The Rendering Pipeline

GAME 300

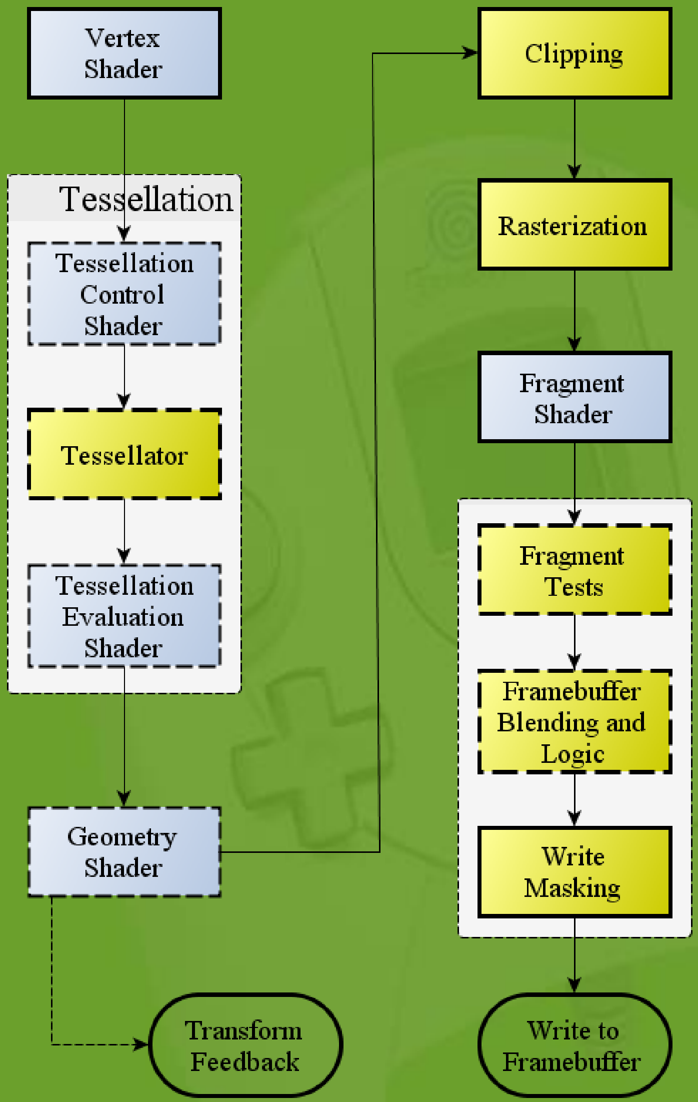
James Dupuis

Objectives

- Learn about:
  - The flow of the Rendering Pipeline
    - Overview of the basic stages
    - Faces
    - Culling vs Clipping

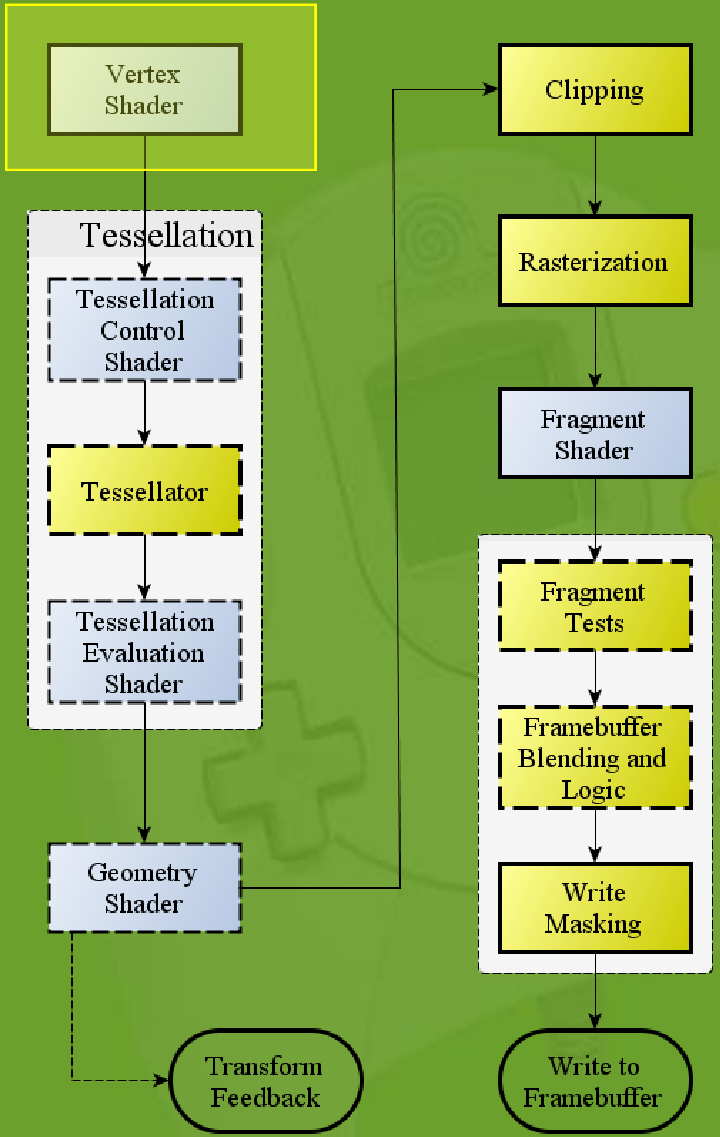
Behind the curtain

Render Pipeline



- The rendering pipeline is a series of shaders and processes which execute on the GPU.
  - Run individually as steps
  - Some are programmable (blue)
  - Some are Fixed (yellow)
    - Means OpenGL does it’s thing and we don’t have access to modify what it does.
    - Usually a process which runs and the settings for how that process functions is set in a previous shader or through our OpenGL States.
  - Some are required (solid borders)
  - Divided into Front end and Back End (columns)
  - The Framebuffer is the end goal where all the data gets sent to representing what will be displayed on screen.

Render Pipeline Flow



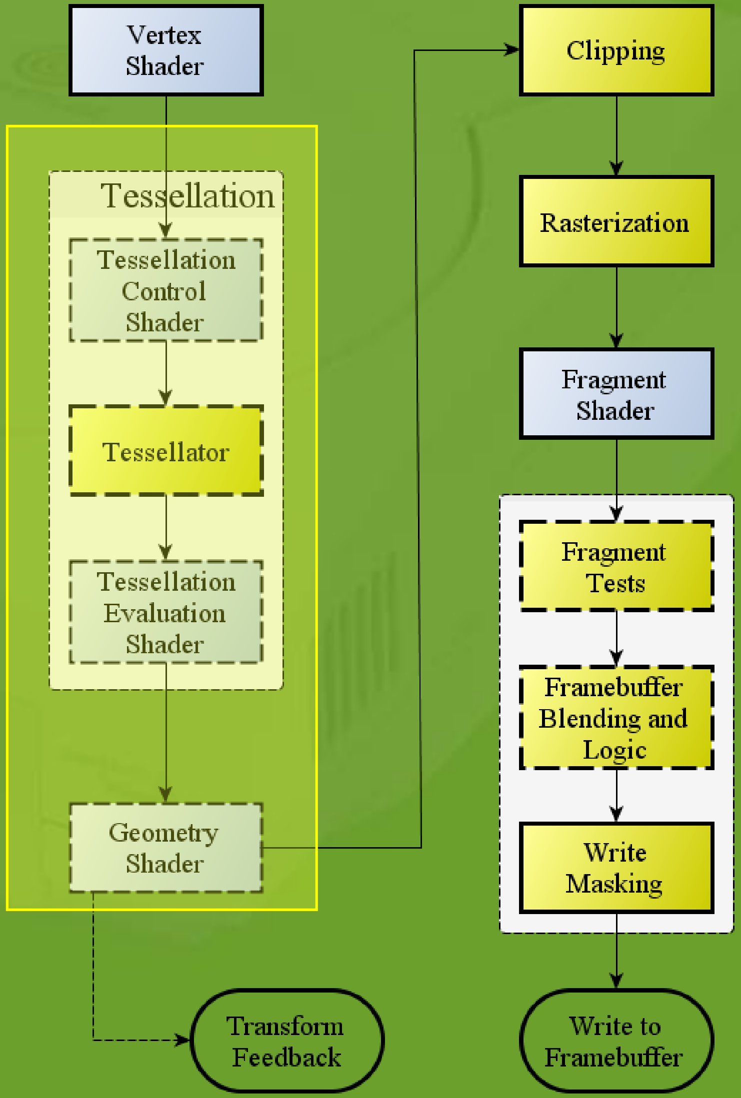
- **Vertex Shader**:
  - Our main entry point into the render pipeline
  - Processes the **WHERE**
    - where each vertex should be located
  - **Requirement** of any graphical output

- Example:
  - glVertex3f(1.0f,0.5f,0.5f);

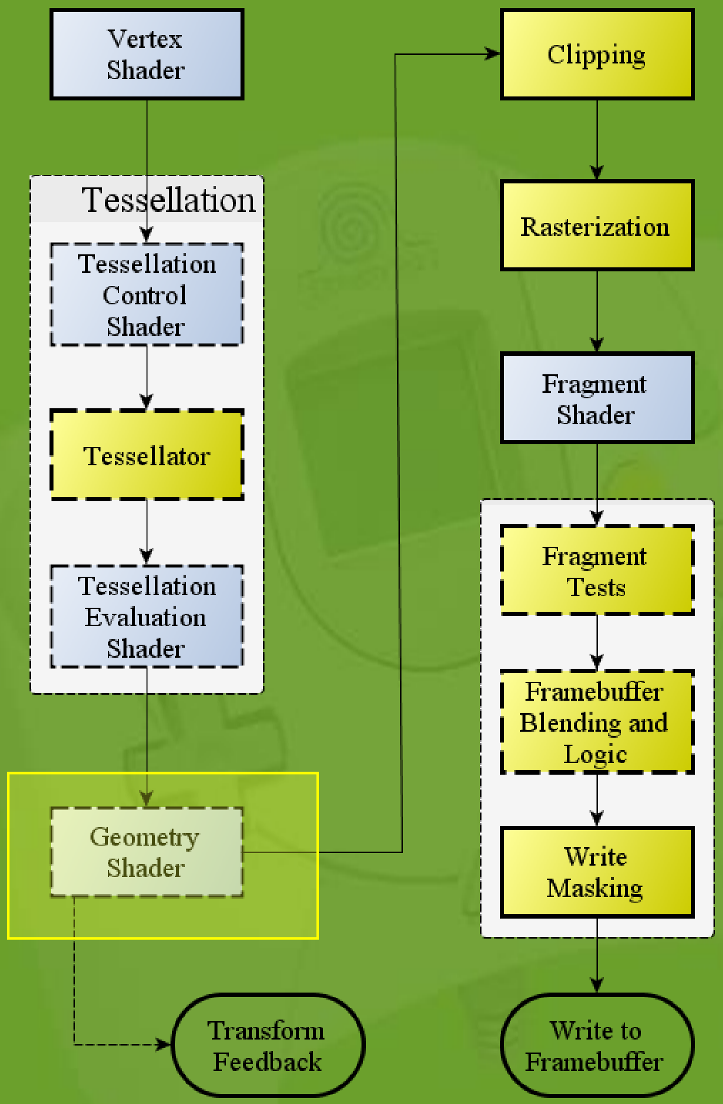
Render Pipeline Flow (Tesselation)

- **Tessellation Shaders:**
  - Performs supplemental vertex modifications (Where / How)
    - Is a process of the rendering pipeline that is **optional**
  - Has 3 parts, 2 are programmable and one is fixed
  - Happens directly after the vertex shader stages
    - before the fragment shader.
  - It's job is to take a larger "path" or group of polygons/ model and split it into smaller primitives.

- Example:
  - tessellatedtri.exe



Render Pipeline Flow (Geometry)



- **Geometry Shaders:**
  - Final shader stage of front end.
    - before the fragment shader but after the vertex/ Tessellation shaders.
  - Executes individually for every primitive processed.
  - Can access data for each vertice.
  - Can switch primitive modes.
  - Take one type of primitive as input and convert it out to another type
    - Change triangles to lines or dots to triangles.

FACES!

- Each polygon that we render has what is known as a face.
  - This is essentially the inside filled in area of colour (white in our example)

Note:

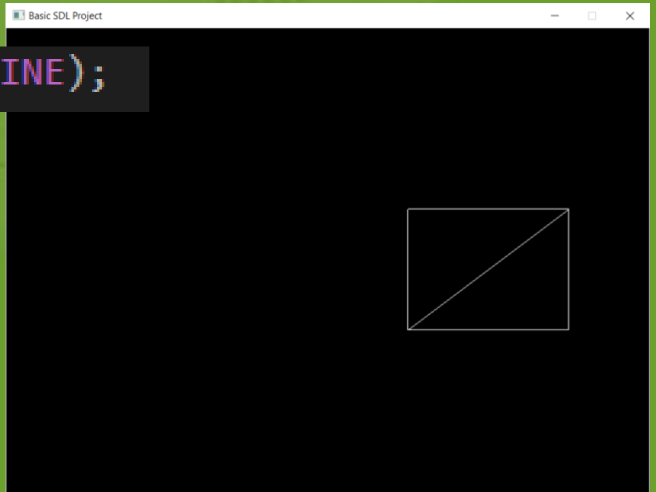
Lines and points do not have faces, the face is the way the individual triangle is rendered.


```
// draw here...
glBegin(GL_TRIANGLE_STRIP);
glVertex3f(0.25f, 0.25f, 1.0f);
glVertex3f(0.25f, -0.25f, 1.0f);
glVertex3f(0.75f, 0.25f, 1.0f);
glVertex3f(0.75f, -.25f, 1.0f);
glEnd();
```

- When we render to the screen we visually see a single face.
- There is, however, a second face to each polygon.
  - FRONT & BACK
- We can alter the way all faces are rendered before rasterization by using the following function:

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
```

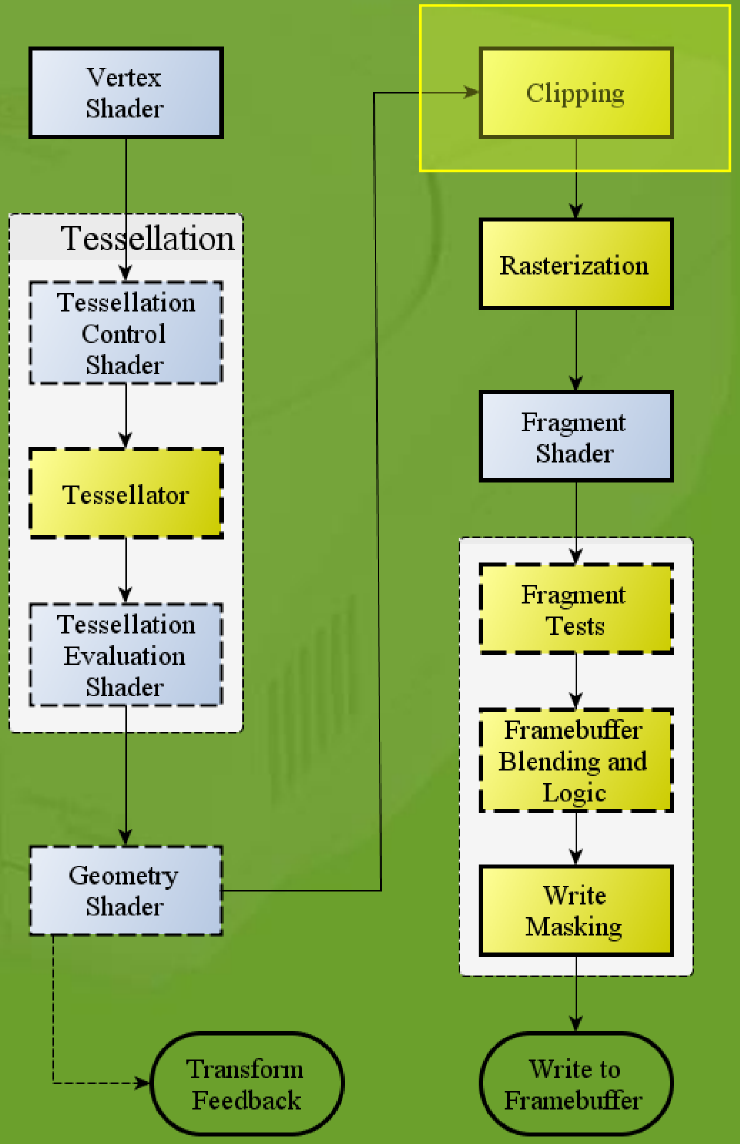
  - The first param defines which face we want to modify.
    - We can supply it GL_FRONT, GL_BACK or GL_FRONT_AND_BACK.
  - The second param specifies the mode we want it to render in.
    - Note: here we use singular value of GL_LINE and not LINES like glBegin().
    - We can supply this GL_POINT, GL_LINE, or GL_FILL.



Render Pipeline Flow (Back End)

- **Primitive Assembly:**
  - In the Clipping Section
  - Not programmable (fixed)
  - First stage of the Back End
  - Essentially just a stage which retrieves all the vertices and data from the front end stages and assembles them into primitive shapes

- **Clipping Stage:**
  - Removes vertices that won't show up on screen area from being further processed.
  - Produces primitives with coordinates between -1.0f-> 1.0f for each axis defining the screen space
    - aka normalized coordinates
  - Note that the z axis is clipped from 0.0f->1.0f as the center is where the screen is.
  - In terms of the 2d screen 0,0 is the absolute center of the screen.

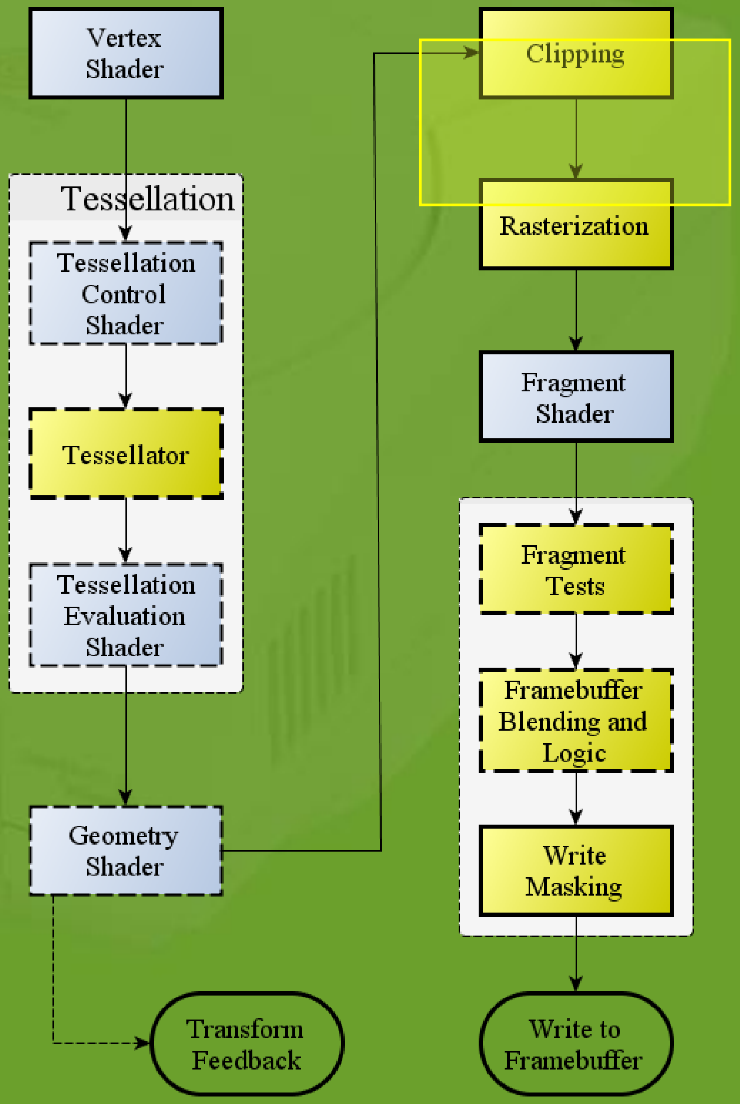


CULLING & CLIPPING

- Culling is the concept of removing information from being processed and rendered if it does not fit within the restrictions imposed.
  - Clipping is culling strictly the dimensions of the scene.
  - Culling is not restricted solely to the screen dimensions
  - We can cull items that meet a certain criteria like culling the back face of a polygon
    - This is a simple optimization many games make to ensure they are not spending processing time working on something that will *hopefully* never be visible to the player.

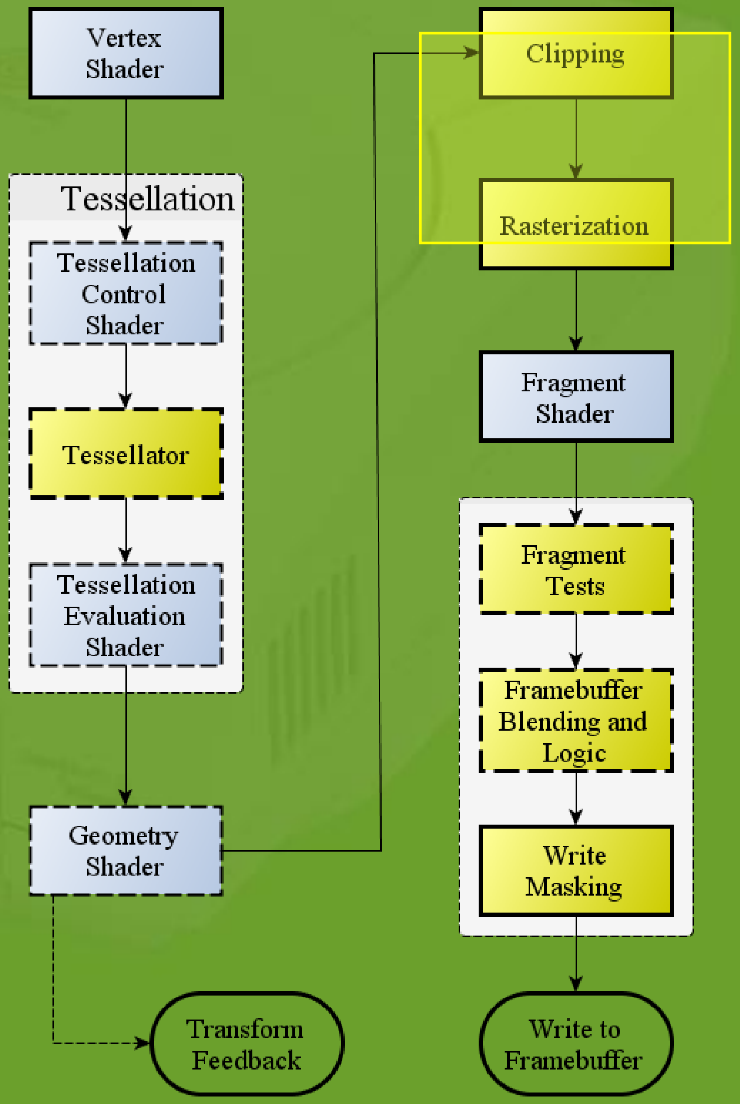
Render Pipeline Flow (Viewport)

- **Viewport Transformation:**
  - Turns the clipped -1.0f -> 1.0f coordinates into a flat screen representation
  - Performed per pixel
    - Screen width-1 x screen height-1 (window coordinates)
  - Also need to take into account the z-depth
    - How far into the screen are we seeing in relation to the width/height coordinates?
    - More on how this is handled later



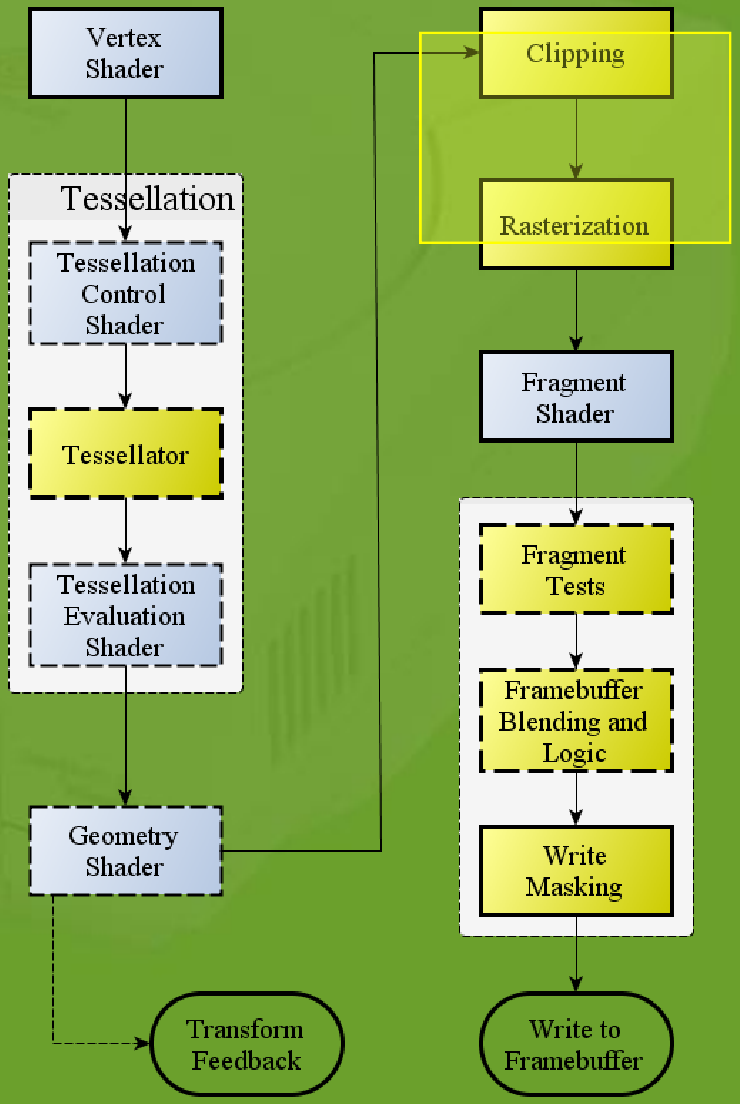
Render Pipeline Flow (Culling)

- **Culling (Optional):**
  - Each polygon comprised of triangles has faces
  - Lines and points don't have areas, hence also no faces.
    - even if your lines all intersect and form a triangle, that's not the same as a primitive Triangle.
    - same as a side of a triangle
  - The decision making process of rendering is based off which way the polygon is facing.
    - Don't need to render something that is facing away from the viewport / viewer, since it should be hidden.
    - Sometimes this is the desired effect (some cell shaders use this as a trick)
- Example:
  - gsculling.exe



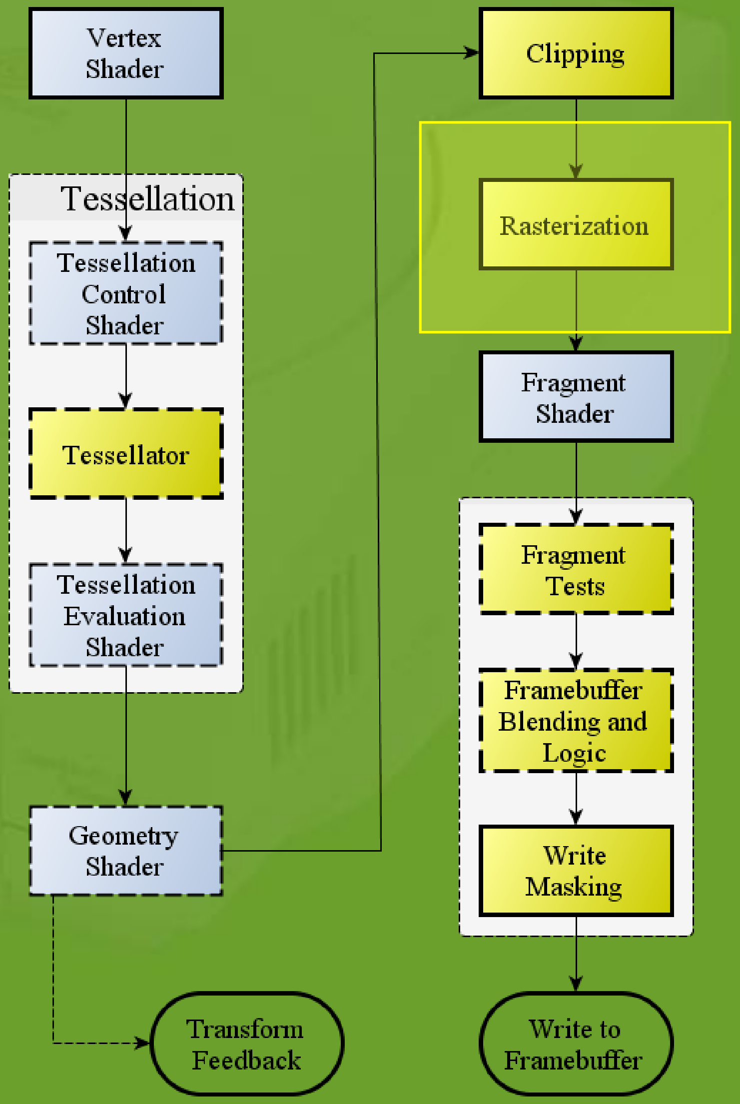
Render Pipeline Flow (Culling II)

- **Culling (Optional):**
  - Default without culling enabled, both sides are rendered on all tris
    - To enable, in program use **glEnable(GL_CULL_FACE);** in program code
  - Once enabled you must define which triangles are being culled using:
    - **glCullFace(GL_FRONT); //GL_BACK** or **GL_FRONT_AND_BACK**
  - Determines front from back by using the first two edges of its perimeters and calculate the cross product.
    - Positive result value indicates a front face
    - Negative value indicates a back face.



Render Pipeline Flow (Rasterizer)

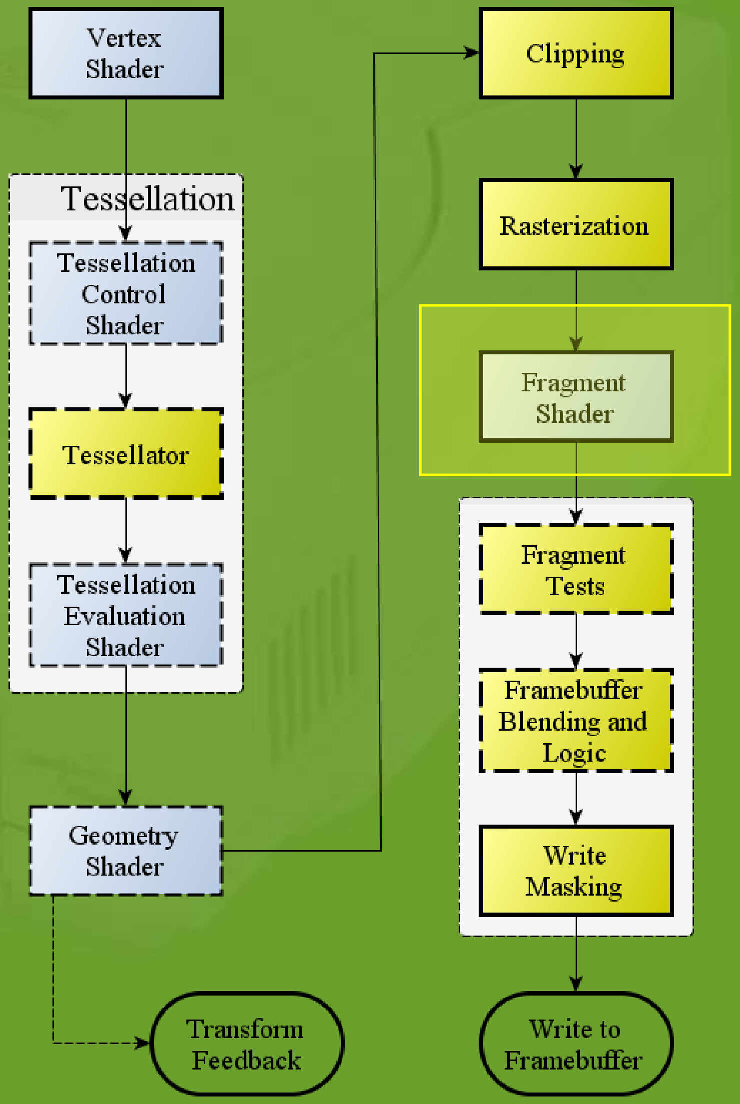
- **Rasterizer:**
  - Collects all the data from the previous mentioned shaders and creates new defined areas. (WHAT)
    - Turns the clipped and culled viewport coordinates into fragments which will tell the fragment shader what it needs to color.
  - Determines areas within the vertices of primitives defined that need to be coloured
  - Forwards all this data to the fragment shader for processing.



Render Pipeline Flow (Fragment)

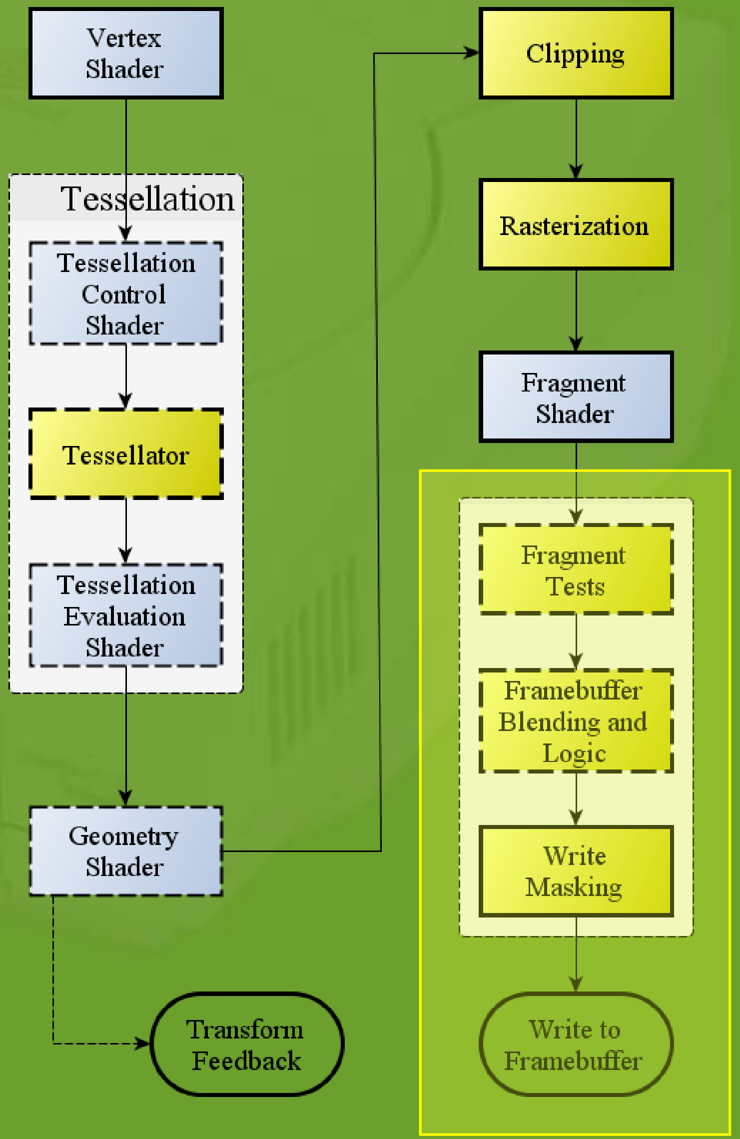
- **Fragment shader** (HOW?):
  - Colours fragments of the screen
  - Final **programmable** shader of the entire pipeline.
  - Sets the colour to each fragment defined by the previous rasterization procedures.
  - sends data to the **framebuffer**
  - Communication to the fragment Shader is from whatever programmatic shader preceded it.

- Example:
  - glColor4f(1.0f, 0.0f, 0.0f, 1.0f);
  - fragcolorfrompos.exe



Frame Buffers

- Last stage of the pipeline
  - Defines the When
    - Rendered to screen when applied
  - Essentially the final image which will be rendered to the screen.
    - Or a part of it if multiple shader programs are being used.
- Where the data all flows into
  - stores state data about the fragments



Frame Buffer Testing

- **Scissor test:**
  - Essentially a masking procedure
  - An area is defined and it only renders fragments from within or external to the area depending on the options provided.
- **Stencil test:**
  - Does a check against a stored stencil in a "stencil buffer" compared to the fragments coming in to determine what to render.
    - Example: you could load in an image and as a stencil to mask out areas of the screen for a UI effect.
- **Depth test:**
  - Determine whether a fragment is rendered based on it's z axis/ depth
  - Can specify not to render anything within the first 0.05f section so that the viewport isn't blocked by close objects
  - Can specify how far into the distance items can be rendered.
  - A 0.0f->1.0f value for depth

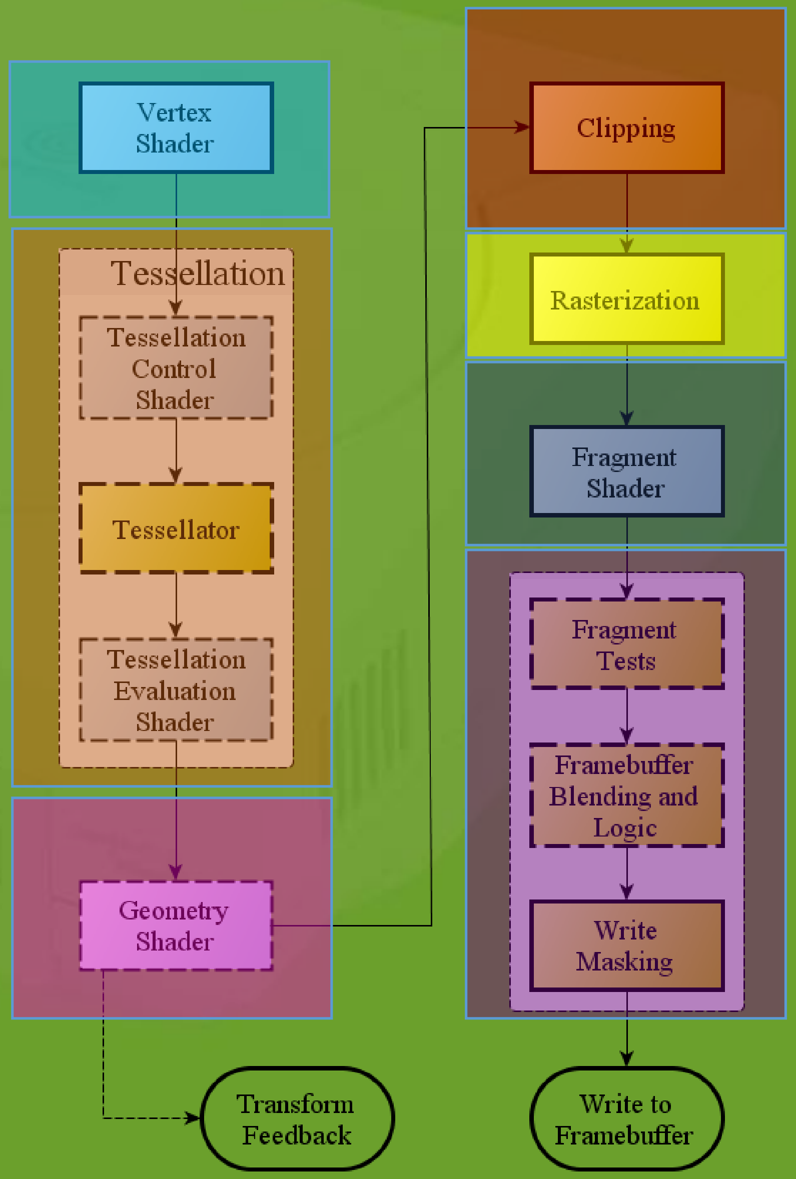
Maverick of Shaders

- 1 Remaining Shader type, the **Compute Shader**:
  - Useful for AI programming (flocking and pathfinding calculations)
  - Not part of the graphics pipeline
  - Used to support the CPU with tasks that are simple but processed in higher amounts.
  - Created in a separate shader program called a workgroup
    - Defined the same as basic shaders, but used to process data not graphics

- Example: csflocking.exe

Demo Exercise (LAB NEXT DAY?)

- Group up for each major shader stage
  - Vertex Shader (2)
    - glBegin()
      - glVertex3f()
    - glEnd()
  - ~~Tessellation (0)~~
  - Geometry Shader (1)
    - glPolygonMode(GL_FRONT_AND_BACK, [type]);
  - Rasterization(1)
    - glPointSize(40.0f);
    - glLineWidth(2.0f);
  - ~~Clipping (0)~~
  - Fragment Shader (2)
    - glColour4f(R,G,B,A);
  - Frame Buffer (0)



Summary

- Learned about the different pieces of a shader program
  - How they all work together to create simple or complex scenes
  - Vertex Shaders
  - Tessellation
  - Geometry Shaders
  - Assembly/ Clipping / Culling/ Viewport
  - Rasterization
  - Fragment Shader
  - Frame Buffer
    - Tests
  - Compute Shaders