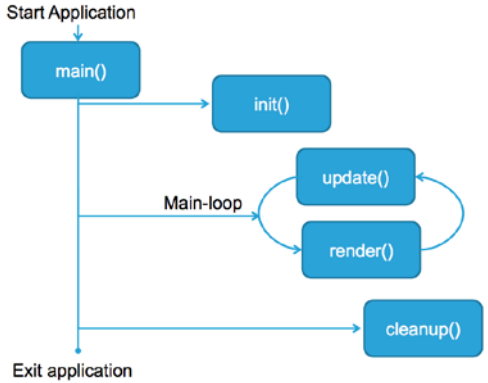
CS380: Introduction to Computer Graphics
Introduction to OpenGLSL

Min H. Kim
KAIST School of Computing



## OpenGL Program Skeleton

- Typical OpenGL-Application:






## OpenGL Program Skeleton

- main():
  - Program-Entry
  - Create window
  - Call init()
  - Start main window-loop Call cleanup()
  - Exit application
- init():
  - Initialize libraries, load config-files, ...
  - Allocate resources, preprocessing, ...



## OpenGL Program Skeleton

- update():
  - Handle user-input, update game-logic, ...
- render():
  - Do actual rendering of graphics here!
  - Note: Calling render() twice without calling update() in between should result in the same rendered image!
- cleanup():
  - Free all resources

## OpenGL Program Skeleton

- Example init()-function:

```
void init() {
        // Create and initialize a window with depth-buffer and
        // double- buffering. See your window-managers documentation.

        // enable the depth-buffer in OpenGL
        glEnable(GL_DEPTH_TEST);

        // enable back-face culling in OpenGL
        glEnable(GL_CULL_FACE);

        // define a clear color
        glClearColor(0.0f, 0.0f, 0.0f, 0.0f);

        // set the OpenGL-viewport
        glViewPort(0, 0, windowWidth, windowHeight);

        // Do other useful things
}
```



## OpenGL Program Skeleton

- The geometry of a 3D-object is stored in an array of vertices called Vertex-Array.
- Each vertex can have so called Attributes, like a Normal Vector and Texture-Coordinates.
- OpenGL also treats vertices as attributes!
- To render geometry in OpenGL, vertex-(attribute)-arrays are passed to OpenGL and then rendered.



## OpenGL Program Skeleton

- To do so:
  - Query the attribute-location in the shader:

```
GLint vertexLocation = glGetAttribLocation(
        myShaderProgram, "in_Position");
```

  - Enable an array for the vertex-attribute:

```
glEnableVertexAttribArray(vertexLocation);
```

  - Then tell OpenGL which data to use:

```
glVertexAttribPointer(vertexLocation, 3, GL_FLOAT,
GL_FALSE, 0, myVertexArray);
```



## OpenGL Program Skeleton

- Draw ("render") the arrays:

```
glDrawArrays(GL_TRIANGLES, 0, 3); // this does the
actual draw
```

- Finally disable the attribute-array:

```
glDisableVertexAttribArray(vertexLocation);
```

## OpenGL Program Skeleton

- Example render()-function:

```
// triangle data
static GLfloat vertices[] = { -0.5, -0.333333, 0, // x1, y1, z1
                              +0.5, -0.333333, 0, // x2, y2, z2
                              +0.0, +0.666666, 0}; // x3, y3, z3
...
void render() {
    // clear the color-buffer and the depth-buffer
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // activate a shader program
    glUseProgram(myShaderProgram);

    // Find the attributes
    GLint vertexLocation = glGetAttribLocation(
        myShaderProgram, "in_Position");
```



## OpenGL Program Skeleton

```
    // enable vertex attribute array for this attribute
    glEnableVertexAttribArray(vertexLocation);

    // set attribute pointer
    glVertexAttribPointer(vertexLocation, 3, GL_FLOAT,
        GL_FALSE, 0, vertices);

    // Draw ("render") the triangle
    glDrawArrays(GL_TRIANGLES, 0, 3);

    // Done with rendering. Disable vertex attribute array
    glDisableVertexAttribArray(vertexLocation);

    // disable shader program
    glUseProgram(0);

    // Swap buffers
}
```



## OpenGL-Object life-cycle

- In OpenGL, all objects, like buffers and textures, are somehow treated the same way.
- On object creation and initialization:
  - First, create a handle to the object (in OpenGL often called a name). Do this ONCE for each object.
  - Then, bind the object to make it current.
  - Pass data to OpenGL. As long as the data does not change, you only have to do this ONCE.
  - Unbind the object if not used.



## OpenGL-Object life-cycle

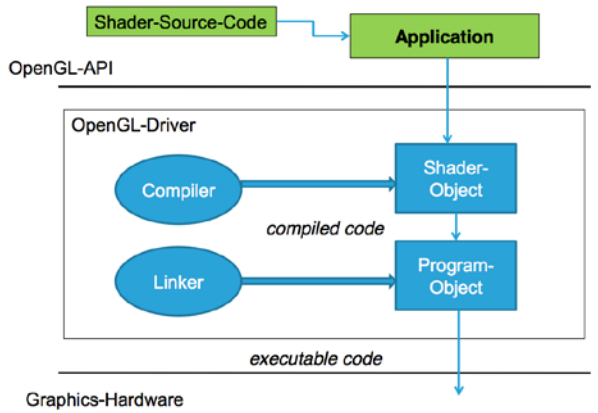
- On rendering, or whenever the object is used:
  - Bind it to make it current.
  - Use it.
  - Unbind it.
- Finally, when object is not needed anymore:
  - Delete object.
  - Note that in some cases you manually have to delete attached resources!
- NOTE: OpenGL-objects are NOT objects in an OOP-sense!

## What shaders are

- Small C-like programs executed on the graphics-hardware
- Replace fixed function pipeline with shaders
- Shader-Types
  - Vertex Shader (VS): per vertex operations
  - Geometry Shader (GS): per primitive operations
  - Fragment Shader (FS): per fragment (pixel) operations, so-called Pixel Shader
- Used e.g. for transformations and lighting



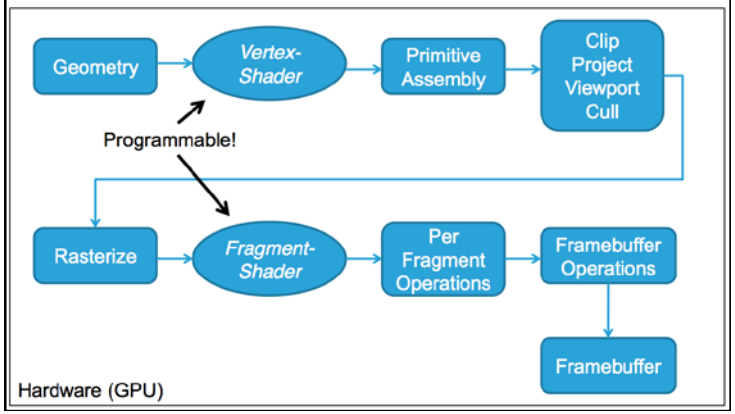
## Shader-Execution model






## Rendering-Pipeline

- OpenGL 3.x Rendering-Pipeline:




## Rendering-Pipeline

- Remember:
  - The Vertex-Shader is executed ONCE per each vertex!
  - The Fragment-Shader is executed ONCE per rasterized fragment (~ a pixel)!
- A Shader-Program consists of both,
  - One VS
  - One FS

## Setting up shaders and programs

- Compile shaders:

```
char* shaderSource; // contains shadersource
int shaderHandle = glCreateShader(GL_SHADER_TYPE);

// shader-types: vertex || geometry || fragment
glShaderSource(shaderHandle, 1, shaderSource, NULL);
glCompileShader(shaderHandle);
```

- Create program and attach shaders to it:

```
int programHandle = glCreateProgram();
glAttachShader(programHandle, shaderHandle);
// do this for vertex AND fragment-shader (AND geometry if needed)!
```

- Finally link program:

```
glLinkProgram(programHandle);
```



## Enabling shaders

- Enable a GLSL program:

```
glUseProgram(programHandle); // shader-program now active
```

  - The active shader-program will be used until glUseProgram() is called again with another program-handle.
  - Call of glUseProgram(0) sets no program active (undefined state!).



## Basic shader layout

- Shader-Programs must have a main()-method
- Vertex-Shader outputs to at least gl_Position
- Fragment-Shader to custom defined output

```
//preprocessor directives like:
#version 150

//variable declarations

void main() {
    //do something and write into output variables
}
```



## Shader Parameter

- Shader variable examples:

```
uniform mat4 projMatrix; // uniform input
in vec4 vertex; // attribut-input
out vec3 fragColor; // shader output
```

- Three types:
  - uniform: does not change per primitive; read-only in shaders
  - in: VS: input changes per vertex, read-only; FS: interpolated input; read-only
  - out: shader-output; VS to FS; FS output.

## Uniform Parameter

- Set uniform parameters in an application:

```
// first get location
projMtxLoc = glGetUniformLocation(programHandle,
  "projMatrix");

// then set current value
glUniformMatrix4fv(projMtxLoc, 1, GL_FALSE,
currentProjectionMatrix);
```

  - First get the „location“ of the uniform-variable
  - Then set the current value
  - Can pass values to vertex- and fragment-shader



## Attribute Parameter

- A vertex can have attributes like a normal-vector or texture-coordinates
- OpenGL also treats the vertex itself as an attribute
- We want to access our current vertex within our vertex-shader (as we used to do with gl_Vertex in former GLSL-versions):
- Therefore, we declare in our vertex-shader:

```
in vec4 vertex; // vertex attribut
```



## Attribute Parameter

- Now, there are two ways to pass data to this shader attribute-variable, depending on:
  - if you just have an array of vertices (Vertex Array),
  - or an VBO (Vertex Buffer Object).
- To do so: Query shader-variable location
  - Enable vertex-attribute array
  - Set pointer to array
  - Draw and disable array



## Attribute Parameter: Vertex-Array

- For a Vertex-Array, pass data like this:

```
// first get the attribute-location
vertexLocation = glGetAttribLocation(programHandle, "vertex");

// enable an array for the attribute
glEnableVertexAttribArray(vertexLocation);

// set attribute pointer
glVertexAttribPointer(vertexLocation, 3, GL_FLOAT, GL_FALSE, 0,
myVertexArray);

// Draw ("render") the triangle
glDrawArrays(GL_TRIANGLES, 0, 3);

// Done with rendering. Disable vertex attribute array
glDisableVertexAttribArray(vertexLocation);
```

## Attribute Parameter: VBO

- Setting attribute parameters with VBO:

```
// first get location
vertexLocation = glGetAttribLocation(programHandle, "vertex");

// activate desired VBO
glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);

// set attribute-pointer
glVertexAttribPointer(vertexLocation, 4, GL_FLOAT, GL_FALSE,
0, 0);

// finally enable attribute-array
glEnableVertexAttribArray(vertexLocation); ...
```



## Fragment Output

- Since GLSL 1.3, gl_FragColor is deprecated.
- Therefore, need to define output on our own.
- Declare output variable in FS:

```
out vec4 fragColor; // fragment color output
```

- In the application, before linking the shader-program with glLinkProgram(), bind the FS-output:

```
glBindFragDataLocation(programHandle, 0,
"fragColor");
```

- Finally assign a value to fragColor in the FS.



## Example usage

- An application using shaders could basically look like this:

```
//Load shader and initialize parameter-handles

//Do some useful stuff like binding texture, activate
//texture-units, calculate and update matrices, etc.

//glUseProgram(programHandle);

//Set shader-parameters
//Draw geometry

//glUseProgram(anotherProgramHandle);
...
```



Appendix A

## HELLO WORLD 2D

## Assignment1

- **HW1 Deadline: 2015.3.12(Thur.) 23:55**
- Go to the course website and download the files: hw2d.pdf, hw2d.zip in Week1
- Solve the three tasks (described in the pdf) and submit your codes with some screenshots to the TA by email.
- **Yeong Beum Lee (yblee@vclab.kaist.ac.kr)**



## Quiz1

- **15/3/11 (Wed.) Quiz1 about what we learned about OpenGL**



## Code (hw2d.zip)

- we use c++
  – see 'C++ primer'.
- now we look at the code from the textbook
- Core codes:
  – asst1.cpp
  – glsupport.h/.cpp
  – asst1-gl3.fshader
  – asst1-gl3.vshader
- Auxiliary codes:
  – ppm.h/.cpp



## Code (hw2d.zip)

## asst1.cpp

- Set g_Gl2Compatible = true to use GLSL 1.0 and g_Gl2Compatible = false to use GLSL 1.3.
- on Mac OS X currently there is no way of using OpenGL 3.x with GLSL 1.3 when GLUT is used.
- If g_Gl2Compatible=true, shaders with -gl2 suffix will be loaded.
- If g_Gl2Compatible=false, shaders with -gl3 suffix will be loaded.
- To complete the assignment you only need to edit the shader files that get loaded



## asst1.cpp

- How to check the version you have

```
cout << "GL ver: " << glGetString(GL_VERSION) <<"\n";
cout << "GLEW ver: " << glewGetString(GLEW_VERSION) <<"\n";
cout << "GLSL ver: "<<
glGetString(GL_SHADING_LANGUAGE_VERSION) <<"\n";
```

- Outputs

```
GL ver: 2.1 NVIDIA-8.12.47 310.40.00.05f01
GLEW ver: 1.10.0
GLSL ver: 1.20
```



## asst1.cpp

```
int main(int argc, char **argv) {
 try {
      initGlutState(argc,argv);
      glewInit(); // load the OpenGL extensions

      initGLState();
      initShaders();
      initGeometry();
      initTextures();

      glutMainLoop();
      return 0;
 }
 catch (const runtime_error& e) {
      cout << "Exception caught: " << e.what() << endl;
      return -1;
 }
}
```