Aleksi Anderson

Rasterization and Rendering in a Console



Opinnäytetyö

Tieto ja viestintätekniikka

Syksy 2024

Tiivistelmä

**Tekijä(t)**: Sukunimi Etunimi & Sukunimi Etunimi

**Työn nimi:** Työn pääotsikko ja mahdollinen alaotsikko

**Tutkintonimike:** esim. Insinööri (AMK), rakennustekniikka

**Asiasanat:** aliquam, dignissim, molestie, tortor, vitae

Kirjoita tiivistelmäsivu otsikkoa (Calibri, fonttikoko: 11, tyyli: **Otsikko Tiivistelmä**) lukuun ottamatta muusta työstä poikkeavasti fonttikoolla 10 (Calibri, tyyli: Tiivistelmän tyyli). Tiivistelmän tulisi sopia yhdelle sivulle.

Kirjoita tiivistelmäsivun ylälaitaan seuraavat tiedot: 1. tekijän/tekijöiden suku- ja etunimi, 2. työn pääotsikko ja mahdollinen alaotsikko, 3. tutkintonimike ja 4. asiasanat tärkeysjärjestyksessä. Suomenkielisiä asiasanoja voi etsiä Yleisen suomalaisen asiasanaston avulla <https://finto.fi/yso/fi/>.

Abstract

**Author(s):** Surname First name & Surname First name

**Title of the Publication:** Main Title of the Publication and Possible Subtitle

**Degree Title:** e.g. Bachelor of Engineering, Construction Engineering

**Keywords:** aliquam, dignissim, molestie, tortor, vitae

Write the abstract page in font size 10 (Arial, style: Tiivistelmän tyyli), except for the abstract **title**, which is written in font size 11. (Arial, style: **Otsikko Tiivistelmä**). Abstract should fit on one page.

Abstract must include the following information: 1. surname and first name of the author(s), 2. main title of the publication and possible subtitle, 3. degree title, and 4. keywords in order of importance

Table of Contents

[1 Introduction 1](#_Toc184295617)

[2 Rendering theory 2](#_Toc184295618)

[2.1 Window 2](#_Toc184295619)

[2.2 Canvas 2](#_Toc184295620)

[2.3 Rasterization 3](#_Toc184295621)

[2.3.1 Rasterizing Lines 3](#_Toc184295622)

[2.3.2 Rasterizing Triangles 8](#_Toc184295623)

[2.4 Rendering 11](#_Toc184295624)

[2.4.1 3D Models 11](#_Toc184295625)

[2.4.2 Transforms 12](#_Toc184295626)

[2.4.3 Camera 14](#_Toc184295627)

[2.4.4 Perspective Projection 15](#_Toc184295628)

[2.4.5 Clipping 17](#_Toc184295629)

[2.4.6 Back Face Culling 21](#_Toc184295630)

[2.4.7 Depth Buffering 21](#_Toc184295631)

[2.5 Texturing 24](#_Toc184295632)

[3 Implementation 25](#_Toc184295633)

[3.1 Tools and Libraries 25](#_Toc184295634)

[3.2 Program Structure 25](#_Toc184295635)

[3.3 Components 25](#_Toc184295636)

[3.3.1 Math 25](#_Toc184295637)

[3.3.2 Console Window 25](#_Toc184295638)

[3.3.3 Camera 27](#_Toc184295639)

[3.3.4 Model and Texture 27](#_Toc184295640)

[3.3.5 Renderer 27](#_Toc184295641)

[3.3.6 Rasterizer 27](#_Toc184295642)

[3.4 Demo 27](#_Toc184295643)

[Sources 28](#_Toc184295644)

Symbols

texel

TODO

# Introduction

TODO

# 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++.

{TODO: maybe write an overview of the render pipeline with some pictures}

## Window

https://learn.microsoft.com/en-us/windows/win32/learnwin32/what-is-a-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: [above]

* 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 and read user input, and since CMD satisfies both, it can work as our window. All the theory and implementation of rendering here will still be completely agnostic to the implementation of the window.

## Canvas

Before rendering anything to the window we will first need a canvas, also known as a framebuffer, to draw the frame on. In this case our canvas is 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 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) [opgl docs].

{TODO add canvas coordinate system picture}

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. [2.] However, we will not be covering shaders since they are outside the scope of this paper, which means the final color of our pixels will be decided entirely based on the material color or texture of the object.

### 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 line

Description automatically generatedA black and white line

Description automatically generated

Drawing some lines with this function produces some interesting results. Line one is {TODO add start and end points and slope}, and line 2 is {TODO}. 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 that is beyond the scope of this paper [1]. The bigger problem is that the second line 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 {line pic 1} where the slope is greater than 1 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

A black and white image of a arrow

Description automatically generated

{TODO: Add more different 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 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. Alternatives with anti-aliasing support include Gupta-Sproull and Xiaolin Wu’s algorithms [3.]

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. [3.]

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 have to 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 [3]. 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. The complete 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:

//Make sure starting point is before ending point

if y0 > y1:

swap(x0, x1)

swap(y0, y1)

//Calculate slopes for x and y

dx = x1 – x0;

dy = y1 – y0;

//If slope is positive increment x, else decrement

xi = 1

if dx < 0:

xi = -1

dx = -dx

x = x0

error = 0

//For each y position, plot the corresponding x

for y from y0 to y1:

PutPixel(x, y, color)

error += 2 \* dx

if error > dy:

x += xi

error -= 2 \* dy

### 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.]

{TODO: maybe wireframe triangle picture}

DrawWireframeTriangle(v0, v1, v2):

DrawLine(v0, v1)

DrawLine(v1, v2)

DrawLine(v2, v0)

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:

rightBound, leftBound = CalculateBounds()

DrawLine(rightBound, y, leftBound, 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 lines [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.] Now, 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: {TODO add picture explaining triangle and vertices and sides}

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 [simo]. 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. Puttin all this together we have a simple function to draw filled triangles: [1.]

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. 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.

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)

{TODO add filled triangle picture}

## 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).

{TODO add picture of above}

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. The below image is an overview of everything we need to do in the pipeline.

model space -> apply model matrix -> world space -> calculate normals -> cull backwards faces -> apply view matrix -> view space -> clip faces -> apply projection matrix -> clip space -> apply viewport transform -> screen space -> send to rasterizer

{TODO Add picture of above}

### 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 those triangles describing which three vertices make 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.]

{TODO Add picture of cube}

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. 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. [4.]

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 out 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 around objects. 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 +Y is up. [5.]

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. 3-dimensional transformation matrices are represented in homogenous coordinates (4x4 matrices), but all our vertices are in cartesian coordinates (3x1 matrices, or vector 3s). Therefore, we need to first 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. Converting back from homogenous coordinates to cartesian coordinates is also simple; we divide xyz by w. [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 [6]:

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. [5.]

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: [5.]

scale = Scale(sX, sY, sZ)

rotation = RotateZ(rZ) \* RotateY(rY) \* RotateX(rX)

translation = Translate(tx, ty, tz)

model = translation \* rotation \* scale

### Camera

We can now move around objects; next we need a camera that we can 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 rotating the camera by R produces identical results to rotating everything in the world by -R, there is no difference in the result. The same goes for translation. [1.]

{TODO: add picture}

This is much better than the alternative since it greatly simplifies the projection of 3D vertices to 2D points. It is also easy 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 will 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

### 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. [5.]

Perspective projection of a point can be calculated using basic trigonometry. Consider the below diagram 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 is at the origin facing toward -Z. [7.]

{TODO Add picture}

The points ABP and ACP' form two similar triangles, whose properties we can use to calculate p'. By the properties of similar triangles AC/AB = CP'/BP, substituting our known values and solving for BP we get BP = P'y = n \* Py / -Pz. Also 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. [7.]

Now we have point P projected to the near clip plane, but we still need to map it to NDC. APIs like OpenGL calculate this by defining the left (l), right (r), top (t), and bottom (b) edges of the camera and mapping the point inside those bounds, we will do the same. We also need to map the z coordinate to between (-1, 1) or (0, 1), which we do with the help of the near (n) and far (f) clip planes. The OpenGL projection matrix maps z to (-1, 1), but (0, 1) is a more standard range and easier for us to work with, so we will slightly modify the matrix to achieve this result. [7.]

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. [7.]

{TODO add picture}

From the above image 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 convert its homogenous coordinates back into cartesian coordinates by diving xyz by w. [7.] Our DrawTriangle function expects the vertices to be in canvas coordinates, luckily converting from NDC is very simple. 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.

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

//Transform every vertice 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. 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.]

{TODO add clipping volume picture}

We can first 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 much 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:

{TODO add picture explaining clip space calcs from x perspective}

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. Following the same logic and applying the aspect ratio 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*. Actually calculating a bounding sphere for a 3D object is surprisingly complicated, but we can approximate it by calculating the average vertex and its distance to the farthest vertex. Now 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 otherwise the sphere is intersecting the plane. [1.]

{TODO: Add picture of sphere plane}

GetClipPlanes(fov, near, far):

//Calculate the horizontal and vertical angles

vAngle = fov / 2

hAngle = vAngle \* aspectRatio

//Calculate all the clip planes

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

]

return clipPlanes

//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.

{TODO: add picture of this}

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 decompose the triangle into multiple new triangles at the point where it intersects the plane.

{TODO: picture of two and three tri clip cases}

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. We will also make use of *t* in future sections to interpolate vertex attributes, such as texture coordinates, for our clipped triangles.

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

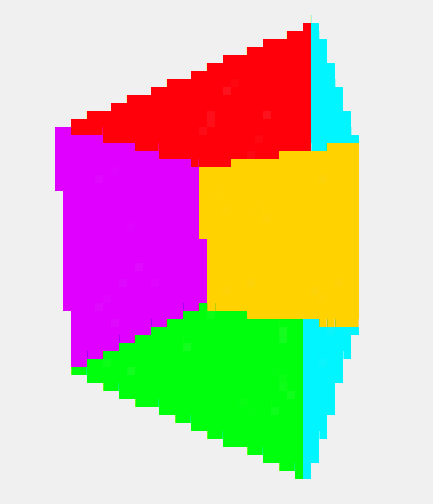
e = b + (c - b) \* tE

//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.



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. First though, we need to make sure to initialize the array to the farthest possible point, infinity, at the start of every frame. 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)

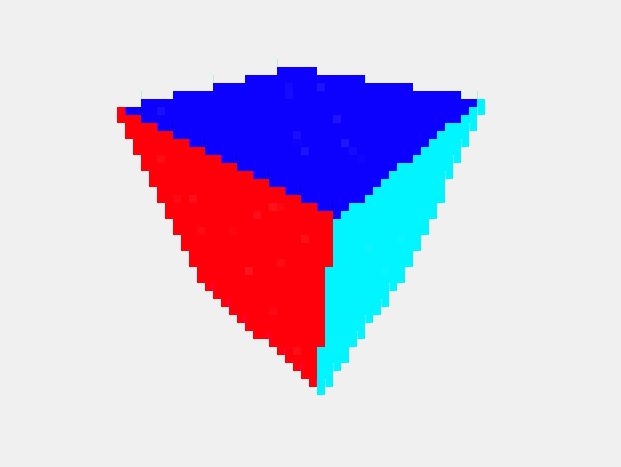
i += m

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.]



### 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.

{TODO: add picture}

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 [4]. This is way out of the scope of this paper, so we will instead focus on the other technique, texturing.

Textures are 2D images which we essentially “paint” on to the triangles of an object. To do this, we must first specify what region of the texture is applied to which triangle. We will do this on a per-triangle basis by mapping each vertex of the triangle to a point on the texture. For this we need a coordinate system to refer to the points in the texture. Since a texture is an image and an image 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. Therefore, it is recommended you implement object loading from file (see 3.3.4 for example), as modeling programs such as Blender make the process of modeling and UV mapping much easier.

Consider the image below; here 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 resolution, or size, of the texture image irrelevant. It is also the convention used by OpenGL and the way texture coordinates are stored in object files. [logl,4.] Therefore, mapping this texture directly to the side of a cube means the two triangles forming that side will have the coordinates (0, 0), (0, 1), (1, 0) and (1, 1), (1, 0), (0, 1).

{TODO: add image of 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 texture coordinates by the texture image 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 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. However, these are out of scope for this paper so we will not be going over them here. Other improvements to the renderer would be adding lighting, but since the goal was to create a very simple renderer, we will not be covering that either.

{TODO: add picture of final textured cube}

# Implementation

For the implementation of this paper, the broad goal was to implement the rasterization and rendering techniques on the CPU, using CMD as a window. It is intended as a real-time rendering engine capable of drawing a simple 3D scene. It is made almost entirely using only C++ and its standard library, the exceptions to this are discussed in the next section. The idea of rendering 3D graphics to CMD comes from Ben Ryves’ demo *ASCII Madness*. Although here the idea was to create a playable demo and a more general-purpose rendering engine. The full source code can be found on GitHub: {TODO: open repo}

## Tools and Libraries

As stated above this implementation is made in C++ 23. For building, CMake and VisualStudio were used. The library tinyobjloader was used for loading 3D models from .obj files. This was chosen for its simplicity and because it has a header-only implementation, simplifying the build process.

TODO: C++, MSVC, Visual Studio, Blender, Cmake, tiny obj loader, stb image, git

## Program Structure

TODO: Explain style, data types, functions, processes

## Components

TODO: open window console, renderer, rasterizer, camera, math, texture, model

Order these as if following the render pipeline

### Math

TODO: Explain the custom vector and matrix library

### Console Window

{TODO: Rewrite this}

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 we will only be using characters from the Windows-1252 character {TODO: at least for now I might upgrade to Unicode} set since we only need a few characters from it, and it simplifies the actual implementation. [msdoc.] The most important characters for us 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. Characters 176(░), 177(▒), and 178(▓) are also potentially useful since they could allow us to blend colors through dithering, however using these would mean cutting the vertical resolution in half.

There are two ways to operate the console, either through the console API, or through virtual terminal sequences. 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. We will use virtual terminal sequence in this paper because Microsoft recommends them over the API, and they are cross compatible with many other terminal emulators besides just CMD and PowerShell. [msdoc.]

The Windows console is capable of rendering 16 different colors in both the background and foreground, each character can have a different color set for its background and foreground color, however we cannot set the background and foreground color to be the same [msdoc]. Since we're using one character to represent two pixels if those pixels have a different color, we need to use the background to represent one of them. It doesn't really matter which of the two is represented by the background, so we can just set a convention of always rendering the 223(▀) character in the case the two pixels are different colors.

{From canvas} which uses a total of 4 bytes per pixel, however our canvas will only have 16 colors due to the limitations of CMD, so we can use just 1 byte per pixel. We will however have to define how this byte is used to represent the 16 colors. A good convention would be to just use the colors Microsoft uses which range from 0 to 15, then we can also fit transparency in there, however we can only have one value of transparency so a simple method would be to consider the color transparent if it is greater than 15

### Camera

TODO: Explain the camera object and its methods

### Model and Texture

TODO: Explain model and texture loading with tinyobjloader and stbimage

### Renderer

TODO: Explain DrawModel function step by step

### Rasterizer

TODO: Explain RasterizeTriangle function

## Optimizations

TODO: still need to do implementation

## Demo

TODO: Make a demo program somehow, possibly a playable demo, like a walkable dungeon or forest scene, maybe add pictures here or a link to a video.

Sources

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

. ogl doc {TODO cite} https://registry.khronos.org/OpenGL-Refpages/gl4/html/glViewport.xhtml

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

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

4. {how to cite below} Murray JD, VanRyper W. Encyclopedia of graphics file formats. 2nd ed. 1996. Available: https://www.fileformat.info/format/wavefrontobj/egff.htm

5. De Vries J. Learn OpenGL – Coordinate Systems [Internet]. [Referenced 1.12.2024]. Available: <https://learnopengl.com/Getting-started/Coordinate-Systems>

6. {How to cite} http://www.et.byu.edu/~ered/ME537/Notes/ch1-537.pdf

7. {How to cite} https://www.scratchapixel.com/lessons/3d-basic-rendering/perspective-and-orthographic-projection-matrix/opengl-perspective-projection-matrix.html

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

9. Simo Määttä. 21.11.24. Personal Lecture.

{How to cite glm source