ADVANCED GLSL

In this chapter, we'll discuss some interesting built-in variables, new ways to organize shader input and output, and uniform buffer objects.

GLSL's Built-in Variables

There are a few variables defined by GLSL prefixed with `gl_` that we have not used yet that give us an extra means to gather and/or write data.

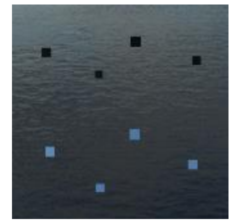
- The two we've seen so far are `gl_Position`, the clip-space output position vector of the vertex shader, and `gl_FragCoord`, the window-relative input coordinates of the fragment (in the fragment shader).

We won't discuss ALL of GLSL's built-in variables, but you can check out the [wiki]() if you'd like to see all of them.

Vertex Shader Variables

Setting `gl_Position` in the vertex shader is required if you're trying to render anything. We already know this.

`gl_PointSize`



One of the render primitives we're able to choose from is `GL_POINTS`, in which case each single vertex is a primitive and rendered as a point. You can set the size of the points by using the `glPointSize` function, or by configuring the `gl_PointSize` output variable in the vertex shader.

- `gl_PointSize` is a float variable where you can set the point's width and height in pixels.
    - By setting the point's size in the vertex shader, we get per-vertex control over this point's dimensions.
- Influencing the point sizes in the vertex shader is disabled by default, but you can enable it with `glEnable(GL_PROGRAM_POINT_SIZE)`.

Enable point size modification and modify the vertex shader of the cube to adjust the point size based on the distance from the viewer.

- [gl_PointSize]()

Varying the point size per vertex is interesting for techniques like particle generation.

`gl_VertexID`

`gl_VertexID` is an input variable in the vertex shader that holds the current ID of the vertex we're drawing.

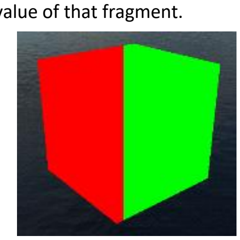
- When doing indexed rendering (via `glDrawElements`), this variable holds the current index of the vertex we're drawing.
- When drawing without indices (via `glDrawArrays`), this variable holds the number of the currently processed vertex since the start of the render call.

Fragment Shader Variables

GLSL gives us two interesting input variables called `gl_FragCoord` and `gl_FrontFacing`.

`gl_FragCoord`

We've used `gl_FragCoord` before during the discussion of depth testing because the z-component of the `gl_FragCoord` vector is equal to the depth value of that fragment. However, we can also use the x- and y-component of that vector for some interesting effects.



- `gl_FragCoord`'s x- and y-component are the window- or screen-space coordinates of the fragment, originating from the bottom-left of the window.
    - Example: If you specified a render window of `800x600` with `glViewport`, then the screen-space coordinates of the fragment will have x-values ranging from 0 to 800 and y-values ranging from 0 to 600.

A common usage for the gl_FragCoord variable is for comparing visual output of different fragment calculations, as usually seen in tech demos.

- Example: Splitting the screen in two by rendering one output to the left side of the window and another output to the right side of the window.

Write a fragment shader that outputs red for fragments of the cube in the left-half of the screen and green for fragments in the right-half.

- [gl_FragCoord]()

This is great for testing out, for example, different lighting techniques.

gl_FrontFacing

In the face culling chapter, we mentioned that OpenGL is able to figure out if a face is a front or back face due to the wind ing order of the vertices. The `gl_FrontFacing` variable is a `bool` that tells us if the current fragment is part of a front-facing or a back-facing face.

- We could use this information to output different colors for all back faces.

Using two textures, draw a cube with one texture for all the front-facing faces and the other texture for all the back-facing faces.

- **NOTE:** If you've enabled the default face culling, you won't be able to see any faces inside the container.
- [gl_FrontFacing]()

gl_FragDepth

`gl_FragCoord` is an input variable that allows us to read screen-space coordinates and get the depth value of the current fragment, but it is read-only.

We can't influence the sceen-space coordinates of the fragment, but it is possible to set the depth value of the fragment, using `gl_FragDepth`.

- To set the depth value in the shader, we write any value between `0.0` and `1.0` to the output variable.
- If the shader does not write anything to `gl_FragDepth`, the variable will automatically take its value from `gl_FragCoord.z`.

The major disadvantage to using gl_FragDepth is that, as soon as it is included in the fragment shader, early depth testing is disabled. This is because OpenGL cannot know what depth value the fragment will have <u>before</u> we run the fragment shader, since it's being updated <u>in</u> the fragment shader.

- However, from OpenGL 4.2, we can lessen this performance penalty by declaring the `gl_FragDepth` variable at the top of the fragment shader with a **depth condition**, like so:
    - `layout (depth_<condition>) out float gl_FragDepth;`
    - This `condition` can take any of the values shown in the image to the right.

Condition	Description
`any`	The default value. Early depth testing is disabled.
`greater`	You can only make the depth value larger compared to `gl_FragCoord.z`.
`less`	You can only make the depth value smaller compared to `gl_FragCoord.z`.
`unchanged`	If you write to `gl_FragDepth`, you will write exactly `gl_FragCoord.z`.

By specifying `greater` of `less` as the depth condition, OpenGL can make the assumption that you'll only write depth values larger or smaller than the fragment's depth value, respectively. This way, OpenGL is still able to do early depth testing when the depth buffer value is part of the other direc tion of `gl_FragCoord.z`.

- Example: If using the `greater` condition, early depth testing can only occur on fragments with depth values less than the current fragment's `gl_FragCoord.z` value.

Write a fragment shader that specifies a depth condition that allows you to make the fragments' depth values larger than `gl_FragCoord.z`. Then, increase the depth value of all fragments by `0.04`.

- You made find it hard (or impossible) to see your cube, depending on where your camera is positioned relative to it. Try increasing the depth value by less until you can just see the cube as you approach it.
- gl_FragDepth

Interface Blocks

So far, we've only sent data from the vertex shader to the fragment shader by declaring matching input/output variables. This is easy for smaller applications, but can become cumbersome for larger ones the require sending more than a few variables over.

To help us organize these variables, GLSL offers us **interface blocks** that allow us to group variables together.

- The declaration of an interface block looks like a struct declaration, except that it is declared using an in or out keyword, based on the block being an input or output block.

Specify an output interface block VS_OUT to send texture coordinates from the vertex shader to the fragment shader.

- Output Interface Block

We also need to delcare an input interface block in the fragment shader. The block name (VS_OUT) should be the same in the fragment shader, but the instance name (`vs_out`) can be anything you want, though you should avoid confusing names like vs_out for a fragment struct containing input values.

Specify a corresponding input interface block in the fragment shader.

- Input Interface Block

Uniform Buffer Objects

As of right now, when we use more than one shader, we continuously have to set uniform variables where most of them are exact ly the same for each shader.

Uniform buffer objects allow us to declare a set of global uniform variables that remain the same over any number of shader programs. This way, we only have to set frequently used uniforms once in fixed GPU memory.

- **NOTE:** You'll still need to manually set the uniforms that are unique per shader.

Uniform Block Layout

The content of a uniform block is stored in a buffer object, which is basically just a reserved piece of global GPU memory. Because this memory holds no inf ormation about what kind of data it holds, we need to tell OpenGL what parts of the memory correspond to which uniform variables in the shader.

Take a look at the example uniform block to the right.

- The block is named ExampleBlock.
- `layout (std140)` specifies that this uniform block uses a specific memory layout for its content.
  - This statement sets the uniform block layout.
  - This is discussed more later on.
- OpenGL does not clearly state the spacing between variables, so, depending on your hardware, the `vec3 vector` may be padded to an array of 4 floats (instead of 3) before appending the next data.

```
layout (std140) uniform ExampleBlock
{
    float value;
    vec3  vector;
    mat4  matrix;
    float values[3];
    bool  boolean;
    int   integer;
};
```

NOTE: Variables in a uniform block can be directly accessed without the block name as a prefix, unlike interface blocks.

By default, GLSL uses a uniform memory layout called a **shared** layout.

- It's known as a shared layout because once the offsets are defined by the hardware, they are consistently shared between multiple programs; with a shared layout, GLSL is allowed to reposition the uniform variables for optimization as long as the variables' order remains intact.
- Because we don't know at what offset each uniform variable will be, we don't know how to precisely fill our uniform buffer.
  - You can query this information with functions like glGetUniformIndices, but that's a different approach that we're not going to use.
- While a shared layout gives us some space-saving optimizations, we'd need to query the offset for each uniform variable, which translates to a lot of work.

The general practice is to not use the shared layout, but to use the **`std140`** layout.

- The `std140` layout explicitly states the memory layout for each variable type by standardizing their respective offsets governed by a set of rules.
  - Since this is standardized, we can manually figure out the offsets for each variable.
- Each variable has a **base alignment** equal to the space a variable takes (including padding) within a uniform block using the `std140` layout rules.
  - For each variable, we calculate its **aligned offset** - the byte offset of a variable from the start of the block.
  - The aligned byte offset of a variable must be equal to a multiple of its base alignment.

There are two other layouts to choose from that require us to query each offset before filling the buffers.

- We've already seen the shared layout.
- The other layout is the **packed** layout.
  - When using the packed layout, there is no guarantee that the layout remains the same between programs (not shared) because it allows the compiler to optimize uniform variables away from the uniform block, which may differ per shader.

To the right is the list of the most common rules for the `std140` layout.

- Each variable type in GLSL, such as `int`, `float`, and `bool`, are defined to be four-byte quantities, with each entity of four bytes represented as N.

Type	Layout rule
Scalar e.g. `int` or `bool`	Each scalar has a base alignment of N.
Vector	Either 2N or 4N. This means that a `vec3` has a base alignment of 4N.
Array of scalars or vectors	Each element has a base alignment equal to that of a `vec4`.
Matrices	Stored as a large array of column vectors, where each of those vectors has a base alignment of `vec4`.
Struct	Equal to the computed size of its elements according to the previous rules, but padded to a multiple of the size of a `vec4`.

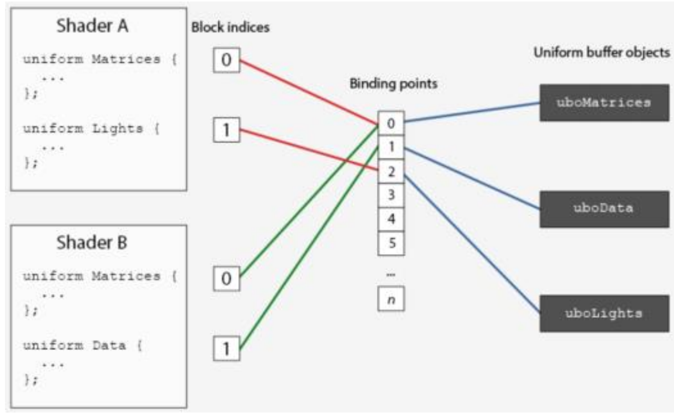
Calculate the base alignments and aligned offsets of each of the variables in the ExampleBlock image above using the layout rules provided (also above).

- ExampleBlock Aligned Offsets

With these calculated offset values, based on the rules of the `std140` layout, we can fill the buffer with data at the appropriate offsets using functions like `glBufferSubData`. While not the most efficient, the `std140` layout does guarantee us that the memory layout remains the same over each program that declared this uniform block.

Using Uniform Buffers

After we've created a uniform block, we need to store the necessary values in a uniform buffer object so that each shader that declares that uniform block has access to the data.



Create a uniform buffer object for the ExampleBlock uniform block.

- [Uniform Buffer Object]

Whenever we want to update or insert data into the UBO, we bind to the UBO and use `glBufferSubData` to update its memory.

- We only have to update this uniform buffer once. After that, all shaders that use this buffer will use its updated data.

In OpenGL, there is a number of **binding points** defined where we can link a uniform buffer to.

- Once we've created a uniform buffer, we link it to one of those binding points and we also link the uniform block in the shader to the same binding point, effectively linking them together.
- The image to the right visually represents the concept of binding points.
  - Because Shader A and Shader B both have a uniform block linked to the same binding point (`0`), their uniform blocks share the same uniform data found in the `uboMatrices` UBO.

To set a shader uniform block to a specific binding point, we call `glUniformBlockBinding`, which takes a program object, a uniform block index, and the binding point to link to.

- The **uniform block index** is a location index of the defined uniform block in the shader. This can be retrieved by calling `glGetUniformBlockIndex`, which accepts a program object and the name of the uniform block.

Write the code necessary to set the `Lights` uniform block (from the diagram) to binding point 2.

- [Setting Binding Point]

NOTE: We have to repeat this process for each shader.

NOTE: From OpenGL 4.2 and onwards, it is also possible to store the binding point of a uniform block explicitly in the shader by adding another layout specifier, saving us the calls to glGetUniformBlockIndex and `glUniformBlockBinding`. [This] code sets the binding point of the `Lights` uniform block explicitly.

Next, we need to bind the UBO to the same binding point as the relevant uniform block, which can be accomplished with either `glBindBufferBase` or `glBindBufferRange`.

- `glBindBufferBase` expects a target, binding point index, and a UBO, and is used to link all the data of a UBO to a specific binding point.
- `glBindBufferRange` expects an extra offset and size parameter so that you can bind only a specific range of the UBO to a binding point.
  - Using this, you could have multiple different uniform blocks linked to a single UBO.

Bind the uboExampleBlock UBO (from the diagram) to binding point 2.

- [Binding UBO to Binding Point]

Finally, we need to add data to the UBO. We could add all the data as a single array, or update parts of the buffer when we need to by using `glBufferSubData`.

Update the uniform variable `boolean` (from the diagram) in the UBO to `true`.

- [Updating a UBO]

A Simple Example

In all of the previous code samples, we've been using the projection, view, and model matrices. Only the model matrix changes frequently, so if we have multiple shaders that use this same set of matrices, we would be better off using UBOs.

Display 4 differently colored cubes that each use a different shader program with unique fragment shaders, but use the same vertex shader. The view and projection matrices of the vertex shader should utilize a uniform block, and should thus be populated by a UBO with the view and projection matrices' data.

- [Four Cubes UBO]

Uniform buffer objects have several advantages over single uniforms:

1. Setting a lot of uniforms at once is faster than setting multiple uniforms one at a time.
2. If you want to change the same uniform over several shaders, it is much easier to change a uniform once in a uniform buffer.
3. You can use a lot more uniform in shaders using UBOs because you can bypass the uniform data limit of OpenGL.
   - You can query the uniform data limit with `GL_MAX_VERTEX_UNIFORM_COMPONENTS`.
   - When using UBOs, the limit is much higher.

gl_PointSize

Friday, May 13, 2022 2:38 PM

Vertex Shader

```
void main() {
   ...
   gl_Position = projection * view * model * vec4(aPos, 1.0);
   gl_PointSize = 2 * gl_Position.z;
}
```

main()

```
glEnable(GL_PROGRAM_POINT_SIZE);
```

Render Loop

```
glBindVertexArray(reflectCubeVAO);
glDrawArrays(GL_POINTS, 0, 36);
```

gl_FragCoord

Friday, May 13, 2022 3:00 PM

Fragment Shader

```
void main() {
   // The fragments in the left-half of the screen will be red and the fragments in the right-half will be green
   if (gl_FragCoord < 400) {
      FragColor = vec4(1.0, 0.0, 0.0, 1.0);
   }
   else {
      FragColor = vec4(0.0, 1.0, 0.0, 1.0);
   }
}
```

gl_FrontFacing

Friday, May 13, 2022 6:40 PM

Fragment Shader

```
void main() {
   if (gl_FrontFacing) {
      FragColor = texture(texFront, texCoords);
   }
   else {
      FragColor = texture(texBack, texCoords);
   }
}
```

main.cpp

```
Texture2D containerTexture{ "Textures/container.jpg", 0, GL_CLAMP_TO_EDGE, GL_CLAMP_TO_EDGE };
Texture2D containerTexture2{ "Textures/container2.png", 1, GL_CLAMP_TO_EDGE, GL_CLAMP_TO_EDGE };
```

Render Loop

```
advancedShader.use();
containerTexture.bind();
containerTexture2.bind();

glUniform1i(glGetUniformLocation(advancedShader.id, "texFront"), 0);
glUniform1i(glGetUniformLocation(advancedShader.id, "texBack"), 1);

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

glBindVertexArray(objVAO);
glDrawArrays(GL_TRIANGLES, 0, 36);

glActiveTexture(GL_TEXTURE0);
```

gl_FragDepth

Friday, May 13, 2022 8:23 PM

Fragment Shader

```
#version 420 core // Note the GLSL version!

layout (depth_greater) out float gl_FragDepth;

out vec4 FragColor;

void main() {
   FragColor = vec4(1.0);
   gl_FragDepth = gl_FragCoord.z + 0.04;
}
```

Output Interface Block

Friday, May 13, 2022 8:35 PM

Vertex Shader

```
#version 330 core

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

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

out VS_OUT {
   vec2 texCoords;
} vs_out;

void main() {
   vs_out.texCoords = aTexCoords;

   gl_Position = projection * view * model * vec4(aPos, 1.0);
}
```

NOTE: This is somewhat of a trivial example, but you can imagine that this helps organize your shaders' inputs/outputs. It is also useful when you want to group shader input/output arrays as we'll see in the next chapter about geometry shaders.

Input Interface Block

Friday, May 13, 2022 8:35 PM

Vertex Shader

```
#version 330 core

out vec4 FragColor;

uniform sampler2D texFront;
uniform sampler2D texBack;

in VS_OUT {
    vec2 texCoords;
} fs_in;

void main() {
    if (gl_FrontFacing) {
        FragColor = texture(texFront, fs_in.texCoords);
    }
    else {
        FragColor = texture(texBack, fs_in.texCoords);
    }
}
```

ExampleBlock Aligned Offsets

Monday, May 16, 2022 6:46 PM

```
layout (std140) uniform ExampleBlock {
                        // Base Alignment                  // Aligned Offset
    float value;        // 4                               0
    vec3 vector;        // 16                              16 (offset must be a multiple of 16, so 4 -> 16)
    mat4 view;          // column 0 = 16                   32
                        // column 1 = 16                   48
                        // column 2 = 16                   64
                        // column 3 = 16                   80
    float values[3];    // values[0] = 16                  96
                        // values[1] = 16                  112
                        // values[2] = 16                  128
    bool boolean;       // 4                               144
    int integer;        // 4                               148
};
```

Uniform Buffer Object

Monday, May 16, 2022 7:35 PM

```
unsigned int uboExampleBlock;
glGenBuffers(1, &uboExampleBlock);
glBindBuffer(GL_UNIFORM_BUFFER, uboExampleBlock);
glBufferData(GL_UNIFORM_BUFFER, 152, NULL, GL_STATIC_DRAW); // allocate 152 bytes of memory
glBindBuffer(GL_UNIFORM_BUFFER, 0);
```

Setting Binding Point

Monday, May 16, 2022 8:54 PM

```
unsigned int lights_index = glGetUniformBlockIndex(shaderA.ID, "Lights");
glUniformBlockBinding(shaderA.ID, lights_index, 2);
```

OpenGL 4.2 Binding Layout Specifier

Monday, May 16, 2022 9:02 PM

```
layout(std140, binding = 2) uniform Lights {
   ...
};
```

Binding UBO to Binding Point

Monday, May 16, 2022 9:09 PM

```
glBindBufferBase(GL_UNIFORM_BUFFER, 2, uboExampleBlock);
// or
glBindBufferRange(GL_UNIFORM_BUFFER, 2, uboExampleBuffer, 0, 152);
```

Updating a UBO

Monday, May 16, 2022 9:21 PM

```
glBindBuffer(GL_UNIFORM_BUFFER, uboExampleBlock);
int b = true; // bools in GLSL are represented by 4 bytes, so we store it as an integer (true = 1)
glBufferSubData(GL_UNIFORM_BUFFER, 144, 4, &b);
glBindBuffer(GL_UNIFORM_BUFFER, 0);
```

This applies to updating all the other uniform variables inside the uniform block, but with different range arguments.

Four Cubes UBO

Monday, May 16, 2022 9:31 PM

ubo.vert

```
#version 330 core

layout (location = 0) in vec3 aPos;

// Matrices uniform block
layout (std140) uniform Matrices {
   mat4 projection;
   mat4 view;
};

uniform mat4 model;

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

uboRed.frag

```
#version 330 core

out vec4 FragColor;

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

The green, blue, and white fragment shaders are the exact same as the above, except with their respective output color.

main.cpp

```
// Bind the Matrices uniform block for each shader program to binding point 0
glUniformBlockBinding(uboRedShader.id, glGetUniformBlockIndex(uboRedShader.id, "Matrices"), 0);
glUniformBlockBinding(uboGreenShader.id, glGetUniformBlockIndex(uboGreenShader.id, "Matrices"), 0);
glUniformBlockBinding(uboBlueShader.id, glGetUniformBlockIndex(uboBlueShader.id, "Matrices"), 0);
glUniformBlockBinding(uboWhiteShader.id, glGetUniformBlockIndex(uboWhiteShader.id, "Matrices"), 0);

// Create the UBO
unsigned int ubo;
glGenBuffers(1, &ubo);
glBindBuffer(GL_UNIFORM_BUFFER, ubo);
glBufferData(GL_UNIFORM_BUFFER, 2 * sizeof(glm::mat4), NULL, GL_STATIC_DRAW); // allocates 128 bytes of memory
glBindBuffer(GL_UNIFORM_BUFFER, 0);

glBindBufferBase(GL_UNIFORM_BUFFER, 0, ubo);
// or
// glBindBufferRange(GL_UNIFORM_BUFFER, 0, ubo, 0, 2 * sizeof(glm::mat4));

// Fill the buffer
glBindBuffer(GL_UNIFORM_BUFFER, ubo);
glm::mat4 projection = glm::perspective(camera.fov, (float)WIDTH / (float)HEIGHT, 0.1f, 100.0f);
glBufferSubData(GL_UNIFORM_BUFFER, 0, sizeof(glm::mat4), glm::value_ptr(projection));
// NOTE: This will disable zooming and translating/rotating the camera
glm::mat4 view = camera.GetViewMatrix();
glBufferSubData(GL_UNIFORM_BUFFER, sizeof(glm::mat4), sizeof(glm::mat4), glm::value_ptr(view));
glBindBuffer(GL_UNIFORM_BUFFER, 0);
```

Render Loop

```
glBindVertexArray(cubeVAO);

uboRedShader.use();
glm::mat4 model{ 1.0f };
model = glm::translate(model, glm::vec3(-0.75, 0.75, 0.0f)); // move top-left
glUniformMatrix4fv(glGetUniformLocation(uboRedShader.id, "model"), 1, GL_FALSE, glm::value_ptr(model));
```

```
glDrawArrays(GL_TRIANGLES, 0, 36);

uboGreenShader.use();
model = glm::mat4{ 1.0f };
model = glm::translate(model, glm::vec3(0.75, 0.75, 0.0f)); // move top-right
glUniformMatrix4fv(glGetUniformLocation(uboGreenShader.id, "model"), 1, GL_FALSE, glm::value_ptr(model));
glDrawArrays(GL_TRIANGLES, 0, 36);

uboBlueShader.use();
model = glm::mat4{ 1.0f };
model = glm::translate(model, glm::vec3(-0.75, -0.75, 0.0f)); // move bottom-left
glUniformMatrix4fv(glGetUniformLocation(uboBlueShader.id, "model"), 1, GL_FALSE, glm::value_ptr(model));
glDrawArrays(GL_TRIANGLES, 0, 36);

uboWhiteShader.use();
model = glm::mat4{ 1.0f };
model = glm::translate(model, glm::vec3(0.75, -0.75, 0.0f)); // move bottom-right
glUniformMatrix4fv(glGetUniformLocation(uboWhiteShader.id, "model"), 1, GL_FALSE, glm::value_ptr(model));
glDrawArrays(GL_TRIANGLES, 0, 36);
```