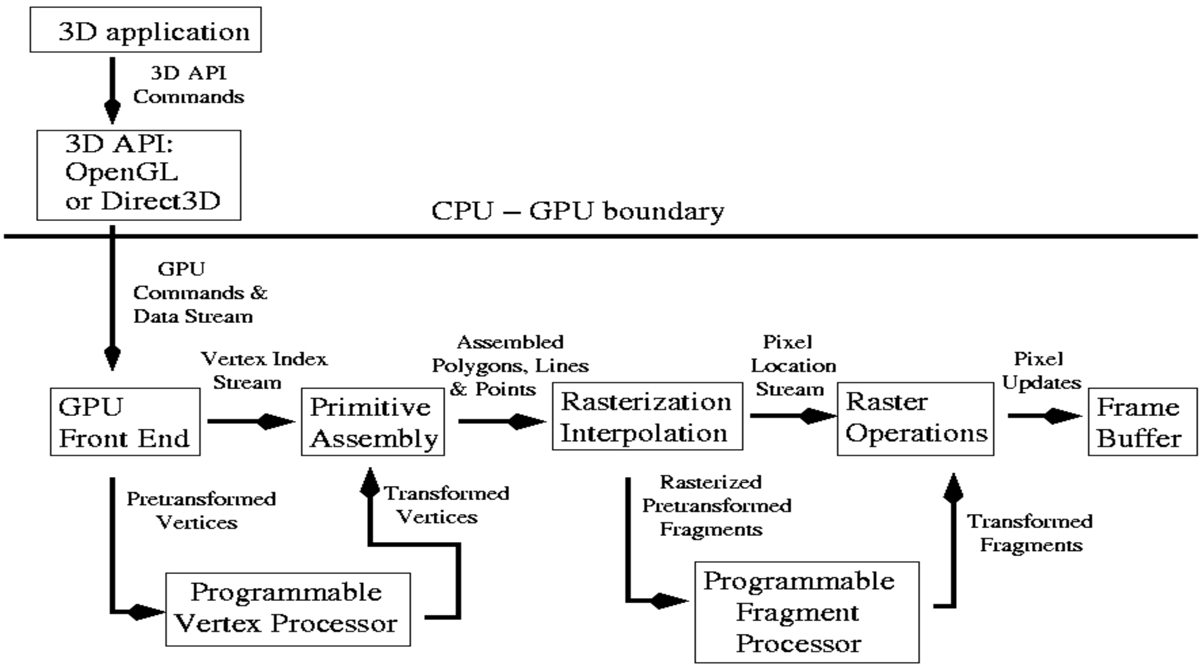


Fig 2: The stages of the graphics pipeline

The graphics pipeline is well suited to the rendering process because it allows the GPU to function as a stream processor since all vertices and fragments can be thought of as independent. This allows all stages of the pipeline to be used simultaneously for different vertices or fragments as they work their way through the pipe.

2.4 Survey of Existing Font Rendering Techniques in OpenGL:

In order to make our own font engine, we have to look into the existing font rendering techniques used by programmers. OpenGL does not provide direct font support, so the programmers need to rely on other graphics libraries to render font.

Basically there are three main approaches for rendering font – Bitmap font, Outline font and Texture mapped approach.

Bitmap fonts are single bit representations of characters of the font. One advantage of bitmap fonts is that it can be rendered quickly. The disadvantage is that they cannot be rotated or scaled and are of low quality.

Outline fonts involves rendering characters as polygons. These polygons can be manipulated in the same way as any other polygon in OpenGL, so it can be rotated, scaled etc. However it is slow. The Outline font approach is generally used for 3D extrusion effects.

The approach we are following in the project is the Texture Mapped approach. The idea here is to create a texture map that contains all characters of a font. To render an individual character a texture mapped quad is drawn with texture coordinates configured to select the desired individual character. This method is typically faster than bitmaps and outline fonts. The figure below shows how a texture image containing certain characters is used to display a string in an OpenGL application.

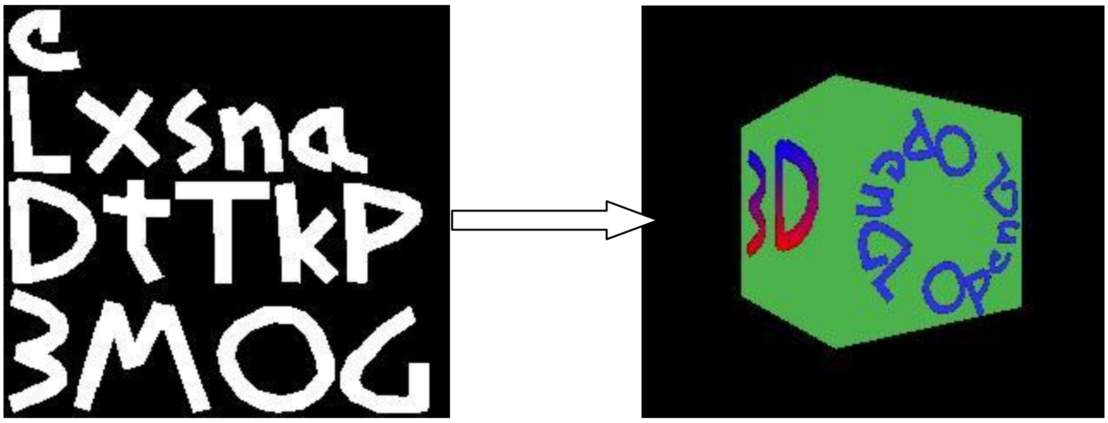


Fig 3: The Texture mapping approach for rendering text

2.5 Font Support in Existing Libraries:

These font rendering techniques are provided by different libraries. Some of the libraries that are most commonly used are analyzed and summarized below:

WGL: (Windows Graphics Library) supports bitmap and outline fonts. The advantage is that you can use any windows system font. The disadvantage is that it is platform dependent.

GLUT: There are two options for drawing fonts in GLUT: Bitmap and Stroke. Advantage is that this approach is very easy to use also platform independent. However the disadvantage is that the choice of fonts is limited.

glFont API and FNT library: These two libraries use the texture mapped approach to render strings. glFont API provides a program to generate a texture from a system font. The disadvantage is that we have to create the fonts beforehand and also it is of low quality. FNT library uses TXF texture font format (pre generated textures) which can be downloaded. The disadvantage here is that there are limited fonts.

GLTT & FTGL: These are wrappers around the Freetype library. Freetype is a free and portable True Type engine. GLTT uses Freetype to extract contours and for rasterization, and provides a simple but powerful API for rendering text. FTGL is a successor to GLTT and it supports all font rendering techniques. The Only disadvantage of these two libraries is the font loading time.

By analyzing the advantages and disadvantages of the approaches used by the font rendering libraries we were able to make a plan for our library. We will be using the texture mapped approach for font rendering considering its advantages. Also we shall be using optimization techniques like using VBOs and FBOs for rendering strings to reduce the font loading time.

2.6 TTF Font Format:

There are 3 basic types of font file formats: Bitmap fonts, Outline fonts, Stroke fonts. The most commonly used format is Outline fonts. Outline fonts (also called vector fonts) use Bezier curves, drawing instructions and mathematical formulas to describe each glyph, which make the character outlines scalable to any size. The TTF font is an example of an outline font format.

Outline fonts have a set of drawing instructions for lines and curves to create a shape for each character (glyph). By following the drawing instructions the computer creates an outline shape at a specific size, and then fills it with ink to create the character. Glyphs scale to any size and are independent of the resolution of the screen.

A True type font file consists of a sequence of concatenated tables. A table is a sequence of words. The first of the tables is the font directory, a special table that facilitates access to the other tables in the font. The directory is followed by a sequence of tables containing the font data. These tables can appear in any order. Certain tables are required for all fonts. Others are optional depending upon the functionality expected of a particular font. The tables have names known as tags. Currently defined tag names consist of four characters. The most common tables are: 'cmap'- character to glyph mapping 'glyf' - glyph data 'head' - font header.

- The font directory, the first of the tables, is a guide to the contents of the font file. It provides the information required to access the data in the other tables. The directory consists of two parts: the offset sub table and the table directory. The offset sub table records the number of tables in the font and provides offset information enabling quick access to the directory tables. The table directory follows the offset sub table. The table directory consists of sequence of entries, one for each table in the font.

- The ’cmap’ table maps character codes to glyph indices. The choice of encoding for a particular font is dependent upon conventions used by the intended platform. A font intended to run on multiple platforms with different encoding conventions will require multiple encoding tables. Character codes that do not correspond to any glyph in the font should be mapped to glyph index 0. At this location in the font there must be a special glyph representing a missing character, typically a box. No character code should be mapped to glyph index -1, which is a special value reserved in processing to indicate the position of a glyph deleted from the glyph stream.

- The 'glyf' table contains the data that defines the appearance of the glyphs in the font. This includes specification of the points that describe the contours that make up a glyph outline and the instructions that grid-fit that glyph. The 'glyf' table supports the definition of simple glyphs and compound glyphs, that is, glyphs that are made up of other glyphs. The number of glyphs in the font is restricted only by the value stated in the 'head' table. The order in which glyphs are placed in a font is arbitrary.

- The 'head' table contains global information about the font. It records such facts as the font version number, the creation and modification dates, revision number and basic typographic data that apply to the font as a whole. This includes a specification of the font bounding box, the direction in which the font's glyphs are most likely to be written and other information about the placement of glyphs in the em square.

To dissect TTF files you will need to be able to write a parser that extracts the information from the TTF file in a more understandable format. Building the parser is a very complex process. The parser could be built incrementally, starting with a process that identifies the blocks of data, and then decodes each block into its components.

2.7 Freetype Library:

FreeType is a software library written in C that implements a font rasterization engine. It is used to rasterize characters into bitmaps and provides support for other font-related operations. It allows client applications to access font files easily, independent of the font format they are stored. It allows easy *retrieval of individual glyph data* (metrics, images, name, encoding/charmaps). FreeType is designed to be small, efficient, highly customizable and portable while capable of producing high-quality output (glyph images). Using Freetype we can make a Font face object which contains the font information in an understandable format. This way we can access individual glyph data. Here are some design details of The Freetype library.

- The **FT_Library class:** This type corresponds to a handle to a single instance of the library. The library object is the parent of all other objects in FreeType 2. You need to create a new library instance before doing anything else with the library. Similarly, destroying it will automatically destroy all its children (i.e. faces and modules).

Typical client applications should call FT_Init_FreeType() in order to create a new library object, ready to be used for further actions.

- The **FT_Face class:** A face object corresponds to a single *font face*, i.e., a specific typeface with a specific style. For example, "Arial" and "Arial Italic" correspond to two distinct faces. A face object is normally created through FT_New_Face().This function takes the following parameters: an FT_Library handle, a C file pathname used to indicate which font file to open, an index used to decide which face to load from the file (a single file may contain several faces in certain cases), and the address of a FT_Face handle.
- The face object contains several fields used to describe global font data that can be accessed directly by client applications. For example, the total number of glyphs in the face, the face's family name, style names, the EM size for scalable formats, etc.
- The **FT_Size class:** Each FT_Face object *has* one or more FT_Size objects. A *size object* is used to store data specific to a given character width and height. Each newly created face object has one size, which is directly accessible as face->size. The contents of a size object can be changed by calling either FT_Set_Pixel_Sizes() or FT_Set_Char_Size().
- The **FT_GlyphSlot class:** The purpose of a glyph slot is to provide a place where glyph images can be loaded one by one easily, independently of the glyph image format (bitmap, vector outline, or anything else). Ideally, once a glyph slot is created, any glyph image can be loaded into it without additional memory allocation. In practice, this is only possible with certain formats like TrueType which explicitly provide data to compute a slot's maximum size. Finally, each face object has a single glyph slot that is directly accessible as face->glyph.
- The **FT_CharMap class:** The FT_CharMap type is used as a handle to character map objects, or *charmaps*. A charmap is simply some sort of table or dictionary which is used to translate character codes in a given encoding into glyph indices for the font.

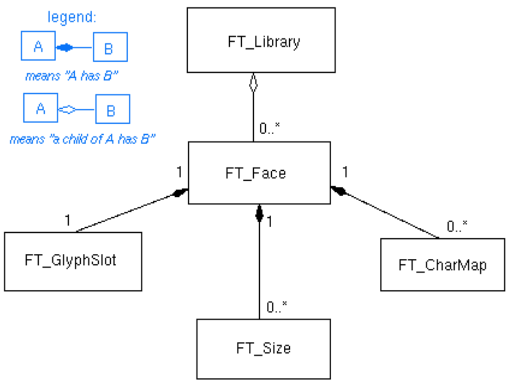


Fig 4: The Freetype library class diagram

2.8 Glyph Metrics:

Dots per Inch (dpi): Device characteristics are expressed in dpi (dots per inch). For example a printer with resolution of 300X600 dpi has 300pixels per inch in horizontal direction, and 600 in the vertical one. The resolution of a typical computer monitor varies with its size (15" and 17" monitors don't have the same pixel sizes at 640X480).

Point size: The size of text is usually given in points, rather than device-specific pixels. Points are a simple physical unit, where 1pt = 1/72th of an inch, in digital typography. It is possible to compute the size of text in pixels from size in points with the following formula: pixelsize = pointsize * resolution / 72. The resolution is expressed in dpi. Since horizontal and vertical resolution may differ, a single point size usually defines a different text width and height in pixels.

Vectorial representation (Outline fonts): The source format of outlines is a collection of closed paths called contours. Each contour delimits an outer or inner region of the glyph, and can be made of either line segments or Bezier arcs.

The arcs are defined through control points, and can be either second-order (these are quadratic Bezier) or third order (cubic Beziers) polynomials, depending on the font format. Each point of the outline has an associated flag indicating its type (normal or control point). And scaling the points will scale the whole outline.

EM Square: In creating the glyph outlines, a type designer uses an imaginary square called the EM square. Typically, the EM square can be thought of as a tablet on which the characters are drawn. It is the reference used to scale the outlines to a given text dimension. For example, a size of 12pt at 300x300 dpi corresponds to 12*300/72 = 50 pixels. This is the size the EM square would appear on the output device if it was rendered directly.

Grid Fitting (Hinting): The outline as stored in a font file is called the "master" outline, as its points coordinates are expressed in font units. Before it can be converted into a bitmap, it must be scaled to a given size/resolution. This is done through a very simple transformation, but always creates undesirable effects, e.g. stems of different widths or heights in letters like "E" or "H".

As a consequence, proper glyph rendering needs the scaled points to be aligned along the target device pixel grid, through an operation called *grid-fitting* (often called *hinting*). One of its main purposes is to ensure that important widths and heights are respected throughout the whole font, as well as to manage features like stems and overshoots, which can cause problems at small pixel sizes.

Base Line, Pens, Layout:

- The baseline is an imaginary line that is used to "guide" glyphs when rendering text.
- To render text, a virtual point, located on the baseline, called the *pen position* or *origin*, is used to locate glyphs.

- With horizontal layout, glyphs simply "rest" on the baseline. Text is rendered by incrementing the pen position, either to the right or to the left.
- The distance between two successive pen positions is glyph-specific and is called the *advance width*.

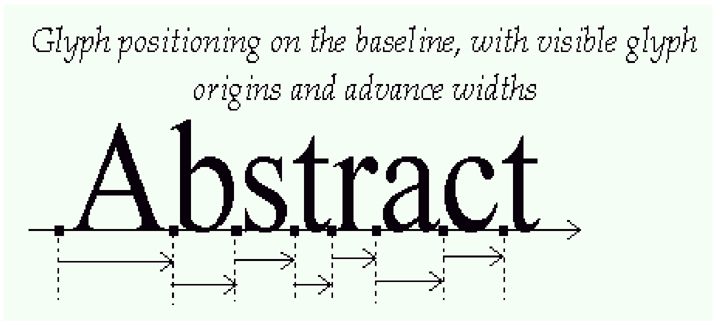

Glyph positioning on the baseline, with visible glyph origins and advance widths
Abstract

Fig 5: Text rendering using advance widths of characters

Typographic metrics and bounding boxes:

- A various number of face metrics are defined for all glyphs in a given font.

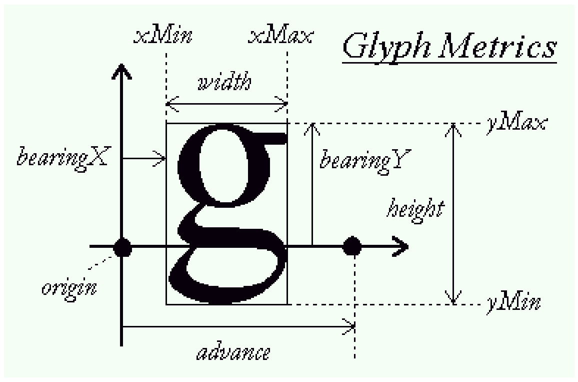


Fig 6: Glyph metrics

- *Ascent*: The distance from the baseline to the highest/upper grid coordinate used to place an outline point. It is a positive value, due to the grid's orientation with the Y axis upwards.
- *Descent*: The distance from the baseline to the lowest grid coordinate used to place an outline point. This is a negative value, due to the grid's orientation.
- *Linegap*: The distance that must be placed between two lines of text. The baseline-to-baseline distance should be computed as: ascent - descent + linegap
- *Bbox: The* glyph's bounding box, also called *bbox*. This is an imaginary box that encloses all glyphs from the font, usually as tightly as possible. It is represented by four fields, namely xMin, yMin, xMax, and yMax.

Each glyph has also distances called *bearings* and *advances*. Their definition is constant, but their values depend on the layout, as the same glyph can be used to render text either horizontally or vertically:

- *BearingX*: The horizontal distance from the current pen position to the glyph's left bbox edge. It is positive for horizontal layouts, and in most cases negative for vertical ones.
- *BearingY*: The vertical distance from the baseline to the top of the glyph's bbox. It is usually positive for horizontal layouts, and negative for vertical ones.
- *Advance width* or *advanceX*: The horizontal distance the pen position must be incremented (for left-to-right writing) or decremented (for right-to-left writing) by after each glyph is rendered when processing text. It is always positive for horizontal layouts, and null for vertical ones.
- *Advance height advanceY*: The vertical distance the pen position must be decremented by after each glyph is rendered. It is always null for horizontal layouts, and positive for vertical layouts.
- *Glyph width*: The glyph's horizontal extent. For unscaled font coordinates, it is bbox.xMax-bbox.xMin. For scaled glyphs, its computation requests specific care, described in the grid-fitting chapter below.
- *Glyph height:* The glyph's vertical extent. For unscaled font coordinates, it is bbox.yMax-bbox.yMin. For scaled glyphs, its computation requests specific care, described in the grid-fitting chapter below.

2.9 Optimization in Graphics Applications:

Traditional software tuning focuses on finding and tuning hot spots - the 10% of code in which a program spends 90% of its time. Computer graphics systems present a pipeline architecture in which 3D data about the scene that is to be displayed passes through different stages. Because these stages operate in parallel, it is appropriate to use a different approach for optimization ie: look for bottlenecks - overloaded stages that are holding up other processes. Depending on the type of application or the nature of the data, it is possible that some of these pipeline stages become a bottleneck, thus drastically reducing the overall performance of graphics devices. It is therefore essential to locate these bottlenecks so that the graphics hardware can operate properly.

The basic strategy for isolating bottlenecks is to measure the time it takes to execute a part or a whole program and then change the source code in ways that add or subtract work at a single point in the graphics pipeline. If changing the amount of work at a given stage does not alter performance appreciably, that stage is not the bottleneck. If there is a noticeable difference in performance, a bottleneck exists.

In our application, we are currently using immediate mode for drawing. Thus vertex information are continuously passed from the CPU to the GPU. The bottleneck for our application exists here, it is known as Transfer-limited bottleneck. Inorder to minimize this transfer, we shall use Vertex Buffer Objects (VBOs). VBOs have advantages over the other methods of specifying geometrical primitives: immediate mode, vertex arrays, display lists and interleaved arrays. Vertex buffer object (VBO) allows vertex array data to be stored in high-performance graphics memory on the server side and promotes efficient data transfer.

2.9.1 Vertex Buffer Objects:

OpenGL traditionally provides four approaches for rendering geometric data – immediate mode, display lists, vertex arrays and interleaved arrays.

When using **immediate mode**, applications send all the geometric data to the graphics processor (GPU) every frame, which is advantageous in situations such as modelling or animation where geometry is frequently created or modified. If geometric data does not change frequently immediate mode can result in wasted data transfer when compared with storing the same geometric data within graphics memory. Immediate mode frequently causes data retrieval, transfer and CPU bottlenecks that inhibit overall graphics performance.

As an alternative to immediate mode, OpenGL provides **display lists**. These enable a series of graphics commands to be grouped together. This gives OpenGL implementations more opportunity to process and store data in ways that can improve overall graphics performance. Display lists can be stored within graphics memory, for example, to avoid transfer over the graphics bus. Despite these benefits, display lists do have some disadvantages. In some situations, geometric data changes require creating a new display list. Depending on the frequency with which geometric data is updated, the potential performance advantages may be outweighed by the complexities of managing creation/deletion of display lists.

As an alternative to display lists, OpenGL also implements **vertex arrays**. These allow vertex and attribute data to be grouped and treated as a block, which promotes some of the data transfer efficiencies afforded by display lists. Vertex arrays also allow data such as geometry and color to be interleaved, which can be convenient when creating and referencing. Unfortunately, vertex arrays prohibit assuming that any individual piece of data will not change. As a result, when drawing an object using vertex arrays, the data in the array must be validated each time it is referenced. This adds overhead into data transfer. Vertex arrays do not suffer, however, from the limitation of storing two copies of all data.

VBOs are intended to enhance the capabilities of OpenGL by providing many of the benefits of immediate mode, display lists and vertex arrays, while avoiding some of the limitations. They allow data to be grouped and stored efficiently like vertex arrays to promote efficient data transfer. They also provide a mechanism for programs to give hints about data usage patterns so that OpenGL implementations can make decisions about the form in which data should be stored and its location. VBOs give applications the flexibility to be able to modify data without causing overhead in transfer due to validation.

The basic idea of this mechanism is to provide some chunks of memory (buffers) that will be available through identifiers. As with any display list or texture, we can bind such a buffer so that it becomes active. Internal memory management can choose the best type of memory (system, video, or AGP), depending on the way we want to use the buffers. VBOs support 3 modes of transfer: GL_STREAM_DRAW (when vertices data could be updated between each rendering), GL_DYNAMIC_DRAW (when vertices data could be updated between each frames), GL_STATIC_DRAW (when vertices data are never or almost never updated).

2.9.2 Frame Buffer Objects (FBO):

The frame buffer object architecture (FBO) is an extension to OpenGL for doing flexible off-screen rendering, including rendering to a texture. By using frame buffer object (FBO), an OpenGL application can redirect the rendering output to the application-created frame buffer object (FBO). By capturing images that would normally be drawn to the screen, it can be used to implement a large variety of image filters, and post-processing effects. It is used in OpenGL for its efficiency and ease of use.

CHAPTER 3: PROBLEM STATEMENT & SCOPE

Rendering of text is an important aspect in graphics and game development. This is because the graphics API does not provide direct font support to use system fonts. Thus the programmers have to rely on bitmap rendering of character images which eventually leads to pixilated low quality text. However in some situations the programmer may require high quality text as it adds a real-world richness. The primary aim of this project is to develop such a font library which enables programmers to use system fonts for rendering text in graphics applications. This project is actually a part of a bigger project (Creating a game engine) which Robosoft Solutions Ltd is planning to develop. The font engine is a subpart of the game engine. Our task is to build an easy to use font engine which would be specific to the requirements of the game engine. The objectives that were stated to us are as follows:

1. To provide a cross-platform Font API for OpenGL.
2. The API will act as an interface between fonts on the system (ttf fonts) and the OpenGL application.
3. Rendering of high quality anti-aliased text.
4. An easy to use API for font generation.
5. Basic transformation support for font display like translation, rotation, scaling.
6. Support for customized texture fonts.
7. Support for different styles: bold, italics and color.
8. Optimization of rendering by using VBOs to bring vertex data closer to the graphical processor.
9. Use of FBOs to optimize the rendering process.

CHAPTER 4: METHODOLOGY/DESIGN USED

4.1 Division of Work:

Our ultimate goal is to provide a Font API for use in an OpenGL application. We have decided to divide this task into two parts. The first part of the project involves making a basic font API without added optimization or additional features. This part involves two subtasks.

(1) One task is to provide a mechanism to read system font files and render all the glyphs (characters) into an image.
(2) The Second task is to load this image as a texture and use it to render strings given as an input by the programmer.

For the first task we created a Font Generator application to test the rendering of glyphs onto an image. The Second task involved using OpenGL to load the image as a texture use texture mapped quads to render a string. We completed both tasks in the 7th semester. However the second task required more fine tuning. During vacations we tested the application for different fonts and worked on providing an easy to use API for the users to draw fonts at a particular location and size. The work during 8th semester focused mainly on optimizing the font rendering process, providing support for custom fonts and adding additional features like transformation support, styles etc.

4.2 Design:

We have come up with a design for the basic font system. As said earlier the basic font system involves two subparts – one is interfacing with font files present in system by creating an image containing all the glyphs, second part is to use this image as a texture in the video memory to render strings given as input by the programmer. Below is a context diagram showing the main function of the Font API which is to draw texture mapped quads containing characters using the texture containing characters of a particular font.

4.2.1 Context Diagram:

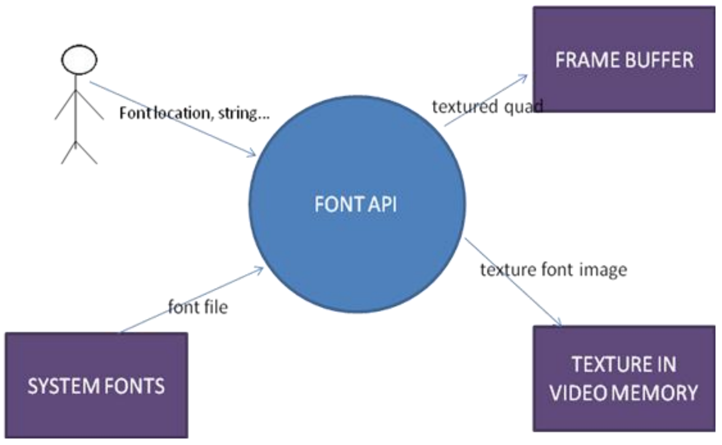


Fig 7: Context Diagram

4.2.2 Data Flow Diagram:

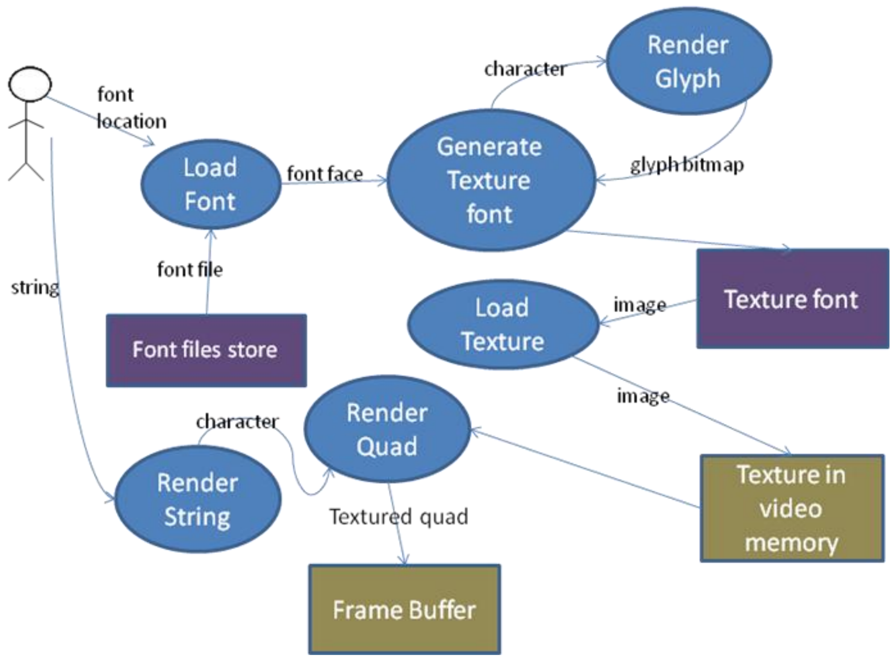


Fig 8: Data Flow Diagram

Above is a data flow diagram showing the working of the API in detail. To use a system font file the programmer specifies the font location. The LoadFont function creates a font face structure which holds information about the glyphs present in the font file in an understandable format. Using this font face the system generates a Texture font ie: an image containing all the glyphs present in the font. This is done by rendering a bitmap image of each glyph (from 32 to 127) into the texture image one after another. After this image is created it can be loaded into the video memory as a texture. The programmer can load a number of fonts together and store it as a texture. When he requires a particular font he can load that texture by referring to the TextureFont id. The next part of the system is to render strings using the texture of the font. The programmer gives the string as the input. Each character in the string is considered and to render an individual character a texture mapped quad is drawn with texture coordinates configured to select the desired individual character.

4.2.3 Entity-Relationship Diagram:

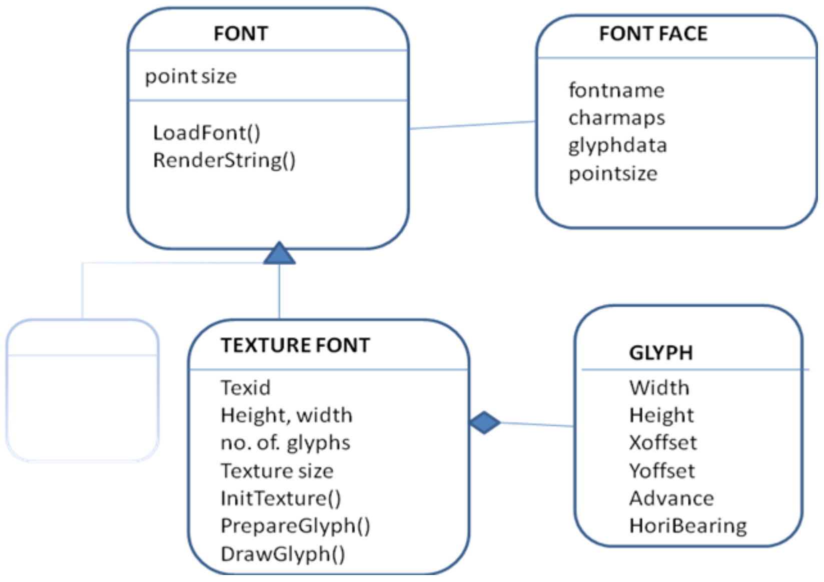


Fig 9: Entity Relationship Diagram

Font Face: Loading the font is done by storing the font information in a structure called font face. This structure contains point size, character mapping, glyph data etc.. The texture font can be created by using the info in the font face.

Texture Font: This structure will contain information on the image that is to be loaded as the texture. Also it will contain info on the glyphs present. This is the main class which the programmer will use to load the font and render strings.

Glyph: A Texture font will be composed of a number of glyphs. Each glyph structure will contain information like offset, width, height, advance width, horizontal bearing etc.

CHAPTER 5: IMPLEMENTATION DETAILS

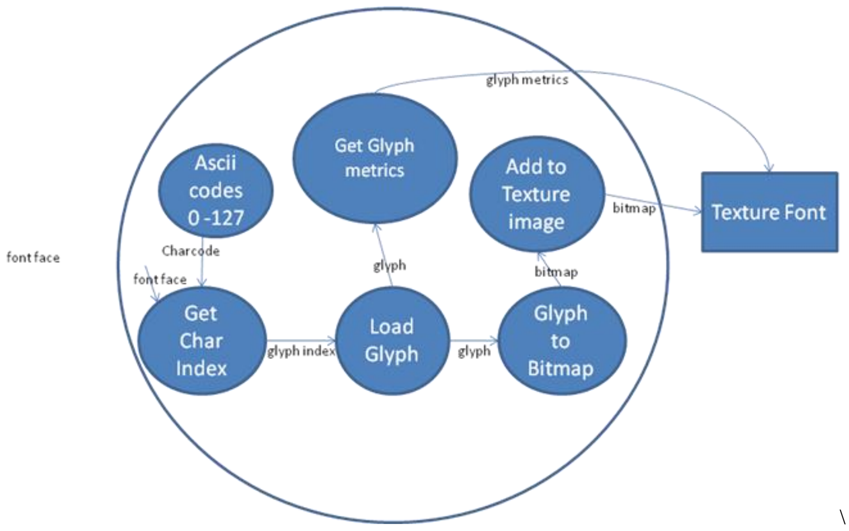
As stated earlier there were two main tasks that had to be accomplished:

1. One task is to provide a mechanism to read system font files and render all the glyphs (characters) into an image.
2. The Second task is to load this image as a texture and use it to render strings given as an input by the programmer.

5.1 Task One – Rendering Glyphs into an Image:

The first task involves reading system font files and rendering all the glyphs into an image. In order to test the output for different inputs we created a Font Generator application which generates an image for a given font file and point size. We are using the Freetype 2.0 library to read font files and render the glyphs.

The process involves loading a new font face structure using the given font file. A character code can then be mapped to the glyph index of that particular character. Using this glyph index we can load a glyph structure corresponding to the character code. This glyph structure is then used to generate a bitmap of the character. This glyph structure also contains glyph metrics that are required to be stored for later use. The process is carried out for all valid characters of the font (32 -127) and the final output is an image containing all the glyphs. Along with this image a text file is also created containing information regarding the metrics of each glyph such as width, height, advance width, horizontal bearing etc. The process is shown below:



\

Fig 10: Data Flow Diagram showing creation of texture map using system fonts

Initially the output didn’t appear as per requirement. After going through the glyph metrics information we were able to draw the glyphs correctly considering advance width and horizontal bearing. So the Font Generator application helps us to test the output of the image being created for different inputs (fonts, point sizes) given by the user. This is needed so that we can decide on the final algorithm to create the image required for texture mapping.

5.2 Task Two – Loading Texture Image and using it to Render Strings:

Now that we have the image ready we have to load it into the video memory as a texture to use it to render strings. For this we use a function called glGenTextures() to get a texture id (say tid). Then we use glBindTexture() to set that (having textureid – tid) as the current texture so that any texture related operation will be carried on this texture. Next we load the image as a texture using glTexImage2D(). The process is shown diagrammatically below:

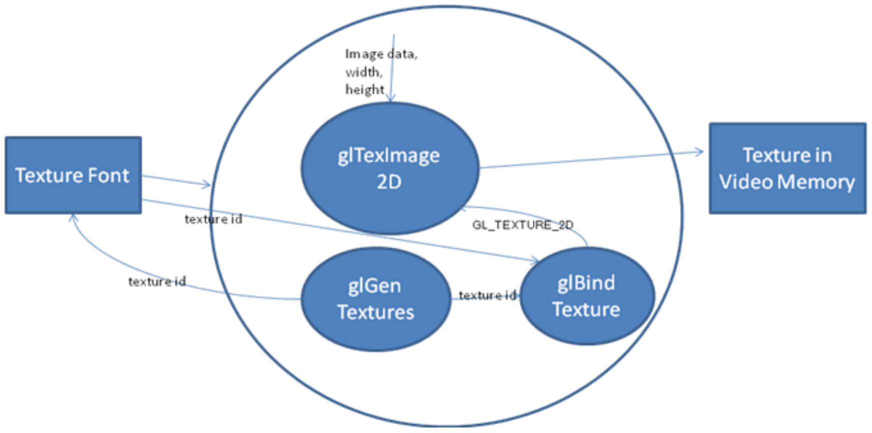


Fig 11: Data Flow Diagram showing loading of image as texture using OpenGL

Once we have the image in the video memory we can render strings by mapping each character with a texture mapped quad. Each character of the string is considered one after another. The glyph metrics is found by using the character code. These provide us with the x, y location and the width, height of the glyph. The glyph is rendered by calculating the corresponding Texture co-ordinates using the glyph information. Once the coordinates have been calculated the glyph can be rendered on to the screen at the required position. After a glyph is rendered the pen position has to be advanced by the advance width of the glyph so as to draw the next glyph. The following diagram provides a data flow model of the render string functionality:

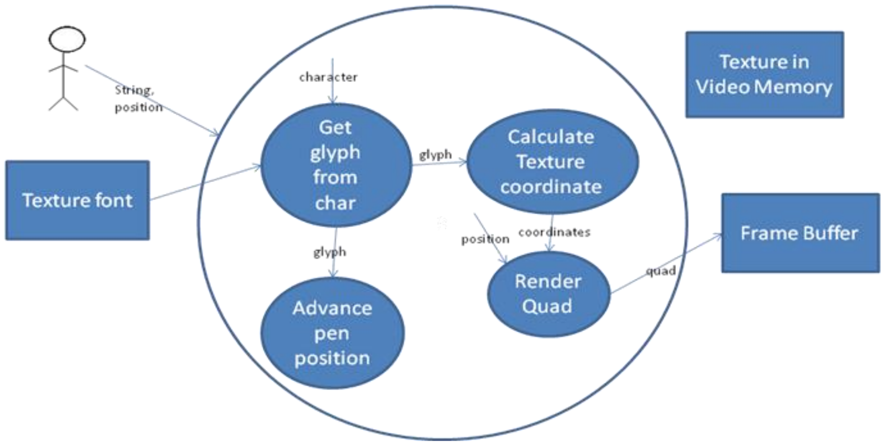


Fig 12: Data Flow Diagram showing rendering of strings by drawing texture mapped quads

5.3 Font API Interface:

We have to provide the string rendering functionality in the form of an API (in the form of library functions). Our aim was to make it easy to use for the programmer. The programmer should be able to specify the location and the point size of the font for rendering. The following are the steps required to load a font and use it to render strings in an OpenGL scene:

1. Declare and initialize the texture font:

   *TextureFont *Arial;*
   Arial = new TextureFont();

2. Load the Texture:

   Arial->LoadFont("Fonts//Arial.ttf",16);

   Also possible to load it directly during initialization:

   Arial = new TextureFont("Fonts//Arial.ttf",16);

3. Now render a string at a particular location using the texture font:

   Arial->RenderString(-0.55f,0.280f,"OpenGL FontAPI");

   Also possible to specify a rectangle in which the text must be placed:

   Arial->RenderString(-0.55f,0.280f,0.5f,0.5f,"OpenGL Font API"); /*where 0.5f represents the width and height of the rectangle*/

5.4 Optimizations:

The first main task of creating a basic font API was completed successfully. Our next objective was to optimize the rendering process. In graphics applications optimization is all about finding and eliminating bottlenecks in the pipeline. In our algorithm we were using immediate mode for sending vertex information of texture quads. So each time vertex information is being transferred from CPU to GPU. This may result in a Transfer Bottleneck, which will result in slower rendering. To reduce the Transfer bottleneck we can store pre-computed vertices in the graphics memory. OpenGL provides a few methods of rendering which makes use of the graphics memory like display lists. However the best method which combines all advantages of the different rendering techniques is by using Vertex Buffer Objects (VBOs). In addition to this we will be using an extension called Frame Buffer Objects (FBO) which slows off screen rendering. Using FBOs we will render the string onto a texture and then to display the string we would just have to display the texture containing the string.

5.4.1 Implementation of VBO in API:

VBOs are intended to enhance the capabilities of OpenGL by providing many of the benefits of immediate mode, display lists and vertex arrays, while avoiding some of the limitations. They allow data to be grouped and stored efficiently like vertex arrays to promote efficient data transfer. They also provide a mechanism for programs to give hints about data usage patterns so that OpenGL implementations can make decisions about the form in which data should be stored and its location. VBOs give applications the flexibility to be able to modify data without causing overhead in transfer due to validation.

The basic idea of this mechanism is to provide some chunks of memory (buffers) that will be available through identifiers. As with any display list or texture, we can bind such a buffer so that it becomes active. Internal memory management can choose the best type of memory (system, video, or AGP), depending on the way we want to use the buffers. VBOs support 3 modes of transfer: GL_STREAM_DRAW (when vertices data could be updated between each rendering), GL_DYNAMIC_DRAW (when vertices data could be updated between each frames), GL_STATIC_DRAW (when vertices data are never or almost never updated).

In each font we are creating VBOs for each character. The vertex information is stored in a structure containing location of screen coordinate and the texture coordinate. One character is a texture mapped quad and hence a collection of 4 vertices. Once the vertex information is stored in the buffer specify the starting position of vertex information and starting position of texture information. (eg: glVertexPointer(3, GL_FLOAT, sizeof(vertex), BUFFER_OFFSET(0));) Now we can use this VBO for drawing. First Bind the buffer using the vbo id. Then specify the starting postion of vertex and texture info for drawing. Use glDrawArrays to draw the character. Now OpenGL refers to the vertex buffer to draw the texture mapped quad (character), hence reducing amount of information transferred from CPU to GPU.

5.4.2 Implementation of FBO in API:

The frame buffer object architecture (FBO) is an extension to OpenGL for doing flexible off-screen rendering, including rendering to a texture. By using frame buffer object (FBO), an OpenGL application can redirect the rendering output to the application-created frame buffer object (FBO). By capturing images that would normally be drawn to the screen, it can be used to implement a large variety of image filters, and post-processing effects. It is used in OpenGL for its efficiency and ease of use.

We are creating a fbo for each string that is being rendered. During initialization the fbo is generated and a texture is attached to it. So now if we bind the fbo and render, it will render directly to the texture. When we render a string for the first time, it will be rendered on to the texture. Then to draw the string this texture will be referred directly. The problem with the FBO approach is that the size of texture is huge (512X512). This is because OpenGL renders the whole screen onto the texture. So for each string we are rendering a texture of size 512X512. This takes up a lot of speed hence reducing the fps.

5.5 Transformations (Scaling, Rotation, Translation):

Another objective of our Font API project was to provide functionality for transformations like scaling, rotation and translation. This was implemented with the help of built in functions provided by OpenGL for 3D Transformations.

3D Transformations are achieved by modifying the given coordinate system. For example when you want to change the camera view, you apply the Viewing Transformation to the "main coordinate system" which is the world coordinate system. After that transformation is performed, any time you draw a new object, it is drawn according to the position of the new coordinate system, which is what makes the objects appear from a different point of view.

There are 4 main types of transformations supported by OpenGL: VIEWING, MODELING, PROJECTION and VIEWPORT transformations.

The Viewing Transformation specifies location of the viewer or the camera. The viewing transformation is analogous to positioning and aiming a camera. The viewing transformation is specified with gluLookAt(). The arguments for this command indicate where the camera (or eye position) is placed, where it is aimed, and which way is up.

The Modeling Transformation is used to modify the position, rotation angle and size of your object (or parts of it) in a scene. Scaling is a type of modeling transformation where the object will either appear smaller or bigger (depending on the scaling factor) in the view. Any movement applied to the objects, for instance if you move your model from one point to another, is achieved through the modeling transformation. (Functions like glRotate(), glScale(), glTranslate() are provided by OpenGL for performing modelling transformations)

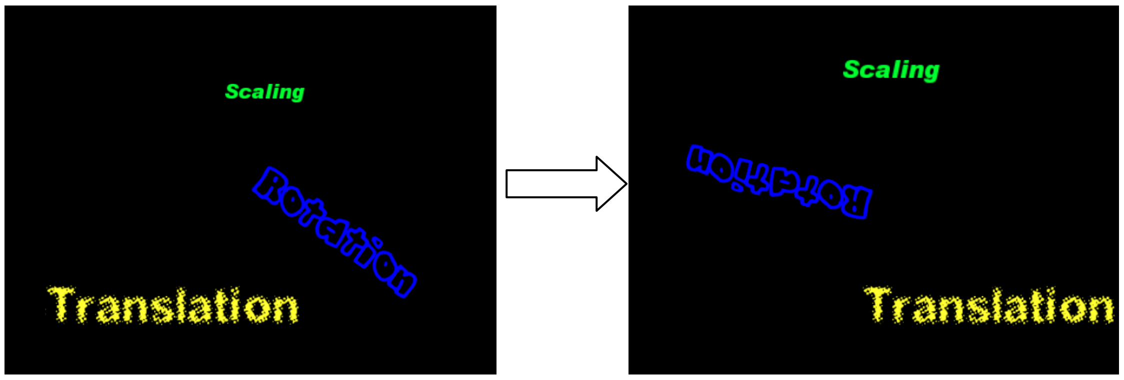


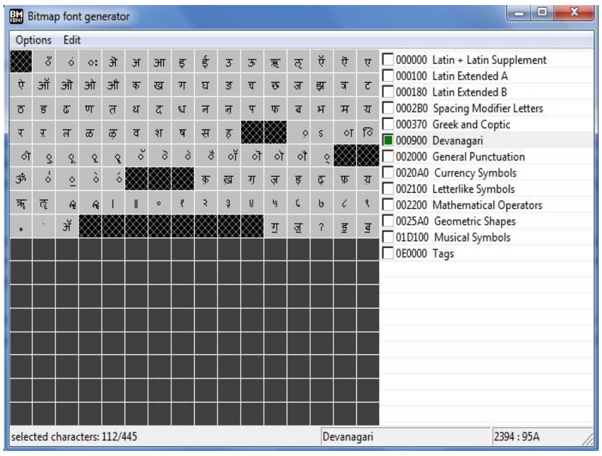
Fig 13: An animation showing different transformations – scaling, rotation and translation

5.6 Customized Texture Fonts:

Another objective of our project is to provide support for customized/user-defined texture fonts. The user will be able to define his own fonts by providing it in a certain format – this would be an image containing all characters of the font along with a text file containing information about the font. The information in the text file include ‘characters in the font’, ‘width of each character’, ‘offset position of each character’ etc...Our job is to incorporate this customized font in the API.

In order to generate custom texture fonts an application called BMFont is usually used. The application generates both image files and character descriptions that can be read by a game for easy rendering of fonts. We have provided support for the custom font format provided by the BMFont application.

The user can select the required character set to create the image. This includes the complete Unicode character set (so there is support for different languages, symbols etc.) The application outputs a png image and an xml file containing description of the font and characters in the image.



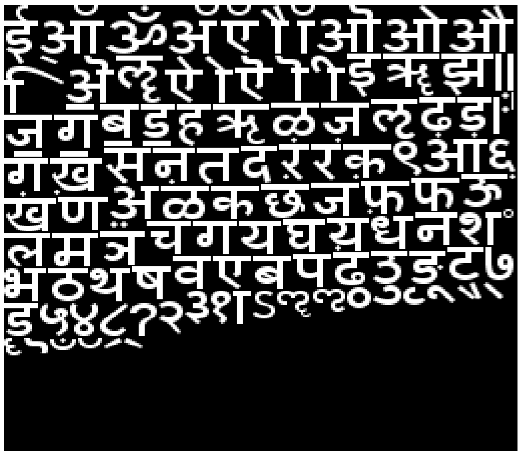


Fig 14: The image on the left shows the BMFont application being used to get all the characters of the hindi font. The image on the right shows the custom texture font that is created.

The programmer will provide the xml file as input for loading the custom font. The xml file contains information regarding the font like location of image file, glyph metrics etc. We need to be parse this xml file to use the information in our API. For this purpose we are using an xml parser in C called Tinyxml. The information in the xml file is used to create the TextureFont object. Also we need to load the png file as a texture in the video memory. For this we use a library called glpng which enables us to load an image file (png) as a texture using a simple interface. Now that we have the TextureFont object ready and the image loaded as a texture we can render strings.

Providing support for custom texture fonts allows us to fullfil a number of other objectives. The BMFont applications allows us to add style to characters ie: make characters appear bold, italic. Moreover since there is support for complete Unicode set the programmers can create a texture map for any language.

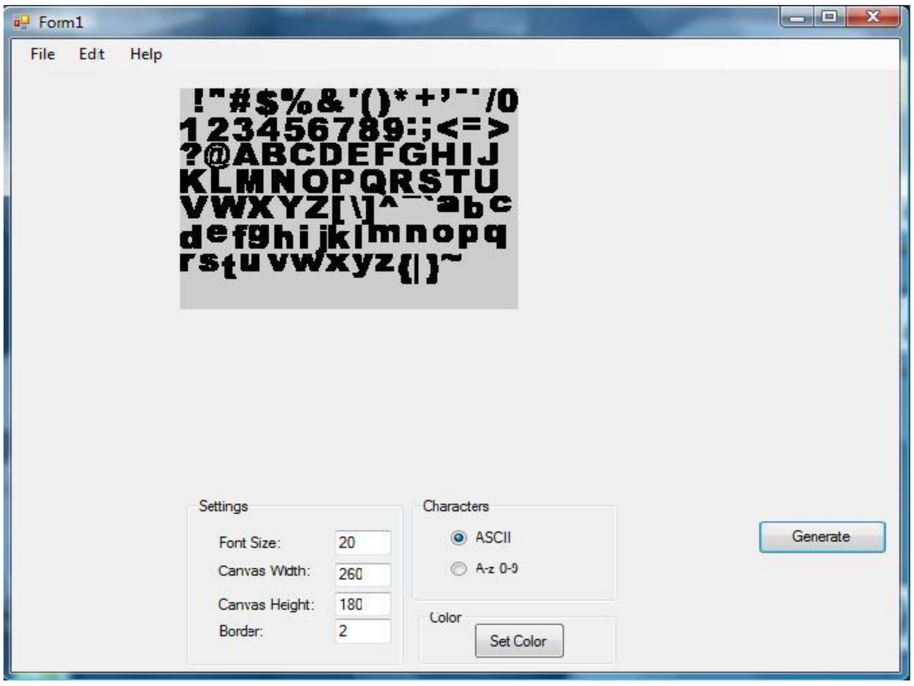
CHAPTER 6: RESULTS & ANALYSIS

6.1 Font Generator Application:

The first part of our system is to render all the glyphs of a font file into a bitmap. We decided to make an application which allows the user to select a font file and generate a font map of all the glyphs into a bitmap image. Font Generator application helps us to test the output of the image being created for different inputs (fonts, point sizes) given by the user. This is needed so that we can decide on the final algorithm to create the image required for texture mapping.

We did the program using Visual Studio.NET. The user gives the font location as the input along with certain parameters like point size, canvas width and height, font colour etc... The program reads the font file and generates all the glyphs and puts all of it into a bitmap file. The user can also decide which glyphs to generate. (A-Z,a-z etc..).

We are using Freetype 2 engine to read the font files and generate bitmaps for glyphs. The output is an image containing all the glyphs of the font file and also a text file which contains the different metrics of each glyph in the font.



Font Size: 20
Canvas Width: 260
Canvas Height: 180
Border: 2

Fig 15: The Font Generator application being used to generate a character map image

6.2 The Font API:

Using the texture containing characters we provided the functionality to render string through our font API. This basically involved completing the two tasks which we had stated earlier: to provide a mechanism to read system font files and render all the glyphs (characters) into an image; to load this image as a texture and use it to render strings given as an input by the programmer.

In the 7th semester we were able to get a string rendered using the texture font. However it had slight problems in alignment. This was because we hadn’t considered the horizontal bearing of the glyphs. Also we had to measure the advance width required between each character, so that the string is rendered correctly. Another problem was that all characters wouldn’t get rendered onto the texture due to the size of the texture. We devised a mechanism to calculate the size of the texture required based on the point size of the font. The final output after testing on a number of fonts is shown below:

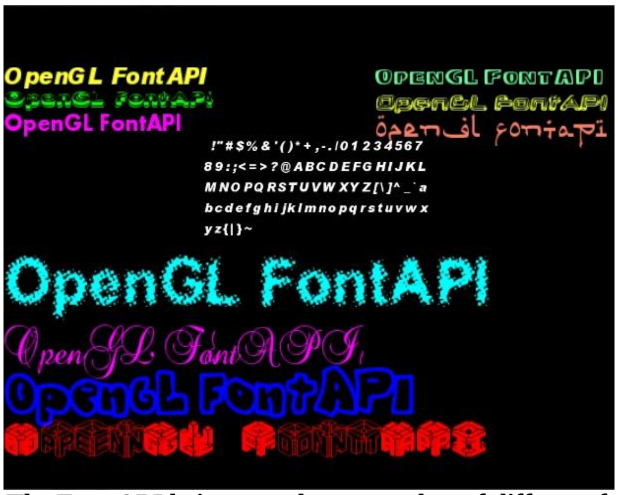


Fig 16: The Font API being tested on a number of different fonts

6.3 VBO Comparisons:

We compared the fps (frames per second) of two implementations – The font API with VBOs and one without VBOs. Initially we used a software called FRAPS to find the frames per second. However this software was not reliable for giving the correct fps rate. So we made our own fps counter. The algorithm used for this is given below.

Using the timeGetTime() function, we get the current system time, in milliseconds. We define a frame rate interval of 2 seconds. As the program loop starts, we initialize two variables frame and fpsTime to 0. Then we find the elapsed time since the program has started. This is done by subtracting the old time from the current time. Once the elapsed time is found, the frame rate interval is found out by adding the elapsed time to it. If this time is greater than the frame rate interval defined earlier, we display the required text and subtract the fixed frame rate interval from the current frame rate time and reset the frame variable. After that we set the old time to new time and continue with the loop. The FPS displayed on the screen is found by frame / fpsTime. We keep incrementing the frame variable after the message loop.

Initially when we compared the two implementations there was very little difference between them. We had to check the application in real life situations to find a difference. The Font API without VBOs uses immediate mode for drawing. So in a graphic intensive application where a number of models in the scene are drawn using immediate mode, it will lead to a bottleneck in the pipeline (transfer b/w CPU and GPU). This is known as Transfer limited bottleneck. On the other hand the implementation with VBO will not lead to a bottleneck as it stores the vertex information in the Graphics memory, therefore no transfer is required.

Due to this reason we had to test the application in a graphic intensive environment to check if there is an optimization. So we added a number of models to the scene. This increases the amount of transfer of vertices between CPU and GPU. The Font API which uses immediate mode for drawing will add vertices to be transferred from CPU to GPU, thus increasing the bottleneck and reducing the fps.

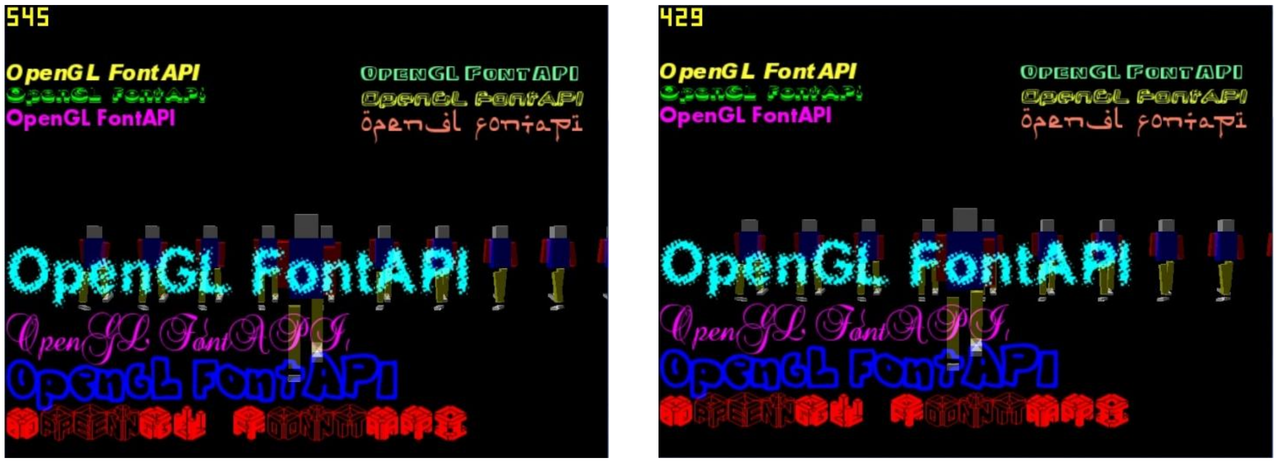


Fig 17: From the above screenshots it can be seen that incorporating VBOs in the Font API leads to an increase in the frame rate by about 100 frames per second

6.4 FBO Optimization:

Unlike VBOs, FBOs did not bring about any optimization as expected. In fact it led to a reduction in the frame rate. This is because in this particular case FBOs will not work to provide optimization. Using FBOs we render a string on to a texture. Then we render this texture onto the screen. However the size of the texture is 512X512 because FBOs should render the entire screen onto the texture. So in the end we are drawing a texture of size 512X512 for each string on the screen. This takes up a lot of speed hence reducing fps. Previously we render texture mapped quads of each character of the string. However these quads are much smaller in size and hence take lesser time to process. So in this particular case FBOs will not work to bring about optimization.

6.5 Custom Texture Fonts:

Providing support for custom texture fonts was a huge advantage for us. The custom texture fonts are generated using an application called BMFont. Using this application we can get a large number of fonts. Moreover we can also add styling (bold, italics or both) and use many different languages as it supports the entire Unicode character set. Below image show text being rendered in different style – bold, italics and both. The image after that shows text being rendered in different languages.

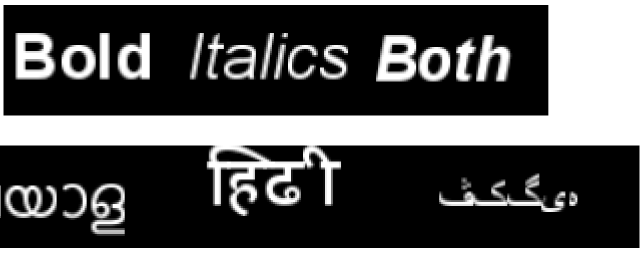
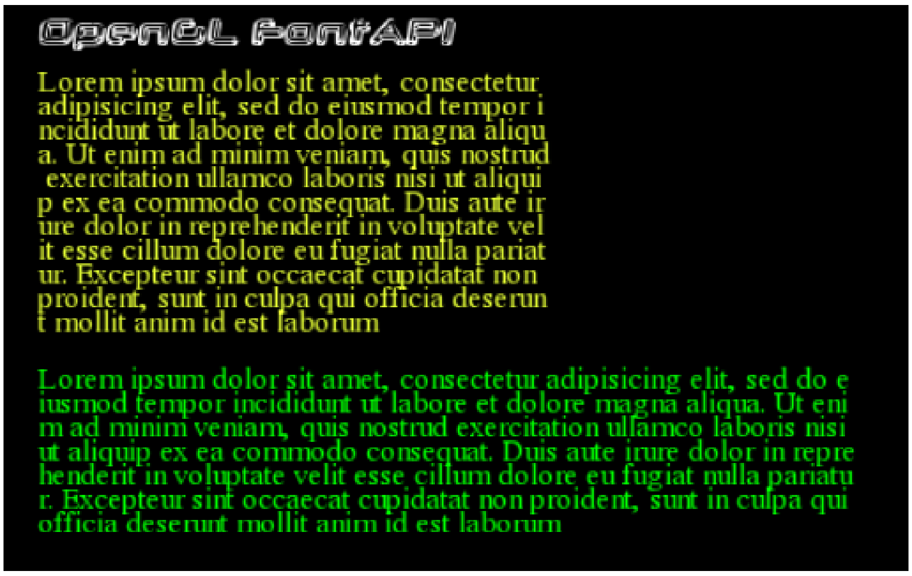


Fig 18: Using custom fonts we can create styled text. Also we can provide support for languages

6.6 Rendering String given a Rectangle:

The font API supports two methods for rendering strings. In the first method the programmer enters the x, y coordinate where the text must appear and the text appears in a line without any line breaks. In the second method along with the x,y coordinates the programmer sends width and height of the rectangle in which the text must be contained in. Basically this can be done by introducing a line break as soon as the width of the rectangle has been crossed during rendering. The diagram below shows rendering of text using rectangles of different width.



```
OpenGL FontAPI

Lorem ipsum dolor sit amet, consectetur
adipisicing elit, sed do eiusmod tempor i
ncididunt ut labore et dolore magna aliqu
a. Ut enim ad minim veniam, quis nostrud
 exercitation ullamco laboris nisi ut aliqui
p ex ea commodo consequat. Duis aute ir
ure dolor in reprehenderit in voluptate vel
it esse cillum dolore eu fugiat nulla pariat
ur. Excepteur sint occaecat cupidatat non
proident, sunt in culpa qui officia deserun
t mollit anim id est laborum

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do e
iusmod tempor incididunt ut labore et dolore magna aliqua. Ut eni
m ad minim veniam, quis nostrud exercitation ullamco laboris nisi
ut aliquip ex ea commodo consequat. Duis aute irure dolor in repre
henderit in voluptate velit esse cillum dolore eu fugiat nulla pariatu
r. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui
officia deserunt mollit anim id est laborum
```

Fig 19: Rendering of text within a rectangle

6.7 Writing Text on Shapes:

In most games text may be written on shapes giving a realistic feel. So the programmer might require the functionality for writing text on a shape. This can be done with the help of FBOs. Using FBOs we can render the string off-screen onto a texture. Once we have a texture containing the string, we can wrap this around any shape by using the texture coordinates. The diagram below shows text being rendered on the 6 sides of a cube.

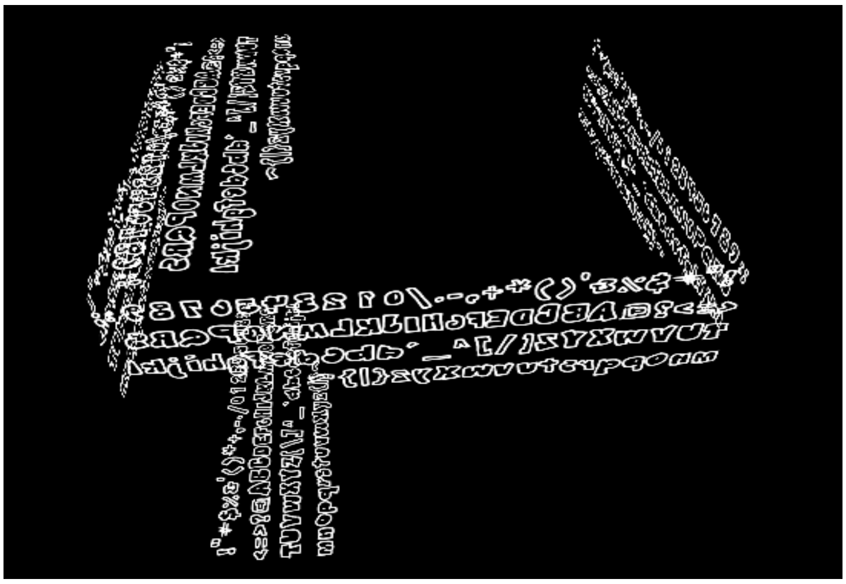


Fig 20: Text being rendered on all the different sides of a cube

CHAPTER 7: CONCLUSION & FUTURE WORK

This report has presented details of design and implementation of our Font API project. First we made a basic font API which would provide the functionality for rendering strings. Later we added more features and optimizations for the rendering. Developing the basic font API involved two main tasks: One task is to provide a mechanism to read system font files (ttf files) and render all the glyphs (characters) into an image. The Second task is to load this image as a texture and use it to render strings given as an input by the programmer. For rendering strings we need to use glyph metrics like advance width and horizontal bearing to get the alignment right. After the basic font API was completed we made use of Vertex Buffer Objects (VBOs) and Frame Buffer Objects (FBOs) to optimize the process. Moreover we also added support for rendering custom texture fonts. This allowed support for styling of characters (bold, italics) and writing in any language (as it supports the entire Unicode character set).

We have successfully completed all objectives of our font API. Now we need to package the library so that the font engine module can be easily plugged in to the game engine that Robosoft Technologies is working on. Our API is cross platform, easy to use and has faster rendering time (use of VBOs). It has been seen that VBOs can lead to a considerable improvement in performance. FBO on the other hand does not work as expected for this particular case. Future work for this project would be to apply some commonly used techniques in OpenGL for optimization. This requires understanding the internal working of OpenGL and modifying our code based on it. Also a number of other features can be added to the API such as adding effects to text, providing support for more font formats, applying shadow and extrusion (3d text) to fonts.

REFERENCES:

Journal References:

[1] Thomas Scott Crow, Dr. Frederick C. Harris (2004). “Evolution of the Graphics Processing”. University of Nevada Reno
[2] Mathias Trapp, (2004) “OpenGL Performance and Bottlenecks”, Hasso Plattner Institute
[3] Dave Shreiner, (2001) “Performance OpenGL – Platform Independent Techniques”, SIGGRAPH 2001
[4] Simon Green, (2007) “The OpenGL Frame Buffer Object Extension”, NVIDIA
[5] Ian Williams, (2006) “Efficient Rendering of Geometric data using OpenGL VBOs”, NVIDIA Corporation
[6] “Using VBOs – White Paper” (2007), NVIDIA Corporation

Books

[1] Jackie Neider, Tom Davis, Mason Woo. (1994) “The OpenGL Programming Guide (RedBook)”. Addison-Wesley Publishing Company, Massachusetts Menlo Park
[2] Tom McRonalds, David Blythe (2001) “Advanced Graphics Programming Using OpenGL”. Morgan Kaufmann Publisher
[3] Richard S. Wright, Michael Sweet (2004) “OpenGL SuperBible”
[4] Chris Seddon (2005), “OpenGL Game Development”. Wordware Publishin Inc

Web Pages:

[1] “OpenGL Introduction”. Wikipedia, http://en.wikipedia.org/wiki/OpenGL
[2] “The Graphics Pipeline”. Wikipedia, http://en.wikipedia.org/wiki/Graphics_pipeline
[3] “Survey Of OpenGL Font Technology”, http://www.opengl.org/resources/features/fontsurvey
[4] “Using Fonts in OpenGL”. http://www.opengl.org/resources/faq/technical/fonts.htm
[5] “Text Rasterization”. http://www.antigrain.com/research/font_rasterization/index.html
[6] “Texture Mapping”. http://www.gamedev.net/reference/articles/article947.asp
[7] “Texture Mapped Text”. http://www.opengl.org/resources/code/samples/mjktips/TexFont/TexFont.html
[8] “The True Type Font File”. http://developer.apple.com/textfonts/TTRefMan/RM06/Chap6.html
[9] “Free Type 2.0 Documentation”. http://www.freetype.org/freetype2/documentation.html
[10] “Glyph Metrics”. http://www.freetype.org/freetype2/docs/glyphs/glyphs-3.html
[11] “OpenGL Performance Optimization”. http://www.mesa3d.org/brianp/sig97/perfopt.htm
[12] “Frame Buffer Objects”. http://www.songho.ca/opengl/gl_fbo.html
[13] “Vertex Buffer Objects”. http://www.songho.ca/opengl/gl_vbo.html
[14] “OpenGL Transformations”. http://www.falloutsoftware.com/tutorials/gl/gl5.htm