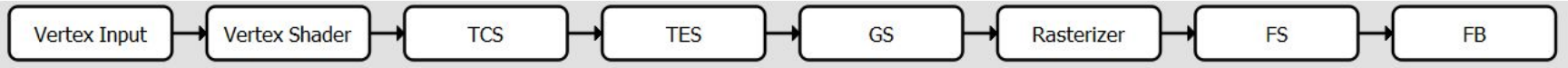
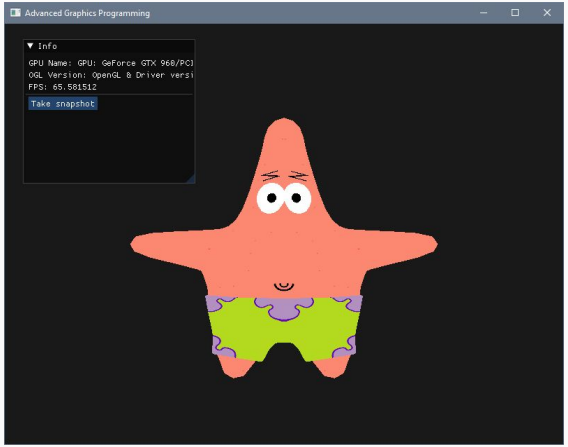
OpenGL

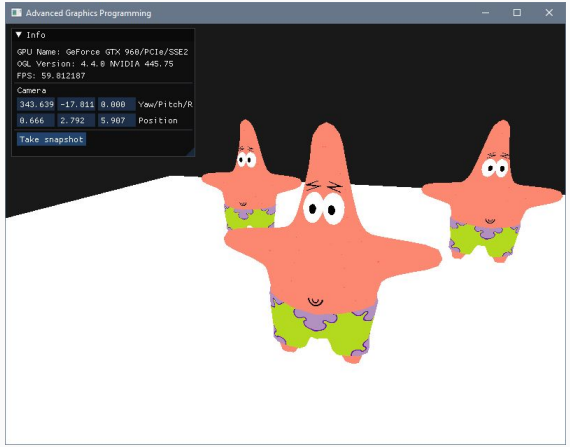
Uniform blocks and uniform buffers

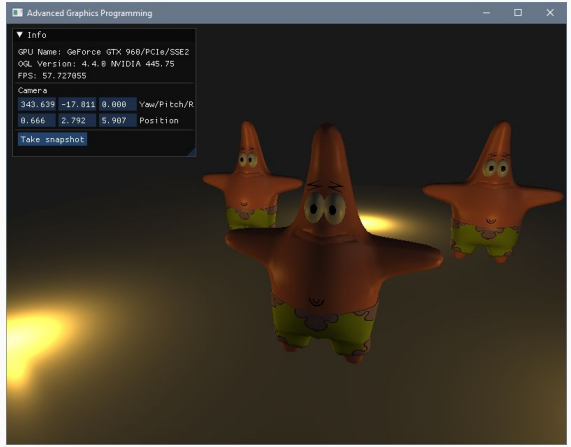
Advanced Graphics Programming

We added transforms, now lights

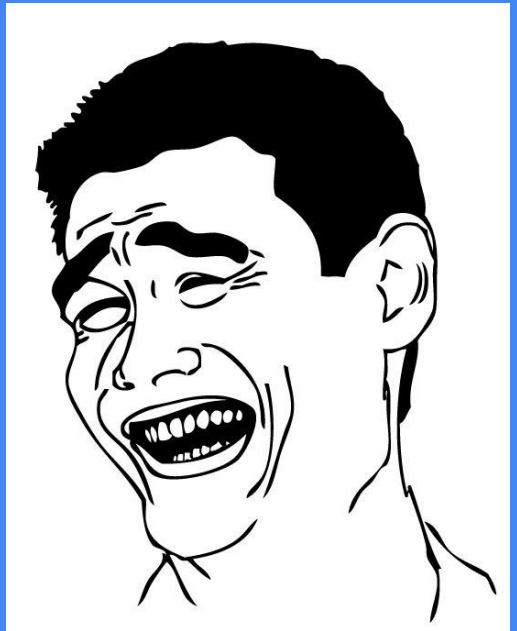






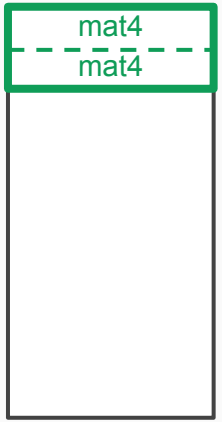


You think that alignment stuff was difficult?



Shader block member alignment

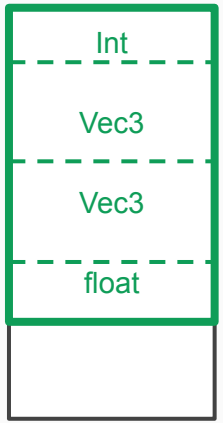
Speaking about alignment, if we start filling a buffer from offset 0 with values of type **mat4**… we are lucky, it is a type that works ‘out of the box’ (vec4 too)



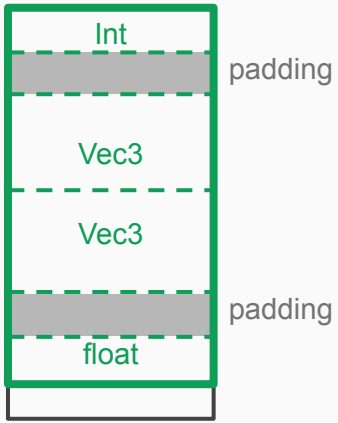
Shader block member alignment

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.

std140 uniform block layout (alignment rules)

Scalar `bool`, `int`, `uint`, `float` and `double`	Both the size and alignment are the size of the scalar in basic machine types (e.g., sizeof(GLfloat)).
Two-component vectors (e.g., `ivec2`)	Both the size and alignment are twice the size of the underlying scalar type.
Three-component vectors (e.g., `vec3`) and Four-component vectors (e.g., `vec4`)	Both the size and alignment are four times the size of the underlying scalar type.
An array of scalars or vectors	The size of each element in the array will be the size of the element type, rounded up to a multiple of the size of a `vec4`. This is also the array's alignment. The array's size will be this rounded-up element's size times the number of elements in the array.

std140 uniform block layout (alignment rules)

A column-major matrix or an array of column-major matrices of size C columns and R rows	Same layout as an array of N vectors each with R components, where N is the total number of columns present.
A row-major matrix or an array of row-major matrices with R rows and C columns	Same layout as an array of N vectors each with C components, where N is the total number of rows present.
A single-structure definition, or an array of structures	Structure alignment will be the alignment for the biggest structure member, according to the previous rules, rounded up to a multiple of the size of a `vec4`. Each structure will start on this alignment, and its size will be the space needed by its members, according to the previous rules, rounded up to a multiple of the structure alignment.

A couple of examples

How would the alignment of these structs be?

```
struct Light
{
    unsigned int type;
    vec3         color;
    vec3         direction;
    vec3         position;
};
```

```
struct Light
{
    vec3         color;
    vec3         direction;
    vec3         position;
    unsigned int type;
};
```

```
layout(binding = 0, std140) uniform GlobalParams
{
    vec3         uCameraPosition;
    unsigned int uLightCount;
    Light        uLight[16];
};
```

Adding lights

Forward shading vertex shader

```
struct Light
{
    unsigned int type;
    vec3         color;
    vec3         direction;
    vec3         position;
};
```

```
#if defined(VERTEX) ////////////////////////////////////////////////

layout(location = 0) in vec3 aPosition;
layout(location = 1) in vec3 aNormal;
layout(location = 2) in vec2 aTexCoord;

layout(binding = 0, std140) uniform GlobalParams
{
    vec3         uCameraPosition;
    unsigned int uLightCount;
    Light        uLight[16];
};

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

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

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

Convenience functions to work with buffers

```
struct Buffer
{
    GLuint handle;
    GLenum type;
    u32    size;
    u32    head;
    void*  data; // mapped data
};
```

You have these functions available in the Atenea campus

```
Buffer CreateBuffer(u32 size, GLenum type, GLenum usage) { ... }

#define CreateConstantBuffer(size) CreateBuffer(size, GL_UNIFORM_BUFFER, GL_STREAM_DRAW)
#define CreateStaticVertexBuffer(size) CreateBuffer(size, GL_ARRAY_BUFFER, GL_STATIC_DRAW)
#define CreateStaticIndexBuffer(size) CreateBuffer(size, GL_ELEMENT_ARRAY_BUFFER, GL_STATIC_DRAW)

void BindBuffer(const Buffer& buffer) { ... }

void MapBuffer(Buffer& buffer, GLenum access) { ... }

void UnmapBuffer(Buffer& buffer) { ... }

void AlignHead(Buffer& buffer, u32 alignment) { ... }

void PushAlignedData(Buffer& buffer, const void* data, u32 size, u32 alignment) { ... }

#define PushData(buffer, data, size) PushAlignedData(buffer, data, size, 1)
#define PushUInt(buffer, value) { u32 v = value; PushAlignedData(buffer, &v, sizeof(v), 4); }
#define PushVec3(buffer, value) PushAlignedData(buffer, value_ptr(value), sizeof(value), sizeof(vec4))
#define PushVec4(buffer, value) PushAlignedData(buffer, value_ptr(value), sizeof(value), sizeof(vec4))
#define PushMat3(buffer, value) PushAlignedData(buffer, value_ptr(value), sizeof(value), sizeof(vec4))
#define PushMat4(buffer, value) PushAlignedData(buffer, value_ptr(value), sizeof(value), sizeof(vec4))
```

Light struct to store a list of lights in our application

You will have to define these types in Engine.h

Then in your Application struct, create an array/list of lights that you can fill at Init()

```
enum LightType
{
    LightType_Directional,
    LightType_Point
};

struct  Light
{
    LightType type;
    vec3      color;
    vec3      direction;
    vec3      position;
};
```

Pushing values for the GlobalParams block into the uniform buffer

Evolved version of uniform buffer setup. Right after mapping the buffer we can start pushing the global parameters.

```
// -- Global params
app->globalParamsOffset = app->cbuffer.head;

PushVec3(app->cbuffer, camera.position);

PushUInt(app->cbuffer, app->lights.size());

for (u32 i = 0; i < app->lights.size(); ++i)
{
    AlignHead(app->cbuffer, sizeof(vec4));

    Light& light = app->lights[i];
    PushUInt(app->cbuffer, light.type);
    PushVec3(app->cbuffer, light.color);
    PushVec3(app->cbuffer, light.direction);
    PushVec3(app->cbuffer, light.position);
}

app->globalParamsSize = app->cbuffer.head - app->globalParamsOffset;
```

Pushing values for the LocalParams block into the uniform buffer

And here we traverse all entities to push all local parameters.

Do not forget to unmap the buffer once you've finished pushing values.

```
// -- Local params
for (u32 i = 0; i < app->entities.size(); ++i)
{
    AlignHead(app->cbuffer, app->uniformBufferAlignment);

    Entity& entity = app->entities[i];
    mat4    world  = entity.worldMatrix;
    mat4    worldViewProjection = projection * view * world;

    entity.localParamsOffset = app->cbuffer.head;
    PushMat4(app->cbuffer, world);
    PushMat4(app->cbuffer, worldViewProjection);
    entity.localParamsSize = app->cbuffer.head - entity.localParamsOffset;
}
```

Bind both buffer ranges to the shader blocks

IMPORTANT:

In the previous slides our shader only had one uniform block (LocalParams).

No we have added an extra uniform block into the shader (GlobalParams) and inserted information for it into our uniform buffer.

Do not forget to bind the buffer range with the global parameters (camera position, lights…) to the GlobalParams block in the shader!!!

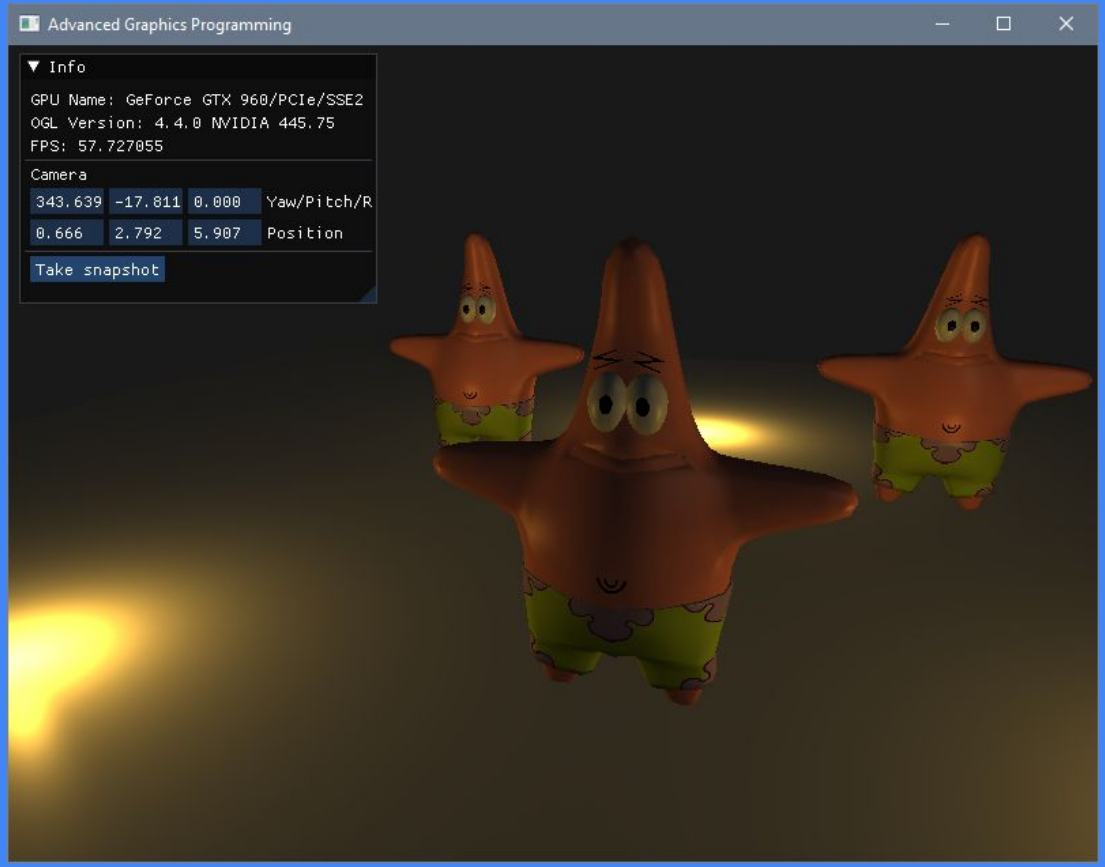
You only need to bind it once per frame, as all entities share the same GlobalParams.

Forward shading fragment shader

```
struct Light
{
    unsigned int type;
    vec3         color;
    vec3         direction;
    vec3         position;
};
```

```
in vec2 vTexCoord;
in vec3 vPosition; // In worldspace
in vec3 vNormal;   // In worldspace
in vec3 vViewDir;  // In worldspace

uniform sampler2D uTexture;

layout(binding = 0, std140) uniform GlobalParams
{
    vec3         uCameraPosition;
    unsigned int uLightCount;
    Light        uLight[16];
};

layout(location = 0) out vec4 oColor;

void main()
{
    // TODO: Sum all light contributions up to set oColor final value
}
```

Questions?


Advanced Graphics Programming
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