3D Graphics Programming

T163 - Game Programming

Week 6

Review

OpenGL Application

1. Main program
2. Vertex Shader
3. Fragment Shader

OpenGL Application

- All starts with the main function

1- Initialize main components


```
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_RGBA | GLUT_3_2_CORE_PROFILE | GLUT_DOUBLE | GLUT_DEPTH);

glutInitWindowSize(1024, 768);
glutInitWindowPosition(0, 0);
glutCreateWindow("Hello World");

glewExperimental = true;
glewInit();    //Initializes the glew and prepares the drawing pipeline.

glEnable(GL_CULL_FACE); // cull face
glCullFace(GL_BACK); // cull back face
glFrontFace(GL_CCW); // GL_CCW for counter clock-wise
glEnable(GL_DEPTH_TEST);
```

OpenGL Application

- All starts with the main function

2- Initialize your rendering variables (ones that do not change over the course of your application)

Best practice is to have a separate function for this stage. Usually called Init or Initialize.

OpenGL Application

- All starts with the main function

2- Initialize your rendering variables (ones that do not change over the course of your application)

Main function

Init function

a- Initialize your shaders

```
//Specifying the name of vertex and fragment shaders.
ShaderInfo shaders[] = {
   { GL_VERTEX_SHADER, "triangles.vert" },
   { GL_FRAGMENT_SHADER, "triangles.frag" },
   { GL_NONE, NULL }
};

//Loading and compiling shaders
GLuint program = LoadShaders(shaders);
glUseProgram(program);    //My Pipeline is set up
```

OpenGL Application

- All starts with the main function

2- Initialize your rendering variables (ones that do not change over the course of your application)

Main function

Init function

b- Initialize your projection matrices.

```
// Get a handle for our "MVP" uniform
MatrixID = glGetUniformLocation(program, "MVP");

// Projection matrix : 45∞ Field of View, 4:3 ratio, display range : 0.1 unit <-> 100 units
Projection = glm::perspective(glm::radians(45.0f), 4.0f / 3.0f, 0.1f, 100.0f);
// Or, for an ortho camera :
Projection = glm::ortho(-1.0f,1.0f,-1.0f,1.0f,0.0f,100.0f); // In world coordinates

currentCamPos = glm::vec3(3.0f,3.0f,4.0f);

// View matrix
View = glm::lookAt(
            currentCamPos, // Camera is at (4,3,3), in World Space
            glm::vec3(0,0,0), // and looks at the origin
            glm::vec3(0,1,0)  // Head is up (set to 0,-1,0 to look upside-down)
            );
```

OpenGL Application

- All starts with the main function

  2- Initialize your rendering variables (ones that do not change over the course of your application)

Describe how each object in your scene looks like by following steps C to F. Steps C to F are to be repeated for each unique object in your scene.

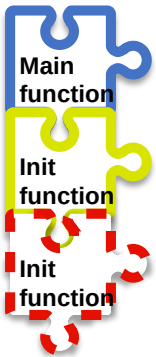
Note: two cubes of different sizes are not different objects, rather they are two instances of the same object with different transformation matrices.

Best practice is to have a separate function for each object. Better yet, a class that generates your objects with a function to generate each object resources

OpenGL Application

- All starts with the main function

2- Initialize your rendering variables (ones that do not change over the course of your application)



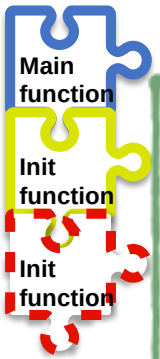
c- Initialize your VAO.

```
glGenVertexArrays(1, &cubeVAO);
glBindVertexArray(cubeVAO);
```

OpenGL Application

- All starts with the main function

2- Initialize your rendering variables (ones that do not change over the course of your application)



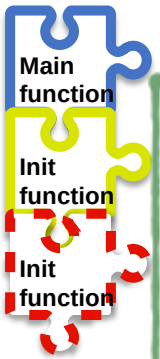
d- Initialize your VBOs to describe your object vertices, starting with your vertex positions array.

```
float cube_vertices[] = {
    // front
    -0.45, -0.45,  0.45,
     0.45, -0.45,  0.45,
     0.45,  0.45,  0.45,
    -0.45,  0.45,  0.45,
    // back
    -0.45, -0.45, -0.45,
     0.45, -0.45, -0.45,
     0.45,  0.45, -0.45,
    -0.45,  0.45, -0.45,
  };
  GLuint cubeVertices_vbo = 0;
  glGenBuffers(1, &cubeVertices_vbo);
  glBindBuffer(GL_ARRAY_BUFFER, cubeVertices_vbo);
  glBufferData(GL_ARRAY_BUFFER, sizeof(cube_vertices), cube_vertices, GL_STATIC_DRAW);

  glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
  glEnableVertexAttribArray(0);
```

OpenGL Application

- All starts with the main function

2- Initialize your rendering variables (ones that do not change over the course of your application)



e- Initialize your VBOs to describe your object vertices, starting with your vertex colours array.

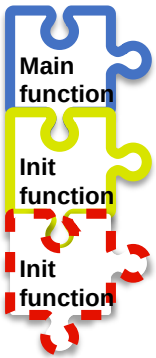
```
float cube_colors[] = {
    // front colors
    1.0, 0.0, 0.0,
    0.0, 1.0, 0.0,
    0.0, 0.0, 1.0,
    1.0, 1.0, 1.0,
    // back colors
    1.0, 0.0, 0.0,
    0.0, 1.0, 0.0,
    0.0, 0.0, 1.0,
    1.0, 1.0, 1.0,
  };
  GLuint cube_colors_vbo = 0;
  glGenBuffers(1, &cube_colors_vbo);
  glBindBuffer(GL_ARRAY_BUFFER, cube_colors_vbo);
  glBufferData(GL_ARRAY_BUFFER, sizeof(cube_colors), cube_colors, GL_STATIC_DRAW);

  glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, 0);
  glEnableVertexAttribArray(1);
```

OpenGL Application

- All starts with the main function

2- Initialize your rendering variables (ones that do not change over the course of your application)

e2- Initialize your VBOs, then any other arrays that would describe your vertices.

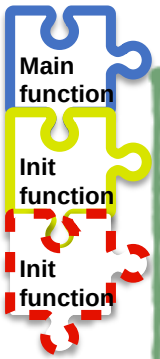


Maybe normals?
Textures?
Etc.

OpenGL Application

- All starts with the main function

2- Initialize your rendering variables (ones that do not change over the course of your application)



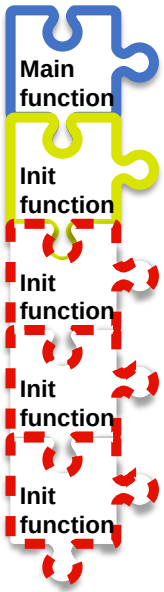
f- Initialize your IBO.

```
GLushort cube_index_array[] = {
    // front
    0, 1, 2,
    2, 3, 0,
    // top
    1, 5, 6,
    6, 2, 1,
    // back
    7, 6, 5,
    5, 4, 7,
    // bottom
    4, 0, 3,
    3, 7, 4,
    // left
    4, 5, 1,
    1, 0, 4,
    // right
    3, 2, 6,
    6, 7, 3,
};
GLuint ibo_cube_elements;
glGenBuffers(1, &ibo_cube_elements);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo_cube_elements);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(cube_index_array), cube_index_array, GL_STATIC_DRAW);
```

OpenGL Application

- All starts with the main function

Repeat steps 2-C to 2-F for each object in your scene.



Cube:
Create cubeVAO then bind it
Create cubeVertexPositionsArray
Create cubeVertexPosVBO and attach cubeVertexPositionsArray to it then bind it
Create cubeVertexColorArray
Create cubeVertexColorVBO and attach cubeVertexColorArray to it then bind it
Create cubeIndexArray
Create cubeIBO and attach cubeIndexArray to it then bind it

Pyramid:
Create pyramidVAO then bind it
Create pyramidVertexPositionsArray
Create pyramidVertexPosVBO and attach pyramidVertexPositionsArray to it then bind it
Create pyramidVertexColorArray
Create pyramidVertexColorVBO and attach pyramidVertexColorArray to it then bind it
Create pyramidIndexArray
Create pyramidIBO and attach pyramidIndexArray to it then bind it

Etc…

OpenGL Application

- All starts with the main function

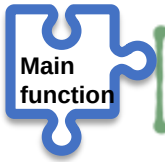
3- Bind OpenGL display function with your display steps.

Best practice is to have a separate function for drawing. Usually called Draw or Display.

OpenGL Application

- All starts with the main function

3- Bind OpenGL display function with your display steps.



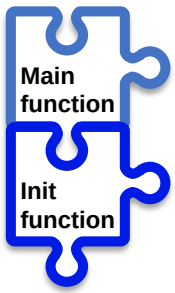

```
glutDisplayFunc(display);
```

OpenGL Application

- All starts with the main function

3- Bind OpenGL display function with your display steps.

a– Clear the buffers.



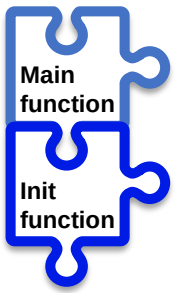

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glClearColor(0.0f, 0.5f, 0.9f, 0.0f);
```

OpenGL Application

- All starts with the main function

3- Bind OpenGL display function with your display steps.

Steps B to D are to be repeated for each instance of an object that you want to render.

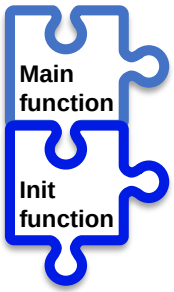


OpenGL Application

- All starts with the main function

3- Bind OpenGL display function with your display steps.

b- Bind the VAO for the object you want to create an instance for.




```
glBindVertexArray(boxVAO);
```

OpenGL Application

- All starts with the main function

3- Bind OpenGL display function with your display steps.

c- Transform your instance.

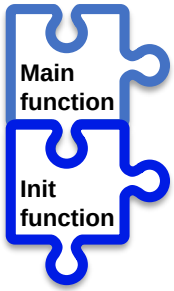
Best practice is to have a separate function that performs the appropriate transformations. Usually called Transform.

OpenGL Application

- All starts with the main function

3- Bind OpenGL display function with your display steps.

c- Transform your instance.



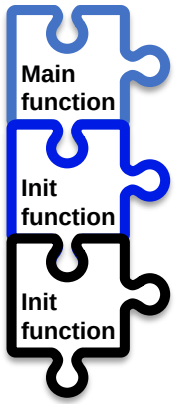

```
// Scale x5 in X, x0.1 in Y, and x5 in Z.
// Rotate 30 degrees around the Z axis.
// Translate -2 units in the Y axis
transformObject(glm::vec3(5.0f,0.1f,5.0f),
                                glm::vec3(0.0f,0.0f,1.0f), 30, glm::vec3(0.0f,-2.0f,0.0f));
```

OpenGL Application

- All starts with the main function

3- Bind OpenGL display function with your display steps.

c- Transform your instance.
c1- Create an empty matrix.
c2- Translate.
c3- Rotate.
c4- Scale.
c5- Multiply your matrices.
c6- Update the MVP matrix on the shader.



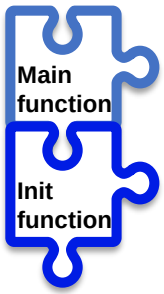

```
glm::mat4 Model;
Model = glm::mat4(1.0f);
Model = glm::translate(Model, translation);
Model = glm::rotate(Model, glm::radians(rotationAngle), rotationAxis);
Model = glm::scale(Model, scale);

MVP = Projection * View * Model;
glUniformMatrix4fv(MatrixID, 1, GL_FALSE, &MVP[0][0]);
```

OpenGL Application

- All starts with the main function

3- Bind OpenGL display function with your display steps.

d- Draw your instance.



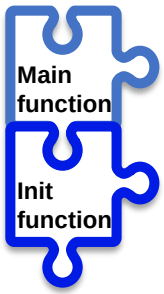

```
glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_SHORT, 0);
```

OpenGL Application

- All starts with the main function

3- Bind OpenGL display function with your display steps.

e- Repeat steps 3-B to 3-D for each instance in your scene.

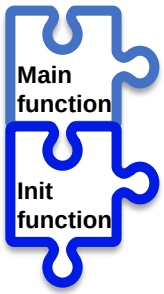


OpenGL Application

- All starts with the main function

3- Bind OpenGL display function with your display steps.

f- Swap the buffers in order to actually draw on the screen.




```
glutSwapBuffers();
```

OpenGL Application

- All starts with the main function

4- Start the glut main loop.


```
glutMainLoop();
```

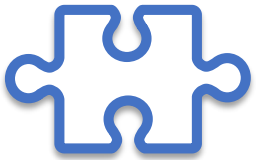
Vertex Shader Components

The vertex shader receives a single vertex.

In order to draw a full shape, OpenGL starts multiple instances of the shader for each vertex in parallel.

Vertex Shader Components

1- Define the OpenGL version



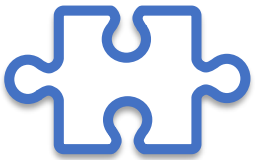
```
#version 410 core
```

Vertex Shader Components

2- Define the input resources.

Note: This has to be identical to how your vertex is defined.

Ex: If a cube is defined by vertices and colors, you will need two input variables to map to these properties.

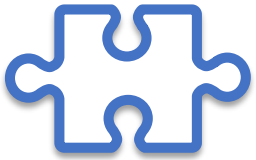


```
layout(location = 0) in vec3 vertex_position;
layout(location = 1) in vec3 vertex_colour;
```

Vertex Shader Components

3- Define the output resources.

Note: This has to be identical to how your next stage in your pipeline expects an input.
In our case for now, the next stage is the fragment shader that expects a color as input.

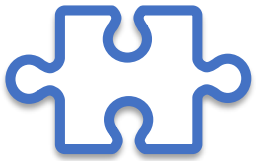


```
out vec3 myColor;
```

Vertex Shader Components

3- Define the uniform variables.

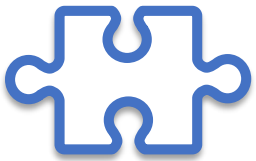
Note: These are variables that remain constant for all vertices of the object.



```
// Values that stay constant for the whole mesh.
uniform highp mat4 MVP;
```

Vertex Shader Components

4- Perform per vertex functions.



```
void main()
{
   myColor = vertex_colour;
      gl_Position = MVP * vec4(vertex_position,1.0f);
}
```

Fragment Shader Components

The fragment shader receives a single color, and outputs a fragment to be rendered.

Fragment Shader Components

1- Define the OpenGL version

```
#version 410 core
```

Fragment Shader Components

2- Define the input resources.

Note: This has to be identical to the output from the previous stage in the pipeline.
In our case for now, the input is coming from the vertex shader.

```
in vec3 myColor;
```

Fragment Shader Components

3- Define the output resources.

```
out vec4 frag_colour;
```

Fragment Shader Components

3- Perform per fragment functions.

```
void main() {
   frag_colour = vec4(myColor, 1.0);
}
```

Week 6

Lab Activities

Week 6 Lab

- For the lab, see Hooman's material (with video)
- OpenGL examples covered:
  - Lots of 3D mathematical models/shapes

Week 6

End