OpenGL & GLUT

527970
Fall 2020
9/10/2020

Kyoung Shin Park
Computer Engineering
Dankook University

OpenGL & GLUT & GLEW

- OpenGL
  - http://www.opengl.org/
  - http://www.sgi.com/software/opengl
  - OpenGL is included in Window95 or later
  - ftp://ftp.microsoft.com/softlib/MSLFILES/opengl95.exe
- freeglut for Win32
  - http://freeglut.sourceforge.net/
  - Prepackaged Release (freeglut 3.0.0. MSVC)
  - **Download freeglut-MSVC-3.0.0-2.mp.zip**
- glew for Win32
  - http://glew.sourceforge.net/
  - **Download glew-2.1.0-win32.zip**
- glm
  - https://github.com/g-truc/glm/releases
  - **Download glm-0.9.9.6.zip**

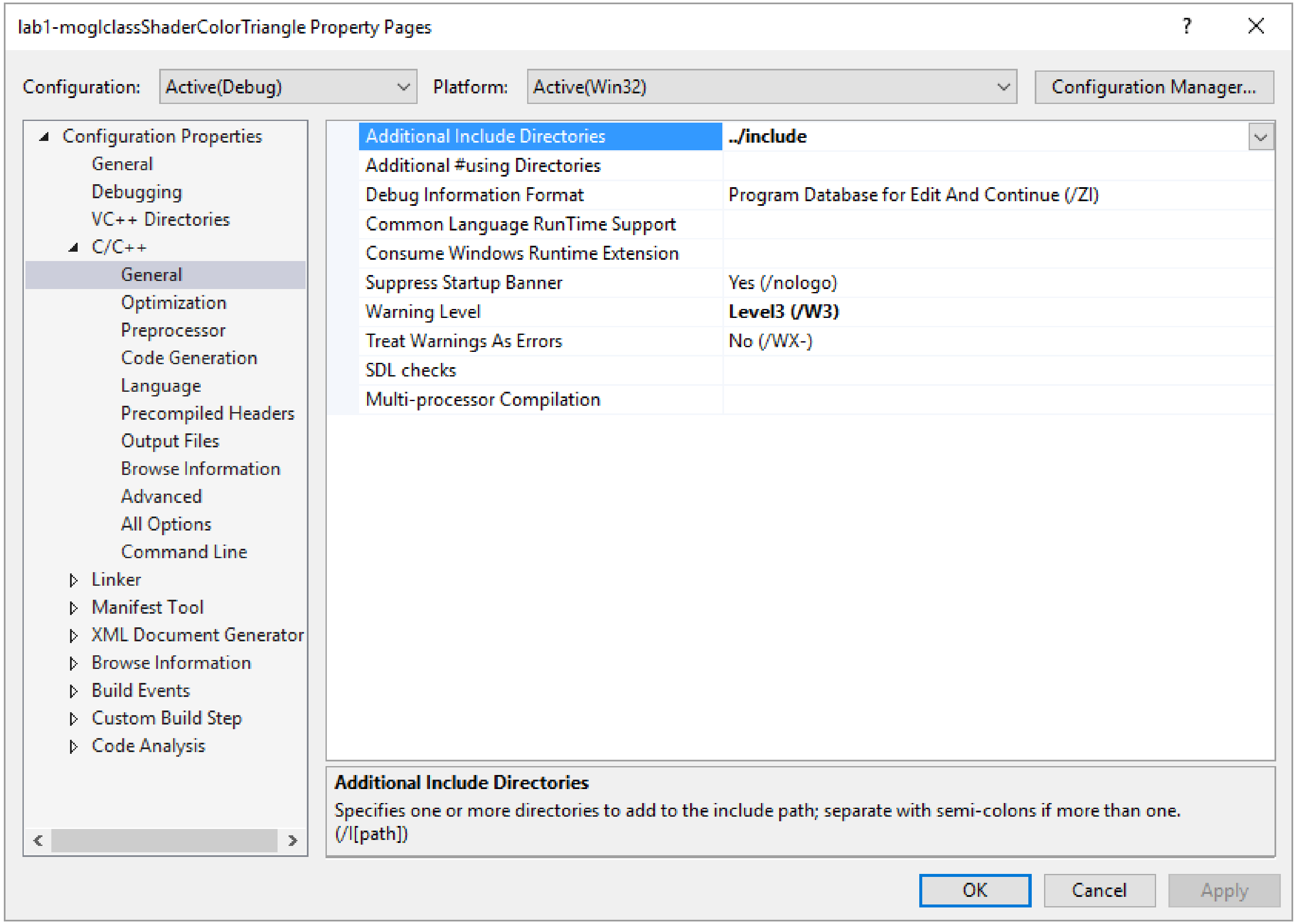
Installing OpenGL & GLUT & GLEW

- Libraries
  - **freeglut.lib**
  - **glew32.lib**
- Include
  - **freeglut.h freeglut_ext.h freeglut_std.h glut.h**
  - **eglew.h glew.h glxew.h wglew.h**
  - **glm (all glm header files)**
- Dynamically-linked libraries in **C:₩Windows₩system32**
  - **freeglut.dll glew32.dll**
  - Install freeglut.dll glew32.dll in **C:₩Windows₩SysWOW64** (64-bit OS)

Compiling OpenGL Programs

- Execute Visual Studio 2019 VC++
- Create a new project
  - In menu, File->New->Projects
  - Select a new empty project (Win32 Console)
  - Give a project name
- Specify include directory in compiler
- Specify library directory & files in linker
  - In menu, Project->Settings->link->Object/library modules (Project->Properties->Linker->Input->Additional dependencies)
  - **glew32.lib freeglut.lib**
- Create a new file in the project
  - In menu, Project->Add to Project-> Files
  - Give a file name
- Build (F7) and Execute (F5)

Compiling OpenGL Programs



lab1-moglclassShaderColorTriangle Property Pages

Configuration: Active(Debug) Platform: Active(Win32) Configuration Manager...

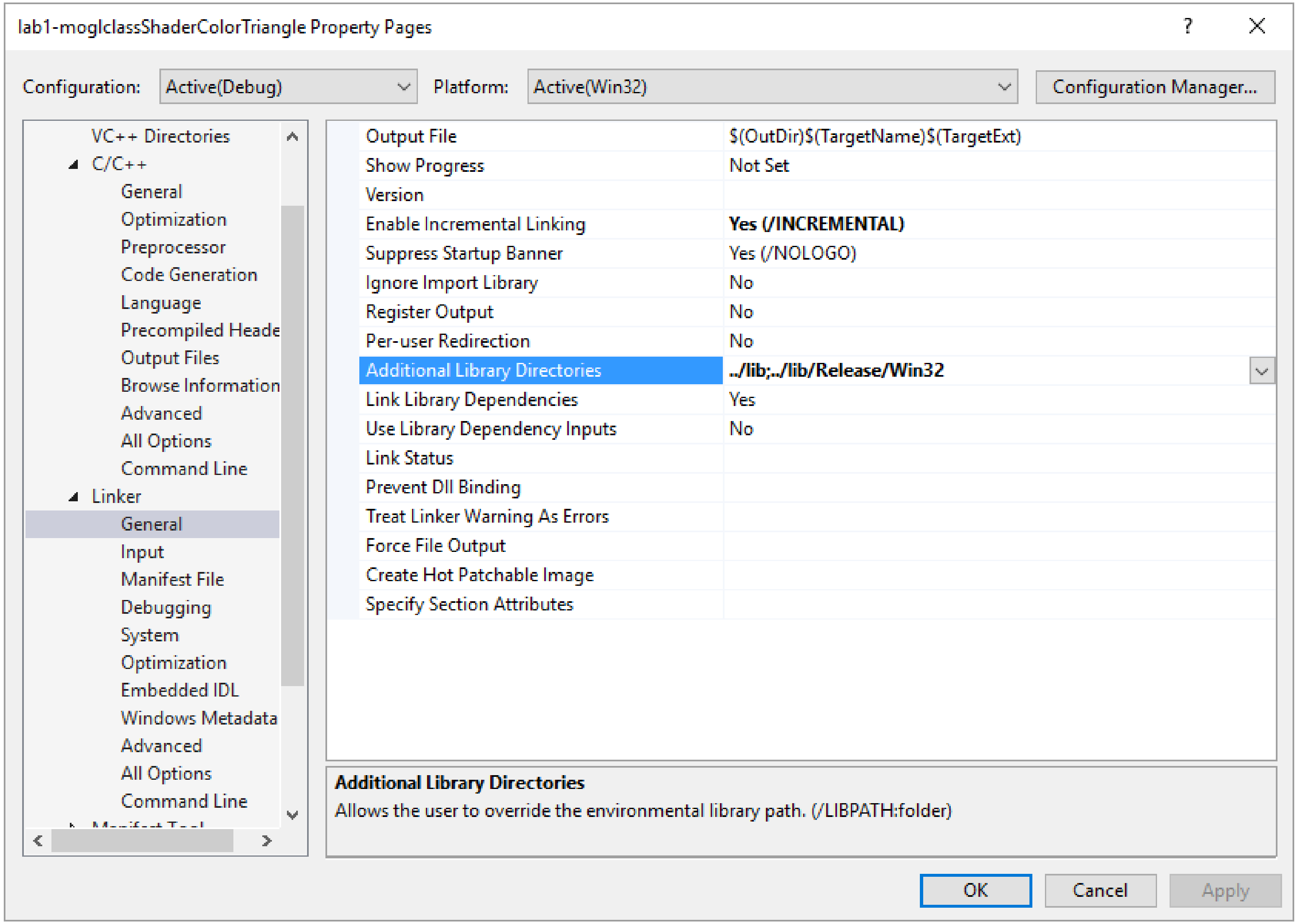
- Configuration Properties
  - General
  - Debugging
  - VC++ Directories
  - C/C++
    - General
    - Optimization
    - Preprocessor
    - Code Generation
    - Language
    - Precompiled Headers
    - Output Files
    - Browse Information
    - Advanced
    - All Options
    - Command Line
  - Linker
  - Manifest Tool
  - XML Document Generator
  - Browse Information
  - Build Events
  - Custom Build Step
  - Code Analysis

Additional Include Directories	**../include**
Additional #using Directories	
Debug Information Format	Program Database for Edit And Continue (/ZI)
Common Language RunTime Support	
Consume Windows Runtime Extension	
Suppress Startup Banner	Yes (/nologo)
Warning Level	**Level3 (/W3)**
Treat Warnings As Errors	No (/WX-)
SDL checks	
Multi-processor Compilation	

Additional Include Directories

Specifies one or more directories to add to the include path; separate with semi-colons if more than one. (/I[path])

Compiling OpenGL Programs



lab1-moglclassShaderColorTriangle Property Pages

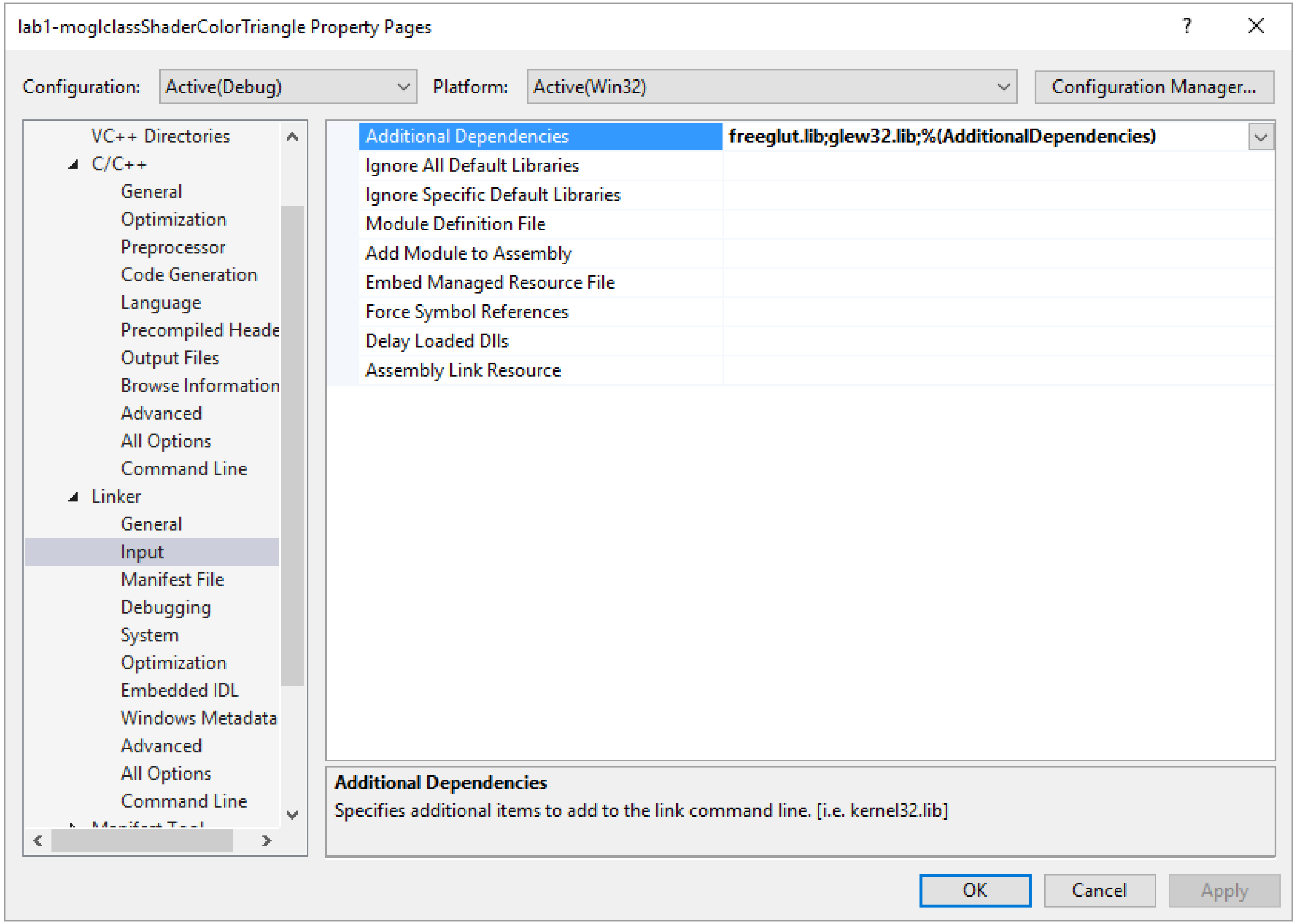
Configuration: Active(Debug)   Platform: Active(Win32)

Output File	$(OutDir)$(TargetName)$(TargetExt)
Show Progress	Not Set
Version	
Enable Incremental Linking	**Yes (/INCREMENTAL)**
Suppress Startup Banner	Yes (/NOLOGO)
Ignore Import Library	No
Register Output	No
Per-user Redirection	No
Additional Library Directories	**../lib;../lib/Release/Win32**
Link Library Dependencies	Yes
Use Library Dependency Inputs	No
Link Status	
Prevent Dll Binding	
Treat Linker Warning As Errors	
Force File Output	
Create Hot Patchable Image	
Specify Section Attributes	

Additional Library Directories

Allows the user to override the environmental library path. (/LIBPATH:folder)

Compiling OpenGL Programs



lab1-moglclassShaderColorTriangle Property Pages

Configuration: Active(Debug)

Platform: Active(Win32)

Additional Dependencies	freeglut.lib;glew32.lib;%(AdditionalDependencies)
Ignore All Default Libraries	
Ignore Specific Default Libraries	
Module Definition File	
Add Module to Assembly	
Embed Managed Resource File	
Force Symbol References	
Delay Loaded Dlls	
Assembly Link Resource	

Additional Dependencies

Specifies additional items to add to the link command line. [i.e. kernel32.lib]

Windows System

- Window system
  - Microsoft Windows
  - X Window systems
- Both Windows and OpenGL are raster-graphics systems
- To do OpenGL programming
  - Window programming based on the raster system
  - Connect to the OpenGL system, in an abstract raster system, in the context of Windows programming
  - 3D graphics programming using functions provided by OpenGL
  - Convert the desired OpenGL operation into a form that can be efficiently understood by the Windows system that is actually controlling the hardware

OpenGL

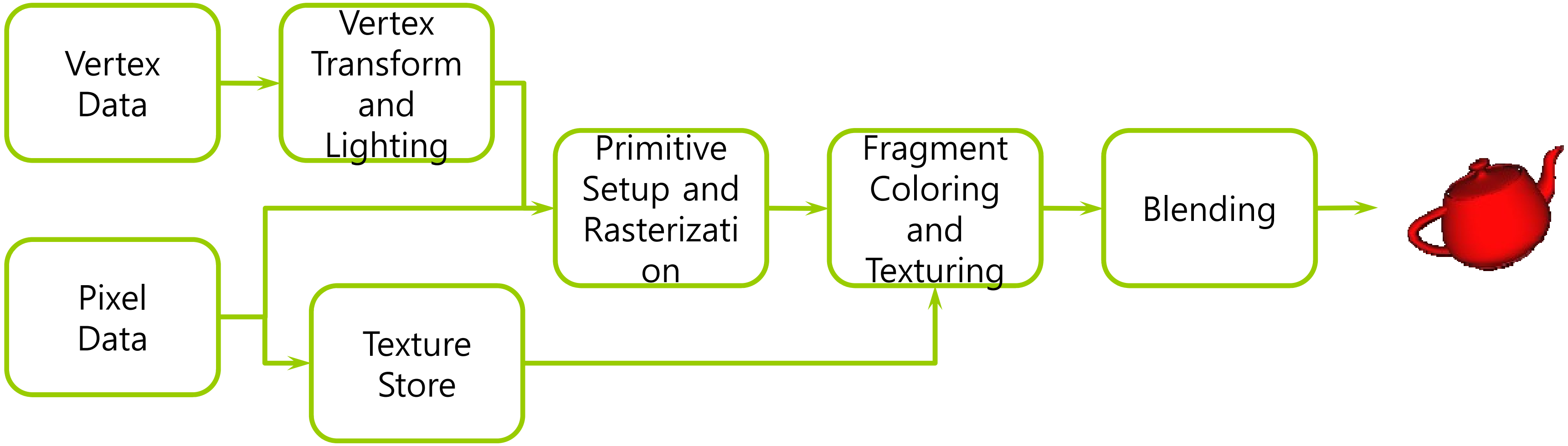
- 3D graphics library API developed by Silicon Graphics Incorporated (SGI) (1992)
- 2D graphics is treated as a special case of 3D (the z-axis is 0)
- OpenGL graphics functions are programing language independent.
  - OpenGL can be used with C/C++, Java, Fortran, Python, etc.
- OpenGL is hardware neutral (Window system or OS independent API)
  - No I/O library
  - No specific model loading mechanism
  - No hardware specific functions (but available as extensions)

OpenGL Evolution

- Initially led by Architectural Review Board (ARB)
  - SGI, Microsoft, Nvidia, HP, 3DLabs, IBM, etc
  - Currently, the Kronos Group
- OpenGL 3.1
  - Entirely based on shader. That is, all OpenGL applications must use vertex and fragment shader.
  - Most 2.5 functions are deprecated (i.e., not recommended to use as they will be missing)
- OpenGL ES (for embedded systems)
  - Ver 1.0 simplifies OpenGL 2.1. Ver 2.0 simplifies OpenGL 3.1.
- WebGL
  - Implement OpenGL ES 2.0 in Javascript
- OpenGL 4.1 & 4.2
  - Add geometry shader and tesselator

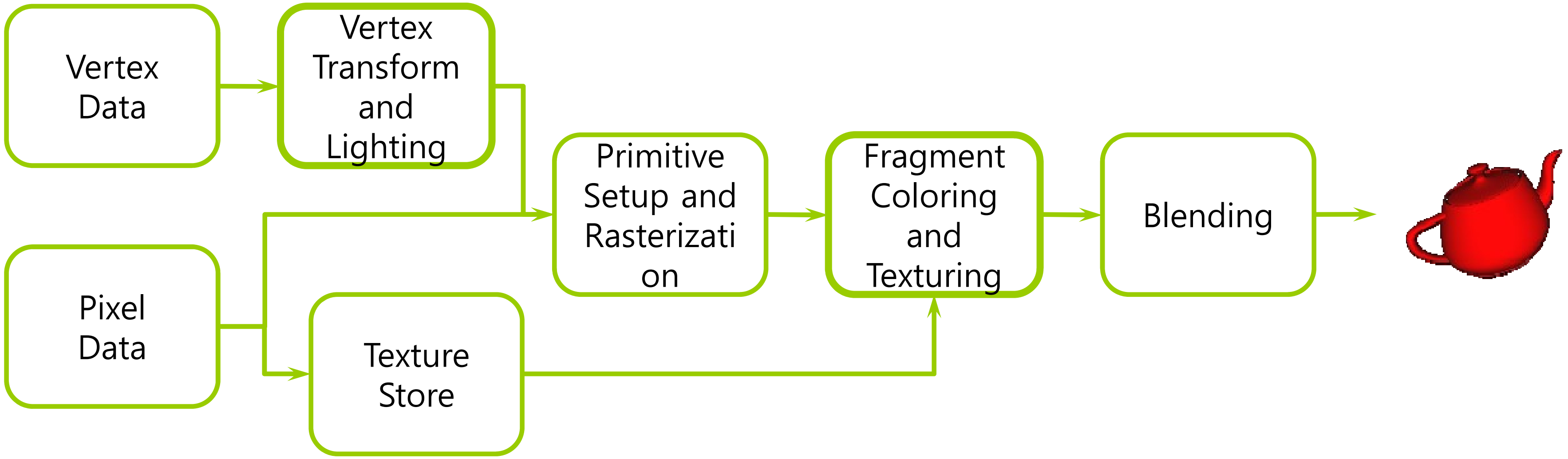
OpenGL 1.0

- The initial version of OpenGL was announced in July of 1994.
- That version of OpenGL implemented what's called a fixed-function pipeline, and is used from OpenGL 1.1 to OpenGL 2.0 (2004/09)


Vertex Data
Vertex Transform and Lighting
Pixel Data
Primitive Setup and Rasterizati on
Texture Store
Fragment Coloring and Texturing
Blending

OpenGL 2.0 Programmable Pipeline

- OpenGL 2.0 officially added programmable shader (GPU).
  - Vertex Shader
  - Fragment Shader
- OpenGL 2.0 also fully supported OpenGL 1.x pipeline, allowing both version: a fixed-function pipeline and programmable pipeline.


Vertex Data
Vertex Transform and Lighting
Pixel Data
Primitive Setup and Rasterizati on
Texture Store
Fragment Coloring and Texturing
Blending

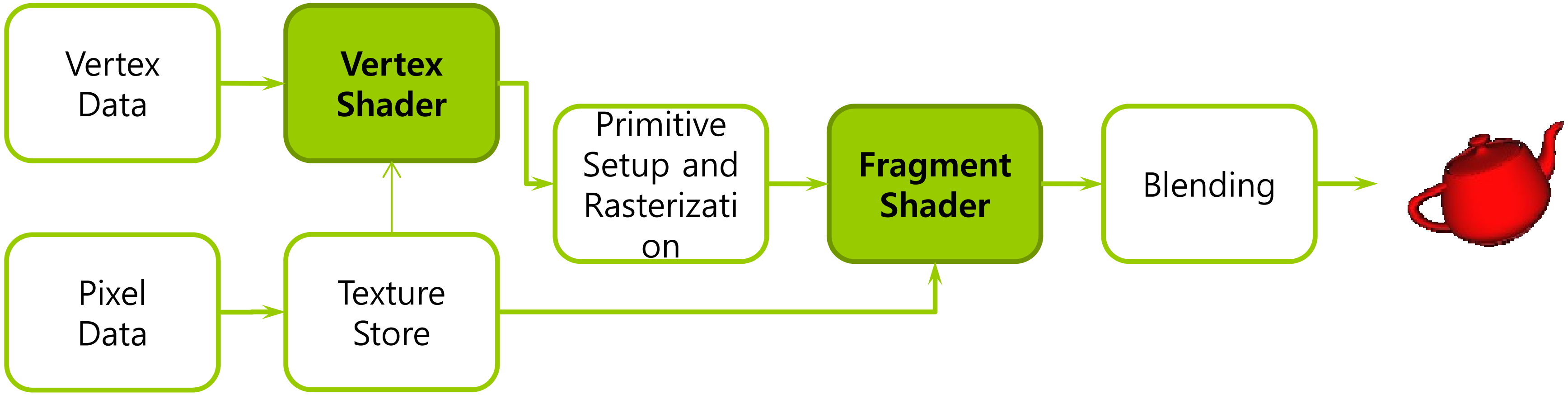
OpenGL 3.0

- OpenGL 3.0 introduces the *deprecation model*
  - Prior to OpenGL 3.0, features were added to support backward compatibility, but OpenGL 3.0 no longer supports backward compatibility, and new features are provided.
- Graphics pipeline is the same until OpenGL 3.1(2009/03/24)
- Forward compatible contexts is supported from OpenGL 3.1

Context Type	Description
Full	Includes all features (including those marked depr ecated) available in the current version of OpenGL
Forward Compatible	Includes all non-deprecated features (i.e., creates a context that would be similar to the next versio n of OpenGL)

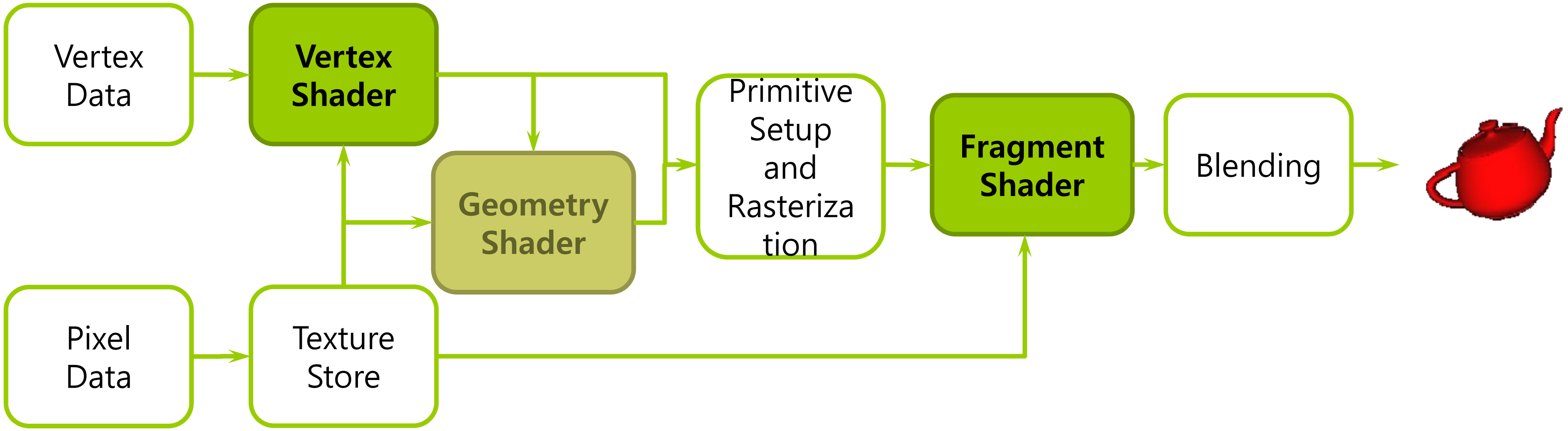
Exclusively Programmable Pipeline

- OpenGL 3.1 removed the fixed-function pipeline
  - The program should only use shaders
- Additionally, all data must be sent to the GPU
  - All vertex data is sent using buffer objects



More Programmability

- OpenGL 3.2 (2009/09/03) added the geometry shader



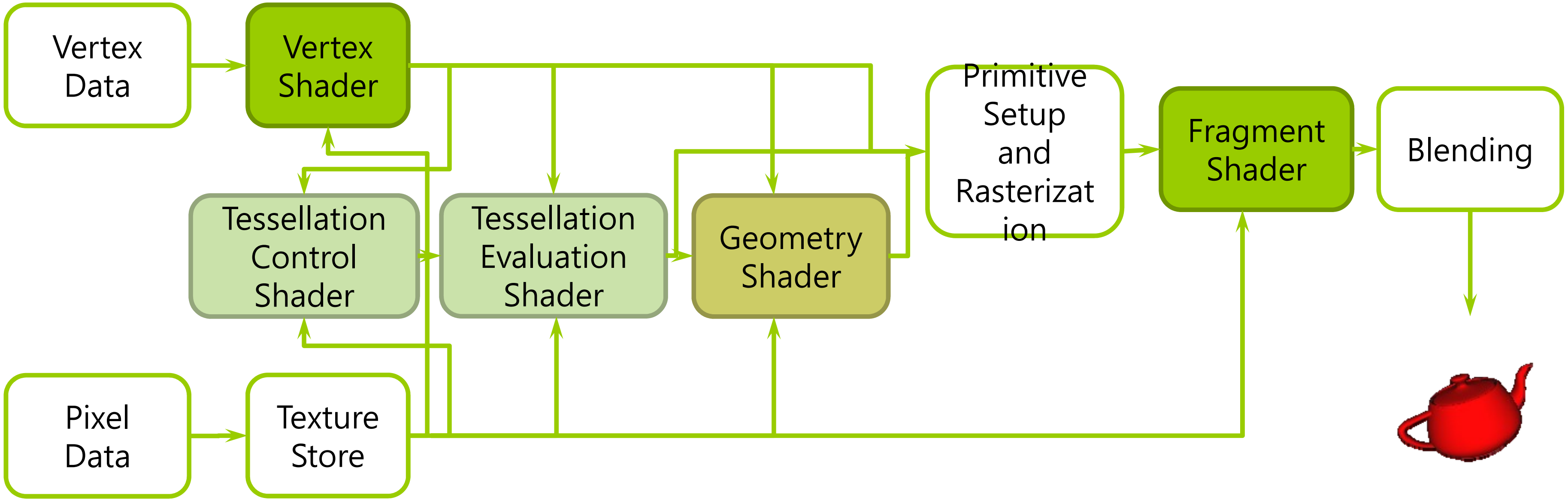
More Evolution – Context Profiles

- OpenGL 3.2 introduces the context profile
  - Profile is provided to make it easy to select features in the application program
  - Currently there are two profiles: core & compatible

Context Type	Profile	Description
Full	core	All features of the current release
	compatible	All features ever in OpenGL
Forward Compatible	core	All non-deprecated features
	compatible	Not supported

OpenGL 4.1 or later

- OpenGL 4.1 (2010/07/25) added the tessellation-control shader and the tessellation-evaluation shader
- The most recent version is OpenGL 4.7 (2017)


Vertex Data
Vertex Shader
Primitive Setup and Rasterizat ion
Fragment Shader
Blending
Tessellation Control Shader
Tessellation Evaluation Shader
Geometry Shader
Pixel Data
Texture Store

OpenGL Libraries

- OpenGL core library
  - OpenGL32 in Windows system
  - libGL.a in Unix/Linux system
- OpenGL Utility Library (GLU)
  - Additional support for OpenGL core library
- OpenGL Windows Toolkit
  - Different platforms have different ways to integrate OpenGL with their windowing environment
  - X Window System (GLX)
  - Apple (AGL)
  - Windows (WGL)
  - IBM OS/2 (PGL)
  - Cross-platform (GLUT – OpenGL Utility Toolkit)

Freeglut & GLEW

- Freeglut is an improved version of GLUT
  - Early GLUT is old and no longer being updated.
  - Freeglut provides GLUT and additional features.
- GLEW (OpenGL Extension Wrangler Library)
  - Hardware-related OpenGL is difficult to use. You need to get a function pointer and pay attention to the extension.
  - GLEW allows convenient use of OpenGL extension, such as GLSL
  - In OpenGL application, simply initialize GLEW by calling #include <glew.h> and glewInit()

OpenGL & GLSL

- Shader based OpenGL
  - Most state variables, attributes and related pre 3.1 OpenGL functions have been deprecated.
  - Everything is done through shaders.
- GLSL (OpenGL Shading Language)
  - Similar to C language and also similar to Nvidia's Cg and Microsoft's HLSL
  - GLSL code is sent to the shader and rendering is done through the GPU.

OpenGL/GLUT Program

- OpenGL example that simply opens a window

```
#include <GL/glew.h>
#include <GL/freeglut.h>
#include <GL/freeglut_ext.h>
void display (void)
{
}
int main(int argc, char *argv[])
{
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_RGBA);
        glutInitWindowSize(512, 512);
        glutCreateWindow(argv[0]);
        glewInit();
        glutDisplayFunc(display);
        glutMainLoop();
        return 0;
}
```

OpenGL/GLUT Program

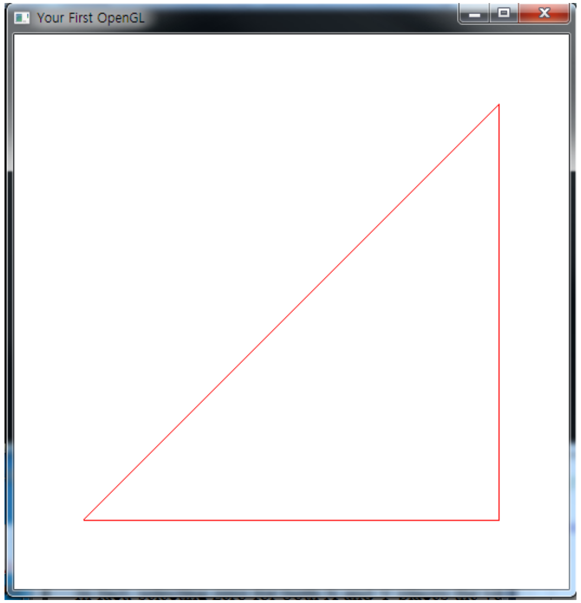
- OpenGL example that paints the window in white.

```
void display (void) {
      glClear(GL_COLOR_BUFFER_BIT);
      glFlush();
}
void init (void)  {
      glClearColor(1.0, 1.0, 1.0, 1.0);
}
int main(int argc, char *argv[])
{
      glutInit(&argc, argv);
      glutInitDisplayMode(GLUT_RGBA);
      glutInitWindowSize(512, 512);
      glutCreateWindow(argv[0]);
      glewInit();
      init();
      glutDisplayFunc(display);
      glutMainLoop();
      return 0;
 }
```

OpenGL/GLUT Program

- OpenGL 2.x program using glVertex

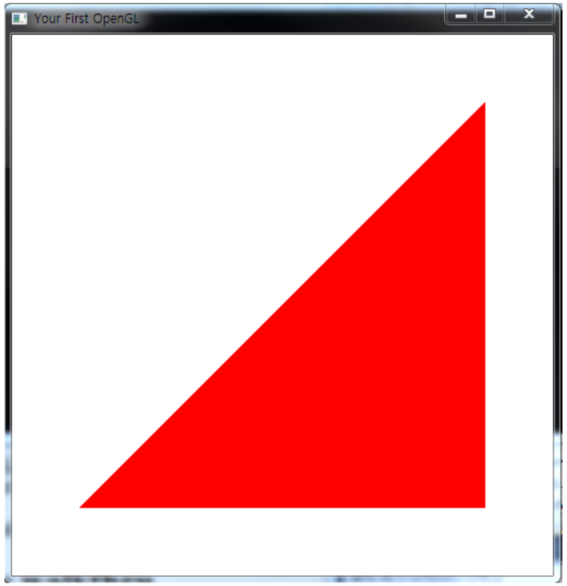
```
void display (void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 0.0, 0.0);
    glBegin(GL_LINE_LOOP);
    glVertex3f(-0.75, -0.75, 0.0);
    glVertex3f(0.75, -0.75, 0.0);
    glVertex3f(0.75, 0.75, 0.0);
    glEnd();
    glFlush();
}
void init (void)
{     /* 변경없음 */
}
int main(int argc, char *argv[])
{     /* 변경없음 */
}
```



OpenGL/GLUT Program

- OpenGL 2.x program using glVertex

```
void display (void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 0.0, 0.0);
    glBegin(GL_POLYGON);
    glVertex3f(-0.75, -0.75, 0.0);
    glVertex3f(0.75, -0.75, 0.0);
    glVertex3f(0.75, 0.75, 0.0);
    glEnd();
    glFlush();
}
```



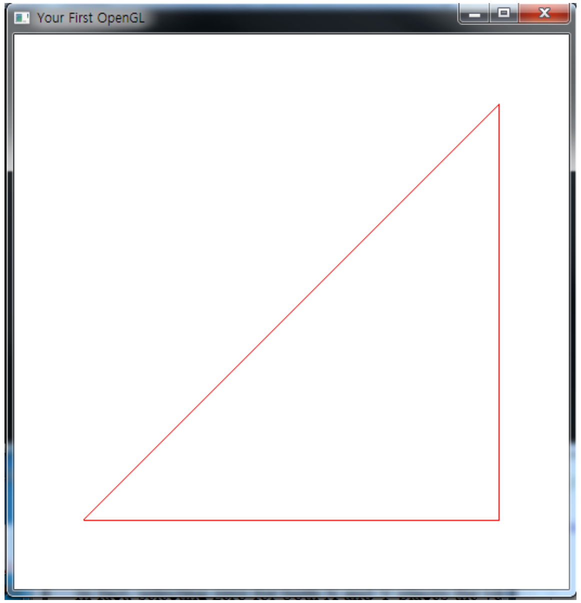
OpenGL/GLUT Program

OpenGL 3.x Program

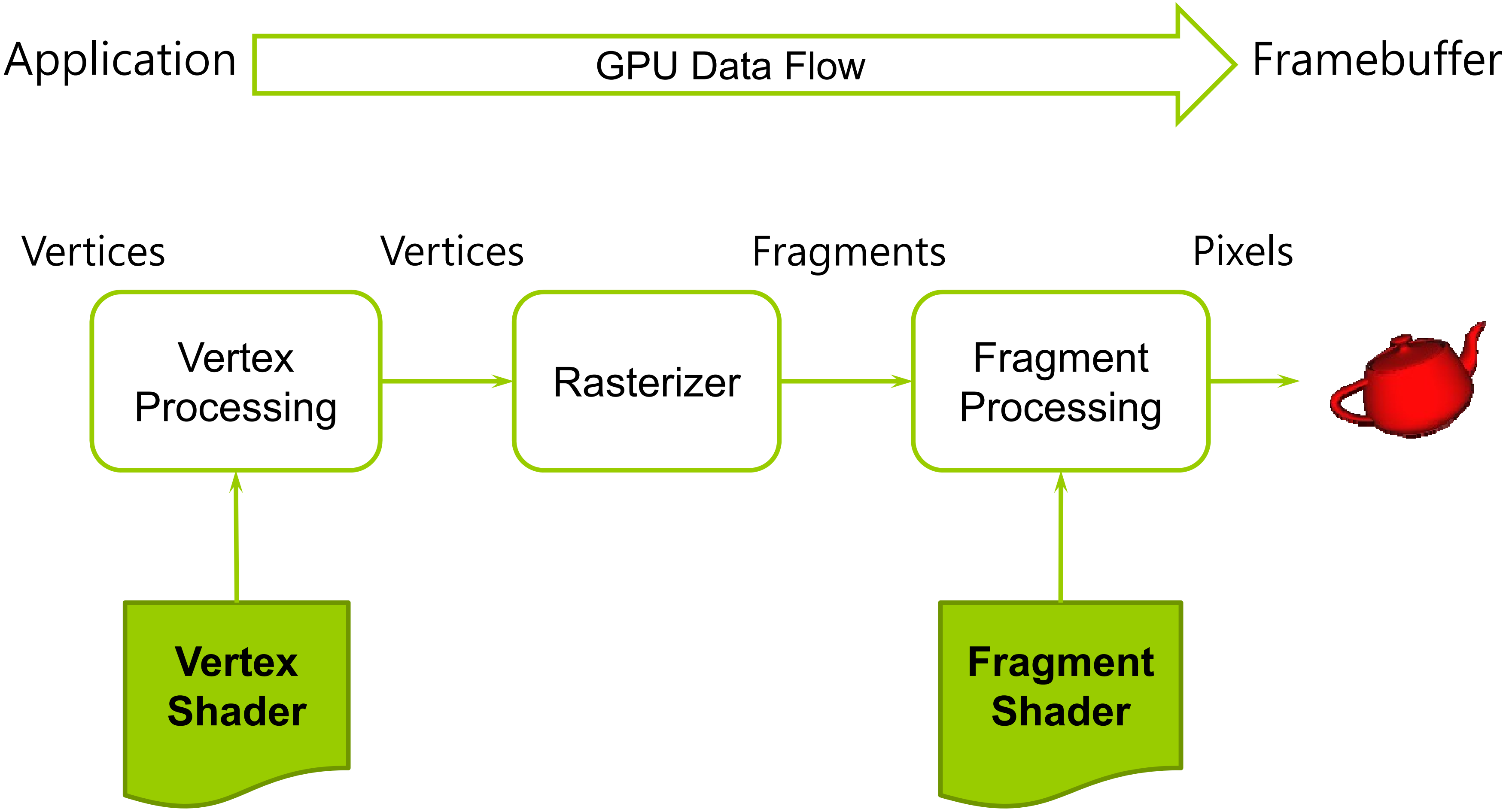
```
void display (void)
{
     glClear(GL_COLOR_BUFFER_BIT);

     /* write this part using shader */

     glFlush();
}
```



OpenGL 3.x Simplified Rendering Pipeline



Application

GPU Data Flow

Framebuffer

Vertices

Vertex Processing

Vertices

Rasterizer

Fragments

Fragment Processing

Pixels

Vertex Shader

Fragment Shader

Modern OpenGL Programming in a Nutshell

- Modern OpenGL (OpenGL 3.x) program proceeds in the following steps
  1. Cerate a Shader program.
  2. Create Vertex Buffer Object (VBO) and Vertex Array Object (VAO) for vertex data and load them into the shader.
  3. Connect this data location and shader variables.
  4. Perform rendering.

OpenGL 3.x Program Structure

- General structure of OpenGL 3.x program
  - **main()**: specify related callback function, etc
    - Specifies the callback functions
    - Opens one or more windows with the required properties
    - Enters event loop (last executable statement)
  - **init()**: specify state variables
    - Viewing
    - Attributes
  - **initShader():** read, compile and link shaders
  - callbacks
    - Display function
    - Input and window functions

OpenGL/GLUT Programming

```
#include <GL/glew.h>
#include <GL/freeglut.h>
#include <GL/freeglut_ext.h>

int main(int argc, char** argv)
{
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
    glutInitWindowSize(512,512);
    glutInitWindowPosition(0,0);
    glutCreateWindow(argv[0]);
    glutDisplayFunc(display);
    glewInit();
    init();
    glutMainLoop();
}
```

includes **gl.h**

specify window properties

display callback

set OpenGL state and initialize shaders

enter event loop

OpenGL/GLUT Programming

- void glutInit(int *argc, char **argv)
  - Initialize the GLUT and OpenGL environments. Pass the arguments of the main function as it is.
- void glutInitWindowSize(int width, int height)

  void glutInitWindowPosition(int x, int y)
  - Specify the window size. Specify the window position.
- int glutCreateWindow(char *name)
  - Open a window. As for the argument name, the name of the window is displayed in the title bar.
- void glutDisplayFunc(void (*func)(void))
  - Argument func is a pointer to the callback function to display (i.e., draw) in the window. This function is executed when a window is opened or a window hidden by another window is redisplayed.
- void glutMainLoop(void)
  - GLUT loop is waiting for an event to be called.

OpenGL/GLUT Programming

- void glutInitDisplayMode(unsigned int mode)
  - Set the display mode of the display. When GLUT_RGBA is specified in mode, it specifies that color designation is used RGB. In addition, efficiency can be improved by specifying the index color mode (GLUT_INDEX).
- void glClearColor(Glclampf R, Glclampf G, Glclampf B, Glclampf A)
  - Specify the color when filling the window. R, G, B, and A have the values between 0 and 1. If (0, 0, 0, 1) is specified, black opacity is drawn.
- void glClear(Glbitfield mask)
  - Paint the window with mask. Mask specifies how the buffer to be filled. In frame buffer, color buffer, depth buffer, stencil buffer, overlay buffer, etc is overlapped. When GL_COLOR_BUFFER is specified, only the color buffer is painted.
- void glFlush(void)
  - Execute all OpenGL commands that have not yet been executed.

Immediate Mode Graphics

- Geometry is specified as vertices
  - Vertex at a point in 2D or 3D space
  - Points, lines, circles, polygons, curves, surfaces
- Immediate mode
  - Every time a vertex is specified in the application, its position value is sent to the GPU.
  - **Use glVertex in the past**
  - Occur bottleneck between CPU and GPU
  - Removed from OpenGL 3.1

Retained Mode Graphics

- Specify all vertex and attribute information as an array, e.g. VBO, VAO
- Send this array to the GPU, store it, and use it for rendering.

Display Callback

```
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glDrawArrays(GL_TRIANGLES, 0, 3);
    glFlush();
}
```

- Draw using glDrawArrays.
- In this case, arrays are buffer objects with vertex arrays.

Vertex Arrays

- Vertices properties
  - Position
  - Color
  - Texture Coordinates
  - Application data

```
const float vertexPositions[] =
{
    -0.75f, -0.75f, 0.0f, 1.0f,
    0.75f, -0.75f, 0.0f, 1.0f,
    0.75f, 0.75f, 0.0f, 1.0f,
};
```

Vertex Array Object

- Vertex Array Object combines all vertex data (positions, colors, ..).
  - Can have multiple buffer objects for one VAO.
  - For example, the vertex position buffers and the vertex color buffers are used in VAO.
  - Usually one VAO is used per mesh.
- Create VAO, and bind it

```
GLuint vao;           // vertex array object
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);
```

  - This means that it has only Vertex Array, so there is no actual data(content) yet.

Buffer Object

- Buffer objects can send large amounts of data to GPU.
- Create buffers, bind them, and put data in them.

```
GLuint buffer;          // vertex buffer object
glGenBuffers(1, &buffer);
glBindBuffer(GL_ARRAY_BUFFER, buffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertexPositions),
                              vertexPositions, GL_STATIC_DRAW);
```

- Then, glVertexAttribPointer is called to tell which attribute buffer the currently bound buffer is and which attribute it has. Then glEnableVertexAttribArray (index) must be called to render the information of the corresponding buffer.

```
glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(0);
```

Initialization

- Specify Vertex Array Objects and Buffer Objects in **init()**
- Also, in general, specify GL clear color and parameters
- Also, specify shaders
  - Read
  - Compile
  - Link

Shader Functions

- glCreateProgram – Create a program (pipeline containing vertex shader, geometry shader, fragment shader)
- glCreateShader – Ask to create the shader space.
- glShaderSource – Connect the allocated shader space and the buffer that reads the shader source file.
- glCompileShader – Compile the shader
- glAttachShader – Attach the compiled shader to the program
- glLinkProgram – Link and check for errors. When the link is complete, the program is ready.
- glUseProgram – Use this program

OpenGL/GLUT Programming

```
// Create a NULL-terminated string by reading the provided file
static char* readShaderSource(const char* shaderFile)
{
    FILE* fp = fopen(shaderFile, "r");
    if ( fp == NULL ) { return NULL; }

    fseek(fp, 0L, SEEK_END);
    long size = ftell(fp);

    fseek(fp, 0L, SEEK_SET);
    char* buf = new char[size + 1];
    fread(buf, 1, size, fp);

    buf[size] = '₩0';
    fclose(fp);

    return buf;
}
```

OpenGL/GLUT Programming

```
// Create a GLSL program object from vertex and fragment shader files
GLuint InitShader(const char* vShaderFile, const char* fShaderFile)
{
    struct Shader { const char* filename; GLenum type; GLchar* source; }
    shaders[2] = { { vShaderFile, GL_VERTEX_SHADER, NULL },
                  { fShaderFile, GL_FRAGMENT_SHADER, NULL }  };
    GLuint program = glCreateProgram();
    for ( int i = 0; i < 2; ++i ) {
        Shader& s = shaders[i];
        s.source = readShaderSource( s.filename );
        if ( shaders[i].source == NULL ) {
            std::cerr << "Failed to read " << s.filename << std::endl;
            exit( EXIT_FAILURE );
        }
        GLuint shader = glCreateShader( s.type );

        glShaderSource( shader, 1, (const GLchar**) &s.source, NULL );
        glCompileShader( shader );
```

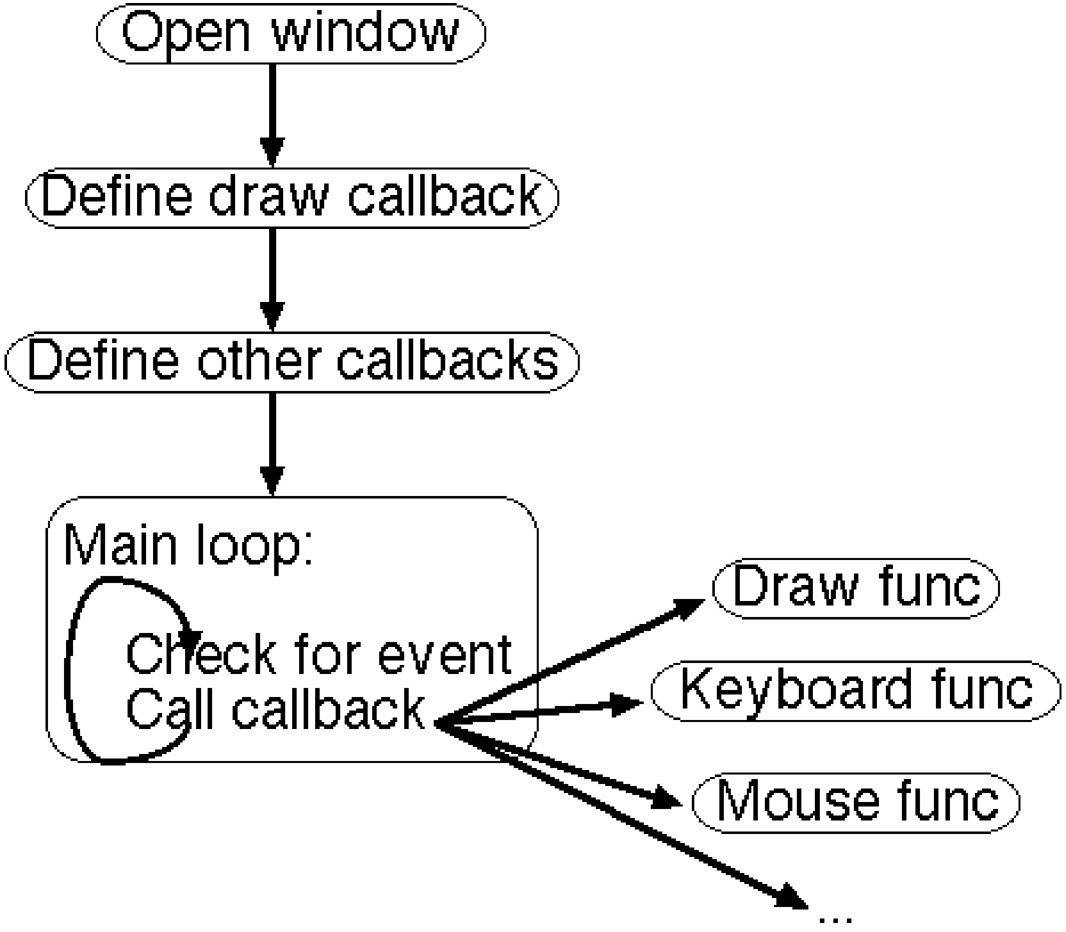
OpenGL/GLUT Programming

```
            // ...
            delete [] s.source;
            glAttachShader( program, shader );
    }
    glLinkProgram(program);
    GLint  linked;
    glGetProgramiv( program, GL_LINK_STATUS, &linked );
    if ( !linked ) {
        std::cerr << "Shader program failed to link" << std::endl;
        GLint  logSize;
        glGetProgramiv( program, GL_INFO_LOG_LENGTH, &logSize);
        char* logMsg = new char[logSize];
        glGetProgramInfoLog( program, logSize, NULL, logMsg );
        delete [] logMsg;
        exit( EXIT_FAILURE );
    }
    glUseProgram(program);
    return program;
}
```

GLUT (OpenGL Utility Toolkit)

- Protable windowing and interaction API developed by Mark J. Kilgard
  - Start with prefix “glut”
  - Provide an upper interface that wraps common functions in most window systems (portable across all PC and workstation OS platforms)
  - Provide utility functions at a higher level than OpenGL
  - Can reuse the code developed in UNIX/X-window
  - No need to know Win32, MFC, Xlib
  - However, it is possible to use the functions of the window system only with limited use

GLUT Program Structure



GLUT Callbacks

- GLUT defines a basic program structure - an *event loop*, with *callback functions*.
- *Callback,* a function that you provide for other code to call when needed; the "other code" is typically in a library
- GLUT uses callbacks for drawing, keyboard & mouse input, and other events.
- Whenever the window must be redrawn, your drawing callback is called.
- Whenever an input event occurs, your corresponding callback is called.

GLUT Callbacks

- glutDisplayFunc
- glutPostRedisplay
- glutIdleFunc
- glutTimerFunc
- glutGetModifiers
- glutIgnoreKeyRepeat
- glutKeyboardFunc
- glutKeyboardUpFunc
- glutSpecialFunc
- glutSpecialUpFunc

GLUT Callbacks

- Handling *Display* Callbacks
  - glutDisplayFunc(void (*func)(void))
    - Specifies a function that would draw the graphical contents
    - Argument: display function name
- Handling *Input* Events
  - glutReshapeFunc(void (*func)(int w, int h))
    - What action should be taken when the window is resized
  - glutKeyboardFunc(void (*func)(unsigned char key, int x, int y))
  - glutMouseFunc(void (*func)(int button, int state, int x, int y))
    - Handle keyboard key and mouse button
  - glutMotionFunc(void (*func)(int x, int y))
    - What to do when the mouse is moved while a mouse button is pressed

The *Reshape* Event Callback

- This callback sets up OpenGL to display images in a window of the new size
- The value w is the new width, and the value h is the new height

```
void reshape(int w, int h)
{
    glViewport(0, 0, w, h);
}
```

The *Keyboard* Event Callback

- This callback simply causes the program to exit when the user hits the ESC key

```
void keyboard(unsigned char key, int x, int y)
{
  switch (key): /* ESC-key & q-key exits the program */
   {
      case 27:
      case ‘q’:
         exit(0);
   }
}
```

GLUT Functions

- glutMainLoop(void)
  - Enter the GLUT event processing loop
- glutPostRedisplay()
  - Ensures that the window gets drawn at most once each time GLUT goes through the event loop.
  - In general, never call the display callback directly, but rather use the glutPostRedisplay() whenever the display needs to be redrawn.