Aleksi Anderson

3D Graphics in CMD



Opinnäytetyö

Tieto ja viestintätekniikka

Kevät 2025

Tiivistelmä

**Tekijä(t)**: Anderson Aleksi

**Työn nimi:** 3D Graphics in CMD

**Tutkintonimike:** Insinööri (AMK), Tieto ja Viestintätekniikka

**Asiasanat:** Rasterization, Rendering, 3D graphics, CMD, Console

Tässä työssä käsiteltiin 3D-maailman renderöinnin perustekniikoita ja selitettiin niitä pseudokoodi esimerkkien avulla. Näihin tekniikoihin sisältyi viiva- ja kolmiorasterointi, 3D-mallit, 3D-muunnokset matriisien avulla, kameran simulointi näkymä- ja perspektiiviprojektioilla, optimointi pintaleikkauksen avulla, syvyyspuskurointi, teksturointi ja perusvalaistus.

Työssä käsiteltiin myös näiden tekniikoiden ja algoritmien toteutusta C++:lla. Toteutus yhdisti käsitelteet tekniikat toimivaksi 3D-renderöintimoottoriksi, joka sovellettiin pelattavaan demoon. Se selittää myös Windows-konsolin käytön perinteisen ikkunan korvikkeena ja kuinka renderöity kuva voidaan näyttää siinä.

Abstract

**Author(s):** Anderson Aleksi

**Title of the Publication:** 3D Graphics in CMD

**Degree Title:** Bachelor of Engineering, Information Technology

**Keywords:** Rasterization, Rendering, 3D graphics, CMD, Console

In this paper, basic techniques involved in rendering 3D scenes were discussed and explained with examples in pseudocode. These techniques included line and triangle rasterization, 3D model formats, 3D transformations using matrices, simulating a camera with view and perspective projections, optimizations through surface culling and clipping, depth buffering, texturing, and basic shading.

The paper also goes over an implementation of these techniques and algorithms in C++. This implementation combines the techniques discussed into a functional 3D rendering engine, which is used in a playable demo. The use of the Windows Console as a substitute for a traditional window and how the output of the rendering engine can be displayed on it is also explained.

Table of Contents

[1 Introduction 1](#_Toc196495325)

[2 Rendering Theory 2](#_Toc196495326)

[2.1 Window 3](#_Toc196495327)

[2.2 Canvas 3](#_Toc196495328)

[2.3 Rasterization 4](#_Toc196495329)

[2.3.1 Rasterizing Lines 4](#_Toc196495330)

[2.3.2 Rasterizing Triangles 9](#_Toc196495331)

[2.4 Rendering 13](#_Toc196495332)

[2.4.1 3D Models 14](#_Toc196495333)

[2.4.2 Transforms 14](#_Toc196495334)

[2.4.3 Camera 17](#_Toc196495335)

[2.4.4 Perspective Projection 18](#_Toc196495336)

[2.4.5 Clipping 21](#_Toc196495337)

[2.4.6 Depth Buffering 26](#_Toc196495338)

[2.4.7 Back Face Culling 29](#_Toc196495339)

[2.5 Texturing 30](#_Toc196495340)

[2.6 Shading 32](#_Toc196495341)

[3 Implementation 35](#_Toc196495342)

[3.1 Tools and Libraries 35](#_Toc196495343)

[3.2 API Structure 35](#_Toc196495344)

[3.3 Components 37](#_Toc196495345)

[3.3.1 Math 38](#_Toc196495346)

[3.3.2 Model and Texture 38](#_Toc196495347)

[3.3.3 Camera 39](#_Toc196495348)

[3.3.4 Renderer 40](#_Toc196495349)

[3.3.5 Rasterizer 42](#_Toc196495350)

[3.3.6 Console Window 45](#_Toc196495351)

[3.4 Demo 49](#_Toc196495352)

[Sources 50](#_Toc196495353)

# Introduction

3D graphics is an incredibly important topic in today’s world, being used in a wide range of applications from video games to movies and even cars. Entire industries have sprung up to create the most efficient and fastest hardware specifically designed for rendering 3D graphics. However, despite how ubiquitous 3D graphics are, most people who use them never delve deeper into the math and algorithms which make them possible.

Therefore, in this paper we will go over the basic techniques used in the rasterization and rendering of a 3D scene to be able to create our own rendering engine from scratch. To go from vector-based models to pixels on a screen. Inspired by Ben Ryves’ demo *ASCII Madness*, we will also be rendering the scene to the Windows Console using only characters from extended ASCII.

# Rendering Theory

First, we will go over the theory, and algorithms for drawing 3D objects onto a 2D screen. We will be going over the rasterization of vector-based objects to pixels and the basic rendering pipeline which takes a 3D environment and draws it on a 2D screen. These algorithms are quite math heavy, so it is recommended you have a basic understanding of vectors and matrices. Here we will go over the algorithms using a Python-style pseudocode. In section 3 the actual implementation will be in C++. Below (see Figure 1) is every step we will go through in converting a 3D model into pixels on a screen [1].

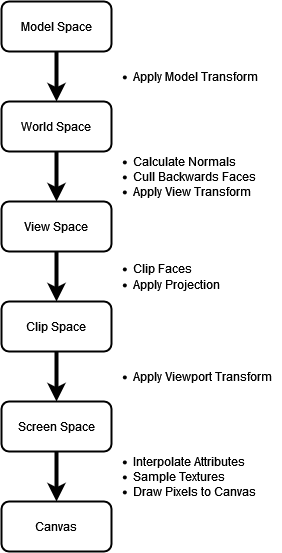


Figure 1. Rendering Pipeline

## Window

A window is really an operating system level concept of a drawable and interactable area of the screen. In Windows it is a programming construct that, as per Microsoft: [2]

* Occupies a certain portion of the screen.
* May or may not be visible at a given moment.
* Knows how to draw itself.
* Responds to events from the user or the operating system.

For us the most important parts are drawing to the window and responding to user input, such as keypresses. There are numerous ways we could make a window to draw on in Windows; APIs such as WinForms in C# or Win32 in C++, but for the implementation of this paper we will be using something a bit more esoteric: the Windows Console, or more specifically in this case CMD. Our only real requirements for a window are the ability to write pixels to the screen, resize it, and read user input, and since CMD satisfies all these, it can work as our window. [2.] All the theory and implementation of the rendering techniques discussed here will still be completely agnostic to the implementation of the window. However the example pictures are rendered onto CMD, which is why they have a low resolution.

## Canvas

Before rendering anything to the window we will first need a canvas, also known as a framebuffer, to draw the whole frame onto. Our canvas will be a 2D array of pixels in which each pixel can be individually colored. A common color format is RGBA, which also stores transparency, but we will just be using RGB since transparency and color blending are outside the scope of this paper. The main functionality of the canvas is the PutPixel function which will change the color of a pixel at a specified x and y coordinate. [1.] For this we need to define a canvas coordinate system. Let’s go with an established standard and use the OpenGL API’s default, where the bottom left is (0, 0) and top right is (width, height) [3].

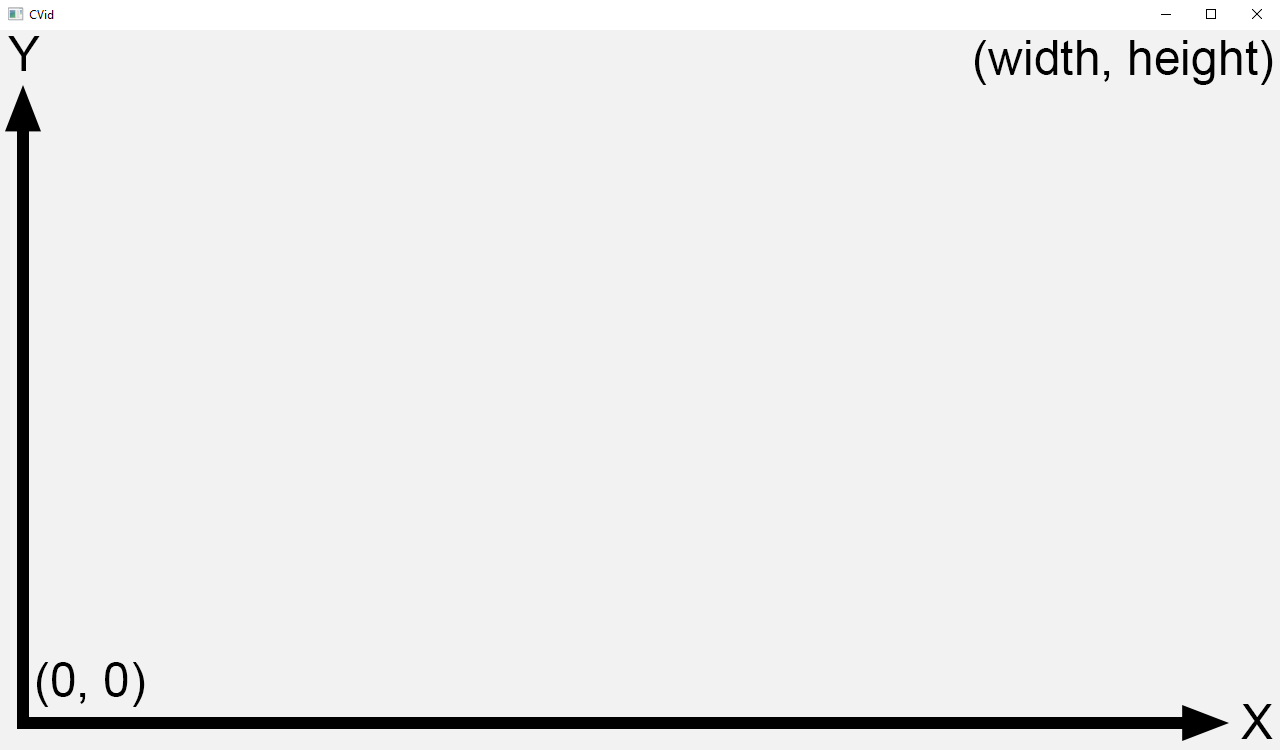


Figure 2. Canvas coordinates.

canvas = Color[width][height]

PutPixel(x, y, color):

//Flip the Y coordinate to start at the bottom

canvas[x][height - 1 – y] = color

This canvas is very simple, but it will work for now. In the following sections, we will be adding some new features to it, such as a depth buffer and depth testing.

## Rasterization

Rasterization is the process of taking a vector-based image, or vertice based 3D object, and converting it to pixels. It is a much faster process of rendering than alternatives such as raytracing but does not directly give information about what color the pixel should be. Therefore rasterization, especially of 3D objects is often combined with pixel shaders to determine the final color of the pixel. [4.]

### Rasterizing Lines

The simplest geometric shape to draw after a point is a line, so we might as well start from there. Lines are often represented in slope-intercept form, which is y = mx + b where x and y are the coordinates of individual points on the line, m is the change in y per x, or the slope, and b is the vertical offset, or the y coordinate where the line intercepts the y-axis. Drawing a line with this formula is as easy as iterating over every x position and plotting the corresponding y value. However we want a function in the form of DrawLine(x0, y0, x1, y1), since drawing line segments with a given start and end point is much more useful. In this case we start drawing from (x, y) = (x0, y0) and add m to x for every integer y position. B will not be useful for us, so we can just ignore it. [1.] The function would look something like this, where x and y are integers:

DrawLine(x0, y0, x1, y1, color):

//Slope is rise/run

m = (y1 - y0) / (x1 - x0)

y = y0

//For each x position, plot the corresponding y

for x from x0 to x1:

PutPixel(x, y, color)

y += m

A black and white graph

Description automatically generatedA dotted line of dots

Description automatically generated

Figure 3 . Shallow line. Figure 4. Steep line.

Drawing some lines with this function produces some interesting results. The lines look jagged because we only have a finite number of pixels to represent a line, and this is the simplest approximation. There are anti-aliasing techniques one can use to smooth out these lines such as FXAA, SSAA, and MSAA, but those are beyond the scope of this paper [1]. The bigger problem is that the line in Figure 3 is missing some pixels.

Since this is a very simple function, there are multiple problems with it. First, the failure to properly draw the line in Figure 3 whose slope is greater than one is because our function can only draw one pixel per x coordinate, thus not being able to draw lines where y increases faster than x. Second, it will not work for vertical lines, as in that case we would divide by 0 when calculating m. Third, if x0 is greater than x1 nothing will be drawn due to the loop immediately terminating. [1.]

We can fix the first two problems by making a copy of the function to draw the line based on the y axis and using that function if the absolute value of the slope is greater than one. The second problem is also easily fixed by swapping the start and end points so that x0 or y0 is always less than x1 or y1. [1.]

DrawLine(x0, y0, x1, y1, color):

//If slope is less than 1

if abs(x1 - x0) > abs(y1 - y0):

//Make sure x1 is smaller than x2

if x0 > x1:

swap(x0, x1)

swap(y0, y1)

//Slope is rise/run

m = (y1 - y0) / (x1 - x0)

y = y0

//For each x position, plot the corresponding y

for x from x0 to x1:

PutPixel(x, y, color)

y += m

else:

//Make sure y1 is smaller than y2

if y0 > y1:

swap(x0, x1)

swap(y0, y1)

//Slope is run/rise

m = (y1 - y0) / (x1 - x0)

x = x0

//For each y position, plot the corresponding x

for y from y0 to y1:

PutPixel(x, y, color)

x += m

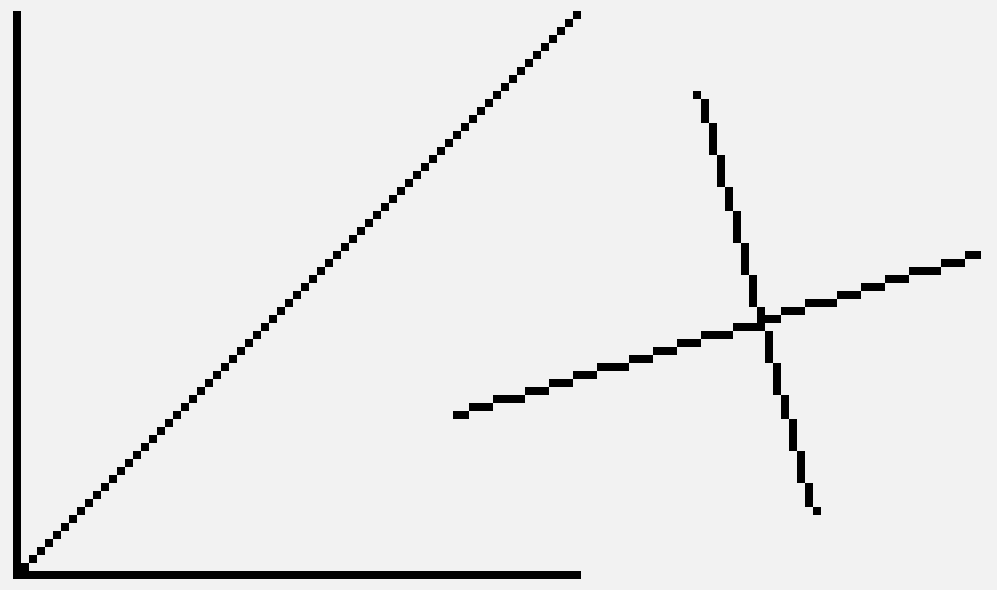


Figure 5. Multiple lines.

This completed function allows us to draw any line between any two points. However, this function is far from optimal since we use a floating-point number for m, meaning there is some computationally expensive division and rounding. It would be nice to get rid of those to get our function running fast on a CPU. For this we can implement Bresenham's line algorithm. It is the best line drawing algorithm for our purpose since it works on any line and is also optimized to only use integer arithmetic. Its main downside is the lack of anti-aliasing, but we won't be using that anyways. [5.] Alternatives with anti-aliasing support include Gupta-Sproull and Xiaolin Wu’s algorithms.

Bresenham's line algorithm works by tracking the accumulated error in the line's actual y and the plotted y at every x position. After each pixel is plotted, the error is increased by the slope. Next, the algorithm decides if the plotted y should be incremented by 1 based on the amount of error: if the error is more than 1/2, y should be incremented and the error should be decremented, thus we always plot the closest possible pixel to the actual y. [5.]

DrawLine(x0, y0, x1, y1, color):

m = (y1 - y0) / (x1 - x0)

y = y0

error = 0.0

//For each x position, plot the corresponding y

for x from x0 to x1:

PutPixel(x, y, color)

error += m

if error > 0.5:

y += 1

error -= 1.0

This implementation still has floating point arithmetic, so to write it in a form which only uses integers we will change around our two problematic lines: error += m, and if error > 0.5.

To rewrite the function to work with only integers we first expand m:

dy = y0 – y1, dx = x0 – x1

error = error + dy / dx

Then, to get rid of the fraction:

dx \* error = dx \* error + dy

Next, to get rid of the fraction in if error > 0.5:

if error \* 2 > 1

To make these two lines work in code:

2 \* dx \* error = 2 \* dx \* error + 2 \* dy

if error \* 2 \* dx > 1 \* dx

Simplify by grouping together 2 \* dx \* error:

error = error + 2 \* dy

if error > dx

Rewriting the function this way avoids floating point division and allows every number to be an integer, which makes the function faster [5]. We will still need to apply the fixes for different slopes from our original algorithm, as well as account for a negative slope by decrementing x or y instead of incrementing. This new function looks something like this:

DrawLine(x0, y0, x1, y1, color):

//If slope is less than 1

if abs(x1 - x0) > abs(y1 - y0):

//Make sure starting point is before ending point

if x0 > x1:

swap(x0, x1)

swap(y0, y1)

//Calculate slopes for x and y

dx = x1 – x0;

dy = y1 – y0;

//If slope is positive increment y, else decrement

yi = 1

if dy < 0:

yi = -1

dy = -dy

y = y0

error = 0

//For each x position, plot the corresponding y

for x from x0 to x1:

PutPixel(x, y, color)

error += 2 \* dy

if error > dx:

y += yi

error -= 2 \* dx

else:

//Same as above, but x and y are swapped

...

### Rasterizing Triangles

The next geometric shape we want to draw is a triangle, since most 3D models are made exclusively of triangles and all polygons can be decomposed into triangles. A triangle is formed by three vertices we will refer to as v0, v1, and v2. Since we're working on a 2D canvas, these vertices will consist of only an x and y coordinate. We can use our DrawLine function to draw a triangle just by drawing lines connecting the vertices: [1.]

DrawWireframeTriangle(v0, v1, v2, color):

DrawLine(v0, v1, color)

DrawLine(v1, v2, color)

DrawLine(v2, v0, color)

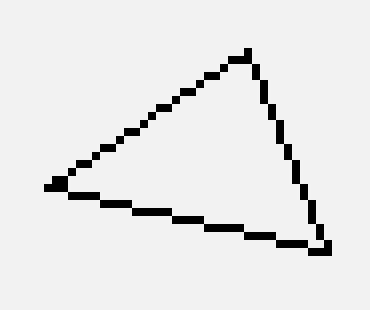


Figure 6. Wireframe triangle.

This function, however, only draws a wireframe triangle, meaning only its edges are colored in. We also need a function to draw a filled in triangle. A simple method for doing this is drawing the triangle entirely out of horizontal lines. To do this, we can simply iterate over every y position in between the triangle's top and bottom vertices and draw a line from the left side to the right: [1.]

for y from topY to bottomY:

leftBound, rightBound = CalculateBounds()

DrawLine(leftBound, y, rightBound, y)

To get the topY and bottomY, we can simply sort the vertices before drawing. The actual tricky part of this implementation is calculating the right and left bounds. To solve this, we can consider that the x bounds are defined by the segments (v0, v1), (v1, v2), and (v0, v2), where one of these lines will be an entire side and the other side will be made up of the remaining two. Since we sorted the vertices, we know (v0, v2) will always be the continuous side, while (v0, v1) and (v1, v2) will make up the segmented side. [1.]

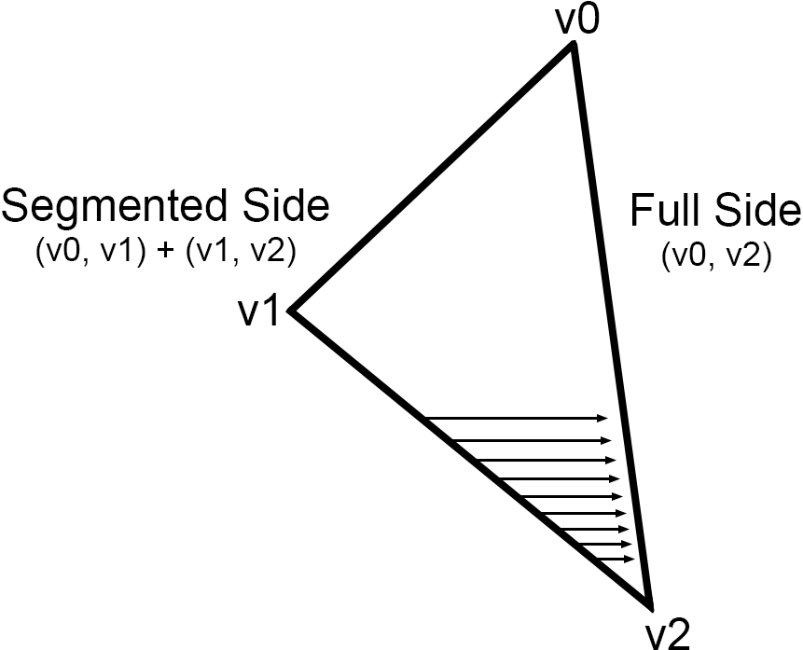


Figure 7. Drawing lines from segmented to full side.

To calculate the x bounds we can use a slightly modified version of our line drawing algorithm, where instead of drawing the point, we store the x value in a list whenever y changes:

This function will return a list of x coordinates for every y position. To get the bounds all we need to do is combine the lists for the segmented side and figure out which list is the left and which is the right one. To figure this out we can calculate a directional vector from v0 to v1 and v2 and check which has a bigger x value, since we know v0 will always be above v1 and v2, this holds true for every triangle [6]. There will also be a duplicate x position in the segmented list right where the two lines meet, so we must make sure to remove that. Putting all this together we have a simple function to draw filled triangles: [1.]

//Adapted from the DrawLine function

CalculateXBounds(x0, y0, x1, y1):

**//Store the x positions in a list**

**xBounds = []**

if abs(x1 - x0) > abs(y1 - y0):

. . .

for x from x0 to x1:

error += 2 \* dy

if error > dx:

**//Add the x position to the list instead of drawing it**

**xBounds.append(x)**

y += yi

error -= 2 \* dx

else:

. . .

for y from y0 to y1:

**//Add the x position to the list instead of drawing it**

**xBounds.append(x)**

error += 2 \* dy

if error > dx:

x += xi

error -= 2 \* dx

**return xBounds**

This is quite a simple function, but it will do for now. We will be coming back to it in the following sections to add more functionality, such as vertex attributes. Notice we also did not use our DrawLine function here. That is due to our lines being exclusively horizontal, so we can make a more optimized implementation for this specific purpose [1].

DrawTriangle(v0, v1, v2, color):

//Sort the vertices in descending y

if v0.y < v1.y: swap(v0, v1)

if v0.y < v2.y: swap(v0, v2)

if v1.y < v2.y: swap(v1, v2)

//Calculate the x bounds of every edge

v01Bounds = CalculateXBounds(y0, x0, y1, x1)

v12Bounds = CalculateXBounds(y1, x1, y2, x2)

v02Bounds = CalculateXBounds(y0, x0, y2, x2)

//Combine the two lists of the segmented side

v01Bounds.removeLast()

v012Bounds = v01Bounds.append(v12Bounds)

//Check which side is left and right

leftBounds = v02Bounds

rightBounds = v012Bounds

v01 = Normalize(v1 – v0)

v02 = Normalize(v2 – v0)

if v01.x < v02.x:

swap(leftBounds, rightBounds)

//Draw each horizontal line

for y from y0 to y2:

for x from leftBounds[y - y0] to rightBounds[y - y0]:

PutPixel(x, y, color)

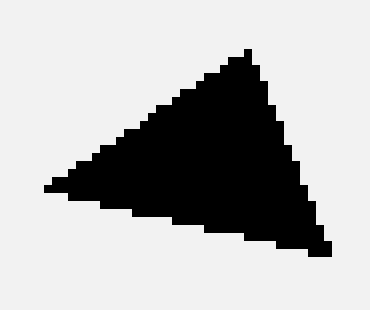


Figure 8. Filled Triangle.

## Rendering

Being able to draw triangles is very useful, because every other polygon can be decomposed into multiple triangles [1]. For example, to draw a rectangle defined by the vertices (v0, v1, v2, v3) we can draw two triangles (v0, v1, v3) and (v1, v2, v3).

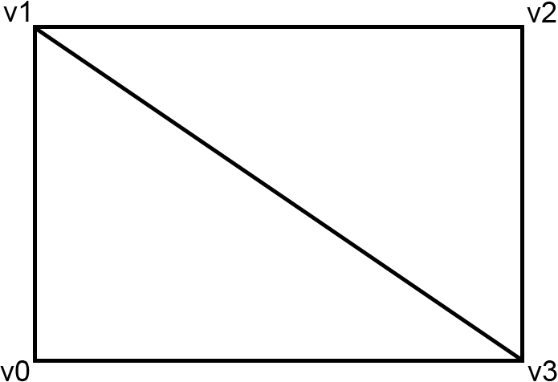


Figure 9. Rectangle decomposed into two triangles.

This works for drawing any 2D polygon, but there are still numerous problems. We need to be able to draw 3D objects, move them, and move the camera. We also need to take perspective into account to render images the way they appear in real life. To solve these, we need to expand our render pipeline to further process the objects before passing them to the rasterizer. Below is every operation we will need to do to the vertices in order.

1. Apply model transform
2. Calculate normal vectors
3. Cull backwards faces
4. Apply view transform
5. Clip faces
6. Apply projection
7. Apply viewport transform

### 3D Models

First, we need to figure out a way to represent 3D objects in a general form. For example, take a cube with the vertices v0 – v7. These vertices are only points in space, so without more information we would have no idea which ones to use to draw the triangles of the cube. Therefore, we also need a list of triangles describing which three vertices make up each triangle. Since a cube has 6 rectangular faces, we need a list of 12 triangles to draw it. These two lists we call *vertices* and *indices* are enough information to render any 3D object made of triangles. [1.]

This is a very common format for storing 3D objects. File formats, such as .obj, work very similarly with the main difference being that faces can be any polygon, not just triangles. These formats are also capable of storing other information, such as materials and textures, which we will cover in a later section. [7.] Manually loading these models from files is outside the scope of this paper, but there are many existing libraries, such as *Assimp* and *tinyobjloader*, which are easy to implement and do some useful operations, such as triangulating the faces at load time. Section 3.3.2 goes over model loading with tinyobjloader in slightly more detail.

To render objects stored this way, all we need to do is loop over every entry in indices and call DrawTriangle with the corresponding vertices. It might also be useful to store these in a class, since we will want to add more information to our objects in the future:

vertices = [v0, v1, v2, v3, …]

indices = [[0, 1, 3], [1, 2, 3], …]

RenderIndexed(vertices, indices, color):

for tri in indices:

DrawTriangle(vertices[tri.v0],

vertices[tri.v0],

vertices[tri.v0],

color)

### Transforms

Another very important feature for our renderer is being able to move objects around. Right now, we are rendering them all in model space, which means they are relative to the object’s local origin. We want to be able to move these objects around the world without having to redefine the model itself. To do this we apply a transformation to each vertex before passing them to the rasterizer. We also need to set a convention for the x, y, and z axes of our world space. A good convention is the one used by graphics API’s such as OpenGL, where in the default view +X is right, +Y is up, and -Z is forward. [8.]

There are three main transforms we want to do: translation, rotation, and scaling. The proper method of applying these is with a transformation matrix, therefore it is important to have at least a basic understanding of matrix multiplication before trying to understand transformations. We will be using column major orderings for matrices, which is the convention used by OpenGL [9]. Three-dimensional transformation matrices are represented in homogenous coordinates (4x4 matrices), but all our vertices are in cartesian coordinates (3x1 matrices, or vector 3s). We will need to convert vertices between these two systems, but first it is good to understand the basic idea of homogenous coordinates and how they are useful for us. [1.]

For us the useful difference is the distinction between points and vectors. Take A = (1, 2, 3), for example. There is no way to know if A is a point or a vector, but if we add a fourth value w, we can now represent a vector when w = 0 and a point when w = 1. The cases where w is some other number also represent point, the important part is the ratio between xyz and w. So, to convert from cartesian to homogenous coordinates, we can simply add the proper value of w, so A = (1, 2, 3, 0) is a vector, and A = (1, 2, 3, 1) is a point, B = (2, 4, 6, 2) is also the same point since the ratio between xyz and w remains the same. To convert back from homogenous coordinates to cartesian coordinates we divide xyz by w, which demonstrates that A = B. [1.]

Now to finally transform our objects we need to construct the translation, rotation, and scale matrices, multiply them together for the transformation matrix, and finally multiply each vertex by this matrix.

Translation by vector *T*:

Scale by vector *S*:

There are three rotation matrices we will need to use: one for rotating around each axis. The order of applying these is also important as rotations are not commutative [10]:

Rotation θ around the X axis:

Rotation θ around the Y axis:

Rotation θ around the Z axis:

The final rotation matrix can be calculated by multiplying these three together. We will use the order z, y, x as a convention:

Note that this method of rotation is not perfect and introduces a problem called gimbal lock. Fixing this would require representing rotations using quaternions, but that is outside the scope of this paper. [8.]

Now that we have these matrices, we can multiply them together to get our transformation matrix, which we will call the model matrix to avoid confusion with future transformation matrices. The order of multiplication also matters; the first transformation will be the last multiplication. Since we want to apply the transforms in the order scale, rotation, translation, the final model matrix is calculated as such: [8.]

scale = Scale(sx, sy, sz)

rotation = RotateZ(rz) \* RotateY(ry) \* RotateX(rx)

translation = Translate(tx, ty, tz)

model = translation \* rotation \* scale

After applying this transformation, our vertices will be in world space.

### Camera

We can now move around objects; next we need a camera that we can also move around the scene. The actual implementation of the camera might, however, be counterintuitive at first. Instead of moving the camera, we keep the camera still, pointing at -Z and move the entire world around it. Since transforming the camera by T produces identical results to transforming everything in the world by -T, there is no difference in the result. [1.]

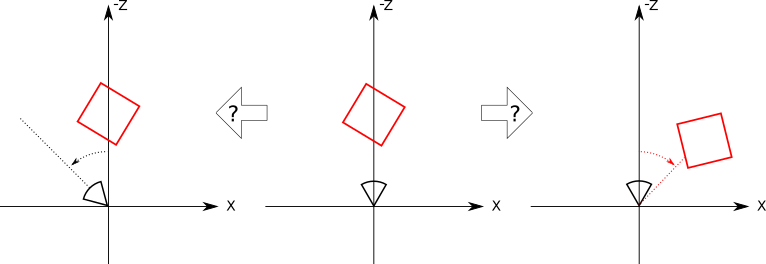


Figure 10. The camera can’t tell whether the object or the camera rotated. [1]

This is much better than the alternative since it greatly simplifies the projection of 3D vertices to 2D points. It is also easier to implement. We can use the same matrices as in the last section, just apply them in the reverse order and invert all the transforms. Here we will also only need rotation and translation since a camera's scale works slightly differently and will be implemented in the next section. Therefore, our camera transformation matrix, which we call the view matrix, is: [1.]

//An easier way to apply the inverse transforms in code is

//inverting them before constructing the matrices

translation = Translate(tx, ty, tz)

rotation = RotateZ(-rz) \* RotateY(-ry) \* RotateX(-rx)

view = rotation \* translation

After applying this transformation, our vertices will be in view space.

### Perspective Projection

The last thing we need to do to get our objects rendering as they would in real life is apply a perspective projection to them. The idea with this is to convert 3D points into 2D points on the viewport and render them as a real camera would see them, with farther away objects appearing smaller. We will also want to normalize these points into the range (-1, 1), which is called normalized device coordinates, or NDC. Our vertices will then be in clip space. [8.]

Perspective projection of a point can be calculated using basic trigonometry. Consider Figure 11 where P is the point we want to project, P' is the projected point, n is the projection plane, also known as the near clip plane, and the camera C is at the origin facing toward -Z. [9.]

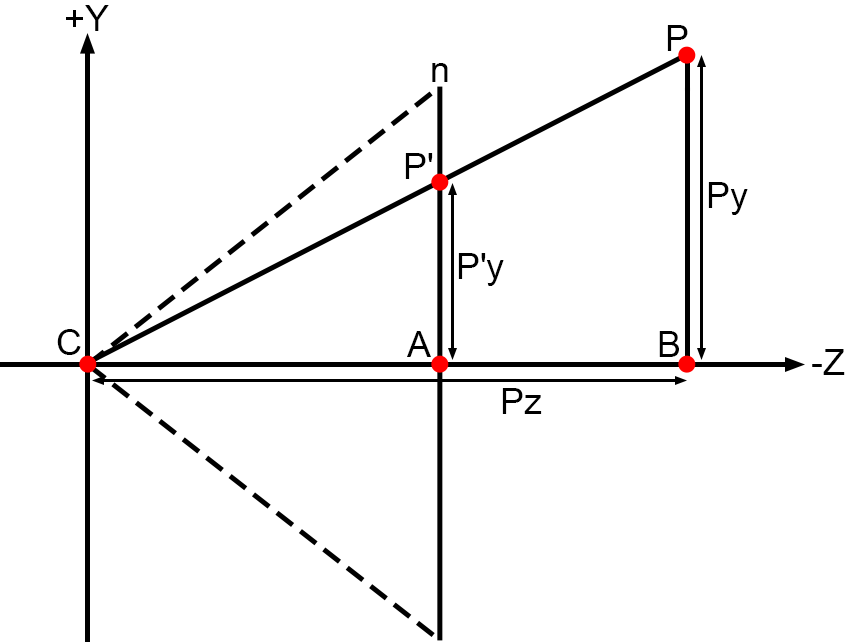


Figure 11. Perspective projection.

The triangles formed by CBP and CAP' are similar, whose properties we can use to calculate P'. By the properties of similar triangles CA/CB = AP'/BP, substituting our known values and solving for AP' we get AP' = P'y = n \* Py / -Pz. Note that since the camera is facing towards -Z, Pz is inverted to preserve the sign of the y coordinate. The same logic works for P'x = n \* Px / -Pz. [9.]

This is the basic idea of perspective projection, but we still have to map the point to NDC and preferably accomplish all this with a single matrix. Therefore, we will directly use the OpenGL projection matrix, which is calculated with the help of the left (l), right (r), top (t), and bottom (b) edges of the camera, as well as the near (n) and far (f) clip planes. This matrix will map the x and y coordinates into NDC range and the z coordinate to between (-1, 1). However, we want the z coordinate to be in range (0, 1) because it is a more standard range and easier for us to work with, so we will slightly modify the matrix to achieve this result. The modified OpenGL projection matrix is below: [9.]

Now that we have the projection matrix, we still need to calculate the values it needs. The near and far planes are easy, as they are given by the user. The other values are slightly more difficult, since we will want to calculate them based on the camera's field of view, or fov, and aspect ratio. The fov can be defined as either the vertical or horizontal view angle. Here we will define it as the vertical angle, since that is the convention used by OpenGL, and it makes more sense with the standard way of defining aspect ratio as width/height. Calculating these values is trivial with basic trigonometry. [9.]

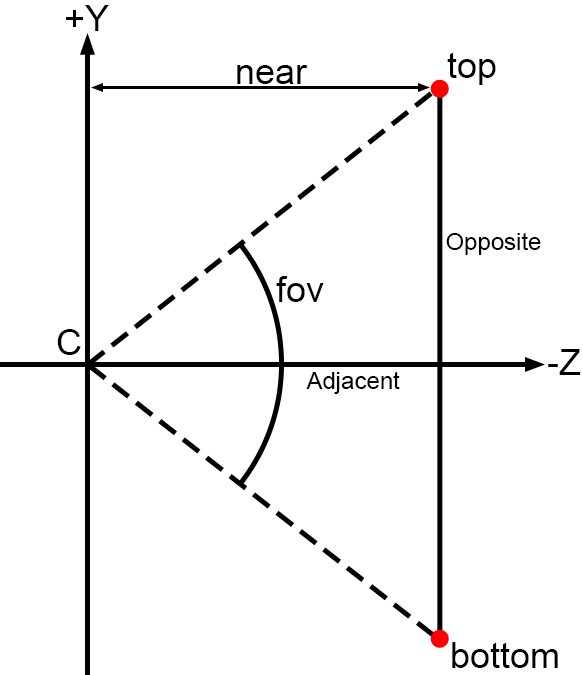


Figure 12. Edge Calculations.

From Figure 12 you can derive the following:

top = tan(fov / 2) \* near

bottom = -top

And for the width and height, we simply factor in the aspect ratio, which is also given by the user.

right = top \* aspectRatio

left = -right

Now we can construct the final projection matrix and apply it to our vertices. For the final steps before sending our triangle to the rasterizer we need to transform the vertices into canvas coordinates. First, we convert the homogenous coordinates back into cartesian coordinates by diving xyz by w. [9.] Then, because we defined our canvas as having its origin at the bottom left, with +Y going up and +X going right, we simply need to multiply and add half the width to x and half the height to y in order to convert the point into canvas coordinates.

The basic program for transforming our vertices from model space to canvas coordinates is below:

//Transform every vertex in the model

for vert in vertices:

//Convert to homogenous coordinates

hVert = Vector4(vert, 1)

//Apply model transform

hVert = model \* hVert

//Apply view (camera) transform

hVert = view \* hVert

//Apply perspective transform

hVert = projection \* hVert

//Convert to cartesian coordinates

vert = hVert.xyz / hVert.w

//Convert to canvas coordinates

vert \*= canvasSize / 2

vert += canvasSize / 2

//Render each triangle by indices

RenderIndexed(vertices, indices, color)

### Clipping

We are now rendering a proper scene with perspective and a movable camera. However, we introduced a big problem: if the vertex is behind the camera, w will be negative, which completely breaks our rendering. Even worse, if the vertex is right on the near clip plane it will cause a division by zero and crash the program. To fix this, we can choose to not render anything behind the near clip plane. In fact, we can also define five more planes to fully describe the viewable area of the camera, called the clipping volume, and not render anything outside it. [1.]

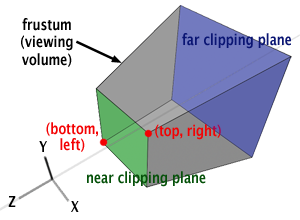


Figure 13. Clipping volume. [9]

We can start by looking at entire objects. There are a few methods for checking if an object is inside the clipping volume, such as an axis-aligned bounding box, but we are going to use a simple bounding sphere because the math and implementation are simpler. Let’s first go over how to define the clipping volume, and then clip a sphere against it. The equation of a 3D is plane is , where *N* is the normal vector of the plane, *P* is a point on the plane, and *D* is the signed distance from the plane to the origin. This is very useful for us since replacing *P* with any point will give us the distance from the point to the plane. [1.] Now to define *N* and *D* for each of our planes, all we need is the aspect ratio and some basic trigonometry:

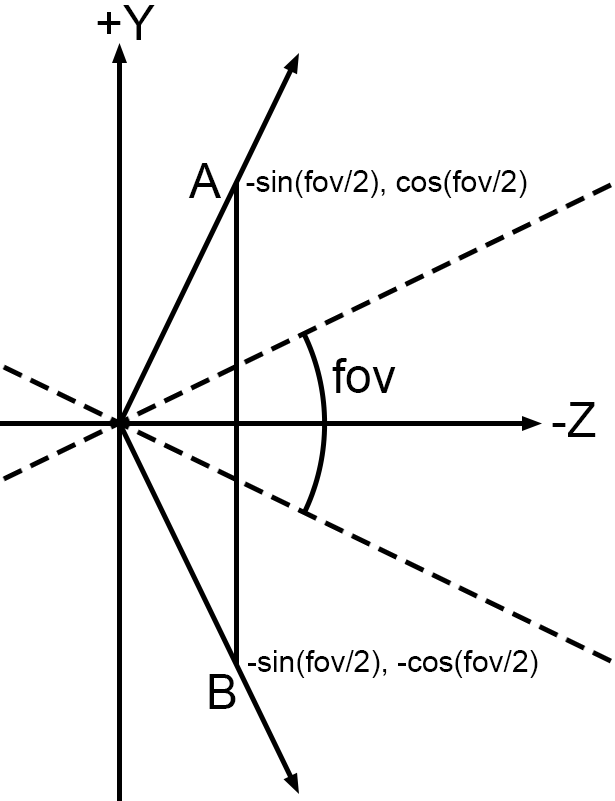


Figure 14. Clip plane normal vectors.

Here vector A is actually the normal of the bottom plane and vector B is the normal of the top plane, as they both point inside the clipping volume. D is zero for both planes since they pass through the origin. Following the same logic with the horizontal aspect ratio, which we can get with the equation , we can calculate the left and right clip planes. The far and near planes are easy, as they just point in -Z and +Z respectively, with *D* being their distance from the camera given by the user. After calculating these planes testing if a sphere is inside them is very easy. Say we have a sphere defined by center *s* and radius *r*. All we have to do is plug *s* in to the plane equation to get the distance *d* and compare it to *r*. If *d* > *r* the sphere is in front, if *d* < *-r* the sphere is behind, and if the absolute value of d < r, the sphere is intersecting the plane. Calculating a bounding sphere for a 3D object is surprisingly complicated, but we can approximate it by calculating the average position of the vertices and its distance to the farthest vertex. [1.]

A diagram of circles and arrows

Description automatically generated

Figure 15. Classifying spheres as above or below a plane.

GetClipPlanes(fov, near, far):

//Calculate the horizontal and vertical angles

vAngle = fov / 2

hAngle = vAngle \* aspectRatio

//Calculate all the clip planes

return clipPlanes = [

(Vector3(0, 0, -1), -near), //Near plane

(Vector3(0, 0, 1), -far), //Far plane

(Vector3(cos(hAngle), 0, -sin(hAngle)), 0) //Left plane

(Vector3(-cos(hAngle), 0, -sin(hAngle)), 0) //Right plane

(Vector3(0, -cos(vAngle), -sin(vAngle)), 0) //Top plane

(Vector3(0, cos(vAngle), -sin(vAngle)), 0) //Bottom plane

]

//Clips the triangles of the model against every plane

ClipModel(model):

//Calculate the clip planes with fixed values

clipPlanes = GetClipPlanes(90, 0.1, 100)

//Test the sphere against each plane

for plane in clipPlanes:

//Calculate the distance from center point to plane

d = plane.n.Dot(model.bounds.center) + plane.d

//Sphere is in front

if d > model.bounds.radius:

continue

//Sphere is behind

else if d < - model.bounds.radius:

return none

//Sphere is intersecting

else:

//Clip each triangle against the plane

for triangle in model.triangles:

//Clip the individual triangle agains a plane

newTri = ClipTriangle(triangle, plane)

clippedTris.append(newTri)

//Set the clipped triangles as the model’s triangles

model.triangles = clippedTris

In the above code, if the bounding sphere is intersecting a plane, we clip each of that object’s triangles against the intersecting plane. To do this, the first step is to see which of the vertices are in front and which are behind the plane. We can accomplish this with the same method we used with the center points of the bounding spheres. This will then leave us with four possible outcomes: three vertices in front, three behind, one in front, and two in front.

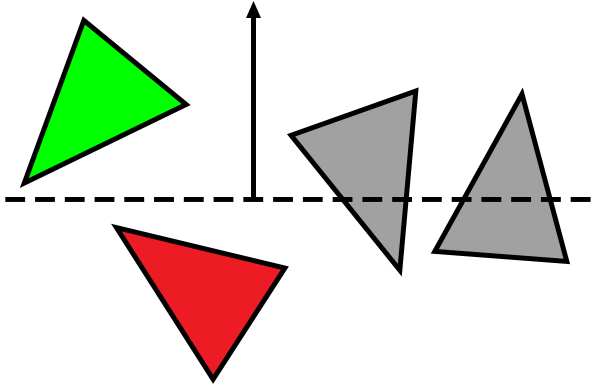


Figure 16. Classifying triangles as above or below a plane.

The first two cases are easy; we either draw the whole triangle, or none of it. The other two are more difficult since we have to create one or two new triangles at the intersection with the plane. For the case of two vertices in front, we have to create two triangles *ABD* and *BED*. For one in front, we create one triangle *AED*.

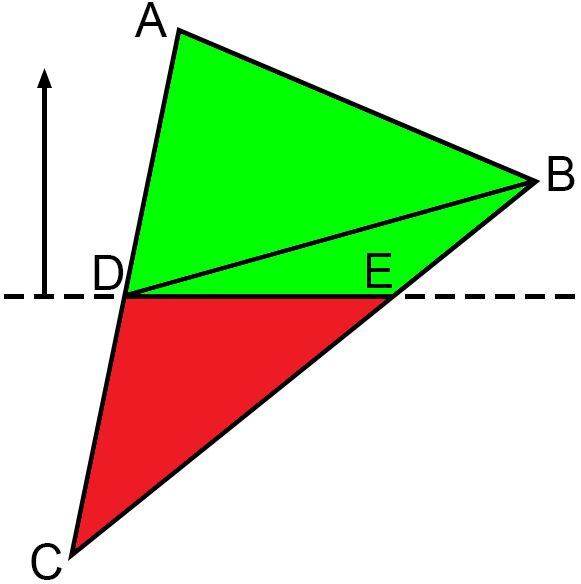
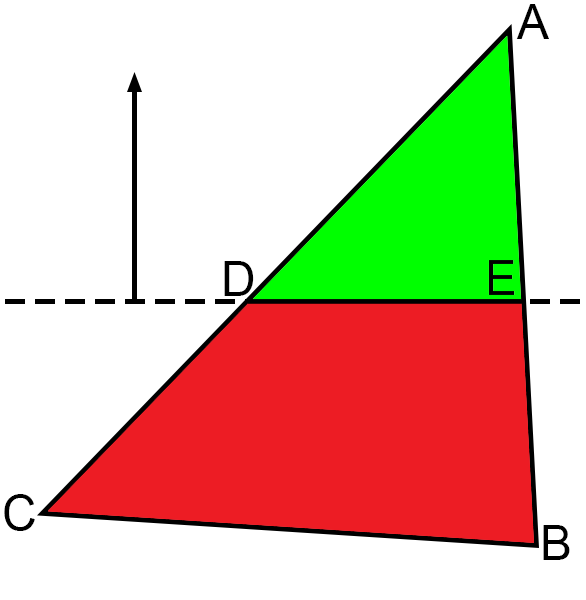
 

Figure 17. Two vertices in front. Figure 18. One vertex in front.

To calculate the vertices *D* and *E*, we can use the following equations, where *t* is the fraction of segment *AB* where the intersection occurs, and *Q* is the point at the intersection. The same equations can be used to calculate the intersection of *AC* and *BC* as well. We will also make use of *t* in future sections to interpolate vertex attributes, such as texture coordinates, for our clipped triangles. [1.]

A pseudocode implementation of this is below:

ClipTriangle(tri, plane):

//Calculate the distances for each vertex

float d0 = plane.n.Dot(tri.v0) + plane.d;

float d1 = plane.n.Dot(tri.v1) + plane.d;

float d2 = plane.n.Dot(tri.v2) + plane.d;

//All are in front

if (d0, d1, d2) > 0:

return tri

//All are behind

else if (d0, d1, d2) < 0:

return none

//One is in front

else if d0 \* d1 \* d2 > 0:

sort tri that a = positive vertex

//Calculate both t values

tD = plane.d – plane.n.Dot(a) / plane.d.Dot(b - a)

tE = plane.d – plane.n.Dot(a) / plane.d.Dot(c - a)

//Calculate both D and E

d = a + (b - a) \* tD

e = a + (c - a) \* tE

//Decompose into 1 triangle

return (a, d, e)

//Two are in front

else:

sort tri that c = negative vertex

//Calculate both t values

tD = plane.d – plane.n.Dot(a) / plane.d.Dot(c - a)

tE = plane.d – plane.n.Dot(b) / plane.d.Dot(c - b)

//Calculate both D and E

d = a + (c - a) \* tD

e = b + (c - b) \* tE

//Decompose into 2 triangles

return (a, b, d), (b, d, e)

### Depth Buffering

Now we can correctly render any one triangle, but rendering objects such as a cube still produces strange results. The cube in the image below doesn’t even look like a cube.

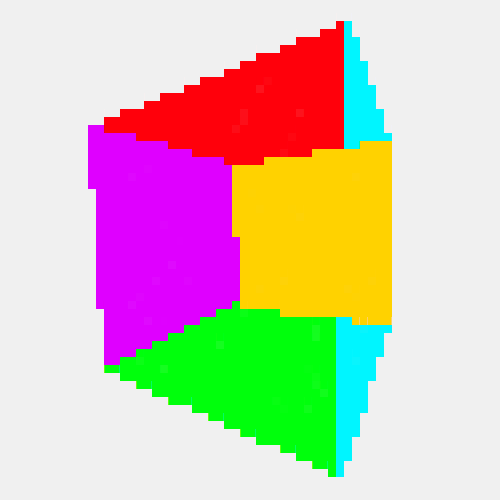


Figure 19. No depth buffer.

This is because some of the cube’s faces are being drawn in the wrong order. In this case we draw the closest faces first and later draw the farther faces overriding the pixels of the closer faces. We could try to sort the faces from back to front, but this is both computationally expensive and impossible for certain combinations of triangles. Therefore, we need to approach the problem on a per-pixel basis. [1.]

The idea is that we keep track of each individual pixel’s z position and only draw over it if the new pixel is closer to the camera. For this we need to add a depth buffer to our canvas. It can simply be a two-dimensional array of floats where every pixel has its own float. Before starting to render a frame, we need to make sure to initialize the array to the farthest possible point, infinity. Then we compare the current depth value to the pixel we’re drawing, discard it if it is farther away, and store it in the depth buffer if it is closer. [1.] Below is the canvas pseudocode updated to include a depth test.

canvas = Color[width][height]

depthBuffer = float[width][height]

PutPixel(x, y, z, color):

//If this pixel is closer than any previously drawn one

if z < depthBuffer[x][y]:

//Store the new closest pixel in the depth buffer

depthBuffer[x][y] = z

canvas[x][height - 1 – y] = color

However, we don’t yet have the z positions for each pixel; we only have the positions of the vertices. To get the z positions, we have to interpolate them for every pixel. Because of how our triangle rasterizer works, it makes the most sense to first interpolate the values for every edge and then for the interior pixels. Creating a function for this is quite easy; we just calculate the difference between the two values we know and add a fraction of that to the first value for every position we want to interpolate for. It is a very similar idea to the basic line drawing algorithm: [1.]

Interpolate(start, end, a, b):

results = []

range = abs(start – end)

//Calculate the change per step

m = (b - a) / (range – 1)

i = a

//For each step in range

for range:

results.append(i)

i += m

return results

//Calling the function for an edge

zValues = Interpolate(v0.y, v1.y, 1 / v0.z, 1 / v1.z)

When interpolating anything for canvas coordinates (pixels), we need to correct for the distortion caused by perspective. In an orthographic projection, z changes linearly with x and y, but this is not true for a perspective projection. However, 1/z does change linearly with a perspective projection, so we can simply interpolate the values of 1/z instead. The only other difference this makes in code is that we prioritize the larger value of 1/z in the depth buffer and initialize it to 0, which is basically 1/∞. With this, our cube looks much better. [1.]

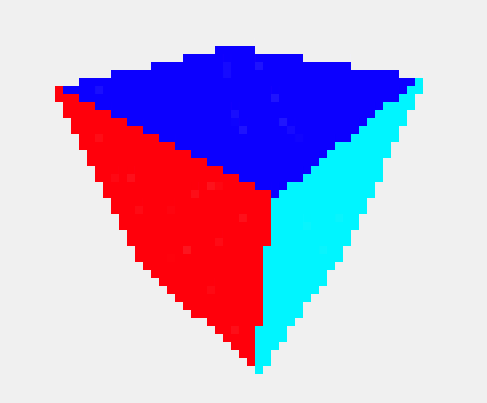


Figure 20. Depth buffer.

### Back Face Culling

A small but very effective optimization we can implement here is to not render the back faces of objects which are obscured by the front faces. The below image shows the basic idea from a 2D view; if the angle between the normal vector of the face and the vector from the face to the camera is more than 90°, we don’t render it. [1.]

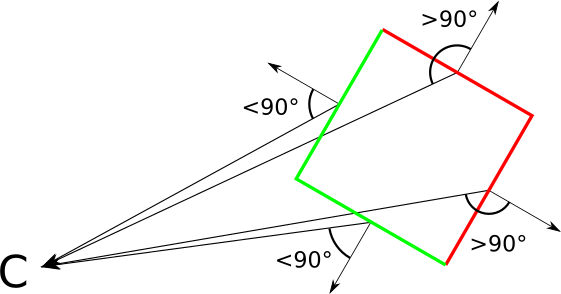


Figure 21. Classifying front and back facing sides. [1]

Calculating this is quite simple: we can simply take the dot product of the two vectors. If the result is greater than zero, the angle is greater than 90°. If we have a vertex of the triangle *V*, the normal of the triangle *N*, and the position of the camera *P*, we cull the faces that satisfy the equation: [1.]

We already have *V* and *P*, but the normal vector *N* is harder to get. Using the cross product of vectors, we can calculate a vector that is perpendicular to two other vectors. This means that we can get the normal vector of our triangle by calculating the cross product of the vectors formed by the vertices of the triangle. Given triangle *ABC*, the normal vector *N* can be calculated as follows: [1.]

The only problem now is that there are always two vectors that are perpendicular to two vertices and depending on the order of the vectors in the cross product, we can get one or the other. Fortunately, the solution is simple: if triangle *ABC* is defined in a clockwise order when looking at it from the front, the normal vector given by our equation will point towards the camera, thus satisfying our definition of front facing. [1.] 3D object file formats such as .obj define the vertices this way, so it is good to use this convention [4].

## Texturing

Currently we can draw any 3D model with the limitation of a maximum of one color per triangle. For more detail, models are often combined with materials and textures. Materials mostly contain information about how light should interact with the object, such as appearing metallic or reflective [7]. This is way out of the scope of this paper, so we will instead focus on the other technique: texturing.

Textures are images which we essentially “paint” on to the triangles of a model. To do this, we must first specify what region of the texture is applied to which triangle, which is done by assigning a texture coordinate to each vertex. For this we need a coordinate system to refer to the points of a texture. Since a texture is a 2D array of colors, we could use x and y for the coordinates, however, x and y already refer to the coordinates of the canvas, so by convention we use u and v for the texture coordinates. Therefore, this process of mapping the vertices of an object to positions on a texture is called UV mapping. [1.] At this point manually defining the model is becoming very arduous. It is a good idea to implement object loading from file, as modeling programs such as Blender make the process of modeling and UV mapping much easier. It is also very difficult to manually define a texture in code, so loading images from files is also very important.

Consider Figure 22 where we have defined the texture coordinates to start from (0, 0) at the bottom left and end at (1, 1) at the top right. The reason we represent texture coordinates as real numbers between 0 and 1 instead of pixel coordinates is because it makes the, or size, of the texture image irrelevant. [8.] Therefore, mapping this texture directly to the side of a cube means the vertices of the two triangles forming that side will have the texture coordinates (0, 0), (0, 1), (1, 0) and (1, 1), (1, 0), (0, 1).

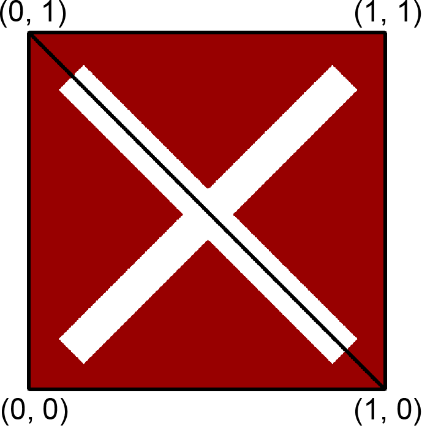


Figure 22. UV mapping.

Now in our rasterizer instead of drawing each pixel of the triangle the same color, we instead look up the proper color from the texture image based on the current pixel’s texture coordinates. Sampling the texture is very simple: all we do is multiply the normalized texture coordinates by the texture’s width and height. However, we still only have the texture coordinates for the vertices. To get each pixel’s texture coordinates, we use the same method as with their depth value: linear interpolation. Here we will also have to be wary of perspective distortion and interpolate u/z and v/z instead. To get u and v back we divide the interpolation results by 1/z which we interpolated in section 2.4.6. [1.] The modified DrawTriangle function is below, with some previously discussed parts omitted for brevity.

DrawTriangle(v0, v1, v2, texture):

...

//Interpolate the edge texture coordinates

v01TexCoords = Interpolate(v0.tex / v0.pos.z, v1.tex / v1.pos.z,

v0.pos.y – v1.pos.y)

v12TexCoords = Interpolate(v1.tex / v1.pos.z, v2.tex / v2.pos.z,

v1.pos.y – v2.pos.y)

v02TexCoords = Interpolate(v0.tex / v0.pos.z, v2.tex / v2.pos.z,

v0.pos.y – v2.pos.y)

//Combine the two lists of the segmented side

v01TexCoords.removeLast()

v012TexCoords = v01Bounds.append(v12Bounds)

//Check which side is left and right

...

if v01.x < v02.x:

...

swap(leftTexCoords, rightTexCoords)

//Draw each horizontal line

for y from v0.pos.y to v2.pos.y:

...

//Interpolate each texture coordinate for this horizontal line

//We don’t divide by z since the edge interpolation already did

rowTexCoords = Interpolate(

leftTexCoords[y - y0], rightTexCoords[y - y0],

leftBounds[y - y0] - rightBounds[y - y0])

//Draw ever pixel in the row

for x from leftBounds[y - y0] to rightBounds[y - y0]:

//Draw the color at the corresponding texture coordinate

//Here we divide by 1/z to get back u and v

PutPixel(x, y, texture.at(rowTexCoords[i] / rowZPos[i++]))

With this our renderer is capable of drawing much more detailed objects. There are still numerous improvements we could add, such as bilinear filtering and mipmapping. These techniques improve the look of textures when they are either very large, or very small on the screen. [1.] However, these are out of scope for this paper so we will not be going over them here.



Figure 23. Textured Cube.

## Shading

Shading is a very important component in rasterization, as without it we lose a lot of definition, especially in more complex models.



Figure 24. A complex model with no shading

In this context, shading doesn’t necessarily mean casting shadows, it encompasses any techniques that apply the effects of light on entire objects [1]. Shaders are a very complex topic, and their effects can even approach the realism of ray tracing with far faster render speeds [4]. However, here we will just go over the most basic form of shading: flat shading. This approach calculates the amount of light on a per triangle basis, so every pixel in a triangle will have the same level of illumination, but pixels in different triangles can have different levels of illumination. [1.]

To start with, for the purposes of flat shading, every point in a triangle will have the same normal vector. Therefore, we can calculate the illumination of an arbitrary point *IP*, using the triangle’s normal and apply it to the entire triangle with the following equation: [1.]

Here *IA* is the ambient light intensity, *N* is the normal vector of the point/triangle, and *L* is a set of the directional vectors of every light. In the context of vectors *|N|* is the length of that vector. The sigma notation might look scary, but in code it is quite simple. We also don’t want to include any lights where *N ∙ L < 0*, since in these cases the light is behind the point. [1.]

//Calculate per triangle

illumination = ambientLight

for light in lights:

n = normal.Dot(light)

d = Length(normal) \* Length(light)

if n > 0:

illumination += n / d \* light.intensity

To apply this illumination, for each pixel, we can simply multiply the color of its sampled texture, with the triangles illumination value. Also make sure to clamp the resulting color to its maximum possible value to prevent unintended behavior.

//Get the final color by multiplying texture with illumination

color = texture.at(u, v) \* illumination

//Clamp the color to a maximum of 255

color = min(color, 255)

PutPixel(x, y, color)

As you can see, even simple flat shading makes a huge difference and is essentially required for rasterization. There are still numerous improvements we could make to this shading, such as Phong shading, which calculates the illumination separately for each pixel. That would fix the issue of each triangle being clearly visible and make the model appear smoother, as well as adding specular highlights. But that is out of the scope of this paper.



Figure 25. A complex model with flat shading

# Implementation

For the implementation of this paper, the broad goal was to implement the rasterization and rendering techniques on the CPU, using the Windows Console as an output window. It is intended as a real-time rendering engine capable of drawing a simple 3D scene. It is made almost entirely using C++ and its standard library, but some features out of the scope of the paper use external libraries. The idea of rendering 3D graphics in the console comes from Ben Ryves’ demo *ASCII Madness*. Although here the idea was to create a more interactable demo and a general-purpose 3D rendering engine. The full source code for this implementation can be found on GitHub: [github.com/Dudeman85/CVid](https://github.com/Dudeman85/CVid)

## Tools and Libraries

As stated above, this implementation is made in C++ 23. For building, CMake and Visual Studio were used. The library tinyobjloader was used for loading 3D models from .obj files. It was chosen for its simplicity and because it has a header-only implementation, simplifying the build process. For loading textures from image files, the STB Image library was used. The reasoning for it is the same as for tinyobjloader. Everything else is implemented using the standard C++ library and the Windows API. All 3D models and textures were made using Blender and Photoshop.

## API Structure

The implementation interface is made with a high level of abstraction to make its usage as easy as possible. Most components, such as the window, camera, model, and texture are simple classes with built in methods for usage such as changing position or other properties. The main function for rendering is DrawModel, which is an abstraction for drawing the vertices and indices of a ModelInstance object. A ModelInstance is a drawable version of a Model, which contains its transform and other instance specific data. This is done for the sake of abstraction and so that the same 3D model can be drawn multiple times while only having to load it once. A basic program for setting up a scene is below:

#include <cvid/Window.h>

#include <cvid/Model.h>

#include <cvid/Renderer.h>

...

//Make a console window with width, height, and name

cvid::Window window(64, 64, "CVid");

window.enableDepthTest = true;

//Make the camera with {x, y, z}, width, height

cvid::Camera cam({0, 0, 100}, 64, 64);

//Set it as perspective with fov, near, and far

cam.MakePerspective(90, 1, 100);

//Load a model from file

cvid::Model cube("../../../resources/cube.obj");

//Create an instance of the model and change its transform

cvid::ModelInstance cubeInstance(&cube);

cubeInstance.SetScale(20);

cubeInstance.SetPosition({10, 20, 0});

//Rotations use radians

cubeInstance.SetRotation({cvid::Radians(-30), cvid::Radians(17), 0});

//Render loop

while (true)

{

//Fill the canvas with some rgb value at the start of frame

window.Fill({0, 0, 0});

window.ClearDepthBuffer();

//Draw the model instance to the window's canvas

cvid::DrawModel(&cubeInstance, &cam, &window);

//Draw the frame to the window, end the program on failure

if (!window.DrawFrame())

return 0;

}

The rasterizer can also be used directly. If you want to draw a point, triangle, or line entirely consisting of one color directly to the canvas, it can be done with the functions below. Also note that depth buffering is still applicable here unless it is turned off and expects visible pixels to be in the range (0, ∞). Since this is rendering to a console, you can also write any ASCII character, or string of them, to the screen. This works slightly differently; there is no depth test, and the vertical resolution is halved, so you need to divide your intended y position by 2. In addition to the character, you can also specify the background and foreground colors to use.

#include <cvid/Rasterizer.h>

...

//Draw a green point directly to the canvas

cvid::RasterizePoint(&window, {12, 35, 0}, {51, 204, 51});

//Draw a red line directly to the canvas

cvid::RasterizeLine(&window, {0, 63, 0}, {63, 0, 0}, {204, 0, 0});

//Draw a blue triangle directly to the canvas

cvid::Tri verts{{5, 5, 0}, {25, 25, 0}, {50, 10, 0}};

cvid::RasterizeTriangle(&window, verts, {0, 153, 255});

//Write a string directly to the canvas

window.PutString(20, 25, "Hello World!",

{153, 0, 153}, {240, 240, 240});



Figure 26. A bit of everything.

## Components

The program is divided into five main parts: the window, camera, model, renderer, and rasterizer. There is also a custom vector and matrix math library which doesn’t directly relate to the render pipeline but is still used in every component. Following the render pipeline the broad job of each component is as follows:

Math: Do mostly vector and matrix math

Model: Load objects and materials from files, triangulate, and calculate their spatial properties.

Camera: Calculate the clip space, view, and projection.

Renderer: Transform, clip, and cull each triangle so it ends up in canvas coordinates.

Rasterizer: Interpolate vertex attributes, sample textures, and draw each triangle to the canvas.

Window: Implement PutPixel, depth buffer, and display the canvas to CMD.

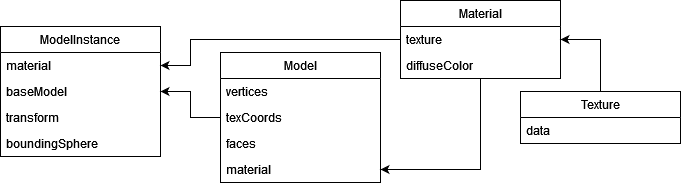
### Math

Since computer graphics make heavy use of vector and matrix math, a good way to get a deeper understanding of these is to make your own math library. The library used here is very simple, having only 2, 3, and 4 sized vectors and matrices. It uses column major matrices and column vectors. The library implements operators to index, compare, and multiply matrices and vectors. It also has useful methods in rendering. For vectors, length, normalization, and the dot and cross products are very important. For matrices there are functions to construct translation, rotation, and scaling matrices. While the interface is quite different, the implementation of this library is roughly based on GLM and heavily tested against it to make sure no bugs in rendering were caused by it.

### Model and Texture

The model loading for this implementation is made with tinyobjloader. It loads the 3D models from .obj files into two arrays, one containing vertex data, such as positions and texture coordinates, and the other containing faces with indexes to the vertex array. Since triangulation is done by tinyobjloader, this data can be copied almost directly to the model class, with the only differences being that vertices and texture coordinates are separated into their own arrays and all other vertex attributes are discarded. Tinyobjloader also loads any applicable materials from .mtl files. These materials contain a lot of information about how to shade the object, but this implementation only uses the diffuse color and texture, which are stored in a material class. Loading textures from image files is done with the stbimage library, which loads the image data into a single array with 1 byte per color channel. The only processing done to this data is combining the separate rgba channels into a struct.

This model class is not yet able to be rendered, as it first needs a transform and bounding sphere for clipping. Here a class called ModelInstance takes care of this problem. Every model instance has its own transform and bounding sphere, as well as a pointer to a model and material. This is one way to allow multiple models to be placed in the scene at different points without having to copy their vertex data. It is also possible to change the material of only one instance of the model this way. This is the part of the program with the most complex structure, so below is a diagram showing the relationship between these classes.



### Camera

The camera here is a very simple class. Its only jobs are to define the clip space and calculate the view and perspective or orthographic projection matrices. It has position and rotation, which are used to calculate the view matrix, as well as width and height, which are used to calculate the projection matrix and clip planes. For perspective projection there is also a field of view. The values of view, projection, and the clip planes are cached and only updated when required. For the view, this is whenever the position or rotation changes. For the projection and clip planes, this is when the projection type, fov, width, or height changes. This is a very simple, but effective optimization, otherwise these would be calculated once every frame for each object. There are also some useful utility functions such as getting the direction vectors of the camera.

### Renderer

The primary function of the renderer is DrawModel. It takes in a model instance, camera, and window, and processes the model’s vertices up to window coordinates. It starts by clipping the entire model against every clip plane. This is done before any vertex data is touched, as clipping the model only requires transforming the center point and scaling the radius of the bounding sphere, no vertex data is necessary. The clipping function returns a bit set with the first bit marking a partial intersection and each subsequent bit corresponding to a clip plane.

void DrawModel(ModelInstance\* model, Camera\* cam, Window\* window)

{

//Clip the bounding sphere agains all planes

bitset<8> clip = ClipModel(model, cam);

//Fully outside clip space

if (clip.none())

return;

...

}

Next it copies the vertices and applies the model transform to them. After this, it calculates the normal vector of each face using the cross product of two vectors formed by the face’s vertices and classifies backwards facing ones as culled. It then applies the view transform to every vertex, even ones which only belong to faces that have been culled, which could be a possible future optimization.

...

//Copy the vertices from the base model

vector<Vertex> vertices = model->GetBaseModel()->vertices;

//Apply model transform to all vertices

for (Vertex& vert : vertices)

vert.position = model->GetTransform() \* Vector4(vert.position, 1.0);

//Recalculate normals, and cull backwards faces

vector<bool> culled;

for (const IndexedFace& face : model->GetBaseModel()->faces)

{

//Calculate the surface normal

Vector3 v1 = vertices[face.verticeIndices[1]].position –

vertices[face.verticeIndices[0]].position;

Vector3 v2 = vertices[face.verticeIndices[2]].position –

vertices[face.verticeIndices[0]].position;

Vector3 normal = v1.Cross(v2);

//Cull backwards facing faces

Vector3 vc = vertices[face.verticeIndices[0]].position –

cam->GetPosition();

culled.push\_back(vc.Dot(normal) >= 0);

}

//Apply view transform to all vertices

for (Vertex& vert : vertices)

vert.position = cam->GetView() \* Vector4(vert.position, 1.0);

...

Finally, it loops through every non-culled face, clips them against every plane the object intersected, and applies the projection and viewport transform to their vertices. After this each face is drawn onto the canvas by the rasterizer. Note that the vertices are copied here again to non-indexed faces, meaning vertices are no longer shared by faces. This is done to solve the problem of adding new vertices in clipping, but it could be optimized for better memory usage in the future.

...

//For each face in the model

for (size\_t i = 0; i < model->GetBaseModel()->faces.size(); i++)

{

if (culled[i])

continue;

//Copy the indexed face's vertices and texture coords

Face face = model->GetBaseModel()->faces[i].Unindex();

//The final list of faces to render

vector<Face> faces{face};

//If model is partially intersecting at least one plane

if (clip.count() > 1)

//Clip the triangle against every intersecting plane

faces = ClipFace(face, cam, clip);

//Loop over every face the original face was decomposed into

for (Face& face : faces)

{

//Convert to clip space

Vector4 v1 = Vector4(face.vertices.v0, 1.0);

Vector4 v2 = Vector4(face.vertices.v1, 1.0);

Vector4 v3 = Vector4(face.vertices.v2, 1.0);

//Apply projection

v1 = cam->GetProjection() \* v1;

v2 = cam->GetProjection() \* v2;

v3 = cam->GetProjection() \* v3;

//Normalize

v1 /= v1.w;

v1 /= v1.w;

v2 /= v2.w;

face.vertices = { v1, v2, v3 };

//Convert from clip space to screen space

face.vertices \*= window->GetDimensions() / 2;

face.vertices += window->GetDimensions() / 2;

//Draw the face (triangle)

RasterizeTriangle(window, face, model->GetMaterial());

}

}

### Rasterizer

Now that the faces of the models have been processed and transformed into canvas coordinates, they can finally be drawn to the canvas by the rasterizer. The primary function of the rasterizer is DrawTriangle. It takes in a face, which is a triangle and its attributes, a window, and optionally a material and draws each pixel of the triangle with the correct color to the canvas. The function starts by sorting the vertices and doing some restructuring of the data into separate vertices and attributes to be interpolated. Here it is important to interpolate 1/z an uv/z to correct for perspective distortion:

void RasterizeTriangle(Window\* window, Face tri, const Material\* mat)

{

//Round the vertices

Vector2Int p0 = tri.vertices.v0;

Vector2Int p1 = tri.vertices.v1;

Vector2Int p2 = tri.vertices.v2;

//x, z, and texture coords to be interpolated

Attributes a0 = {round(tri.vertices.v0.x), 1 / tri.vertices.v0.z,

tri.texCoords.v0 / tri.vertices.v0.z};

Attributes a1 = {round(tri.vertices.v1.x), 1 / tri.vertices.v1.z,

tri.texCoords.v1 / tri.vertices.v1.z};

Attributes a2 = {round(tri.vertices.v2.x), 1 / tri.vertices.v2.z,

tri.texCoords.v2 / tri.vertices.v2.z};

//Sort the vertices in vertically descending order

if (p0.y < p1.y)

SWAP(p0, p1); SWAP(a0, a1);

if (p0.y < p2.y)

SWAP(p0, p2); SWAP(a0, a2);

if (p1.y < p2.y)

SWAP(p1, p2); SWAP(a1, a2);

//If p2 is to the left of p3, the full segment will be on the right

Vector2 v1 = Vector2(p1 - p0).Normalize();

Vector2 v2 = Vector2(p2 - p0).Normalize();

bool fullOnRight = v1.x < v2.x;

...

}

Next, the x bounds and attributes are interpolated. Since the triangle is drawn from left to right in horizontal lines, it is necessary to figure out which sides are on the left and right. Also, since either the left or right bound of the triangle is made up of two sides, we combine them into one list. Here it is also good to note a possible bug with the attribute interpolation. If interpolating edge attributes for each y position the normal way, the results are subtly wrong; the interpolated attribute might not be at the true edge of the triangle, but instead at the center of the horizontal segment making up the y position. To fix this we need to make sure to interpolate the attributes at either the left or edges of the horizontal segment, depending on if the side is on the right or left of the triangle. Figure 27 shows the improper and proper interpolation positions, this problem is worse with more horizontal sides. As you can see, improper interpolation can make the texture appear warped at the edges of the triangle.

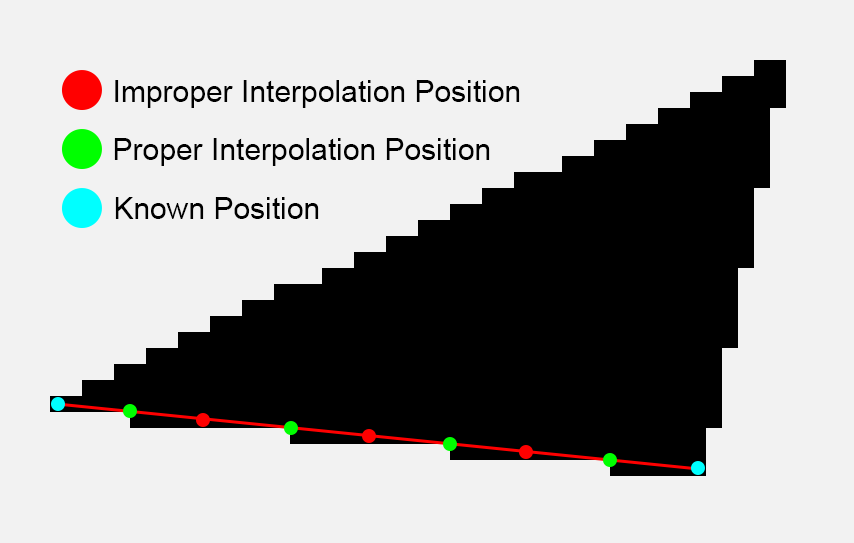


Figure 27. Interpolation positions.

 A red and white x sign

Description automatically generated

Figure 28. Improper Interpolation. Figure 29. Proper Interpolation.

...

//Interpolate the vertex attributes for each side (x, z, texCoord)

vector<Attributes> combinedSegment =

InterpolateAttributes(p2, p1, a2, a1, fullOnRight);

vector<Attributes> shortSegment =

InterpolateAttributes(p1, p0, a1, a0, fullOnRight);

vector<Attributes> fullSegment =

InterpolateAttributes(p2, p0, a2, a0, !fullOnRight);

combinedSegment.insert(combinedSegment.end(),

shortSegment.begin() + 1, shortSegment.end());

//Figure out which segment is on which side

vector<Attributes>\* ls = &combinedSegment;

vector<Attributes>\* rs = &fullSegment;

//If the full segment is on the right side, swap the segments

if (fullOnRight)

SWAP(ls, rs)

...

Now that the left and right bounds of the triangle have been calculated, all that’s left is to loop over every y position of the triangle, interpolate the attributes for the horizontal segment, and draw the pixel. Here the previous interpolation issue does not apply, so normal interpolation is used. This is also the point where texture sampling happens if one is provided, otherwise the final color is determined by the diffuse color.

...

//For each y coordinate in the triangle

int startY = (int)std::round(p2.y);

for (int yi = 0; yi < fullSegment.size(); yi++)

{

//Interpolate for z positions for each horizontal scanline

vector<double> zPos = LerpRange(rs->at(yi).x, ls->at(yi).x,

rs->at(yi).z, ls->at(yi).z);

//Interpolate texture coordinates if applicable

vector<Vector2> texCoords;

if (mat->texture)

texCoords = LerpRange2D(rs->at(yi).x, ls->at(yi).x,

rs->at(yi).texCoord, ls->at(yi).texCoord);

//Draw a line from the full segment to the split segment

int startX = rs->at(yi).x;

for (int xi = 0; xi <= ls->at(yi).x - rs->at(yi).x; xi++)

{

Color color = mat->diffuseColor;

//Get the color from the texture if it exists

if (!texCoords.empty())

{

Vector2Int texScale = mat->texture->size - 1

Vector2Int samplePos(

round(texCoords[xi].x / zPos[xi] \* texScale.x),

round(texCoords[xi].y / zPos[xi] \* texScale.y));

//Sample the texure

color = mat->texture->GetTexel(samplePos);

}

//Attempt to draw the pixel

window->PutPixel(startX + xi, startY + yi, color, zPos[xi]);

}

}

### Console Window

As per Microsoft: "A console is an application that provides I/O services to character-mode applications." This essentially means a console can read user input, such as keypresses or mouse movements, into an input stream, and render the text contents of an output stream onto the screen. The Windows Console can render the entire Unicode character set, but here only characters from CP 850 are used since it is the default and contains every necessary character. [11.] The most important characters are 223(▀), 220(▄), and 219(█), as these can be used to represent an upper, a lower, and two stacked pixels, thereby essentially doubling our vertical resolution since a character in CMD is twice as tall as it is wide. Characters 176(░), 177(▒), and 178(▓) are also potentially useful since they could allow color blending through dithering, giving the render a more ASCII art feel. However, using these would mean cutting the vertical resolution in half.

Dithering is not actually necessary here since the Windows Console supports full rgb colors with a separate background and foreground color for each character. Therefore, rendering pixels using the CP 850 character set the only necessary character to draw is 223(▀), as both pixels represented by this character can be set by changing the background and foreground colors. Admittedly, this does lose the style of ASCII art and ends up looking like a normal low-resolution render, but fixing this by using 176(░), 177(▒), and 178(▓) with a more limited color palette is more of a future improvement than a requirement.

There are two ways to operate the console, either through the console API, or through virtual terminal sequences (VTS). The console API uses a set of C++ functions defined by Microsoft to change the state of the console, such as setting the cursor position, changing the pen color, or writing text. Virtual terminal sequences on the other hand are a set of functions represented as non-printable characters which can be output in between normal text to change the state of the terminal. This implementation uses mostly virtual terminal sequences, because Microsoft recommends them over the API, and they are cross compatible with many other terminal emulators besides just CMD and PowerShell. They are admittedly still missing some functionality of the API, so it is still necessary to use API functions for certain things. [11.]

The window here is a class that mainly contains a frame buffer (canvas), a depth buffer, and PutPixel and PutChar functions. The frame buffer holds an array of characters with their background and foreground colors. This entire array will be printed to the console when the frame is drawn, with the background and foreground colors being set by VTS. The full string for each character is “\x1b[38;2;{r};{g};{b}m\x1b[48;2;{r};{g};{b}m{c}”. This looks a bit complicated, but the structure here is basically “background;r;g;b;foreground;r;g;b;character”. The DrawFrame and PutPixel functions are below, they are quite simple at the end of the day. The PutChar function is even simpler, just placing a character directly onto the framebuffer without a depth test.

void Window::DrawFrame()

{

string frameString;

frameString.reserve((size\_t)29 \* width \* (height / 2));

//For every pixel in the framebuffer

for (size\_t y = 0; y < height / 2; y++)

{

//Windows 11 broke text wrapping, so do we it here

//Also for some reason it starts from 1

frameString.append(std::format("\x1b[{};0f", y + 1));

for (size\_t x = 0; x < width; x++)

{

CharPixel& pixel = frameBuffer[y \* width + x];

//Add the proper vts and character

//Format: \x1b38;2;<r>;<g>;<b>;m

frameString.append(format("\x1b[38;2;{};{};{}m",

pixel.fg.r, pixel.fg.g, pixel.fg.b));

frameString.append(format("\x1b[48;2;{};{};{}m{}",

pixel.bg.r, pixel.bg.g, pixel.bg.b,

pixel.character));

}

}

//Print the frame

cout << frameString;

}

bool Window::PutPixel(uint16\_t x, uint16\_t y, Color color, double z)

{

//Make sure the pixel is in bounds

if (x >= width || y >= height || z < 0)

return false;

//Make sure there is not already a closer pixel

if (enableDepthTest)

{

//Smaller z means further away

if (z < depthBuffer[y \* width + x])

return false;

depthBuffer[y \* width + x] = z;

}

//Always print 223 where fg is the top and bg is the bottom.

CharPixel& pixel = frameBuffer[(height - 1 - y) / 2 \* width + x];

//Set the pixel character

pixel.character = (char)223;

//Top or bottom pixel

if (y % 2 == 0)

pixel.bg = color;

else

pixel.fg = color;

return true;

}

Of course, an important part of a window is the ability to resize it, which luckily the Windows Console supports. However, the newer Windows Terminal doesn’t support resizing, and since it is the default console for Windows 11 this implementation won’t work on it unless the default is changed back to Windows Console by the user. For this implementation it is important to note that the vertical resolution the console uses is in characters, which means it will be half of the window’s vertical resolution. An inconvenience with the API also means the window size must be smaller than the screen buffer size, so it is initially set to its smallest possible value. [11.]

//Resize the console to fit the frame

void Window::Resize(int w, int h)

{

SMALL\_RECT minSize{0, 0, 1, 1};

SMALL\_RECT consoleSize{0, 0, w - 1, ceil((float)h / 2) - 1};

COORD sbSize{w, ceil((float)h / 2)};

//SetConsoleWindowInfo has to be called first with minSize

SetConsoleWindowInfo(consoleOut, true, &minSize);

SetConsoleScreenBufferSize(consoleOut, sbSize);

SetConsoleWindowInfo(consoleOut, true, &consoleSize);

}

## Demo

The demo program of this implementation can also be found on GitHub: [github.com/Dudeman85/CVid](https://github.com/Dudeman85/CVid). It is a simple 3D model showcase, which takes all the .obj models in the resources folder and renders them in the console. Note that for Windows 11 the default console needs to be changed to Windows Console Host.

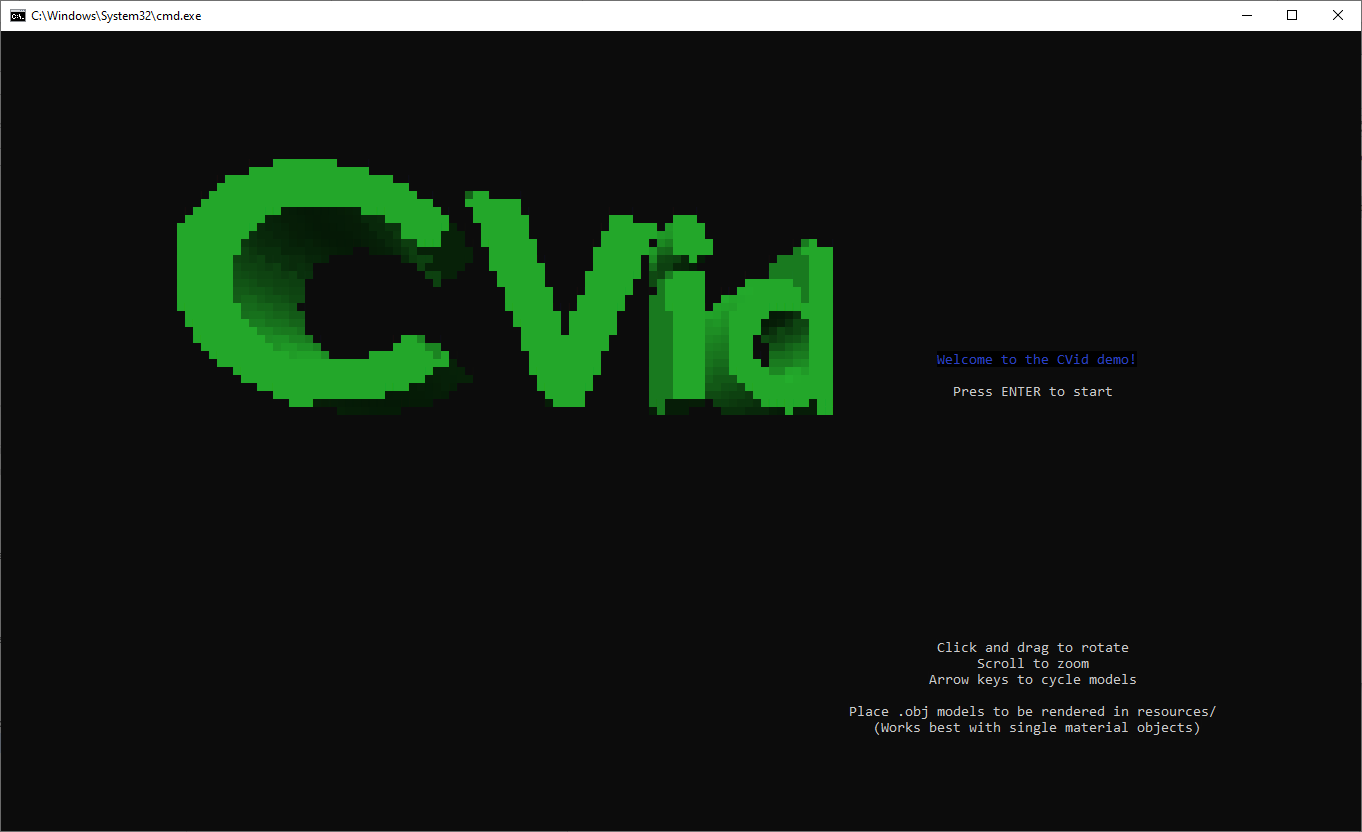


Figure 30. Demo program.

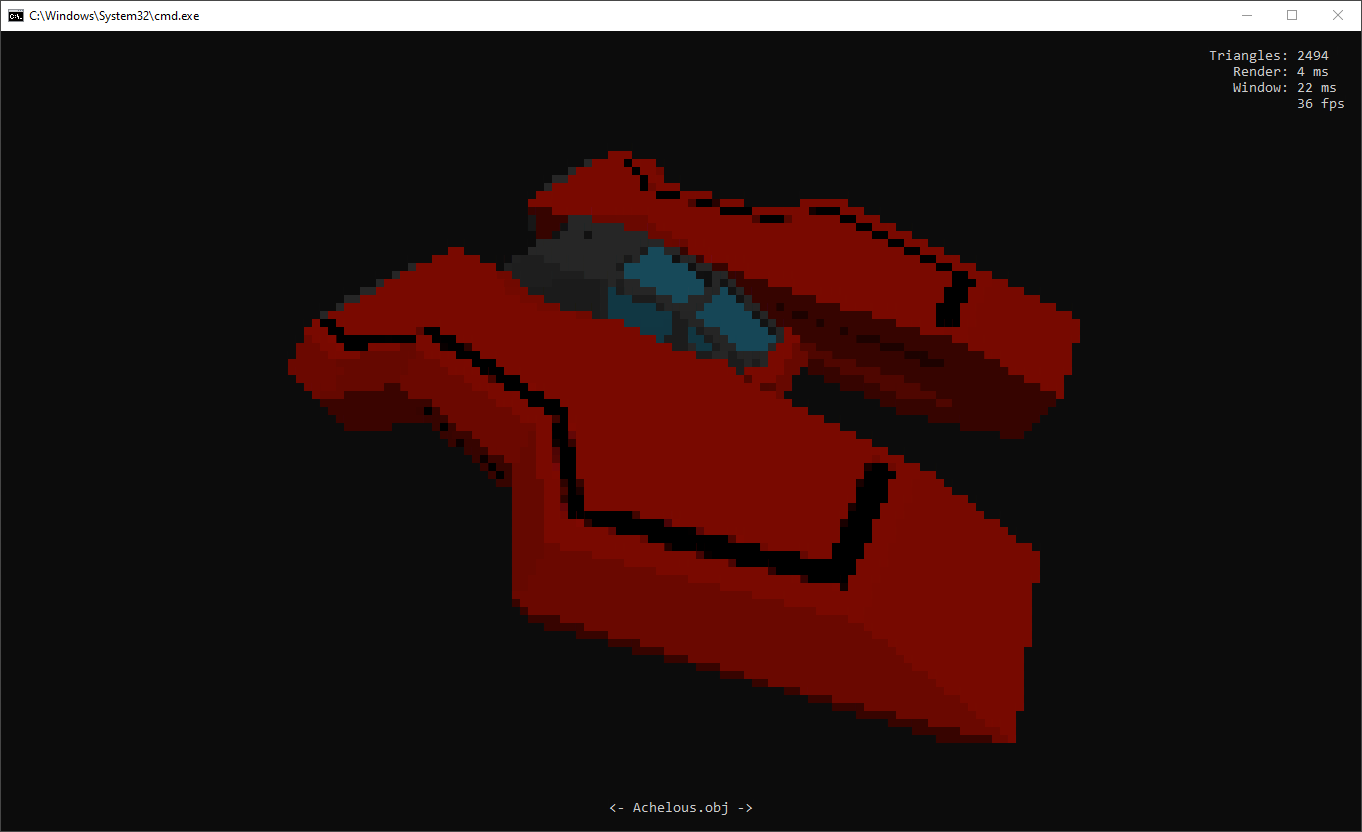


Figure 31. Demo program.

Sources

1. Gambetta G. Computer Graphics from Scratch. 2021. Available: <https://gabrielgambetta.com/computer-graphics-from-scratch/>

2. Microsoft. What Is a Window? 04.27.2021. [Referenced 16.12.2024]. Available: https://learn.microsoft.com/en-us/windows/win32/learnwin32/what-is-a-window-

3. Khronos Group. glViewport. [Referenced 16.8.2024]. Available: https://registry.khronos.org/OpenGL-Refpages/gl4/html/glViewport.xhtml

4. Brian C. What’s the Difference Between Ray Tracing and Rasterization. [Referenced 7.3.2024]. Available: https://blogs.nvidia.com/blog/whats-difference-between-ray-tracing-rasterization/

5. Kennedy J. Bresenham's Integer Only Line Drawing Algorithm. 2012. Available: <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=eb6f1c1f6ee1baf5fdb426af36f575b543ca7f4>

6. Simo Määttä. 21.11.24. Personal Lecture.

7. Murray JD, VanRyper W. Encyclopedia of graphics file formats. 2nd ed. 1996. Wavefront OBJ. [Referenced 7.12.2024]. Available: https://www.fileformat.info/format/wavefrontobj/egff.htm

8. De Vries J. Learn OpenGL [Internet]. [Referenced 1.12.2024]. Available: https://learnopengl.com

9. Jean-Colas Prunier. The Perspective and Orthographic Projection Matrix [Internet]. [Referenced 1.12.2024]. Available: https://www.scratchapixel.com/lessons/3d-basic-rendering/perspective-and-orthographic-projection-matrix/opengl-perspective-projection-matrix.html

10. Edward Red. Homogenous Transformations. ME537: Robotics, Brigham Young University. [Referenced 12.12.2024]. Available: http://www.et.byu.edu/~ered/ME537/Notes/ch1-537.pdf

11. Microsoft. Windows Console Documentation. 10.30.2020. [Referenced 16.8.2024]. Available: <https://learn.microsoft.com/en-us/windows/console/>