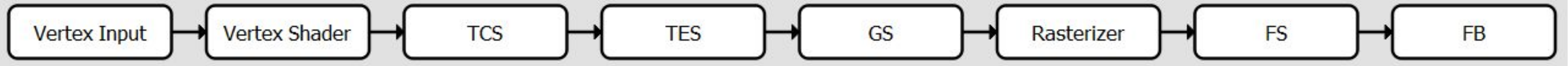
OpenGL

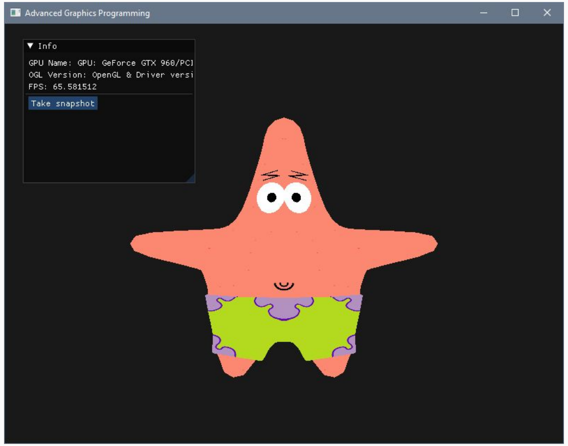
Uniform blocks and uniform buffers

Advanced Graphics Programming

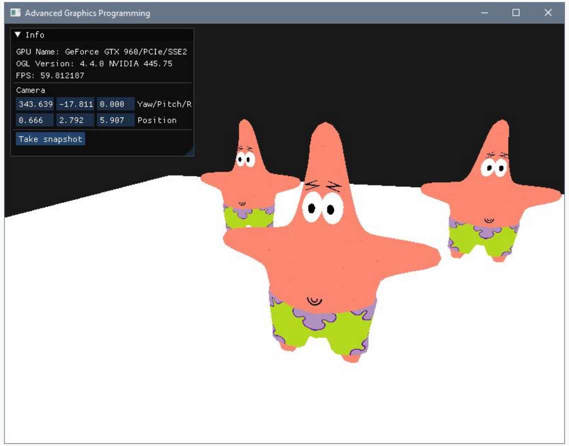
We will add transforms, then lights



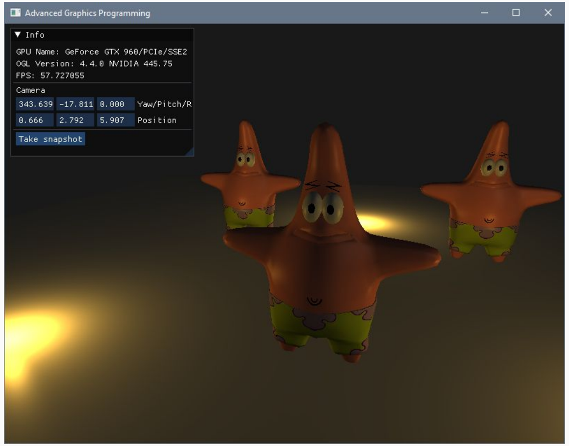
Vertex Input → Vertex Shader → TCS → TES → GS → Rasterizer → FS → FB



```
▼ Info
GPU Name: GPU: GeForce GTX 960/PCI
OGL Version: OpenGL & Driver versi
FPS: 65.581512
Take snapshot
```

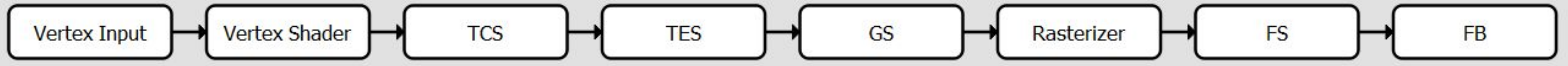


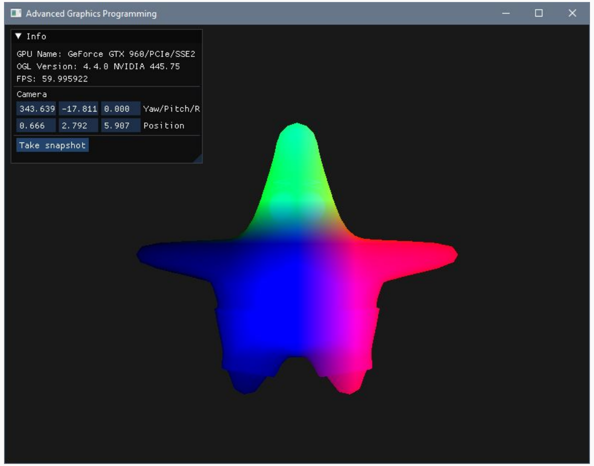
```
▼ Info
GPU Name: GeForce GTX 960/PCIe/SSE2
OGL Version: 4.4.0 NVIDIA 445.75
FPS: 59.812187
Camera
343.639  -17.811  0.000  Yaw/Pitch/R
0.666    2.792    5.907  Position
Take snapshot
```



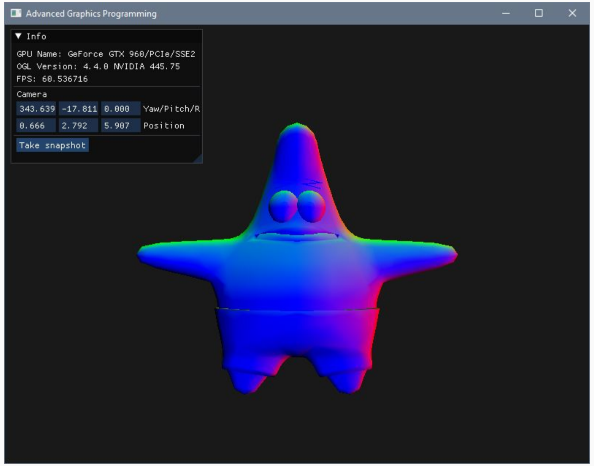
```
▼ Info
GPU Name: GeForce GTX 960/PCIe/SSE2
OGL Version: 4.4.0 NVIDIA 445.75
FPS: 57.727055
Camera
343.639  -17.811  0.000  Yaw/Pitch/R
0.666    2.792    5.907  Position
Take snapshot
```

Make sure to pass attributes VS -> FS





```
GPU Name: GeForce GTX 960/PCIe/SSE2
OGL Version: 4.4.0 NVIDIA 445.75
FPS: 59.995922
Camera
343.639  -17.811  0.000  Yaw/Pitch/R
0.666    2.792    5.907  Position
```



```
GPU Name: GeForce GTX 960/PCIe/SSE2
OGL Version: 4.4.0 NVIDIA 445.75
FPS: 60.536716
Camera
343.639  -17.811  0.000  Yaw/Pitch/R
0.666    2.792    5.907  Position
```

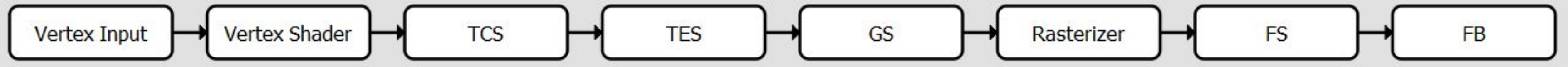
Vertex shader outputs...

```
out vec2 vTexCoord;
out vec3 vPosition;
out vec3 vNormal;
out vec3 vViewDir;
```

are fragment shader inputs

```
in vec2 vTexCoord;
in vec3 vPosition;
in vec3 vNormal;
in vec3 vViewDir;
```

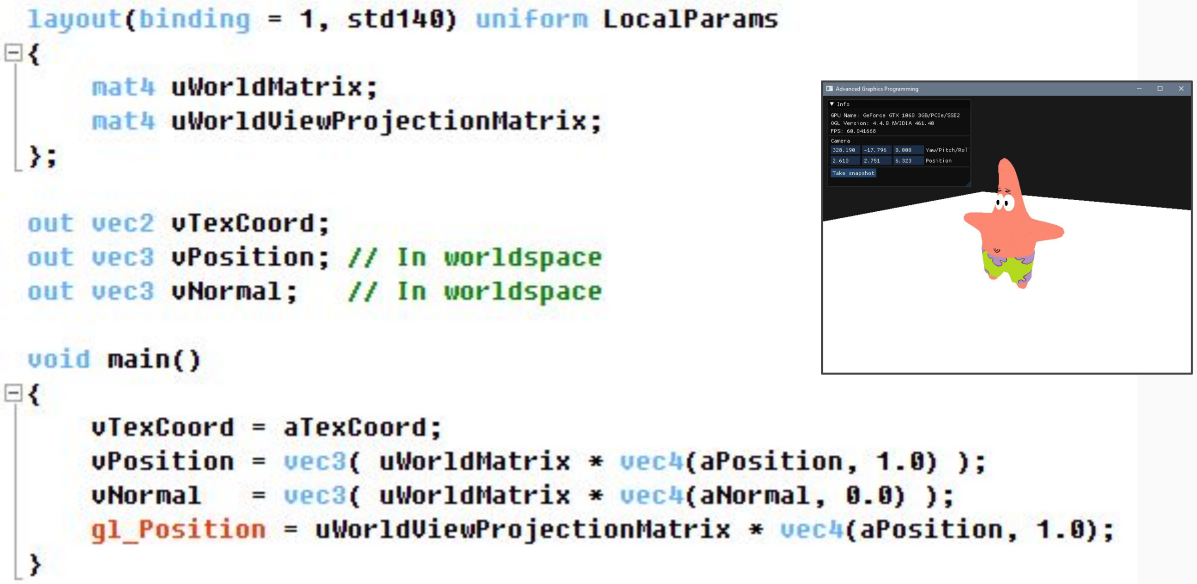
Uniform blocks



Uniform blocks define a data layout to provide input constants to shaders.

In this case, LocalParams defines an interface to pass per-draw-call inputs.

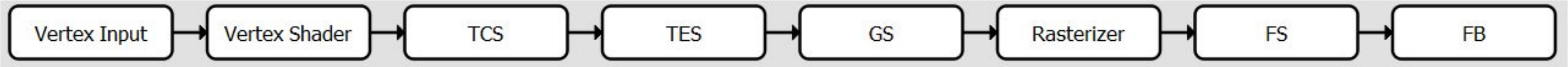
Vertex shader



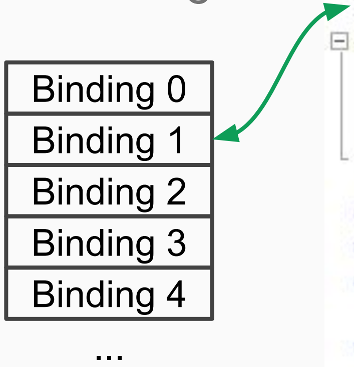
```
layout(binding = 1, std140) uniform LocalParams
{
    mat4 uWorldMatrix;
    mat4 uWorldViewProjectionMatrix;
};

out vec2 vTexCoord;
out vec3 vPosition; // In worldspace
out vec3 vNormal;   // In worldspace

void main()
{
    vTexCoord = aTexCoord;
    vPosition = vec3( uWorldMatrix * vec4(aPosition, 1.0) );
    vNormal   = vec3( uWorldMatrix * vec4(aNormal, 0.0) );
    gl_Position = uWorldViewProjectionMatrix * vec4(aPosition, 1.0);
}
```

```
GPU Name: GeForce GTX 1060 3GB/PCIe/SSE2
OGL Version: 4.4.0 NVIDIA 461.40
FPS: 60.041668
Camera
328.190  -17.796  0.000  Yaw/Pitch/Rol
2.610    2.751    6.323  Position
```

Uniform buffer bindings

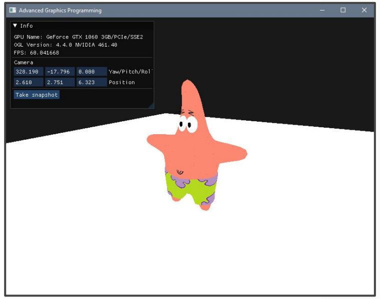


Uniform buffer bindings



Vertex shader

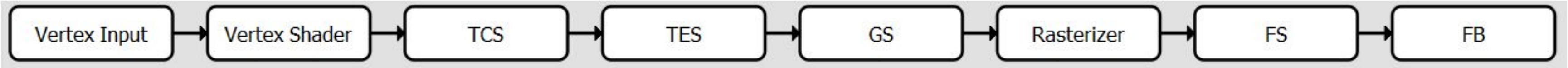
```
layout(binding = 1, std140) uniform LocalParams
{
    mat4 uWorldMatrix;
    mat4 uWorldViewProjectionMatrix;
};

out vec2 vTexCoord;
out vec3 vPosition; // In worldspace
out vec3 vNormal;   // In worldspace

void main()
{
    vTexCoord = aTexCoord;
    vPosition = vec3( uWorldMatrix * vec4(aPosition, 1.0) );
    vNormal   = vec3( uWorldMatrix * vec4(aNormal, 0.0) );
    gl_Position = uWorldViewProjectionMatrix * vec4(aPosition, 1.0);
}
```



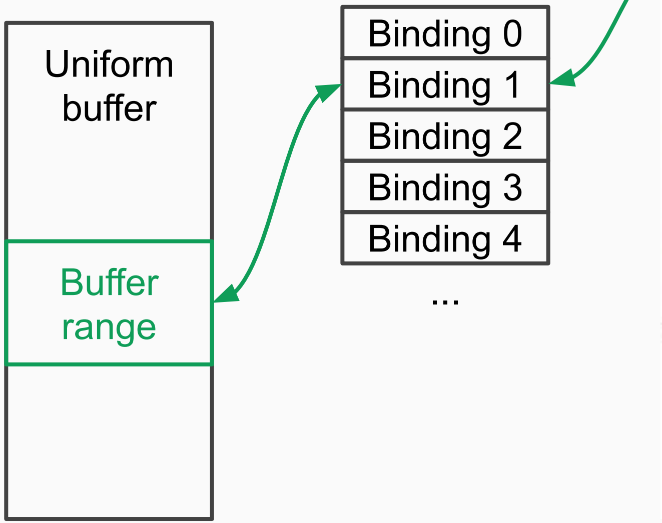
GPU Name: GeForce GTX 1060 3GB/PCIe/SSE2
OGL Version: 4.4.0 NVIDIA 461.40
FPS: 60.041668

328.190	-17.796	0.000	Yaw/Pitch/Rol
2.610	2.751	6.323	Position

Uniform buffer bindings


Vertex Input → Vertex Shader → TCS → TES → GS → Rasterizer → FS → FB

Uniform buffer bindings


Uniform buffer

Buffer range

Binding 0

Binding 1

Binding 2

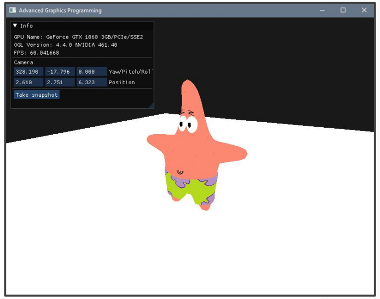
Binding 3

Binding 4

...

Vertex shader

```
layout(binding = 1, std140) uniform LocalParams
{
    mat4 uWorldMatrix;
    mat4 uWorldViewProjectionMatrix;
};

out vec2 vTexCoord;
out vec3 vPosition; // In worldspace
out vec3 vNormal;   // In worldspace

void main()
{
    vTexCoord = aTexCoord;
    vPosition = vec3( uWorldMatrix * vec4(aPosition, 1.0) );
    vNormal   = vec3( uWorldMatrix * vec4(aNormal, 0.0) );
    gl_Position = uWorldViewProjectionMatrix * vec4(aPosition, 1.0);
}
```

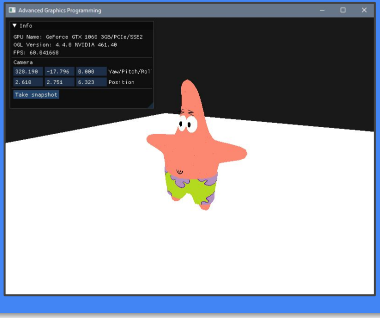


GPU Name: GeForce GTX 1060 3GB/PCIe/SSE2
OGL Version: 4.4.0 NVIDIA 461.40
FPS: 60.041668

Camera

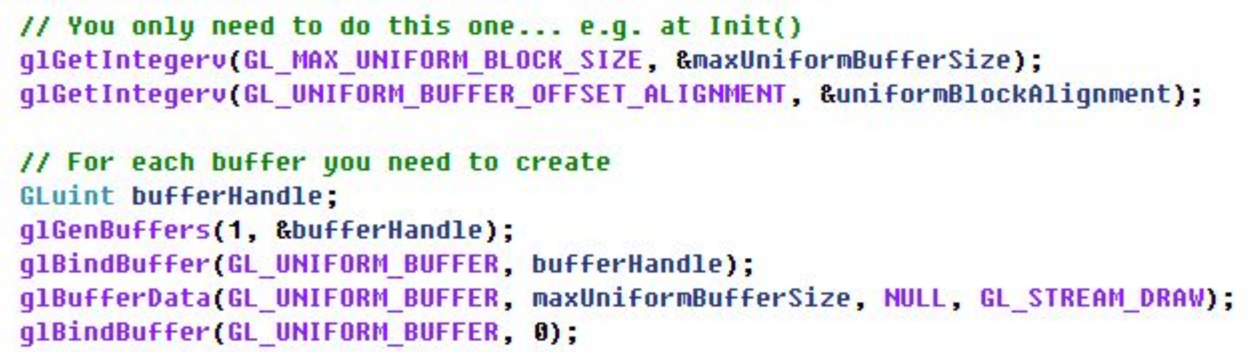
328.190	-17.796	0.000	Yaw/Pitch/Rol
2.610	2.751	6.323	Position

Creating uniform buffers

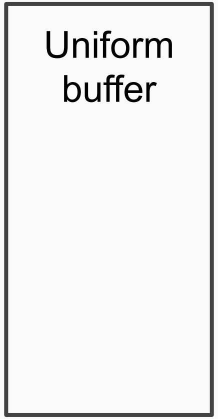


```
GPU Name: GeForce GTX 1060 3GB/PCIe/SSE2
OGL Version: 4.4.0 NVIDIA 461.40
FPS: 60.041668
Camera
328.190   -17.796   0.000   Yaw/Pitch/Rol
2.610     2.751     6.323   Position
```

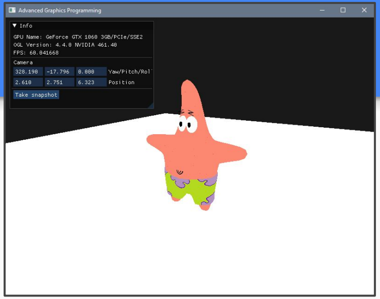
At initialization, we retrieve the maximum size allowed for uniform buffers, the alignment for each data block we will insert, and create some uniform buffer.



```
// You only need to do this one... e.g. at Init()
glGetIntegerv(GL_MAX_UNIFORM_BLOCK_SIZE, &maxUniformBufferSize);
glGetIntegerv(GL_UNIFORM_BUFFER_OFFSET_ALIGNMENT, &uniformBlockAlignment);

// For each buffer you need to create
GLuint bufferHandle;
glGenBuffers(1, &bufferHandle);
glBindBuffer(GL_UNIFORM_BUFFER, bufferHandle);
glBufferData(GL_UNIFORM_BUFFER, maxUniformBufferSize, NULL, GL_STREAM_DRAW);
glBindBuffer(GL_UNIFORM_BUFFER, 0);
```

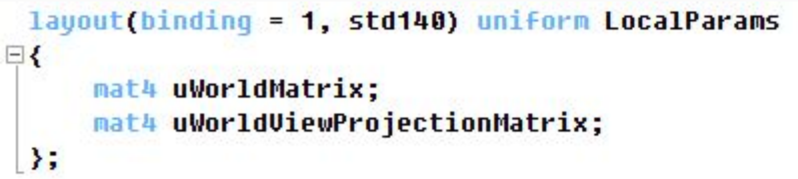


Filling uniform buffers



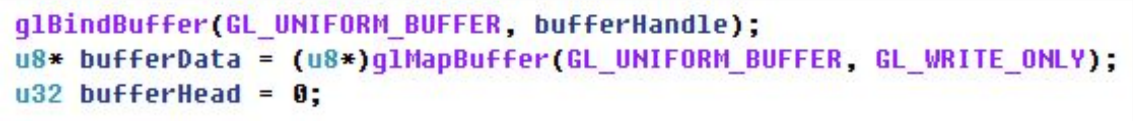
```
GPU Name: GeForce GTX 1060 3GB/PCIe/SSE2
OGL Version: 4.4.0 NVIDIA 461.40
FPS: 60.041668
Camera
328.190  -17.796  0.000  Yaw/Pitch/Rol
2.610    2.751    6.323  Position
```

Uniform block in the vertex shader

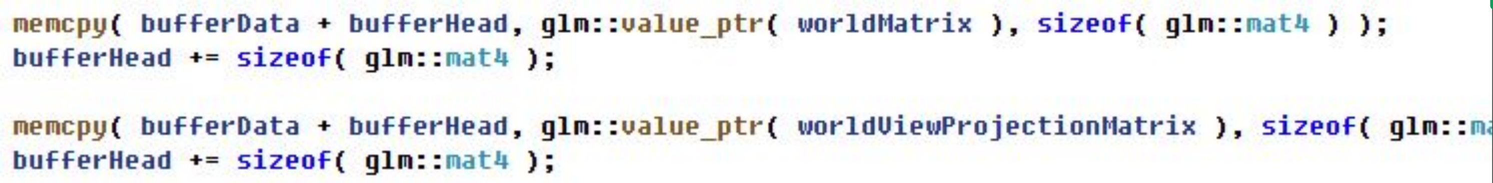


```
layout(binding = 1, std140) uniform LocalParams
{
    mat4 uWorldMatrix;
    mat4 uWorldViewProjectionMatrix;
};
```

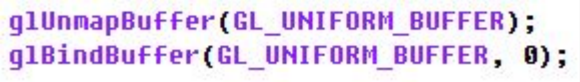
We have to push data into the buffer ordered according to the uniform block



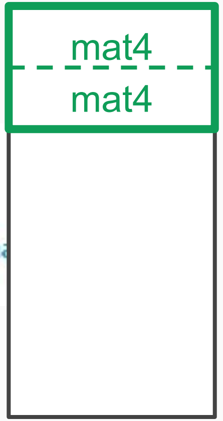
```
glBindBuffer(GL_UNIFORM_BUFFER, bufferHandle);
u8* bufferData = (u8*)glMapBuffer(GL_UNIFORM_BUFFER, GL_WRITE_ONLY);
u32 bufferHead = 0;
```



```
memcpy( bufferData + bufferHead, glm::value_ptr( worldMatrix ), sizeof( glm::mat4 ) );
bufferHead += sizeof( glm::mat4 );

memcpy( bufferData + bufferHead, glm::value_ptr( worldViewProjectionMatrix ), sizeof( glm::ma
bufferHead += sizeof( glm::mat4 );
```

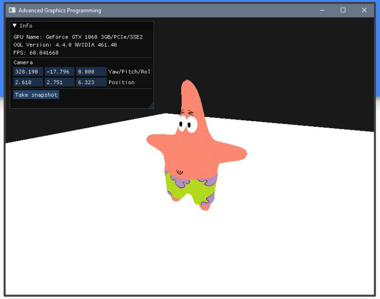


```
glUnmapBuffer(GL_UNIFORM_BUFFER);
glBindBuffer(GL_UNIFORM_BUFFER, 0);
```

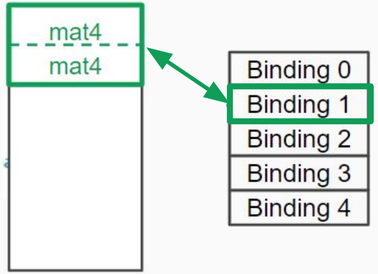


You can fill the buffer once per Update(), for example

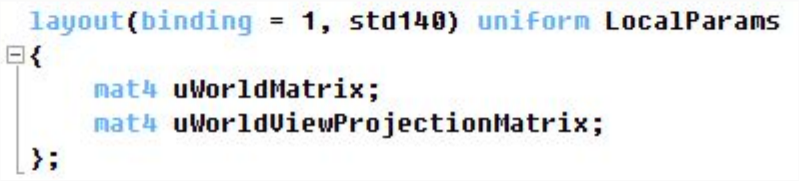
Binding buffer ranges to uniform blocks



```
GPU Name: GeForce GTX 1060 3GB/PCIe/SSE2
OGL Version: 4.4.0 NVIDIA 461.40
FPS: 60.041668
Camera
328.190   -17.796   0.000   Yaw/Pitch/Rol
2.610     2.751     6.323   Position
```

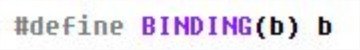


Uniform block in the vertex shader

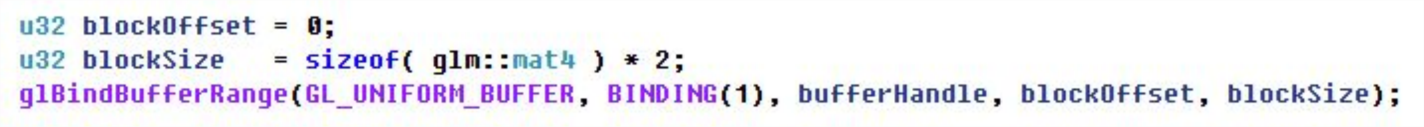


```
layout(binding = 1, std140) uniform LocalParams
{
    mat4 uWorldMatrix;
    mat4 uWorldViewProjectionMatrix;
};
```

We know where the information is in the buffer, so we bind the known buffer range (offset and size) to the corresponding binding point.

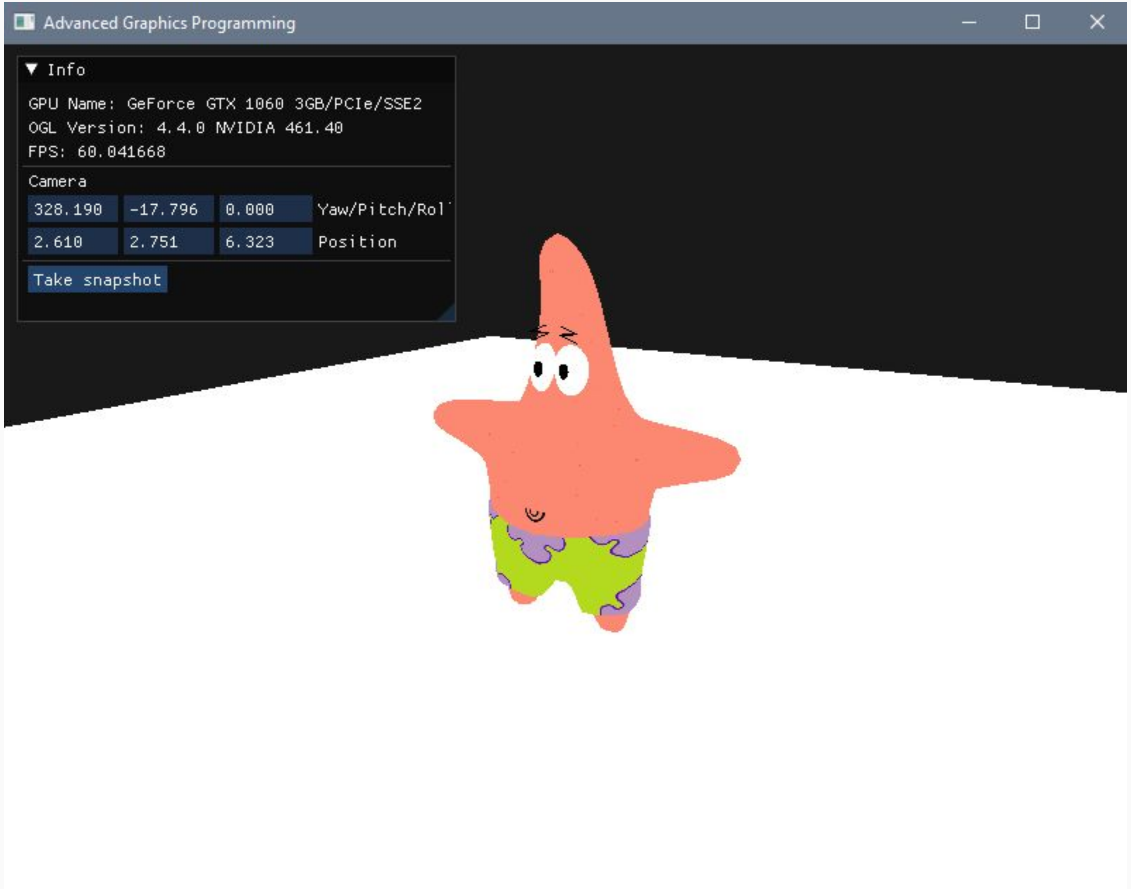


```
#define BINDING(b) b
```



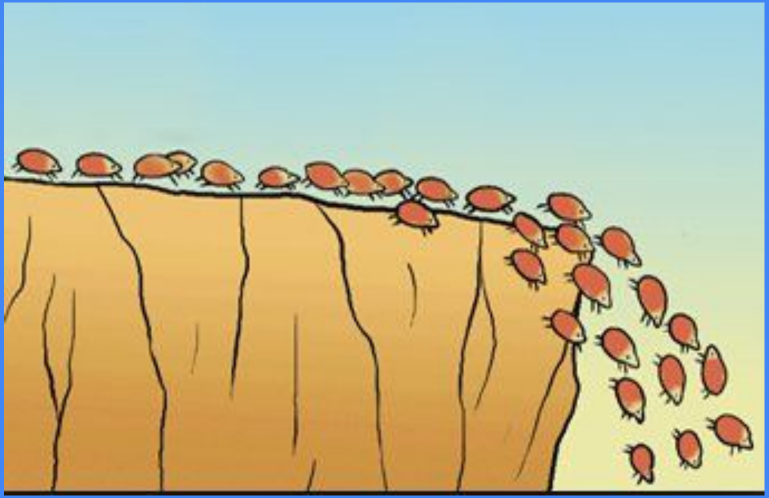
```
u32 blockOffset = 0;
u32 blockSize   = sizeof( glm::mat4 ) * 2;
glBindBufferRange(GL_UNIFORM_BUFFER, BINDING(1), bufferHandle, blockOffset, blockSize);
```

This example is easy because we only have data for one single block in the buffer (so offset is 0), but when we will have data for several blocks… we will need to do some bookkeeping.



```
GPU Name: GeForce GTX 1060 3GB/PCIe/SSE2
OGL Version: 4.4.0 NVIDIA 461.40
FPS: 60.041668
Camera
328.190   -17.796   0.000   Yaw/Pitch/Rol
2.610     2.751     6.323   Position
```

What about having several entities?

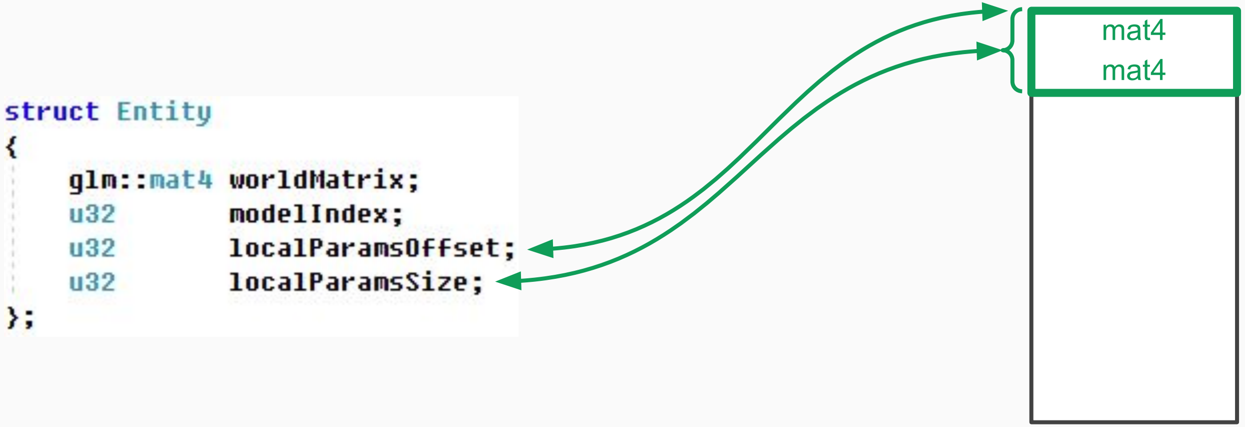


List of entities

At this point you should have a list of entities (or similar).

You will have to copy each entity's matrices into the uniform buffer.

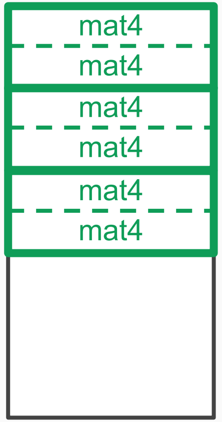
For each entity, you will need to remember where its information is in the buffer.



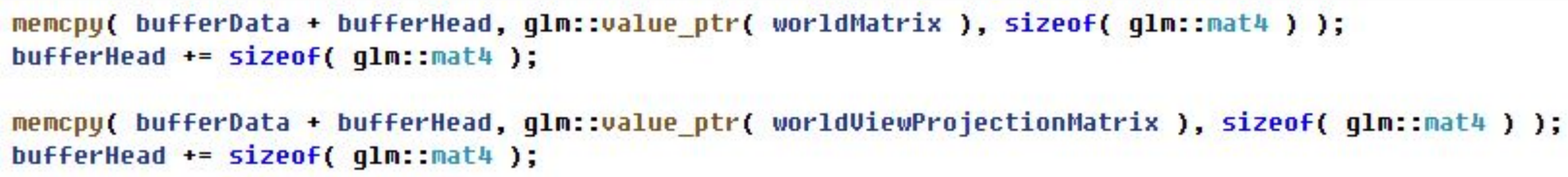
```
struct Entity
{
    glm::mat4 worldMatrix;
    u32       modelIndex;
    u32       localParamsOffset;
    u32       localParamsSize;
};
```

List of entities

You will have to copy all matrices in the buffer



```
For each entity
{

   entity.localParamsOffset = bufferHead;
```

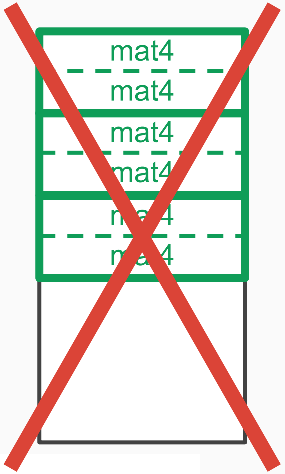


```
memcpy( bufferData + bufferHead, glm::value_ptr( worldMatrix ), sizeof( glm::mat4 ) );
bufferHead += sizeof( glm::mat4 );

memcpy( bufferData + bufferHead, glm::value_ptr( worldViewProjectionMatrix ), sizeof( glm::mat4 ) );
bufferHead += sizeof( glm::mat4 );
```

```
   entity.localParamsSize = bufferHead - entity.localParamsOffset;

}
```

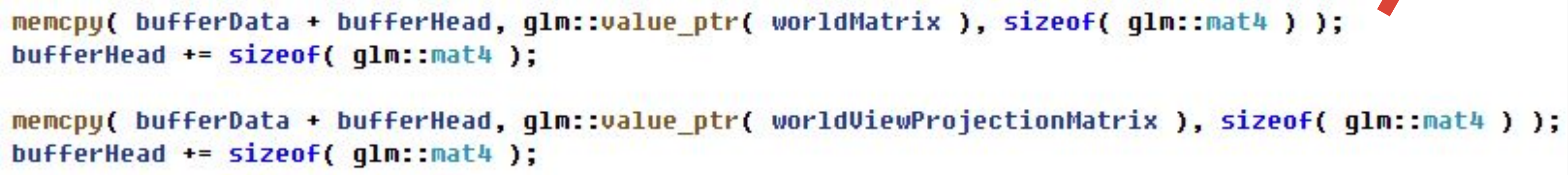
List of entities

You will have to copy all matrices in the buffer



Each shader block needs to be aligned!!!

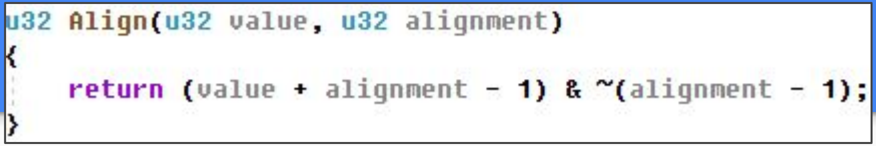
```
For each entity
{

   entity.localParamsOffset = bufferHead;
```



```
   memcpy( bufferData + bufferHead, glm::value_ptr( worldMatrix ), sizeof( glm::mat4 ) );
   bufferHead += sizeof( glm::mat4 );

   memcpy( bufferData + bufferHead, glm::value_ptr( worldViewProjectionMatrix ), sizeof( glm::mat4 ) );
   bufferHead += sizeof( glm::mat4 );
```

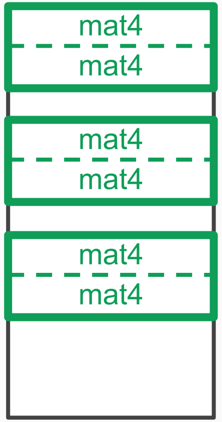
```
   entity.localParamsSize = bufferHead - entity.localParamsOffset;

}
```

List of entities

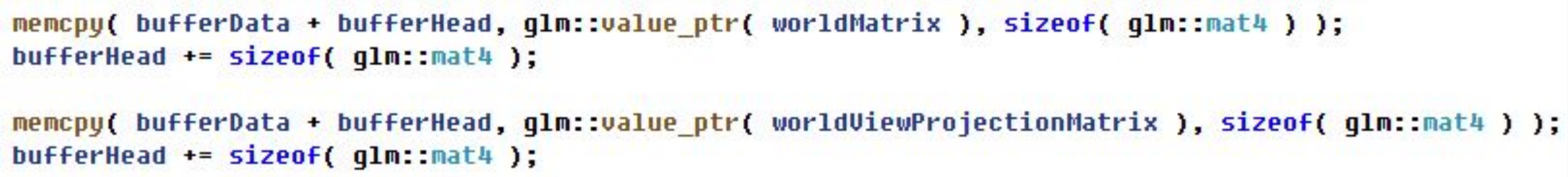


```
u32 Align(u32 value, u32 alignment)
{
    return (value + alignment - 1) & ~(alignment - 1);
}
```

But don’t forget to align each shader block



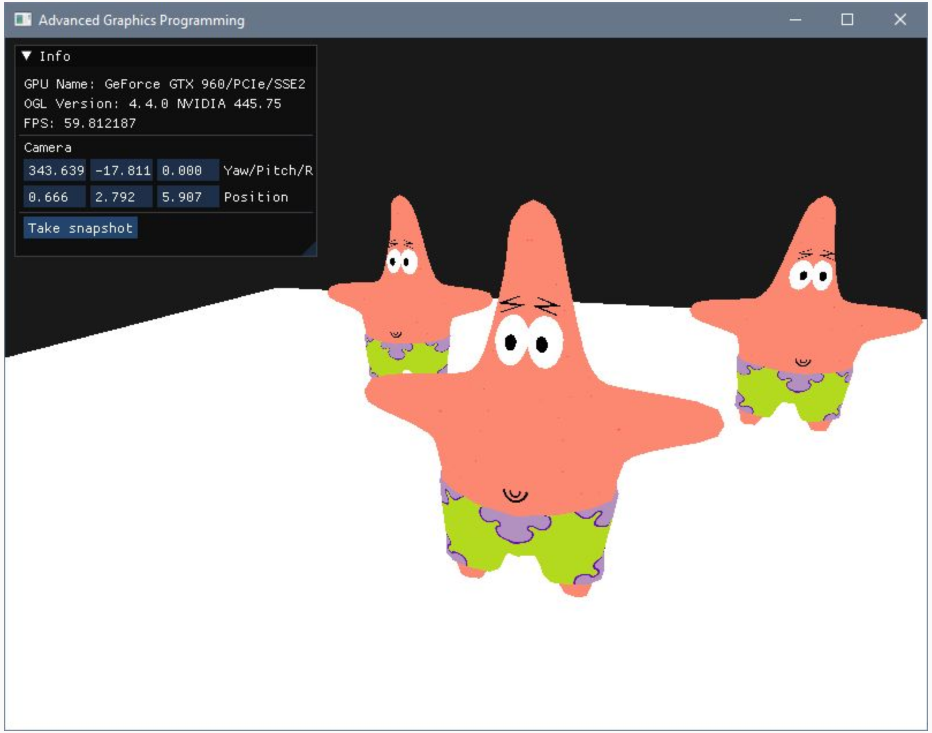
```
For each entity
{
   bufferHead = Align(bufferHead, uniformBlockAlignment);

   entity.localParamsOffset = bufferHead;
```



```
memcpy( bufferData + bufferHead, glm::value_ptr( worldMatrix ), sizeof( glm::mat4 ) );
bufferHead += sizeof( glm::mat4 );

memcpy( bufferData + bufferHead, glm::value_ptr( worldViewProjectionMatrix ), sizeof( glm::mat4 ) );
bufferHead += sizeof( glm::mat4 );
```

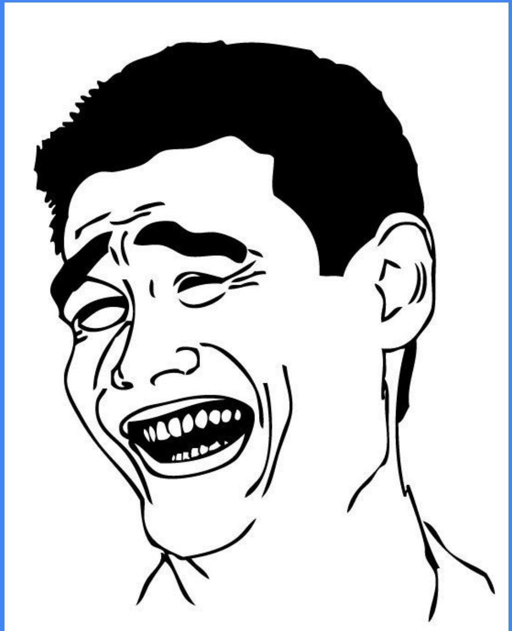
```
   entity.localParamsSize = bufferHead - entity.localParamsOffset;

}
```

List of entities



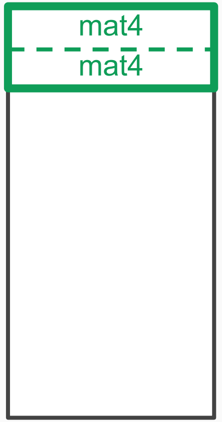
```
GPU Name: GeForce GTX 960/PCIe/SSE2
OGL Version: 4.4.0 NVIDIA 445.75
FPS: 59.812187
Camera
343.639  -17.811  0.000  Yaw/Pitch/R
0.666    2.792    5.907  Position
```

You think that alignment stuff was difficult?



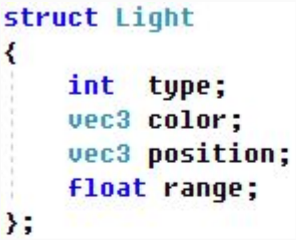
Shader block member alignment

Speaking about alignment, if we start filling a buffer from offset 0 with values of type **mat4**… we are lucky, it is a type that works ‘out of the box’ (vec4 too)

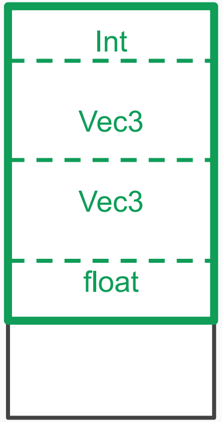


Shader block member alignment

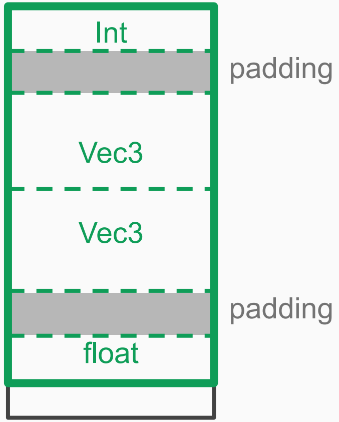
But we cannot generalize. In shader blocks, memory alignment does not work as in a C++ program.



```
struct Light
{
    int  type;
    vec3 color;
    vec3 position;
    float range;
};
```

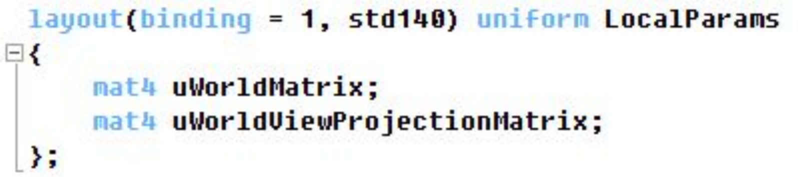


Alignment in main memory



Alignment in video memory

Uniform block layouts



```
layout(binding = 1, std140) uniform LocalParams
{
    mat4 uWorldMatrix;
    mat4 uWorldViewProjectionMatrix;
};
```

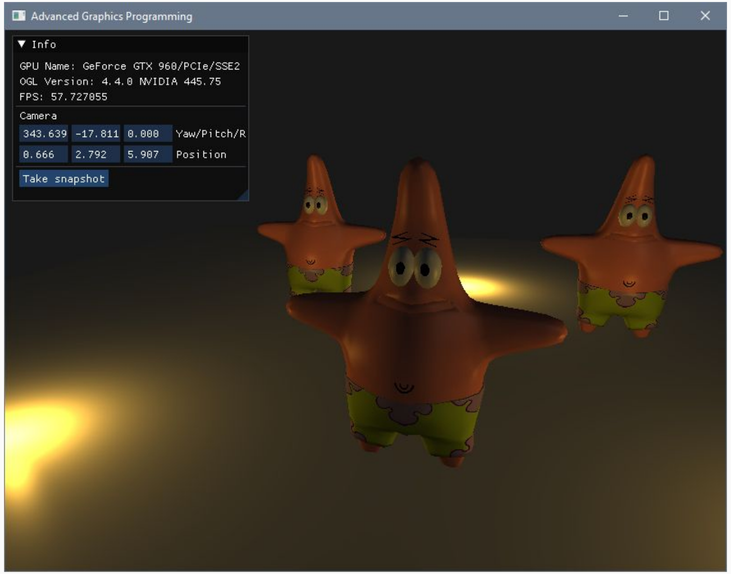
Packed. Platform dependent. Offsets need to be queried. Equal block descriptions can have different offsets on different shaders (cannot be shared). Most performance/memory efficient.

Shared. Platform dependent. Offsets need to be queried. Equal block descriptions will have equal offsets on different shaders (can be shared). Also very performance/memory efficient.

Std140. Platform independent. Layout rules are explicit, so we know the offsets following the layout rules. Quite performance efficient. Not memory efficient.

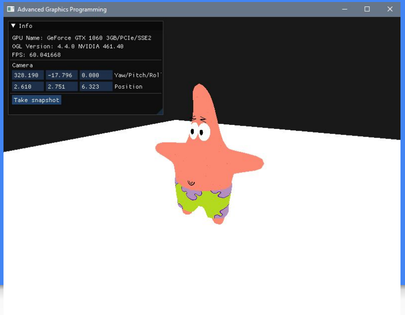
Std430. Platform independent. Layout rules are explicit, so we know the offsets following the layout rules. Less performance efficient than std140. More memory efficient in arrays.

Next day more on uniform buffers and lights

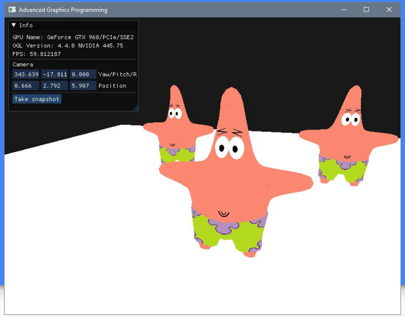


```
▼ Info
GPU Name: GeForce GTX 960/PCIe/SSE2
OGL Version: 4.4.0 NVIDIA 445.75
FPS: 57.727055
Camera
343.639  -17.811  0.000  Yaw/Pitch/R
0.666    2.792    5.907  Position
Take snapshot
```

TODO list



```
▼ Info
GPU Name: GeForce GTX 1060 3GB/PCIe/SSE2
OGL Version: 4.4.0 NVIDIA 461.40
FPS: 60.041668
Camera
328.190  -17.796  0.000  Yaw/Pitch/Rol
2.610    2.751    6.323  Position
```



```
▼ Info
GPU Name: GeForce GTX 960/PCIe/SSE2
OGL Version: 4.4.0 NVIDIA 445.75
FPS: 59.812187
Camera
343.639  -17.811  0.000  Yaw/Pitch/R
0.666    2.792    5.907  Position
```

1) Add the transforms for a single entity

- Adapt the vertex shader to use a uniform shader block that contains transform matrices
- In Init(): **Create** a uniform buffer where we will store the shader block values
- In Update(): **Push** the world transform and the world-view-projection of our entity into the buffer
- In Render(): **Bind** the buffer range that contains the two matrices to the shader block

2) Extend the work for multiple entities

- Create a Entity struct and a list of entities in your application (insert a few entities at Init)
- Adapt Update() to push the transforms for several entities (take into account **block alignment**)
- Adapt Render() to iterate over all entities and bind the appropriate buffer range each time

QUESTION
DOES ANYONE HAVE QUESTIONS?
makeameme.org