Uniform buffer objects

We've been using OpenGL for quite a while now and learned some pretty cool tricks, but also a few annoyances. For example, when using more than one shader we continuously have to set uniform variables where most of them are exactly the same for each shader.

OpenGL gives us a tool called uniform buffer objects that allow us to declare a set of *global* uniform variables that remain the same over any number of shader programs. When using uniform buffer objects we set the relevant uniforms only **once** in fixed GPU memory. We do still have to manually set the uniforms that are unique per shader. Creating and configuring a uniform buffer object requires a bit of work though.

Because a uniform buffer object is a buffer like any other buffer we can create one via `glGenBuffers`, bind it to the `GL_UNIFORM_BUFFER` buffer target and store all the relevant uniform data into the buffer. There are certain rules as to how the data for uniform buffer objects should be stored and we'll get to that later. First, we'll take a simple vertex shader and store our `projection` and `view` matrix in a so called uniform block:

```
#version 330 core
layout (location = 0) in vec3 aPos;

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

uniform mat4 model;

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

In most of our samples we set a projection and view uniform matrix every frame for each shader we're using. This is a perfect example of where uniform buffer objects become useful since now we only have to store these matrices once.

Here we declared a uniform block called `Matrices` that stores two 4x4 matrices. Variables in a uniform block can be directly accessed without the block name as a prefix. Then we store these matrix values in a buffer somewhere in the OpenGL code and each shader that declares this uniform block has access to the matrices.

You're probably wondering right now what the `layout (std140)` statement means. What this says is that the currently defined uniform block uses a specific memory layout for its content; this statement sets the uniform block layout.

Uniform block layout

The content of a uniform block is stored in a buffer object, which is effectively nothing more than a reserved piece of global GPU memory. Because this piece of memory holds no information on what kind of data it holds, we need to tell OpenGL what parts of the memory correspond to which uniform variables in the shader.

Imagine the following uniform block in a shader:

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

What we want to know is the size (in bytes) and the offset (from the start of the block) of each of these variables so we can place them in the buffer in their respective order. The size of each of the elements is clearly stated in OpenGL and directly corresponds to C++ data types; vectors and matrices being (large) arrays of floats. What OpenGL doesn't clearly state is the spacing between the variables. This allows the hardware to position or pad variables as it sees fit. The hardware is able to place a `vec3` adjacent to

a `float` for example. Not all hardware can handle this and pads the `vec3` to an array of 4 floats before appending the `float`. A great feature, but inconvenient for us.

By default, GLSL uses a uniform memory layout called a shared layout - shared 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. We can query this information with functions like `glGetUniformIndices`, but that's not the approach we're going to take in this chapter.

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 however 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. This is a bit of a mouthful, but we'll get to see some examples soon enough to clear things up.

The exact layout rules can be found at OpenGL's uniform buffer specification here, but we'll list the most common rules below. Each variable type in GLSL such as `int`, `float` and `bool` are defined to be four-byte quantities with each entity of 4 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`.

Like most of OpenGL's specifications it's easier to understand with an example. We're taking the uniform block called `ExampleBlock` we introduced earlier and calculate the aligned offset for each of its members using the std140 layout:

```
layout (std140) uniform ExampleBlock
{
                     // base alignment  // aligned offset
    float value;     // 4               // 0
    vec3 vector;     // 16              // 16  (offset must be multiple of 16 so 4->16)
    mat4 matrix;     // 16              // 32  (column 0)
                     // 16              // 48  (column 1)
                     // 16              // 64  (column 2)
                     // 16              // 80  (column 3)
    float values[3]; // 16              // 96  (values[0])
                     // 16              // 112 (values[1])
                     // 16              // 128 (values[2])
    bool boolean;    // 4               // 144
    int integer;     // 4               // 148
};
```

As an exercise, try to calculate the offset values yourself and compare them to this table. 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.

By adding the statement `layout (std140)` in the definition of the uniform block we tell OpenGL that this uniform block uses the std140 layout. 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, with the other remaining layout being `packed`. 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.

Using uniform buffers

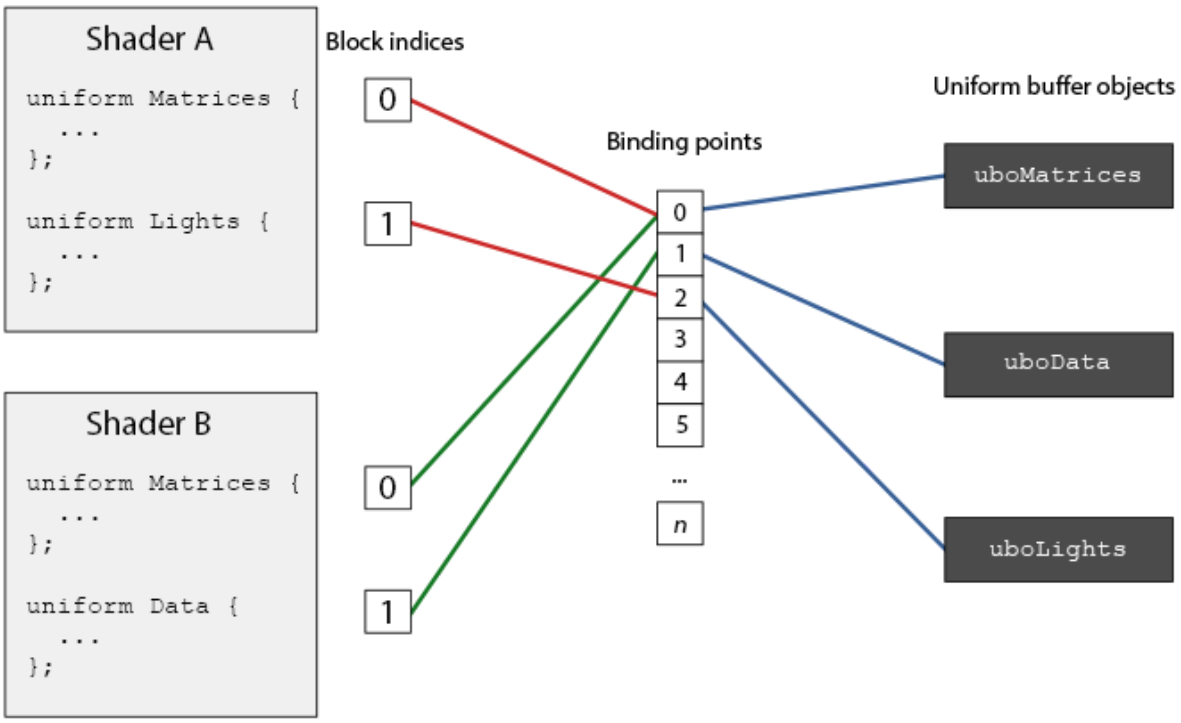
We've defined uniform blocks and specified their memory layout, but we haven't discussed how to actually use them yet.

First, we need to create a uniform buffer object which is done via the familiar `glGenBuffers`. Once we have a buffer object we bind it to the `GL_UNIFORM_BUFFER` target and allocate enough memory by calling `glBufferData`.

```
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);
```

Now whenever we want to update or insert data into the buffer, we bind to `uboExampleBlock` and use `glBufferSubData` to update its memory. We only have to update this uniform buffer once, and all shaders that use this buffer now use its updated data. But, how does OpenGL know what uniform buffers correspond to which uniform blocks?

In the OpenGL context there is a number of binding points defined where we can link a uniform buffer to. Once we 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 following diagram illustrates this:



As you can see we can bind multiple uniform buffers to different 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 `uboMatrices`; a requirement being that both shaders defined the same `Matrices` uniform block.

To set a shader uniform block to a specific binding point we call `glUniformBlockBinding` that 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 via a call to `glGetUniformBlockIndex` that accepts a program object and the name of the uniform block. We can set the `Lights` uniform block from the diagram to binding point 2 as follows:

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

Note that we have to repeat this process for **each** shader.

From OpenGL version 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`. The following code sets the binding point of the `Lights` uniform block explicitly:

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

Then we also need to bind the uniform buffer object to the same binding point and this can be accomplished with either `glBindBufferBase` or `glBindBufferRange`.

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

The function `glBindbufferBase` expects a target, a binding point index and a uniform buffer object. This function links `uboExampleBlock` to binding point 2; from this point on, both sides of the binding point are linked. You can also use `glBindBufferRange` that expects an extra offset and size parameter - this way you can bind only a specific range of the uniform buffer to a binding point. Using `glBindBufferRange` you could have multiple different uniform blocks linked to a single uniform buffer object.

Now that everything is set up, we can start adding data to the uniform buffer. We could add all the data as a single byte array, or update parts of the buffer whenever we feel like it using `glBufferSubData`. To update the uniform variable `boolean` we could update the uniform buffer object as follows:

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

And the same procedure applies for all the other uniform variables inside the uniform block, but with different range arguments.

A simple example

So let's demonstrate a real example of uniform buffer objects. If we look back at all the previous code samples we've continually been using 3 matrices: the projection, view and model matrix. Of all those matrices, only the model matrix changes frequently. If we have multiple shaders that use this same set of matrices, we'd probably be better off using uniform buffer objects.

We're going to store the projection and view matrix in a uniform block called `Matrices`. We're not going to store the model matrix in there since the model matrix tends to change frequently between shaders, so we wouldn't really benefit from uniform buffer objects.

```
#version 330 core
layout (location = 0) in vec3 aPos;

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

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

Not much going on here, except that we now use a uniform block with a std140 layout. What we're going to do in our sample application is display 4 cubes where each cube is displayed with a different shader program. Each of the 4 shader programs uses the same vertex shader, but has a unique fragment shader that only outputs a single color that differs per shader.

First, we set the uniform block of the vertex shaders equal to binding point `0`. Note that we have to do this for each shader:

```
unsigned int uniformBlockIndexRed    = glGetUniformBlockIndex(shaderRed.ID,
"Matrices");
```

```
unsigned int uniformBlockIndexGreen  = glGetUniformBlockIndex(shaderGreen.ID,
"Matrices");
unsigned int uniformBlockIndexBlue   = glGetUniformBlockIndex(shaderBlue.ID,
"Matrices");
unsigned int uniformBlockIndexYellow = glGetUniformBlockIndex(shaderYellow.ID,
"Matrices");

glUniformBlockBinding(shaderRed.ID,    uniformBlockIndexRed, 0);
glUniformBlockBinding(shaderGreen.ID,  uniformBlockIndexGreen, 0);
glUniformBlockBinding(shaderBlue.ID,   uniformBlockIndexBlue, 0);
glUniformBlockBinding(shaderYellow.ID, uniformBlockIndexYellow, 0);
```

Next we create the actual uniform buffer object and bind that buffer to binding point `0`:

```
unsigned int uboMatrices
glGenBuffers(1, &uboMatrices);

glBindBuffer(GL_UNIFORM_BUFFER, uboMatrices);
glBufferData(GL_UNIFORM_BUFFER, 2 * sizeof(glm::mat4), NULL, GL_STATIC_DRAW);
glBindBuffer(GL_UNIFORM_BUFFER, 0);

glBindBufferRange(GL_UNIFORM_BUFFER, 0, uboMatrices, 0, 2 * sizeof(glm::mat4));
```

First we allocate enough memory for our buffer which is equal to 2 times the size of `glm::mat4`. The size of GLM's matrix types correspond directly to `mat4` in GLSL. Then we link a specific range of the buffer, in this case the entire buffer, to binding point `0`.

Now all that's left to do is fill the buffer. If we keep the *field of view* value constant of the projection matrix (so no more camera zoom) we only have to update it once in our application - this means we only have to insert this into the buffer only once as well. Because we already allocated enough memory in the buffer object we can use `glBufferSubData` to store the projection matrix before we enter the render loop:

```
glm::mat4 projection = glm::perspective(glm::radians(45.0f), (float)width/(float)height, 0.1f,
100.0f);
glBindBuffer(GL_UNIFORM_BUFFER, uboMatrices);
glBufferSubData(GL_UNIFORM_BUFFER, 0, sizeof(glm::mat4),
glm::value_ptr(projection));
```

```
glBindBuffer(GL_UNIFORM_BUFFER, 0);
```

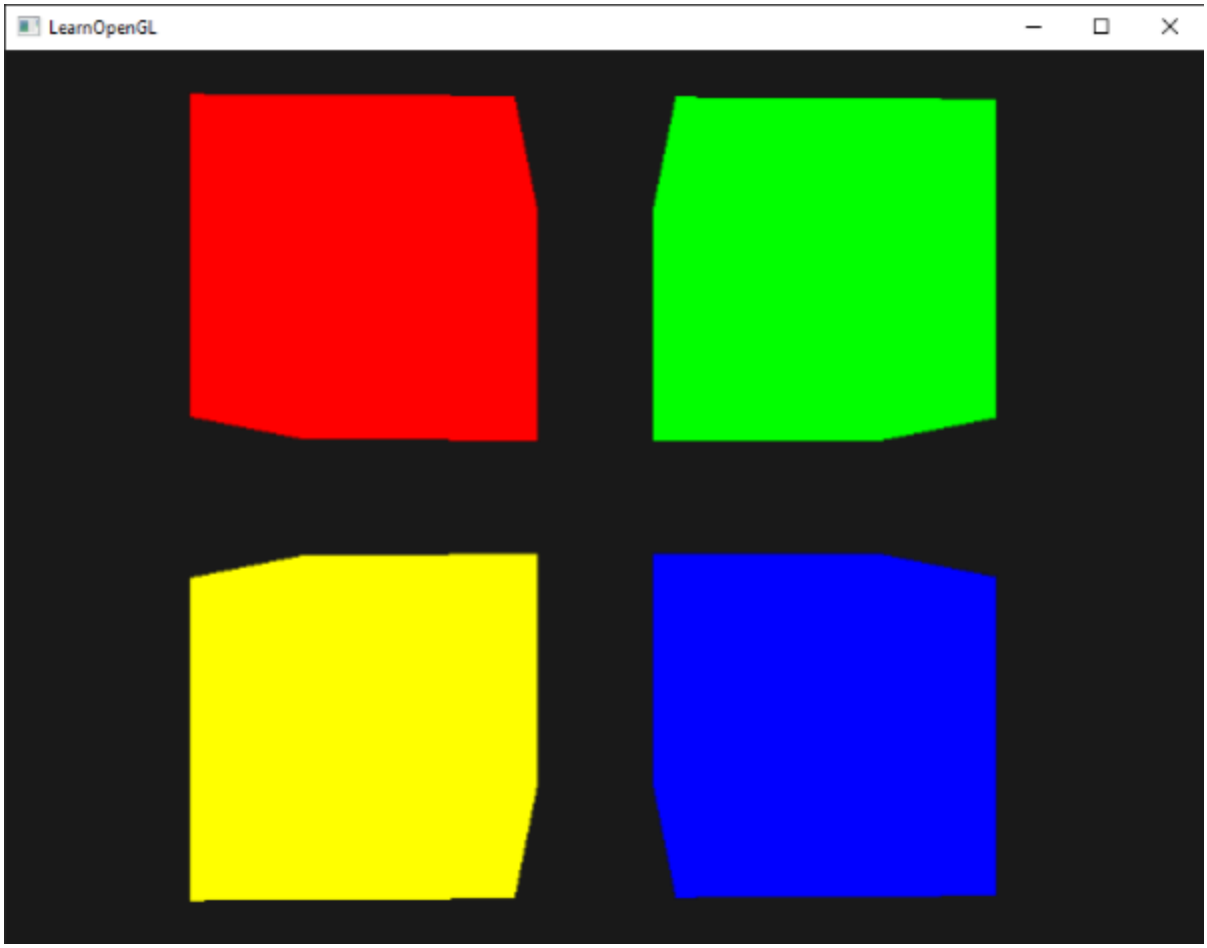
Here we store the first half of the uniform buffer with the projection matrix. Then before we render the objects each frame we update the second half of the buffer with the view matrix:

```
glm::mat4 view = camera.GetViewMatrix();
glBindBuffer(GL_UNIFORM_BUFFER, uboMatrices);
glBufferSubData(GL_UNIFORM_BUFFER, sizeof(glm::mat4), sizeof(glm::mat4),
glm::value_ptr(view));
glBindBuffer(GL_UNIFORM_BUFFER, 0);
```

And that's it for uniform buffer objects. Each vertex shader that contains a `Matrices` uniform block will now contain the data stored in `uboMatrices`. So if we now were to draw 4 cubes using 4 different shaders, their projection and view matrix should be the same:

```
glBindVertexArray(cubeVAO);
shaderRed.use();
glm::mat4 model = glm::mat4(1.0f);
model = glm::translate(model, glm::vec3(-0.75f, 0.75f, 0.0f));      // move top-left
shaderRed.setMat4("model", model);
glDrawArrays(GL_TRIANGLES, 0, 36);
// ... draw Green Cube
// ... draw Blue Cube
// ... draw Yellow Cube
```

The only uniform we still need to set is the `model` uniform. Using uniform buffer objects in a scenario like this saves us from quite a few uniform calls per shader. The result looks something like this:



Each of the cubes is moved to one side of the window by translating the model matrix and, thanks to the different fragment shaders, their colors differ per object. This is a relatively simple scenario of where we could use uniform buffer objects, but any large rendering application can have over hundreds of shader programs active which is where uniform buffer objects really start to shine.

You can find the full source code of the uniform example application here.

Uniform buffer objects have several advantages over single uniforms. First, setting a lot of uniforms at once is faster than setting multiple uniforms one at a time. Second, if you want to change the same uniform over several shaders, it is much easier to change a uniform once in a uniform buffer. One last advantage that is not immediately apparent is that you can use a lot more uniforms in shaders using uniform buffer objects. OpenGL has a limit to how much uniform data it can handle which can be queried with `GL_MAX_VERTEX_UNIFORM_COMPONENTS`. When using uniform buffer objects, this limit is much higher. So whenever you reach a maximum number of uniforms (when doing skeletal animation for example) there's always uniform buffer objects.