EE271 Project Description

1 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 lecture 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 conventional graphic 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 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 Genesis2. As we will discuss soon in class, Genesis2 is used as an extension, or meta-layer on top of SystemVerilog. Genesis2 enables designers to create design generators, or constructors, instead of design instances. We use Genesis2 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 the 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 a overview of the project. Please read the entire description first and only then start working. This documentation has five sections:

2 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 projects implementation.

2.1 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 topic:

- **SystemVerilog**:
  - Verilog Tutorial
  - SystemVerilog Tutorial
    - SystemVerilog Assertions Tutorial
    - SystemVerilog DPI Tutorial
  - Books
    - SystemVerilog For Design
    - SystemVerilog For Verification
      - Interfacing with C
    - Writing Test benches using System Verilog
- **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

2.2 The Modern Real-Time Computer Graphics Pipeline

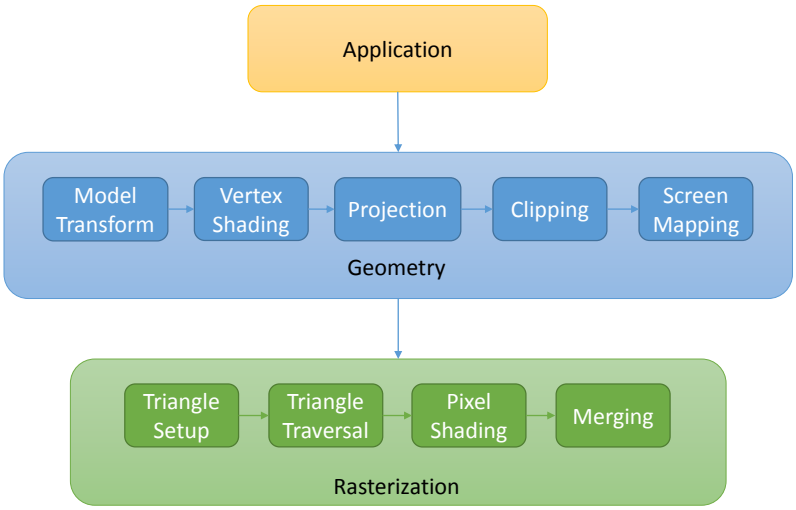

Figure 1: Computer Graphics Pipeline

Computer graphics can be described as the problem of converting 1's and 0'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 processors (CPU), most high end applications require specialized hardware in the form of a graphics processing unit (GPU).

To achieve high speed, the 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 generate 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). An example of such geometry is shown in Figure 2, where the upper left and lower right quadrant show a wire mesh demonstrating the geometric description of the bird.

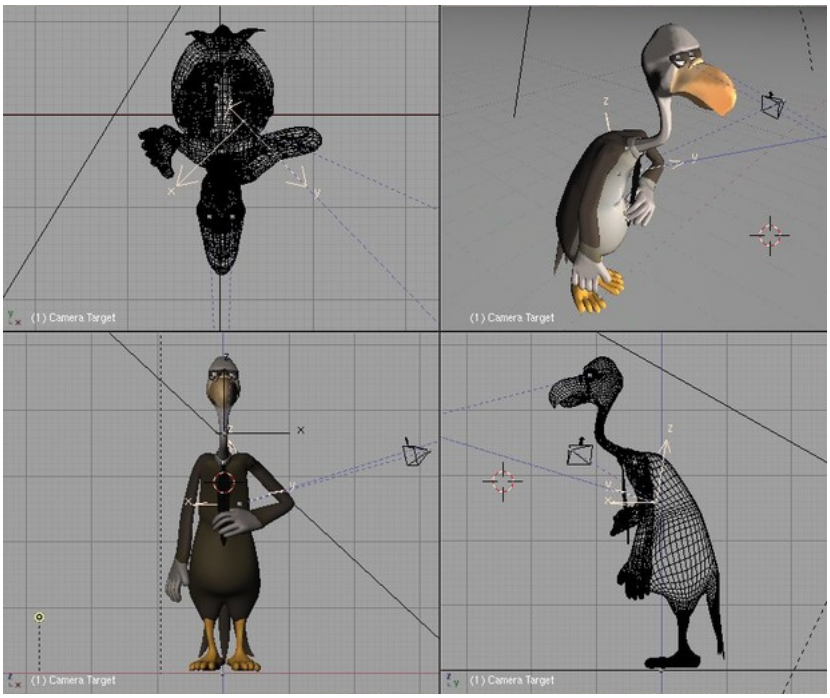

Figure 2: Bird Mesh (Blender Foundation)

Next, the geometry portion of the pipeline applies many kinds of transformations to these mesh data (3-D vertices of every triangles) 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 using it to fill in an image.

2.2.1 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

The data provided by the application may not explicitly specify the location and scale each triangle and quadrilateral in the space, 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 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.

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.

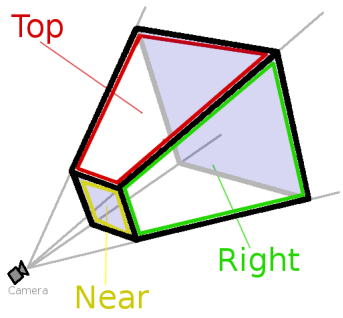

Figure 3: View Frustum (Wikipedia)

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.

2.2.2 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

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.

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 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 polygons 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.

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

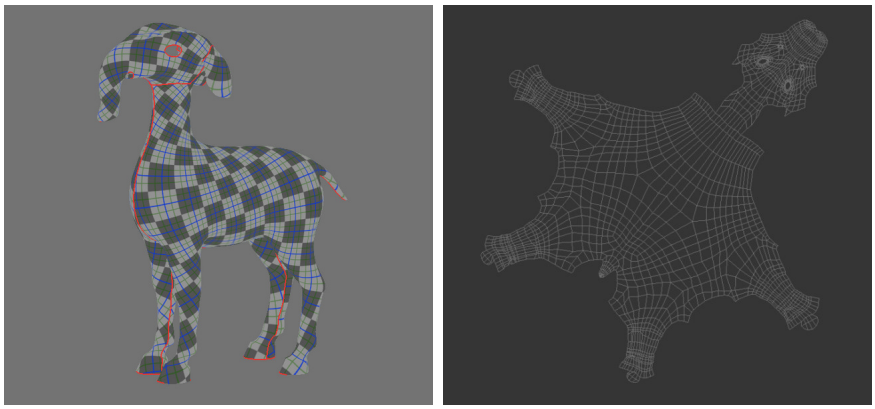

Figure 4: Alien Goat Texturing (Blender Foundation)

occluded triangle to the same pixel as the occluding triangle. During merging, if the occluding triangle is defined as opaque, the occluded triangles pixel is rejected; otherwise, the occluded triangles pixel is mixed with the occluding triangle. The image is complete after all of the pixels have been aggregated.

2.2.3 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.

2.3 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 (Wikipedia)

2.4 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.

2.5 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.

3 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.

3.1 Bounding Box

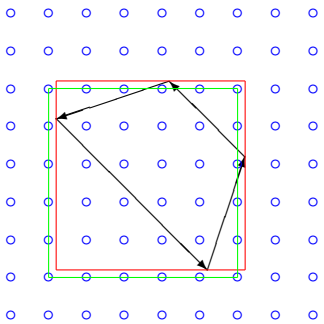

Figure 6: Axis Aligned Bounding Box 16xMSAA

The naive solution for determining the set of samples 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 polygon 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 the upper right coordinate in a left to right and down to up pattern. Because we will be sampling the region with jittered sample, we will need to make sure that our clamped bounding box includes all of pixels whose randomly distributed samples could lie inside the triangle. 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.

3.2 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
- Edges are defined by two vertices's v_1 to v_2
- An equation for the edge can be given in the form:
  $\frac{(y-y_1)}{(x-x_1)} = \frac{(y_2-y_1)}{(x_2-x_1)} = \text{slope}$
- Rewritten:
  $(y - y_1)(x_2 - x_1) = (y_2 - y_1)(x - x_1)$
- Given a fragment position (x, y), it is useful to make a coordination shift such that the fragment resides on the $(0, 0)$ position. This is a shift of $-x$ for X coordinates and $-y$ for Y coordinates. Define $x1' = x1 - x$, $y1' = y1 - y$, $x2' = x2 - x$, $y2' = y2 - y$. Now we test our (x, y) point by placing it in the equation above:
  $0 = x_1'y_2' - x_2'y_1'$
- If the origin is not on the line:
  $0 \neq x_1'y_2' - x_2'y_1'$
- If the origin lies to the left of the line:
  $0 < x_1'y_2' - x_2'y_1'$
- If the origin lies to the right of the line:
  $0 > x_1'y_2' - x_2'y_1'$
- Ex. $v_1 = (1, 0), v_2 = (0, 1) v_s = (0, 0)$ (Figure 7):
  $1 \cdot 1 - 0 \cdot 0 > 0$ therefore left of line.

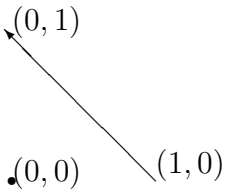

Figure 7: Edge Test 1 - Left of Edge

- Ex. $v_1 = (0, 1), v_2 = (1, 0) v_s = (0, 0)$ (Figure 8):
  $0 \cdot 0 - 1 \cdot 1 < 0$ therefore right of line.

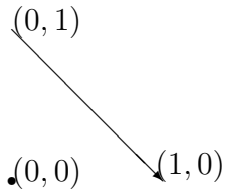

Figure 8: Edge Test 2 - Right of Edge

- Ex. Given a triangle $v_1 = (-2, 3), v_2 = (-1, -1), v_3 = (2, -1), v_s = (0, 0)$ (Figure 9):
  $e_1 : -2 \cdot -1 - -1 \cdot 3 = 5 > 0$ therefore left of line
  $e_2 : -1 \cdot -1 - 2 \cdot -1 = 3 > 0$ therefore left of line
  $e_3 : 2 \cdot 3 - -2 \cdot -1 = 4 > 0$ therefore left of line
  Therefore the point must lie inside the triangle.

- Ex. Given a triangle $v_1 = (-2, 3), v_2 = (2, -1), v_3 = (-1, -1), v_s = (0, 0)$ (Figure 10):
  $e_1 : -2 \cdot -1 - 2 \cdot 3 = -4 < 0$ therefore right of line
  $e_2 : 2 \cdot -1 - -1 \cdot -1 = -3 < 0$ therefore right of line
  $e_3 : -1 \cdot 3 - -2 \cdot -1 = -5 < 0$ therefore right of line
  Therefore the point must lie inside the triangle.

- Ex. Given a triangle $v_1 = (-2, 3), v_2 = (-1, -1), v_3 = (0, -1), v_s = (0, 0)$ (Figure 11):
  $e_1 : -2 \cdot -1 - -1 \cdot 3 = 5 > 0$ therefore left of line
  $e_2 : -1 \cdot -1 - 2 \cdot -1 = 3 > 0$ therefore left of line
  $e_3 : 0 \cdot 3 - -2 \cdot -1 = -2 < 0$ therefore right of line
  Therefore the point must lie outside the triangle.

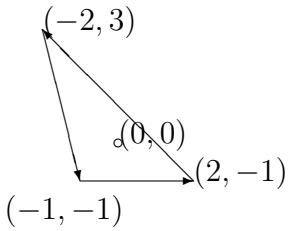

Figure 9: Edge Test 3 - Point Inside

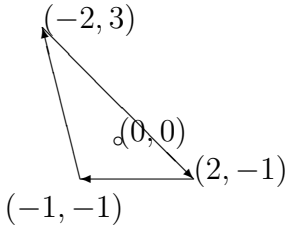

Figure 10: Edge Test 4 - Point Inside

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 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.

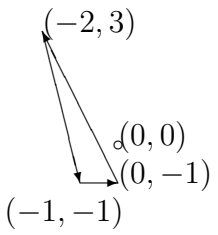

Figure 11: Edge Test 4 - Point Outside

3.3 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 BBox
  ll_x = FLOOR_SS( MIN( x coordinate of poly vertices ) ); //Min X rounded down to subsample grid
  ur_x = FLOOR_SS( MAX( x coordinate of poly vertices ) ); //MAX X rounded down to subsample grid
  ll_y = FLOOR_SS( MIN( y 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 BBox 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 uPoly
  // 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 && b1 && b2  ;

  return ( triRes ); }
```

4 Hardware Overview

Figure 12 is an overview of the hardware. With respect to the Pseudo Code: the BBox stage corresponds to the generation of a clamped axis aligned bounding box, the Iterator stage is a finite state machine (FSM) corresponding to the set of nested for loops which iterate all the sub-pixel positions in the bounding box, the Hash box corresponds to the jitter function to add random noise to the sample position, and the Sample Test box corresponds to the sample test function inside the nested for loops. In the project folder, these function blocks are implemented in rtl/bbox.vp, rtl/test_iterator.vp, rtl/hash_jtree.vp and rtl/sampletest.vp respectively.

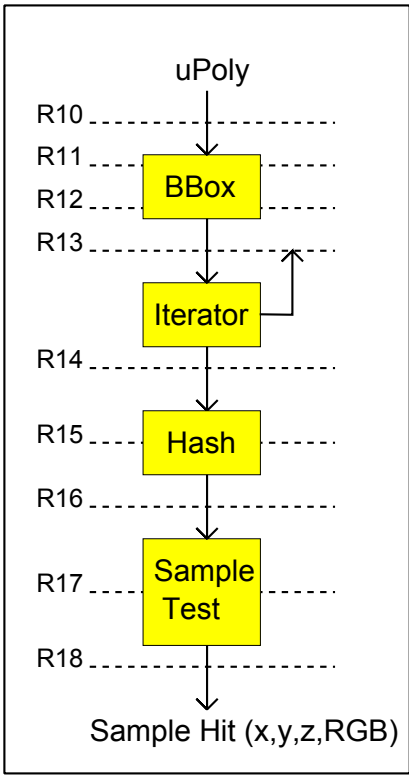

Figure 12: Hardware Pipeline

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 Genesis 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.

4.1 bbox.vp

- Inputs:
  - 3(4) x,y vertices corresponding to tri(quad)
  - 3 signals corresponding to color (RGB) values of the poly
  - 1 signal indicating that quad or tri
  - 1 valid bit, indicating input poly are 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 : 1x 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 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.

4.2 test_iterator.vp

- Inputs:
  - BBox 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.

4.3 hash_jtree.vp

- 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.

4.4 sampletest.vp

- 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 (x_1,y_1) to (x_2,y_2), the sample (x_s,y_s) lies to the right of the line if the following expression is true:
  - $0 > (x_2 - x_1) (y_s - y_1) - (x_s - x_1) (y_2 - y_1)$
  - Otherwise the sample lies on the line (exactly 0) or to the left of the line.

4.5 Signals

Most signals have a suffix of the form _RxxN 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

5 Assignments

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.

5.1 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.

5.1.1 Instructions

To get started, load the EE271 setup script (/usr/class/ee271/setup_ee271.cshrc). The directory containing assignment 1 can be found at /usr/class/ee271/project/assignment1. Copy the incomplete gold model to your own folder.

For the main part of this assignment, you will need to complete two missing functions in the gold model. The first function that you are required to fill in is the bounding box function. The second function is the fragment in micropolygon test. The requirements of these functions are described in the Pseudo Code above. Once you have completed the gold model, there is a set of test vectors you can run, together with a set of reference images to check your results.

1. **Fill in the C++ code for BBox and Sample Test:** In this step we are building a functional implementation of the rasterizer in C++. We will reuse that code as a reference model for our design in Step 2 below.
   - The file is located at assignment1/gold/rastBBox_fix.cpp
   - There are two functions which are missing code:
     - rastBBox_stest_fix // the sample test function
     - rastBBox_bbox_fix // the bounding box function
   - There are some comments in the C++ file, but the pseudo code in Section 3.3 should be your main reference. Keep in mind that it is all in fixed point.
   - As mentioned, we will reuse the code you write here as a gold reference model that is accessed directly from the SystemVerilog testbench. Unfortunately, SystemVerilog does not speak C++, only C. That is, SystemVerilog Direct Programming

Interface (DPI) only supports C language, thus the functions that the testbench uses can only be C. Bottom line: for your code, use only C.

- Once you are satisfied with your addition, compile from assignment1 directory:
  - $: make clean comp_gold
- To run the gold model with a test vector:
  - $: ./rast_gold out.ppm $EE271_VECT/vec_271_01_sv.dat
- The output can be viewed with (in graphical environment only)
  - $: gimp out.ppm &
- To compare the output with the corresponding reference image:
  - $: diff out.ppm $EE271_VECT/vec_271_01_sv_ref.ppm
- To compare the output with the reference image in MATLAB (not necessary but often useful):
  - Start MATLAB:
    $: module load matlab
    (You may need to load Matlab before running the setup script.)
    $: matlab
  - Then the following commands should execute in MATLAB command line:
    >A = imread('output.ppm');
    >B = imread('/usr/class/ee271/project/vect/vec_271_01_sv_ref.ppm');
    >C = abs(A - B);
    >find(C) $ this shows positions of non-zero values in matix C
    >imshow(C*255) $ this shows a image of matix C
- There are some other test vectors and reference images in $EE271_VECT folder which you may (or should) try in order to test your code.

2. **Fill in C code for BBox and Sample Test:**

- Once you have completed the C++ code, add it to the C code used for verification. This should largely be a copy and paste unless you leaned heavily on C++ conventions.
- The file is located at /gold/rastBBox_fix_sv.c
- There are three functions missing code and four blanks to copy paste:
  - rastBBox_bbox_check //bounding box check function
  - rastBBox_stest_check //sample test check
  - rastBBox_check //check the uPoly rasterization

5.1.2 Submission

In this assignment, you need to submit the project folder containing code of C++ gold model, which shoule 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 students SUID on the first line, first students name on the second line, first students email on the third line, second students SUID on the fourth line, second students 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:
$: /usr/class/ee271/project/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 mis-formatted your submission. Submit your assignment1.tar.gz on Canvas.

The DUE DATE of this assignment is 02/24.

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

5.2 Assignment 2

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.

5.2.1 Instructions

We provide starting code for this part (/usr/class/ee271/project/assignment2), but only 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).

1. **Implement bbox module:**

   The bbox module is located at assignment2/rtl/bbox.vp. 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 assignment2/rtl/test_iterator.vp. 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 Genesis parameter "ModifiedFSM" and two branches *if ($mod_FSM eq 'NO')*. 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.

3. **Implement sample test module:**

   The sample test module is located at assignment2/rtl/sampletest.vp. Your job is to produce the value for hit_valid_R16H signal, which indicates whether a sample lies inside the micropolygon. The signal is high if the input sample is valid and the sample is inside the micropolygon (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. **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"
- To compare the output with the corresponding reference image:
  - $: diff sv_out.ppm $EE271_VECT/vec_271_01_sv_short_ref.ppm
- 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 the `diff` command tells you that your output image is different from the reference image, you are now entering the DEBUG phase of the logic design. Here are a few things which can help you debug rtl code.
  - First of all, it is a good start to understand how the verification works in this framework. The main body of verification is in verif/testbench.vp.
  - Then take a look at your output image (assuming one was generated) and compare with the reference image. Sometimes, it is very useful to get visual clues of where goes wrong.
  - There are log files in the sb_log folder generated by the scoreboard modules (verif/*_sb.vp). If there are any wrong signals inside the pipeline which can be trapped by the C functions you just implemented in assignment 1, they will be recorded in the sb_log files. **It is very helpful to take a look at how scoreboards are written in case you want to modify them to help debug.**
  - You can activate the monitors in the testbench.vp by making the 'if' loop always true at line 282. (We turn it off by default because otherwise simulating long test vectors generate large log files which exceed your disk quota.) After turning them on, run simulation again. The log files are placed in the mon_log folder generated by the monitor modules (verif/*_monitor.vp). The monitors are designed to record key signal values in the pipeline. **It is very helpful to take a look at how monitors are written in case you want to modify them to help debug.**
  - Another important method for debugging is to watch the waveforms of signals, like watching variables when debugging software. To dump a waveform database during simulation, use the command `make run_wave` instead of `make run`. After the simulation completes, you can open the waveform file 'vcdplus.vpd' with Synopsys's DVE. Start DVE from the command line (graphical environment needed here) with the command:
    * $: ln -s /usr/lib/x86_64-linux-gnu/libtinfo.so libtermcap.so.2
    * $: setenv LD_LIBRARY_PATH your_assignment2_directory
    * $: dve -full64

    For more usage about DVE, please check DVE User Guide.
- After you clear all the bugs in RTL, it is time to take a look at the performance of your design. You should be able see two performance numbers 'uPoly/cycle' and

'cycle/uPoly' at the end of the simulation, which are literally the average number of micropolygons 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 polygons rather than 'micropolygons', like vec_271_00_sv.dat, your uPoly/-cycle must be lower than those from micropolygons test vectors. You should be able to get more real numbers from running full size micropolygon rendered image inputs, i.e. vec_271_01_sv.dat, vec_271_02_sv.dat and vec_271_04_sv.dat.

- If you implement 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 Genesis parameter 'ModifiedFSM' to 'YES' in test_iterator and run verification (and later synthesis) again. Another very handy usage of Genesis is to overwrite parameter value in runtime. The following example shows how to change 'ModifiedFSM' value using make command:
  - $: make clean run GENESIS_PARAMS="top_rast.rast.test_iterator.ModifiedFSM=YES"

  When using 'GENESIS_PARAMS', make sure that you run 'clean' rule first. Otherwise, Genesis won't generate code again for you based on the dependency defined in Makefile. After passing the verification, you can check the performance of different FSM design.

5. **Run synthesis:**

- Run sythesis from Assignment2's directory:
  - $: ln -s /usr/lib/x86_64-linux-gnu/libtiff.so.5.2.0 libtiff.so.3
  - $: setenv LD_LIBRARY_PATH your_assignment2_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.

6. **Retiming:**

   In Assignment 2, you only need to have a functionally correct design, that can run synthesize 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 0.5ns, 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=0.5 ...`
   - 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 Genesis parameters of rtl/rast.vp module. Run synthesis again to see the results. Recall that any Genesis parameters can be overwritten in runtime. It might be beneficial when you are doing experiments. Here is an example of changing pipeline depths of bbox from make command:
     - $: make clean run_dc GENESIS_PARAMS="top_rast.rast.PipesBox=5"

     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 (0.8ns) without any timing violations. Make sure the area of your chip doesn't exceed 40000(um^2), which is the target area in synthesis. (If you put too many pipe stages, the registers will cost area.)

5.2.2 Submission

For this part of the assignment, you should submit the correct design with bbox, sampletest, and test iterator FSM implemented. You may have a good number of working design configurations, but please adjust your RTL and Genesis 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 Assignment 3 performance competition.

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

We may give extra credit if your clock period is very low under the area constraint.

The DUE date of this assignment is 3/10.

The submission requirement is the same as Assignment 1 (see Section 5.1.2), expect 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 Canvas.

5.3 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 micropolygons onto a 16xMSAA buffer at a rate of 100 Hz. This corresponds to a throughput of 1 micropolygon every 2ns (i.e. 2ns/uPoly or 0.5uPoly/ns).

5.3.1 Instructions

The way to estimate the performance of your design is to take the 'cycle/uPoly' 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/uPoly).

Your goal here is to build a design that matches the leading architect in performance (2ns/uPoly) 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 rasterizaion 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 over-clocking, 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 micropolygon 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.

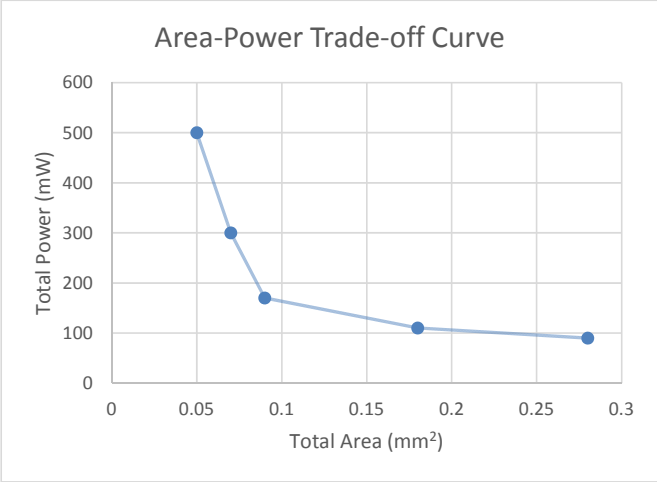

Figure 13: Area versus Power Trade-off Curve

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 micropolygons during the bounding box stage. While halted it may be useful to advance a micropolygon 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. It is possible that one of these other methods might help you reach the throughput you need. One algorithm you might try is given in this work.
- **Quadrilaterals**: Instead of rendering micropolygons triangle by triangle, you can do it in quads. By detecting and combining two adjacent triangle pairs into a quad in the front end, you can make bounding boxes and apply sample tests on these quads. There are some explanations and pseudo code in Appendix A.
- **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 micropolygons per cycle. Be sure to 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.

5.3.2 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 Canvas.

The DUE date of this assignment is 3/24 NOON.

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 Canvas.

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.

For the report, a template will be provided to you by the course assistants. You should utilize this template to assist in the speedy evaluation of your work. The write-up should detail your changes, the process used in evaluating these changes, and the speed-up that each change provided. You should also report your final throughput, your clock speed, your total power, and your total area. You should also use the report to tell us what you found interesting what you found difficult while making these optimization. Extra credit will be provided to those who go beyond to implement both creative and effective optimizations.

Appendices

A Rasterization for Quadrilaterals (Triangle-Pairs)

A.1 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 pipelines micropolygon 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 a 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 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

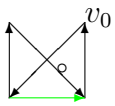

Figure 14: Sample Case 1

- `( !b4 && !b3 &&  b2 && !b1 && !b0 )`
  given in Figure 15
- `( !b4 && !b3 &&  b2 &&  b1 &&  b0 )`
  Given in Figure 16

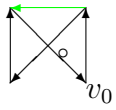

Figure 15: Sample Case 2

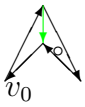

Figure 16: Sample Case 3

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

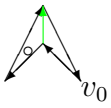

Figure 17: Sample Case 4

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

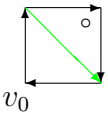

Figure 18: Sample Case 5

- `(  b4 && !b3 && !b2 && !b1 &&  b0 )`
  given in Figure 19
- `(  b4 &&  b3 && !b2 && !b1 && !b0 )`
  given in Figure 20
- `(  b4 &&  b3 && !b2 &&  b1 &&  b0 )`
  given in Figure 21

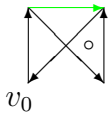

Figure 19: Sample Case 6

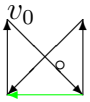

Figure 20: Sample Case 7

Figure 21: Sample Case 8

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

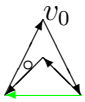

Figure 22: Sample Case 9

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

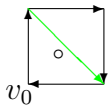

Figure 23: Sample Case 10

A.2 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 BBox
  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 BBox 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 uPoly
  // 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,y_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 poly 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 )); }
```