GEOMETRY SHADER

If you remember from one of the very first chapters, there are two optional shader stages between the vertex shader and fragment shader, the tesselation shader and the **geometry shader**.

A geometry shader takes, as input, a set of vertices that form a single primitive (e.g. triangle, point, line, etc.) and can transform these vertices as it sees fit before sending them to the next shader stage.
- The geometry shader is able to convert the original primitive (set of vertices) to completely different primitives, possibly generating more vertices than were initially given.

Take a look at this example geometry shader for the following notes.

At the start of a geometry shader, we need to declare the type of primitive input we're receiving from the vertex shader. We do this by declaring a layout specifier in front of the `in` keyword. This input layout qualifier can take any of the following primitive values (the number in the parenthesis represents the minimum number of vertices a single primitive contains):
- **`points`**: When drawing `GL_POINTS` primitives (1).
- **`lines`**: When drawing `GL_LINES` or `GL_LINE_STRIP` (2).
- **`lines_adjacency`**: `GL_LINES_ADJACENCY` or `GL_LINE_STRIP_ADJACENCY` (4).
- **`triangles`**: `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, or `GL_TRIANGLE_FAN` (3).
- **`triangles_adjacency`**: `GL_TRIANGLES_ADJACENCY` or `GL_TRIANGLE_STRIP_ADJACENCY` (6).

This is almost all the rendering primitives we're able to give to rendering calls (like `glDrawArrays`).

We also need to specify a primitive type that the geometry shader will output. We do this via a layout specifier in front of the `out` keyword. Like the input layout qualifier, the output layout qualifier can take several primitive values:
- **`points`**
- **`line_strip`**
- **`triangle_strip`**

With just these three output specifiers, we can create almost any shape we want from the input primitives.
- Example: If we wanted to generate a single triangle, we'd specify `triangle_strip` as the output and output three vertices.

The geometry shader also expects us to set the maximum number of vertices it outputs (if you exceed this number, OpenGL won't draw the extra vertices), which we can also do within the layout qualifier of the out keyword.

To generate meaningful results, we need some way to retrieve the output from the previous shader stage. We can use the built-in GLSL variable `gl_in`, which internally (probably) looks something like this.
- Here, `gl_in` is represented as an interface block (as discussed in the previous chapter) that contains variables, such as `gl_Position` which contains the vector we set as the vertex shader's output.
- **NOTE:** `gl_in` is declared as an array since most render primitives contain more than one vertex. The geometry shader retrieves all vertices of a primitive as its input.

Using the vertex data from the vertex shader stage, we can generate new data with two geometry shader functions, `EmitVertex` and `EndPrimitive`.
- The geometry shader expects you to generate/output at least one of the primitives you specified as output.
  - *With regards to the example geometry shader provided, we need to at least generate one line strip primitive.*
- Each time we call `EmitVertex`, the vector currently set to `gl_Position` is added to the output primitive.
- Whenever `EndPrimitive` is called, all emitted vertices for this primitive are combined into the specified output render primitive.
  - By repeatedly calling `EndPrimitive` after one or more `EmitVertex` calls, multiple primitives can be generated.

With knowledge you've just gained about geometry shaders, you might have been able to identify that the example geometry shader you looked at earlier takes a point primitive as its input and creates a horizontal line primitive with the input point at its center.

Now you've seen how geometry shaders can be used to (dynamically) generate new shapes on the fly.

Using Geometry Shaders

Draw four green points on the z-plane in normalized device coordinates (NDC).
- Drawing 4 Z-Plane Points

A **pass-through** geometry shader takes a primitive and passes it to the next shader unmodified.

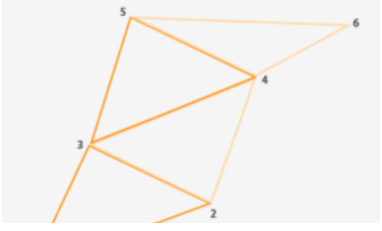
Write a pass-through geometry shader for points.
- Pass-Through Geometry Shader

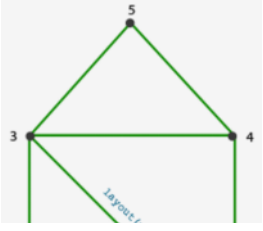
Just like the vertex and fragment shader, the geometry shader has to be compiled and linked to a program. This time, the shader is creted using `GL_GEOMETRY_SHADER` as the shader type.

Update your Shader class to have an additional constructor that takes the vertex, fragment, AND geometry shader paths and links them to a shader program. Then, create a Shader object using the previously created vertex, fragment, and geometry shaders.
- Geometry Shader Program Linking

Let's Build Houses

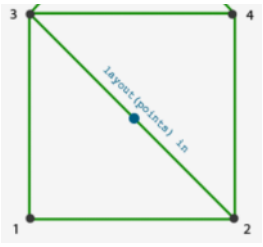
A triangle strip (`triangle_strip`) in OpenGL is a more efficient way to draw triangles with fewer vertices.
- After the first triangle is drawn, each subsequent vertex generates another triangle next to the first triangle (every three adjacent vertices will form a triangle).
  - Example: If we have a total of 6 vertices that form a triangle strip, we'd get four triangles:
    - `(1,2,3)`, `(2,3,4)`, `(3,4,5)`, and `(4,5,6)`.
- A triangle strip needs at least 3 vertices and will generate `n-2` triangles.





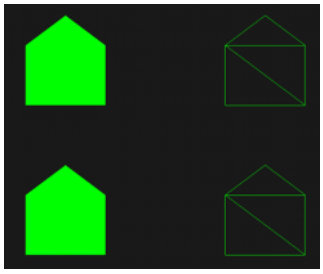
Write a geometry shader that takes a point and generates a triangle strip the form the shape of a house

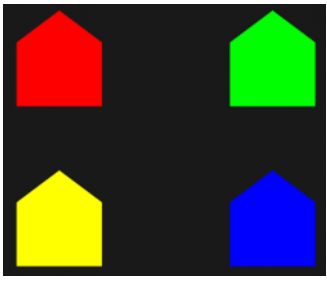
- Example: If we have a total of 6 vertices that form a triangle strip, we'd get four triangles:
  - `(1,2,3)`, `(2,3,4)`, `(3,4,5)`, and `(4,5,6)`.
- A triangle strip needs at least 3 vertices and will generate `n-2` triangles.



Write a geometry shader that takes a point and generates a triangle strip the form the shape of a house.
- Use the image to the right for reference.
- Triangle Strip House

Using this vertex data, updated with color values, recolor the houses.
- Colored Houses



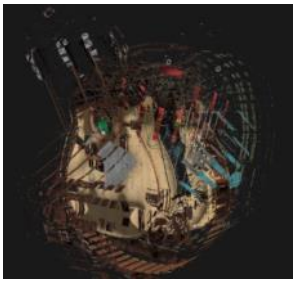


You could make the peaks of the houses a white gradient by specifying a white color for the last vertex before emitting it in the geometry shader.

Because the shapes are generated dynamically on the GPU, which is very fast, this can be more powerful than defining these shapes yourself in vertex buffers.
- Geometry shaders are a great tool for simple (often-repeating) shapes, like cubes in a voxel world, or grass leaves on a large outdoor field.

Exploding Objects

When we refer to "exploding" objects, we're talking about moving the triangles of that object out along their normal vectors over time.
- Because we're going to translate each vertex into the direction of the triangle's normal vector, we first need to calculate this normal vector.
  - We need to calculate a vector that is perpendicular to the surface of a triangle, using the 3 vertices we have access to.
    - The cross product of two vertices parallel to the triangle's surface will produce the vector we're looking for, which points perpendicular to the triangle's surface.

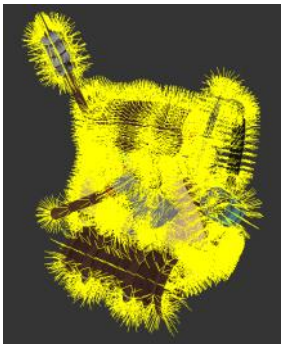


Explode the backpack object used in the previous chapters (or any 3D object).
- Exploding Objects

Visualizing Normal Vectors

When programming lighting shaders, you will eventually run into weird visual outputs of which the cause is hard to determine. A common cause of lighting errors is incorrect normal vectors, sometimes caused by incorrectly loading vertex data, improperly specifying them as vertex attributes, or by incorrectly managing them in the shaders.

We want some way to detect if the normal vectors we supplied for an object are correct, for debugging purposes. A great way to determine if the normal vectors are correct is by visualizing them, which is something very useful that we can use the geometry shader for, like so:
1. First, draw the scene as normal without a geometry shader.
2. Then, draw the scene a second time, only displaying normal vector that we generate via a geometry shader.
   - The geometry shader takes, as input, a triangle primitive and generates 3 lines from them in the directions of their normal - one normal vector for each vertex.



Visualize the normal vectors of the objects in your scene as yellow lines by following the above steps.
- Normal Vectors Visualized

Example Geometry Shader

Wednesday, May 18, 2022 8:48 PM

Geometry Shader

```
#version 330 core

layout (points) in;
layout (line_strip, max_vertices = 2) out;

void main() {
   gl_Position = gl_in[0].gl_Position + vec4(-0.1, 0.0, 0.0, 0.0);
   EmitVertex();

   gl_Position = gl_in[0].gl_Position + vec4(0.1, 0.0, 0.0, 0.0);
   EmitVertex();

   EndPrimitive();
}
```

This geometry shader takes in GL_POINTS primitives and outputs a line strip.

gl_in

Wednesday, May 18, 2022 9:22 PM

```
in gl_Vertex {
   vec4 gl_Position;
   float gl_PointSize;
   float gl_ClipDistance[];
} gl_in[];
```

Drawing 4 Z-Plane Points

Wednesday, May 18, 2022 10:10 PM

geometry.vert

```
#version 330 core

layout (location = 0) in vec2 aPos;

void main() {
   gl_Position = vec4(aPos, 0.0, 1.0);
}
```

geometry.frag

```
#version 330 core

out vec4 FragColor;

void main() {
   FragColor = vec4(0.0, 1.0, 0.0, 1.0);
}
```

main()

```
Shader sampleShader{ "Shaders/geometry.vert", "Shaders/geometry.frag" };

...

unsigned int zPointsVAO, zPointsVBO;
glGenVertexArrays(1, &zPointsVAO);
glBindVertexArray(zPointsVAO);

glGenBuffers(1, &zPointsVBO);
glBindBuffer(GL_ARRAY_BUFFER, zPointsVBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(zPoints), &zPoints, GL_STATIC_DRAW);

glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, sizeof(float) * 2, (void*)0);
glEnableVertexAttribArray(0);

glBindVertexArray(0);
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

Render Loop

```
sampleShader.use();
glBindVertexArray(zPointsVAO);
glDrawArrays(GL_POINTS, 0, 4);
```

Pass-Through Geometry Shader

Wednesday, May 18, 2022 10:16 PM

pass_through.geom

```
#version 330 core

layout (points) in;
layout (points, max_vertices = 1) out;

void main() {
   gl_Position = gl_in[0].gl_Position;
   EmitVertex();
   EndPrimitive();
}
```

Geometry Shader Program Linking

Wednesday, May 18, 2022 10:31 PM

Shader.h

```
Shader(const char* vertexShaderPath, const char* fragmentShaderPath, const char* geometryShaderPath);
```

Shader(vertexShaderPath, fragmentShaderPath, geometryShaderPath)

```
unsigned int geometry = glCreateShader(GL_GEOMETRY_SHADER);
glShaderSource(geometry, 1, &gShaderCode, NULL);
glCompileShader(geometry);
checkShaderComp(geometry, "GEOMETRY");
...
glAttachShader(id, geometry);
glLinkProgram(id);
...
glDeleteShader(geometry);
```

main()

```
Shader sampleShader{ "Shaders/geometry.vert", "Shaders/geometry.frag", "Shaders/pass_through.geom"};
```

If you compile your code now, the output should still be exactly the same as it was before, since we're using a pass-through geometry shader, which doesn't modify anything.

Triangle Strip House

Thursday, May 19, 2022 6:13 PM

house.geom

```
#version 330 core

layout (points) in;
layout (triangle_strip, max_vertices = 5) out;

void build_house(vec4 position) {
   gl_Position = position + vec4(-0.2, -0.2, 0.0, 0.0); // 1:Bottom-left
   EmitVertex();
   gl_Position = position + vec4(0.2, -0.2, 0.0, 0.0); // 2:Bottom-right
   EmitVertex();
   gl_Position = position + vec4(-0.2, 0.2, 0.0, 0.0); // 3:Top-left
   EmitVertex();
   gl_Position = position + vec4(0.2, 0.2, 0.0, 0.0); // 4:Top-right
   EmitVertex();
   gl_Position = position + vec4(0.0, 0.4, 0.0, 0.0); // 5:Peak
   EmitVertex();

   EndPrimitive();
}

void main() {
   build_house(gl_in[0].gl_Position);
}
```

main()

```
Shader sampleShader{ "Shaders/geometry.vert", "Shaders/geometry.frag", "Shaders/house.geom"};
```

You should see that each house is made up of 3 triangles.

Vertex Data with Color

Thursday, May 19, 2022 7:01 PM

```
float zPoints[] {
   // Positions   // Colors
   -0.5f,  0.5f,  1.0f, 0.0f, 0.0f, // top-left
    0.5f,  0.5f,  0.0f, 1.0f, 0.0f, // top-
   right
    0.5f, -0.5f,  0.0f, 0.0f, 1.0f, // bottom-
   right
   -0.5f, -0.5f,  1.0f, 1.0f, 0.0f  // bottom-
   left
};
```

Colored Houses

Thursday, May 19, 2022 6:37 PM

NOTE: It's not required to use an interface block to communicate with the geometry shader (we could just use out vec3 color and in vec3 color[] in the vertex and geometry shader, respectively), but, in practice, geometry shader inputs can get quite large, so grouping them in one large interface block array makes a lot more sense.

geometry.vert

```
#version 330 core

layout (location = 0) in vec2 aPos;
layout (location = 1) in vec3 aColor;

out VS_OUT {
   vec3 color;
} vs_out;

void main() {
   vs_out.color = aColor;
   gl_Position = vec4(aPos, 0.0, 1.0);
}
```

house.geom

```
...

in VS_OUT {
   vec3 color;
} gs_in[]; // Input from the vertex shader is always represented as arrays of vertex data

out vec3 fColor;

void build_house(vec4 position) {
   fColor = gs_in[0].color;
   ...
}

...
```

geometry.frag

```
#version 330 core

out vec4 FragColor;

in vec3 fColor;

void main() {
   FragColor = vec4(fColor, 1.0);
}
```

Remember to update the vertex attribute pointer with the color data in the vertex data:

main()

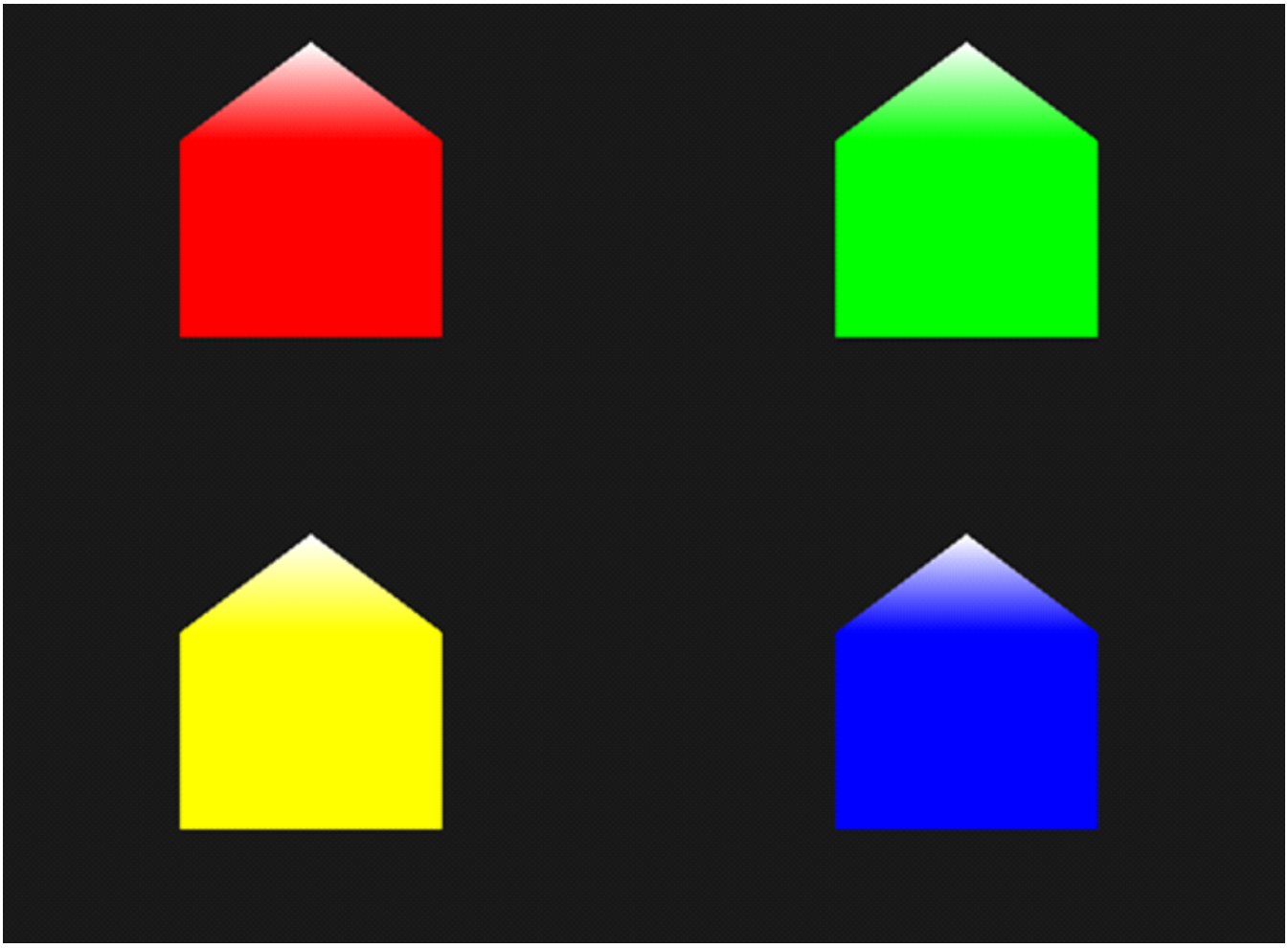
```
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, sizeof(float) * 5, (void*)0);
glEnableVertexAttribArray(0);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(float) * 5, (void*)(sizeof(float) * 2));
glEnableVertexAttribArray(1);
```

Snowy Roofs

Thursday, May 19, 2022 7:15 PM

```
void build_house(vec4 position) {
   gl_Position = position + vec4(-0.2, -0.2, 0.0, 0.0); // 1:Bottom-left
   EmitVertex();
   gl_Position = position + vec4(0.2, -0.2, 0.0, 0.0); // 2:Bottom-right
   EmitVertex();
   gl_Position = position + vec4(-0.2, 0.2, 0.0, 0.0); // 3:Top-left
   EmitVertex();
   gl_Position = position + vec4(0.2, 0.2, 0.0, 0.0); // 4:Top-right
   EmitVertex();
   gl_Position = position + vec4(0.0, 0.4, 0.0, 0.0); // 5:Peak
   fColor = vec4(1.0, 1.0, 1.0); // Makes peak white
   EmitVertex();

   EndPrimitive();
}
```



Exploding Objects

Thursday, May 19, 2022 9:04 PM

You can use the model.vert and model.frag shaders from the model loading chapter, but you will need to altar the formatting o f the texture coordinates variable, as seen in the vertex and fragment shader below:

model.vert

```
#version 330 core

layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoords;

out VS_OUT {
   vec2 texCoords;
} vs_out;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main() {
   gl_Position = projection * view * model * vec4(aPos, 1.0);
   vs_out.texCoords = aTexCoords;
}
```

explode.geom

```
#version 330 core

layout (triangles) in;
layout (triangle_strip, max_vertices = 3) out;

in VS_OUT {
   vec2 texCoords;
} gs_in[];

out vec2 fTexCoords;

uniform float time;

// Returns the normal vector of the triangle
vec3 getNormal() {
   vec3 a = vec3(gl_in[0].gl_Position) - vec3(gl_in[1].gl_Position); // First parallel vector
   vec3 b = vec3(gl_in[2].gl_Position) - vec3(gl_in[1].gl_Position); // Second parallel vector
   return normalize(cross(a, b));                                   // Normal (perpendicular) vector (order is important!)
}

// Returns the new position of the vertex after "exploding" it
vec4 explode(vec4 position, vec3 normal) {
   const float MAGNITUDE = 2.0;
   vec3 direction = normal * ((sin(time) + 1.0) / 2.0) * MAGNITUDE; // By adding 1.0 and dividing by 2.0 we can
                                                                    // prevent the object from imploding
   return (position + vec4(direction, 0.0));
}

void main() {
   vec3 normal = getNormal();

   gl_Position = explode(gl_in[0].gl_Position, normal);
   fTexCoords = gs_in[0].texCoords; // texture coordinates of the first vertex
   EmitVertex();

   gl_Position = explode(gl_in[1].gl_Position, normal);
   fTexCoords = gs_in[1].texCoords; // texture coordinates of the second vertex
   EmitVertex();

   gl_Position = explode(gl_in[2].gl_Position, normal);
   fTexCoords = gs_in[2].texCoords; // texture coordinates of the third vertex
   EmitVertex();

   EndPrimitive();
}
```

model.frag

```
#version 330 core
```

```
struct Material {
    sampler2D texture_diffuse1;
};
uniform Material material;

out vec4 FragColor;

in vec2 fTexCoords;

void main() {
    FragColor = texture(material.texture_diffuse1, fTexCoords);
}
```

main()

```
Shader explodeShader{ "Shaders/model.vert", "Shaders/model.frag", "Shaders/explode.geom" };
Model backpackModel{ "Models/backpack/backpack.obj" };
```

Render Loop

```
explodeShader.use();

glm::mat4 model{ 1.0f };
glm::mat4 view = camera.GetViewMatrix();
glm::mat4 projection = glm::perspective(camera.fov, (float)WIDTH / (float)HEIGHT, 0.1f, 100.0f);

glUniformMatrix4fv(glGetUniformLocation(explodeShader.id, "model"), 1, GL_FALSE, glm::value_ptr(model));
glUniformMatrix4fv(glGetUniformLocation(explodeShader.id, "view"), 1, GL_FALSE, glm::value_ptr(view));
glUniformMatrix4fv(glGetUniformLocation(explodeShader.id, "projection"), 1, GL_FALSE, glm::value_ptr(projection));

glUniform1f(glGetUniformLocation(explodeShader.id, "time"), glfwGetTime());

backpackModel.Draw(explodeShader);
```

Normal Vectors Visualized

Thursday, May 19, 2022 9:58 PM

Here, we pass `gl_Position` to the geometry shader as view-space coordinates. This is because our projection matrix modifies the positions of vertices based on perspective projection, which is non-linear, and we want to adjust the endpoint of the line strip representing the normal vector in a linear space. So, we must work in view-space when modifying the line vector.

- **NOTE:** You control what space the geometry shader works in.
  - Example: If you send `gl_Position` in view-space, the geometry shader will have to work with values in view-space and must convert `gl_Position` to clip-space before emitting a vertex.
  - You don't even have to set `gl_Position` in the vertex shader as long as you set it in the geometry shader.
- You may be thinking "Why haven't we been doing this in the previous parts of this chapter?". Well, I don't know why Joey hasn't fixed this, but for the sake of laziness, the explosion calculations work fine as is. If you want to, go back and make sure the calculations happen in view-space, then convert to clip-space. You will probably notice a difference.

modelVisNorm.vert

```
#version 330 core

layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
// We're not using the texture coordinates, so we can just omit that layout, even though it's being set with the vertex attribute pointer

out VS_OUT {
   vec3 normal;
} vs_out;

uniform mat4 model;
uniform mat4 view;

void main() {
   gl_Position = view * model * vec4(aPos, 1.0);
   mat3 normalMatrix = mat3(transpose(inverse(view * model)));
   vs_out.normal = normalize(normalMatrix * aNormal);
}
```

visualNormals.geom

```
#version 330 core

layout (triangles) in;
layout (line_strip, max_vertices = 6) out;

in VS_OUT {
   vec3 normal;
} gs_in[];

uniform mat4 projection;

const float MAGNITUDE = 0.3; // Controls the length of the line

void generateLine(int index) {
   gl_Position = projection * gl_in[index].gl_Position; // Don't forget to multiply by the projection matrix
   EmitVertex();
   gl_Position = projection * (gl_in[index].gl_Position + vec4(gs_in[index].normal, 0.0) * MAGNITUDE);
   EmitVertex();

   EndPrimitive();
}

void main() {
   generateLine(0);
   generateLine(1);
   generateLine(2);
}
```

modelVisNorm.frag

```
#version 330 core

out vec4 FragColor;

void main() {
    FragColor = vec4(1.0, 1.0, 0.0, 1.0);
}
```

main()

```
Shader modelShader{ "Shaders/model.vert", "Shaders/model.frag" };
Shader normalDisplayShader{ "Shaders/modelVisNorm.vert", "Shaders/modelVisNorm.frag", "Shaders/visualNormals.geom" };

Model backpackModel{ "Models/backpack/backpack.obj" };
```

Render Loop

```
modelShader.use();
```

```
drawScene();
normalDisplayShader.use();
drawScene();
```

There's likely a more efficient way of accomplishing this besides drawing the scene twice.