Computer Graphics

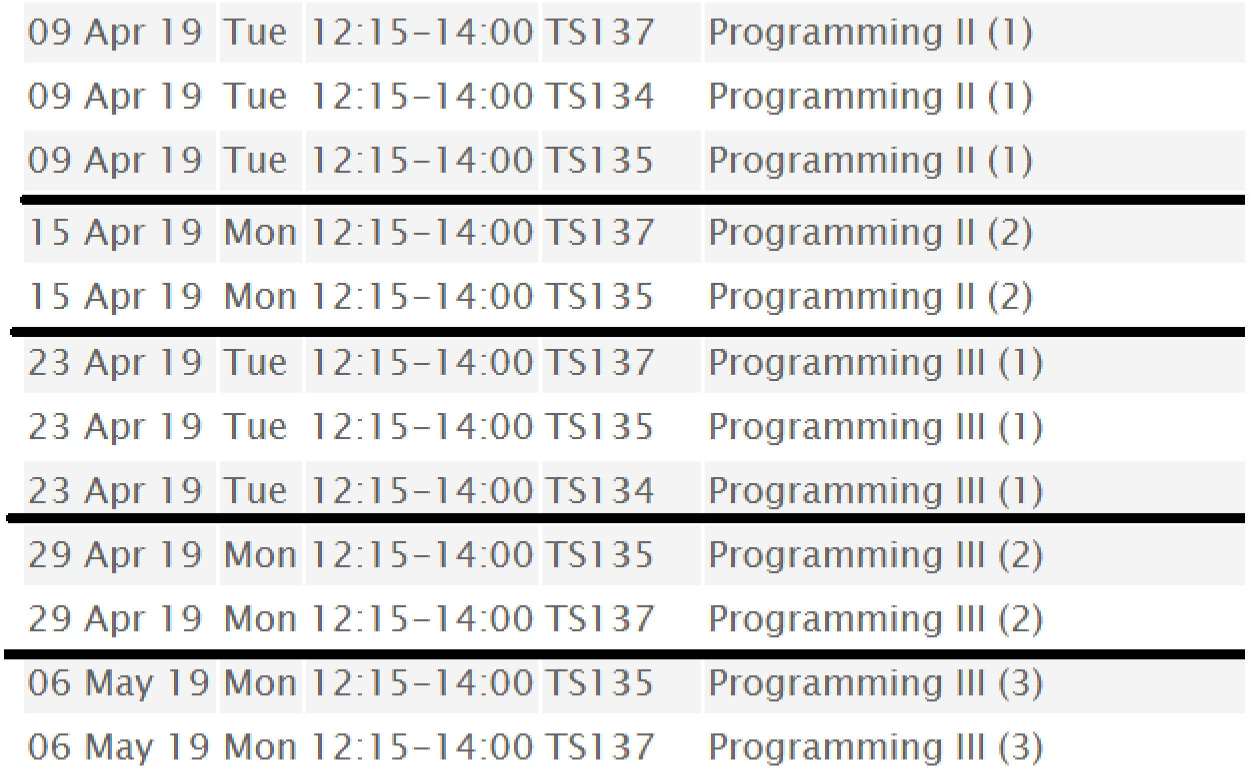
Programming ||

UNIVERSITY OF OULU

Programming Assignment 1

- Deadline today!

Additional Programming Sessions



09 Apr 19	Tue	12:15-14:00	TS137	Programming II (1)
09 Apr 19	Tue	12:15-14:00	TS134	Programming II (1)
09 Apr 19	Tue	12:15-14:00	TS135	Programming II (1)
15 Apr 19	Mon	12:15-14:00	TS137	Programming II (2)
15 Apr 19	Mon	12:15-14:00	TS135	Programming II (2)
23 Apr 19	Tue	12:15-14:00	TS137	Programming III (1)
23 Apr 19	Tue	12:15-14:00	TS135	Programming III (1)
23 Apr 19	Tue	12:15-14:00	TS134	Programming III (1)
29 Apr 19	Mon	12:15-14:00	TS135	Programming III (2)
29 Apr 19	Mon	12:15-14:00	TS137	Programming III (2)
06 May 19	Mon	12:15-14:00	TS135	Programming III (3)
06 May 19	Mon	12:15-14:00	TS137	Programming III (3)

Programming Assignment 2

- Grading
  - – Total points as 8, and 5 to pass the assignment.
- Contents
  - – Vertex array objects, index buffers, manipulating vertices, cameras and transformations
- Deadline 2019-04-23 24:00

UNIVERSITY OF OULU

Requirements

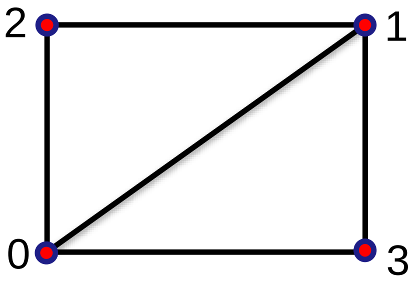
- Rendering a flag pole
  - Use index buffers, and separate Vertex array object as well as shaders
- Animate a flag
  - Calculate transformations in vertex shader
- Create a movable camera that allows you to move and look around
  - Use SDL event handling to catch events
  - Manipulate the view matrix according to events
- Back face culling

How models are created in existing samples

- ExampleScene1
  - Each vertex contains position and color
  - No Element Array Buffer used
- ExampleScene2
  - Each vertex contains position and texture coordinate
  - Element Array Buffer is used
- ExampleScene3 and ExampleScene4
  - Each vertex contains position, color and normal
  - Element Array Buffer is used
- What is needed for Programming assignment 2
  - Position, color or texture coordinate needed for sure
  - Element Array Buffer to be used

Index Buffer Objects

- Provides a level of indirection. Stores vertex indexes in a vertex buffer.
  - No need to duplicate per vertex data
  - Only helps when ALL vertex attributes are the same



```
GLushort indices[num_indices] = { 0, 1, 2, 0, 3, 1};
GLuint ibo;
glGenBuffers(1, &ibo); // Create Index Buffer Object
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo); // Bind it for use
glBufferData(GL_ELEMENT_ARRAY_BUFFER, num_indices * sizeof(GLushort),
&indices[0], GL_STATIC_DRAW);
...
// Draw using indirection: glDrawElements(type, count, index type, offset)
glDrawElements(GL_TRIANGLES, static_cast<GLsizei>(indices.size()),
GL_UNSIGNED_SHORT, 0);

// Draw without indirection: glDrawArrays(type, offset, count)
// glDrawArrays(GL_TRIANGLES, 0, static_cast<GLsizei>(vertices.size()));
```

Defining layout of data

void glVertexAttribPointer(GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const GLvoid * pointer);

```
GLuint posAttribute = 0; // User selects attribute locations!
GLint sizeParam = 3; // Number of components per vertex [1..4]
GLenum typeParam = GL_FLOAT; // Each component is a float
GLboolean normalized = GL_FALSE; // Affects integer values
GLsizei strideParam = 3 * sizeof(GLfloat); // Step to next
const GLvoid *pointer = 0; // Offset in buffer.
glVertexAttribPointer(posAttribute, sizeParam, typeParam,
normalized, pointer);
glEnableVertexAttribArray(posAttribute); // Enable attribute
```

If strideParam is 0, data is assumed to be tightly packed.

Variable *pointer* is actual pointer type only because of legacy use of this API function. Think of it as an integer value.

Vertex Array Objects

- Vertex Array Object (VAO) stores (almost) all of the **state** needed to supply vertex data
  - What Vertex Buffer Objects to use – not the actual contents of buffers
  - The format of vertex data defined for Vertex Buffer Objects
  - Current GL_ARRAY_BUFFER binding is NOT part of the state

```
GLuint vao, ibo, vbo; // Vertex Array Object and Vertex Buffer Object handles
glGenVertexArrays(1, &vao); // Allocate a Vertex Array Object
glBindVertexArray(vao); // Bind our Vertex Array Object as the current object
glGenBuffers(1, &ibo); // Create Index Buffer Object
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo); // Bind it for use
glBufferData(GL_ELEMENT_ARRAY_BUFFER, ...); // Copy data to buffer
glGenBuffers(1, &vbo); // Allocate one Vertex Buffer Object
glBindBuffer(GL_ARRAY_BUFFER, vbo); // Bind our VBO as being the active buffer
glBufferData(GL_ARRAY_BUFFER, ...); // Copy data to buffer
glVertexAttribPointer(shaderProgram.get...AttribLocation(), ...); // VAO remembers
glEnableVertexAttribArray(shaderProgram.getPositionAttribLocation());
glBindVertexArray(0); // Break the binding with current VAO
```

Vertex Array Objects

```
// Use some shader etc, ...
...
// Bind Vertex Array Object – turns on attributes and uses correct buffers for vao1
glBindVertexArray(vao1);

// Draw elements from buffers defined for vao1
glDrawElements(GL_TRIANGLES, num_vao1_indices, GL_UNSIGNED_INT, 0);

// Bind Vertex Array Object – turns on attributes and uses buffers for vao2
glBindVertexArray(vao2);

// Draw elements from buffers defined for vao2
glDrawElements(GL_TRIANGLES, num_vao2_indices, GL_UNSIGNED_INT, 0);

// To remove any association with previous vertex array:
glBindVertexArray(0);
```

Vertex shader deformations

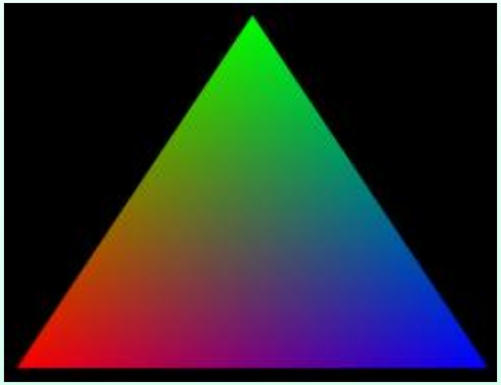
Remember this simple shader?

```
#version 330 core
in vec3 in_Position;
in vec3 in_Color;

uniform mat4 mvpmatrix;

out vec3 ex_Color;

void main(void) {


  gl_Position = mvpmatrix * vec4(in_Position, 1.0);
  ex_Color = in_Color;
}
```

Inputs

Output

Built-in output variable



Vertex shader deformations

What if ..

```
#version 330 core
in vec3 in_Position;
in vec3 in_Color;

uniform mat4 mvpmatrix;

out vec3 ex_Color;

void main(void) {
  vec4 newPos = vec4(in_Position.x, in_Position.y, 0.4 * in_Position.x *
sin(in_Position.x * 40.0), 1.0);
 gl_Position = mvpmatrix * newPos;
  ex_Color = in_Color;
}
```

Vertex shader deformations

What if we make it a bit more interesting..

```
#version 330 core
in vec3 in_Position;
in vec3 in_Color;

uniform mat4 mvpmatrix;
uniform float time;
out vec3 ex_Color;

void main(void) {
  vec4 newPos = vec4(in_Position.x, in_Position.y, 0.4 * in_Position.x *
sin(in_Position.x * 40.0 - time), 1.0);
  gl_Position = mvpmatrix * newPos;
  ex_Color = in_Color;
}
```

Camera-specific tools

- Using glm::lookAt() is very useful
  - Also used in all examples
  - Defines position with camera position, target position and a vector pointing up
    - Very intuitive
- It is easy to build different movement modes on top of that function
  - Moving while tracking a target
  - Setting target to a camera position + direction vector