Architectural Specification – 03/12/2021

# Introduction

The instruction needs to be compliant with section 14.6.1 of the OpenGL 4.6 Core Specification. The polygon rasterization stage just describes the inner loop of rasterisation. However, the process described can just be repeated for each triangle in the scene.

The instruction aims to implement both loops of the rasterisation algorithm. This reduces flexibility of the block but can provide massive performance gain by accelerating the outer loop in hardware. I imagine that a great deal of the design work done to accelerate the out loop will be to design high throughput data transfer of triangles from memory to the unit.

There are three ways of representing triangles given by OpenGL 4.6, a triangle fan, triangle strip and a list of independent triangles.

Triangle Strips

A triangle strip is a series of triangles connected along shared edges, and is specified with mode TRIANGLE\_STRIP. In this case, the first three vertices define the first triangle (and their order is significant). Each subsequent vertex defines a new triangle using that point along with two vertices from the previous triangle. If fewer than three vertices are specified, no primitive is produced. See figure 10.2. The required state consists of a flag indicating if the first triangle has been completed, two stored processed vertices (called vertex A and vertex B), and a one bit pointer indicating which stored vertex will be replaced with the next vertex. When a series of vertices are transferred to the GL, the pointer is initialized to point to vertex A. Each successive vertex toggles the pointer. Therefore, the first vertex is stored as vertex A, the second stored as vertex B, the third stored as vertex A, and so on. Any vertex after the second one sent forms a triangle from vertex A, vertex B, and the current vertex (in that order).

Triangle Fans

A triangle fan is specified with mode TRIANGLE\_FAN, and is the same as a triangle strip with one exception: each vertex after the first always replaces vertex B of the two stored vertices.

Separate Triangles

Separate triangles are specified with mode TRIANGLES. In this case, the 3i + 1st, 3i + 2nd, and 3i + 3rd vertices (in that order) determine a triangle for each i = 0, 1, . . . , n − 1, where there are 3n + k vertices drawn. k is either 0, 1, or 2; if k is not zero, the final k vertices are ignored. For each triangle, vertex A is vertex 3i and vertex B is vertex 3i + 1. Otherwise, separate triangles are the same as a triangle strip.

Diagram

Description automatically generated

A polygon is defined by the specification as: A polygon results from a triangle arising from a triangle strip, triangle fan, or series of separate triangles.

I will therefore assume that the rasterization block will be passed a list of polygons (individual triangles). This list will be produced by the primitive assembly stage implemented in software which will parse a set of triangles, triangle strips or triangle fans into a list of polygons.

# Instruction Format

Takes a pointer to, pointer to index array, orientation of the triangles should be adjustable clockwise or anti-clockwise winding order, anti-clockwise winding order is default.

rasteriseTriangle riI, riV, riF, roF, roD

## Inputs

What do other GPU’s take as input here to make them OpenGL compliant?

* riI: Pointer to Index array of ints stored in GPR
* riV: Pointer to Vertex array stored in GPR
* riF: flags register

### Vertex array

Let the number of supported generic vertex attributes (the value of MAX\_VERTEX\_- ATTRIBS) be n. (section 10.6)

Array of 4 element generic vertex attributes, the first element always being the vertex coordinates, the array will therefore have the following format:

Diagram

Description automatically generated

## Outputs

What do other GPU’s produce as output here to make them OpenGL compliant?

* roF: Pointer to Fragment array written to GPR (what is the format of the fragment array in OpenGL??)
* roD: Filled Depth buffer (optional) written to GPR

# Steps

The steps detailed in the spec are as follows, brackets indicate flags that need to be set for each stage:

## Interpretation of polygons (face, mode)

|  |  |  |
| --- | --- | --- |
| face | Mode | Note |
| FRONT\_AND\_BACK | POINT | Vertices are rasterised as points |
| FRONT\_AND\_BACK | LINE | Polygon edges are rasterised as line segments |
| FRONT\_AND\_BACK | FILL | Rasterise polygon as described in following sections |

For the purposes of the rasteriseTriangles instruction Mode is always set to FILL. However, I will leave in the parameter to allow building a line and point rasteriser accelerators as an extension to the project.

## Determine if the triangle is front or back facing (winding order) - always done)

* 1. Computed based on the (unclipped/clipped) polygon’s area computed in window coordinates (computed based on a customisable winding order)

## Face culling (mode, enable) – (can be enabled or disabled)(by default back faces are culled)

* 1. Enable = 1, Mode = BACK – front facing rasterised, back facing not rasterised
  2. Enable = 1, Mode = FRONT – front facing not rasterised, back facing rasterised
  3. Enable = 1, Mode = FRONT\_AND\_BACK – nothing rasterised
  4. Enable = 0, everything rasterised

## Orthographic Projection

* 1. Just take x and y coordinates of triangle

## Point Sampling - Determining which fragments are produced by rasterisation of a polygon

* 1. Fragment centers that lie inside the projected triangle are rasterised
  2. If a fragment center lies exactly on an edge
     1. if two polygons lie on either side of a common edge (with identical endpoints) on which a fragment center lies,
     2. then exactly one of the polygons results in the production of the fragment during rasterization.
        1. I need to define which triangle is used

## Interpolation of arbitrary vertex attributes and z (noperspecitve, flat, provokeMode)

### Triangles

* 1. Define barycentric coordinates for the triangle
     1. a,b,c defined in the range [0,1] where a+b+c = 1
     2. these uniquely specify and point p in or on the triangles boundary as p = apa + bpb + cpc
     3. pa/b/c are the vertices of the triangle
     4. a,b,c can be found as:

Where A(l,m,n) is the area of the triangle with vertices l,m,n

* 1. The data associated with a fragment must be sampled at the fragments centre
  2. For a given attribute fn associated with a vertex pn, f can be calculated using:
  3. Where w\_a,w\_b,w\_c are the clip coordinates of pa,pb,pc
  4. Depth however is computed differently using the following formula

|  |  |  |
| --- | --- | --- |
| noperspective | flat | Result |
| 0 | 0 | Interpolation using equation with f |
| 1 | 0 | Interpolation of all attributes done with the depth interpolation equation |
| 0 | 1 | No interpolation done, attribute is assigned to be the same as that of the **provoking vertex** of the triangle |
| 1 | 1 | n/a |

|  |  |  |  |
| --- | --- | --- | --- |
| ProvokeMode | Triangle | Triangle Fan | Triangle Strip |
| FIRST\_VERTEX\_CONVENTION | i | i+1 | i |
| LAST\_VERTEX\_CONVENTION | 3i | i+2 | i+2 |

For calculating the provoking vertex of the ith polygon in the scene

### Polygon with >3 edges

Like those produced by clipping a triangle (makes a quad). These polygons will be turned into a set of triangles without adding vertices before entering the rasteriser. This means we only need to do triangle rasterisation.

Or can do generalised interpolation which is:

Where f is value of the attribute at vertex i and a is the barycentric coordinate corresponding to vertex i.

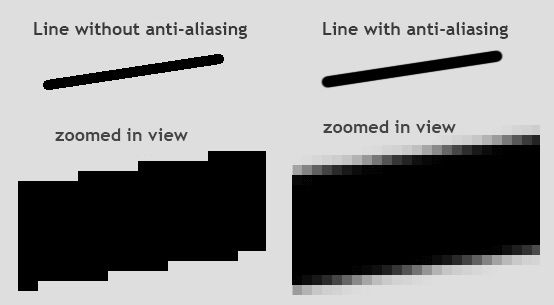
Seems much harder to do in hardware though.

## Anti-Aliasing

For each fragment produced a coverage value is calculated:

The above f calculated using interpolation is multiplied by the coverage to produce the actual fragment attribute value f’ :

This prevents jaggies from forming and makes likes seem much smoother to the human eye, see the figure below:



<https://www.definition.net/image.ashx?img=https://www.definition.net/upload/media/1_mpzptseunyptpkpvzvctyyrqxxqzxbzz.png>