Introduction to Graphics Programming and its Applications

繪圖程式設計與應用

OpenGL Objects

Instructor: Hung-Kuo Chu

Department of Computer Science

National Tsing Hua University

CS4595

Codeblock Conventions

Yellow Codeblock => Application Program (CPU)

- Create OpenGL Context
- Create and Maintain OpenGL Objects
- Generate Works for the GPU to Consume

Blue Codeblock => Shader Program (GPU)

- Shader for a Certain Programmable Stage
- Process Geometry or Fragment in Parallel
- Starts with #version 410 core Declaration

Recall

- OpenGL works as a ***state machine***
- There are roughly two types of APIs:

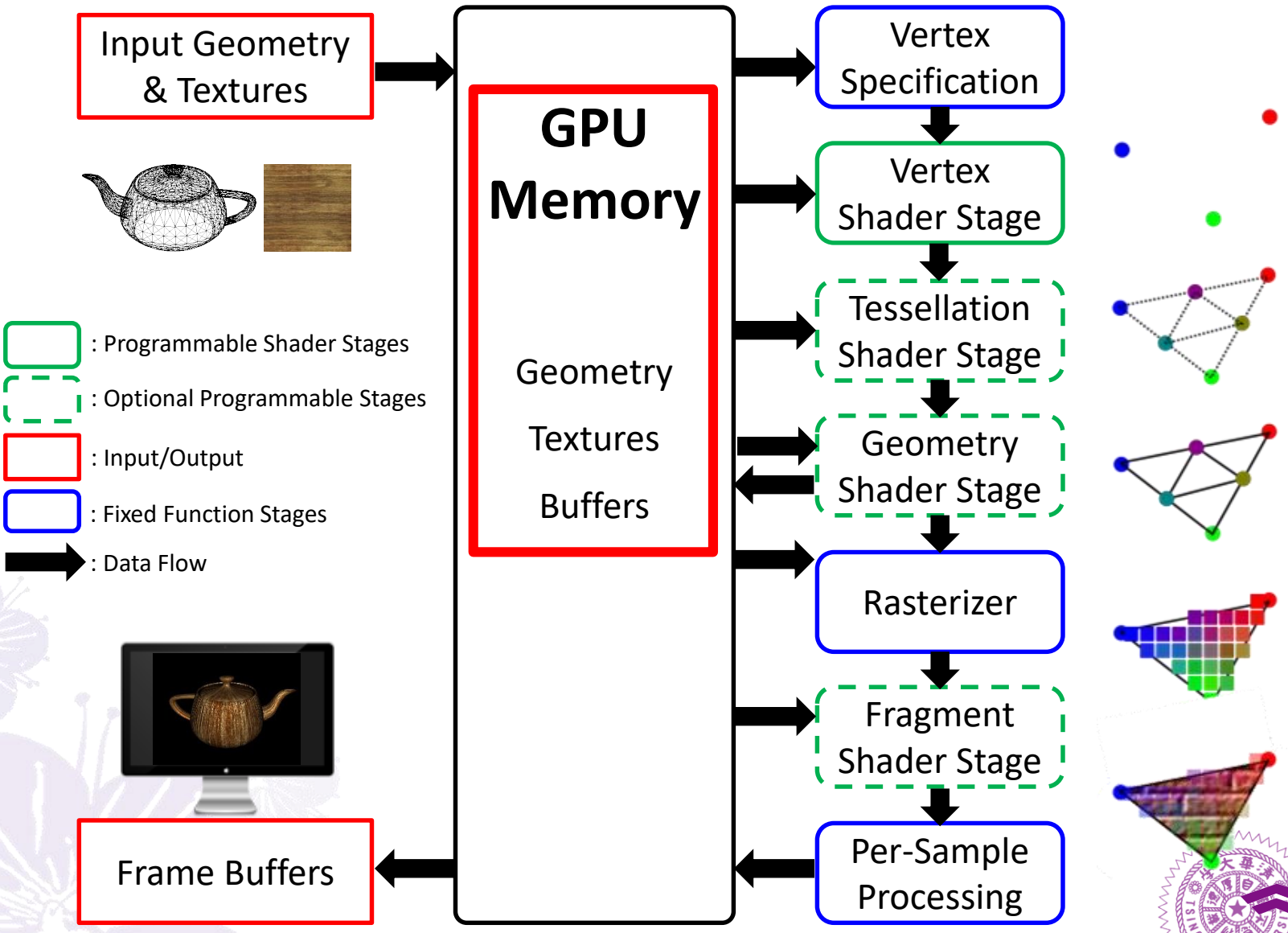
APIs to Specify Objects

- Vertex or Index Buffers
- 1D, 2D, 3D Textures
- Shader Programs
- Framebuffer Objects

APIs to Configure the Pipeline

- State of Fixed Function Stages
- Specify the Shader Program begin Used
- Specify the Render Target being Used

The OpenGL Rendering Pipeline



OpenGL Resource Objects

- **Buffers** and **Textures**
- Blocks of memory that can be **bound to the pipeline**, and **used for input or output**
- Blocks of memory that is **available for the GPU to use and manipulate**
- Stores geometry data, texture maps, look up tables or intermediate rendering results…

OpenGL Buffers

Buffers

- Buffers are **linear allocations** of memory
  - Resides in the main memory or the video memory
- Buffer categories based on usages:
  - Vertex Buffers
  - Index Buffers
  - Uniform Buffers
  - Shader Storage Buffers
  - …

Vertex and Index Buffers

- Provide **per-vertex input** to the pipeline
- Contains **array-like data** for each input vertex attributes
  - Vertex position (X, Y, Z)
  - Vertex texture coordinate (U, V)
  - …
- Consumed by the Vertex Specification stage to generate a vertex stream as the input to the rendering pipeline (or more precisely, the Vertex Shader stage)

Uniform Buffers

- Data in uniform buffers **remains constant throughout the execution of a draw call**
  - Optimized for GPU **read-only** accesses
  - Every invocation of each programmable stages **accesses the same content**
- More like a **structure** instead of an array
  - Provide **global configurations** for the draw call
  - Camera settings, light positions, …

Shader Storage Buffers

- An array of custom type data. Much like Uniform Buffers, but the programmable shaders have **read/write access** to them
- Intended to provide a flexible memory resource for simplifying **custom algorithm development**
- Available since OpenGL 4.3. For advanced usage. We won't cover it in this lecture

Buffer Object Creation

1. Call **`glGenBuffers`** to generate **names** (unsigned int ids) for buffers
   - The buffer object is NOT created yet!
2. Call **`glBindBuffers`** to bind a name to a Binding Point
   - Binding the buffer CREATES the buffer object
3. Call **`glBufferData`** to initialize (allocate) the buffer content storage
4. If updating the buffer content is required, call **`glBufferSubData`** or **`glMapBuffer`**

glGenBuffers

*void glGenBuffers(GLsizei n, GLuint * buffers);*

- **n**: Specifies the number of buffer object names to be generated.
- **buffers**: Specifies an array in which the generated buffer object names are stored.
- **Description**: *glGenBuffers* generate buffer object names and returns n buffer object names in buffers.

Binding Points

- Buffers in OpenGL are used based on the location to which they are bound
  - The creation and data uploading process are **the same** for all kinds of buffer types
- To bind a buffer to the pipeline, it must be bound to a **Binding Point (Binding Target)**
- **Binding Points defines the target for buffer object binding or manipulations <= Please Pay Attention to This!**

glBindBuffer

void glBindBuffer(GLenum target, GLuint buffer);

- **target**: Specifies the target to which the buffer object is bound.

Buffer Binding Target	Purpose
GL_ARRAY_BUFFER	Vertex buffer
GL_ELEMENT_ARRAY_BUFFER	Index buffer
GL_COPY_READ_BUFFER	Buffer copy source
GL_COPY_WRITE_BUFFER	Buffer copy destination
GL_TEXTURE_BUFFER	Texture data buffer
GL_UNIFORM_BUFFER	Uniform block storage
GL_SHADER_STORAGE_BUFFER	Shader storage buffer
GL_TRANSFORM_FEEDBACK_BUFFER	Transform feedback buffer

glBindBuffer (Cont'd)

void glBindBuffer(GLenum target, GLuint buffer);

- **buffer**: Specifies the name of a buffer object.
- **Description:** *glBindBuffer* binds a buffer object to the specified buffer binding point. Calling glBindBuffer with target set to one of the accepted symbolic constants and buffer set to the name of a buffer object binds that buffer object name to the target.

Buffer Usage

- Buffer Usage is a flag required by OpenGL when specifying a buffer data storage
- It gives hints for the runtime to decide **how and where** to allocate the buffer data storage
  - For example, a **GL_STATIC_DRAW** usage flag means the data can stay entirely in the GPU memory, there is no need to keep a copy in the main memory

Table of Buffer Usage

Buffer Usage
GL_STREAM_DRAW
GL_STREAM_READ
GL_STREAM_COPY
GL_STATIC_DRAW
GL_STATIC_READ
GL_STATIC_COPY
GL_DYNAMIC_DRAW
GL_DYNAMIC_READ
GL_DYNAMIC_COPY

Type	Description
STREAM	Be set once by the application and used infrequently
STATIC	Be set once by the application and used frequently
DYNAMIC	Be updated frequently and used frequently

Type	Description
READ	For queried.
COPY	For drawing or copying to other images.
DRAW	For drawing.

Example : ***GL_STATIC_DRAW*** : Buffer contents will be set once by the application and used frequently for drawing .

Buffer Data

- Buffer data storage can be manipulated by:
- **`glBufferData`**
  - Initialize and allocate a new storage with optional initial data
  - Old storage will be freed
- **`glBufferSubData`**
  - Update part of or all of the buffer content
  - Faster than **`glBufferData`**
- **`glMapBuffer`**
  - Update part of or all of the buffer content by **writing to a block in the main memory directly**
  - Might be faster than **`glBufferSubData`**

Buffer Data

- **glBufferData**, **glBufferSubData** and **glMapBuffer** operates on the currently bound buffer object of a **Binding Point**
- Don't forget to bind the buffer object to a Binding Point before manipulating it!

```
// bind Buffer Object "buffer" to the "GL_ARRAY_BUFFER" Binding Point
glBindBuffer(GL_ARRAY_BUFFER, buffer);

// manipulate the buffer bound to "GL_ARRAY_BUFFER", which is "buffer"
glBufferData(GL_ARRAY_BUFFER, sizeof(data), data, GL_STATIC_DRAW);
```

glBufferData

void glBufferData(GLenum target, GLsizeiptr size,
*const GLvoid * data, GLenum usage);*

- **target**: Specifies the target to which the buffer object is bound for glBufferData.
- **size**: Specifies the size of a buffer object.
- **data:** Specifies a pointer to data that will be copied into the data store for initialization, or NULL if no data is to be copied.

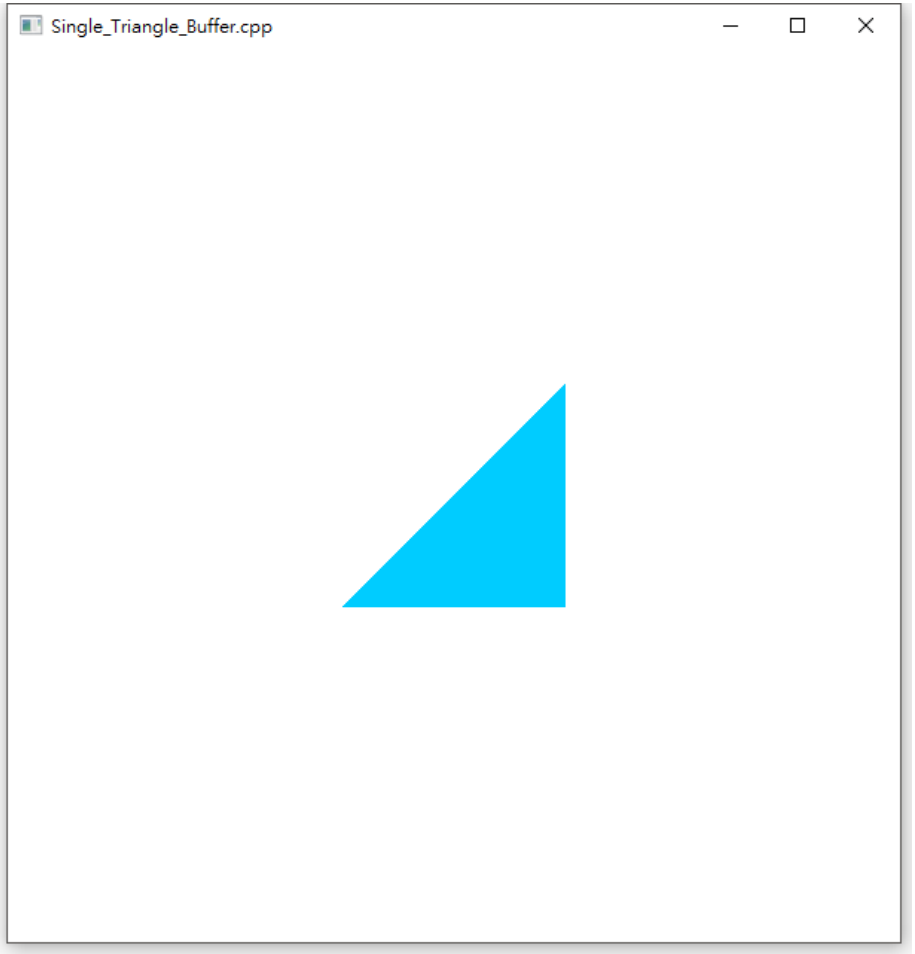
glBufferData (Cont'd)

void glBufferData(GLenum target, GLsizeiptr size,
*const GLvoid * data, GLenum usage);*

- **usage**: Specifies the expected usage pattern of the data store.
- **Description**: *glBufferData* create a new data store for a buffer object. The buffer object currently bound to target is used. While creating the new storage, any pre-existing data store is deleted. The new data store is created with the specified size in bytes and usage. If data is not NULL, the data store is initialized with data from this pointer.

Draw a Triangle with Buffer
Example: Single Triangle Buffer

Draw a Triangle with Buffer



Code: Vertex Shader

```
#version 410

layout(location = 0) in vec3 iv3vertex;
layout(location = 1) in vec3 iv3color;

out vec3 vv3color;

void main()
{
        gl_Position = vec4(iv3vertex, 1.0);
        vv3color = iv3color;
}
```

Code: Fragment Shader

```
#version 410

in vec3 vv3color;

layout(location = 0) out vec4 fragColor;

void main()
{
        fragColor = vec4(vv3color, 1.0);
}
```

Code: Buffer Data

```
float data[18] =

{
        -0.5f, -0.4f, 0.0f,   //Position
         0.5f, -0.4f, 0.0f,
         0.0f,  0.6f, 0.0f,

         1.0f,  0.0f, 0.0f,   //Color
         0.0f,  1.0f, 0.0f,
         0.0f,  0.0f, 1.0f
};
```

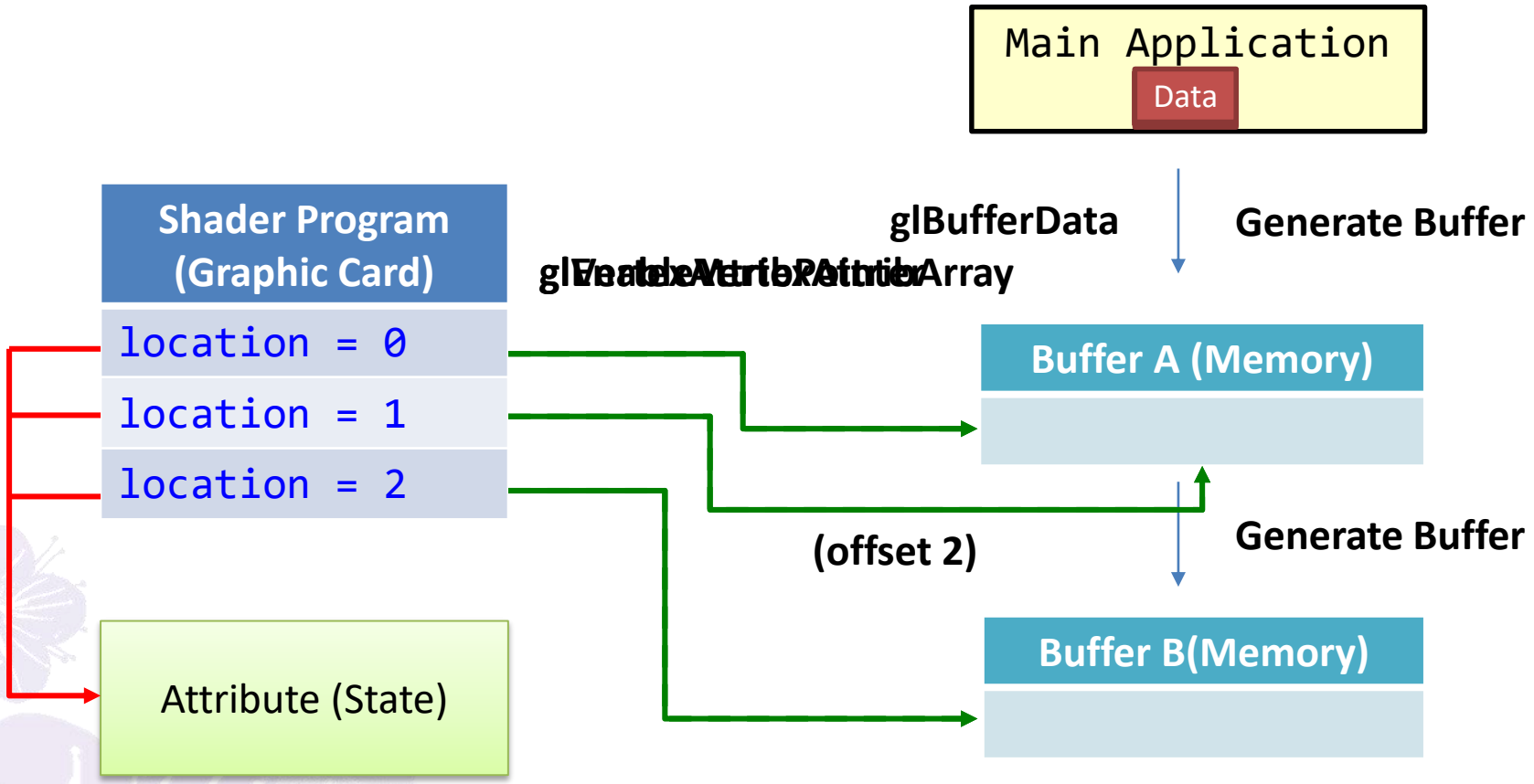
Code: Set Up Buffer

```
// The type used for names in OpenGL is GLuint
GLuint buffer;

// Generate a name for the buffer
glGenBuffers(1, &buffer);

// Now bind it to the context using the GL_ARRAY_BUFFER binding point
glBindBuffer(GL_ARRAY_BUFFER, buffer);

// Specify the amount of storage we want to use for the buffer
glBufferData(GL_ARRAY_BUFFER, sizeof(data), data, GL_STATIC_DRAW);
```

Code: Set Up VAO and Attribute

```
GLuint vao;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);

glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0,
                      (void*)(sizeof(float) * 9));//offset

glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);
```

Buffer & Data & Attribute



Code: Display Function

```
void My_Display()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glDrawArrays(GL_TRIANGLES, 0, 3);
  glutSwapBuffers();
}
```

Update Buffer Data

Code: glBufferSubData

```
// The type used for names in OpenGL is GLuint
GLuint buffer;

// Generate a name for the buffer
glGenBuffers(1, &buffer);

// Now bind it to the context using the GL_ARRAY_BUFFER binding point
glBindBuffer(GL_ARRAY_BUFFER, buffer);

// Specify the amount of storage we want to use for the buffer
glBufferData(GL_ARRAY_BUFFER, 1024 * 1024, NULL, GL_STATIC_DRAW);
```

Code: glBufferSubData (Cont'd)

```
static const float data[] =
{
        0.25, -0.25, 0.5, 1.0,
        -0.25, -0.25, 0.5, 1.0,
        0.25, 0.25, 0.5, 1.0
};

glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(data),
data);
```

glBufferSubData

void glBufferSubData(GLenum target, GLintptr offset,
*GLsizeiptr size, const GLvoid * data);*

- **target**: Specifies the target buffer object. The symbolic constant must be GL_ARRAY_BUFFER, GL_ELEMENT_ARRAY_BUFFER,GL_PIXEL_PACK_BUFFER, or GL_PIXEL_UNPACK_BUFFER.
- **offset:** Specifies the offset into the buffer object's data store where data replacement will begin, measured in bytes.

glBufferSubData (Cont'd)

void glBufferSubData(GLenum target, GLintptr offset,
*GLsizeiptr size, const GLvoid * data);*

- **size**: Specifies the size in bytes of the data store region being replaced.
- **data:** Specifies a pointer to the new data that will be copied into the data store.
- **Description:** *glBufferSubData* redefines some or all of the data store for the buffer object currently bound to target. Data starting at byte offset offset and extending for size bytes is copied to the data store from the memory pointed to by data.

Code: glMapBuffer

```
static const float data[] =
{
        0.25, -0.25, 0.5, 1.0,
        -0.25, -0.25, 0.5, 1.0,
        0.25, 0.25, 0.5, 1.0
};

// Get a pointer to the buffer's data store
void * ptr = glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);

// Copy our data into it...
memcpy(ptr, data, sizeof(data));

// Tell OpenGL that we're done with the pointer
glUnmapBuffer(GL_ARRAY_BUFFER);
```

glMapBuffer

1. Calling **`glMapBuffer`** on a buffer object
2. OpenGL will return a block of memory in the main memory for reading/writing
3. Calling **`glUnmapBuffer`** will flush all the changes to take effect in the buffer content

- This can be potentially faster than **`glBufferSubData`**. For example, when loading a geometry into a buffer, you don't need to manually allocate a block of memory and call **`glBufferSubData`**. **You can load the file content directly to the memory returned by `glMapBuffer`**

glMapBuffer

*void *glMapBuffer(GLenum target, GLenum access);*

- **target**: Specifies the target to which the buffer object is bound for glMapbuffer.
- **access**: Specifies the access policy for glMapBuffer .The symbolic constant must be GL_READ_ONLY, GL_WRITE_ONLY, or GL_READ_WRITE.

glMapBuffer (Cont'd)

*void *glMapBuffer(GLenum target, GLenum access);*

- **Description:** *glMapBuffer* map the entire data store of a specified buffer object into the client's **address space.** The data can then be directly read and/or written relative to the returned pointer, depending on the specified access policy.

Uniforms

Uniforms

- Uniforms are variables declared in the programmable shaders that allow the application to **pass data directly to the shaders**
- Uniforms are named because they **remain constant** during one draw command execution
  - Cannot be written to
  - **Every invocation of the programmable shaders access the same data content**

Uniforms

- Uniforms are generally used to provide **data that are shared across all primitives or fragments**, like:
  - Camera Projection Matrices
  - Light Source Settings
  - Texture Map Samplers (will be covered later)
- Uniforms are stored in a special hardware memory that is optimized for **synchronized read-only access pattern**
  - **Link**

Uniforms

- OpenGL provides two mechanisms for declaring and storing uniforms:
- **Default Block Uniform**
  - Old mechanism since OpenGL 2.0
  - Data is stored in the **shader program object**
- **Buffer-Backed Uniform Block**
  - Since OpenGL 3.1
  - Data is stored in a **buffer object**

Default Block Uniform

Default Block Uniforms

- Declare default block uniforms:

```
struct MyType
{
    float var1;
    int var2;
};

uniform float float_uniform;
uniform int int_uniform;
uniform vec4 vec4_uniform;
uniform mat4 mat4_uniform;
uniform MyType type_uniform;
```

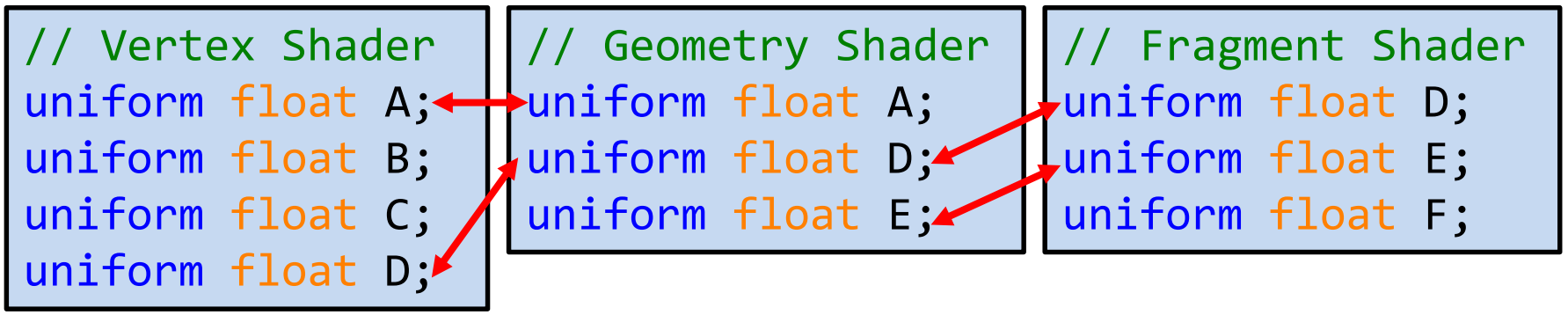
- Both intrinsic types and custom structure types are supported

Default Block Uniforms

- Uniform declarations are **shared between all programmable stages**
- **Same declarations are seem as references to the same variable**
- You don’t need to declare every uniform in every programmable stages. Only those which are accessed by the shader program are needed

Default Block Uniforms

- Consider the following GLSL program:



- There are **6** different uniforms: A, B, C, D, E, F
- Red arrows mark the uniforms that reference the same content

Default Block Uniforms

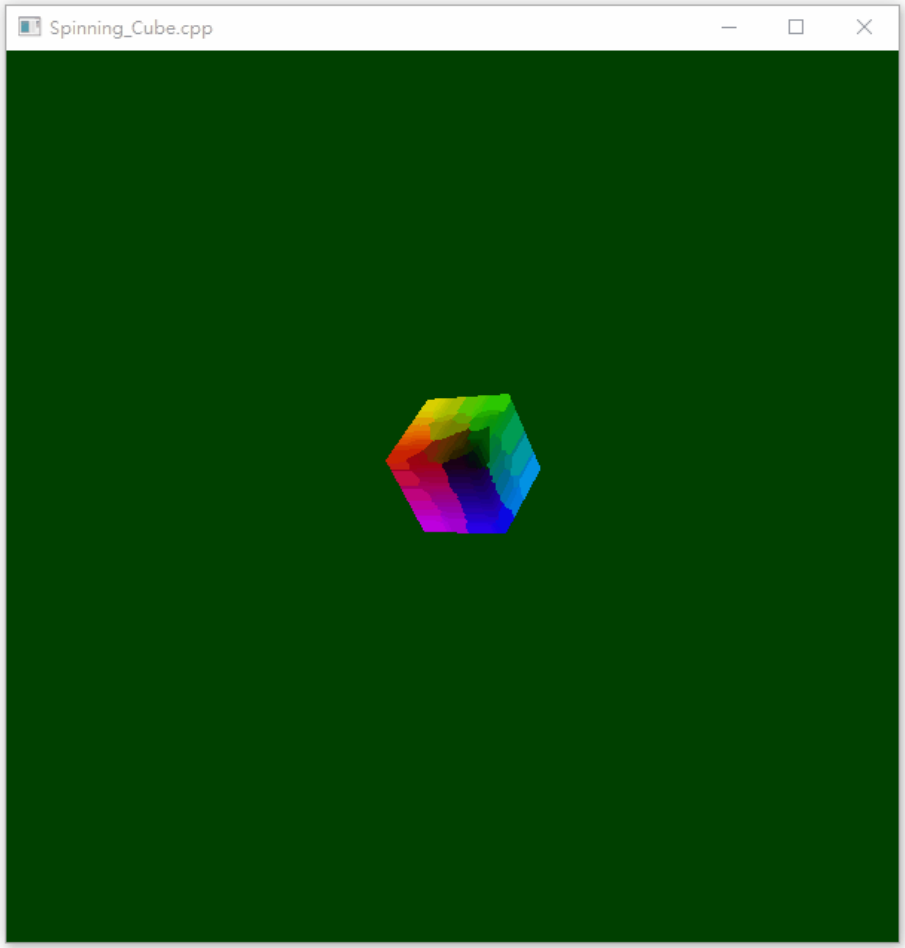
- Default block uniforms are part of the shader program object state
- You have to ***bind the shader program object to the pipeline before you try to update the uniform values***
- The uniform values will be saved in the shader program object, so **you don't need to update them every frame** if their values are not changed
- Every uniform are identified by an **uniform location**, an unsigned integer, in a program

Default Block Uniforms

- To use default block uniforms:
  1. Declare uniforms in the shader code
  2. Compile, attach and link a shader program
  3. Use **`glGetUniformLocation()`** to get the **uniform location** for every uniform in the program
  4. Use **`glUseProgram()`** to bind the shader program
  5. Use **`glUniform*()`** to update an **uniform value** at an **uniform location**
  6. Issue draw commands

Using Uniforms for Transformation Example: Spinning Cube

A Spinning Cube !



Code: Vertex Shader

```
#version 410 core

in vec4 position;

out VS_OUT
{
    vec4 color;
} vs_out;

uniform mat4 mv_matrix;
uniform mat4 proj_matrix;

void main(void)
{
    gl_Position = proj_matrix * mv_matrix * position;
    vs_out.color = position * 2.0 + vec4(0.5, 0.5, 0.5, 0.0);
}
```

Code: Fragment Shader

```
#version 410 core

in VS_OUT
{
    vec4 color;
} fs_in;

out vec4 color;

void main(void)
{
    color = fs_in.color;
}
```

Code: Global Variables

```
GLuint          vao;
GLuint          program;
GLuint          buffer;
GLint           mv_location;
GLint           proj_location;
glm::mat4       proj_matrix;
```

Code: Initialization

```
// Generate some data and put it in a buffer object
static const GLfloat vertex_positions[] = my_generate_data();
glGenBuffers(1, &buffer);
glBindBuffer(GL_ARRAY_BUFFER, buffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertex_positions),
             vertex_positions, GL_STATIC_DRAW);

// Set up our vertex attribute
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
glEnableVertexAttribArray(0);

// Compile and link program, then get uniform locations
program = my_compile_and_link_program();
mv_location = glGetUniformLocation(program, "mv_matrix");
proj_location = glGetUniformLocation(program, "proj_matrix");
```

glGetUniformLocation

*GLint glGetUniformLocation(GLuint program, const GLchar *name);*

- **program**: Specifies the program object to be queried.
- **name**: Points to string containing the name of the uniform variable whose location is to be queried.
- **Description**: *glGetUniformLocation* returns an integer that represents the location of a specific uniform variable within a program object.

Code: Reshape Function

```
void OnReshape(int w, int h)
{
    float aspect = (float)w / (float)f;

    // remember that angles are in radian in glm
    proj_matrix = glm::perspective(
        deg2rad(50.0f),
        aspect,
        0.1f,
        1000.0f
    );
}
```

Code: Render Function

```
// Clear the framebuffer with dark green
static const GLfloat green[] = { 0.0f, 0.25f, 0.0f, 1.0f };
glClearBufferfv(GL_COLOR, 0, green);

// Compute ModelView Matrix based on time
float f = (float)currentTime * 0.3f;
glm::mat4 mv_matrix = my_compute_mv_matrix(f);

// Activate our program
glUseProgram(program);

// Set the model-view and projection matrices
glUniformMatrix4fv(mv_location, 1, GL_FALSE, mv_matrix);
// Note that you can update proj_matrix in Reshape function
// The value of proj_matrix is not updated every frame!
glUniformMatrix4fv(proj_location, 1, GL_FALSE, proj_matrix);

// Draw 6 faces of 2 triangles of 3 vertices each = 36 vertices
glDrawArrays(GL_TRIANGLES, 0, 36);
```

glUniform*

void glUniform{1|2|3|4}{f|i|ui}(Glint location, GL{float/int/uint}v0 ,v1 ,v2,v3)
void glUniform{1|2|3|4}{f|i|ui}v(GLint location, GLsizei count,
*const GL{float/int/uint} *value)*
void glUniformMatrix{2|3|4|2x3|3x2|2x4|4x2|3x4|4x3}fv
*(GLint location, GLsizei count, GLboolean transpose, const GLfloat *value)*

- **location**: Specifies the location of the uniform variable to be modified.
- **count**: Specifies the number of elements that are to be modified. This should be 1 if the targeted uniform variable is not an array, and 1 or more if it is an array.

glUniform* (Cont'd)

void glUniform{1|2|3|4}{f|i|ui}(Glint location, GL{float/int/uint}v0 ,v1 ,v2,v3)
void glUniform{1|2|3|4}{f|i|ui}v(GLint location, GLsizei count,
*const GL{float/int/uint} *value)*
void glUniformMatrix{2|3|4|2x3|3x2|2x4|4x2|3x4|4x3}fv
*(GLint location, GLsizei count, GLboolean transpose, const GLfloat *value)*

- **transpose**: For the matrix commands, specifies whether to transpose the matrix as the values are loaded into the uniform variable.
- **v0,v1,v2,v3**: For the scalar commands, specifies the new values to be used for the specified uniform variable.

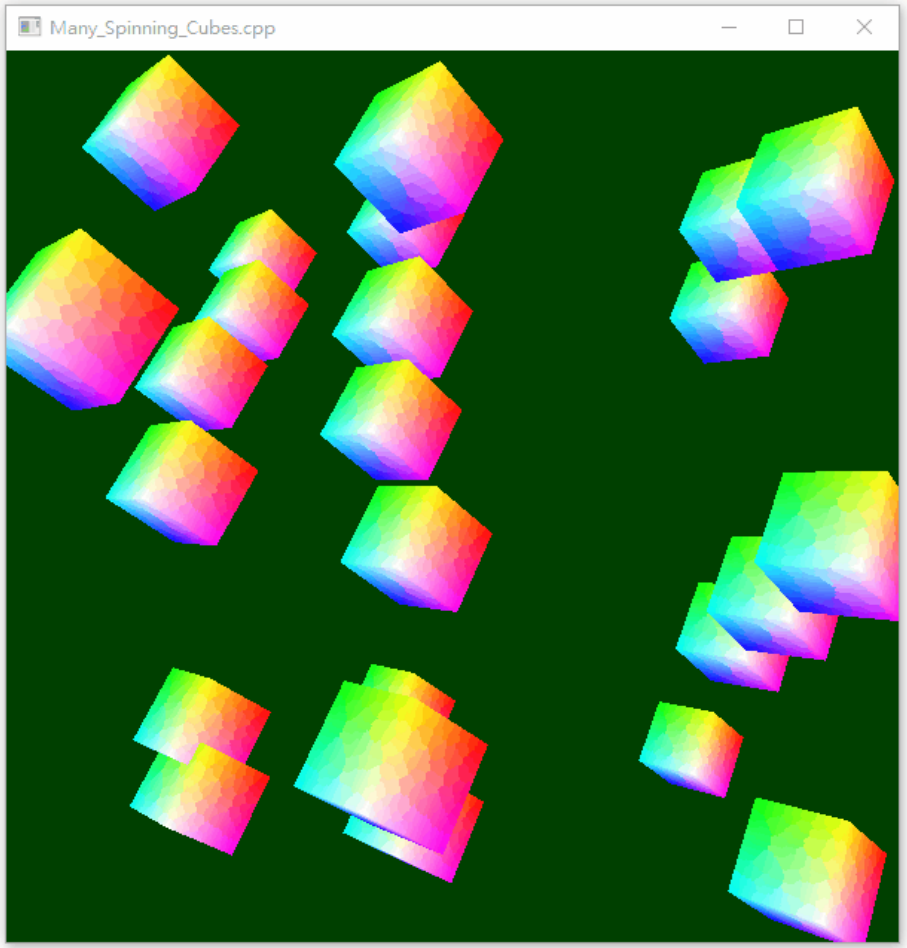
glUniform* (Cont'd)

void glUniform{1|2|3|4}{f|i|ui}(Glint location, GL{float/int/uint}v0 ,v1 ,v2,v3)
void glUniform{1|2|3|4}{f|i|ui}v(GLint location, GLsizei count,
*const GL{float/int/uint} *value)*
void glUniformMatrix{2|3|4|2x3|3x2|2x4|4x2|3x4|4x3}fv
*(GLint location, GLsizei count, GLboolean transpose, const GLfloat *value)*

- **value**: For the matrix commands, specifies a pointer to an array of count values that will be used to update the specified uniform variable.
- **Description**: *glUniform* modifies the value of a uniform variable or a uniform variable array.

Example: Many Spinning Cubes

Many Spinning Cubes



Code: Render Function

```
glUseProgram(program);

// Note that you can update proj_matrix in Reshape function
// The value of proj_matrix is not updated every frame!
glUniformMatrix4fv(proj_location, 1, GL_FALSE, proj_matrix);

for(int i = 0; i < 24; ++i)
{
    // Compute ModelView Matrix based on time with offset i
    float f = (float)i +(float)currentTime * 0.3f;
    glm::mat4 mv_matrix = my_compute_mv_matrix(f);

    // Set the model-view and projection matrices
    glUniformMatrix4fv(mv_location, 1, GL_FALSE, mv_matrix);
    glDrawArrays(GL_TRIANGLES, 0, 36);
}
```

Uniform Location Layout

Uniform Arrays

- Generally speaking, an uniform array will take **consecutive uniform locations**

```
#version 410 core

uniform float array[3];
```

```
GLint loc_array;
loc_array = glGetUniformLocation(myShader, "array[0]");

glUseProgram(myShader);
glUniform1f(loc_array, 1.0f);
glUniform1f(loc_array + 1, 2.0f);
glUniform1f(loc_array + 2, 3.0f);
```

Uniform Arrays

- This is equal to the last slide:

```
#version 410 core

uniform float array[3];
```

```
GLint loc_array;
loc_array = glGetUniformLocation(myShader, "array[0]");

glUseProgram(myShader);
float data[] = { 1.0f, 2.0f, 3.0f };
glUniform1fv(loc_array, 3, data);
```

Uniform Arrays

- This is also equal to the last slide:

```
#version 410 core

uniform float array[3];
```

```
GLint loc_array[3];
loc_array[0] = glGetUniformLocation(myShader, "array[0]");
loc_array[1] = glGetUniformLocation(myShader, "array[1]");
loc_array[2] = glGetUniformLocation(myShader, "array[2]");

glUseProgram(myShader);
glUniform1f(loc_array[0], 1.0f);
glUniform1f(loc_array[1], 2.0f);
glUniform1f(loc_array[2], 3.0f);
```

Uniform Arrays

- Note that matrices will take more uniform locations!

```
#version 410 core

uniform mat4 array[3];
```

```
GLint loc_array;
loc_array = glGetUniformLocation(myShader, "array");

glUseProgram(myShader);
glUniformMatrix4fv(loc_array, mat1);
glUniformMatrix4fv(loc_array + 4, mat2);
glUniformMatrix4fv(loc_array + 8, mat3);
```

Uniform Structures

- To use uniform structures:

```
struct MyStruct
{
    float var1;
    mat4 var2;
};
uniform MyStruct struct;
```

```
GLint loc_var1, loc_var2;
loc_var1 = glGetUniformLocation(myShader, "struct.var1");
loc_var2 = glGetUniformLocation(myShader, "struct.var2");

glUseProgram(myShader);
glUniform1f(loc_var1, 1.0f);
glUniformMatrix4fv(loc_var2, mat);
```

Uniform Location Layout

- To prevent misunderstanding or confusion about uniform locations:
  - Always use **`glGetUniformLocation()`** for **EVERY** uniform
  - Use **Buffer-Backed Uniform Block** instead

Buffer-Backed Uniform Block

Uniform Block

- OpenGL allows you to combine a group of uniforms into a **uniform block** and store the whole block in a **buffer object**
- Such uniform block is called **buffer-backed**
- Its declaration is similar to the **Interface Blocks**, but uses **uniform** qualifier for the block

Code: Declare an Uniform Block

```
// declare an uniform block with name “transform”
uniform TransformBlock
{
    float scale;
    vec3 translation;
    float rotation[3];
    mat4 projection_matrix;
} transform;

void main()
{
    // uniform values can be accessed like this
    float scale = transform.scale
}
```

Uniform Block

- The rules for matching the uniform blocks in each programmable stage are the same as in default block uniforms

```
// Vertex Shader
uniform TransformBlock
{
    float scale;
    vec3 translation;
    float rotation[3];
    mat4 projection_matrix;
} transform;
```

```
// Geometry Shader
uniform TransformBlock
{
    float scale;
    vec3 translation;
    float rotation[3];
    mat4 projection_matrix;
} transform;
```

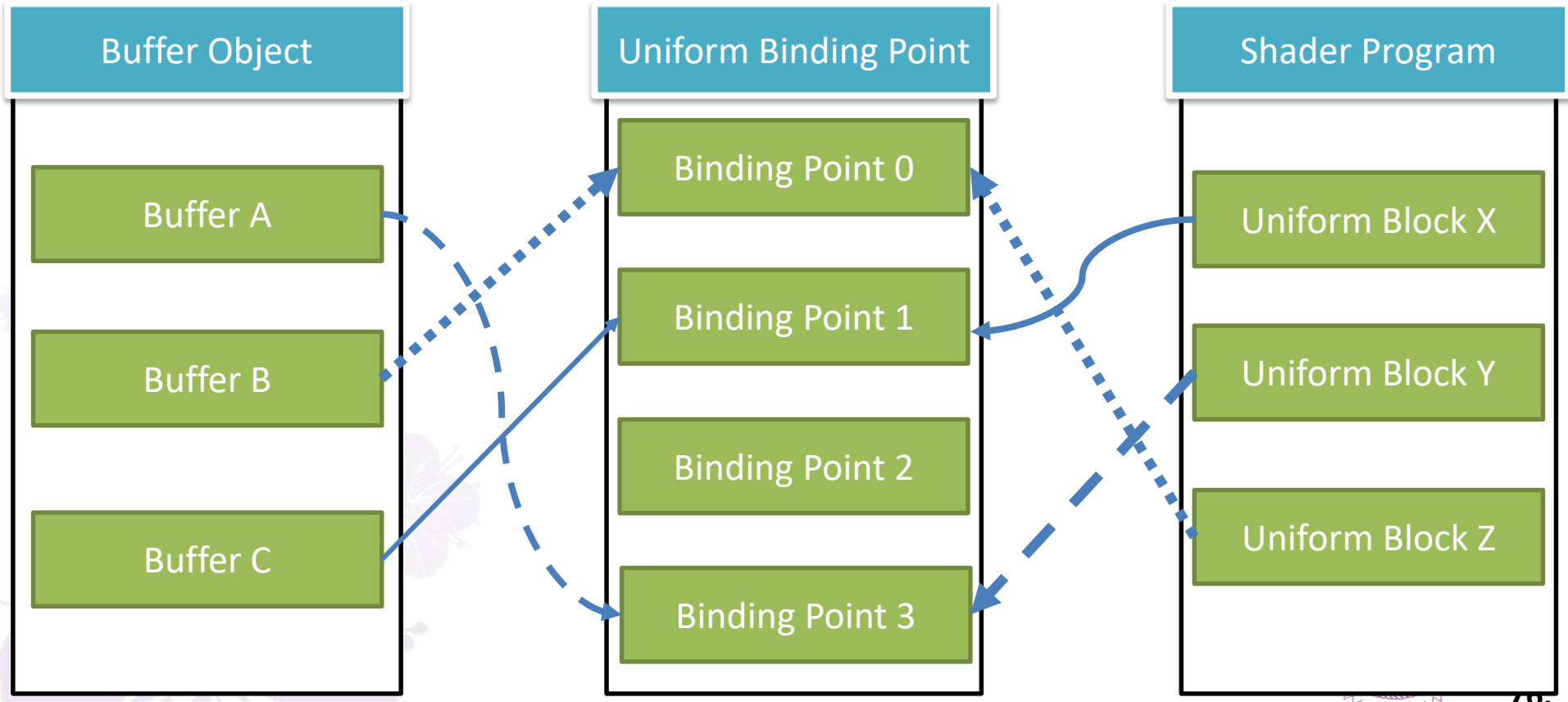
- Both uniform blocks reference the same data content, or more precisely, the same buffer object

Uniform Buffer Binding

- An uniform block in a shader program is identified by an **uniform block index**
  - Can be queried by **glGetUniformBlockIndex()**
- Remember that the type of a buffer object is decided by **how they are used**. Every type of buffer objects are **created and manipulated in the same way**
- An uniform block and an uniform buffer object can be connected by **binding to the same uniform binding point**

Diagram : Uniform Binding Point

- For example: Uniform Block X uses data from Buffer C, Y from A and Z from B



Code: Uniform Blocks

```
uniform X
{
    float a;
    mat4 b;
};

uniform Y
{
    int c;
    ivec4 d;
};

uniform Z
{
    mat4 e[10];
};
```

Code: Bindings for uniform blocks

```
// Get the indices of the uniform blocks using glGetUniformBlockIndex
GLuint X_index = glGetUniformBlockIndex(program, "X");
GLuint Y_index = glGetUniformBlockIndex(program, "Y");
GLuint Z_index = glGetUniformBlockIndex(program, "Z");

// Assign buffer bindings to uniform blocks, using their indices
glUniformBlockBinding(program, X_index, 1);
glUniformBlockBinding(program, Y_index, 3);
glUniformBlockBinding(program, Z_index, 0);

// Bind buffers to the binding points
// Binding 0, buffer B, Z's data
glBindBufferBase(GL_UNIFORM_BUFFER, 0, buffer_b);
// Binding 1, buffer C, X's data
glBindBufferBase(GL_UNIFORM_BUFFER, 1, buffer_c);
// Note that we skipped binding 2. Binding doesn't need to be consecutive
// Binding 3, buffer A, Y's data
glBindBufferBase(GL_UNIFORM_BUFFER, 3, buffer_a);
```

glGetUniformBlockIndex

GLuint glGetUniformBlockIndex(GLuint program,
*const GLchar *uniformBlockName);*

- **program:** Specifies the name of a program containing the uniform block.
- **uniformBlockName**: Specifies the address an array of characters to containing the name of the uniform block whose index to retrieve.

glUniformBlockBinding

void glUniformBlockBinding (GLuint program,
GLuint uniformBlockIndex, GLuint uniformBlockBinding);

- **program:** The name of a program object containing the active uniform block whose binding to assign.
- **uniformBlockIndex:** The index of the active uniform block within program whose binding to assign.
- **uniformBlockBinding:** Specifies the binding point to which to bind the uniform block with index uniformBlockIndex within program.

glUniformBlockBinding (Cont'd)

void glBindBufferBase(GLenum target, GLuint index, GLuint buffer);

- **Description:** Binding points for active uniform blocks are assigned using glUniformBlockBinding. Each of a program's active uniform blocks has a corresponding uniform buffer binding point.

glBindBufferBase

void glBindBufferBase(GLenum target, GLuint index, GLuint buffer);

- **target:** Specify the target of the bind operation. target must be one of GL_TRANSFORM_FEEDBACK_BUFFER or GL_UNIFORM_BUFFER
- **index:** Specify the index of the binding point within the array specified by target.
- **buffer:** The name of a buffer object to bind to the specified binding point.

glBindBufferBase (Cont'd)

void glBindBufferBase(GLenum target, GLuint index, GLuint buffer);

- **Description:** *glBindBufferBase* binds the buffer object buffer to the binding point at index index of the array of targets specified by target. Each target represents an **indexed array** of buffer binding points, as well as a single general binding point that can be used by other buffer manipulation functions such as glBindBuffer or glMapBuffer.

Data Layout of Uniform Buffers

- The buffer data should **match the uniform block layout** to make sure the data is accessed correctly
- In other words, you might need to **add "padding bytes" to align the data** based on some rules
- The uniform block generally uses a memory layout that is **different from C/C++**. You should use the **std140 layout qualifier**, or use other APIs to query the data layout

Data Layout of Uniform Buffers

Layout Qualifiers	Description
shared	Specify that the uniform block is shared among multiple programs. (This is the default layout)
packed	Lay out the uniform block to minimize its memory use; however, this generally disables sharing across programs.
std140	Use the standard layout for **uniform** blocks.
row_major	Cause matrices in the uniform block to be stored in a row-major element ordering.
column_major	Specify matrices should be stored in a column-major element ordering. (This is the default ordering)

Data Layout of Uniform Buffers

```
layout (std140, column_major) uniform X
{
    float a;
    float a2;
    mat4 b;
    vec3 c;
    float d;
}
```

Mem	0-3 Bytes	4-7 Bytes	8-11 Bytes	12-15 Bytes
0x00	a	a2	-	-
0x10	b[0][0]	b[1][0]	b[2][0]	b[3][0]
0x20	b[0][1]	b[1][1]	b[2][1]	b[3][1]
0x30	b[0][2]	b[1][2]	b[2][2]	b[3][2]
0x40	b[0][3]	b[1][3]	b[2][3]	b[3][3]
0x50	c[0]	c[1]	c[2]	-
0x60	d			

std140 Layout Rules

Variable Type	Variable Size and Alignment
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.
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.

Data Layout of Uniform Buffers

```
layout (std140, column_major) uniform X
{
    float a;
    float a2;
    mat4 b;
    vec3 c;
    float d;
}
```

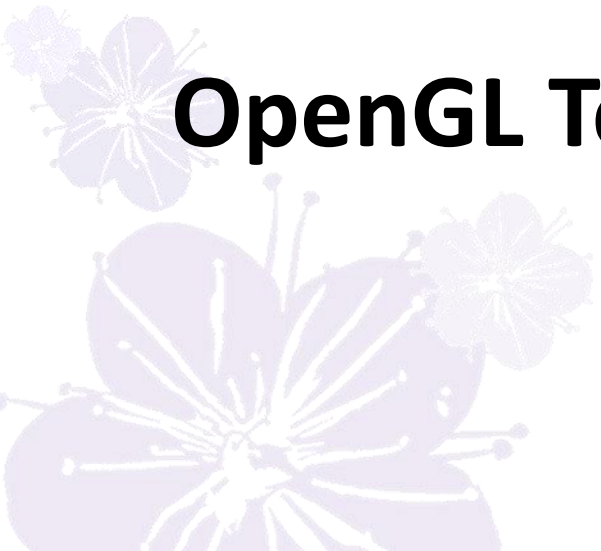
```
struct MyStruct
{
    float a;
    float a2;
    float padding_a[2];
    float b_column_1[4];
    float b_column_2[4];
    float b_column_3[4];
    float b_column_4[4];
    float c[3];
    float c_padding;
    float d;
};
```

```
MyStruct myStruct = my_uniform_data();
GLuint buffer;
glGenBuffers(1, &buffer);
glBindBuffer(GL_UNIFORM_BUFFER, buffer);
glBufferData(GL_UNIFORM_BUFFER,
    sizeof(MyStruct), &myStruct,
    GL_DYNAMIC_DRAW
);
```

Uniform Block

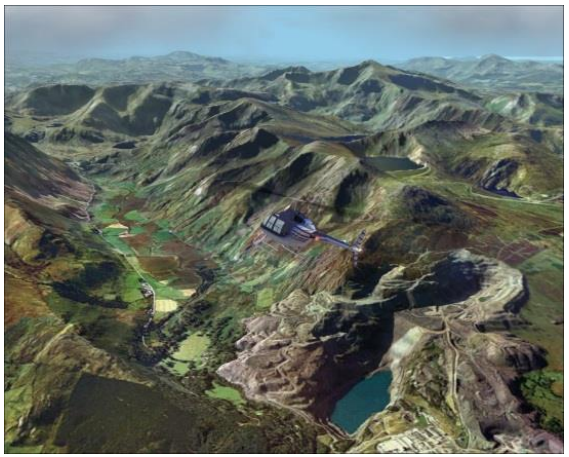
- Try to use uniform blocks instead of default block uniforms for performance gain!
  - Update every uniform value with 1 **`glBufferSubData()`** call
  - If an uniform buffer data is shared across multiple shader programs, **just bind them to the same uniform binding point** (uniform buffer object)

OpenGL Textures



The Limit of Geometric Modeling

- Modern graphics cards can render over 10 billion triangles per second
- However, geometry alone still cannot provide enough details in many photorealistic rendering contexts


Modeling an Orange

- Consider the problem of modeling an orange
- Start with an orange-colored sphere
  - Too simple
- Replace sphere with a more complex shape
  - Can not capture surface characteristics
- Add small dimples
  - It is expensive to model all the dimples
  - ~1 million triangles just for an orange?

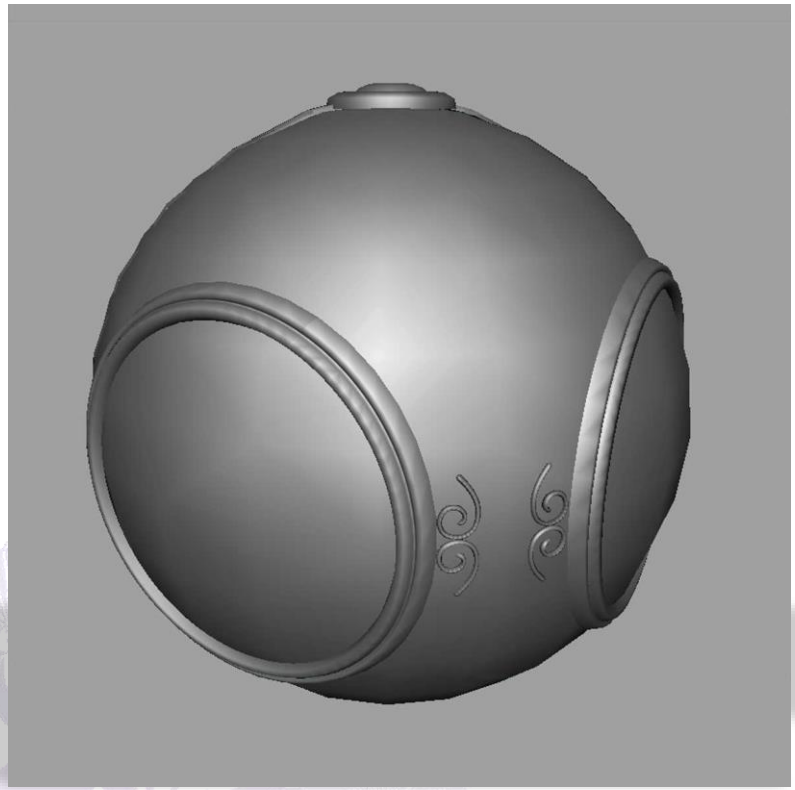
Modeling an Orange (Cont'd)

- Take a picture of a real orange and "paste" it onto a simple geometric model
- This process is known as ***texture mapping***
- Might not be good enough because the surface will be too smooth!
  - Need to change local shape
  - Use ***Bump Mapping***

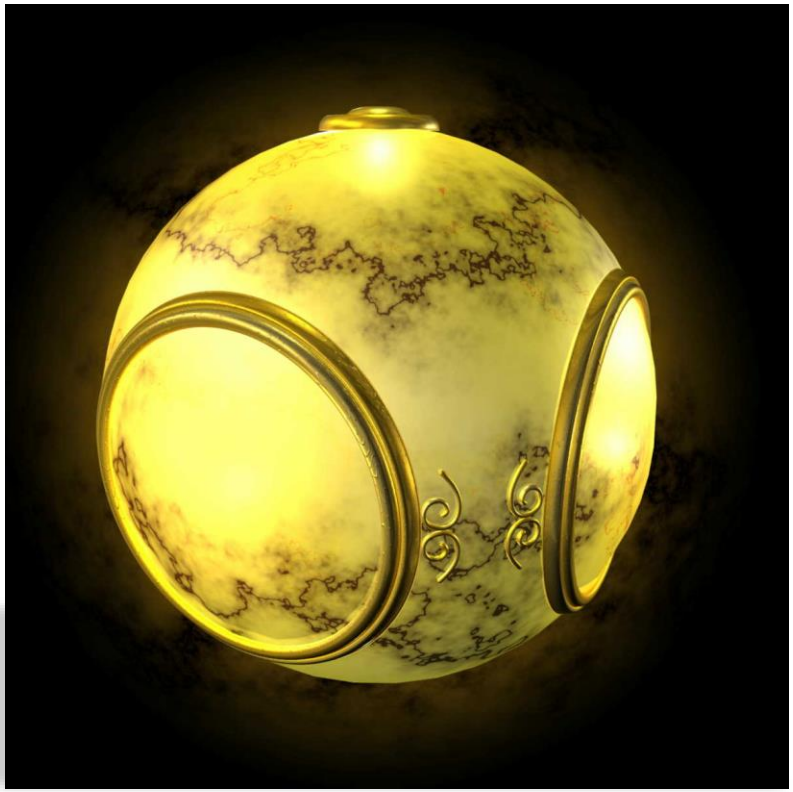
Types of Texture Mapping

- **Texture mapping**
  - Using images to fill inside of polygons
- **Environment mapping**
  - Using a picture of the environment for texture mapping
  - Simulating highly specular surfaces
- **Bump mapping**
  - Altering the surface normal during the rendering process

Texture Mapping

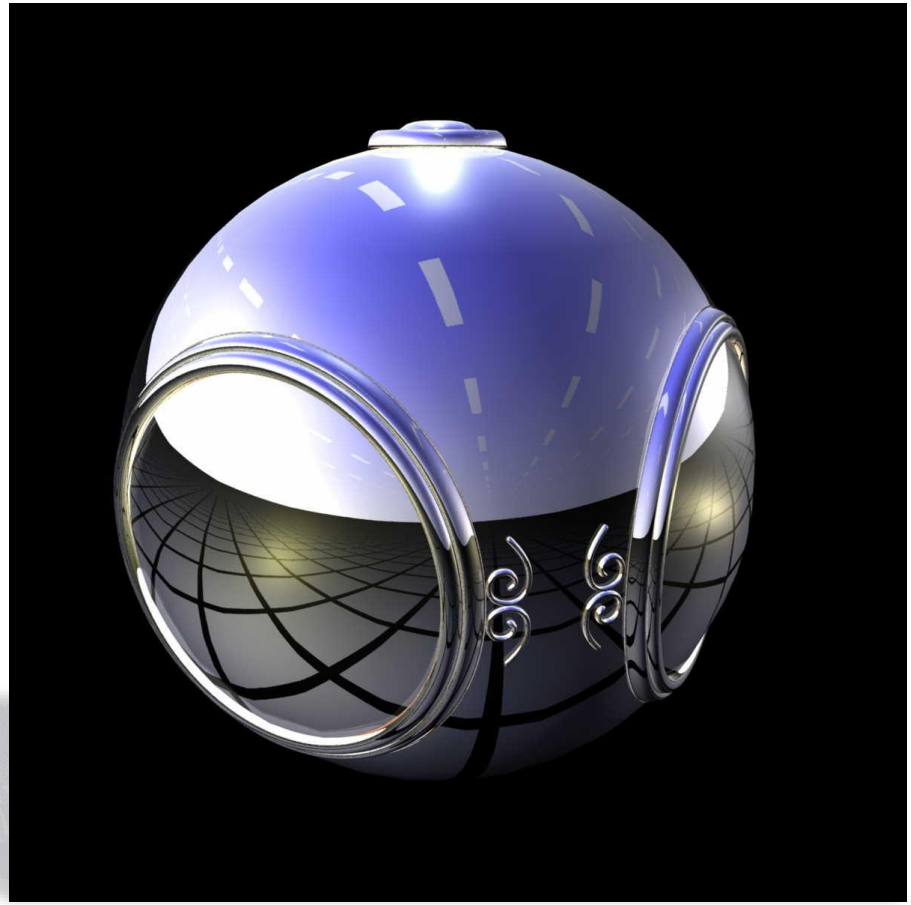


geometric model

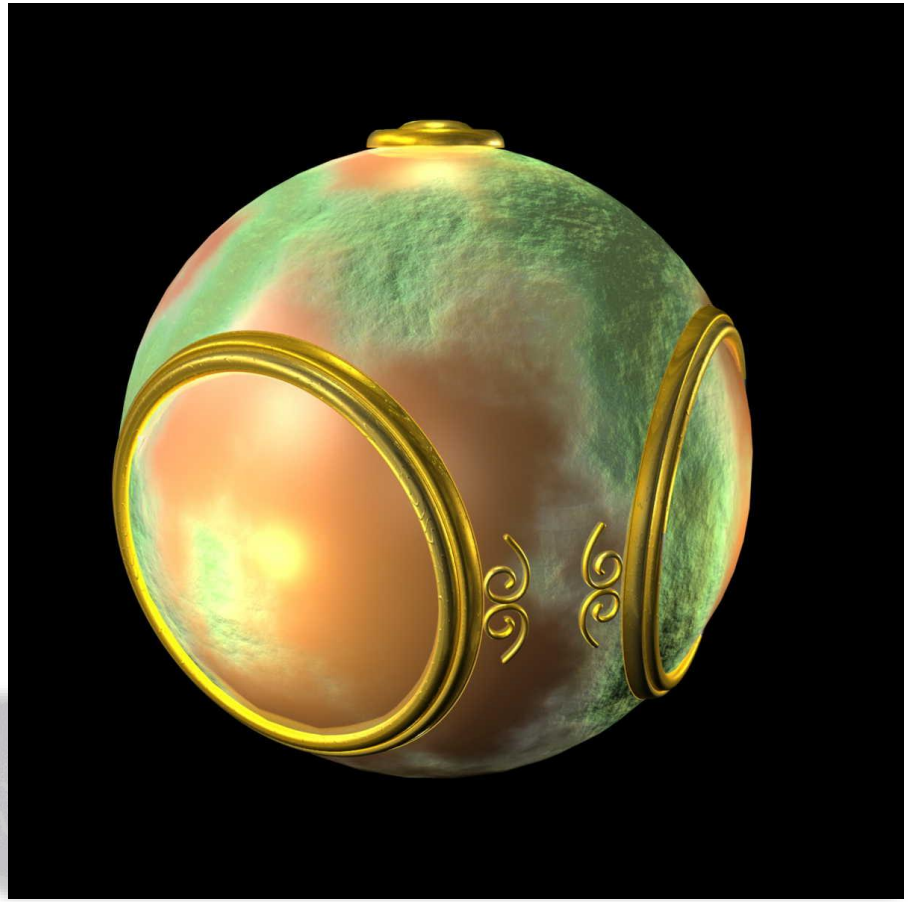


texture mapped

Environment Mapping

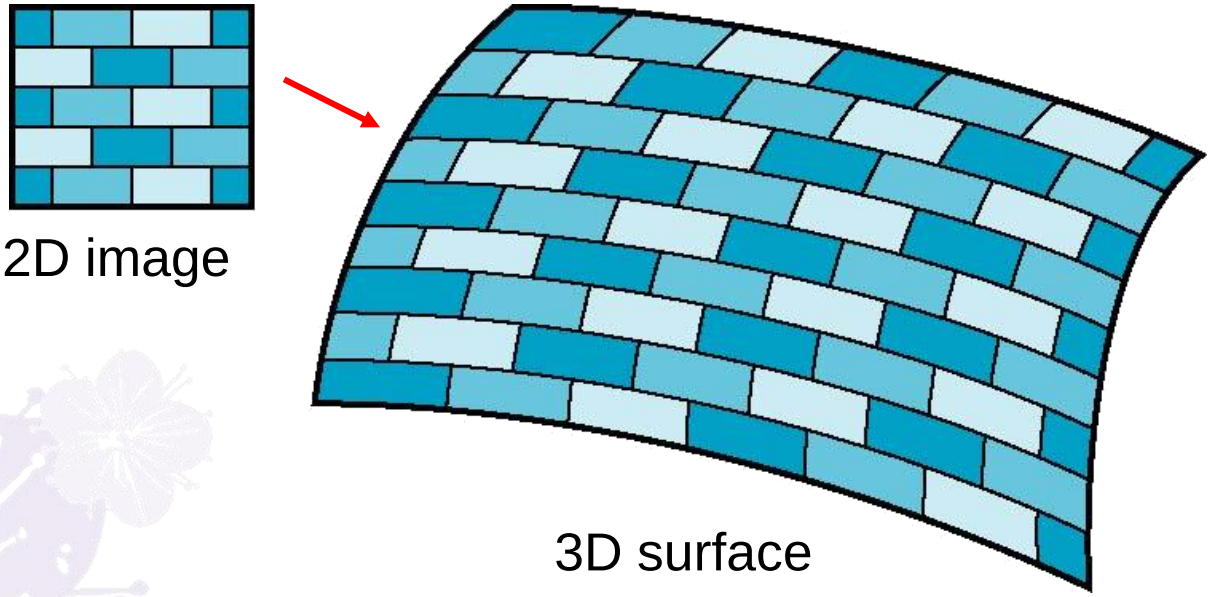


Bump Mapping

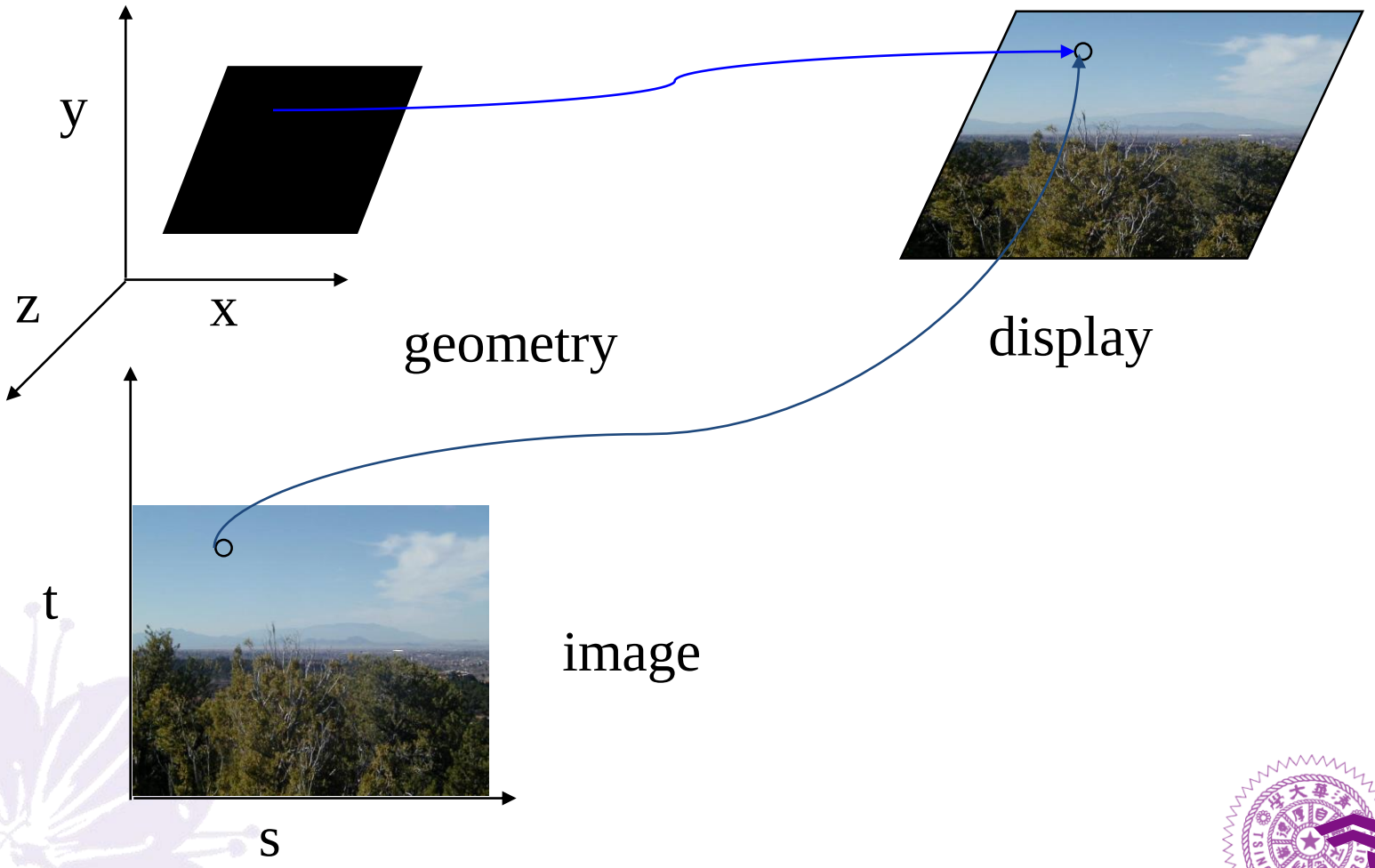


Is it Simple?

- Although the idea is simple (map an image to a surface), there are 3 or 4 coordinate systems involved



Texture Mapping

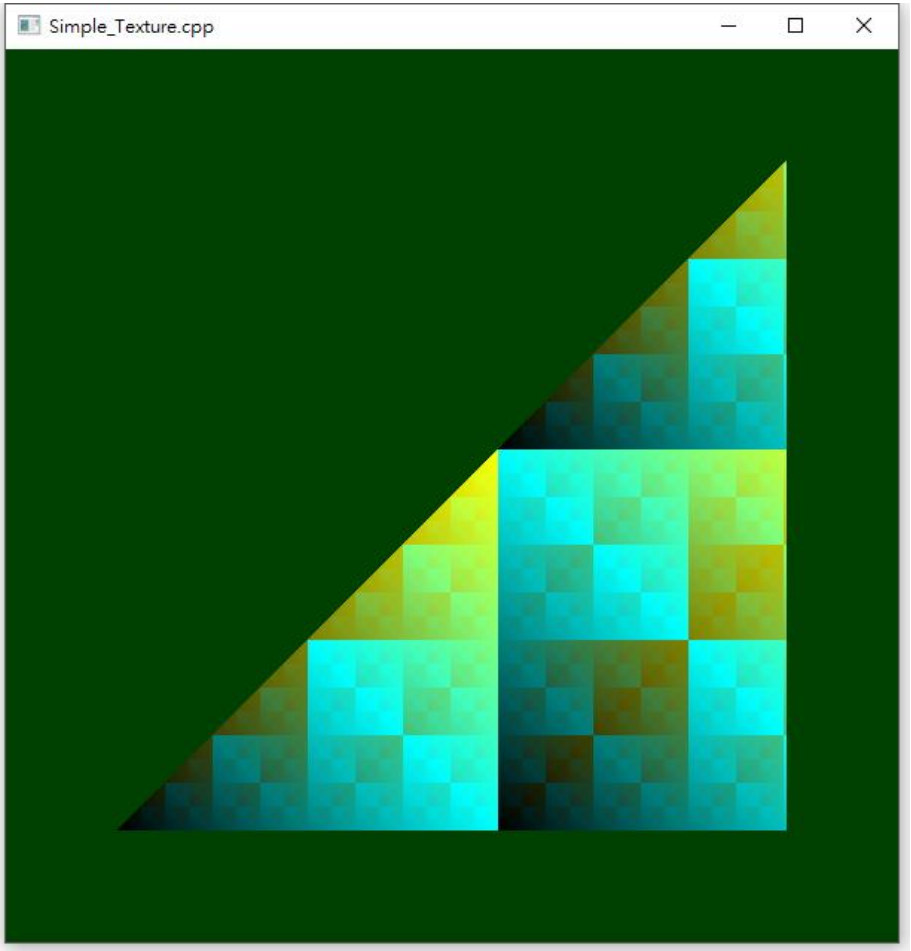


Textures

- A block of memory that is similar to an image
  - A **texel** is its smallest element
- Often used to provide **surface details**
  - Texture-mapped geometry rendering
- Additional dedicated hardware on the GPU is available for certain **texture operations**

Creating and Initializing Textures
Example: Simple Texture

A Simple Texture



Texture Objects

- Texture object manipulations are very similar to the buffer object ones

1. Call **`glGenTextures()`** to generate names for textures
2. Call **`glBindTexture()`** to bind the texture name to a texture binding target
3. Call **`glTexStorage*()`** to initialize texture data storage
4. Call **`glTexParameter*()`** to configure the texture parameters of the texture
5. Call **`glTexSubImage*()`** to update texture content

Texture Objects

- Similar to uniform buffers, texture objects can be accessed in shader programs by declaring **sampler uniforms**

```
uniform sampler2D texture1;
uniform sampler3D texture2;
uniform sampler1DArray texture3;
```

- Sampler uniforms are basically **integer type uniforms**. Its value indicates the **texture image unit** being used by the sampler

Diagram : Texture Image Unit

- For example: Sampler X uses data from texture C, sampler Y from texture A and sampler Z from texture B

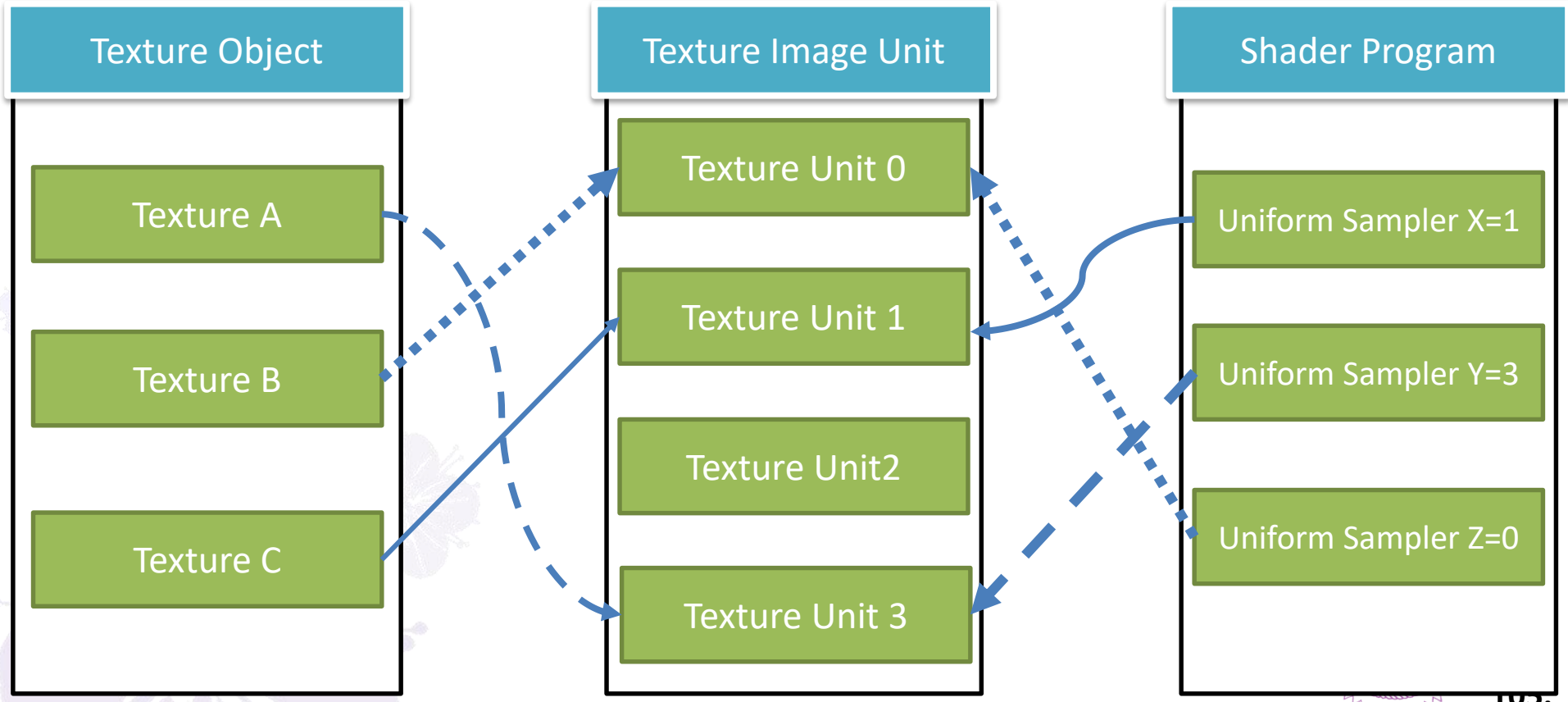
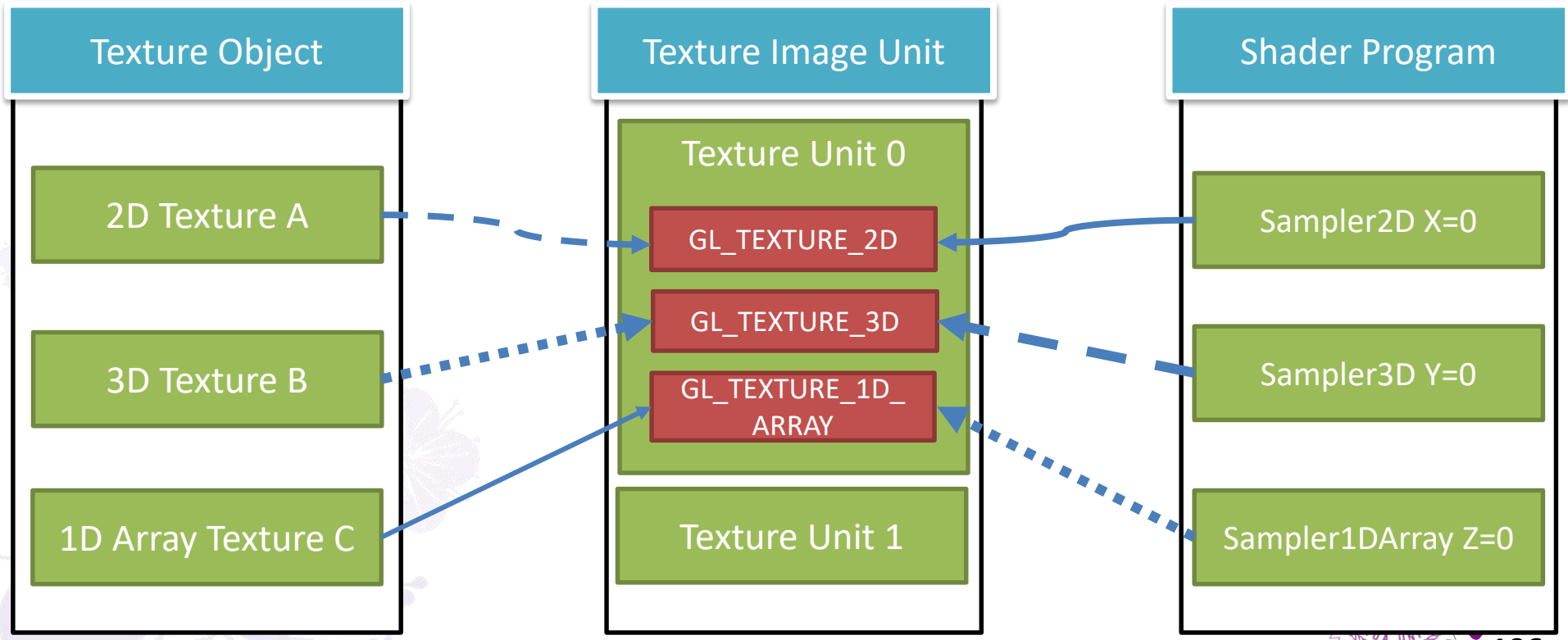


Diagram : Texture Image Unit

- Each Texture Image Unit can have **multiple textures bound to different texture targets**. It is up to **sampler uniform types** to decide which one to use



Code: Initialize Texture Object

```
// The type used for names in OpenGL is GLuint
GLuint texture;
// Generate a name for the texture
glGenTextures(1, &texture);
// Now bind it to the context using the GL_TEXTURE_2D binding point
glBindTexture(GL_TEXTURE_2D, texture);
// Specify the amount of storage we want to use for the texture
glTexStorage2D(GL_TEXTURE_2D, // 2D texture
               1, // 1 mipmap level
               GL_RGBA32F, // 32-bit floating-point RGBA data
               256, 256); // 256 x 256 texels
```

Code: Initialize Texture Object

```
// Define some data to upload into the texture
float *data = new float[256 * 256 * 4];
// generate_texture is a function that fills memory with image data
generate_texture(data, 256, 256);

// Assume the texture is already bound to the GL_TEXTURE_2D target
glTexSubImage2D(GL_TEXTURE_2D, // 2D texture
                0, // Level 0
                0, 0, // Offset 0, 0
                256, 256, // 256 x 256 texels, replace entire image
                GL_RGBA, // Four channel data
                GL_FLOAT, // Floating-point data
                data); // Pointer to data
// Free the memory we allocated before - OpenGL now has our data
delete[] data;
```

Code: generate_texture

```
void generate_texture(float *data, int width, int height)
{
    int x, y;
    for (y = 0; y < height; y++)
    {
        for (x = 0; x < width; x++)
        {
            data[(y * width + x) * 4 + 0] = (float)((x & y) & 0xFF) / 255.0f;
            data[(y * width + x) * 4 + 1] = (float)((x | y) & 0xFF) / 255.0f;
            data[(y * width + x) * 4 + 2] = (float)((x ^ y) & 0xFF) / 255.0f;
            data[(y * width + x) * 4 + 3] = 1.0f;
        }
    }
}
```

glTexStorage2D

void glTexStorage2D(GLenum target, GLsizei levels,
GLenum internalformat, GLsizei width, GLsizei height);

- **target**: Specifies the target to which the texture object is bound for glTexStorage2D.
- **levels**: Specify the number of texture levels.
- **internalformat**: Specifies the sized internal format to be used to store texture image data.
- **width**: Specifies the width of the texture in texels.
- **Height**: Specifies the height of the texture in texels.

glTexStorage2D (cont'd)

void glTexStorage2D(GLenum target, GLsizei levels,
GLenum internalformat, GLsizei width, GLsizei height);

- **Description:** glTexStorage2D and specify the storage requirements for all levels of a two-dimensional texture or one-dimensional texture array simultaneously.

glTexSubImage2D

void glTexSubImage2D(GLenum target, GLint level, GLint xoffset,
GLint yoffset, GLsizei width, GLsizei height,
*GLenum format, GLenum type, const GLvoid * pixels);*

- **target:** Specifies the target to which the texture is bound for.
- **levels**: Specify the number of texture levels.
- **xoffset , yoffset**: Specifies a texel offset in the x and y direction.
- **width**: Specifies the width of the texture in texels.
- **Height**: Specifies the height of the texture in texels.

glTexSubImage2D (cont'd)

void glTexSubImage2D(GLenum target, GLint level, GLint xoffset,
GLint yoffset, GLsizei width, GLsizei height,
*GLenum format, GLenum type, const GLvoid * pixels);*

- **format:** Specifies the format of the pixel data.
- **type:** Specifies the data type of the pixel data.
- **pixels**: Specifies a pointer to the image data in memory.
- **Description:** Texturing maps a portion of a specified texture image onto each graphical primitive for which texturing is enabled.

Code: Fragment Shader

```
#version 410 core

uniform sampler2D texture;

out vec4 color;

void main(void)
{
    color = texelFetch(texture, ivec2(gl_FragCoord.xy), 0);
}
```

Code: Render Function

```
// Clear the framebuffer with dark green
static const GLfloat green[] = { 0.0f, 0.25f, 0.0f, 1.0f };
glClearBufferfv(GL_COLOR, 0, green);

glUseProgram(program);
glBindVertexArray(vao);

// Bind the texture to Texture Image Unit 0
glActiveTexture(GL_TEXTURE0);
// glBindTexture will bind Texture Objects to Texture Image Units!
glBindTexture(GL_TEXTURE2D, texture);
// Set the sampler2D uniform value to 0 (Texture Image Unit 0)
GLuint texture_location = glGetUniformLocation(program, "texture");
glUniform1i(texture_location, 0);

// Draw a triangle
glDrawArrays(GL_TRIANGLES, 0, 3);
```

Texture Targets and Types

Texture Target (GL_TEXTURE_*)	Description
1D	One-dimensional texture
2D	Two-dimensional texture
3D	Three-dimensional texture
RECTANGLE	Rectangle texture
1D_ARRAY	One-dimensional array texture
2D_ARRAY	Two-dimensional array texture
CUBE_MAP	Cube map texture
CUBE_MAP_ARRAY	Cube map array texture
BUFFER	Buffer texture
2D_MULTISAMPLE	Two-dimensional multi-sample texture
2D_MULTISAMPLE_ARRAY	Two-dimensional array multi-sample texture

Texture Targets and Types

Texture Target (GL_TEXTURE_*)	Sampler Type
1D	sampler1D
2D	sampler2D
3D	sampler3D
RECTANGLE	sampler2DRect
1D_ARRAY	sampler1DArray
2D_ARRAY	sampler2DArray
CUBE_MAP	samplerCube
CUBE_MAP_ARRAY	samplerCubeArray
BUFFER	samplerBuffer
2D_MULTISAMPLE	sampler2DMS
2D_MULTISAMPLE_ARRAY	sampler2DMSArray

texelFetch

gvec4 texelFetch(gsampler{123}D{Array} sampler, ivec{123} P, int lod)

- **sampler**: Specifies the sampler to which the texture from which texels will be retrieved is bound.
- **p**: Specifies the texture coordinates at which texture will be sampled.
- **lod**: If present, specifies the level-of-detail within the texture from which the texel will be fetched.

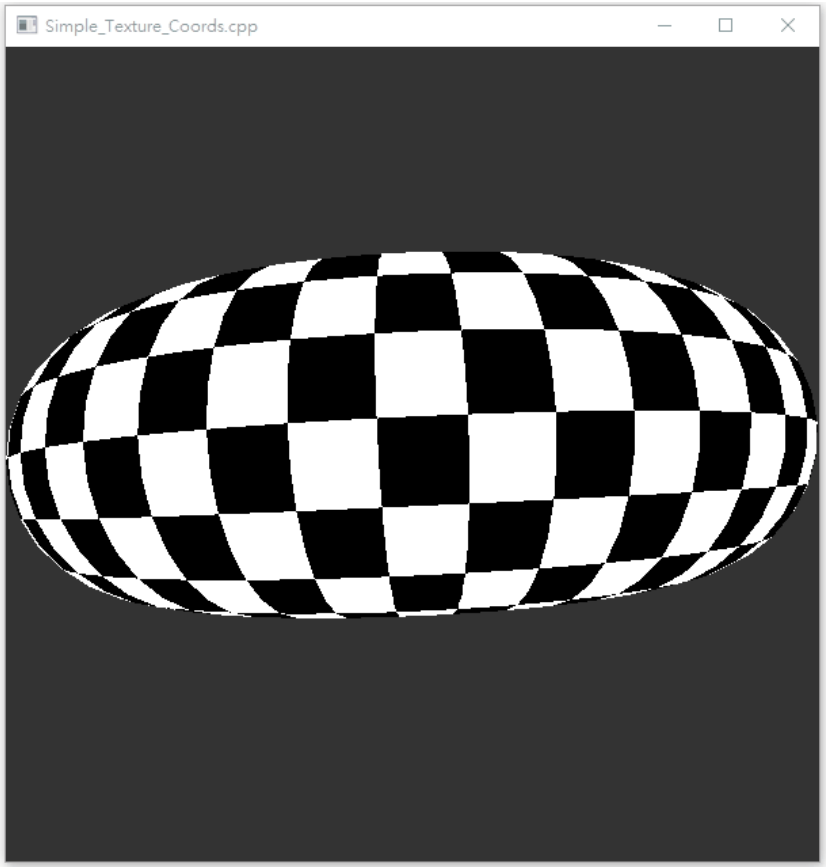
texelFetch

gvec4 texelFetch(gsampler{123}D{Array} sampler, ivec{123} P, int lod)

- **Description**: texelFetch performs a lookup of a single texel from texture coordinate P in the texture bound to sampler.

Loading Textures from Files
Example: Simple Texture Coords

A Simple Textured Model



Code: Vertex Shader

```
#version 410 core

layout (location = 0) in vec4 position;
layout (location = 4) in vec2 tc;

uniform mat4 mv_matrix;
uniform mat4 proj_matrix;

out VS_OUT
{
    vec2 tc;
} vs_out;

void main(void)
{
    // Calculate the position of each vertex
    vec4 pos_vs = mv_matrix * position;
    // Pass the texture coordinate through
    vs_out.tc = tc;
    gl_Position = proj_matrix * pos_vs;
}
```

Code: Fragment Shader

```
#version 410 core

layout uniform sampler2D tex_object;

in VS_OUT
{
  vec2 tc;
} fs_in;

void main(void)
{
  // Simply read from the texture at the (scaled)
  // coordinates, and assign the result to the
  // shader's output.
  color = texture(tex_object, fs_in.tc * vec2(3.0, 1.0));
}
```

texture

gvec4 texture(gsampler{{123}D, Cube}{Shadow, Array}sampler,
vec4 P)

- **sampler:** Specifies the sampler to which the texture from which texels will be retrieved is bound.
- **P**: Specifies the texture coordinates at which texture will be sampled.
- **Description**: *texture* samples texels from the texture bound to sampler at texture coordinate P.

texelFetch v.s. texture

- *texture* is the usual texture access function which handles **filtering and normalized** ([0,1]) texture coordinates
- *texelFetch* directly accesses a texel in the texture (no filtering) using unnormalized coordinates (e.g. (64,64) in the middle-ish texel in a 128x128 texture vs (0.5,0.5) in normalized coordinates)

Using Multiple Textures

Code: Fragment Shader

```
#version 410 core

uniform sampler2D texture1;
uniform sampler2D texture2;

out vec4 color;

void main(void)
{
    // Perform texture operations…
}
```

Code: Render Function

```
glUseProgram(program);
glBindVertexArray(vao);

// Bind texture1 to Texture Image Unit 0
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE2D, texture1);

// Bind texture2 to Texture Image Unit 1
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE2D, texture2);

// Set the sampler2D uniform value to 0 (Texture Image Unit 0)
GLuint texture_location1 = glGetUniformLocation(program, "texture1");
glUniform1i(texture1_location, 0);

// Set the sampler2D uniform value to 1 (Texture Image Unit 1)
GLuint texture_location2 = glGetUniformLocation(program, "texture2");
glUniform1i(texture2_location, 1);

// Draw a triangle
glDrawArrays(GL_TRIANGLES, 0, 3);
```

Texture Operations

Texture Operations

- As mentioned before, additional dedicated hardware on the GPU is available for certain **texture operations**
  - Texture Filtering
  - Texture Mipmapping
  - Texture Wrapping
  - Texture Comparison
- Texture operations, especially **texture filtering**, is essential for producing good image quality
- This is also one of the major differences between textures and buffers: **textures can be filtered**, while buffer can not

Texture Filtering

- Goal: compute the **total contribution of all texels covered by a fragment**
- Image the fragment being a "**Window**" of a room
- You are allowed to use **only one color** to describe the outside view
- In theory, the best approaching answer is the **average color of all inbound light rays**

Texture Filtering

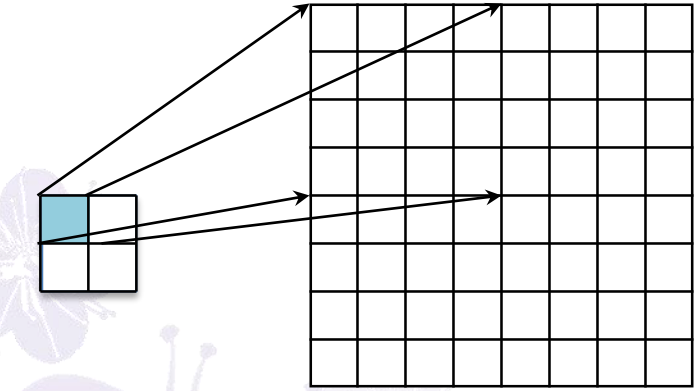
Which Color Will You Use To Describe This View?

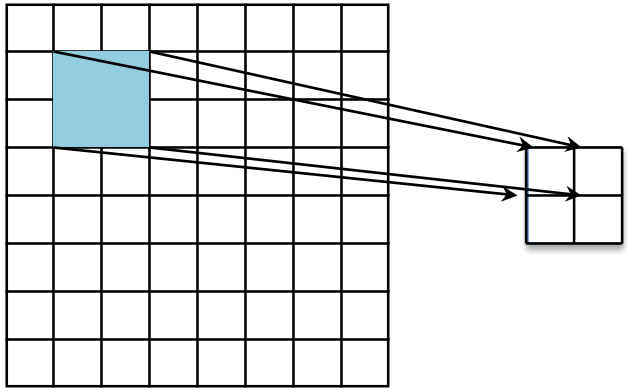
Texture Filtering

- To correctly compute the average color, methods like **Monte Carlo sampling** should be used
  - For textures, this process is called “filtering”
  - The **cost is too high** for real-time rendering
- OpenGL provides simplified, hardware-accelerated versions of texture filtering:
  - Mipmapping
  - Nearest-Neighbor Filtering (Use the Nearest Texel)
  - Bilinear Interpolation Filtering (Use 2x2 Neighbor)
  - Anisotropic Filtering

Magnification and Minification

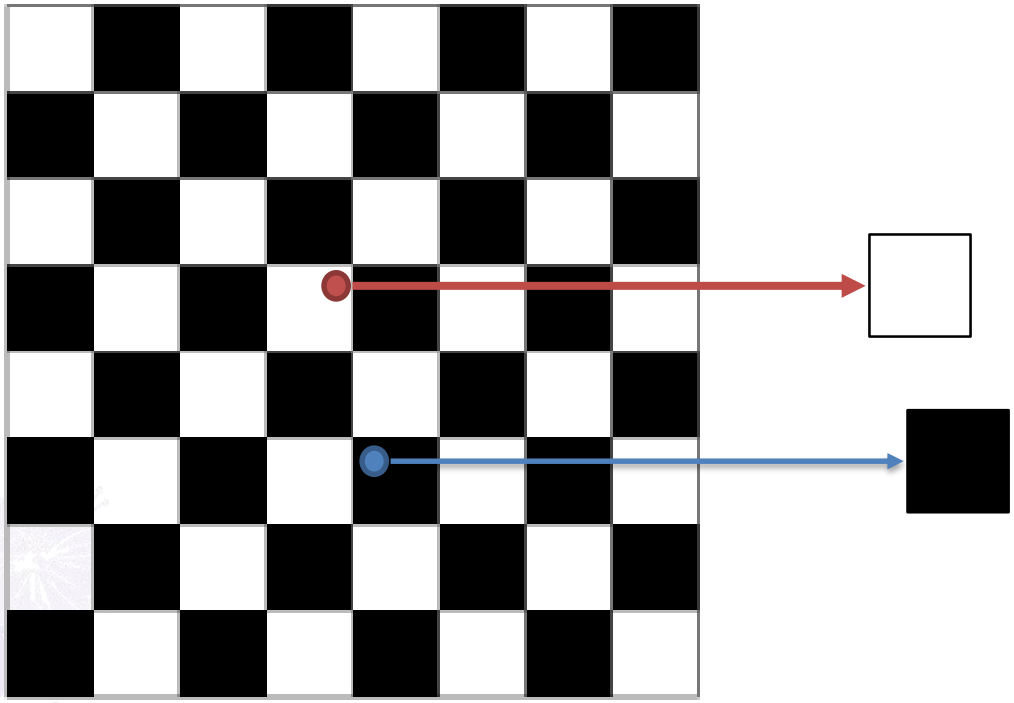
- **Magnification**
  - We want to interpolate the color to avoid "**blocky**" effect
- **Minification**
  - We want to fetch all the texels with 1 texture read





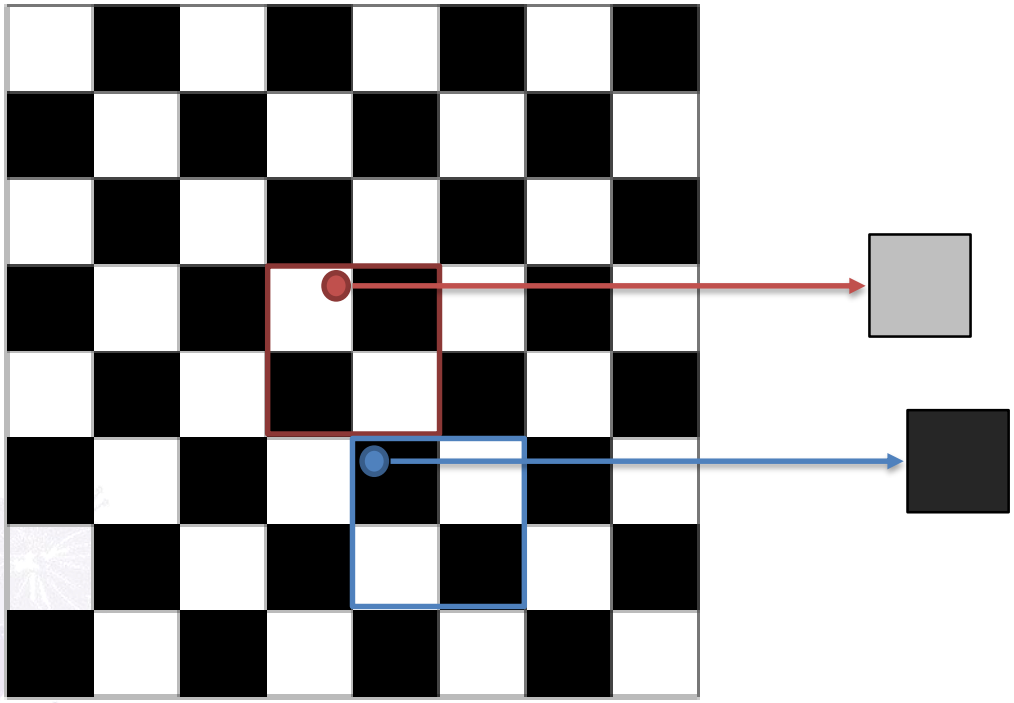
Nearest-Neighbor Filtering

- The texel which is the nearest to the sample location is used. Cheap, but produces lowest image quality
  - Sometimes used deliberately to produce low-pixel style (e.g. Minecraft)



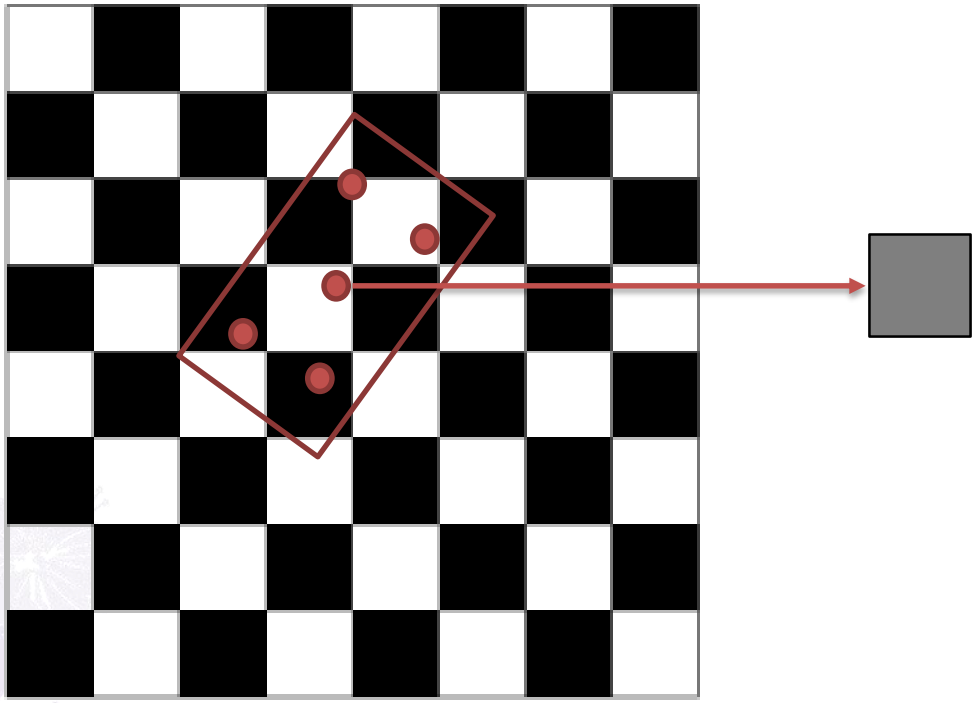
Bilinear Interpolation Filtering

- The returned value is interpolated based on the 2x2 region of the sample location. Slower than nearest-neighbor, but with better quality



Anisotropic Filtering

- The returned value is interpolated based on a region that respects to the camera projection. Can be very slow due to more samples taken, but the quality is the best



*This function is implementation-dependent. Your hardware might not support it!

Mipmapping

- Each ***level*** is the ***pre-filtered*** 2x2 average of its last level, from full-resolution to 1x1 size
  - Effectively make the sample region larger for **minification**
  - **1 texel read in level 1 = 4 texel reads in level 0!**

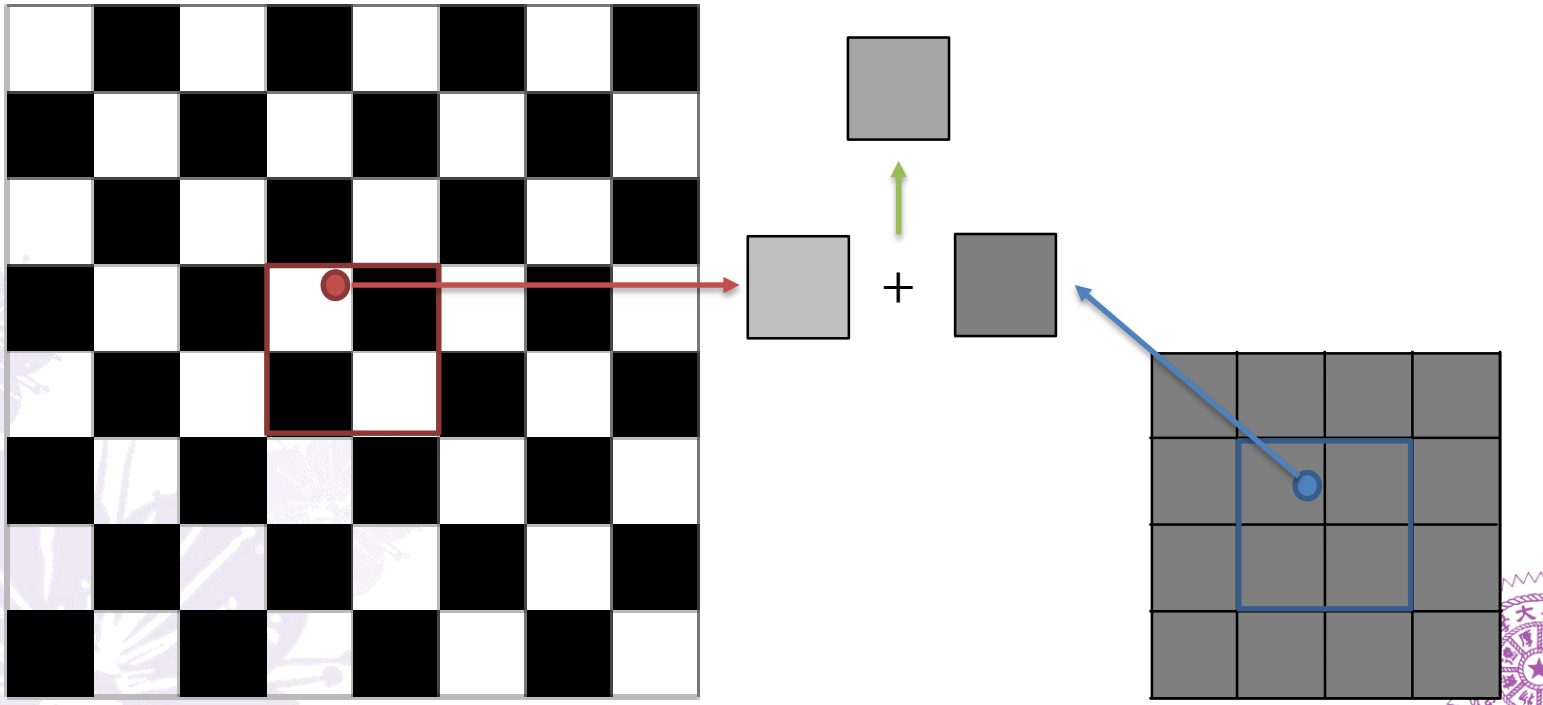
Level 0

Level 1

Level 2

Mipmapping

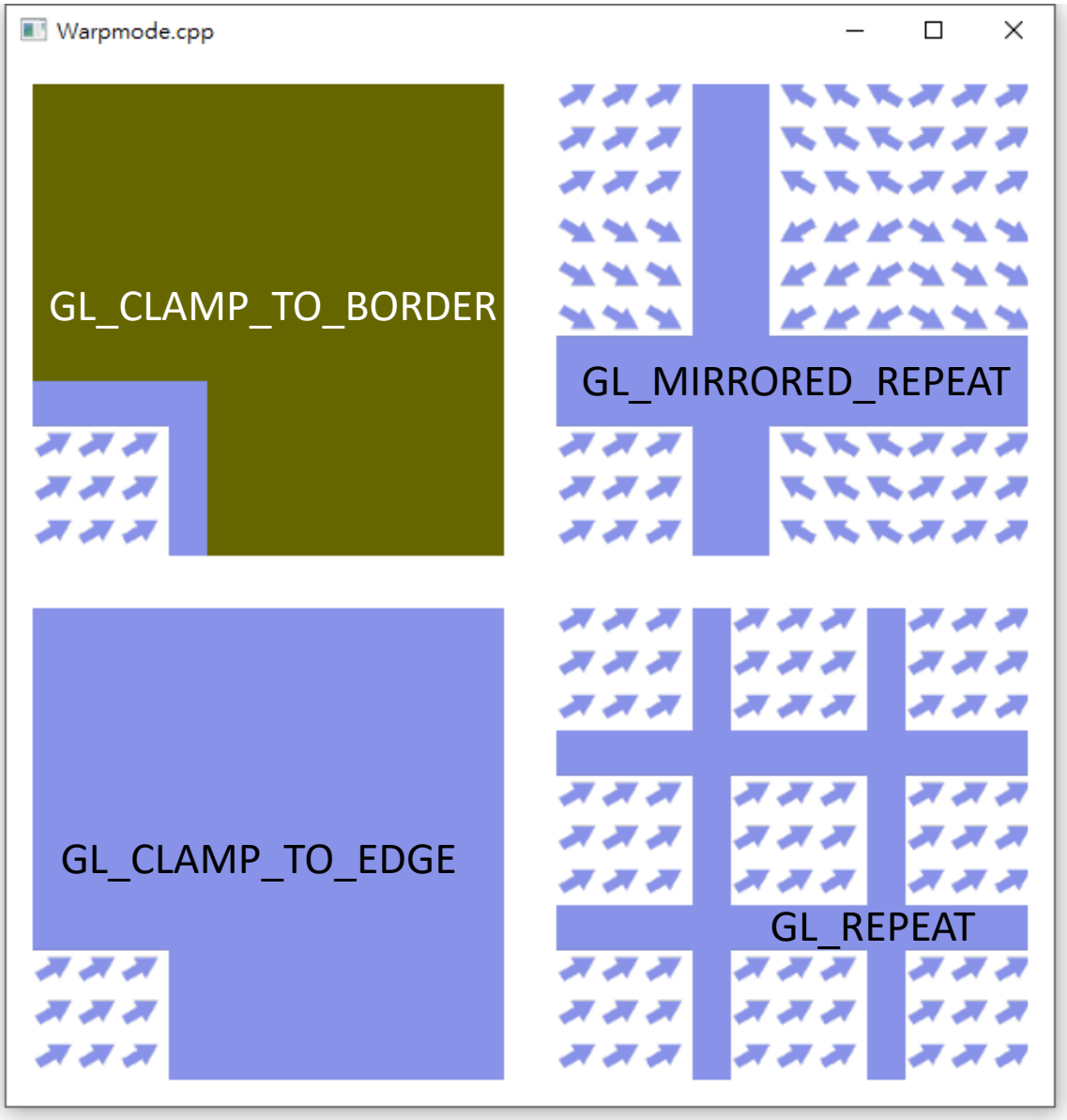
- The filtered results on different levels of mipmaps can be further interpolated to produce smooth transition between mipmap levels



Texture Wrapping

- Define the behavior of texture fetch for UV (ST) coordinates beyond $[0,1]$
- Set the **GL_TEXTURE_WRAP_S** and **GL_TEXTURE_WRAP_T** texture parameters
- Example: Wrapmode

Texture Wrapping



Texture Comparison

- **Perform a comparison** before returning the value of the texture fetch
- The value will be 1.0 if the comparison passes; 0.0 if it fails. The results are then combined with the chosen filtering mode and returned
- Set the **GL_TEXTURE_COMPARE_MODE** and **GL_TEXTURE_COMPARE_FUNC** texture parameters
- Used to accelerate **shadow mapping** algorithms

Texture Parameter

- These operations can be configured by setting the **texture parameters**

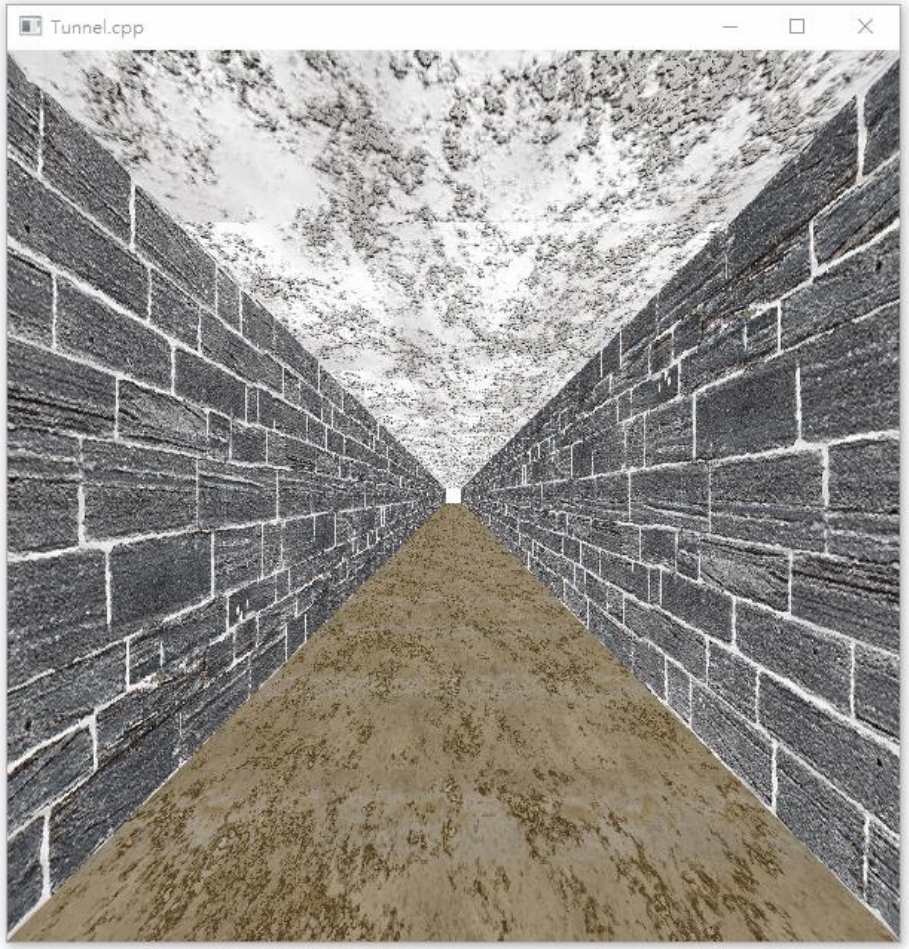
```
glBindTexture(GL_TEXTURE_2D, texture);

// (1) nearest-neighbor filtering
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

// (2) or bilinear filtering
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

// (3) or bilinear filtering with mipmapping
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
```

Texture Parameter

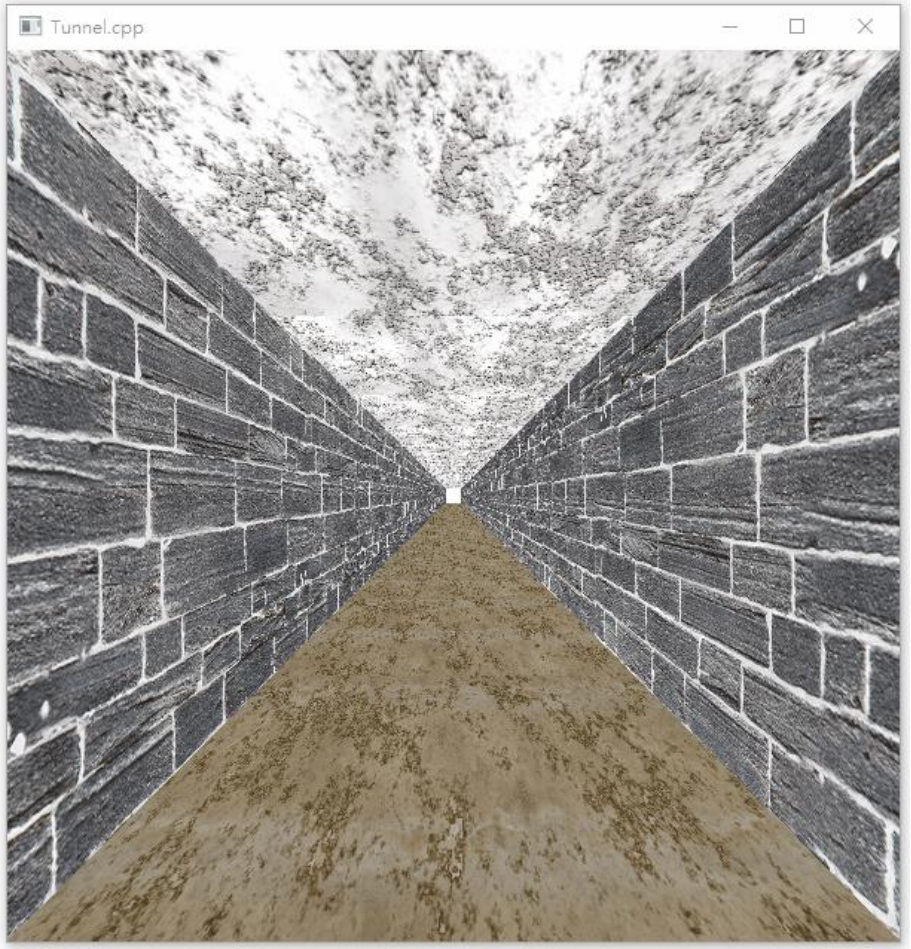
- Please refer to the docs for detailed usages:
  - https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glTexParameter.xhtml
  - https://www.khronos.org/opengl/wiki/Sampler_Object
- To use anisotropic filtering:
  - http://gamedev.stackexchange.com/questions/69374/how-to-achieve-anisotropic-filtering
  - Note: your hardware might not support this!

Mipmaps & Texture Filtering Example: Tunnel

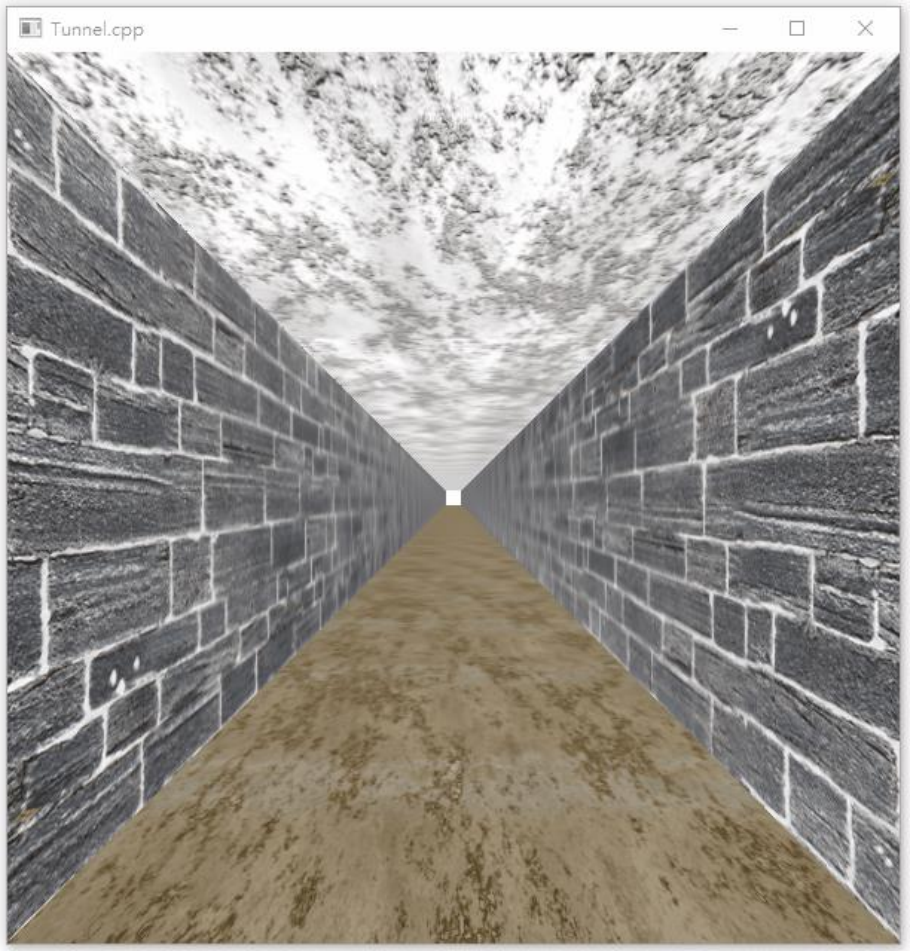
Nearest Neighbor



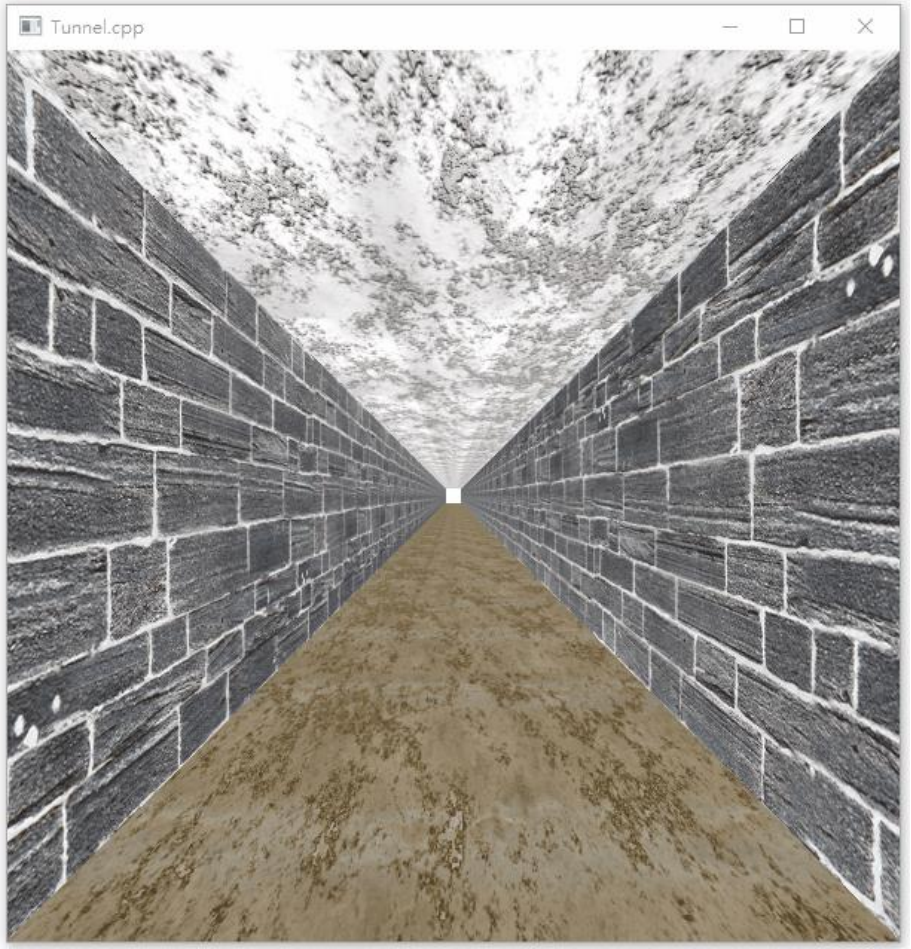
Bilinear Filtering



Bilinear Filtering with Mipmapping



Anisotropic Filtering with Mipmapping



Mipmaps

- Mipmap levels are loaded with **glTexSubImage2D**

*void glTexSubImage2D(GLenum target, **GLint level**, GLint xoffset,*
GLint yoffset, GLsizei width, GLsizei height,
*GLenum format, GLenum type, const GLvoid * pixels);*

- When you allocate your texture with **glTexStorage2D,** you can set the number of levels to include in the texture in the levels parameter. Then, you can use mipmapping with the levels present in the texture.

*void glTexStorage2D(GLenum target, **GLsizei levels**,*
GLenum internalformat, GLsizei width, GLsizei height);

Generating Mipmap Levels

- It is convenient and somewhat common to have OpenGL generating the texture mipmap levels
- You can generate all the mipmap with the function **`glGenerateMipmap()`** once level 0 is loaded by **`glTexSubImage*()`**

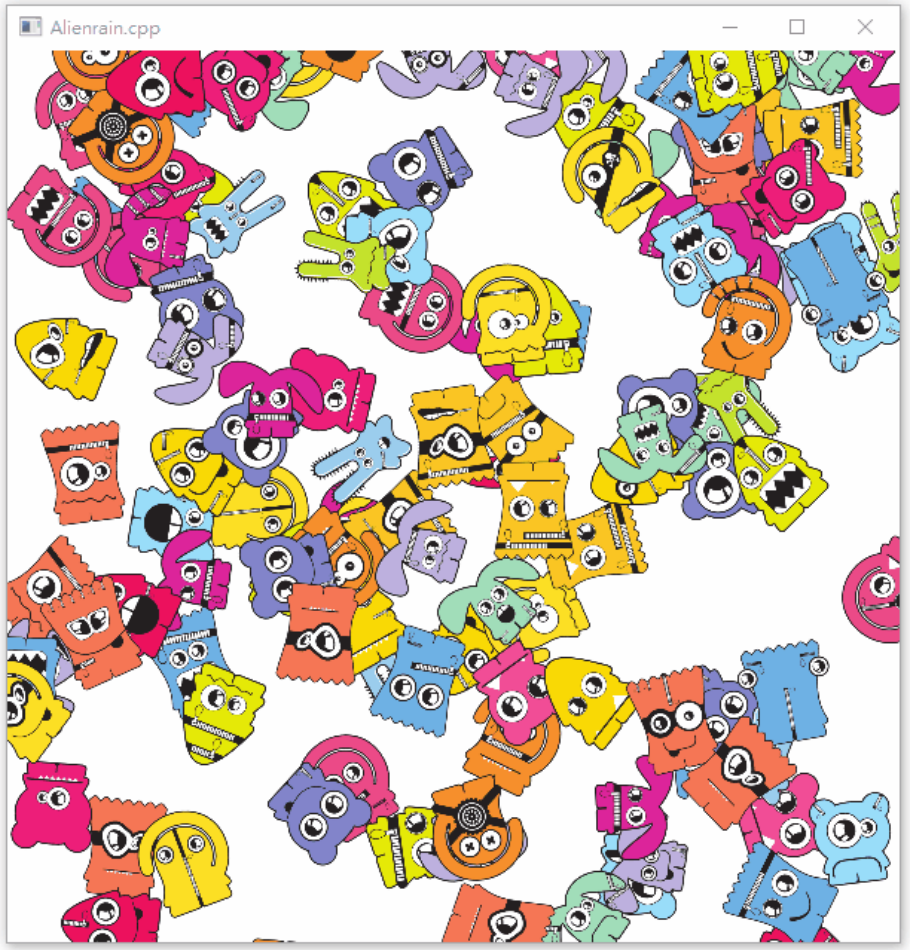
glGenerateMipmap

*void glGenerateMipmap (*GLenum *target);*

- **target:** Specifies the target to which the texture object is bound for glGenerateMipmap.
- **Description:** *glGenerateMipmap* generates mipmaps for the specified texture object. For *glGenerateMipmap*, the texture object is that bound to the target.

Array Textures
Example: Alienrain

Alien Rain!



Array Textures

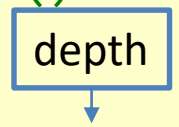
- **Array textures** allow a single texture object to contain many **texture layers**
- Every type except 3D textures can be array'ed
  - GL_TEXTURE_2D_ARRAY, etc.
  - 3D texture arrays are not supported (not very useful for now)
- Allows **customized texture data structures**. **Reduced texture binding calls** when many textures are used together

Code: Variables

```
GLuint          program;
GLuint          vao;
GLuint          vertex_shader;
GLuint          fragment_shader;
GLuint          tex_alien_array; //array texture object
GLuint          rain_buffer; // UBO for storing
                             // location and orientation

float           droplet_x_offset[256];
float           droplet_rot_speed[256];
float           droplet_fall_speed[256];

unsigned char **image_data;
```

Code: Initialization

```
// for location and orientation, there are 256 aliens in our program
glGenBuffers(1, &rain_buffer);
glBindBuffer(GL_UNIFORM_BUFFER, rain_buffer);
glBufferData(GL_UNIFORM_BUFFER, 256 * sizeof(glm::vec4), NULL,
GL_DYNAMIC_DRAW);

// [256*256*64] Only 64 kinds of alien textures
// note that 2D array data is specified by glTexStorage3D() calls!
glGenTextures(1, &tex_alien_array);
glBindTexture(GL_TEXTURE_2D_ARRAY, tex_alien_array);
glTexStorage3D(GL_TEXTURE_2D_ARRAY, 8, GL_RGBA8, 256, 256, 64);

for (int i = 0; i < 64; i++)
{
  glTexSubImage3D(GL_TEXTURE_2D_ARRAY, 0, 0, 0, i, 256, 256, 1,
                  GL_RGBA, GL_UNSIGNED_BYTE, image_data[i]);
}
```



Code: Initialization (Cont'd)

```
for (int i = 0; i < 256; i++)
{
 droplet_x_offset[i]  = random_float() * 2.0f - 1.0f;
 droplet_rot_speed[i]=(random_float()+0.5f)*((i & 1)?-3.0f:3.0f);
 droplet_fall_speed[i]= random_float() + 0.2f;
}
```

Code: Render Function

```
void render()
{
    static const GLfloat black[] = { 0.0f, 0.0f, 0.0f, 0.0f };
    float t = (float)currentTime;
    glClearBufferfv(GL_COLOR, 0, black);
    glUseProgram(program);
    glBindBufferBase(GL_UNIFORM_BUFFER, 0, rain_buffer);

    glm::vec4*droplet = (glm::vec4*)glMapBufferRange
        (GL_UNIFORM_BUFFER,0,256*sizeof(glm::vec4),
        GL_MAP_WRITE_BIT|GL_MAP_INVALIDATE_BUFFER_BIT);
```

Code: Render Function (Cont'd)

```
    for (int i = 0; i < 256; i++)
    {
        droplet[i][0] = droplet_x_offset[i];
        droplet[i][1] =
        2.0f - fmodf((t + float(i)) * droplet_fall_speed[i], 4.31f);
        droplet[i][2] = t * droplet_rot_speed[i];
        droplet[i][3] = 0.0f;
    }

    glUnmapBuffer(GL_UNIFORM_BUFFER);

    int alien_index;
    for (alien_index = 0; alien_index < 256; alien_index++)
    {
        glVertexAttribI1i(0, alien_index);
        glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
    }
}
```

Code: Vertex Shader

```
#version 410 core

layout (location = 0) in int alien_index;
out VS_OUT
{
    flat int alien;
    vec2 tc;
} vs_out;
struct droplet_t
{
    float x_offset;
    float y_offset;
    float orientation;
    float unused;
};
layout(std140) uniform droplets
{
    droplet_t droplet[256];
};
```

Code: Vertex Shader (Cont'd)

```
#version 410 core

void main(void)
{
    const vec2[4] position =vec2[4](vec2(-0.5, -0.5),
                vec2(0.5,-0.5),vec2(-0.5, 0.5),vec2( 0.5, 0.5));
    vs_out.tc = position[gl_VertexID].xy + vec2(0.5);
    float co = cos(droplet[alien_index].orientation);
    float so = sin(droplet[alien_index].orientation);
    mat2 rot = mat2(vec2(co, so),vec2(-so, co));
    vec2 pos = 0.25 * rot * position[gl_VertexID];
    gl_Position = vec4(pos.x + droplet[alien_index].x_offset,
                pos.y + droplet[alien_index].y_offset,0.5, 1.0);

    vs_out.alien = alien_index % 64; //for texture index
    //use int(mod(float(alien_index),64.0)) if your card doesn't
    //support % operator
}
```

Code: Fragment Shader

```
#version 410 core

layout (location = 0) out vec4 color;
layout uniform sampler2DArray tex_aliens;

in VS_OUT
{
    flat int alien;
    vec2 tc;
} fs_in;


void main(void)
{
    color=texture(tex_aliens,vec3(fs_in.tc,
                                  float(fs_in.alien)));
}
```

flat Qualifier

- In the general case, there is not a 1:1 mapping between a vertex and a fragment. By default, the associated data per vertex is **interpolated** across the primitive to generate the corresponding associated data per fragment
- Using the flat keyword, no interpolation is performed, so every fragment generated during the rasterization of that particular primitive will get the same data