Computer Graphics II
– Advanced Data/GLSL

Kai Lawonn

Introduction

- So far we used buffers in OpenGL to store data for quite some time
- More interesting ways to manipulate buffers and also other interesting methods to pass large amounts of data to the shaders via textures
- Will discuss more interesting buffer functions and how we can use texture objects to store large amounts of data

Introduction

- A buffer in OpenGL is only an object that manages a certain piece of memory and nothing more
- We give a meaning to a buffer when binding it to a specific buffer target
- A buffer is only a vertex array buffer when we bind it to GL_ARRAY_BUFFER, but we could just as easily bind it to GL_ELEMENT_ARRAY_BUFFER
- OpenGL internally stores a buffer per target and based on the target, processes the buffers differently

Introduction

- We managed the memory of buffer objects by calling glBufferData which allocates a piece of memory and adds data into this memory
- If we were to pass NULL as its data argument, the function would only allocate memory and not fill it
- This is useful if we first want to reserve a specific amount of memory and later come back to this buffer to fill it piece by piece

Introduction

- Instead of filling the entire buffer, can also fill specific regions of the buffer by calling glBufferSubData
- This function expects a buffer target, an offset, the size of the data and the actual data as its arguments
- With this we can give an offset that specifies from where we want to fill the buffer
- This allows us to insert/update only certain parts of the buffer's memory

Introduction

- The buffer should have enough allocated memory so a call to glBufferData is necessary before calling glBufferSubData on the buffer:

```
glBufferSubData(GL_ARRAY_BUFFER, 24, sizeof(data), &data);
```

- Buffer, offset, size, data

Introduction

- Another method for getting data into a buffer is to ask for a pointer to the buffer's memory and directly copy the data to the buffer
- By calling glMapBuffer OpenGL returns a pointer to the currently bound buffer's memory for us to operate on:

```
float data[] = { 0.5f, 1.0f, -0.35f, ...};
glBindBuffer(GL_ARRAY_BUFFER, buffer);
// get pointer
void* ptr = glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);
// now copy data into memory
memcpy(ptr, data, sizeof(data));
// make sure to tell OpenGL
glUnmapBuffer(GL_ARRAY_BUFFER);
```

Introduction

- Tell OpenGL we are finished with the pointer operations: glUnmapBuffer
- By unmapping, the pointer becomes invalid and the function returns GL_TRUE if OpenGL was able to map the data successfully:

```
float data[] = { 0.5f, 1.0f, -0.35f, ...};
glBindBuffer(GL_ARRAY_BUFFER, buffer);
// get pointer
void* ptr = glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);
// now copy data into memory
memcpy(ptr, data, sizeof(data));
// make sure to tell OpenGL
glUnmapBuffer(GL_ARRAY_BUFFER);
```

Introduction

- Using glMapBuffer is useful to directly map data to a buffer, without first storing it in temporary memory
- Think of directly reading data from file and copying into the buffer’s memory

Batching Vertex Attributes

Introduction

- Using glVertexAttribPointer we were able to specify the attribute layout of the vertex array buffer's content
- Within the vertex array buffer we interleaved the attributes; that is, we placed the position, normal and/or texture coordinates next to each other for each vertex
- Now that we know a bit more about buffers we could take a different approach

Introduction

- We could also batch the vector data into large chunks per attribute type instead of interleaving them
- Instead of an interleaved layout:
  123123123123
  we take a batched approach:
  111122223333

Batching

- When loading vertex data from file you generally retrieve an array of positions, an array of normals and/or an array of texture coordinates
- Cost effort combining these into one large array of interleaved data
- Batching is then an easier solution, implement using glBufferSubData:

```
float positions[] = { ... };
float normals[] = { ... };
float tex[] = { ... };
// fill buffer
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(positions), &positions);
glBufferSubData(GL_ARRAY_BUFFER, sizeof(positions), sizeof(normals),
&normals);
glBufferSubData(GL_ARRAY_BUFFER, sizeof(positions) + sizeof(normals),
sizeof(tex), &tex);
```

Batching

- Directly transfer the attribute arrays as a whole into the buffer
- Could also combined them in one large array and fill the buffer right away using glBufferData
- Have to update the vertex attribute pointers to reflect these changes:

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), 0);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float),
        (void*)(sizeof(positions)));
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(float),
        (void*)(sizeof(positions) + sizeof(normals)));
```

Batching

- This gives us yet another approach of setting and specifying vertex attributes
- Using either approach has no immediate benefit to OpenGL, it is mostly a more organized way to set vertex attributes
- The approach you want to use is purely based on your preference and the type of application

Copying Buffers

Copying Buffers

- Once your buffers are filled with data, we may want to share it with other buffers or copy the buffer's content into another buffer
- The function glCopyBufferSubData allows to copy the data from one buffer to another buffer
- The function's prototype is as follows:

```
void glCopyBufferSubData(GLenum readtarget, GLenum writetarget, GLintptr
readoffset, GLintptr writeoffset, GLsizeiptr size);
```

Copying Buffers

```
void glCopyBufferSubData(GLenum readtarget, GLenum writetarget, GLintptr
readoffset, GLintptr writeoffset, GLsizeiptr size);
```

- The readtarget and writetarget parameters expect to give the buffer targets that we want to copy from and to (e.g., copy from VERTEX_ARRAY_BUFFER to VERTEX_ELEMENT_ARRAY_BUFFER)
- The buffers currently bound to those buffer targets will then be affected

Copying Buffers

- What if we wanted to read and write data into two different vertex array buffers?
- Cannot bind two buffers at the same time to the same buffer target
- For this reason only, OpenGL gives us two more buffer targets called GL_COPY_READ_BUFFER and GL_COPY_WRITE_BUFFER
- We then bind the buffers of our choice to these new buffer targets and set those targets as the readtarget and writetarget argument

Copying Buffers

- glCopyBufferSubData then reads data of a given size from a given readoffset and writes it into the writetarget buffer at writeoffset
- An example of copying the content of two vertex array buffers is shown below:

```
float vertexData[] = { ... };
glBindBuffer(GL_COPY_READ_BUFFER, vbo1);
glBindBuffer(GL_COPY_WRITE_BUFFER, vbo2);
glCopyBufferSubData(GL_COPY_READ_BUFFER, GL_COPY_WRITE_BUFFER, 0, 0,
sizeof(vertexData));
```

Copying Buffers

- Could also done this by only binding the writetarget buffer to one of the new buffer target types:

```
float vertexData[] = { ... };
glBindBuffer(GL_ARRAY_BUFFER, vbo1);
glBindBuffer(GL_COPY_WRITE_BUFFER, vbo2);
glCopyBufferSubData(GL_ARRAY_BUFFER, GL_COPY_WRITE_BUFFER, 0, 0,
sizeof(vertexData));
```

Copying Buffers

- With these knowledge we can already use his in more interesting ways
- The further we get in OpenGL the more useful these new buffer methods start to become
- Next, we will discuss uniform buffer objects, which make good use of glBufferSubData

Advanced GLSL

Introduction

- This part will not show advanced new features that give an enormous boost to the visual quality
- It is more or less into some interesting aspects of GLSL and some nice tricks that might help in the future
- Basically, some good to knows and features when creating OpenGL applications in combination with GLSL
- We will discuss some built-in variables, new ways to organize shader's input and output and a very useful tool called uniform buffer objects

GLSL's Built–in Variables

- Shaders are minimal, if we need data from any other source outside the current shader we'll have to pass data around
- We learned to do this via vertex attributes, uniforms and samplers
- There are however a few extra variables defined by GLSL prefixed with gl_ that give us an extra means to gather and/or write data
- E.g.: gl_Position that is the output vector of the vertex shader and the fragment shader's gl_FragCoord

GLSL's Built–in Variables

- We will discuss a few built-in variables in GLSL and explain how they might benefit us
- We will not discuss all built-in variables that exist in GLSL
- Check the WIKI (below) for all built-in variables

https://www.khronos.org/opengl/wiki/Built-in_Variable_(GLSL)

Vertex Shader Variables

- gl_Position: is the clip-space output position vector of the vertex shader
- Setting gl_Position in the vertex shader is a strict requirement if you want to render anything on the screen
- Nothing we haven’t seen before

Vertex Shader Variables

- gl_PointSize:
- One of the render primitives we're able to choose from is GL_POINTS (single vertex is a primitive and rendered as a point)
- It is possible to set the size of the points being rendered via OpenGL's glPointSize function, but we can also influence this value in the vertex shader

Vertex Shader Variables

- gl_PointSize is an GLSL float output variable (set the point's width and height in pixels)
- In the vertex shader, we can influence this point value per vertex
- Influencing the point sizes in the vertex shader is disabled by default, to enable it:

```
glEnable(GL_PROGRAM_POINT_SIZE);
```

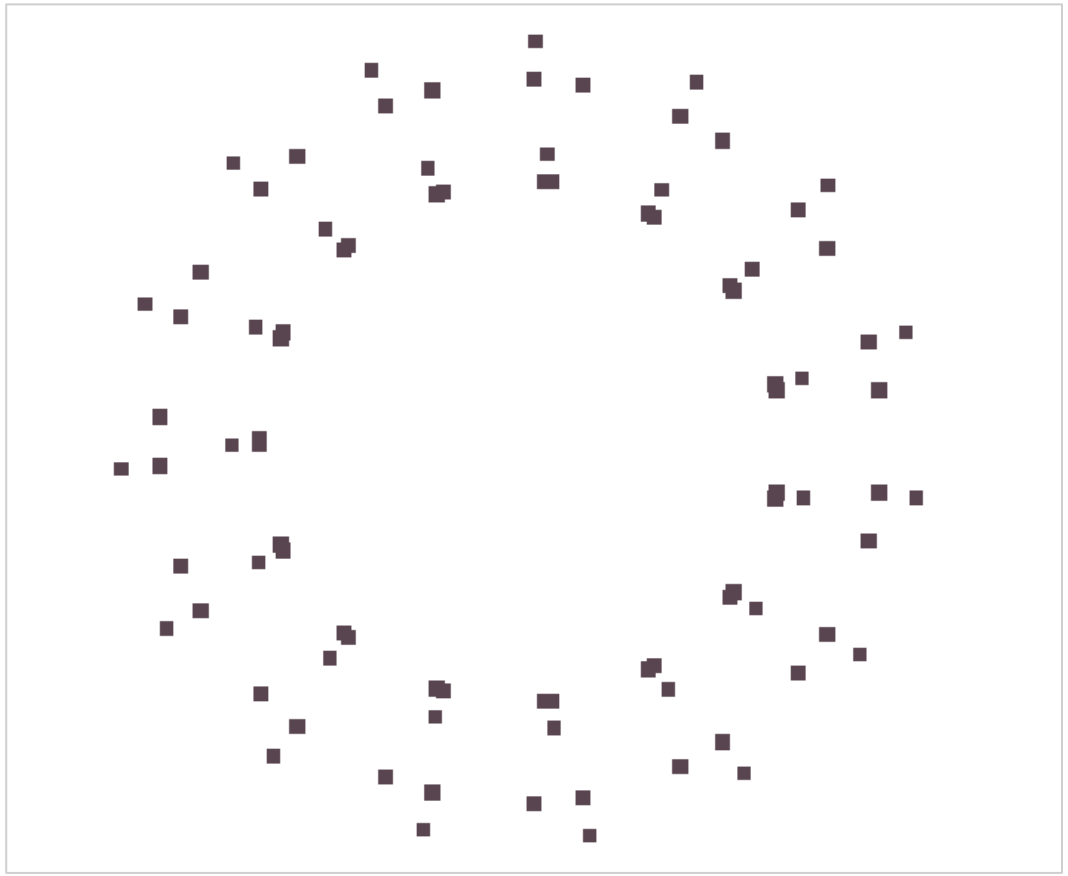
Vertex Shader Variables

- Example: set the point size equal to the clip-space position's z value (equal to the vertex's distance to the viewer)
- The point size should then increase the further we are from the vertices as the viewer

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

F5…

- … size depends on the distance



Vertex Shader Variables

- gl_VertexID:
- gl_Position & gl_PointSize are output variables → influence the result
- Interesting input variable (read only) called gl_VertexID
- gl_VertexID = current ID of the vertex to draw
- When doing indexed rendering (with glDrawElements) this variable holds the current index of the vertex
- When drawing without indices (via glDrawArrays) this is 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

Fragment Shader Variables

- gl_FragCoord:
- gl_FragCoord.z is equal to the depth value of that particular fragment
- Can also use x and y of the vector for some interesting effects
- gl_FragCoord's x and y component are the window-space coordinates of the fragment (start from bottom-left)
- Specified a window of 800x600 with glViewport → x values between 0 and 800, and y values between 0 and 600

Fragment Shader Variables

- gl_FragCoord:
- Calculate a different color based on the coordinate of the fragment
- A common usage for the gl_FragCoord variable is for comparing visual output of different fragment calculations

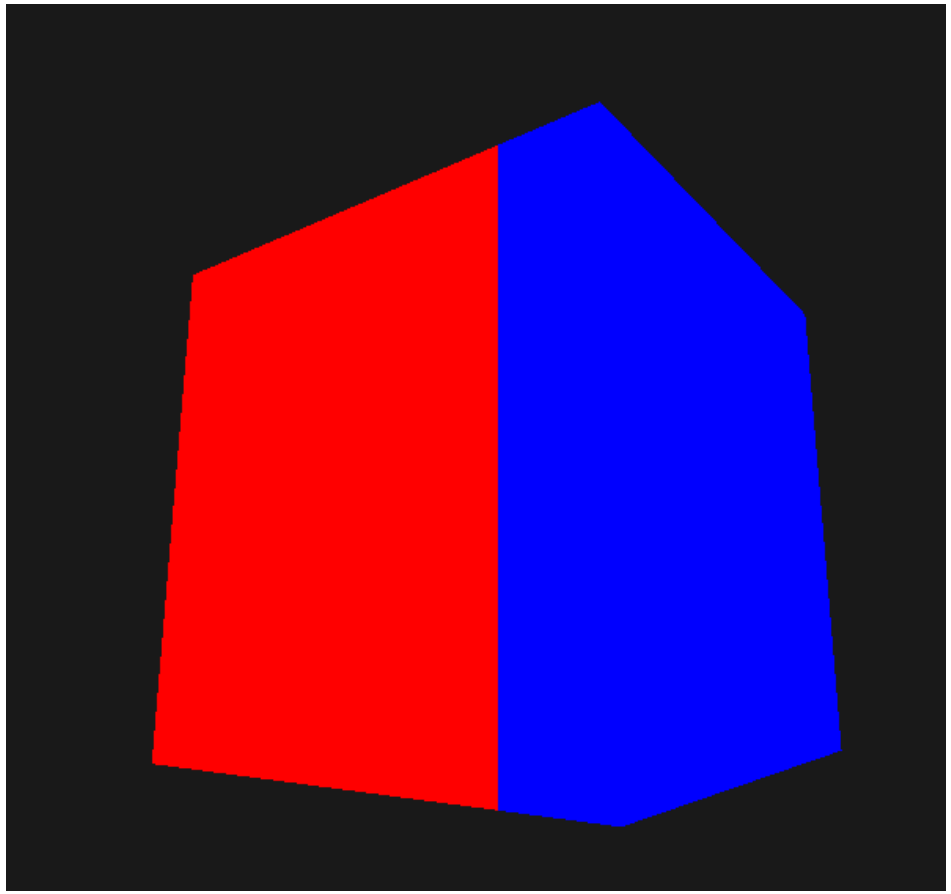
Fragment Shader Variables

- gl_FragCoord:
- E.g., split the screen in two
- An example fragment shader that outputs a different color based on the fragment's window coordinates:

```
void main()
{
    if(gl_FragCoord.x < 400)
        FragColor = vec4(1.0, 0.0, 0.0, 1.0);
    else
        FragColor = vec4(0.0, 0.0, 1.0, 1.0);
}
```

F5…

- … a color-divided cube



Fragment Shader Variables

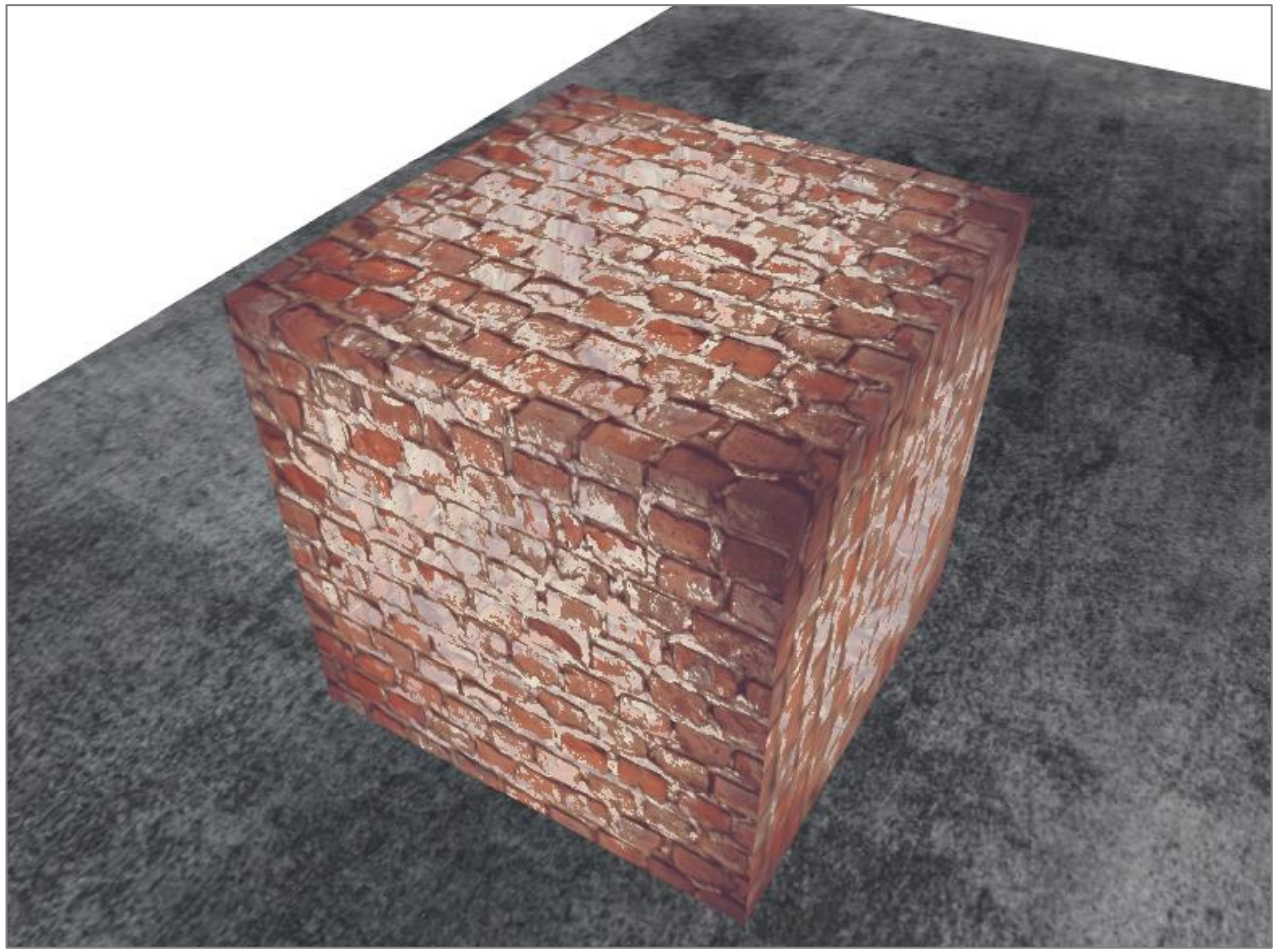
- gl_FragCoord:
- Could calculate different fragment shader results and display each of them on a different side of the window
- This is great for testing out different lighting techniques for example

Fragment Shader Variables

- gl_FrontFacing:
- We already mentioned that OpenGL is able to figure out if a face is a front or back face (winding order)
- gl_FrontFacing tells us if the current fragment is part of a front-facing or a back-facing face
- Could calculate different colors for front faces for example

Remember: Lec2_Blending

- Already used this



Fragment Shader Variables

- gl_FrontFacing:
- Note if face culling is enabled, we won't see any faces inside the box → and using gl_FrontFacing would then be pointlessc

Fragment Shader Variables

- gl_FragDepth:
- gl_FragCoord input variable that read window-space coordinates and get the depth value of the current fragment (read-only)
- It is possible to set the depth value of the fragment
- Output variable gl_FragDepth to set the depth value of the fragment

Fragment Shader Variables

- gl_FragDepth:
- To actually set the depth value in the shader we simply write a float value between 0.0 and 1.0 to the output variable:

```
gl_FragDepth = 0.0; // this fragment now has a depth value of 0.0
```

Fragment Shader Variables

- gl_FragDepth:
- If the shader does not write a value to gl_FragDepth the variable will automatically take its value from gl_FragCoord.z
- Set depth value manually has a disadvantage: OpenGL disables all early depth testing

Fragment Shader Variables

- gl_FragDepth:
- Writing to gl_FragDepth, we should consider this disadvantage
- From OpenGL 4.2 however, we can still sort of mediate between both sides by redeclaring the gl_FragDepth variable at the top of the fragment shader with a depth condition:

```
layout (depth_<condition>) out float gl_FragDepth;
```

Fragment Shader Variables

- gl_FragDepth:
- This condition can take the following values:

Condition	Description
any	The default value. Early depth testing is disabled and you lose most performance.
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.

Fragment Shader Variables

- gl_FragDepth:
- Specifying greater or less, OpenGL can make the assumption that only depth values larger or smaller than the fragment’s depth value are used
- This way OpenGL is still able to do an early depth test in cases where the depth value is smaller than the fragment’s depth value

Fragment Shader Variables

- gl_FragDepth:
- An example of where we increment the depth value in the fragment shader, but still want to preserve some of the early depth testing is shown in the fragment shader below:

```
#version 420 core // note the GLSL version!
out vec4 FragColor;
layout (depth_greater) out float gl_FragDepth;

void main()
{
    FragColor = vec4(1.0);
    gl_FragDepth = gl_FragCoord.z + 0.1;
}
```

Interface Blocks

Introduction

- Every time we send data from the vertex to the fragment shader, we declared several matching input/output variables
- Declaring these one at a time is the easiest way to send data from one shader to another
- When applications become larger, send more than a few variables over which may include arrays and/or structs

Introduction

- GLSL offers interface blocks that allows to group variables
- The declaration is like a struct declaration, except that it is now declared using an in or out keyword based on the block being an input or an output block

Introduction

- Interface called vs_out:

```
#version 330 core
…
layout (location = 1) in vec2 aTexCoords;
…
out VS_OUT
{
    vec2 TexCoords;
} vs_out;

void main()
{
    gl_Position = …;
    vs_out.TexCoords = aTexCoords;
}
```

Introduction

- out VS_OUT → in VS_OUT

```
#version 330 core
out vec4 FragColor;
uniform sampler2D texture;

in VS_OUT
{
    vec2 TexCoords;
} fs_in;

void main()
{
    FragColor = texture(texture, fs_in.TexCoords);
}
```

Introduction

- As long as both interface block names are equal, their corresponding input and output is matched together
- This is another useful feature that helps organize your code

Uniform Buffer Objects

Introduction

- Learned some pretty cool tricks so far, but also a few annoyances
- E.g., when using several shader, have to set uniform variables where most of them are exactly the same for each shader - so why bother to even set them again?

Introduction

- Uniform buffer objects (UBO) allow to declare a set of global uniform variables that remain the same over several shader programs
- Using UBOs, have to set the relevant uniforms only once
- Still have to manually set uniforms that are unique per shader
- Creating and configuring a uniform buffer object requires a bit of work though

Introduction

- UBO is a buffer like any other buffer: create via glGenBuffers, bind it, and store all the relevant uniform data into the buffer (later)
- First, take a simple vertex shader and store our projection and view matrix in a so called uniform block

```
#version 330 core
…
layout (std140) uniform Matrices
{
    mat4 projection;
    mat4 view;
};
…
void main(){
    gl_Position = projection * view * model * vec4(aPos, 1.0);
}
```

Introduction

- Mostly, we set a projection and view uniform matrix each render iteration for each shader we're using → perfect example where uniform buffer objects become useful (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 declared this uniform block has access to the matrices
- The layout (std140) statement means that the currently defined uniform block uses a specific memory layout for its content; this statement sets the uniform block layout

Uniform Block Layout

Introduction

- The content of a uniform block is stored in a buffer object (nothing more than a reserved piece of 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 corresponds to which uniform variables in the shader

Introduction

- 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;
};
```

Uniform Block Layout

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

- Want to know the size (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
- Size of the elements is stated in OpenGL and directly corresponds to C++ data types; vectors and matrices being (large) arrays of floats
- OpenGL does not clearly state the spacing between the variables
- This allows the hardware to position variables as it sees fit
- Some 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

Uniform Block Layout

- By default GLSL uses a uniform memory layout (shared layout) - shared because once the offsets are defined by the hardware, they are consistently shared between multiple programs
- With this 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
- (Can query this information with functions like glGetUniformIndices, but that is out of the scope)

Uniform Block Layout

- Shared layout gives space-saving optimizations, but need to query each offset for each uniform variable → a lot of work
- General practice is to not use the shared layout, but the std140 layout
- std140 layout explicitly states the memory layout for each variable type by stating their respective offsets governed by a set of rules
- Since this is explicitly mentioned we can manually figure out the offsets for each variable

Uniform Block Layout

- Each variable has a base alignment - equal to the space a variable takes (including padding) within a uniform block - this base alignment is calculated using the std140 layout rules
- For each variable, we calculate its aligned offset which is 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**

Uniform Block Layout

- Exact layout rules can be found at the link below
- We list common rules (each variable are defined to be four-byte quantities with each entity of 4 bytes being 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.

https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_uniform_buffer_object.txt

Uniform Block Layout

- Example: uniform block 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 (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
};
```

Uniform Block Layout

- **The aligned byte offset of a variable must be equal to a multiple of its base alignment**

```
layout (std140) uniform ExampleBlock
{                       // base alignment// aligned offset
    float value;        // 4               // 0
    vec3 vector;        // 16              // 16 (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
};
```

Uniform Block Layout

- With the calculated std140 layout offset values, we can fill the buffer with the variable data at each offset with , e.g., 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

Uniform Block Layout

- By adding layout (std140) before 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, e.g., shared -, packed layout
- With packed layout, 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 might differ per shader

Using Uniform Buffers

Using Uniform Buffers

- To use them, we need to create a UBO (glGenBuffers)
- Then, bind it to GL_UNIFORM_BUFFER and allocate enough memory (glBufferData)

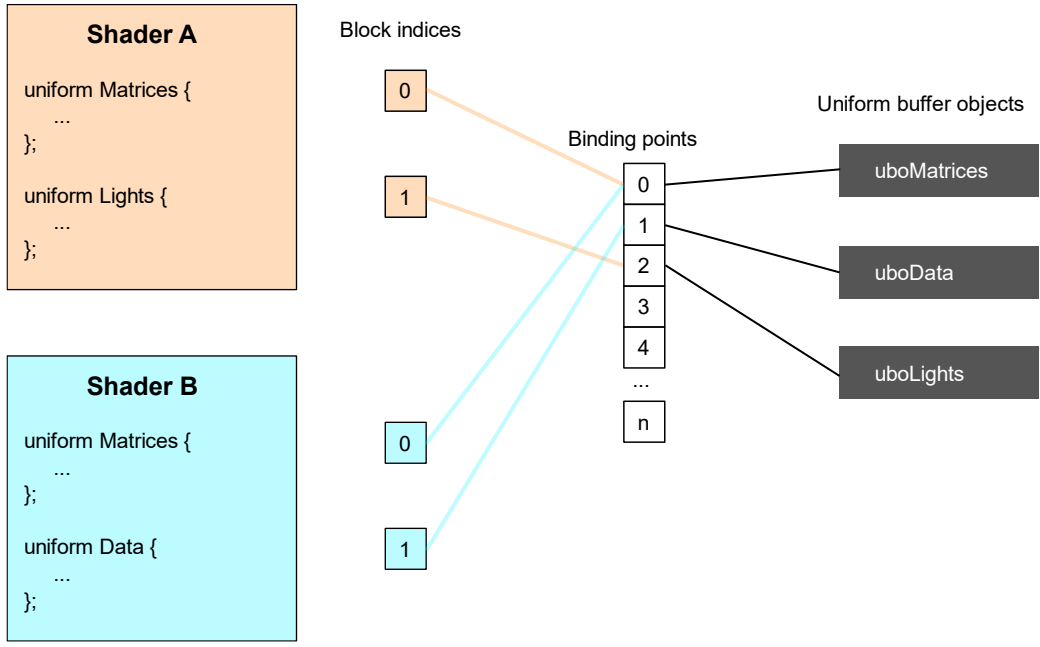
```
unsigned int uboExampleBlock;
glGenBuffers(1, &uboExampleBlock);
glBindBuffer(GL_UNIFORM_BUFFER, uboExampleBlock);
glBufferData(GL_UNIFORM_BUFFER, 152, NULL, GL_STATIC_DRAW);// allocate 150
                                                           // bytes of memory
glBindBuffer(GL_UNIFORM_BUFFER, 0);
```

Using Uniform Buffers

- When we want to update or insert data into the buffer, we bind to uboExampleBlock and use glBufferSubData to update its memory
- Only 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?

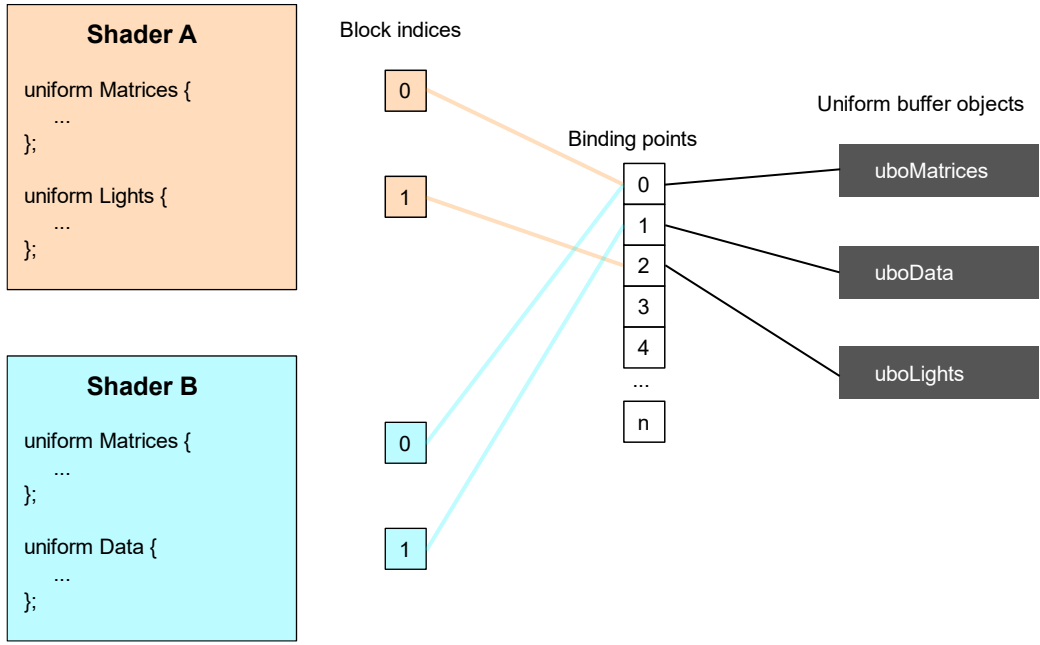
Using Uniform Buffers

- In the OpenGL context is a number of binding points defined where we can link a uniform buffer to
- Once we created a uniform buffer, link it to one of those binding points and also link the uniform block in the shader to same binding point, effectively linking those to each other
- The following diagram illustrates this:



Using Uniform Buffers

- Can bind multiple uniform buffers to different binding points
- Because shader A and shader B have a uniform block linked to the same binding point 0 their uniform blocks share the same uniform data found in uboMatrices



Using Uniform Buffers

- Set uniform block to a specific binding point: glUniformBlockBinding (arguments: program object, uniform block index, and binding point to link to)
- Uniform block index is a location index of the defined uniform block in the shader (retrieved via glGetUniformBlockIndex, accepts 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);
```

Using Uniform Buffers

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

Using Uniform Buffers

From OpenGL version 4.2 and onwards, possible to store the binding point of a uniform block explicitly in the shader (via 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 { ... };
```

Using Uniform Buffers

- Need to bind the UBO to the same binding point: with either glBindBufferBase or glBindBufferRange.

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

Using Uniform Buffers

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

- glBindbufferBase expects a target, a binding point index and a uniform buffer object as its arguments (here: links uboExampleBlock to binding point 2)
- glBindBufferRange expects an extra offset and size parameter (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

Using Uniform Buffers

- Everything is set up, and can start adding data to the uniform buffer.
- 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 repr. as 4 bytes, so we store it in an int
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

Example

Introduction

- We continually been using 3 matrices: projection, view and model
- 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

Example

- Store the projection & view matrix in a uniform block called Matrices
- Do not store the model matrix (model matrix tends to change quite frequently between shaders), wouldn't really benefit from UBOs

```
#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);}
```

Example

- In our example application: want to display 4 cubes, each cube is displayed using a different shader program
- Each of the 4 shader programs uses the same vertex shader, but has a different fragment shader that only outputs a single color that differs per shader

Example

- 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");
// then we link each shader's uniform block to this uniform binding point
glUniformBlockBinding(shaderRed.ID, uniformBlockIndexRed, 0);
glUniformBlockBinding(shaderGreen.ID, uniformBlockIndexGreen, 0);
glUniformBlockBinding(shaderBlue.ID, uniformBlockIndexBlue, 0);
glUniformBlockBinding(shaderYellow.ID, uniformBlockIndexYellow, 0);
```

Example

- Next, create UBO and also bind the 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);
// define the range of the buffer that links to a uniform binding point
glBindBufferRange(GL_UNIFORM_BUFFER, 0, uboMatrices, 0, 2 *
sizeof(glm::mat4));
```

Example

- Allocate enough memory: 2 times the size of glm::mat4
- The size of GLM's matrix types correspond directly to mat4 in GLSL
- Then, link a specific range of the buffer, in this case is the entire buffer, to binding point 0

Example

- Now, fill the buffer: keep the field of view value constant of the projection matrix (so no more camera zoom), so define it once → insert this into the buffer only once as well
- Allocated enough memory in the buffer object → can use glBufferSubData to store the projection matrix before game loop:

```
glm::mat4 projection = glm::perspective(45.0f, (float)SCR_WIDTH /
                                                (float)SCR_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);
```

Example

- Store the first half of the uniform buffer with the projection matrix
- Before we draw the objects each render iteration we then 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);
```

Example

- That's it…
- 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 remain the same:

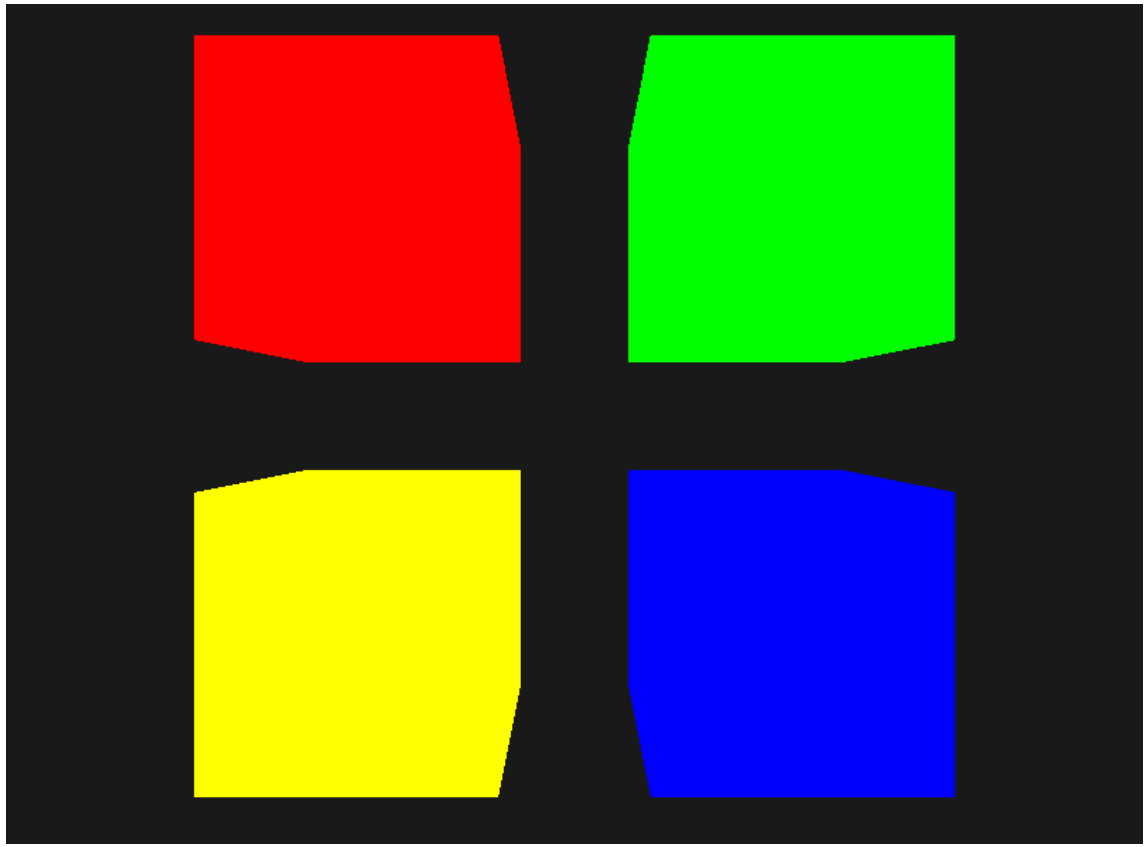
```
// RED Cube
glBindVertexArray(cubeVAO);
shaderRed.use();
glm::mat4 model = glm::mat4(1.0f);
model = glm::translate(model, glm::vec3(-0.75f, 0.75f, 0.0f));
shaderRed.setMat4("model", model);
glDrawArrays(GL_TRIANGLES, 0, 36);
// GREEN, BLUE, YELLOW Cube…
```

Example

- 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

F5…

- Each cube is moved (altering model matrix), with different fragment shaders different colors
- Simple example might use uniform buffer objects, but any large rendering application could have over hundreds of shader programs active → UBO shine



Note

- UBOs several advantages over single uniforms:
  - Setting a lot uniforms once faster than setting multiple uniforms one at a time
  - Changing the same uniform over several shaders, it is much easier to change a uniform once in a uniform buffer
  - Not immediately apparent that you can use a lot more uniforms in shaders using uniform buffer objects (OpenGL has a limit, queried with GL_MAX_VERTEX_UNIFORM_COMPONENTS) → with UBOs, limit is much higher, when reach a maximum number of uniforms (when doing skeletal animation for example) you could always use UBOs

Questions???