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](#_Toc167637615)

[2 Rendering theory 2](#_Toc167637616)

[2.1 Window 2](#_Toc167637617)

[2.2 The Console 2](#_Toc167637618)

[2.3 Canvas 3](#_Toc167637619)

[2.4 Rasterization 3](#_Toc167637620)

[2.4.1 Rasterizing Lines 4](#_Toc167637621)

[2.4.2 Rasterizing Triangles 10](#_Toc167637622)

[2.4.3 Texturing 13](#_Toc167637623)

[2.5 Drawing Sprites 13](#_Toc167637624)

[Sources 14](#_Toc167637625)

Symbols

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

## The Console

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 is capable of rendering 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. [3.] 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. [3.]

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 [3]. 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.

## 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.]

A simple canvas:

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 ray-tracing, 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. [3.] 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

{TODO: This section is way too long, I might simplify the bresenham explanation}

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. [2.] The function would look something like this, where x and y are integers:

DrawLine(x0, y0, x1, 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)

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 {}. 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. The bigger apparent 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. First, 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. 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 abs(slope) > 1. The second problem is also easily fixed by swapping the start and end points if they are not in proper order. [1.]

DrawLineY(x0, y0, x1, y1):

//Make sure starting point is before ending point

if y0 > y1:

swap(x0, x1)

swap(y0, y1)

//Slope is run/rise

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

x = x0

//For each y position, plot the corresponding x

for y from y0 to y1:

PutPixel(x, y)

x += m

Were we to combine these two functions into one we would have a working DrawLine function, but this 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 as is also optimized to only use integer arithmetic [4]. Its main downside is the lack of anti-aliasing, however we won't be able to use any of that anyways.

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. [4.]

DrawLine(x0, y0, x1, y1):

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

y = y0

accumulatedError = 0.0

//For each x position, plot the corresponding y

for x from x0 to x1:

PutPixel(x, y)

accumulatedError += m

if accumulatedError > 0.5:

y += 1

accumulatedError -= 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: "accumulatedError += m", and "if accumulatedError > 0.5".

First, accumulatedError += m can be rewritten as: accumulatedError = accumulatedError + dy / dx, where dx and dy are x0 - x1 and y0 - y1.

Then, to get rid of the fraction: dx \* accumulatedError = dx \* accumulatedError + dy

Next, to get rid of the fraction in if accumulatedError > 0.5 we can do: if accumulatedError \* 2 > 1

Now, we can avoid the \* 2 and \* dx by writing the two equations as: 2 \* dx \* accumulatedError = 2 \* dx \* accumulatedError + 2 \* dy, and if 2 \* dx \* accumulatedError > dx. This allows us to group the 2 \* dx \* accumulatedError into a single integer variable which we will call accumulatedErrorInt. [4.]

The optimized function:

DrawLine(x0, y0, x1, y1):

dx = x1 - x0

dy = y1 - y0

y = y0

accumulatedErrorInt = 0

//For each x position, plot the corresponding y

for x from x0 to x1:

PutPixel(x, y)

accumulatedErrorInt += 2 \* dy

if accumulatedErrorInt > dx:

y++

accumulatedErrorInt -= 2 \* dx

We will still need to apply all 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):

dx = x1 - x0

dy = y1 - y0

//Slope is < 1

if abs(dx) > abs(dy):

//If slope is positive increment y, else decrement

yi = 1

if dy < 0:

yi = -1

dy = -dy

//Make sure starting point is before ending point

if x0 > x1:

swap(x0, x1)

swap(y0, y1)

y = y0

accumulatedErrorInt = 0

//For each x position, plot the corresponding y

for x from x0 to x1:

PutPixel(x, y)

accumulatedErrorInt += 2 \* dy

if accumulatedErrorInt > dx:

y += yi

accumulatedErrorInt -= 2 \* dx

//Slope is >= 1

else:

//If slope is positive increment x, else decrement

xi = 1

if dx < 0:

xi = -1

dx = -dx

//Make sure starting point is before ending point

if y0 > y1:

swap(x0, x1)

swap(y0, y1)

x = x0

accumulatedErrorInt = 0

//For each y position, plot the corresponding x

for y from y0 to y1:

PutPixel(x, y)

accumulatedErrorInt += 2 \* dy

if accumulatedErrorInt > dx:

x += xi

accumulatedErrorInt -= 2 \* dx

### Rasterizing Triangles

The next geometric shape we want to draw is a triangle, since most 3D models are made exclusively of 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):

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 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 [v2, v0], 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 [v2, v0] will always be the continuous side, while [v0, v1] and [v1, v2] will make up the segmented side. [1.] Now, to get the x bound we will interpolate the x position of the line at each y:

InterpolateX(v0, v1):

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

x = x0

xPositions = []

xPositions.PushBack(x)

//For each y position, get the corresponding x

for y from y0 to y1:

x += m

xPositions.PushBack(x)

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. There will also be a duplicate 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.]

DrawTriangle(v0, v1, v2):

//Sort the vertices in descending y

if y1 < y0: swap(v1, v0)

if y2 < y0: swap(v2, v0)

if y2 < y1: swap(v2, v1)

//Calculate the x coordinate of every edge

topSegmentBounds = Interpolate(y0, x0, y1, x1)

bottomSegmentBounds = Interpolate(y1, x1, y2, x2)

continuousBounds = Interpolate(y0, x0, y2, x2)

//Combine the two lists of the segmented side

topSegmentBounds.PopBack()

segmentedBounds = topSegmentBounds + bottomSegmentBounds

//Check which is left and right

leftBounds = continuousBounds

rightBounds = segmentedBounds

i = round(continuousBounds.length / 2)

if leftBounds[i] > rightBounds[i]:

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)

This is actually quite a simple function and even though it is not a very optimized one, it will do for our purposes. Notice we also didn't use our DrawLine function here. That is due to our lines exclusively being horizontal, so we can make a more optimized implementation for this specific purpose.

### Texturing

TODO

## Drawing Sprites

TODO

Sources

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

2. Gambetta G. Computer Graphics from Scratch. 4.2021. Available: https://gabrielgambetta.com/computer-graphics-from-scratch/

3. 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/

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