Contents

Introduction

For the EE271 class project you are going to work on a micropolygon rasterization unit. If you have no idea of what a micropolygon or rasterization is, don't panic (yet). We will explain both of these terms in the document, and in a review session we have scheduled for the class. Right now all you need to know is that this is one of the current trends in computer graphics. Unfortunately, this trend does not work well with a conventional graphics processor (GPU), so it is a good target for hardware acceleration.

We tried to construct this project in stages so that you will get to perform several kinds of tasks that you might be asked to do in industry. You will first start by finishing a C++ gold model for this design. Once this is completed, we hope you will have a better understanding of what the logic is supposed to do. Next, you will implement a missing module in RTL, synthesize it, and try to modify it to make it work better.

When you come to make RTL modifications, you will notice that the design is implemented using SystemVerilog. SystemVerilog's parameters and `generate` blocks enables designers to create design generators, or constructors, instead of design instances. We use this because we want you to parameterize your design, and procedurally encode your solution as a generator. This will help you explore more design alternatives, achieve better optimizations, and generally be a much more productive designer.

Finally, you will try a bunch of optimization choices in order to largely improve performance or save energy (or both). It should be fun. We hope you enjoy doing it as much as we enjoyed creating it.

This document will serve as an overview of the project. Please read the entire description first and only then start working.

Background

This section provides the preliminary information required to understand the project. First, we review a number of topics that you should already be familiar with to make sure we are all on the same page. Second, we present three computer graphics topics to help you understand the context of the project. Finally, a section on fixed point arithmetic is provided since it is an essential part of the project's implementation.

What You Should Know

The project will require some level of experience with SystemVerilog and C/C++. In this project, we will also introduce a new language called Genesis2, which is a meta-programming language for building chip generators. Here are some useful references on these topics:

- **SystemVerilog**:
    - Verilog Tutorial

- SystemVerilog Tutorial
    - SystemVerilog Assertions Tutorial
    - SystemVerilog DPI Tutorial
- Books
    - SystemVerilog For Design
    - SystemVerilog For Verification
        - Interfacing with C
        - Writing Testbenches using SystemVerilog

- **Genesis2**: Genesis2 is a meta-programming system. In this project, you will learn how to use Genesis2 to write a chip generator rather than a fixed logic design in plain Verilog. We use Genesis2 for everything related to the elaboration of the design. That includes design parameters, procedural Verilog code generation, introspection and more. In particular, all RTL and testbench code for this project is written using Genesis2.

  Genesis2 provides hardware designers with a rich software language for writing instructions that specify how to generate modules from a set of input parameters. These instructions can be seen as an explicit "elaboration program" or as an object-oriented constructor for generating elaborated instances. The behavioral description, however, remains in SystemVerilog. Granted, in software coding, the semantics of coding a constructor for a class is the same as the semantics of coding the rest of that class's functionality. In contrast, the description of the functionality of a hardware module must obey strict rules of synthesizability. This historically resulted in HDLs that enforced strict rules on the construction of the system, also known as its elaboration. With Genesis2, we remove this artificial limitation, by allowing the designer to code in two languages simultaneously and interleaved: one that describes the hardware proper (SystemVerilog), and one that decides what hardware to use for a given instance (Perl). However, Genesis2 maintains the notion of modules, hierarchy and system, by forcing the two language layers to share the same scopes. When you look at our code, or write your own, lines that start with `//;` are Perl lines and use strictly Perl syntax. Lines that have ‘expression‘ (an expression between two back-ticks) is Verilog with an inlined Perl expression. At the meta-layer, Genesis2 provides a number of built-in parametrization, generation and introspection methods. This is described in the documentation wiki page at the link below.

    - Genesis2 User Guide
    - (DAC2012 paper) Avoiding game over: bringing design to the next level

- **Perl**: In Genesis2, we use Perl to manipulate the generation of SystemVerilog code. Therefore you will need to be able to read and manipulate some lightweight Perl code. Here are some references that will help you if you are not already familiar with Perl. **Note: One can write really perlish (un-readable) code in Perl. That type of code is more concise**

but really bad for collaborative projects. Make sure your code is readable and well commented.

  - Perl online documentation: Perldoc
  - Introduction to Perl: Essential Perl

- **C/C++**: C++ code will be used to implement the golden model in assignment one. C++ is very similar to C. Some of the major features implemented in C++ are:
  - Pass By Reference
    * A Word on References
  - Standard Template Library (STL)
    * An Introduction
  - Templates
    * An Explanation

The Modern Real-Time Computer Graphics Pipeline

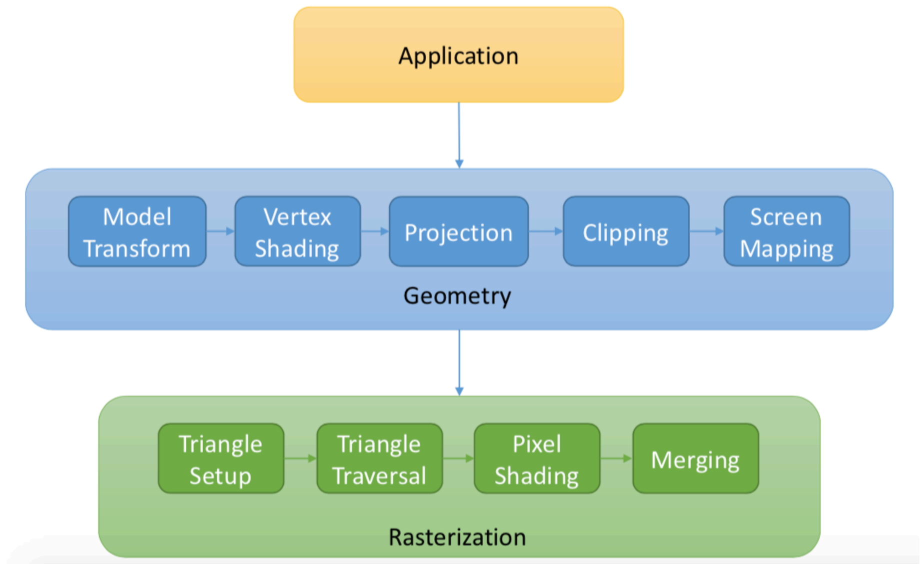

Figure 1: Computer Graphics Pipeline

Figure 1: Computer Graphics Pipeline

Computer graphics can be described as the problem of converting 1's and O's into a two dimensional array of pixels. This covers every image you see on a computer display, including font rendering, flight simulators, interface design, CAD, computer generated movies, and video games.

The real-time computer graphics problem adds an additional constraint to the general computer graphics problem. The real time constraint is that the processing of data visualization be fast enough to be imperceptible to the user. In most cases, this means sustaining a frame rate - the speed that the image on the screen is replaced - of 60Hz. In graphics intensive applications like flight simulators, architectural simulators, CAD, and video games, this often means performing significant amounts of computation very quickly.

While some applications can afford to do most of the graphics work entirely on the central processing unit (CPU), most high-end applications require specialized hardware in the form of a graphics processing unit (GPU).

To achieve high speed, graphics operations are broken into different pipeline stages. At the highest level of abstraction, the real-time graphics pipeline, shown in Figure 1, can be broken into three parts: application, geometry, and rasterization.

For this discussion, we can assume that the application portion takes the objects that need to be drawn on the screen and generates their positions and geometries in the 3-D space. The resulting data could consist of basic shapes (easy for hardware to render), like spheres, cones, or arbitrary 3-D objects; in general, however, it is best thought of as surfaces in 3-D space each represented by meshes composed of triangles (and quadrilaterals - polygons in general). An example is shown in Figure 2, where the upper left and lower right quadrant show a wire mesh demonstrating the geometric description of the bird.

Figure 2: Bird Mesh (Blender Foundation)

Next, the geometry portion of the pipeline applies many kinds of transformations to these mesh data (3-D vertices of every triangle) and generates their proper positions on the screen (2-D space). Finally, the rasterization portion converts the 2-D vector data of triangles to raster format, corresponding to pixels on the screen and uses it to fill in an image.

Geometry Pipeline

While this project does not directly touch the geometry pipeline, this section will give you a brief background, to provide context to the later discussion about rasterization. The geometry pipeline can be broken into five parts:

- Model Transform
- Vertex Shading
- Projection
- Clipping
- Screen Mapping

Model Transform: The data provided by the application may not explicitly specify the location and scale of each triangle and quadrilateral in the space,

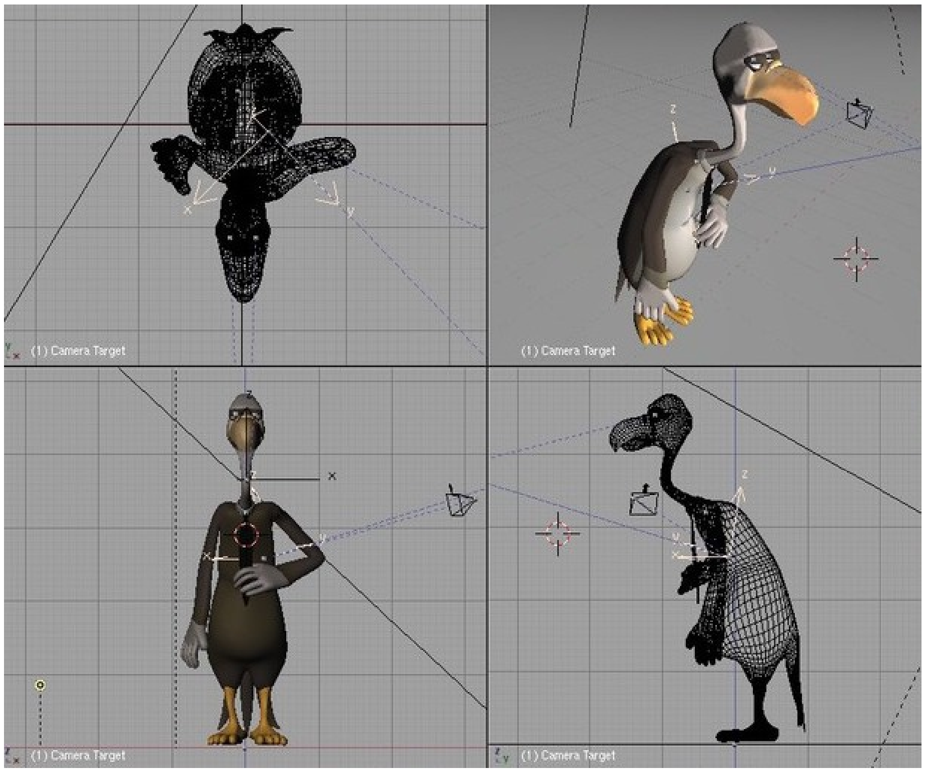

Figure 2: Bird Mesh

which is the space of the scene being drawn. For example, a squadron of C-130 aircraft may be defined compactly in terms of one 1:16 C-130 model. The application might provide one 1:16 C-130 model, the position, and the orientation of each plane. The model would then need to be instanced 5 times, scaled 16x to match the environment of the simulation, and rotated appropriately. The model transform block of the pipeline takes all of the relative definitions and transforms them into triangles and quadrilaterals that appear in the world.

Vertex Shading: Vertex shading refers to altering the data associated with the vertices that define the triangles and quadrilaterals in a scene. For example the vertex shader may want to alter the color of the vertices, or even displace them to generate some effect like waves on the ocean.

Projection: The projection portion of the pipeline performs an additional transformation on the scene data. While the geometric data defined in world space contains information about the absolute location of the vertices, we want to know what to draw on a 2-D screen. We can determine this information by placing a virtual camera at where our eye should be. The projection portion transforms all of the data such that things far away from the camera will appear smaller and things close-up will appear larger. The projection portion also does the work of rejecting data that doesn't fit inside the view space of the camera, for example something behind the camera. A more formal explanation for the work done in the projection portion would be that it converts all of the geometry located in the viewing frustum and transforms it such that the frustum becomes a unit cube. A view frustum is shown in Figure 3. The geometry produced by this portion of the pipeline is therefore defined inside a cube with sides of length 1.

Figure 3: View Frustum (Wikipedia)

Screen Mapping: The screen mapping portion of the pipeline takes this view space data and maps it to the pixels in the screen. This would mean transforming the cube and the geometry inside it such that one face of the cube matches the size of the image to be colored. Once the geometric data has been mapped to the screen it can be colored.

Rasterization

Rasterization is the essence of this class project. The purpose of this section is to give you the "what?" and "why?" (the "how?" is discussed later). The rasterization pipeline can be broken into four parts:

- Triangle Setup
- Triangle Traversal
- Pixel Shading
- Merging

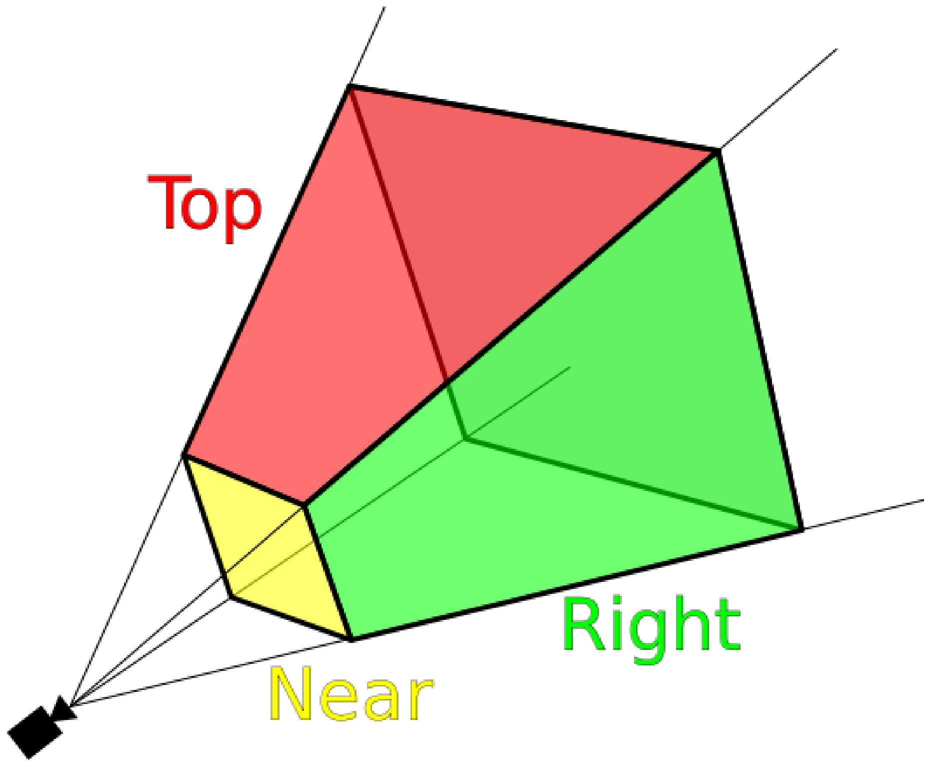

Figure 3: View Frustum

Triangle Setup: The triangle setup stage encompasses all of the work done to prepare for traversing the polygon. This could include calculating the slopes of edges or whether the polygon is occluded.

Triangle Traversal: The triangle traversal stage determines which pixels lay inside a polygon. In old pipelines this was done with scan algorithms that would traverse the polygon from left to right and top to bottom, testing each pixel to see whether or not it lies inside a polygon. In modern pipeline the work is done hierarchically using a few tests to determine whether or not large numbers of pixels lie inside a polygon.

A trivial algorithm for triangles would test four non-adjacent pixels. If all four pixels fell inside the triangle then all pixels bounded by those four pixels are inside the triangle. These types of hierarchical tests take advantage of pixel coherence, the idea that neighboring pixels are likely to lie in the same polygon. However, as computer graphics advance, triangles become smaller and smaller (and hence the term 'micro-polygons'). This means that neither the old scan nor the hierarchical algorithm is efficient. This is where you come into play (you'll see how next).

Pixel Shading: Pixel shading is the stage of the pipeline where a pixel is given a color based on the polygon it falls into. In the case of flat shading, where a polygon has a solid color, the pixel color is simply assigned the polygon's color. In more complicated forms of shading, lighting calculations are used to determine the color based on the orientation of the polygon with respect to all of the light sources in the scene. It is also possible to texture a polygon. Texturing a polygon refers to the application of an image to the surface of a polygon. Texturing would allow you to make a brick wall by simply generating a quadrilateral and then applying the picture of the wall to the quadrilateral. An example of a textured mesh is given in Figure 4. The image on the left shows the application of a checkered pattern to the mesh while the right image shows a two-dimensional image with an outline of the polygons in the mesh. One could color this two dimensional image of the alien goat skin and then apply it to the alien goat.

Figure 4: Alien Goat Texturing (Blender Foundation)

Merging: Finally, these pixels need to be aggregated to form the final image. In the process of mapping many triangles to the screen, however, there is a possibility that we may have mapped an occluded triangle to the same pixel as the occluding triangle. During merging, if the occluding triangle is defined as opaque, the occluded triangle's pixel is rejected; otherwise, the occluded triangle's pixel is mixed with the occluding triangle. The image is complete after all of the pixels have been aggregated.

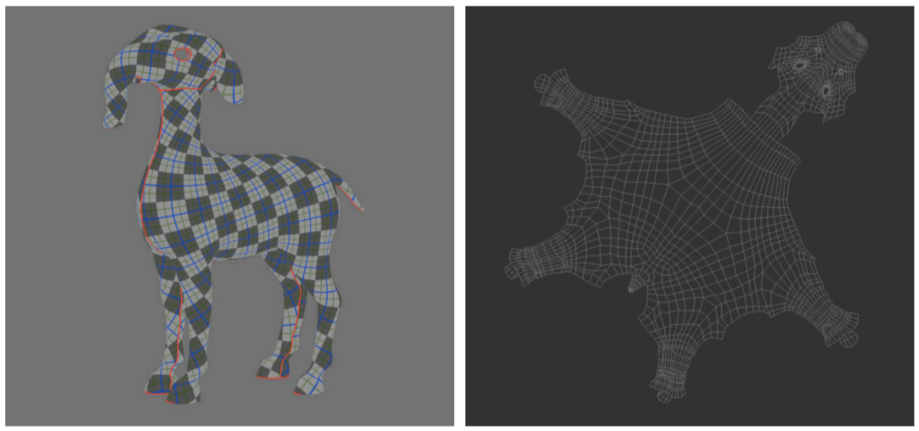

Figure 4: Alien Goat Texturing

More Information

- Real Time Rendering is a useful text that describes the computer graphics pipeline and many modern rendering techniques. Their blog can also be very informative. Stanford students can view this text on-line through the library here.
- OpenGL Programming Guide is a useful reference for the OpenGL interface. Its introduction contains a summary of the OpenGL rendering pipeline. This book is available to all Stanford students free at O'Reilly's Safari.
- Computer Graphics and Geometric Modeling contains a more in-depth discussion of the components of the computer graphics pipeline.

Multisample Anti-Aliasing

As many will recall from discussions in signal processing, sampling a signal below the Nyquist sampling rate will introduce aliasing into the sampled signal. In computer graphics, aliasing is observed in the artifacts that occur along sloped edges in images called jaggies. The character "A" on the left of Figure 5 shows an example of aliasing. Additionally, in an animated scene, aliasing will cause polygons to look like they are stuttering across the scene as they jump from pixel to pixel. Jaggies and stuttering are undesirable and decrease the overall quality of an image and animation.

A solution to this problem is to sample the image at a higher rate to reduce the aliasing error. In computer graphics, Full Screen Anti-Aliasing (FSAA) can be thought of as generating a high resolution image and then averaging it down to a lower resolution picture. The pixels in the high resolution image are referred to as fragments which are averaged into the pixels in our final image. The character "A" on the right of Figure 5 shows an example of anti-aliasing. Multi-sample Anti-Aliasing (MSAA) is similar to FSAA, but requires less storage and computation to generate the final image. Our micropolygon rasterization pipeline implements varying degrees of MSAA, so you should be comfortable with the idea that we will rasterize fragments which will later be averaged into a pixel.

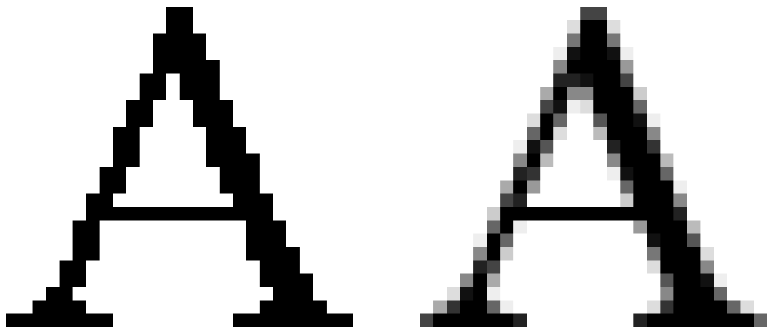

Figure 5: Aliasing of A

Figure 5: Aliasing of A (Wikipedia)

Micropolygon Rendering

One concern in computer graphics is the generation of highly realistic and highly stylized images. One problem with modern pipelines is that while many tricks can be played, when an image is composed of too few polygons it can take on an undesirably faceted appearance. Early video games and CAD applications are memorable for their flat and faceted appearance.

A solution proposed long ago by PIXAR, and used in films like Wall-E and Toy Story, is the use of micropolygons. A micropolygon avoids the problem of faceting by occupying a very small area - the area of half a pixel. The algorithm was referred to as the Reyes Image Rendering Architecture. The part of the pipeline that we are concerned with in this project is the hide stage. We will use hide and micropolygon rasterization interchangeably. Before the hide stage the micropolygons have already been positioned in screen space and shaded, as opposed to modern pipelines where fragments are shaded. Then, during the hide stage, the micropolygons are sampled to determine the color of the fragments that correspond to the micropolygons. Generally, the hide stage can be broken into two steps: determining which fragments should be tested against a micropolygon and testing those fragments against the micropolygon to determine if they lie inside.

While Pixar hoped to achieve memorable movies, the goal in our project is to achieve Toy Story like rendering in real-time using the specialized hardware. The hardware design in this project is based on the software implementation presented in this document. This hardware represents a variation on the design and concepts presented in this paper. The major difference between micropolygon rasterization and modern rasterization is that hierarchical approaches are no longer efficient, as it is difficult to find large numbers of fragments that are trivially inside a small polygon.

Fixed Point Arithmetic

With your experience in computer programming you have already encountered two traditional data types used to represent numbers. These data types are integer and floating point numbers. The interesting tradeoff from the perspective of micropolygon rasterization is that integer computation is fast while floating point numbers represent fractional numbers well. Unfortunately, floating point computation is more complex which makes it high power and slow when compared to integers. As a compromise, we could pick some constant position in the integer to represent the decimal. In this way everything to the left of this decimal is the integer portion and everything to the right is the fractional portion. This representation is referred to as fixed point notation.

Fixed point operations are generally the same as integer operations. Comparisons, additions, and subtractions are all implemented in the same manner as traditional integer operations. The one exception is multiplication which requires shifting the result to the correct position, as you will recall from your childhood introduction to the multiplication of decimal numbers.

The implementation of the micropolygon hider that you will be working with utilizes a fixed point representation and fixed point operations. You should be comfortable with the idea that the values represented in the RTL may have a data type of integer but actually represent a fixed point notation.

Micropolygon Rasterization Overview

Recall that the goal of rasterization is to determine which fragments correspond to which micropolygons in an image. To make the problem easier, we have broken it into two steps: first determine which fragments we should test against the micropolygon and second determine if the fragments lie inside the micropolygon. We will refer to the first as the bounding box problem and the second as the sample test problem.

Bounding Box

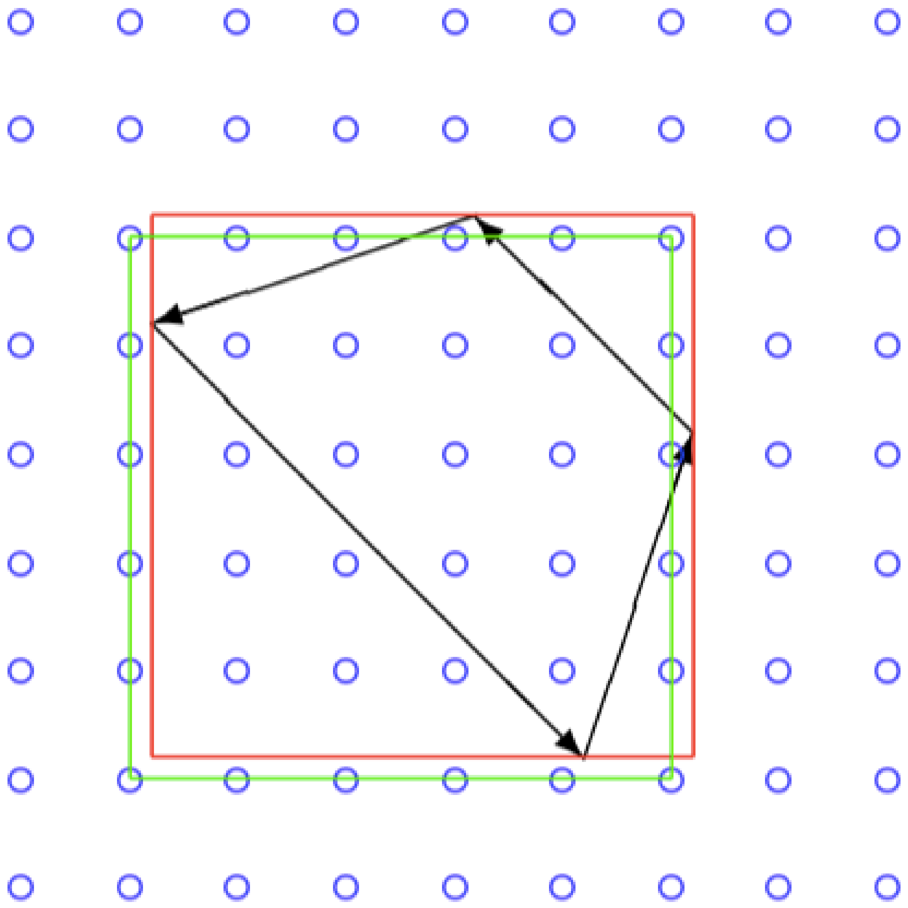

Figure 6: Axis Aligned Bounding Box 16xMSAA

Figure 6: Axis Aligned Bounding Box 16xMSAA

The naive solution for determining the set of samples (fragments) to test is the set of all samples on the screen. This is extremely inefficient and would require a significant number of sample tests. To reduce the number of sample tests we can calculate an axis aligned bounding box for the microploygon which would limit the number of samples we are required to test. A bounding box can be calculated using the minimum x coordinate of the vertices, the minimum y coordinate of the vertices, the maximum x coordinated of the vertices, and the maximum y coordinate of the vertices. The axis aligned bounding box is shown in Figure 6 in red while the micropolygon is shown in black and the fragment sample locations are shown as blue circles.

While this axis aligned bounding box is a boundary for both the micropolygon and the set of fragments that exist inside it, it does not describe the set of fragments to test. To do this we would like to round the coordinates to nearby fragments, so that the sample set can be described as iterating from the lower left coordinate to the upper right coordinate in a left to right and down to up pattern. Because we will be sampling the region with jittered sampling, we will need to make sure that our clamped bounding box includes all of pixels whose randomly distributed samples could lie inside the micropolygon. This optimally clamped bounding box is shown in green in Figure 6.

Finally, it is desirable to clip the bounding box to the screen space so that fragments which lie outside the image are not tested against the polygons. The clip operation simply replaces one of the lower-left coordinate value with 0 if it is less then zero, or one of the upper right coordinates with the image width/height if its value is out of bounds. Finally, in the cases where the bounding box does not appear on the screen at all, it is useful to reject the micropolygon entirely.

Sample in Polygon Test

The goal for a sample test is to convert the geometric properties of a point in a polygon into an equation that can be evaluated as true or false. The approach we take is to use edge equations. Edge equations can be used to indicate whether a sample lies to the left, right, or on a directed edge.

In the simple case of a triangle, when the sample lies to the same side of all edges, the sample can be considered to lie inside the micropolygon.

- A micropolygon triangle is defined by three vertices `(v1, v2, v3)` where `v1 = (x1, y1)` and so on.
- Edges are defined by two vertices, for example `(v1, v2)` is an edge from `v1` to `v2`.
- An equation for the edge can be given in the form: `(y - y1)/(x - x1) = (y2 - y1)/(x2 - x1) = slope` or `(y - y1)(x2 - x1) = (y2 - y1)(x - x1)`
- Given a sample position `(x, y)`, it is useful to make a coordinate shift such that the sample resides on the `(0, 0)` position. This is a shift of `-x` for X coordinates and `-y` for Y coordinates. Define: `xl' = xl - x   yl' = yl - y   x2' = x2 - x   y2' = y2 - y` Now we test our `(x, y)` point by placing it in the equation for the edge described above: `0 = x1'y2' - x2'y1'`
- If the origin is not on the line: `0 != x1'y2' - x2'y1'`
- If the origin lies to the left of the line: `0 < x1'y2' - x2'y1'`
- If the origin lies to the right of the line: `0 > x1'y2' - x2'y1'`

- For example,
  - `v1 = (1, 0)`, `v2 = (0, 1)`, `vs = (0, 0)` (Figure 7):
    - `1 · 1 - 0 · 0 > 0` therefore left of line.

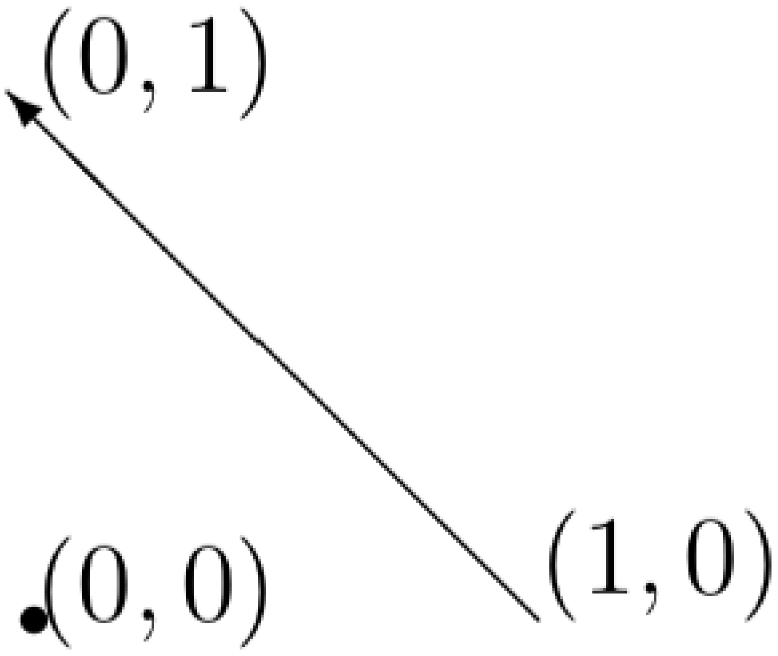

Figure 7: Edge Test 1 - Left of Edge

Figure 7: Edge Test 1 - Left of Edge

  - `v1 = (0, 1)`, `v2 = (1, 0)`, `vs = (0, 0)` (Figure 8):
    - `0 . 0 - 1 . 1 < 0` therefore right of line.

  Figure 8: Edge Test 2 - Right of Edge

  - Given a triangle `v1 = (-2, 3), v2 = (-1, -1), v3 = (2, -1)` and sample `vs = (0, 0)` (Figure 9):
    - e1: `-2 · -1 - -1 · 3 =  5 > 0` therefore left of line
    - e2: `-1 · -1 - 2 · -1 = 3 > 0` therefore left of line
    - e3: `2 · 3 - -2 · -1 = 4 > 0` therefore left of line

  Therefore the point must lie inside the triangle.

  Figure 9: Edge Test 3 - Point Inside

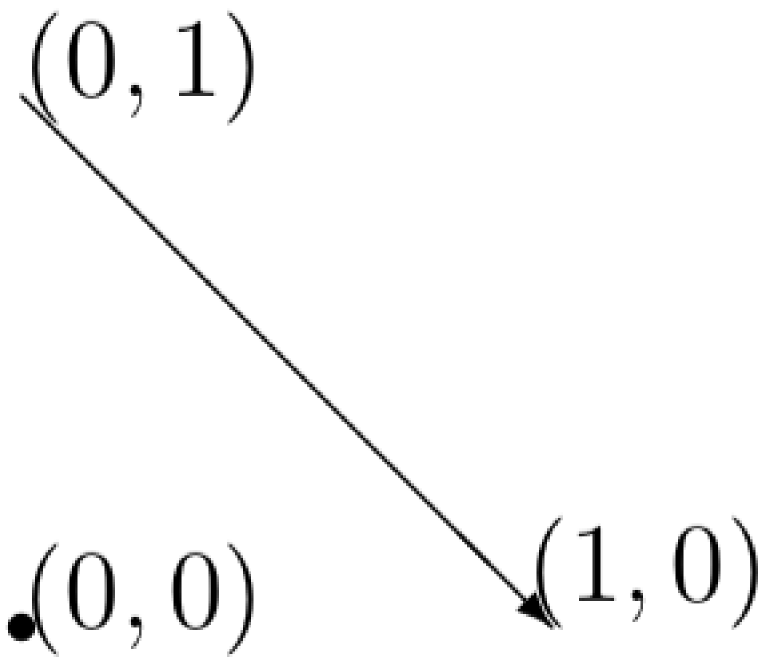

Figure 8: Edge Test 2 - Right of Edge

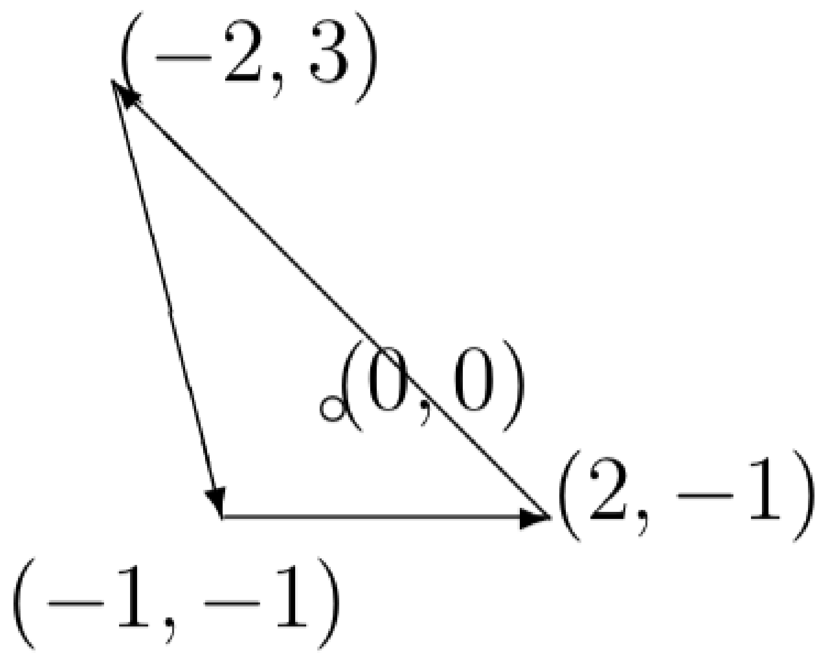

Figure 9: Edge Test 3 - Point Inside

- Given a triangle `v1 = (-2, 3), v2 = (2, -1), v3 = (-1, -1)` and sample `vs = (0, 0)` (Figure 10):
  - e1: `-2 . -1 - 2 . 3 = -4 < 0` therefore right of line
  - e2: `2 . -1 - -1 . -1 = -3 < 0` therefore right of line
  - e3: `-1 . 3 - -2 . -1 = -5 < 0` therefore right of line

  Therefore the point must lie inside the triangle.

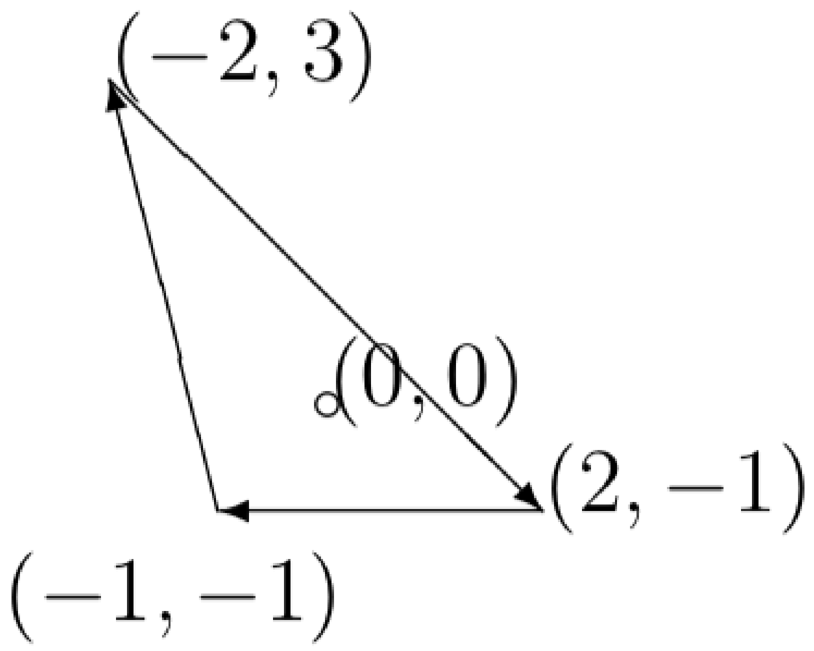

Figure 10: Edge Test 4 - Point Inside

Figure 10: Edge Test 4 - Point Inside

- Given a triangle `v1 = (-2, 3), v2 = (-1, -1), v3 = (0, -1)` and sample `vs = (0, 0)` (Figure 11):
  - e1: `-2 . -1 - -1 . 3 = 5 > 0` therefore left of line
  - e2: `-1  . -1  -  2 . -1 = 3 > 0` therefore left of line
  - e3: `0 . 3 - -2 . -1 = -2 < 0` therefore right of line

  Therefore the point must lie outside the triangle.

  Figure 11: Edge Test 4 - Point Outside

In many graphics systems, two dimensional polygons are considered to have a facing. This means that viewed from one side a polygon is visible, and from

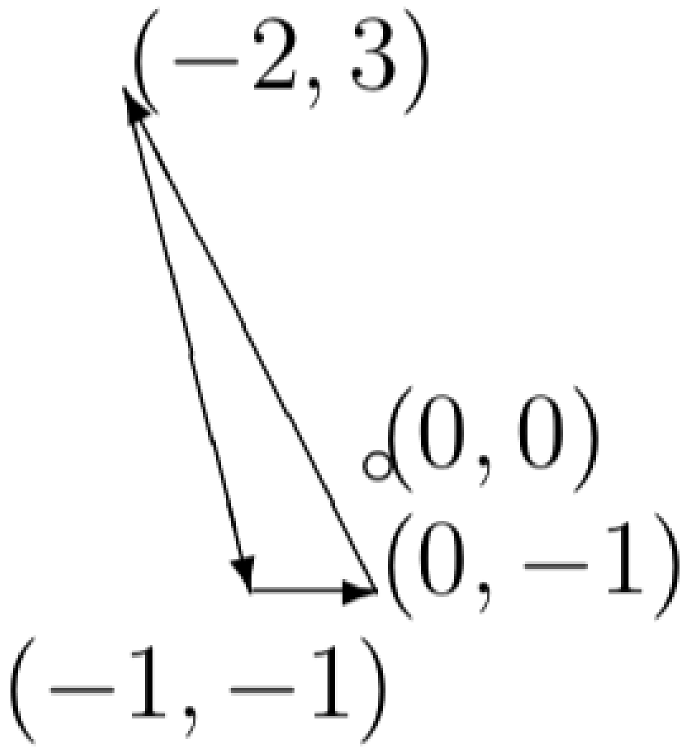

Figure 11: Edge Test 4 - Point Outside

the other it is not. This is useful in reducing the amount of work required, as back-facing, or looking away from the display, polygons can be removed. In our case, when the sample point lies to the right of all of the edges then the sample lies inside a forward facing micropolygon, otherwise the sample test fails. This is referred to as back-face culling and corresponds to accepting the sample point in Figure 10 and rejecting the sample point in Figure 9.

Pseudo Code

Pseudo code for micropolygon rasterization with triangles only:

```
void rast(vector<u_Poly> polys){
    for(i = 0; i < polys.size(); i++){
        rast_uPoly(polys[i]);
    }
}

void rast_uPoly(poly){

    // Calculate clamped bounding box
    // ll = lower left, ur = upper right

    // Min x rounded down to subsample grid
    ll_x = FLOOR_SS(MIN(x coordinate of poly vertices));
    // Max x rounded down to subsample grid
    ur_x = FLOOR_SS(MAX(x coordinate of poly vertices));
    // Min y rounded down to subsample grid
    ll_y = FLOOR_SS(MIN(y coordinate of poly vertices));
    // Max y rounded down to subsample grid
    ur_y = FLOOR_SS(MAX(y coordinate of poly vertices));

    // Clip bounding box to visible screen space

    ur_x = ur_x > screen_width ? screen_width : ur_x;
    ur_y = ur_y > screen_height ? screen_height : ur_y;
    ll_x = ll_x < 0 ? 0 : ll_x;
    ll_y = ll_y < 0 ? 0 : ll_y;

    // Iterate over samples, test if in micropolygon
   // Note that offscreen bounding boxes are rejected by for loop test

    for(sl_x = ll_x; sl_x <= ur_x; sl_x += subsample_width){
        for (sl_y = ll_y; sl_y <= ur_y; sl_y += subsample_width){
         [j_x, j_y] =  jitter(s_x, s_y); // Compute noise for sample
            [s_x, s_y] =  [sl_x, sl_y] + [j_x, j_y]; // Add noise
            if(sample_test(poly, s_x, s_y)){
                process_fragment(poly, s_x, s_y);}}}}

int sample_test(poly, s_x, s_y){

    // Shift vertices such that sample is origin
    v0_x = poly.v[0].x - s_x;
    v0_y = poly.v[0].y - s_y;
    v1_x = poly.v[1].x - s_x;
    v1_y = poly.v[1].y - s_y;
    v2_x = poly.v[2].x - s_x;
    v2_y = poly.v[2].y - s_y;

    // Distance of origin shifted edge
    dist0 = v0_x * v1_y - v1_x * v0_y; // 0-1 edge
    dist1 = v1_x * v2_y - v2_x * v1_y; // 1-2 edge
    dist2 = v2_x * v0_y - v0_x * v2_y; // 2-0 edge

    // Test if origin is on right side of shifted edge
    b0 = dist0 <= 0.0;
    b1 = dist1 <  0.0;
    b2 = dist2 <= 0.0;

    // Triangle min terms with no culling
    //triRes = (b0 && b1 && b2) || (!b0 && !b1 && !b2);

    // Triangle min terms with backface culling
    triRes = b0 && bl && b2;

    return(triRes);
}
```

Hardware Overview

Figure 12 is an overview of the hardware. With respect to the pseudo code:

- BBox stage corresponds to the generation of a clamped axis aligned bounding box (implemented in `rtl/bbox.sv`).
- Iterator stage is a finite state machine (FSM) correcponding to the set of nested for loops which iterate all the sub-pixel positions in the bounding box (`rtl/test_iterator.sv`).
- Hash box corresponds to the jitter function to add random noise to the sample position (`rtl/hash_jtree.sv`).
- Sample Test box corresponds to the sample test function inside the nested

for loops (`rtl/sampletest.sv`).

Figure 12: Hardware Pipeline

In addition, Figure 12 also implies the default timing for this pipeline: there are three pipeline stages in BBox, one stage in Iterator, two in Hash, and two in Sample Test. The good news is that all the pipeline depths of these blocks (except the Iterator) are coded as SystemVerilog parameters, so you can easily change the numbers when optimizing the design. The the depth of Iterator is fixed at one since it is an FSM. Besides, the signals in this figure (the gray dotted lines) are labeled with a pipe stage number. As a convention, the data enters the raster block at pipe-stage 10.

More details of the function blocks will be discussed in the following parts of this section.

`rtl/bbox.sv`

- **Inputs:**
  - 3 or 4 x, y vertices corresponding to ploy (tri, quad)

  - 3 signals corresponding to color (RGB) values of the poly
  - 1 signal indicating whether quad or tri
  - 1 valid bit, indicating input poly is valid
- **Controls:**
  - 1 halt signal indicating that no work should be done
  - 2 x, y vertices indicating screen dimensions
  - 1 four bit vector indicating the degree of supersampling required for the final image. The valid values are one hot.
    * 4'b1000 : lx MSAA
    * 4'b0100 : 4x MSAA
    * 4'b0010 : 16x MSAA
    * 4'b0001 : 64x MSAA
- **Outputs:**
  - 2 vertices describing a clamped bounding box
  - 1 valid signal indicating that bounding box value is valid
  - Signals propagating useful information about poly (color and tri/quad bit)
- **Function:**
  - Determine a bounding box for the polygon represented by the vertices. Clamp the bounding box to the sub sample pixel space.
  - Clip the bounding box to screen space.
  - Halt operating but retain values if next stage is busy.
- **Long Description:**
  - This bounding box block accepts a micropolygon described by three/four vertices and determines a set of sample points to test

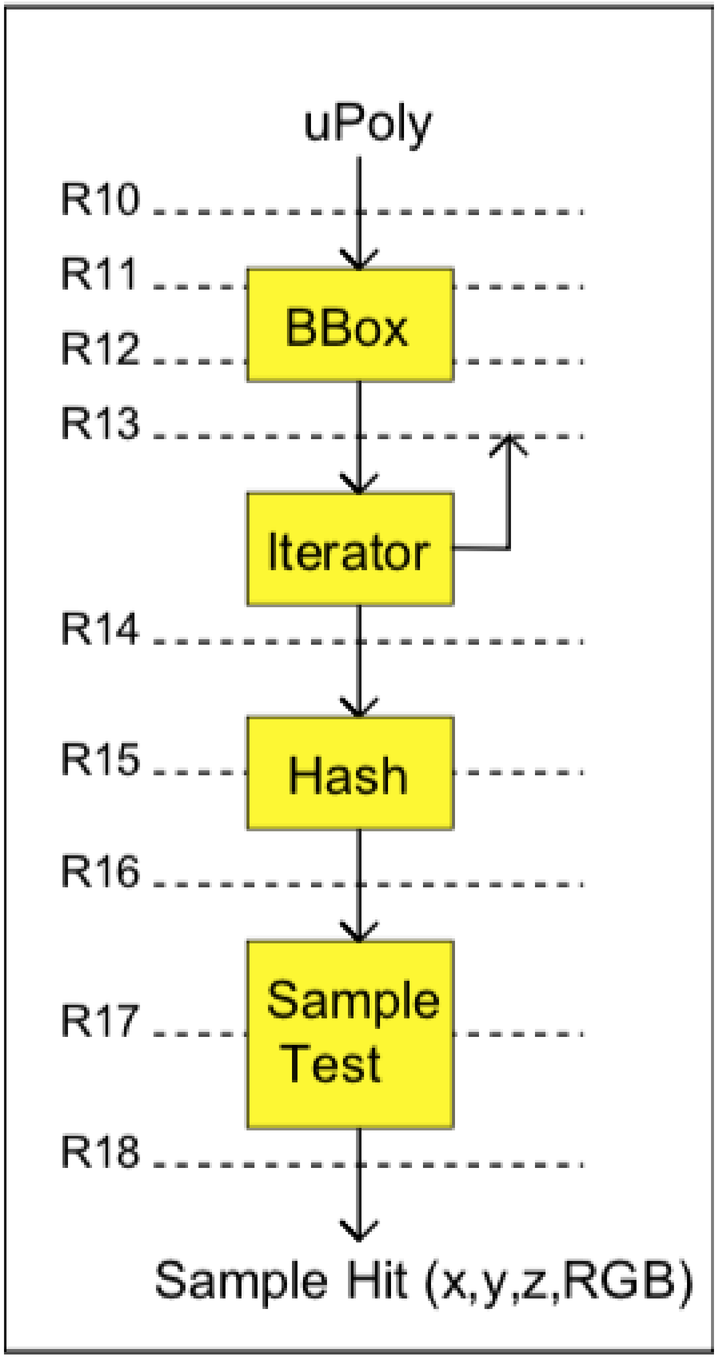

Figure 12: Hardware Pipeline

  against the micropolygon. These sample points correspond to the sub-pixel fragments that compose the pixel, since multi-sample anti-aliasing (MSAA) is enabled. The clamped bounding box was chosen as the description for the set of samples to test as it is concise and it is easy to calculate.
- The bounding box can be determined through calculating the maximum and minimum for x and y to generate a lower left vertices and upper right vertices. Next, the bounding box needs to be clamped to the fragment grid. This can be accomplished through rounding the bounding box values to the fragment grid. Additionally, any sample points that exist outside of screen space should be rejected. So the bounding box can be clipped to the visible screen space. This clipping is done using the screen signal, by limiting the bounding box dimensions to the screen dimensions.
- The halt signal is used to hold the current polygon bounding box. The reason is that only one sample can be tested per cycle. As a bounding box can hold multiple samples, the box data must be held while the samples in the box are iterated over. The halt signal is also required for when the write device, which is located down the pipeline, is full/busy.
- The valid signal is used to indicate whether or not a micropolygon is actually available. This can be useful if the device being read from, has no more micropolygons, or if a bounding box exists entirely off-screen.

`rtl/test_iterator.sv`

- **Inputs:**
  - Bounding box and micropolygon information
- **Controls:**
  - 1 four bit vector indicating the degree of supersampling
- **Outputs:**
  - Subsample location and micropolygon information
  - 1 halt signal indicating that previous stages of pipeline should be halt
- **Function:**
  - Iterate from left to right and bottom to top across the bounding box.
  - While iterating set the halt signal in order to hold the bounding box pipeline in place.
- **Long Description:**
  - The iterator starts in the waiting state. When a valid micropolygon bounding box appears at the input, the iterator will enter the testing state the next cycle. The first sample is equivalent to the lower left coordinate of the bounding box.
  - While in the testing state, the next sample for each cycle should be one sample interval to the right, except when the current sample is

at the right edge. If the current sample is at the right edge, the next sample should be one row up and all the way to the left. Additionally, if the current sample is on the top row and the right edge, the next cycles sample should be invalid and equivalent to the lower left vertices. In this case, the iterators next state should be waiting.

`rtl/hash_jtree.sv`

- **Inputs:**
  - Sample coordinates and micropolygon information
- **Controls:**
  - 1 four bit vector indicating the degree of supersampling
- **Outputs:**
  - Jittered sample coordinates and micropolygon information
- **Function:**
  - Calculate an x displacement and y displacement distributed over the sample area adding small noises in order to prevent Moir pattern. The hash function calculates this displacement using an XOR tree that takes the sample coordinates as inputs.

`rtl/sampletest.sv`

- Samples are tested
- **Inputs:**
  - Sample and micropolygon information
- **Outputs:**
  - Subsample hit flag, subsample location, and micropolygon information
- **Function:**
  - Using edge equations determine whether the sample location lies inside the micropolygon. In the simple case of the triangle, this will occur when the sample lies to one side of all three lines (either all left or all right). Additionally, if backface culling is performed, then only keep the case of all right. This is arbitrarily determined by the tessellation unit that appears much earlier in the pipeline. Currently, the behavior is to back-face cull. This behavior can be changed by altering the logic after distance evaluation.
  - This block evaluates the three edges described by the micropolygons vertices, to determine which side of the lines the sample point lies. Then it determines if the sample point lies in the micropolygon by and'ing the result of the three distance evaluations.

- Edge equation:
    * For an edge defined as traveling from the vertices `(x1, y1)` to `(x2, y2)`, the sample `(xs, ys)` lies to the right of the line if the following expression is true:
    * `0  > (x2 - x1)(ys - yl) - (xs  - x1)(y2 - yl)`
    * Otherwise the sample lies on the line (exactly 0) or to the left of the line.

Signals

Most signals have a suffix of the form `_Rxx:N` where R indicates that it is a Raster Block signal, xx indicates the clock slice that it belongs to and N indicates the type of signal that it is. H indicates logic high, L indicates logic low, U indicates unsigned fixed point, and S indicates signed fixed point.

Here is a list of common signals in the Raster pipeline.

- `poly_RxxS`: 3 sets of X,Y,Z signed fixed Point values
- `color_RxxS`: An RGB vector describing the color of the micropolygon
- `isQuad_RxxH`: A flag indicating that the micropolygon is a quad
- `validPoly _RxxH`: A valid data flag for the micropolygon
- `halt_RnnL`: Flag that indicates no work should be done
- `screen _RnnS`: Designates the width and height of the screen
- `subSample_RnnU`: Designates the level of super sampling
- `clk`: Clock
- `rst`: Reset
- `box_RxxS`: 2 sets X,Y fixed point values
- `sample_RxxS`: A sample location to be tested, 1 set X,Y fixed point values
- `validSamp_RxxH`: The sample is valid to be tested
- `hit_valid_RxxH`: Flag indicating if sample lies inside the micropolygon

This project is broken into three assignments. The first assignment will challenge you to understand the hardware design. The second assignment will require you to complete RTL implementation and synthesize the whole design. Finally, the third assignment will challenge you to increase the performance or the energy efficiency of the design to meet specifications.

- Assignment 1
- Assignment 2
- Assignment 3

Assignment 1

In the first assignment, you are asked to finish the gold model for the design. As you may recall from lecture, the gold model is a model for the functionality of the design absent of many of the details that make it hardware implementable.

The purpose of this exercise is for you to become more familiar with how the overall algorithm works and better understand the initial design.

In the next assignment, we will use this code as part of our testbench to check the functionality of our RTL design.

Instructions

To get started, load the EE271 setup script `setup_ee271.cshrc`.

For this assignment, you will need to fill in C code for the BBox and Sample Test functions.

The file is located at `gold/rasterizer.c`. Look for the comments "// START CODE HERE" and "// END CODE HERE".

- There are some comments in the file, but the pseudo code should be your main reference. Keep in mind that it is all in fixed point.
- Once you are satisfied with your addition, compile from assignment1 directory:

  ```
  make clean comp_gold
  ```

- To run the gold model with a test vector:

  ```
  ./rasterizer_gold out.ppm $EE271_VECT/vec_271_01_sv.dat
  ```

- To compare the output with the corresponding reference image:

  ```
  diff out.ppm $EE271_VECT/vec_271_01_sv_ref.ppm
  ```

  Note: No output means that the files match.
- The output can be viewed with (not necessary but possibly useful):

  ```
  display out.ppm &
  ```

- To compare the output with the reference image in MATLAB (not necessary but possibly useful):
- Start MATLAB:

  ```
  matlab
  ```

- Then the following commands should execute in MATLAB command line:

  ```
  >A = imread('output.ppm');
  >B = imread('/path/to/reference.ppm`);
  >C = abs(A - B);
  >find(C) $ this shows positions of non-zero values in matrix C
  >imshow(C*255) $ this shows a image of matrix C
  ```

- There are some other test vectors and reference images in your `$EE271_VECT` folder which you should try in order to test your code.

Submission

In this assignment, you need to submit the project folder containing code of the C gold model, which should be able to pass all the test vectors. The folder you wish to submit should be called: `assignment1`

Clean up the folder with the command:

```
make cleanall
```

The folder `assignment1` should contain an extra file: `names.txt`. The file `names.txt` should contain 6 lines: the first student's SUID on the first line, first student's name on the second line, first student's email on the third line, second student's SUID on the fourth line, second student's name on the fifth line, and second student's email on the sixth line. For example:

00001111

Olivier Jin

ojin@stanford.edu

00001112

Zelin Zhang

zelinzh@stanford.edu

You should tar ball this directory with the command:

```
tar -czvf assignment1.tar.gz assignment1
```

You can also test your submission with the command:

```
./grading_ass1.pl assignment1.tar.gz
```

This is more or less the script we will use to evaluate your submissions. If the script fails while processing your submission you may have misformatted your submission. Submit your assignment1.tar.gz on Gradescope.

The DUE DATE of this assignment is Monday, November 11

ONLY ONE SUBMISSION is needed for a group of two students.

Assignment 2

(This is a continuation tutorial to your Assignment 1 tutorial)

Your gold model is (hopefully) functional now, but you realize that TAs give you incomplete RTL. Assignment 2 challenges you to implement the bounding box function and the sample test function, as well as a finite state machine, which is currently missing in the test iterator module, and then verify the functionality of your design.

Instructions

We provide starting code for this part as part of the distribution, including a gold solution from Assignment 1 (after the time Assignment 1 is due). Yet, we encourage you to start Assignment 2 early.

Rather than waiting, use your Assignment 1 code. The only difference is that after Assignment 1 is due, we will provide a solution with a complete gold model (thus students that do not do very well on assignment 1, can still do well on assignment 2).

As with Assignment 1, make sure you are first inside of tcsh before proceeding…

```
tcsh
```

and that you have sourced the class setup file…

```
source setup_ee271.cshrc
```

0. Become familiar with the design and verification environment

```
make debug
```

will open up Verdi and show you the module hierarchy on the left side. Click on modules to see the corresponding source files. Click Tools (on the toolbar) -> New Schematic from Source -> New Schematic to view a schematic of the overall design. Back in the source code tab, double click on signals to trace drivers and right click for more options, including tracing loads. Become familiar with the overall design in each of the modules.

1. Implement bbox module: The bbox module is located at `rtl/bbox.sv`. There are three tasks you need to do. The first one is to determine a bounding box represented by the vertices. The second task is to clamp the bounding box to the sub-sample pixel space. Then you need to clip the bounding box to screen space and output the final bounding box and its valid signal. We have declared the signals that connect these three steps to guide your design. You may want to declare other intermediate signals. There are some useful comments in the code. Please read them carefully. You should also refer to the description in Section 4.1.

2. Implement test iterator's FSM: The incomplete FSM is located at `rtl/test_iterator.sv`. The FSM should be able to iterate across the bounding box and output the sample points to next stage. To allow you to explore different FSM designs, we set up a Systemverilog parameter `MOD_FSM` and two branches later in the code. You only need to implement ONE FSM in the first branch and you can revisit it for performance

improvement in the third part of this project. However, if you can come up with a better FSM design, please implement it in the second if branch in test iterator. There are some useful comments in the code. Please read them carefully. You should also refer to the description in Section 4.2. To use the modified FSM, you can set `MOD_FSM=1` in the parameter definition of `test_iterator.sv` (line 92).

3. Implement sample test module: The sample test module is located at `rtl/sampletest.sv`. Your job is to produce the value for `hit_valid_R16H` signal, which indicates whether a sample lies inside the triangle. The signal is high if the input sample is valid and the sample is inside the triangle (consider back-face culling). We have declared some helper signals. If you want, you can follow suggested steps (see comments in code) and use these helper signals. However, you are free to choose how to implement it, as long as you can produce the correct value for the `hit_valid_signal`. As always, please read the comments carefully.

4. Implement verification for the hash module The smpl_cnt_sb module is located at `verif/smpl_cnt_sb.sv`, and the interface between the testbench and the gold model is located at `gold/rasterizer_sv_interface.c`. The check_hash function checks that for a given sample, the jitter value and the jittered sample are correct. Import and call this function in `verif/smpl_cnt_sb.sv`.

5. Run verification: After you complete your RTL code, run the testbench to check your design:

```
$: make run RUN="+testname=$EE271_VECT/vec_271_01_sv_short.dat"
```

As you may already know, there are more test vectors and reference images in `$EE271_VECT` folder. You should try to run simulation using short test vectors first, since the long vectors will take a long time.

If your simulation shows a mismatch between the gold and RTL, you are now entering the DEBUG phase of logic design.

The best way to debug is with a waveform. Generate a waveform by using run_wave instead of run:

```
$: make run_wave RUN="+testname=$EE271_VECT/vec_271_01_sv_short.dat"
```

And then launch Verdi with the waveform:

```
make debug_wave
```

The scoreboard should have specified the signals that mismatch at a given time. Open up the corresponding module, and add signals to the waveform by right-clicking on the signal name in the source code, and selecting “Add to waveform”. Double click on signals to find their drivers, and add those to the waveform as well. It may be helpful to turn on Active Annotation (Source->Active Annotation). In the waveform window on the bottom, change the time (right of the golden arrow) to the time at which the simulation reported the error.

Besides looking at the waveform, it's also helpful to look at values in a schematic. Click on a signal in the source window, and then click on Tools->New schematic from Source->Fan-in Cone. From here, it might be useful to see all the net names in the schematic, which can be turned on by View->Net name in the schematic window toolbar.

Resolve all errors, and once the simulation completes successfully, verify that the output matches the corresponding reference image:

```
$: diff verif_out.ppm $EE271_VECT/vec_271_01_sv_short_ref.ppm
```

It is time to take a look at the performance of your design. You should be able see two performance numbers `triangle/cycle` and `cycle/triangle` at the end of the simulation, which are literally the average number of triangles processed per cycle and its reciprocal collected by a performance monitor `perf_monitor` in the testbench. Note the actual values of these numbers depend on the test vectors you are running. For example, if the input has a lot large triangles rather than triangles, like `vec_271_00_sv.dat`, your `triangle/-` cycle must be lower than those from triangles test vectors. You should be able to get more real numbers from running full size triangle rendered image inputs, i.e. `vec_271_01_sv.dat`, `vec_271_02_sv.dat` and `vec_271_04_sv.dat`.

If you implemented another FSM in the second if branch in `test_iterator` and want to run verification with the modified FSM, you can change the default value of SystemVerilog parameter `MOD_FSM` to `1` in test_iterator and run verification (and later synthesis) again.

After passing the verification, you can check the performance of a different FSM design.

We have also provided you some debug assertions. You might find it easier to run formal verification, as it provides a faster counterexample. The length of the bounded model checker is set by bmc_length parameter in the problem.txt file. Set it initially to a smaller value (bmc_length = 5), you can increase it once you are confident enough that your code is correct.

Instructions to run formal verification can be found at the end of the assignment.

6. Run synthesis: Run synthesis from the project directory:

```
$ make run_dc
```

The directory `reports` inside the `synth` directory contains a number of reports related to the synthesis run. You are interested in the file `timing_report_max`, which contains the longest paths in the design. The report will list the path signal names and whether the path has violated timing. The first and last signals in the critical path should contain an instance name related to a flip flop. Since we have turned on retiming, the dff instance name will change after synthesis tool moves it around. So in order to correlate the signal names, you want to check the name of dff's parent instance carried in the signal name and get a sense of where the critical path is. For example, `sampletest/DP_OP_28J1_125_9254/clk_r_REG379_S13/D` is the name of the last signal in a critical path. `DP_OP_28J1_125_9254` is obviously a tool-generated name rather than a given name in RTL. However, you can some how know that it is a signal connected to the `D` port of a register (flip-flop) inside `sampletest` instance. After you find where the critical path is, you can make changes to the RTL which do not affect its functionality but result in eliminating or improving the critical path, such as retiming.

There are a bunch of other handy reports in the `reports` folder. For example, `area_report` summarizes the area information of the chip. `design_check` is a list of synthesis warnings, while `error_checking_report` is a list of errors. Make sure you don't have any errors, and we will check it in the grading script. `power_report` is the power report of your design.

Once you have made your changes rerun the synthesis script and verify that your fix worked. It is also imperative that you rerun your verification bench in order to make sure that functionality was not adversely affected.

7. Retiming: In assignment 2, you only need to have a functionally correct design, that can run synthesis with default clock period and chip area budget. Optimization is the goal of the next assignment. However, you must be interested to see how fast your design can run. You may also want to try some simple techniques, like retiming, to improve the performance. Now try to run the synthesis again with a faster clock, say 0.8ns, by changing the value of the variable `CLK_PERIOD` in the Makefile. You can do this by running the command `make run_dc CLK_PERIOD=0.8 ...`

After synthesis is done, there will likely be timing violations, since even the powerful synthesis tool fails to put all the logic in each stage into one clock cycle. However, by analysing the timing report, you should be able to see where the critical path is. Recall the retiming technique we have introduced in lectures. We can increase the pipeline depth and move around the registers to break a long combinational path into small slices.

The good thing is that the synthesis tool can handle the retiming pretty well after you tell it how many stages you want to allocate for each part of the pipe. In this project, you can change the pipeline depths of bbox, hash tree and sampletest modules by modifying the SystemVerilog parameters of `params/rast_params.sv` module. Run synthesis again to see the results.

If you are doing this, don't forget to hard code your final parameter values in the submission, since we use default command for grading.

Try to hit the highest clock rate by retiming. From your results, which design can run at a higher frequency? Think of why this is the case. You should submit a final design which can pass synthesis at default clock period (1.2ns) without any timing violations. Make sure the area of your chip doesn't exceed

42000(um2), which is the target area in synthesis. (If you put too many pipe stages, the registers will cost area.)

Formal Verification

For this part, you are expected to write some SystemVerilog Assertions using CoSA in `rtl/bbox.sv` and `rtl/test_iterator.sv`. Follow the comments in the code to find where you should insert your assertions and what properties they should check.

To perform the verification, first move to the formal directory.

```
cd formal
./cat-script.sh
```

The cat-script generates a single file (`rast.sv`) in your formal directory from your project directory to input into CoSA.

Then run

```
CoSA --problems problem.txt
```

You can also run `CoSA --help` if you'd like to see more options.

CoSA will try to prove the assertions defined in your SystemVerilog code. If it's able to successfully prove it, it returns TRUE. Sometimes, it's not able to formally prove it but still cannot find a counterexample in the given bounds, and therefore outputs UNKNOWN. If your bounds are large enough (you don't have to set it, we have set it for you), it is a reasonable guarantee that the assertion is true.

The submission for this part is the output of the command `CoSA --problems problem.txt` with `bmc_length = 10`. You can change `bmc_length` (and other properties) inside of the provided `problem.txt`.

Submission

For this part of the assignment, you should submit the correct design with bbox, sampletest, test iterator FSM implemented, and output of COSA. You may have a good number of working design configurations, but please adjust your RTL and SystemVerilog parameters so that in the default configuration, it can run synthesis with default clock period and chip area budget. The submitted RTL code must have correct functionality, which means it should be able to pass all the test vectors (short and long vectors). This part is graded by correctness (your budget is large enough). However, it's better to resolve critical paths and hit a higher clock rate. This will place your team in a better position in the Assignment 3 performance competition.

You will receive partial credit if you pass the verification but have timing violation after synthesis.

The DUE date of this assignment is 11/20. The submission requirement is the same as Assignment 1 (see Section 5.1.2), except to name your folder and tar ball as assignment2. Please clean up your folder, add a names.txt, tar ball it, test with the grading script and submit the tar ball on Gradescope .

Assignment 3

This part of the project will challenge you to meet the hardware specification and is more open ended than the prior two assignments. The lead architect has indicated to us that we must render scenes containing 5 million triangles onto a 16xMSAA buffer at a rate of 100 Hz. This corresponds to a throughput of 1 triangle every 2ns (i.e. 2ns/triangle or 0.5triangle/ns).

Instructions

The way to estimate the performance of your design is to take the 'cycle/triangle' from simulation and multiply it by the clock period from synthesis (make sure the timing is MET) to get the average time to process a miropolygon (ns/triangle).

Your goal here is to build a design that matches the leading architect in performance (2ns/triangle) and then try your best to optimize the power consumption or the chip area or both. One easy way to quickly match the performance is to simply duplicate the hardware instances. We don't want you to do this duplication in RTL, so in calculation please just state how may copies (should be an integer) of the rasterization pipeline you need to hit the target. Since this graphic application can be well-paralleled, you may assume that throughput scales linearly with the number of instances, but don't forget to scale the power and area correspondingly at the same time.

After you match the throughput target by simply duplicating hardware or overclocking, now it is time to do some real optimization to reduce the chip area or power dissipation. Remember VLSI is always about tradeoffs, which means here you can easily trade more area for low power and vice versa. Therefore, a good way to evaluate your design optimizations is to look at the tradeoff curve, which is also more or less what we will do to grade your optimization results. Figure 13 shows a reference area-power tradeoff curve of this triangle pipeline after some optimizations. If the point (area, power) of your design is to the lower left of the curve, it means your design is better than the reference design and vice versa. The figure of merit is Area * Power * Num_Rasterizer_Units^2. A lower FoM is better.

Figure 13: Area versus Power Trade-off Curve

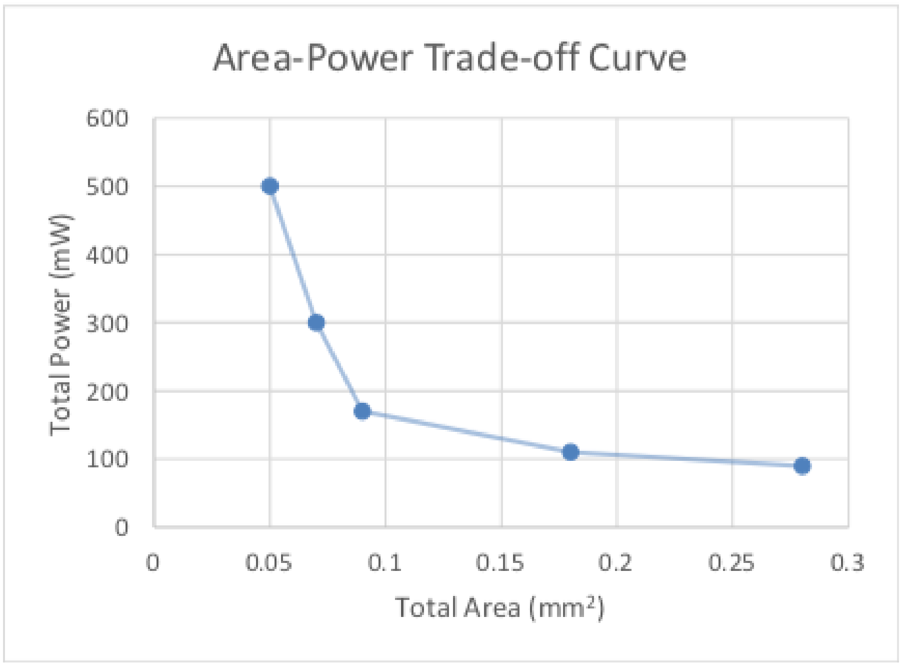

Figure 13: Area versus Power Trade-off Curve

The following list of ideas will be helpful in this assignment.

Code Optimization: For the same logic, the way you implement it could make a difference to the synthesized hardware, and thus the timing, area, and power.

FSM: Continue to optimize the FSM in the test iterator. Try to eliminate wasted cycles, make critical paths inside shorter, minimize the numbers of states (registers), or build a completely new FSM.

Retiming: If you decrease the clock period it would increase throughput. See Assignment 2 for instructions.

BackFace Culling: Currently the design waits until sample test time to determine if a triangle is backfacing or forward facing and only for individual samples. Performing the test earlier would eliminate those extra tests and improve throughput.

Lower Precision: High precision operations, like multiplications, are expensive in terms of delay, power and area. Try to optimize the precision of your arithmetic operations, but be careful that this optimization should not change the results. You still need to generate the same images as corresponding reference images. No error at output is tolerable.

Bubble Smashing: In the current design it is possible for bubbles to appear between triangles during the bounding box stage. While halted it may be useful to advance a triangle into a pipe stage which doesn't hold any values. This will be especially useful if you implement backface culling.

MultiTest: Testing multiple samples per cycle is a sure way to increase throughput. This change will impact the verification collateral, be sure to maintain consistency between the two.

Better Bounding Box: The bounding box method is only one method of iterating over the samples that might appear in a bounding box. There are many others. One of these other methods might help you reach the throughput you need. One algorithm you might try is given in this work.

Clock Gating: Clock gating is a technique, which enable you to dynamically prune the clock tree. When you disable the clock signal to those parts that don't generate valid values, you save all the dynamic energy dissipation (leakage still there) in those parts.

Be Creative: Find a way to increase the throughput not listed here.

Last but Not Least: Try to do some research on this topic, sometimes it is important to stand on the shoulders of giants. Here is a good starting paper.

To be able to do well on this assignment you **should**:

1. **Predict the Efficacy of Your Approach**: Using back of the envelope calculations predict what the effect of your change will be. It is very important that you attempt to quantify your changes in advance. Failing

to do so can result in significant amounts of work with only minor performance benefits. Go for the lowest hanging fruit, and the biggest bang for the buck!

2. **Measure Throughput**: Using the verification environment, calculate your throughput in triangles per cycle. Use the various traces and to measure the incremental benefits of each of your changes.
3. **Verify Your Design**: Using the verification environment provided, plus verification environment modifications you needed to make to accommodate your functional changes, test your design. Then test it again. And again. Be thorough!
4. **Synthesize Your Design**: Make sure that your design meets timing and is still synthesizeable. You can alter the clock period in the synthesis script if this is the approach you wish to take.
5. **Check Your Specification**: The report directory also contains a power report and area report which you can use to evaluate whether your design is inside the power and area limits.

Write-Up

For the write-up, please report the following:

- **Total Dynamic Power (mW)**
- **Total Leakage Power (mW)**
- **Total Power (mW)**
- **Total Area (mm^2)**
- **Total Performance (triangles/second)**
- **Number of Rasterization Units**
- **Clock Speed (ns)**

List all optimizations, and for each, report (no more than 1 page per optimization):

- **Before implementing the optimization, how did you evaluate the efficacy of the optimization? How did you estimate cost and performance?**
- **How did you implement the optimization? Was there anything particularly difficult that you were not expecting?**
- **What were the actual changes in cost and performance? How/Why did this deviate from your estimation?**

- **You are encouraged to include effective and clear figures or plots to help comment on any of the above.**
- **It's helpful to include tables or plots to compare the change of the throughout, the number of rasterization units needed, the clock period, the total power, and the total area for each of your optimizations.**

Finally:

- **Is there any analysis/discussion that you did for the project that you would like to share (no more than a page)?**
- **Please provide any concluding thoughts on the project and share anything you think that can improve the design.**

Submission

For this part of the assignment you will submit the final RTL design and a short write-up detailing your work. There will be two submissions - report and the assignment3 tar ball. Submit both together on Gradescope.

The DUE date of this assignment is 12/4.

The requirement for RTL submission is the same as Assignment 1 (see Section 5.1.2), expect to name your folder and tar ball as assignment3. Please clean up your folder, add a names.txt, tar ball it, test with the grading script and submit the tar ball on Gradescope. The grade of this assignment consists of two parts: 60% for the optimization result and 40% for the write-up report. The optimization result part will be graded relatively based on the results of the class. # Appendices

Rasterization for Quadrilaterals (Triangle-Pairs)

Sample Test Algorithm

The triangle-pair case is a little more complicated, but there is an advantage to allowing triangle-pairs into the pipeline. To a first degree, rasterizing a triangle-pair is a lot like rasterizing two triangles at once. A better explanation is that the bounding box is a tighter bound on the pair than on the singleton. This tighter bound means that the ratio of sample hits to misses is higher. This higher ratio is desirable as it increases the pipeline's microp-olygon throughput.

The representation for these pairs is slightly different than for a single triangle. For triangle pairs the shared edge is assumed to be from the second vertex to the fourth vertex. These complications generate a set of min-terms as opposed to the single min-term for the triangle test. However these terms reduce to an expression representing the case when a sample is in one triangle or the other.

From the psuedocode in the following section you can see that there are a total of 10 cases where the sample in quadrilateral test is a hit. The following list enumerates those cases and gives an example for each one. Note that in these cases `b0` represents the test for the sample point being right of edge 0, `b1` edge 1, `b2` edge 2 and so on and is consistent with the psuedocode given on the next page. The green edge corresponds to the fourth edge from the first to third vertices whose sample right of edge test result is `b4`.

- `(!b4 && !b3 && !b2 && b1 && !b0)` given in Figure 14

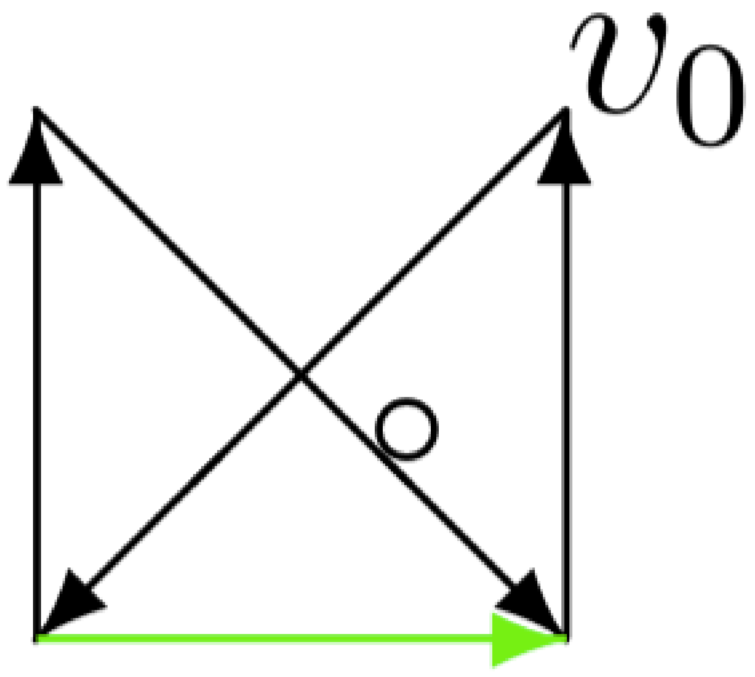

Figure 14: Sample Case 1

Figure 14: Sample Case 1

- `(!b4 && !b3 && b2 && !b1 && !b0)` given in Figure 15

  Figure 15: Sample Case 2

- `(!b4 && !b3 && b2 && b1 && b0)` given in Figure 16

  Figure 16: Sample Case 3

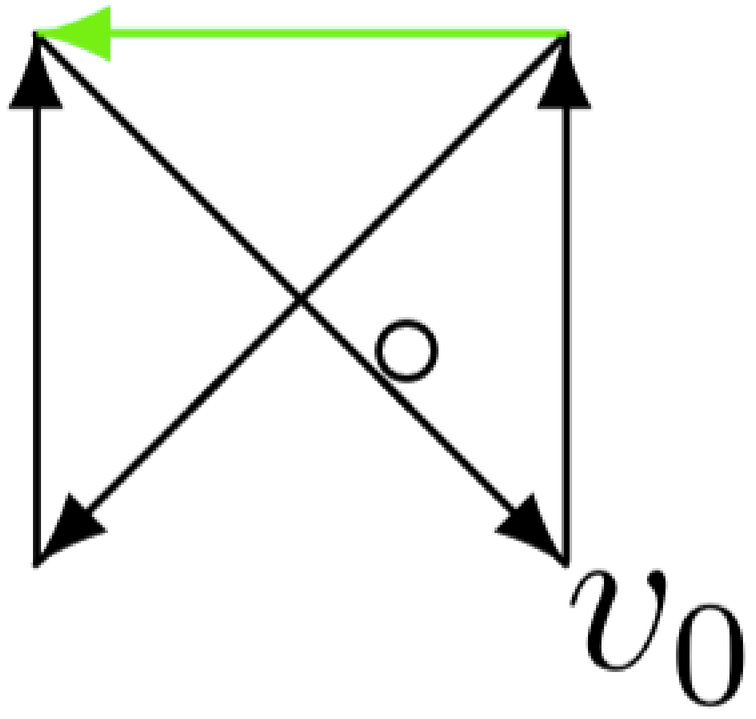

Figure 15: Sample Case 2

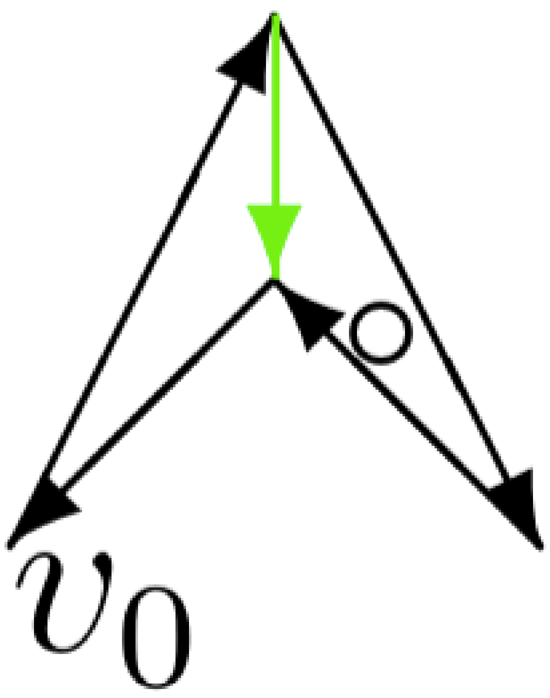

Figure 16: Sample Case 3

- `(!b4 && b3 && b2 && b1 && !b0)` given in Figure 17

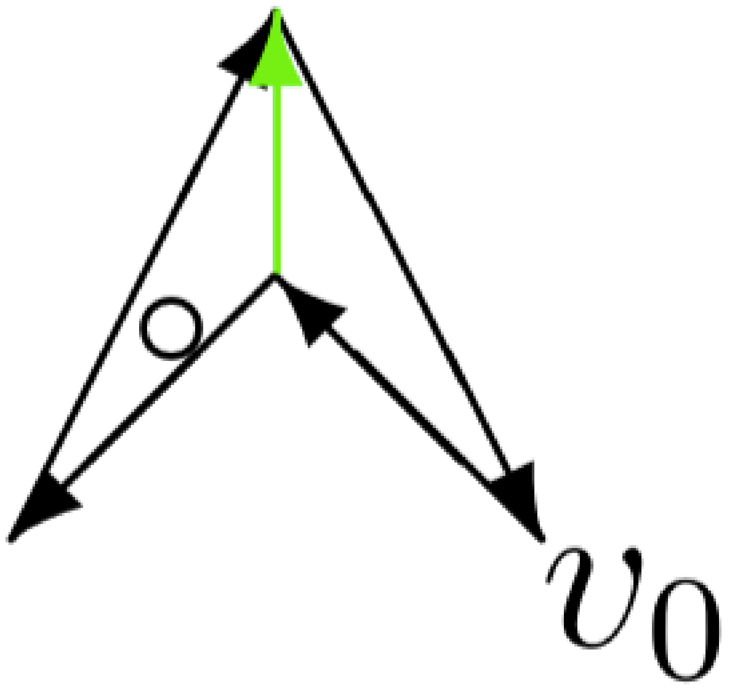

Figure 17: Sample Case 4

Figure 17: Sample Case 4

- `(!b4 && b3 && b2 && b1 && b0)` given in Figure 18

  Figure 18: Sample Case 5

- `(b4 && !b3 && !b2 && !b1 && b0)` given in Figure 19

  Figure 19: Sample Case 6

- `(b4 && b3 && !b2 && !b1 && !b0)` given in Figure 20

  Figure 20: Sample Case 7

- `(b4 && b3 && !b2 && b1 && b0)` given in Figure 21

  Figure 21: Sample Case 8

- `(b4 && b3 && b2 && !b1 && b0)` given in Figure 22

  Figure 22: Sample Case 9

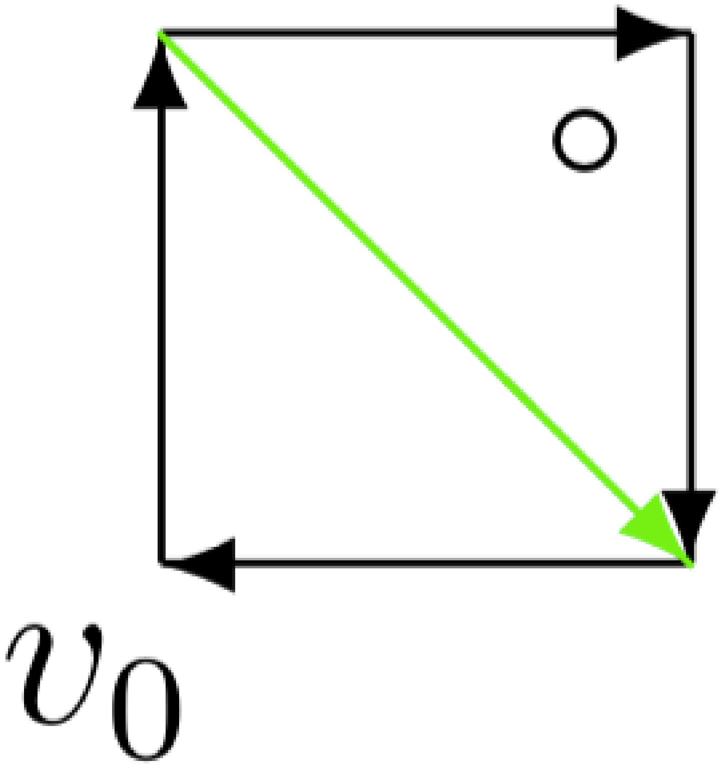

Figure 18: Sample Case 5

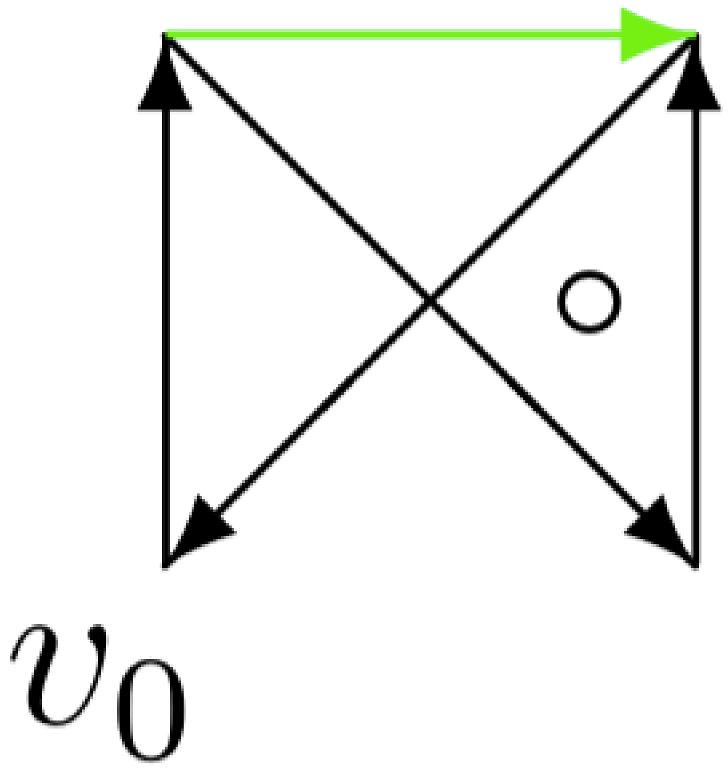

Figure 19: Sample Case 6

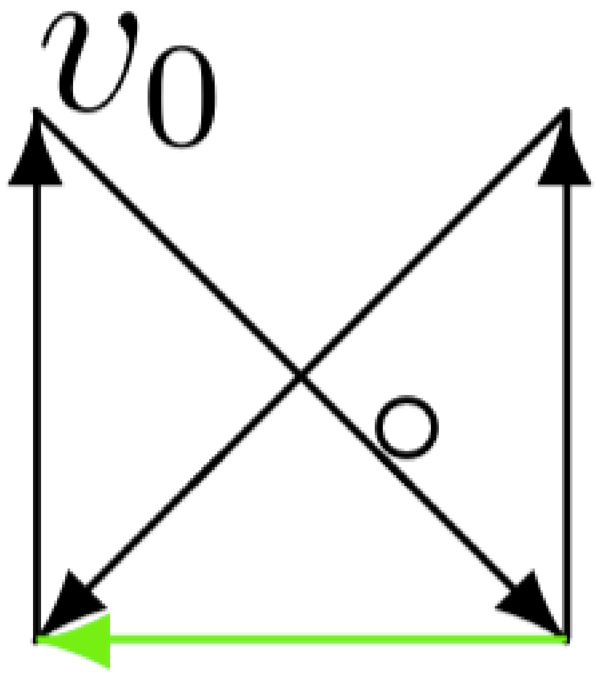

Figure 20: Sample Case 7

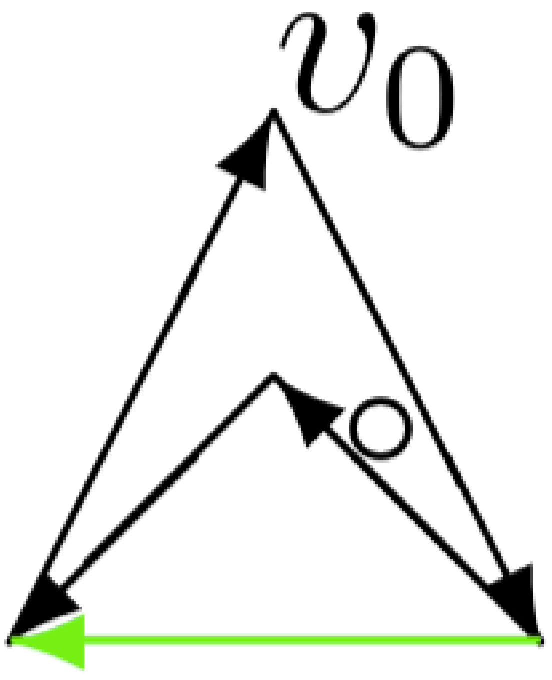

Figure 21: Sample Case 8

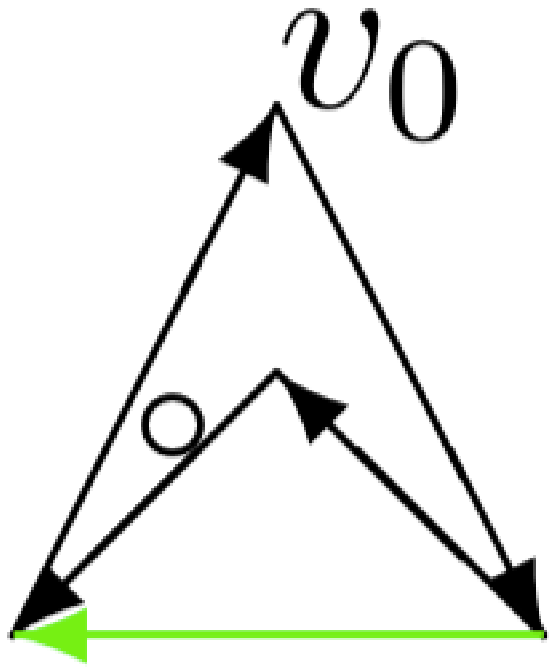

Figure 22: Sample Case 9

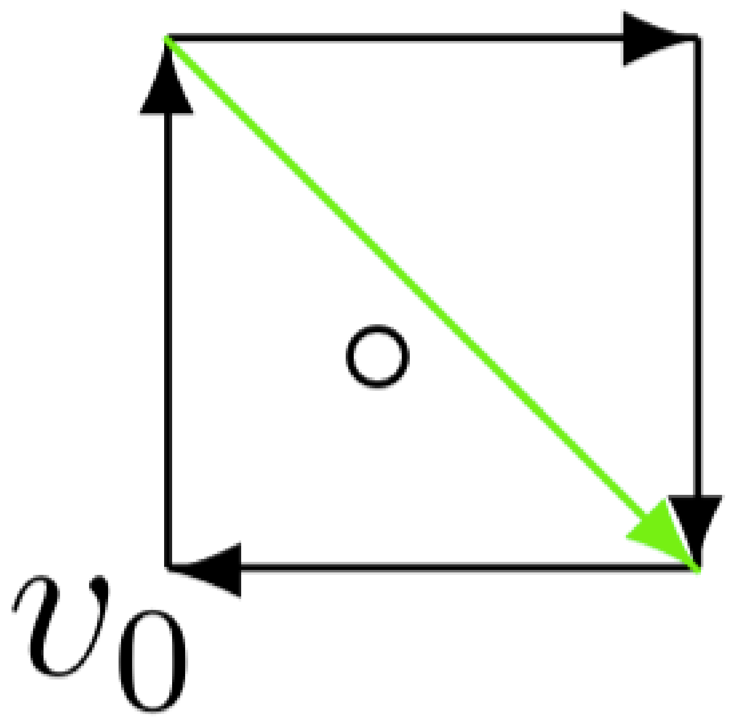

Figure 23: Sample Case 10

- `(b4 && b3 && b2 && b1 && b0)` given in Figure 23

  Figure 23: Sample Case 10

Pseudo Code

Pseudo code for micropolygon rasterization supporting either triangle singletons or triangle pairs:

```
void rast(vector<u_Poly> polys){
    for(i = 0; i < polys.size(); i++){
        rast_upoly(polys[i]);
    }
}

inline void rast_upoly(poly) {
    // Calculate clamped bounding box
   ll_x = FLOOR_SS(MIN( x coordinate of poly vertices )); // Min X rounded down to subsample grid
   ur_x = FLOOR_SS(MAX( y coordinate of poly vertices )); // Max X rounded down to subsample grid
   ll_y = FLOOR_SS(MIN( x coordinate of poly vertices )); // Min Y rounded down to subsample grid
   ur_y = FLOOR_SS(MAX( y coordinate of poly vertices )); // Max Y rounded down to subsample grid

    // Clip bounding box to visible screen space
    ur_x = ur_x > screen_width ? screen_width : ur_x;
    ur_y = ur_y > screen_height ? screen_height : ur_y;
    ll_x = ll_x < 0 ? 0 : ll_x;
    ll_y = ll_y < 0 ? 0 : ll_y;

    // Iterate over samples, test if in micro polygon
   // Note that offscreen bounding boxes are rejected by for loop test
    for(s_x = ll_x; s_x <= ur_x; s_x += subsample_width){
        for(s_y = ll_y; s_y <= ur_y; s_y += subsample_width){
            [j_x, j_y] = jitter(s_x, s_y); // Noise for sample
            if(sample_test(poly, s_x + j_x, s_y + j_y)){
                process_fragment(poly, s_x, s_y);
            }
        }
    }
}

inline int sample_test(poly, s_x , s_y){
    q = poly.vertices == 4 ; // Is polygon a quadrilateral

    // Shift vertices such that sample is origin
    v0_x = poly.v[0].x - s_x;
    v0_y = poly.v[0].y - s_y;
    v1_x = poly.v[1].x - s_x;
    v1_y = poly.v[1].y - s_y;
    v2_x = poly.v[2].x - s_x;
    v2_y = poly.v[2].y - s_y;
    v3_x = poly.v[3].x - s_x;
    v3_y = poly.v[3].y - s_y;

    // Distance of origin shifted edge
    dist0 = v0_x * v1_y - v1_x * v0_y; // 0-1 edge
    dist1 = v1_x * v2_y - v2_x * v1_y; // 1-2 edge
    dist2 = v2_x * v3_y - v3_x * v2_y; // 2-3 edge
    dist3 = v3_x * v0_y - v0_x * v3_y; // 3-0 edge
    dist4 = v1_x * v3_y - v3_x * v1_y; // 1-3 edge
    dist5 = v2_x * v0_y - v0_x * v2_y; // 2-0 edge

    // Test if origin is on right side of shifted edge
    b0 = dist0 <= 0.0;
    b1 = dist1 <  0.0;
    b2 = dist2 <  0.0;
    b3 = dist3 <= 0.0;
    b4 = dist4 <  0.0;
    b5 = dist5 <= 0.0;

    // Triangle min terms with no culling
    //triRes = (b0 && b1 && b2) || (!b0 && !b1 && !b2);

    // Triangle min terms with backface culling
    triRes = b0 && b1 && b5;

    // Quad min terms with back face culling
    // Can be implemented as a look up table
    quadRes = ( b1 &&  b2 && !b4 && (b0 ||  b3))
           || (!b1 && !b2 &&  b4 && (b0 xor b3))
           || ( b0 &&  b3 &&  b4 && (b1 ||  b2))
           || (!b0 && !b3 && !b4 && (b1 xor b2));

    return ((triRes && !q) || (q && quadRes));
}
```