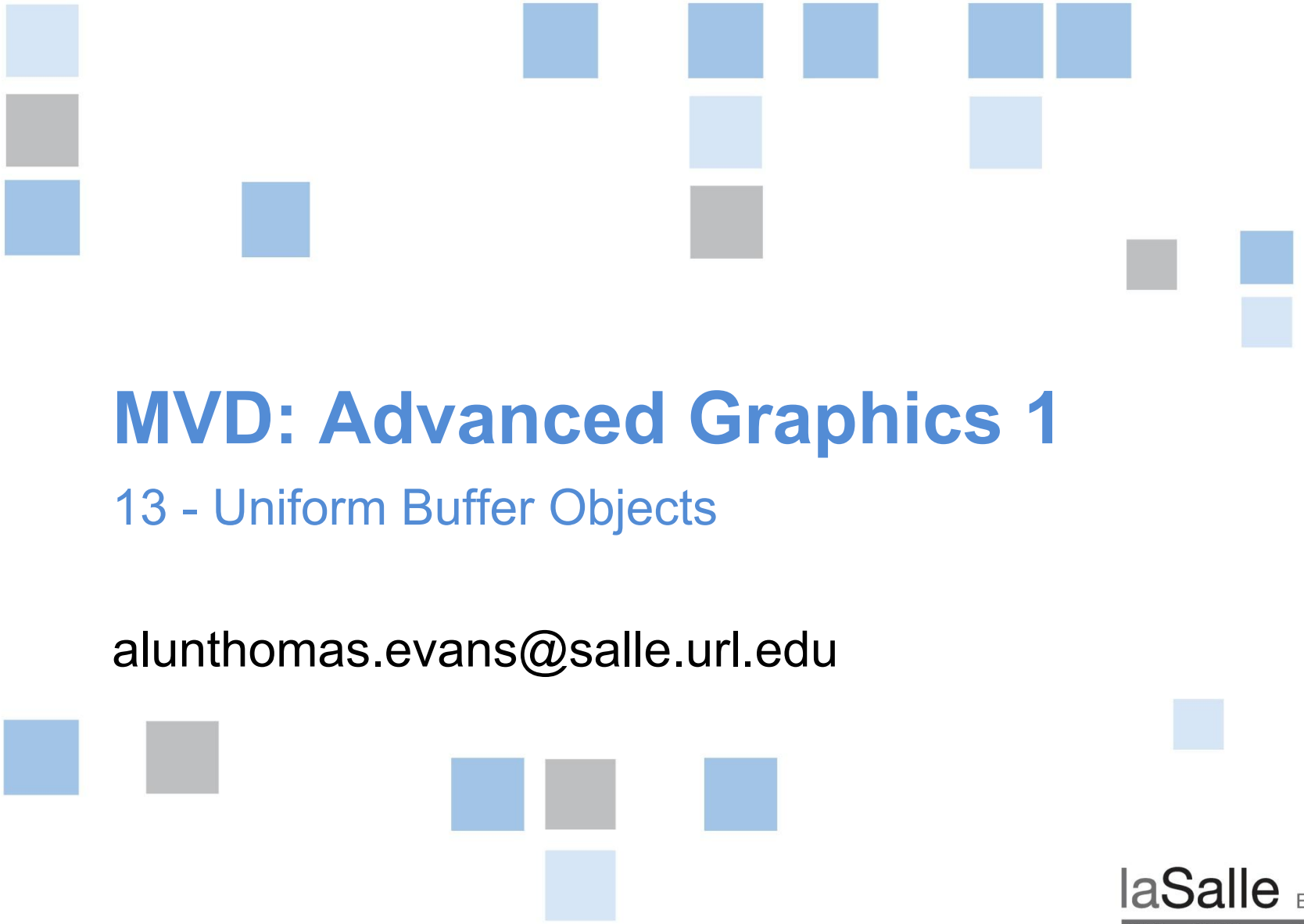


# MVD: Advanced Graphics 1

## 13 - Uniform Buffer Objects

alunthomas.evans@salle.url.edu

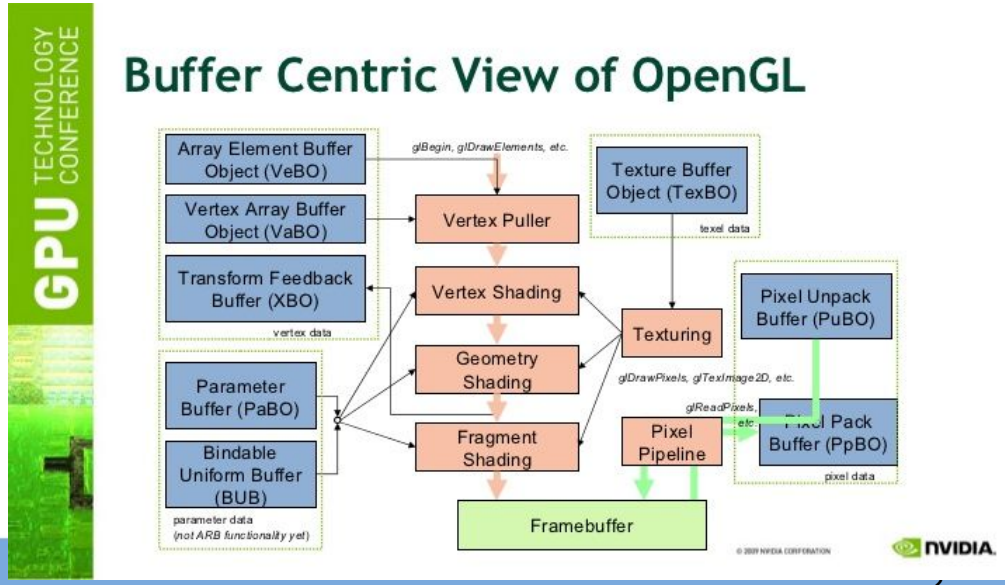
laSalle ENG
Universitat Ramon Llull

# Buffer buffer buffers

You may have noticed that graphics programming is all about buffers. Why?

Cos buffers are bit of memory. We get a pointer (or a handle) to them, and then we can do stuff with them

*What buffers have be used so far?*



# Uniform Buffers

Uniform buffers are self descriptive - memory which represents a uniform.

**IMPORTANT** - they are independent of shaders!

*In which scenarios would this be useful?*

# Uniform Buffers

*In which scenarios would this be useful?*

Any scenario where the same data can be sent to various shaders.

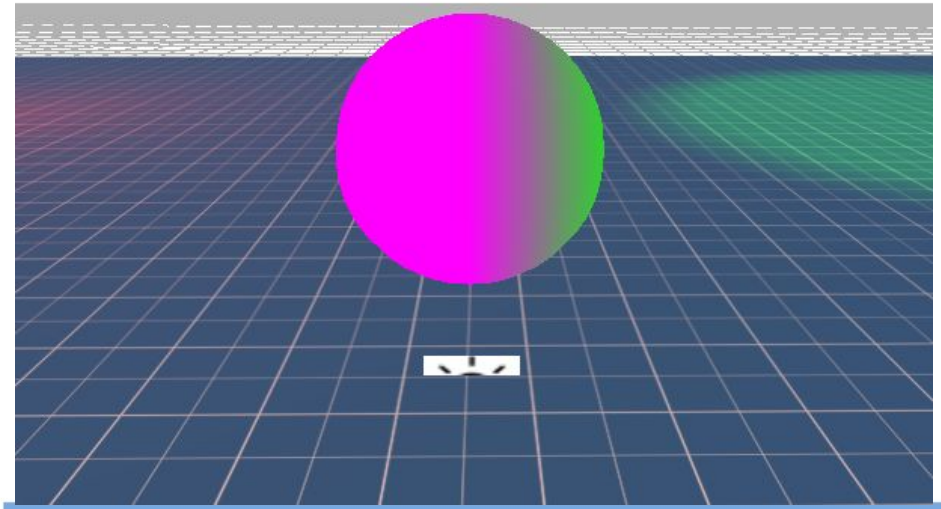
e.g. lights!

*Can you think of other potential uses?*

# Download code

Look at ubo_test fragment shader

then at temporary code in setMaterialUniforms



# Uniform block layout

Uniform buffers require a strict layout structure

scalar types have base alignment of 4 bytes

vectors, and matrices have base alignment of 16 bytes

# Uniform block layout in shader

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

# Rules

NEVER use vec3s - always use vec4

Try to group scalar types together (float + float+float+float)

If you have one scalar, followed by an vec or mat, that single scalar is ‘worth’ 16 bytes (instead of 4)

# Using std140 layout in shader

Declare the struct as before (but with vec4s)

Declares a uniform with std140 layout

dont forget to modifty the mix function to use xyz

```
struct TestStruct {
    vec4 color_a;
    vec4 color_b;
};

layout (std140) uniform UBO_test
{
    TestStruct u_test_struct;
};
```

```
vec3 final_color = mix(u_test_struct.color_a.xyz, u_test_struct.color_b.xyz, col_f);
```

# Creating UBO in OpenGL: step 1

Declare size

```
GLsizeiptr size_test_struct = 16 + 16; //vec4 + vec4
```

this is going to be the size, in bytes, of the std140 uniform (UBO_test) in our example

```
struct TestStruct {
    vec4 color_a;
    vec4 color_b;
};

layout (std140) uniform UBO_test
{
    TestStruct u_test_struct;
};
```

# Creating UBO in OpenGL: step 2

Create and bind buffers

```
GLuint uboTest;
glGenBuffers(1, &uboTest);
glBindBuffer(GL_UNIFORM_BUFFER, uboTest);
glBufferData(GL_UNIFORM_BUFFER, size_test_ubo, NULL, GL_STATIC_DRAW);
```

Similar to creating and binding vertex buffers!

# Creating UBO in OpenGL: step 3

Write data to buffer

```
GLintptr offset = 0; //pointer to top of buffer

GLfloat color_a_data[4] = { 1.0, 0.0, 1.0, 0.0 };
glBufferSubData(GL_UNIFORM_BUFFER, //constant id
                offset, //where to start writing memory
                16, //how much memory to write
                color_a_data); //pointer to start of memory

offset += 16; //move offset ready to write next variable
```

# Bind UBO to shader uniform with a binding point

```
GLuint uboTest;
glGenBuffers(1, &uboTest);
glBindBuffer(GL_UNIFORM_BUFFER, uboTest);
```

**OpenGL UBO**

"ubo_test"

**Binding Points**

0
1
2
3
4
…
N

```
struct TestStruct {
    vec4 color_a;
    vec4 color_b;
};

layout (std140) uniform UBO_test
{
    TestStruct u_test_struct;
};
```

**Shader std140**

uniform "UBO_test

```
GLuint TEST_BINDING_POINT = 0;
glBindBufferRange(GL_UNIFORM_BUFFER, TEST_BINDING_POINT, uboTest, 0, size_test_ubo);

GLuint UBO_test = glGetUniformBlockIndex(shader_->program, "UBO_test");
if (UBO_test != -1) glUniformBlockBinding(shader_->program, UBO_test, TEST_BINDING_POINT);
```

# Task: convert test to UBO

Change Test struct to have vec4s and declare std140 layout

Create uniform buffer object and fill it with some data

Link ubo to binding point

link shader to binding point

# Dealing with arrays

Similar to declaring regular uniform arrays

```
const int MAX_LIGHTS = 8;
uniform int u_num_lights;

layout (std140) uniform Lights
{
    Light lights[MAX_LIGHTS];
};
```

But buffer MUST be multiple of 16

e.g. if you finish a struct with a ‘hanging’ int or float, you must ‘pad’ the buffer out by 16 bytes (not 4) in order to read next one okay

# Setting data

Recommend grouping data of similar types and setting it from a single array:

```
GLfloat light_data[16] = {
    lt.m[12], lt.m[13], lt.m[14], 0.0,
    l.direction.x, l.direction.y, l.direction.z, 0.0,
    l.color.x, l.color.y, l.color.z, 0.0,
    l.linear_att,l.quadratic_att,spot_inner_cosine,spot_outer_cosine
};
//the data
glBufferSubData(GL_UNIFORM_BUFFER, offset, 64, light_data); //colour
offset += 64;
//type
glBufferSubData(GL_UNIFORM_BUFFER, offset, 4, &(l.type)); // type
offset += 16;
```

# Task: set UBO for all lights

COMMENT previous UBO

COMMENT light uniforms setting (except u_num_lights)

change shader to use vec4 (add .xyz where required)

result should look exactly the same

# Advanced task

Abstract light setting into a separate function updateLights_()

this function is called during renderloop ONLY if a flag is set

function updates ALL shaders in graphics system