The Purpose and Utility of Geometry Shaders

Diogo Oliveira Almeida, dio.almeida@ua.pt

***Abstract*—Geometry shaders were introduced in Direct3D 10 and OpenGL 3.2 back in 2009 at a time when GPUs and their technologies were far from being as efficient and powerful as in today's days.**

These shaders can accomplish tasks that neither the vertex nor pixel shaders can perform as efficiently, they can also have dynamic outputs, meaning that the size of the output may not be known in advance.

They act as an intermediary between the vertex shader and the fragment shader, applying necessary complex transformations such as amplification of the geometry from as much as a single vertex received from vertex shader.

We will explain in further detail these geometry shaders as well as its pros and cons in the current visual computation world and demonstrate a simple but illustrative tutorial on how to work with them that will demonstrate the points of the discussion. The tutorial presented in this work was made with Python using the game engine pygame and the PyOpenGL library to bind Python with OpenGL.

***Index Terms*—Visual computing, geometry shader, Level-of-Detail**

I. Introduction

APART from the vertex and fragment shaders there is a third type of shader that sits between those two, it is called the geometry shader, it is optional and it makes possible the creation of new geometry on the fly by using the output of the vertex shader as its input.

Whatever geometry shaders can do, can also be accomplished through other methods, what makes them so powerful and useful is the simplicity of the syntax and the small amount of input data needed that is sent from the CPU to the GPU, thus, facilitating development by its simplicity and reducing bandwidth usage between the CPU and GPU saving it for other running applications.

Unlike the vertex shader (which allows the manipulation of input vertices) and the fragment shader (which turns the input vertices into pixels in the screen) which work at a vertex and fragment level respectively, the geometry shader works with primitives and is capable of drawing geometries that are completely different from the original ones, it can create different amounts of them by taking the original primitive (set of vertices) and transforming them as needed before the next shader stage, it works on all kinds of objects, no matter their complexity, it is limited though to one output type, not allowing mixtures.

One of the first original uses for the geometry shader was for LOD (Level-of-Detail), meaning that we could change the geometry of an object depending on how distant a player was from said object. In the tutorial we look at a cylinder as an example, far away players don't need to see such perfect cylinders as it doesn't (or almost doesn't) make a difference visually to the player. This means that we could generate a lower detailed cylinder, thus having less vertex and therefore less memory is needed to display that object when rendering it. In a scenario where we have multiple objects that can take advantage of this it could make a big difference, for example, let us say that we have a forest with 1000 tress in our game, and that a single tree has 1000 vertex, for the scenario to be rendered we would need 1 000 000 vertexes in total just to render the forest. If the player were to move considerably far away from the forest it would make no sense to keep those trees with the same level of detail as if the player was right next to them. In this scenario a geometry shader could be used to reduce the number of vertexes in those trees. Even if the vertexes in the trees were just reduced by 20% it would mean that a total of 200 000 vertexes were excluded from the rendering.

However, geometry shaders are not all sunshine and rainbows, despite their initial usage for LOD they fell out of use as they proved to have performance problems in general and specially with some hardware manufacturers in specific, certain output limitations and, with the evolution of computer graphics new methods and technologies could do the same as geometry shaders but better and more efficiently, despite this they are still befitting for dynamic particle systems, fur generation amongst other usages.

II. Geometry Shader Tutorial

As mentioned before the geometry shader supports amplification and de-amplification, and it operates by receiving an input primitive outputting one or more primitives or it can entirely discard the input geometry. It executes its code for every primitive generated previously in the pipeline. In our case we use just **one** vertex per cylinder, for a total of 8 vertex as input to the vertex shader and geometry shader and we will create completely different geometries from the original ones.

The geometry shader can also calculate a lot more information than the vertex shader because it only has access to one vertex at a time whereas the geometry shader has access to all the vertices in the primitive. We will explore the results further ahead.

We would like to point out that we ran into a problem when creating our cylinders, we ask that you ignore the cylinders length in comparison with its "further away" counterparts (smaller cylinders). This is due to the previously mentioned output limitations, as geometry shaders have two major and competing limitations on their outputs. The first limitation dictates a maximum number of vertices that a single invocation of a geometry shader can output and the second

dictates the total maximum number of output components that a single invocation of a geometry shader can output. Despite this setback the images used as examples are still clear.

In this tutorial we start by instancing a pygame window and initialize the VBO (Vertex Buffer Object) that receives the following geometry:

```
geometry = [
    # Px, Py, Pz, Cx, Cy, Cz, n, s
    [-1.8, 1, 0, 0.76, 0.24, 0.24, 32, 1.4],  # 1
    [-0.5, 1, 0, 0.82, 0.25, 0.80, 28, 1.2],  # 2
    [0.6, 1, 0, 0.29, 0.63, 0.86, 24, 1],  # 3
    [1.6, 1, 0, 0.70, 0.59, 0.80, 20, 0.8],  # 4
    [-1.8, -1.4, 0, 0.39, 0.92, 0.16, 16, 0.6],  # 5
    [-0.5, -1.4, 0, 0.27, 0.74, 0.51, 12, 0.4],  # 6
    [0.6, -1.4, 0, 0.78, 0.43, 0.86, 10, 0.2],  # 7
    [1.6, -1.4, 0, 0.24, 0.24, 0.24, 6, 0.05],  # 8
]
```

Fig. 1-Data used for the vertex buffer to create our geometries. This will be used as the input of the vertex shader containing all necessary data.

Looking at Fig.1, each element of the *geometry* list variable is a different vertex, in turn, each element of the sub lists are the vertexes coordinates, color, number of vertices to be rendered by the geometry shader and the scale of the geometry respectively. Once the VBO is initialized we read and compile the vertex, geometry, and fragment shaders into our program.

Once all of that is complete it is time to render our geometry and here are some aspects to consider and pay attention to:

```
# Vertex positions
glBindBuffer(GL_ARRAY_BUFFER, self.__vbo_geometry)
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 * 4, None)
glEnableVertexAttribArray(0)
# glDrawArrays(GL_LINE_LOOP, 0, 4)
glDrawArrays(GL_POINTS, 0, len(self.__geometry))

# Vertex Colors
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 8 * 4, ctypes.c_void_p(3 * 4))
glEnableVertexAttribArray(1)

# Number of Vertex
glVertexAttribPointer(2, 1, GL_FLOAT, GL_FALSE, 8 * 4, ctypes.c_void_p(6 * 4))
glEnableVertexAttribArray(2)

# Scale of Vertex
glVertexAttribPointer(3, 1, GL_FLOAT, GL_FALSE, 8 * 4, ctypes.c_void_p(7 * 4))
glEnableVertexAttribArray(3)

# Define shaders to use
glUseProgram(self.__shader)
```

Fig. 2- Code for passing information to the vertex shader from the geometry data in Fig.1, splitting the data accordingly to be read correctly.

Each vertex information seen in Fig.1 is read and sent to the vertex shader independently, giving each piece of information an index, which is used to identify it on the OpenGL code, its length, type, size, and offset are also important for reading the data correctly.

```
#version 330
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 color;
layout (location = 2) in float  sides;
layout (location = 3) in float  scale;
out vec3 vColor;
out float vSides;
out float vScale;

void main() {
    gl_Position =  vec4(position, 1.0);// Vertex position
    vColor = vec3(color);// Vertex color
    vSides = sides;//Number of vertex
    vScale = scale;//Drawing scale
    gl_PointSize = 5;// In case the shape is a point sets point size in px
}
```

Fig. 3- Vertex shader code. Declaration of input and output variables, initialization of the geometry's position and assigning values to output variables that are then sent to the geometry shader.

Looking at Fig.3 we can see how the OpenGL code receives the different information through the different indexes (locations) mentioned in Fig.2, and how we send that data to the geometry shader with the **out** variables. Our fragment shader is very simple simply receiving and outputting the color of the fragment. At this point, without a geometry shader the only thing we would see would be dots in the screen.

Now we will introduce the geometry shader, and here is where the real magic happens, for a **single** vertex, we will create cylinders (keep in mind our vertex limitation problem mentioned before) but first we will briefly analyze its code.

```
in float vSides[]; //Number of vertex for each geometry
in vec3 vColor[];// Output from vertex shader for each vertex
out vec3 fColor;// Output to fragment shader
const float PI = 3.1415926;
in float vScale[];
float a =0;

layout(points) in;// Describes what kind of primitives our shader will process.
layout(line_strip, max_vertices = 146) out;// Determines what kind of geometry our shader will output.

void main()
{
    fColor = vColor[0];// Point has only one vertex
    for (int i = 0; i <=12; i++) {
        a+=0.01;
        for (int i = 0; i <=vSides[0]; i++) {
            // Angle between each side in radians
            float ang = PI * 2.0 / vSides[0]* i;

            // Offset from center of point (0.45 to accomodate for aspect ratio)
            //cos - horz | sin - vert
            vec4 offset = vec4((cos(ang) * 0.45)*vScale[0]+a, (-sin(ang) * 0.8)*vScale[0]+a, 1.5, 1.5);
            gl_Position = gl_in[0].gl_Position + offset;

            EmitVertex();
        }
    }

    EndPrimitive();//Generates the primitive
}
```

Fig. 4- Geometry shader code. Creating new geometries (circles) by emitting new vertices based on the original vertex received previously in the pipeline.

In Fig.4, in the variable declaration outside **main**, it's important to specify what kind of input and output types we will be using with the **layout** statement (don't need to be the same), there are several different types available. In **main**, the inner for loop creates a circle depending on the number of sides on the input and the outer **for** loop will replicate those circles with a slight offset between each other thus creating a cylinder or a pipe. For every vertex that we want to create in the geometry shader we must call the **EmitVertex** function and finalize our geometry with the **EndPrimitive** function. The result of the code in Fig.4 can be seen below on Fig.5.

With just 8 initial vertices we were able to create 8 cylinders with a total of 1168 vertices, each appearing “further away” (in fact it is just the scale that gets smaller) from the camera, as we can see it is difficult to see the difference between a faraway cylinder (bottom right) which in fact has only 6 sides and a much closer cylinder with 32 sides (top left), thus proving the usefulness of the geometry shader in LOD.

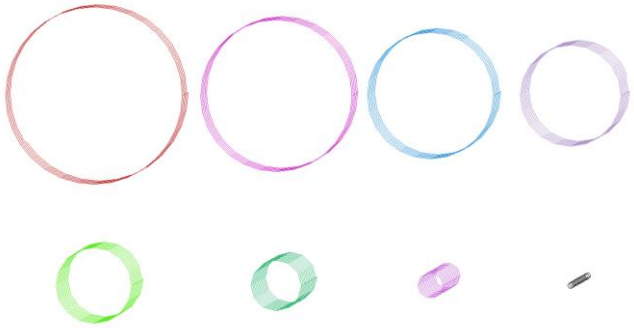

Fig. 5- Result with a geometry shader.

III. Conclusions

We were able to prove the geometry shaders utility in a LOD context by showing a simple example in a scenario where an object became more detailed or less detailed depending on the player’s distance to the object but we could also see some of the limitations when using a geometry shader (max number of vertices), despite this limitation we think that the provided images were sufficient to exemplify both its usefulness and constraints. It is still important to mention that another barrier when using the geometry shader, is its performance even though we did not cover it in an analytical way, we also did not cover other pertinent usages for the geometry shader such as fur generation or dynamic particle systems due to their intrinsic complexity.

In conclusion, we could see that geometry shaders once had a promising start, but as everything related to visual computing or IT in general, they became outdated and somewhat obsolete being surpassed by other technologies such as mesh or tessellation shaders or even simply by using instancing as a method, nevertheless they are relevant and useful in some scenarios.

References

J. Barczak, www.joshbarczak.com/blog/?p=667. March 2015.
The Khronos Group Inc, www.khronos.org/opengl/wiki/Geometry_Shader. March 2021.
J. de Vries, www.learnopengl.com/Advanced-OpenGL/Geometry-Shader. June 2020.
A. Overvoorde, www.open.gl/geometry. January 2019
Microsoft, www.docs.microsoft.com/en-us/windows/uwp/graphics-concepts/geometry-shader-stage--gs-. June 2020