

Fixed Function Configurable Graphics Pipeline

Purpose

This project seeks to design a simple, fixed-function, configurable graphics pipeline and program it to an FPGA. The aim is not to make an architecture based on highly parallel general compute cores as seen in modern graphics processing units, but instead to implement some of the functions that might have been available in early graphics accelerators. Such a design run on an FPGA should be able to perform rendering operations much faster than a program running on a CPU.

Background

We count the pixels of modern screens by the millions, and applications such as 3D rendering require many calculations per pixel. Fortunately, much of this work is highly parallelizable. Modern graphics processing units (GPUs) are highly sophisticated architectures, combining specialized hardware with thousands of parallel processors. These architectures are extremely powerful, but are out of reach for a student project to design.

Nonetheless, the basic theory and algorithms for how to convert vertex information into an image are relatively accessible. By carefully selecting just some essential and simple features, we can define a purely feed-forward pipeline that will be relatively easy to implement in hardware. The function of this pipeline is to take in a stream of vertex data, describing a 3D mesh, and output pixel data to be combined in an image.

1. Vertex input

The position values of the vertices of triangles we want to render will be read from a file on the host machine and fed into the pipeline on the FPGA. The vertex values are composed of (x, y, z) coordinates.

2. Transformation

`transform(objectSpace.xyz) -> cameraSpace.xyz`

This step will be configured via a separate command through the host-to-FPGA interface to store a 3x3 transformation matrix and translation vector. These two provide all necessary information for relevant 3D transformations up to and including stretching to fit a non-square viewport. Typically, 4x4 matrices are preferred for their more general applicability, but this optimization reduces the

computational cost (compared with a 4x4 matrix) from 16 multiplications and 12 additions to 9 multiplications and 9 additions.

3. Perspective Divide

`perspectiveDivide(cameraSpace.xyz) -> clipSpace.xyz`

This is the magic that allows for perspective projection. And all you have to do is divide by z. More explicitly,

`clipSpace.xy = - cameraSpace.xy / cameraSpace.z`

As a separate concern, we want to remap z in order to increase depth precision near to the camera.

`clipSpace.z = cameraSpace.z * a - b`

Where a and b are constants calculated based on the near and far clipping planes. As a result, z is mapped non-linearly so that the clipping bounds are 0 and 1.

4. Triangle Processing

`generateLines(clipSpace.xyz stream) -> clipSpace.xyz pair stream`

The input stream at the beginning will be in the form of triangles. Each triangle will generate up to three lines, and this stage has the simple job of feeding each pair of a triangle to the Line Rasterization stage.

Each line can here be clipped or culled. If no part of a line would appear within the clipping bounds, then the line does not need to be passed on to the line rasterization stage.


5. Line Rasterization

`drawLine(clipSpace.xyz, clipSpace.xyz) -> clipSpace.xyz stream`

There are many possible approaches. We will use XiaoLin Wu's¹ algorithm for its relative simplicity, superior performance, and anti-aliasing. Some multiplication is required for each line, but not for each pixel, making the algorithm especially efficient for long lines.

6. Fragment Processing

¹ Wu, X. (1991). An efficient antialiasing technique. *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques - SIGGRAPH 91*.



At first, fragments will be colored uniformly. Then per-vertex colors may be added. The colors would be blended along lines between vertices in the line drawing algorithm.

7. Frame Buffer

Combining the generated fragments into an image is an important step. We will take the simple approach of selecting only the fragment closest to the screen for each pixel. For the extent of this project, this work will be done on the host machine.

Test Plan

In the end, a few rendered images should be pretty good evidence that everything is working correctly. That leaves us with little for debugging when things do not work, however. There is a great benefit from the architecture being a feed-forward pipeline: each stage of the pipeline is a well-defined function of its input, and there are no complex state interactions to consider.

We will first write a fully-functioning version of the pipeline in C++ code. This will make it easier to iterate on and perfect each stage's algorithmic implementation. With this code as a reference, we can feed data into each stage of the hardware pipeline and know what output to expect. Any deviation in the output will be easily traced to the underlying error; simply start at the beginning and see which stage has the first incorrect output.

Microarchitecture and Host Program

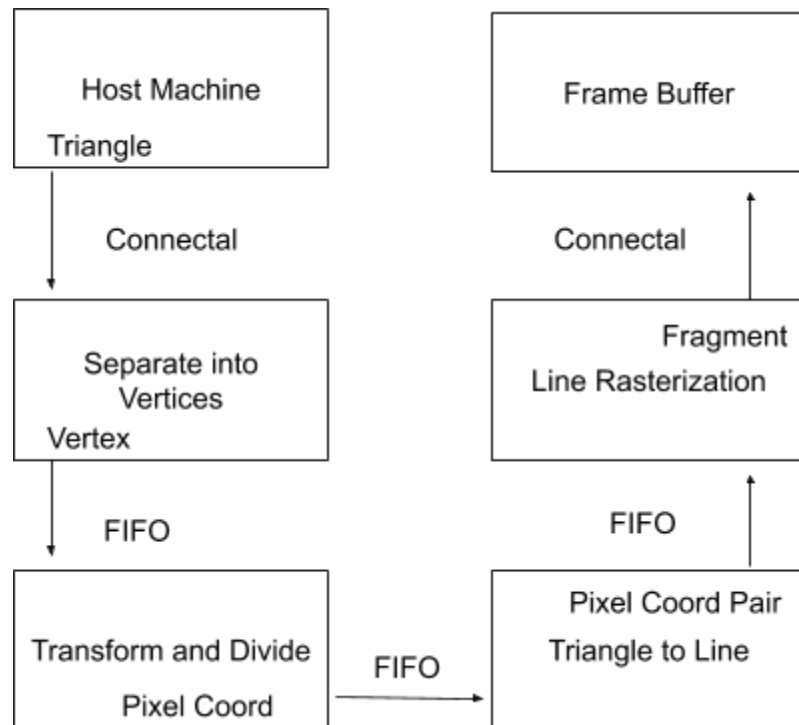
Host-FPGA Communication

The project consists primarily of Bluespec System Verilog code, which ultimately compiles to a format that programs the FPGA. Separately, we have C++ code, which is compiled and run on a linux host machine to generate the inputs to the FPGA pipeline, process the output, and run tests. We needed to have some way of facilitating the communication between the two ends of the system. This alone might have been a great deal of work, but we used a powerful build tool called Connectal². Using Connectal, we could simply declare interfaces between our hardware and software code, and everything that needed to happen in the middle to make it happen was generated automatically.

² <https://github.com/cambridgehackers/connectal>

Triangle Input

The first order of business for the C++ program was to generate the Transform (consisting of a 3x3 matrix and a vector) and send it to the FPGA. A small library of code for all of the relevant math was written. The Transform represents all of the transforms necessary to take a vertex position from object space (the position of a vertex relative to the object's own center) to camera space (the position of a vertex relative to the camera's location and orientation).



After the Transform is set on the FPGA, the C++ program takes care of sending a stream of vertex information to the FPGA. First, the information is loaded from an OBJ file. This yields an array of triangles, each three vertices. The floating point values are converted to a fixed point representation, which we used on the FPGA, and these values are sent over the Connectal interface.

Separating into Vertices

Instead of sending vertices to the FPGA one at a time, we sent them in groupings of three, representing a triangle each. The next steps of the pipeline, however, operate on individual vertices. In order to process one at a time, we used two vertex registers as buffers, with two corresponding boolean registers.

```

Vec3 bufferA
Vec3 bufferB
Bool validA
Bool validB
rule input_to_transform:
    if validA:
        transform_stage.push(bufferA)
        bufferA = bufferB
        validA = validB
        validB = False
    else:
        Triangle t = input.pop()
        transform_stage.push(t.a)
        bufferA = t.b
        bufferB = t.c
        validA = validB = True
  
```

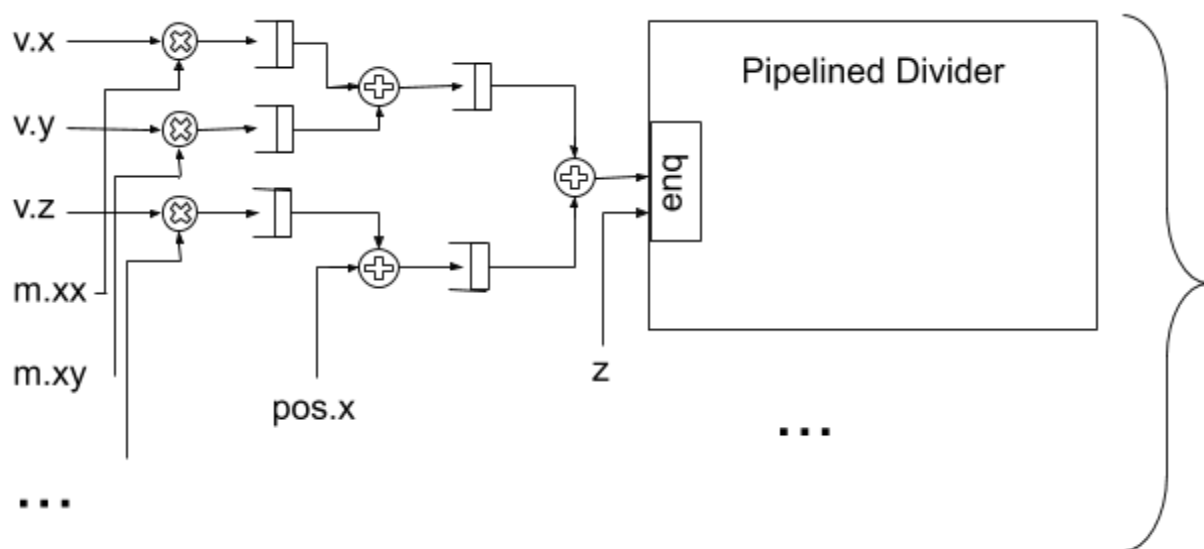
When a new triangle is dequeued from the FIFO, the first vertex goes straight to the Transformation stage, and the other two are stored in the two vertex registers, with their corresponding boolean registers set to True. On the next step, the first buffered vertex goes to the Transformation stage, and the second takes its place. Repeat for the third vertex. The cycle knows to repeat itself based on the values stored in the boolean registers. See pseudocode above.

Alternatively, if this stage were a bottleneck in the pipeline, we could have created three parallel instantiations of the Transformation stage.

Transformation and Perspective Divide

The Transformation stage itself consists of nine multiplications, nine additions, and three divisions. Even though many of these operations are in parallel, trying to do them all in one cycle had too long a critical path. In one build, the critical path delay was over 45 nanoseconds, well over our target of 20 nanoseconds. It was a fairly straightforward process to break the operation up into a pipeline, with each stage having a depth of one operation, either addition, multiplication, or multi-cycle division. Each of the nine multiplications are in the first cycle. Then, to add four items together, we add two pairs in one cycle, and add the two sums in the next. The result is enqueued to a pipelined divider.

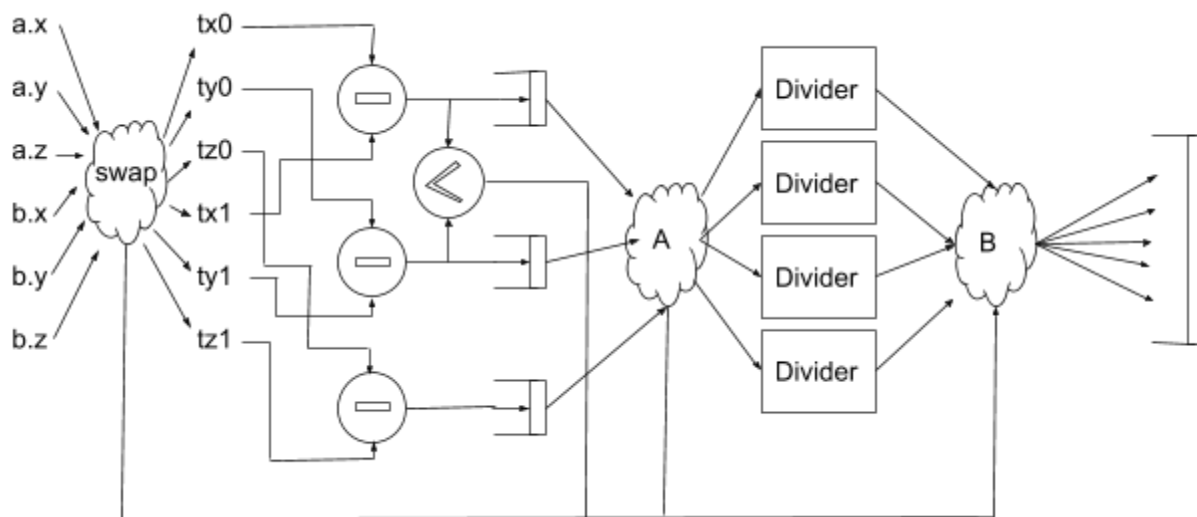
After pipelining this stage, the critical path was reduced from over 45 nanoseconds to under 13 nanoseconds.



Line Rasterization

A software implementation of XiaoLin Wu's line-drawing algorithm can be very short and simple, but designing an efficient implementation for FPGA required more code than any other part of this project. See the Appendix for a less rigorous introduction.

We can break the algorithm into two logical stages: setup, and iteration. The setup stage, without pipelining, has a much greater critical path. In our build, it had a path of approximately 43 nanoseconds, approximately the same as the unpipelined Transformation and Perspective Divide stage. On the other hand, the critical path of the iteration stage is dominated by a 12-bit addition. Without building it in isolation, it is not clear what the critical path length is, but it is at worst less than 13 nanoseconds.



A rough view of how the setup stage of the line drawing algorithm was pipelined

Fragment Processing and Frame Buffer

In this project, the pipeline returns a stream of pixel locations with intensities, and it is the responsibility of the host machine to compile that information into an image. This requires very little code, using a graphics or image library, but actually this step is very significant, computationally. We benefit from the CPU's caching. If instead we were to do a naive implementation on the FPGA, using its connected DRAM, our system would spend most of its time waiting for the memory. We would want to implement a caching system that would cache some portion of the framebuffer at a time. This would be a great project itself, and it would be very interesting to see how well different approaches work. How many line rasterizers could we have running in parallel? How to allocate and update framebuffer tiles to different rasterizers?

Remarks

If I were to choose one lesson learned from this project, I would say data locality. I am not sure if this is even the right term. It is not specifically regarding instructions/data being near each other in memory. Rather, it seems extremely important how the data is organized during computation. The closer it is to computation when you want to compute on it, the better. The less work you have to do to organize it, the better.

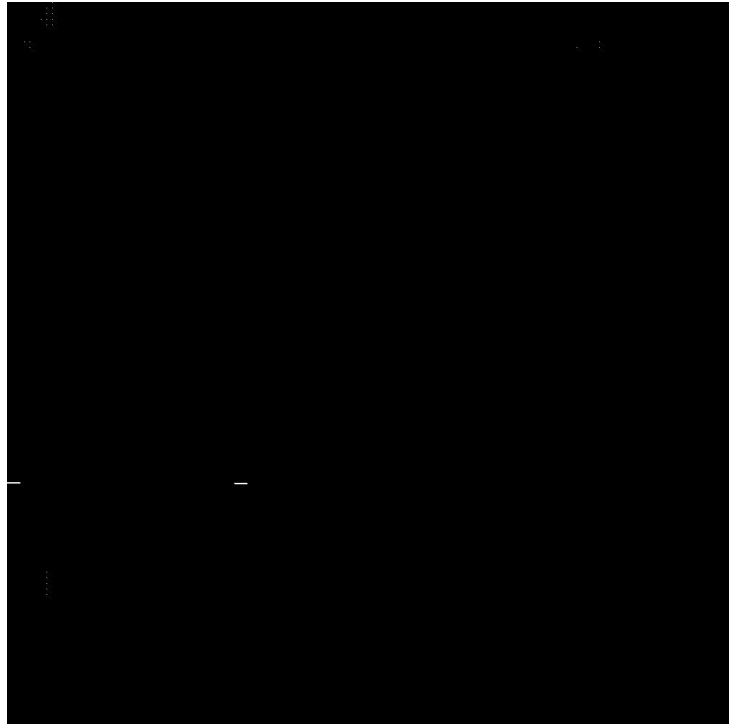
One apparent weakness to a pipeline like this is that the output data is very disorganized. You can compute fragments at a very high rate, but then you have to route each of those fragments to some location in the framebuffer. This can easily take more silicon and power than the more interesting computations! Just think of how much area of a modern CPU or GPU is consumed by its cache.

The pipeline currently has a throughput of one fragment/pixel per cycle. At around 66 MHz, this can achieve a 60 Hz frame rate on a 1024x1024 resolution display, even in near worst-case scenarios, writing one fragment per pixel. Of course, there is no limit to how bad a worst case can be - what if we tried to draw a trillion triangles?

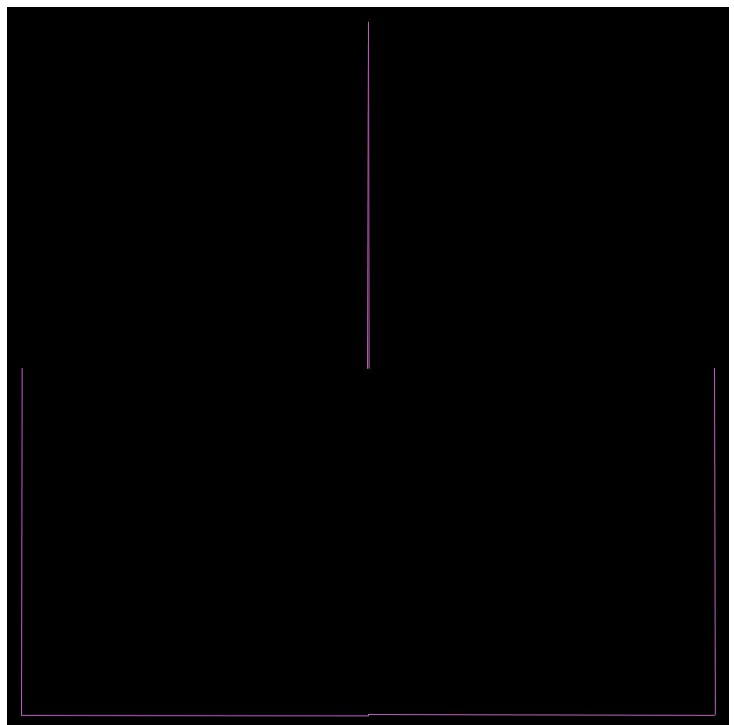
Appendix

Images

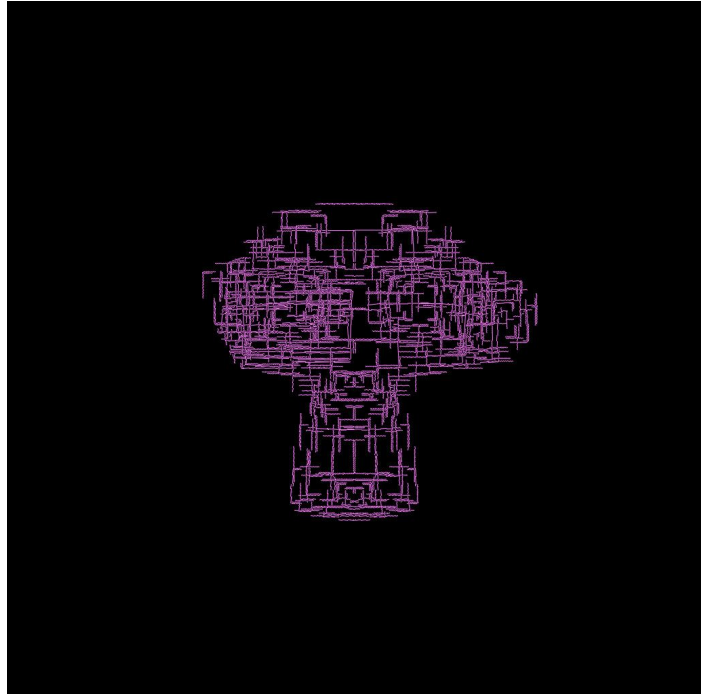
Right is the first image generated. It was supposed to be a triangle



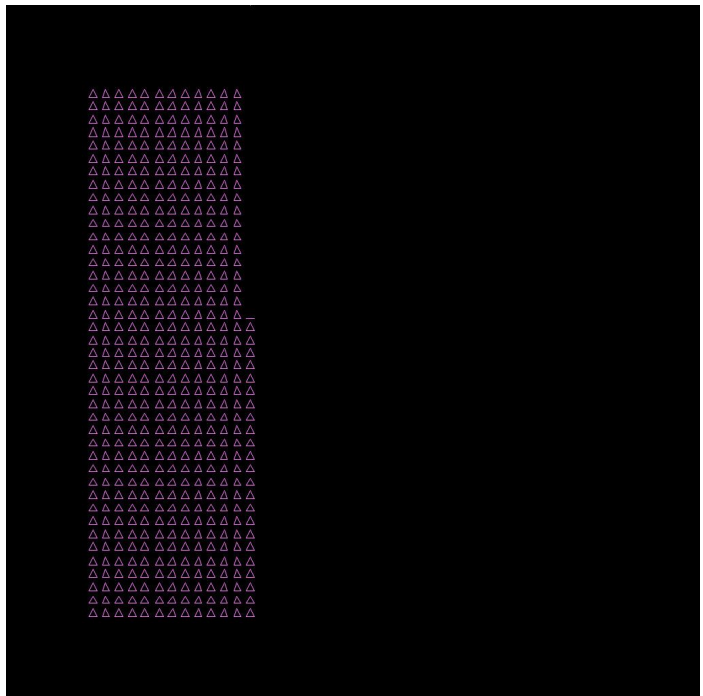
This was also supposed to be a triangle. The problem here was that the slope for the line-drawing algorithm was not being correctly calculated.



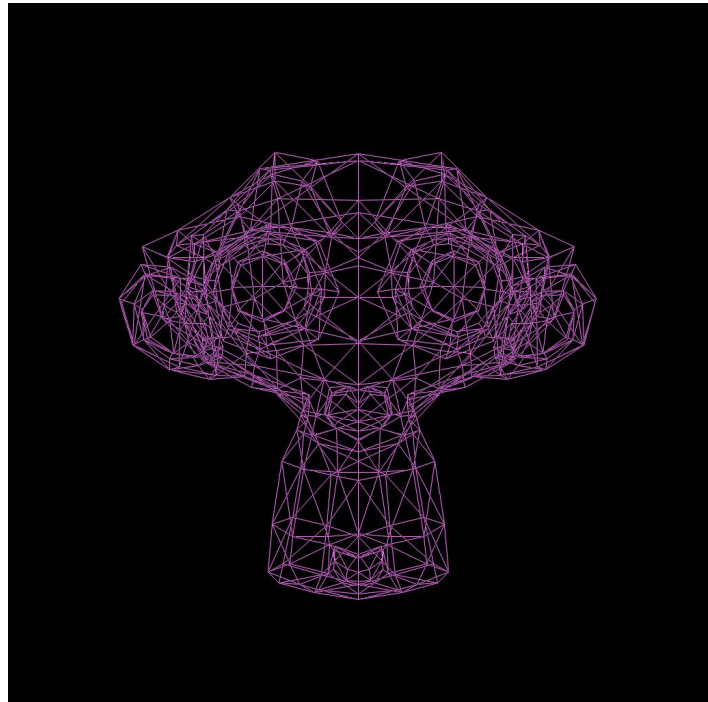
This was an attempted render of the Suzanne monkey head model, which ships with the free software Blender3D



Only after lots of debugging did we discover that the reason for many errors was simply that we did not give the pipeline enough time to finish. The solution was to implement a sort of handshake. The host machine signalled that it was done. Then the FPGA would signal when it was ready to be terminated. Only then would the host machine terminate the FPGA.



In the end, we were able to generate images successfully.



Multiplication

The Xilinx VC707 boards used for this project have a large number of Digital Signal Processing units, which can perform some common, complex functions, including multiplication. Using these "DSP48E1" slices should give significant improvements in both path delay and area, when compared to implementing the same functions in look-up tables.

Without requiring us to do anything special in our code, Xilinx's build tool was able to implement our multiplication with its DSP slices. In all, there were 10 DSP48E1 units used: nine in the Transformation stage, and one in the Line Rasterization stage.

Xiaolin Wu Algorithm

Line-drawing requires casting from two-dimensional continuous space to two-dimensional discrete space. A naive algorithm may operate something like this:

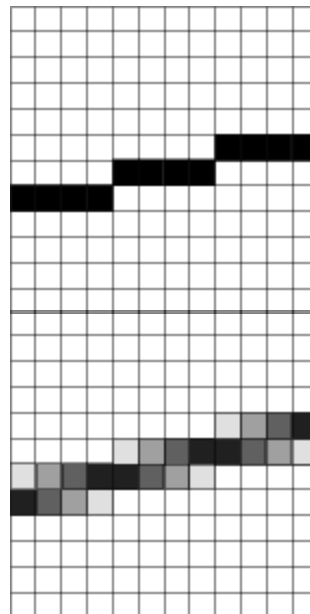
1. Calculate the slope and offset of the line
 - 1.1. $k = \frac{(y_b - y_a)}{(x_b - x_a)}$, $b = y_a$
2. For each x along the line, find y
 - 2.1. $\forall x_i \in \{x_a + 1, \dots, x_b - 1\}$, $y_i = \text{floor}((x_i - x_a)k + b)$

For most cases, this will work just fine. There comes some trouble, however, when $(x_b - x_a)$ approaches zero. Swapping the x and y values before performing the algorithm puts the zero in the numerator, and the algorithm works. Just remember to swap them back in the end! Further, we can easily do some similar swapping to transform any line to be in the first

octant. This simplifies the problem so that we only need to consider lines with a positive slope less than or equal to one.

There are two kinds of improvements we would like to make to the naive algorithm:

1. *Avoid having to do multiplication for each pixel.* A straight line (or line segment) is the epitome of linearity. It only makes sense that we should be able to traverse a line using only repeated addition.
2. *Perform some sort of antialiasing.* Our output space has finite resolution, but we can make lines less jagged. The examples to the right show the difference. Above is using the naive algorithm, with no antialiasing. Below is the expected output of XiaoLin Wu's algorithm.

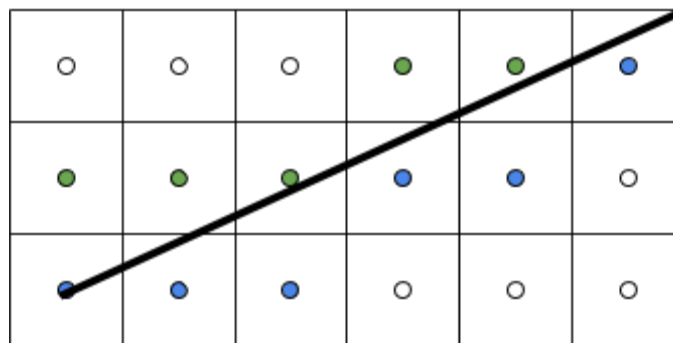


Xiaolin Wu's algorithm and other types of antialiasing create a softer line, which can have a rope-like effect

The main trick behind XiaoLin Wu's algorithm is in an n -bit unsigned integer, called \mathbf{D} . This integer represents the fractional distance of the line from the center of a pixel, measured on the scale $[0, 1)$, with the integer value $2^n - 1$ representing the maximum distance, $1 - 2^{-n}$.

Before most of the line-drawing can be done, we must calculate the value of \mathbf{d} , also an n-bit unsigned integer, representing the slope of the line. Recall that we have constrained all lines to a form where the slope fits in the range $[0, 1)$.

At the beginning of drawing a line, \mathbf{D} is initialized to the value 0, and the first pixel output (x_0, y_0) is already given as input to this function. For the next pixel, to the right of the first, we consider the column $x_0 + 1$ and must determine whether to draw at row y_0 or $y_0 + 1$. For this, we add \mathbf{d} to \mathbf{D} and see if the result has resulted in integer overflow. If there has been integer overflow, then we know that the line is now above the center of the row $y_0 + 1$, and we should use that as our new offset.



Computing \mathbf{d} requires some division and multiplication, but only once per line. This already makes the algorithm much faster than our naive one. XiaoLin Wu's algorithm further exploits some symmetry for even greater efficiency.

First, without requiring any additional computational work, the integer value \mathbf{D} can be used to determine the intensity of each pixel. For the current offset (shown with blue dots above), the intensity is proportional to the bitwise inverse of \mathbf{D} , and for the pixel immediately above it (shown in green), the intensity is proportional to \mathbf{D} . The original paper and article¹ do a great job explaining why.

Finally, the number of fragments generated per cycle is doubled by mirroring the line across its middle. Each step made from left to right is also made from right to left, in the opposite direction. The intensity values are shared across this symmetry.

In the end, the algorithm can produce four pixel fragments per cycle. There is little computational complexity beyond a single addition, so this step can be run at a fairly high frequency and only take up a small portion of the FPGA.