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](#_Toc184034602)

[2 Rendering theory 2](#_Toc184034603)

[2.1 Window 2](#_Toc184034604)

[2.2 Canvas 2](#_Toc184034605)

[2.3 Rasterization 2](#_Toc184034606)

[2.3.1 Rasterizing Lines 3](#_Toc184034607)

[2.3.2 Rasterizing Triangles 8](#_Toc184034608)

[2.4 Rendering 11](#_Toc184034609)

[2.4.1 3D Models 11](#_Toc184034610)

[2.4.2 Transforms 12](#_Toc184034611)

[2.4.3 Camera 12](#_Toc184034612)

[2.4.4 Perspective Projection 13](#_Toc184034613)

[2.4.5 Clipping 13](#_Toc184034614)

[2.4.6 Back Face Culling 13](#_Toc184034615)

[2.4.7 Depth Buffering 13](#_Toc184034616)

[2.5 Texturing 13](#_Toc184034617)

[2.6 The Console 14](#_Toc184034618)

[Sources 16](#_Toc184034619)

Symbols

texel

TODO

# Introduction

TODO

# Rendering theory

Drawing anything is quite math heavy, so here we will go through the process of rasterizing and rendering common objects, as well as explain the algorithms we will be using in a python-style pseudocode. We will also go over our chosen display method of the Windows console.

## Window

TODO

## Canvas

Before rendering anything to the console we will first need a canvas to draw things 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 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. A canvas also needs a PutPixel function which will change the color of a pixel at a specified x and y coordinate. With it we can already draw points. [1.]

canvas = Color[][]

PutPixel(x, y, color):

canvas[x][y] = color

## 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 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. [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)

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)

//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 \* dy

if error > dx:

x += xi

error -= 2 \* dx

### 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. 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*, 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:

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

https://learnopengl.com/Getting-started/Coordinate-Systems

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.

There are three main transforms we want to do: translation, rotation, and scaling. The proper method of applying these is with a transform matrix. Transformation matrices, however, are represented in homogenous coordinates, and all of out vertices are in cartesian coordinates.

### Camera

TODO

### Perspective Projection

TODO

### Clipping

TODO

### Back Face Culling

TODO

### Depth Buffering

TODO

## Texturing

Currently we can only draw shapes consisting of one color, for more detail we need to add a texture, which is an image we essentially “paint” on to the shape.

First, we need to define how the texture is applied to the shape. We will do this on a per-triangle basis by specifying a point on the texture which corresponds to each vertex of the triangle. For this we need a coordinate system to refer to the texels. 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. [1.]

## The Console

{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.

Sources

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

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. Murray JD, VanRyper W. Encyclopedia of graphics file formats. 2nd ed. 1996. Available: https://www.fileformat.info/format/wavefrontobj/egff.htm

5. Windows Console Documentation. Microsoft. 10.30.2020. Available: https://learn.mcrosoft.com/en-us/windows/console/