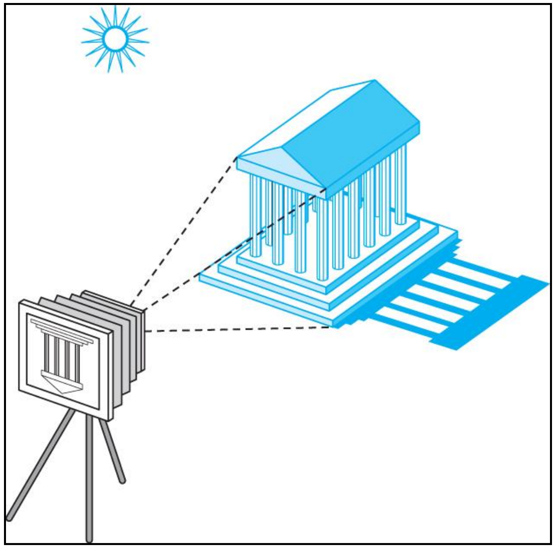
GPU based OpenGL

Sang Il Park
Dept. of Software

Review: Elements of Image Formation

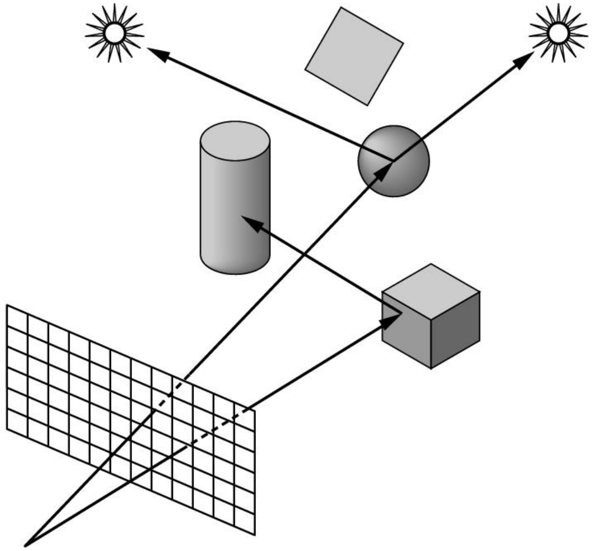
- Objects
- Viewer
- Light source(s)
- Attributes (materials)
  - govern how light interacts with the materials in the scene



Attributes is getting more important

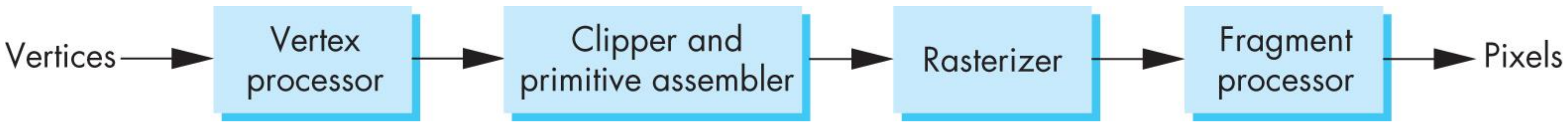
Review: Ray tracing

- **Ray tracing**: follow rays of light from center of projection until they either are absorbed by objects or go off to infinity
  - Can handle global effects
    - Multiple reflections
    - Translucent objects
  - Slow
  - Must have whole data base available at all times



Review: “Simplified” Approach

- Process objects one at a time in the order they are generated by the application
  - Can consider only local lighting
- Pipeline architecture



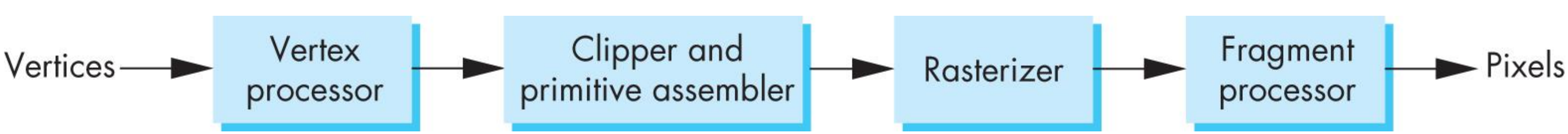
- All steps can be implemented in hardware on the graphics card

Review: ***Vertex*** (vertices) Processing

- Much of the work in the pipeline is in converting object representations from one coordinate system to another
  - Object coordinates
  - Camera (eye) coordinates
  - Screen coordinates

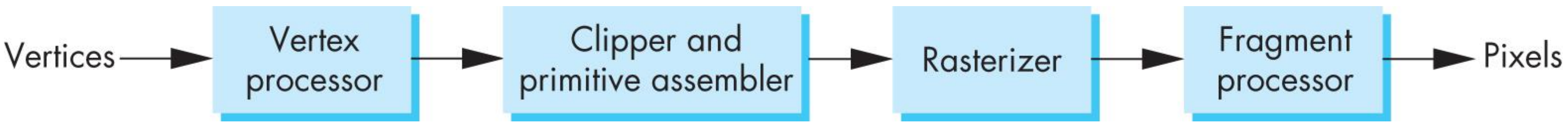
PROJECTION

- Every change of coordinates is equivalent to a matrix transformation



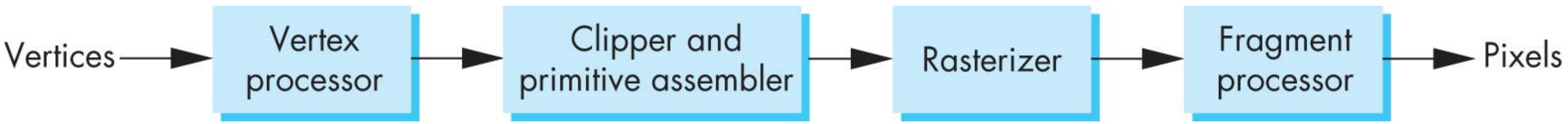
Review: Rasterization

- If an object is visible, the appropriate pixels in the frame buffer must be assigned colors
- Rasterizer produces a set of fragments for each object
- ***Fragments*** are “potential pixels”
  - Have a location in frame bufffer
  - Color and depth attributes
- Vertex attributes are interpolated over objects by the rasterizer



Review: ***Fragment*** Processing

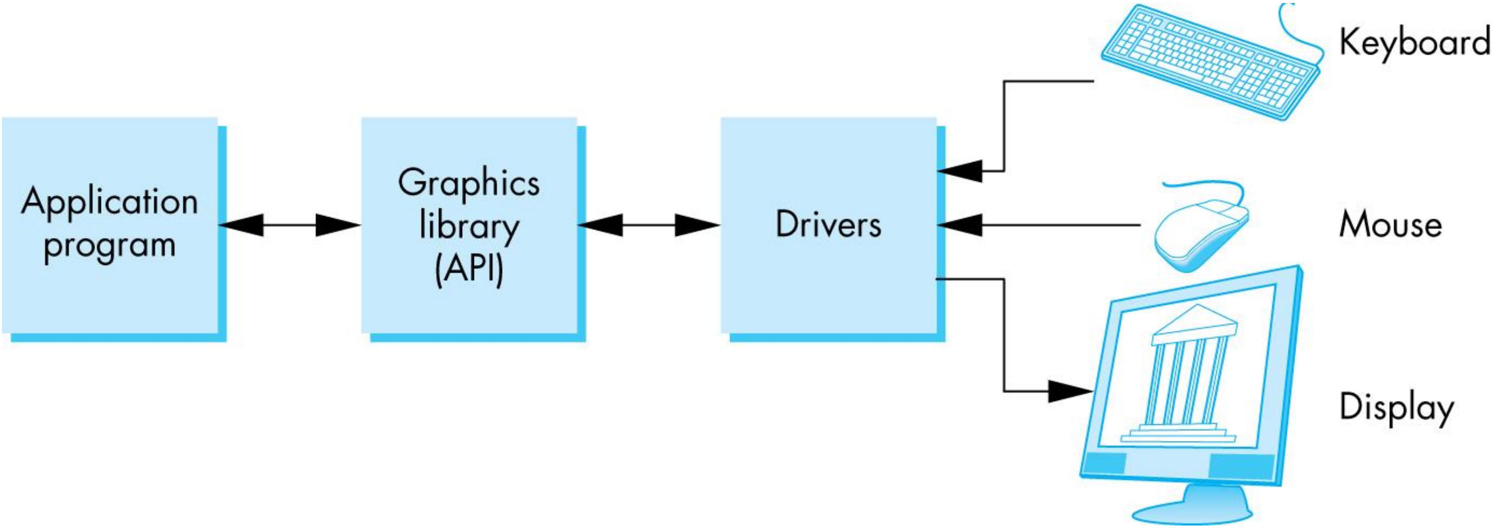
- Fragments are processed to determine the color of the corresponding pixel in the frame buffer
- Colors can be determined by texture mapping or interpolation of vertex colors
- Fragments may be blocked by other fragments closer to the camera
  - Hidden-surface removal



OpenGL: the API

The Programmer's Interface

- Programmer sees the graphics system through a software interface: the Application Programmer Interface (API)



API Contents

- Functions that specify what we need to form an image
  - Objects
  - Viewer
  - Light Source(s)
  - Attributes (Materials)
- Other information
  - Input from devices such as mouse and keyboard
  - Capabilities of system

Object Specification

- Most APIs support a limited set of primitives including
  - Points (0D object)
  - Line segments (1D objects)
  - Polygons (2D objects)
  - Some curves and surfaces
    - Quadrics
    - Parametric polynomials
- All are defined through locations in space or *vertices*

Object Specification (old style)

type of object

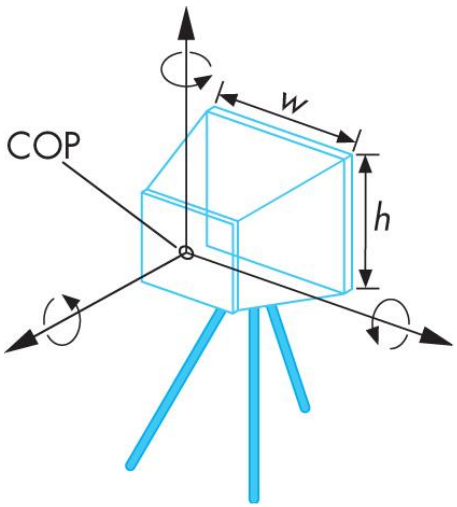
location of vertex

```
glBegin(GL_POLYGON)
 glVertex3f(0.0, 0.0, 0.0);
 glVertex3f(0.0, 1.0, 0.0);
 glVertex3f(0.0, 0.0, 1.0);
glEnd( );
```

end of object definition

Camera Specification (old style)

- Six degrees of freedom
  - Position of center of lens
  - Orientation
- Lens
- Film size
- Orientation of film plane



- → glPerspective(…), gluLookAt(…)

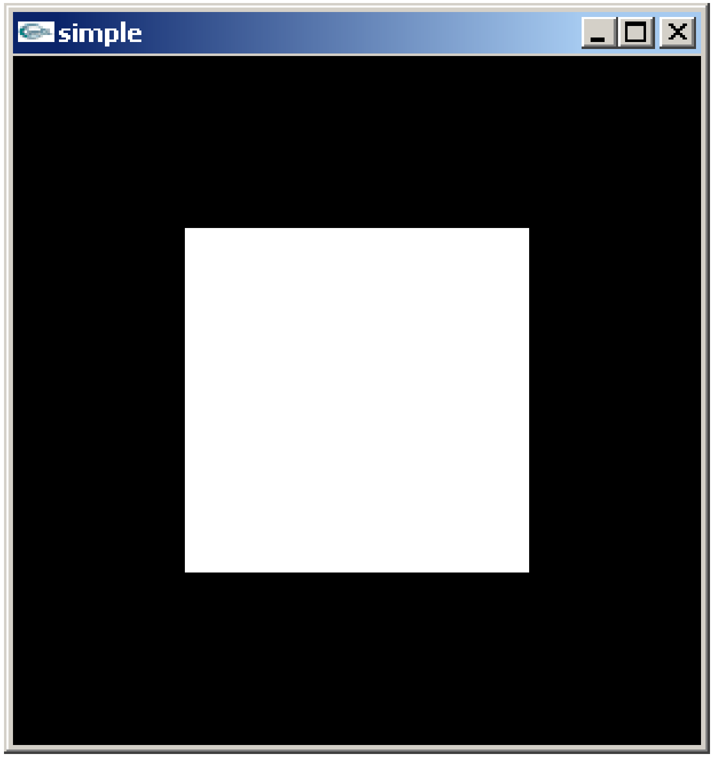
Lights and Materials (old style)

- Types of lights
  - Point sources vs distributed sources
  - Spot lights
  - Near and far sources
  - Color properties
- Material properties
  - Absorption: color properties
  - Scattering
    - Diffuse
    - Specular

→ glLight(…), glMaterial(…)

A Simple Program (old style)

Generate a square on a solid background



Let's start to CODE it!

- Preparation

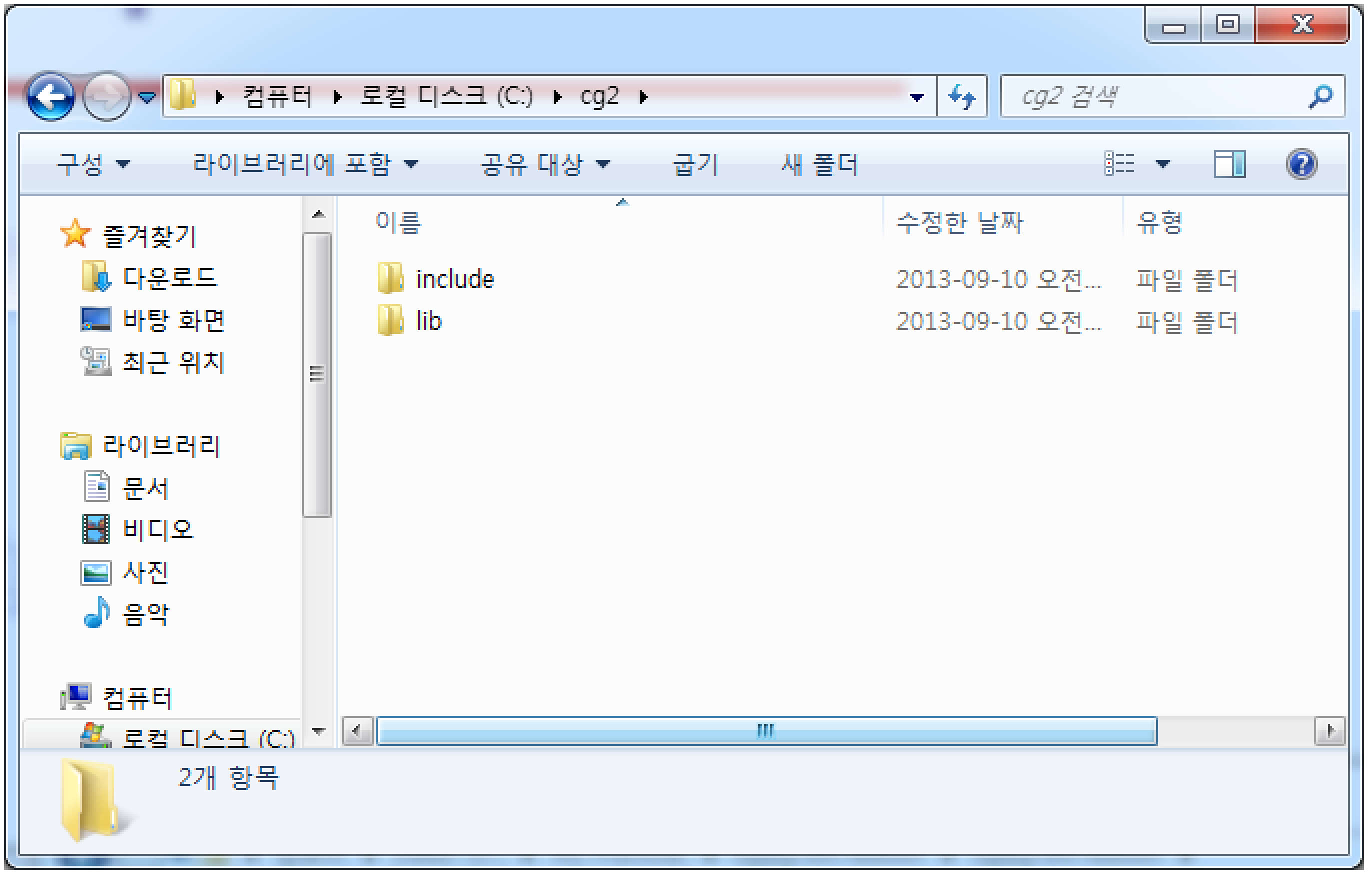
1. Download necessary libraries
    - Header files: Include folder
    - LIB files : lib folder
    - DLL files : bin(or system32) folder
2. Change the project setting
    - Directory setting

Required Libraries

- freeglut: http://freeglut.sourceforge.net/
- GLEW: http://glew.sourceforge.net/

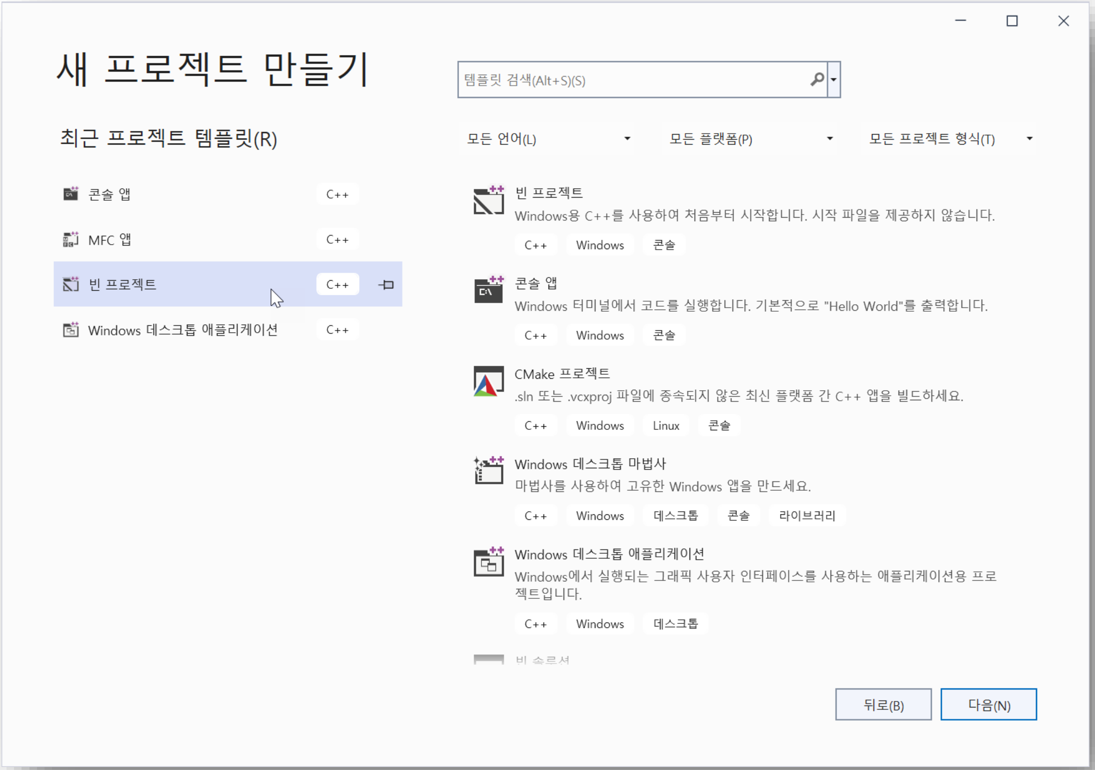
I put both of them and more at our homepage:
freeglut_and_glew.zip
➔ Download it and unzip it at “c:/cg2/”

What you should have in your "c:/cg2/" folder:



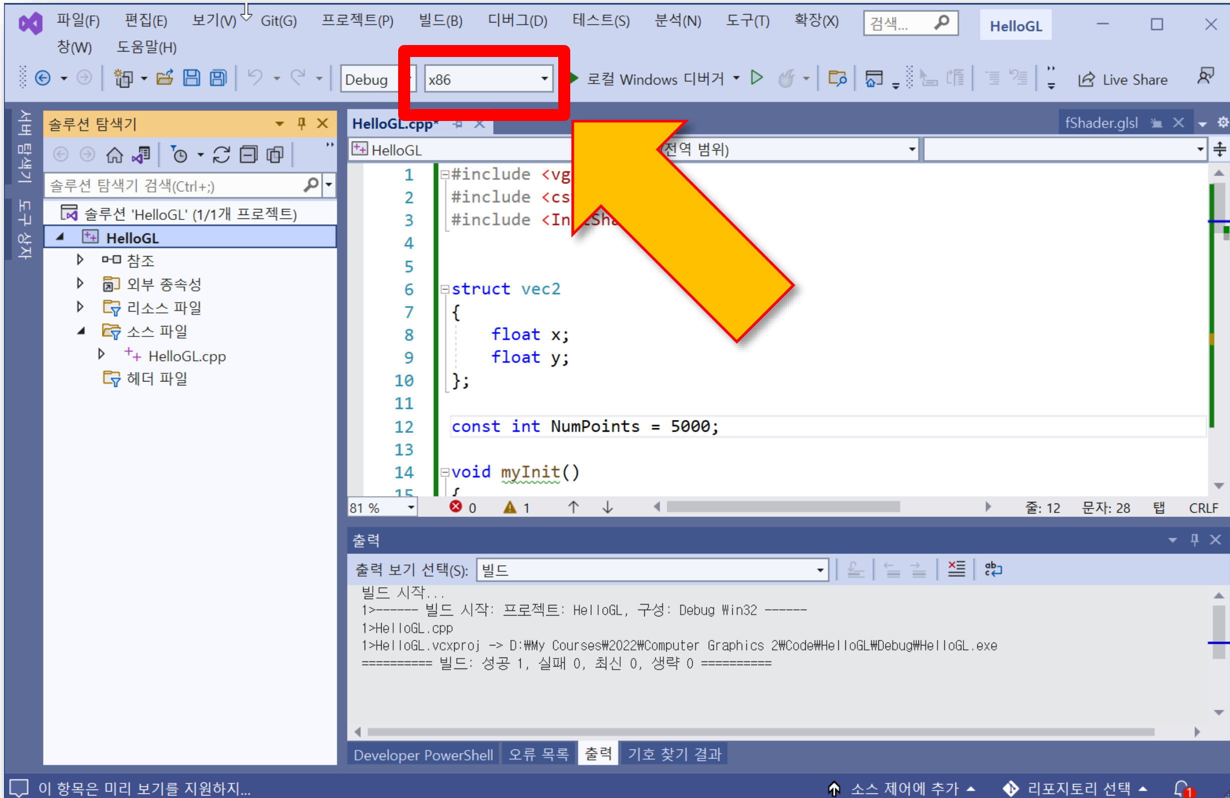
Project Setting:

- Start a new project with a "console application" project with an empty project option.



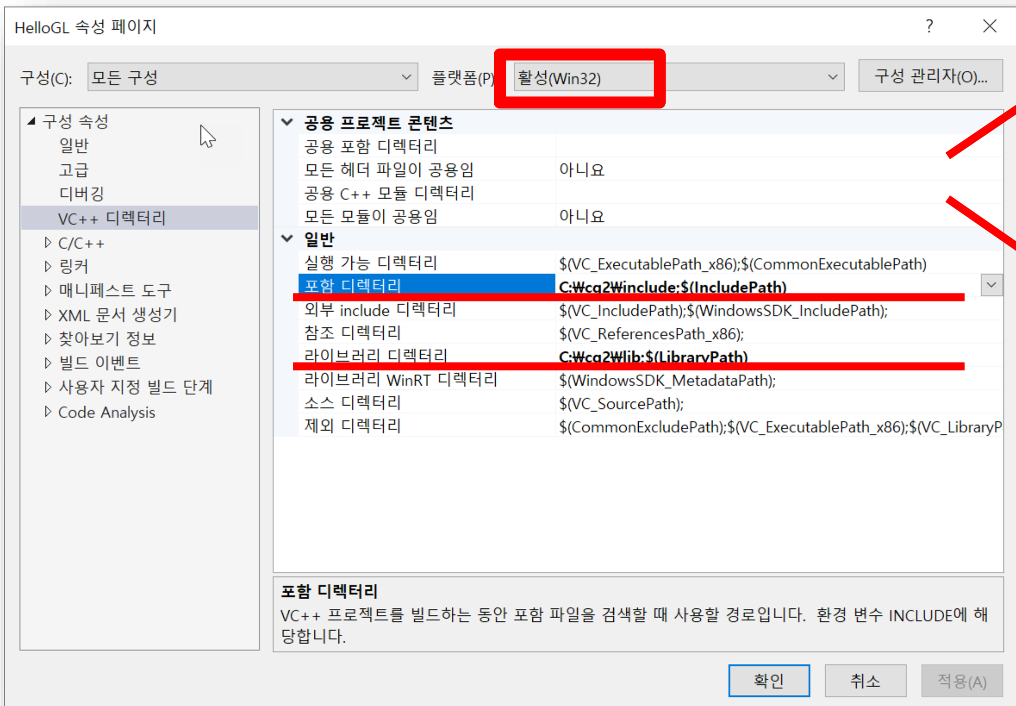
Set Target Platform: x86

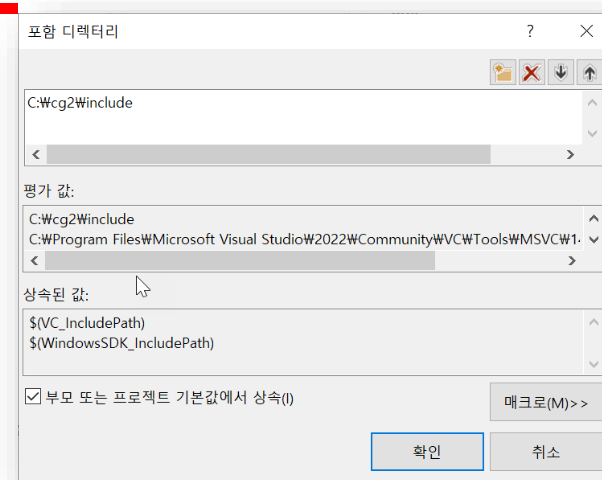
- 다음과 같이 반드시 Target Platform을 x86으로 설정 (x64 아님!)

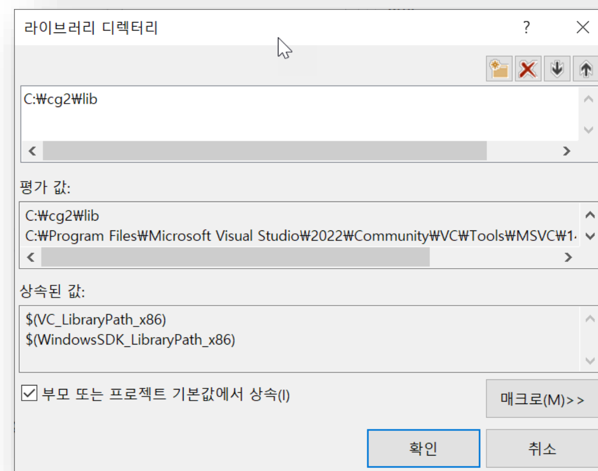


Project Setting

- Set the directories:







Create a new main.cpp file

- And add the following line at the beginning of the code:

```
#include <vgl.h>
```

Now ALL SET!!!!

Hello GL Program:

```
#include <vgl.h>

void display()
{
        glClear(GL_COLOR_BUFFER_BIT);
        glBegin(GL_TRIANGLES);
                glVertex2f(-0.5, -0.5);
                glVertex2f(0.5, -0.5);
                glVertex2f(-0.5, 0.5);
                glVertex2f(0.5, -0.5);
                glVertex2f(0.5, 0.5);
                glVertex2f(-0.5, 0.5);
        glEnd();
        glFlush();
}
```

```
int main(int argc, char** argv)
{
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA);
        glutInitWindowSize(512, 512);
        glutCreateWindow("Hello GL");

        glutDisplayFunc(display);
        glutMainLoop();

        return 0;
}
```

Summary

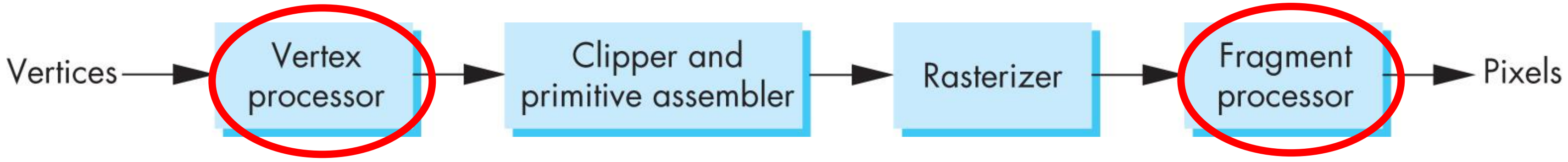
- OpenGL Setting for libraries
  - Set include/lib folder
  - `#include <vgl.h>`

- With OpenGL, drawing was done by just sending the data into the predefined pipeline

Programming with OpenGL in a modern way

Changing in OpenGL

- Performance is achieved by using GPU rather than CPU
- Control GPU through programs called ***shaders***

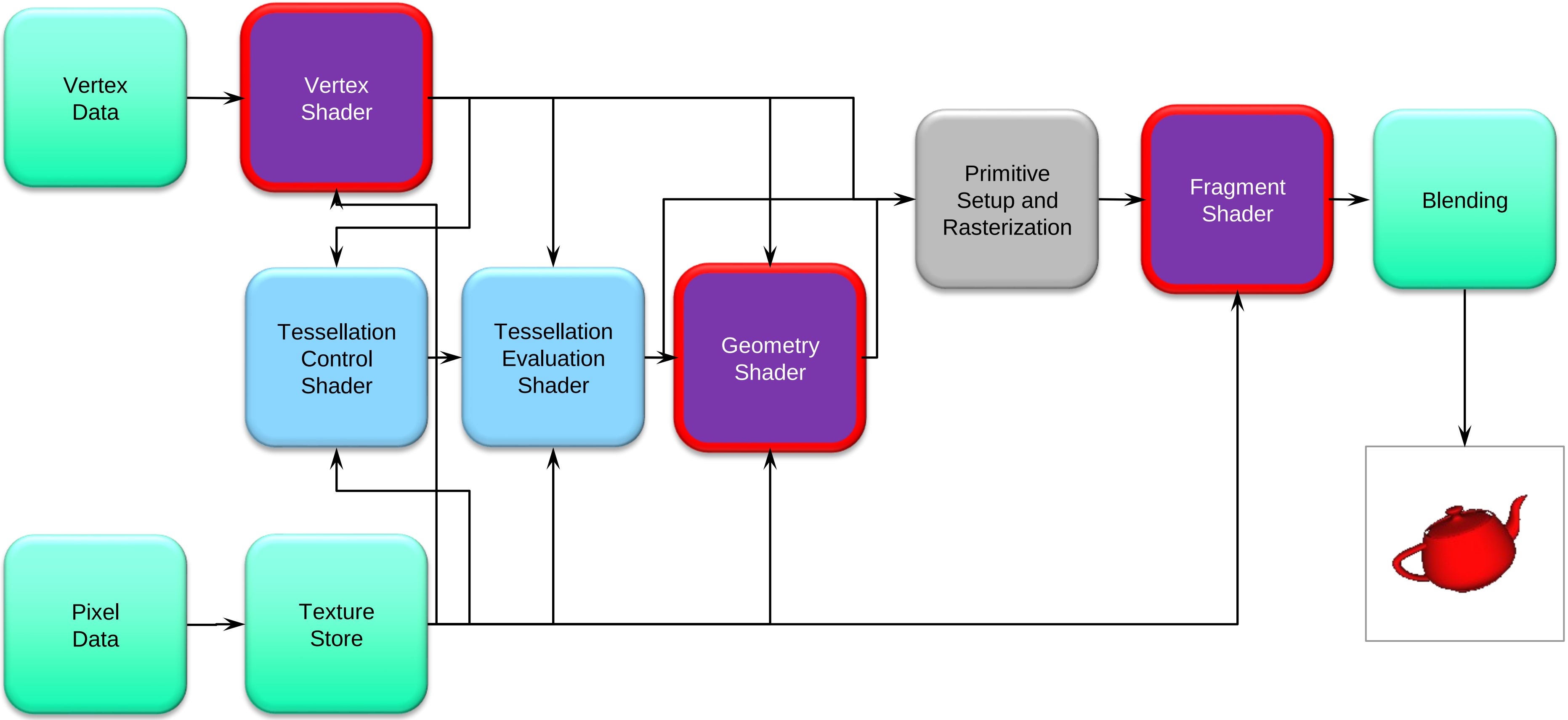


“new” OpenGL

- We’ll concentrate on the latest versions of OpenGL
  - The currently available version is 4.5
  - At least higher than OpenGL 3.0 will work fine with the class
- They enforce a new way to program with OpenGL
  - Allows more efficient use of GPU resources
- modern OpenGL doesn’t support
  - Fixed-function graphics operations
    - lighting
    - transformations
- All applications must use ***shaders*** for their graphics processing

The Latest Pipelines

- Latest version is 4.5 (2014)



OpenGL Libraries

- OpenGL core library
  - OpenGL32 on Windows
  - GL on most unix/linux systems (libGL.a)
- OpenGL Utility Library (GLU)
  - Provides functionality in OpenGL core but avoids having to rewrite code
  - Will only work with legacy code
- Links with window system
  - GLX for X window systems
  - WGL for Windows
  - AGL for Macintosh

GLUT

- OpenGL Utility Toolkit (GLUT)
  - Provides functionality common to all window systems
    - Open a window
    - Get input from mouse and keyboard
    - Menus
    - Event-driven
  - Code is portable but GLUT lacks the functionality of a good toolkit for a specific platform
    - No slide bars

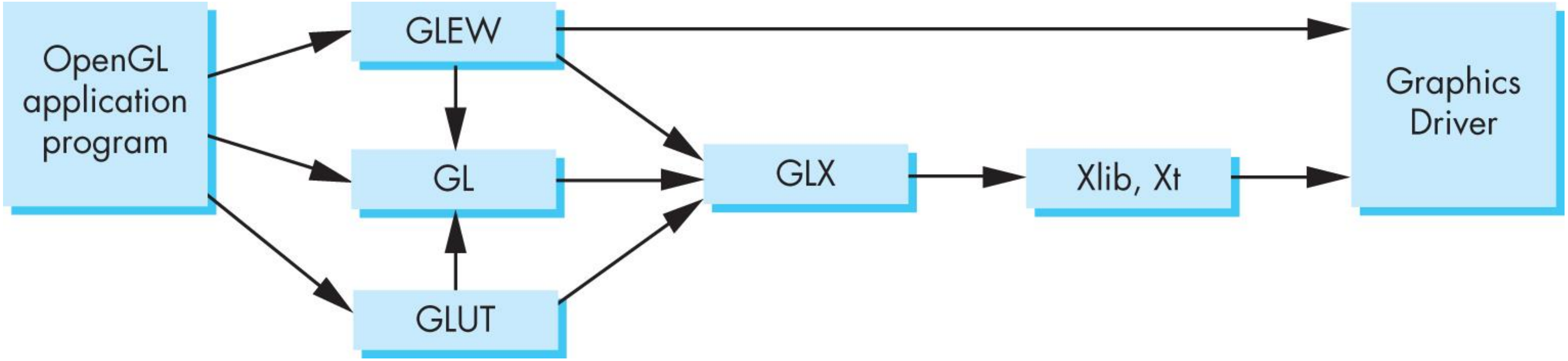
freeGLUT

- GLUT was created long ago and has been unchanged
  - Amazing that it works with OpenGL 3.1
  - Some functionality can't work since it requires deprecated functions
- *freeglut* updates GLUT
  - Added capabilities
  - Context checking

GLEW

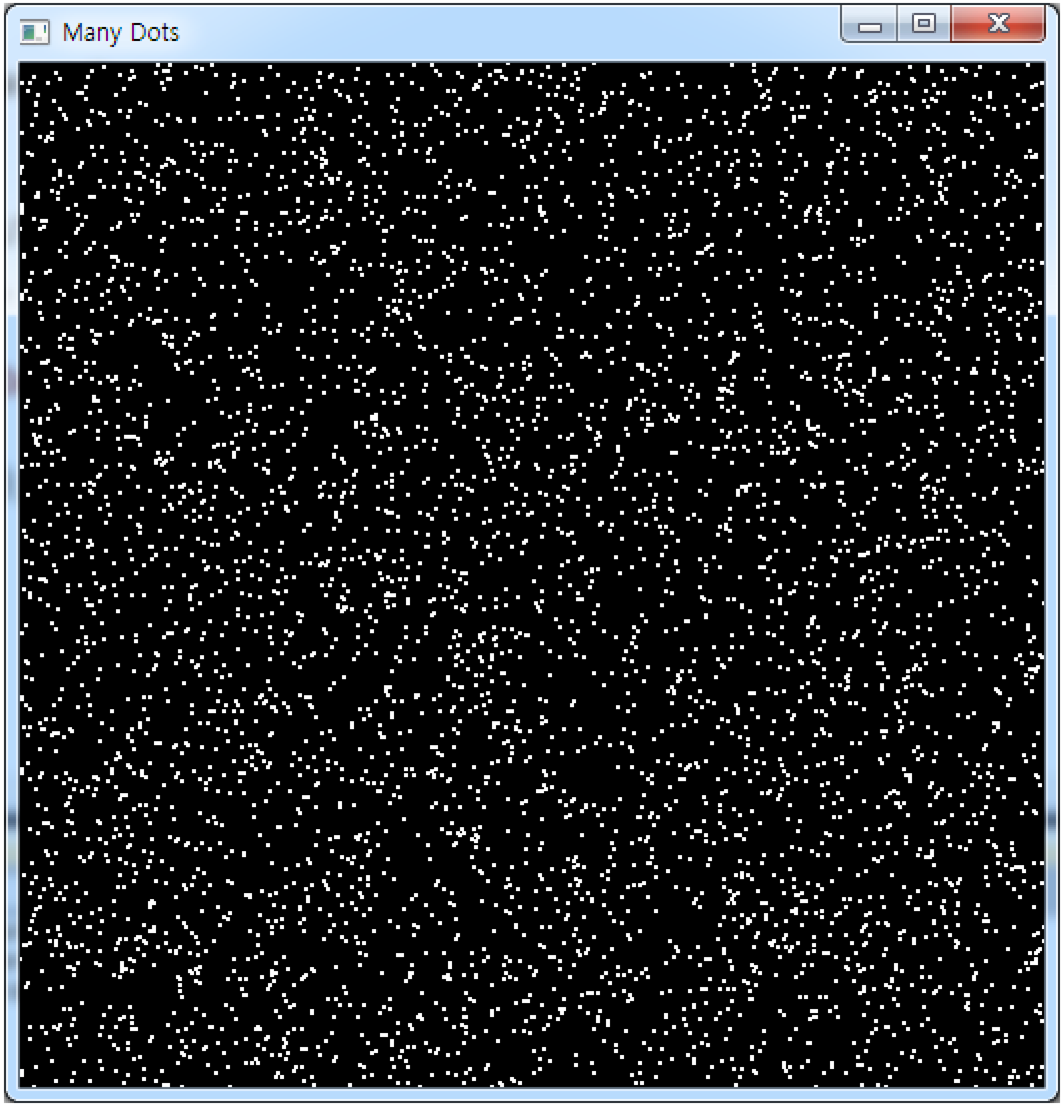
- OpenGL Extension Wrangler Library
- Makes it easy to access OpenGL extensions available on a particular system
- Avoids having to have specific entry points in Windows code
- Application needs only to include glew.h and run a glewInit()

Software Organization



Three different drawing approaches: Many points

Drawing random dots on the screen



Three approaches

1. Immediate mode graphics
2. Retained mode graphics
3. GPU based graphics

Immediate mode graphics

- Generate one primitive at a time and draw it immediately

```
void display()
{
      for(num_points)
      {
            p = generate_a_point();
            Draw_a_point(p);
      }
}
```

Immediate mode graphics:

```
#include <vgl.h>

void display()
{
	glClear(GL_COLOR_BUFFER_BIT);
	glBegin(GL_POINTS);
	for(int i=0; i<5000; i++)
	{
		float x = (rand()%200)/100.0f-1.0f;
		float y = (rand()%200)/100.0f-1.0f;
		glVertex2f(x,y);
	}
	glEnd();
	glFlush();
}
```

```
int main(int argc, char ** argv)
{
	glutInit(&argc, argv);
	glutInitDisplayMode(GLUT_SINGLE|GLUT_RGBA);
	glutInitWindowSize( 512, 512 );
	glutCreateWindow("Many Points");

	glutDisplayFunc(display);
	glutMainLoop();

	return 0;
}
```

Retained mode Graphics

- Generate all primitives first, then draw them all

```
p[num_points];
void initialize()
{
        for(num_points)
        {
                q = generate_a_point();
                p = store_the_point(q);
        }
}

void display()
{
        draw_all_the_points(p)
}
```

GPU-based Graphics

- Generate all the points first, send them to GPU, and then draw them.

```
p[num_points];
void initialize()
{
        for(num_points)
        {
                q = generate_a_point();
                p = store_the_point(q);
        }
        Send_all_points_to_GPU(p);
}

void display()
{
        display_data_on_GPU();
}
```

What is different from the previous one?

A First Program: Many points with GPU

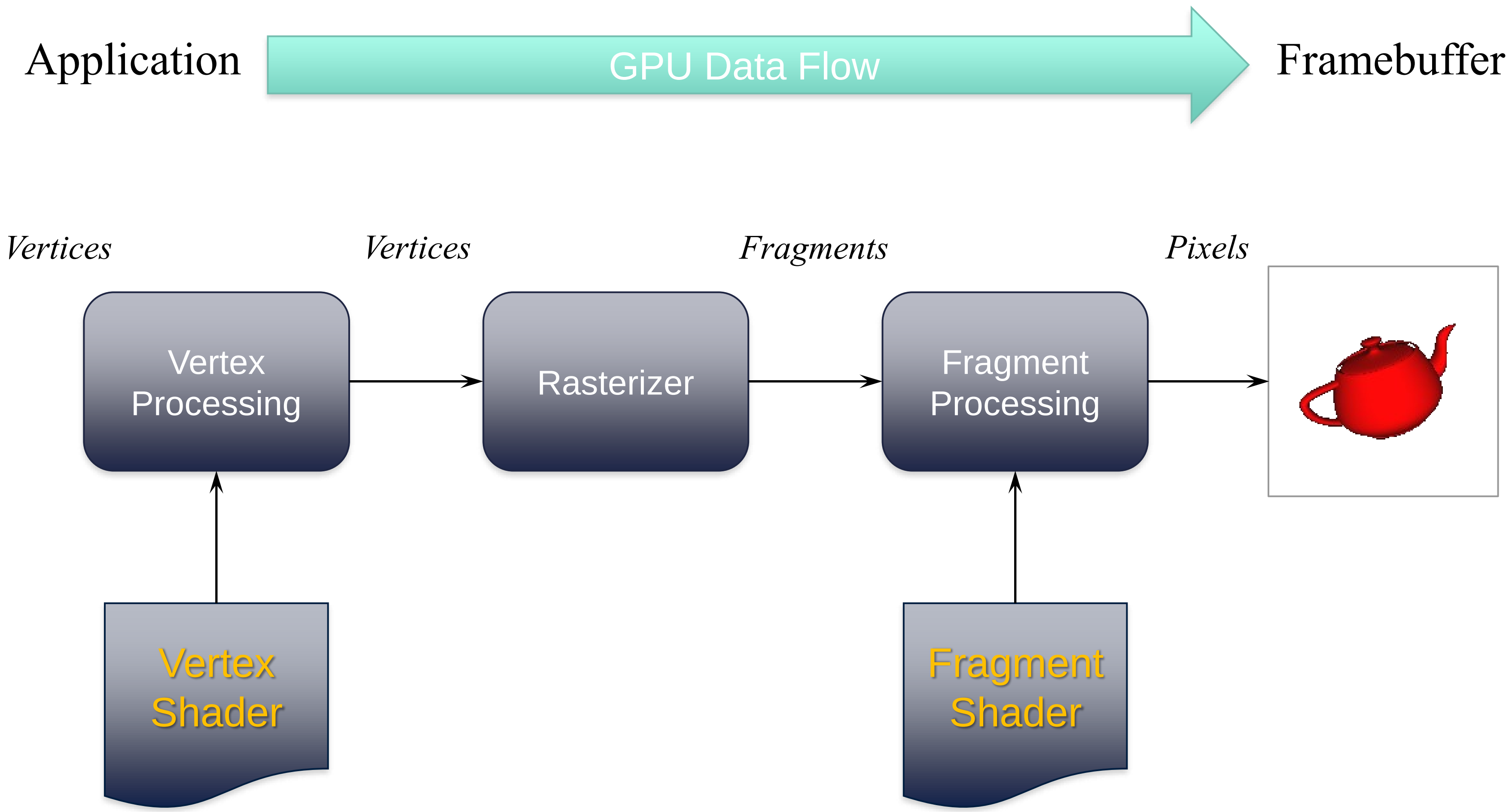
Setting for the most current opengl version for your computer

```
int main(int argc, char ** argv)
{
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_SINGLE|GLUT_RGBA);
        glutInitWindowSize(512,512);
        glutCreateWindow("Many Points GPU");

        glewExperimental = true;
        glewInit();

        printf("OpenGL %s, GLSL %s\n",
                glGetString(GL_VERSION),
                glGetString(GL_SHADING_LANGUAGE_VERSION));

        glutDisplayFunc(display);
        glutMainLoop();

        return 0;
}
```

For using the modern OpenGL

To check the Current version

OpenGL Pipeline (Simplified)

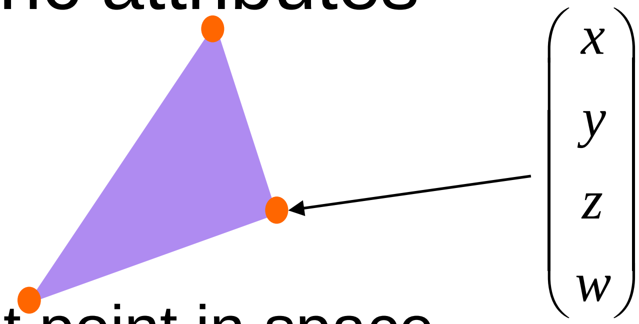


OpenGL Programming in a Nutshell

- Modern OpenGL programs essentially do the following steps:
  1. Create buffer objects and load data into them
  2. Create shader programs
  3. “Connect” data locations with shader variables
  4. Render

Representing Geometric Objects

- Geometric objects are represented using vertices
- A vertex is a collection of generic attributes
  - positional coordinates
  - colors
  - texture coordinates
  - any other data associated with that point in space

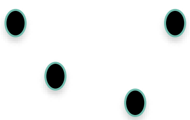


- Position stored in 4 dimensional homogeneous coordinates

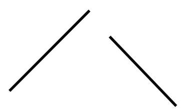
- Vertex data must be stored in *vertex buffer objects* (***VBO***s)
- VBOs must be stored in *vertex array objects* (***VAO***s)

OpenGL's Geometric Primitives

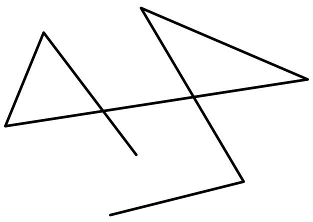
- All primitives are specified by vertices



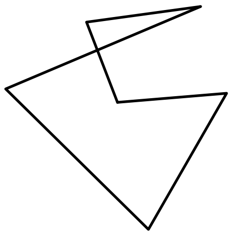
GL_POINTS



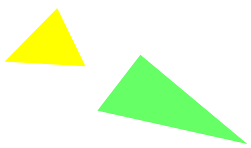
GL_LINES



GL_LINE_STRIP



GL_LINE_LOOP



GL_TRIANGLES

GL_TRIANGLE_STRIP

GL_TRIANGLE_FAN

Creating Data

- Define an array for storing all the points

```
struct vec2
{
	float x;
	float y;
};

const int NumPoints = 5000;

void init()
{
	vec2 points[NumPoints];

	for ( int i = 0; i < NumPoints; i++ )
	{
		points[i].x = (rand()%200)/100.0f-1.0f;
		points[i].y = (rand()%200)/100.0f-1.0f;
	}
}
```

Draw the array at once

- Define an an array for storing all the points

```
void display()
{
      glClear(GL_COLOR_BUFFER_BIT);
      glDrawArrays(GL_POINTS, 0, NumPoints);
}
```

Above code draws the data in GPU.
But we didn’t send the data to GPU at all!!

How to send your data

- Vertex data must be stored in *vertex buffer objects*
(***VBO***s)
- VBOs must be stored in *vertex array objects*
(***VAO***s)

How to send your data

Generate a Vertex Array	`glGenVertexArray(..)`
Bind the Vertex Array	`glBindVertexArray(..)`
Generate a Buffer Object	`glGenBuffers(..)`
Bind the Buffer Object	`glBindBuffer(..)`
Set the Buffer Object data	`glBufferData(..)`

Vertex Array Objects (VAOs)

- VAOs store the data of a geometric object
- Steps in using a VAO
  - generate VAO names by calling `glGenVertexArrays()`
  - bind a specific VAO for initialization by calling `glBindVertexArray()`
  - update VBOs associated with this VAO
  - bind VAO for use in rendering
- This approach allows a single function call to specify all the data for an objects
  - previously, you might have needed to make many calls to make all the data current

VAOs in Code

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

Storing Vertex Attributes

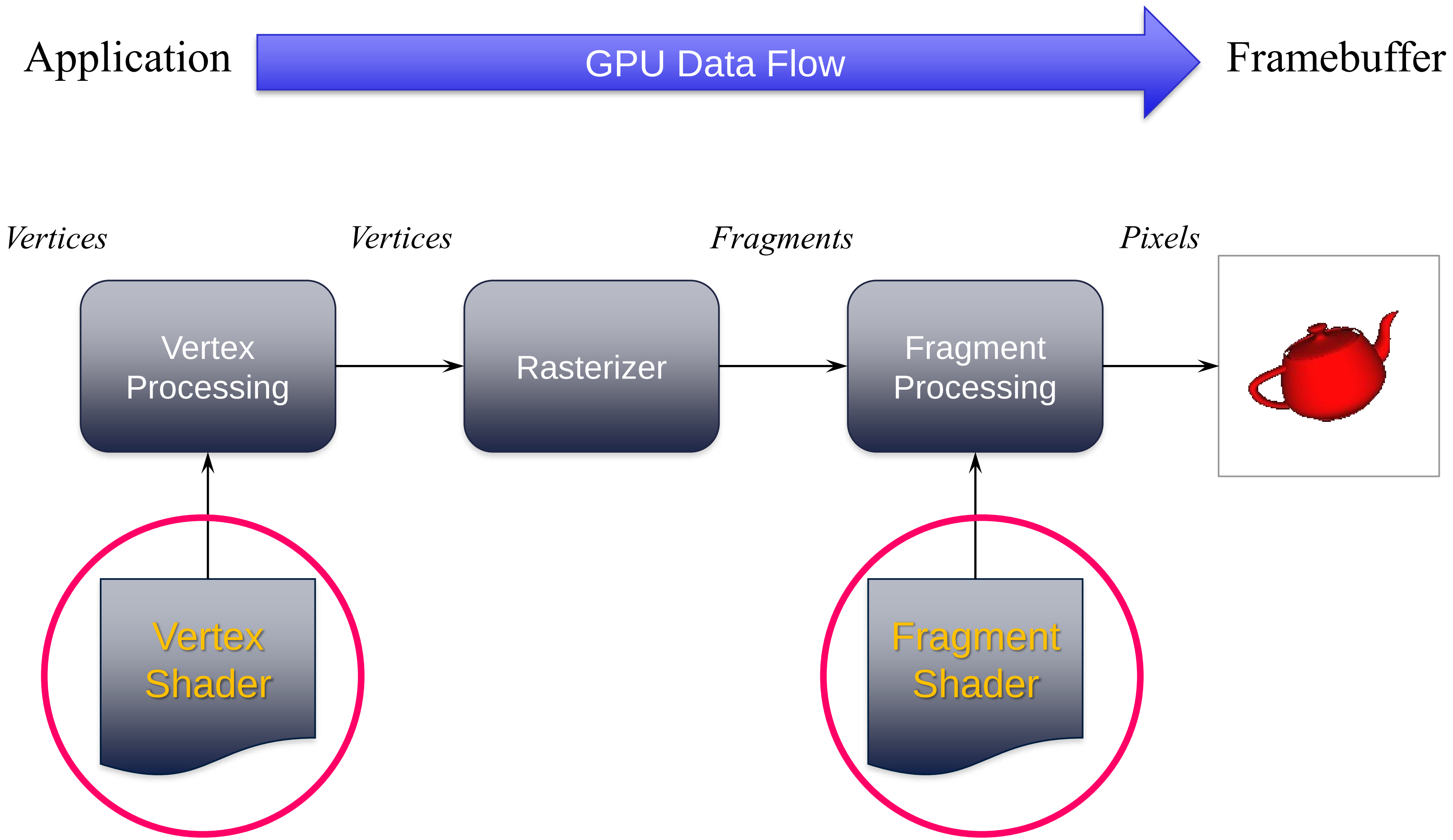
- Vertex data must be stored in a VBO, and associated with a VAO
- The code-flow is similar to configuring a VAO
  - generate VBO names by calling `glGenBuffers()`
  - bind a specific VBO for initialization by calling `glBindBuffer(GL_ARRAY_BUFFER, …)`
  - load data into VBO using `glBufferData(GL_ARRAY_BUFFER, …)`

  - bind VAO for use in rendering later `glBindVertexArray()`

VBOs in Code

```
// Create and initialize a buffer object
GLuint buffer;
glGenBuffers(1, &buffer);
glBindBuffer(GL_ARRAY_BUFFER, buffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(points),
                points, GL_STATIC_DRAW);
```

We need shaders before drawing



Vertex Shader Code (vshader.glsl)

```
#version 330

in vec4 vPosition;

void main()
{
      gl_Position = vPosition;
}
```

Fragment Shader Code (fshader.glsl)

```
#version 330

out vec4 fColor;

void main()
{
      fColor = vec4(1.0,0.0,0.0,1.0);
}
```

Loading Shaders

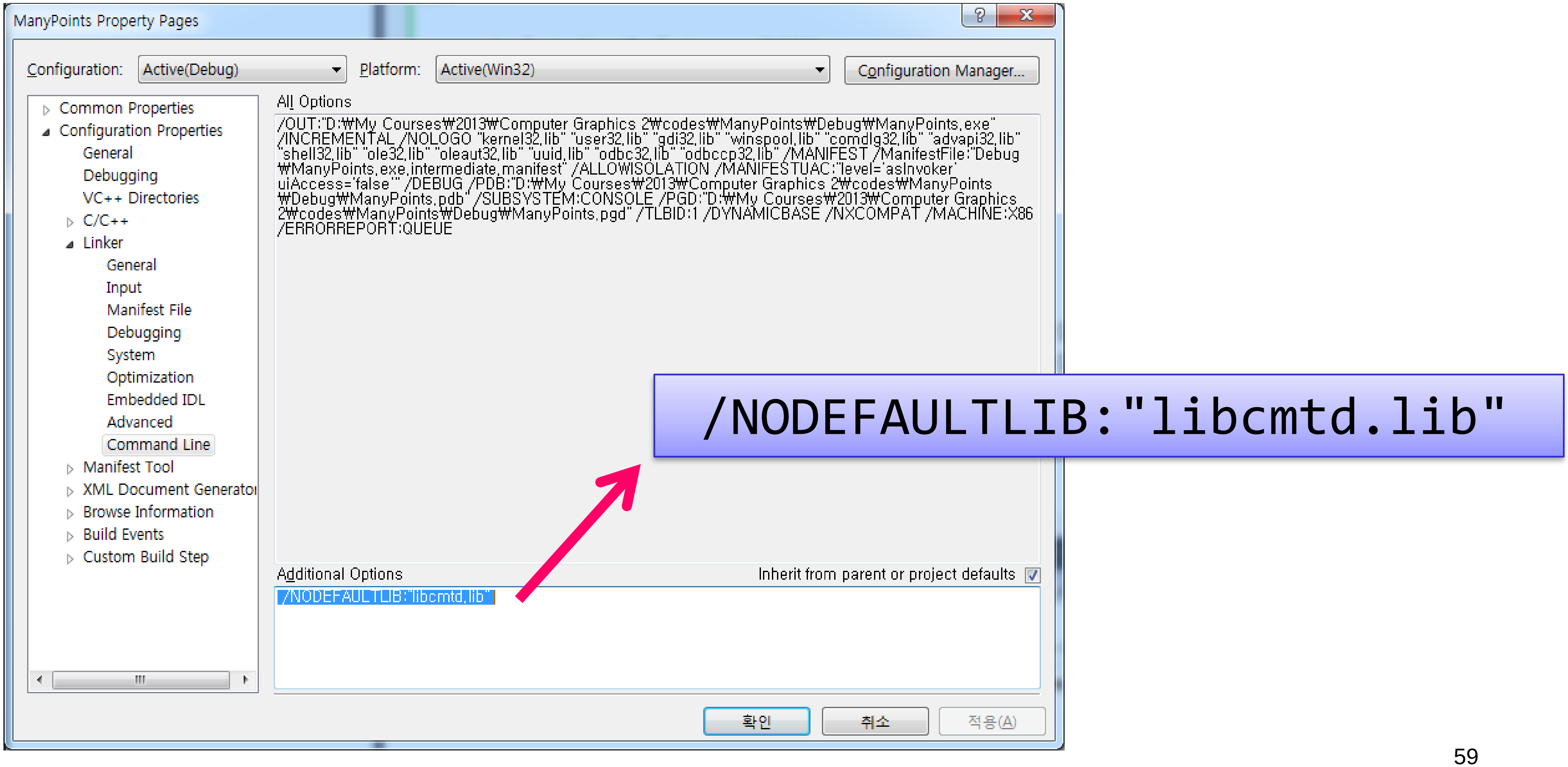
```
#include <InitShader.h>
```

```
// Load and use shaders
GLuint program
     = InitShader( "vshader.glsl", "fshader.glsl" );
glUseProgram( program );
```

glsl : opengl shader language
Those are provided in our homepage.

If you see an error: You should change Project Setting

- Conflict with an existing lib “libcmtd.lib”:



Connecting Vertex Shaders with Geometry

- Application vertex data enters the OpenGL pipeline through the vertex shader
- Need to connect vertex data to shader variables
  - requires knowing the attribute location
- Attribute location can either be queried by calling `glGetVertexAttribLocation()`

Vertex Array Code

```
// set up vertex arrays (after shaders are loaded)
int vPosition = 0;
glEnableVertexAttribArray( vPosition );
glVertexAttribPointer( vPosition, 2, GL_FLOAT,
                GL_FALSE, 0, BUFFER_OFFSET(0) );
```

Drawing Geometric Primitives

- For contiguous groups of vertices

```
glDrawArrays(GL_POINTS, 0, NumPoints);
```

- Usually invoked in display callback
- Initiates vertex shader

Summary

- Setting for libraries
  - Set include/lib folder
  - #include <vgl.h>
  - #include <initshader.h>
- Creating data (in an array form)
- Sending the data
  - VAO – vertex array object
  - VBO – vertex buffer object
- Loading the shaders
- Draw it with `glDrawArrays(...)`