|  |
| --- |
| Swansea Metropolitain University |
| Advanced Graphics |
| Software Renderer |
|  |
| **Sion Williams** |
| **2008/2009** |

Table of Contents

[Vector and Matrix Classes 3](#_Toc231917850)

[Look-Up Tables 3](#_Toc231917851)

[The Pipeline 3](#_Toc231917852)

[Line Clipping 3](#_Toc231917853)

[Line Drawing 4](#_Toc231917854)

[Transformation 5](#_Toc231917855)

[Lighting 6](#_Toc231917856)

[Depth Testing and Surface Culling 7](#_Toc231917857)

[Bibliography 8](#_Toc231917858)

[Testing 9](#_Toc231917859)

## 

## Vector and Matrix Classes

These were implemented as fully functional data types with overloaded operators, copy constructors and variable access functions. They support add, subtract, scalar divide and multiply, cross and dot products, transpose etc.

## Look-Up Tables

Look up tables offered access to pre-calculated variables that would otherwise be costly to calculate each and every time they were used. I have implemented look-up tables for both the Sin and Cos of 360 angles. The reason for only using 360 angles is that it’s not often angles of decimal values is used.

## The Pipeline

## 

Figure The four stages of the rendering pipeline in OpenGL.

The software renderer implements the pipeline in the draw triangle function itself. First, the function obtains the model view matrix and multiplies this by the un-translated vertex, the next step is to then multiply this new vertex by the projection matrix. After this perspective division is performed on each new vertex and the triangle is drawn to the frame buffer this frame buffer is then sent to the screen.

## Line Clipping

The code for clipping a line is called from within the line drawing algorithm and its purpose is to determine which area of the canvas the line falls on. If the line is determined to be either completely outside or within the canvas, there is nothing the clip function can do to the geometry, so the function is free to return. If the line is not completely on one side, each of the line points that exist outside of the canvas is moved based on the area it is on. For points that lie on multiple sides (corners) a loop is used to ensure that the function doesn’t return until each point is completely clipped to the canvas. Although the code is long, it is composed of nested conditional statements. This means only a few operations of the whole function are actually executed. Since the operations are not expensive, this clipping algorithm is not slow.

The algorithm can be proven by commenting out the clipLine( ) call in the draw line function and testing a line that is purposely outside the canvas. When the program is run, an assert in the draw pixel function will be provoked and the program will stop running (shown in Figure 2). If clipLine( ) is called this doesn’t happen and a clipped line is drawn.

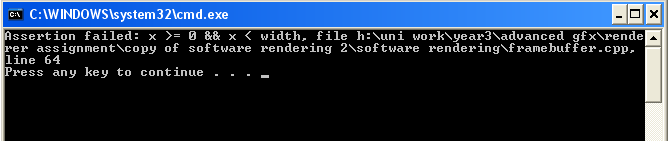


Figure Assert In The Draw Pixel Function

Due to the nature of scan conversion being the process of drawing lines we know all objects will be clipped regardless of geometry.

## Line Drawing

I opted to use the Bresenham line drawing algorithm, the reason for doing this is that Bresenham’s algorithm is an optimized version of the Digital Differential Analyzer algorithm of which is an improvement on the slope intercept algorithm (which is a direct implementation of the slope intercept formula).

This function accepts six parameters - the colour, x position, and y position of the first point of the line followed by the colour, x position, and y position of the second point.

The first thing that’s done inside the function is compute the *dx* and *dy* (the differences of the given x and y coordinates). At this point a check is made to see if the differences are 0 (where, x0 is equal to x1 and y0 is equal to y1), meaning that the line consists of one point, a single pixel is coloured.

Next, the function checks if the absolute value of xdiff is greater than ydiff this allows the line to be drawn in all four quadrants. If true, the algorithm checks which of the two values passed for *x* is greater, since x0 may be lower than x1 and vice versa.

float xmin, xmax;

// set xmin to the lower x value given

// and xmax to the higher value

if(x0 < x1) {

xmin = x0;

xmax = x1;

} else {

xmin = x1;

xmax = x0;

}

Now, the algorithm calculates the slope (or tangent) of the line and loops through each whole number between the minimum/maximum *x* values to draw it:

// draw line in terms of y slope

float slope = ydiff / xdiff;

for(float x = xmin; x <= xmax; x += 1.0f)

{

float y = y0 + ((x - x0) \* slope);

RGB color = color0 + ((color1 - color1) \* ((x - x0) / xdiff));

putPixel(x, y, color);

}

If the result of the original if(fabs(xdiff) > fabs(ydiff)) is false then calculating the y position becomes a simple geometry issue; the current x position relative to the x position of the line's first point is multiplied by the slope and added to the y position of the line's first point.

## Transformation

To handle transformations modelview and projection matrices were implemented. To compute the projection matrix I provided the same parameters as for *gluPerspective*(), which are the field of view fovy, the aspect ratio of the frustum screenWidth/screenHeight, and the near zNear and far zFar planes. The matrix is stored as follows:

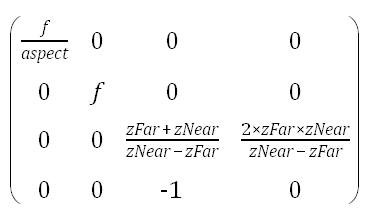
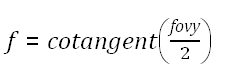


Figure Perspective Matrix

Where:



fovToZoom(float fov) { return 1.0f / tan(fov \* .5f); }

float f = fovToZoom(fovy);

To calculate these matrices the class Matrix4 was used. This class provided common operations such as matrix-matrix and matrix-vector multiplication.

## Lighting

Lighting models fall into two categories:

* Empirical: Simple formulations that approximate observed phenomenon
* Physically-based: Models based on the actual physics of light interacting with matter

A lighting model, also called an illumination model and sometimes referred to as a shading model, is used to calculate the intensity of light that we should see at a given point on the surface of an object. A surface rendering algorithm uses the intensity calculations from a lighting model to determine the light intensity for all projected pixel positions for the various surfaces in the scene.

In games its mostly empirical models that are used. As physically accurate lighting models can be very time consuming to calculate, models that are loosely derived from the physical laws are used.

The Phong lighting model is the simplest, and by far the most popular, lighting and shading model for three dimensional computer graphics. The reason for this is that it’s flexible enough to achieve a wide range of visual effects, and secondly, it’s easy to implement into software and hardware. Although it has no physical basis, it works in practice and is the lighting model of choice for essentially all graphics hardware for personal computers, game consoles, and other real-time applications.

The final empirically motivated model for the illumination at a surface includes ambient, diffuse and specular components:

Color = Ka\*ambientColor + (1/(kC+kL\*di+kQdi2))\*(Kd\*diffuseColori\* (N dot Li) + Ks\*specularColori\*(Ri dot V)shininess)

The above equation is a generalized lighting equation that adds support for multiple sources and also adds attenuation. Where attenuation is a general quadratic equation in the form:

attenuation=1/(kC+kL\*di+kQdi2)

In the equation, di is the distance between the point being shaded and the light source. Now all we have to do is tune the kC, kL, and kQ parameters to reach the results we want. A constant attenuation, for example, would be expressed as (kC!=0, kL=kQ=0). On the opposite end, a quadratic equation that mimics the real world would be achieved by (kC=0, kL=0, kQ!=0). And the very popular linear attenuation used by many games is achieved by kC=0, kL!=0, kQ=0.

## Depth Testing and Surface Culling

To reduce the amount of geometry the CPU has to process; back face culling was applied using setCullMode(CullBack). Backface culling is one of the simplest forms of culling to implement, its process is very simple. Mainly, face culling relies on the properties set forward by the cross product [5] .

Another form of culling introduced was depth testing. The idea is to not render any surface that lies behind any other surface. To handle this a z-buffer was implemented. The z-buffer was simply an array of floats the same size as the framebuffer, when the pixel function was called a check is made to see whether the z value is greater than currently stored value, if not the frame buffer is given the new colour.

# Bibliography

1. **Dave Astle, Kevin Hawkins.** *Beginning OpenGL Game Programming.* s.l. : Premier Press, 2004. 1-59200-369-9.

2. **Buss, Samuel R.** *3-D Computer Graphics: a mathematical introduction with OpenGL.* New York : Cambridge University Press, 2003. 0-511-07850-1.

3. **Luebke, David.** Lighting. *Introduction to Computer Graphics.* [Online] 2003. www.cs.virginia.edu/~gfx/Courses/2003/Intro.spring.03/lecture20.ppt.

4. **Dalmau, Daniel Sánchez-Crespo.** *Core Techniques and Algorithms in Game Programming.* s.l. : New Riders Publishing, September 08, 2003. 0-1310-2009-9.

5. **Tremblay, Cristopher.** *Mathematics for Game Developers.* s.l. : Premier Press. 1-59200-038-X.

6. **Parberry, Fletcher Dunn and Ian.** *3D Math Primer for Graphics and Game Development.* s.l. : Wordware Publishing, Inc., 2002. 1-55622-911-9.

7. **Dalmau, Daniel Sánchez-Crespo.** *Core Techniques and Algorithms in Game Programming.* s.l. : New Riders Publishing, 2003. 0-1310-2009-9.

## Testing

With the program I have included several main.cpp’s to quickly demonstrate the capabilities of the renderer. Each main is named after the operation its demonstrating.