Introduction to Graphics Programming and its Applications

繪圖程式設計與應用

OpenGL Rendering Pipeline

Instructor: Hung-Kuo Chu

Department of Computer Science

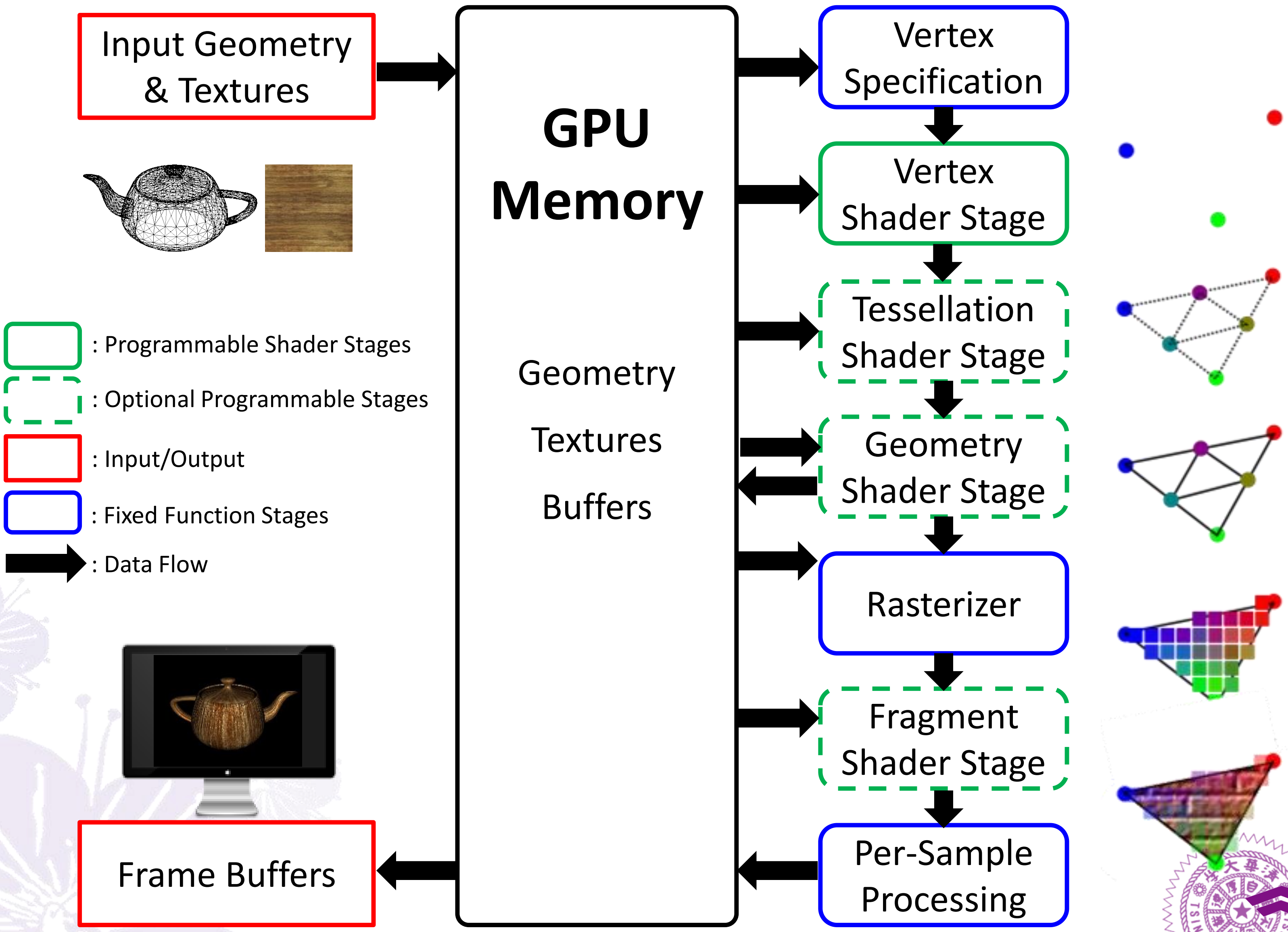
National Tsing Hua University

CS4105

OpenGL Rendering Pipeline

- ***Pipeline***: consists of multiple stages. Data flows in, being processed in each stages, then flows out
- ***Stages***: each stage represents an unique function to process the input data
  - ***Fixed function stages***: limited customization capability, typically exposes states for configuration
  - ***Programmable shader stages***: allow custom shader programs to be executed within, providing broader capability of customization

The OpenGL Rendering Pipeline



OpenGL Object

- Before introducing the pipeline, ***OpenGL objects*** should be introduced briefly
- OpenGL has an ***object-oriented resource management system***
- GPU memories are represented by resource objects:
  - ***Buffer Objects***: array-like or structure-like data (geometry, camera settings, look up tables…)
  - ***Texture Objects***: 1D, 2D, 3D or cube images, or their array form
- OpenGL configurations are also represented by objects:
  - ***Vertex Array Objects***: configure the **input** of the pipeline
  - ***Shader Objects***: represents a single shader of a programmable stage
  - ***Shader Program Objects***: represents all the shaders used in the pipeline

OpenGL Object

- OpenGL objects are ***identified using a GLuint***, an unsigned integer type
- Manipulation of the object is done by calling OpenGL APIs with the GLuint identifier
- These objects would be ***bound to the OpenGL pipeline to configure it***. The programmer should configure the pipeline correctly and issue ***Draw Commands*** to start a rendering operation

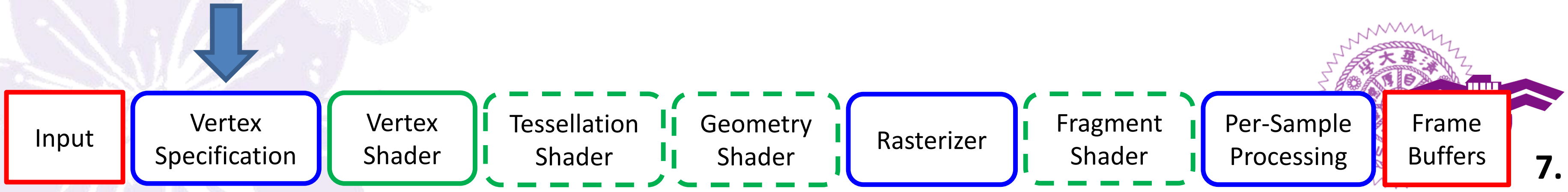
OpenGL Rendering Pipeline

- A common rendering flow looks like:
  - **Configure** the pipeline for Object#1
  - Issue **Draw Command** for Object#1
  - **Configure** the pipeline for Object#2
  - Issue **Draw Command** for Object#2
  - …
- A common consideration is, if two or more objects ***share some same pipeline configurations***, then ***we don't need to modify the configurations*** when rendering the objects consecutively. This can save OpenGL API calls, which directly reduces CPU works

Following the Pipeline

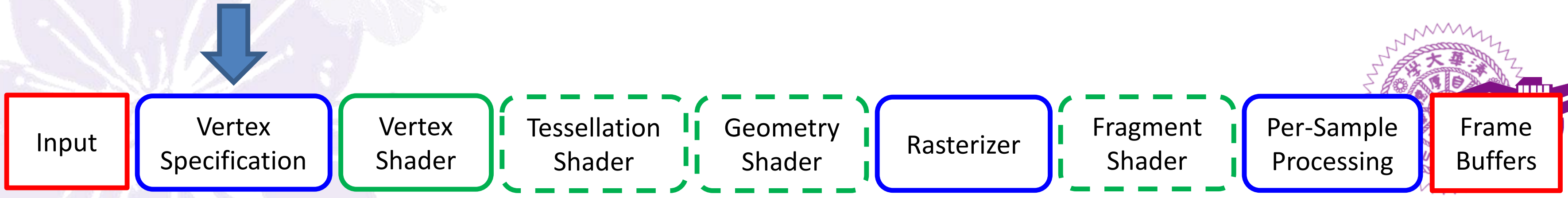
Vertex Specification

- First stage of rendering pipeline
- Fixed function stage
- Assembly ***vertices*** from one or more ***Buffer Objects*** for later stages to consume
- The application provides a ***Vertex Array Object*** as its configuration
- Work with different ***Draw Commands*** to create various input formats to the pipeline



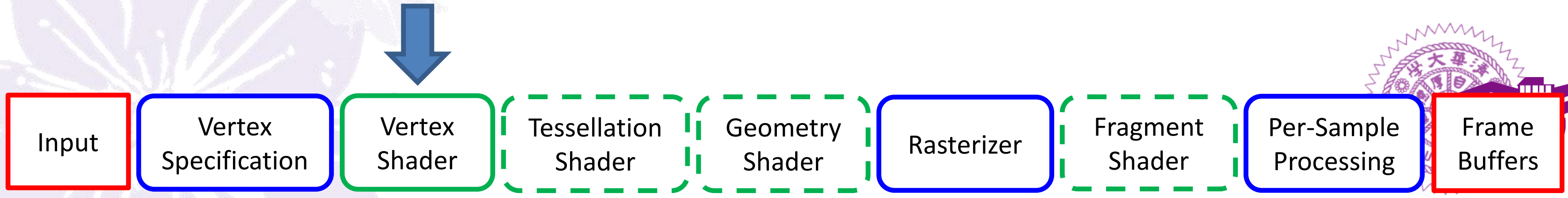
Vertex Specification: APIs

- This stage is mostly configured, but not all, by:
  - glBindVertexArray(vao)
  - glDraw* functions
    - **Draw Command** functions
    - These APIs also mark the beginning of a rendering operation
    - glDrawArrays, glDrawElements, …



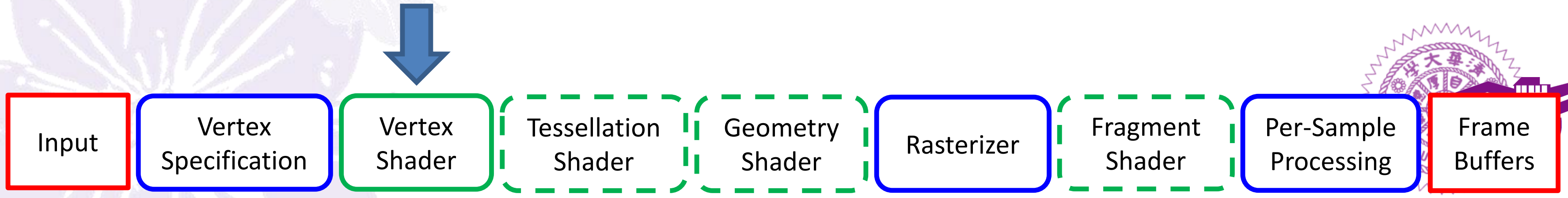
Vertex Shader Stage

- First ***programmable*** stage
- ***Invoked once for each vertex*** of the vertex stream produced by the Vertex Specification
- Isolated from one another, ***unaware of the primitive topology***



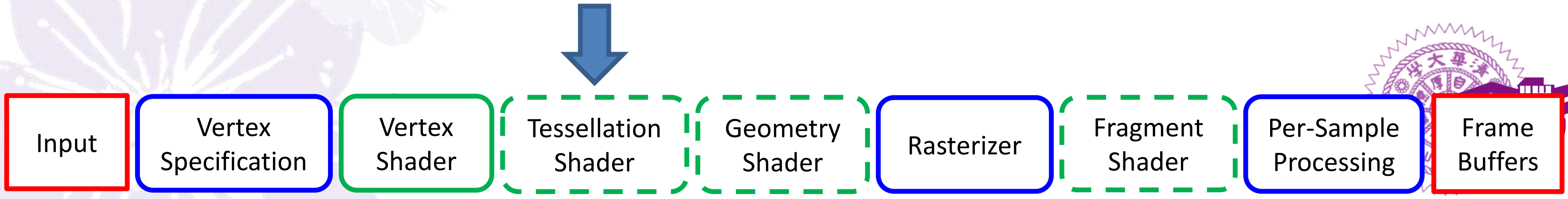
Vertex Shader Stage: Operations

- Apply transform matrices
- Vertex skinning (skeleton-based animations)
- Per-vertex lighting calculations
- Any operations related to the geometry (shape) of the rendered object



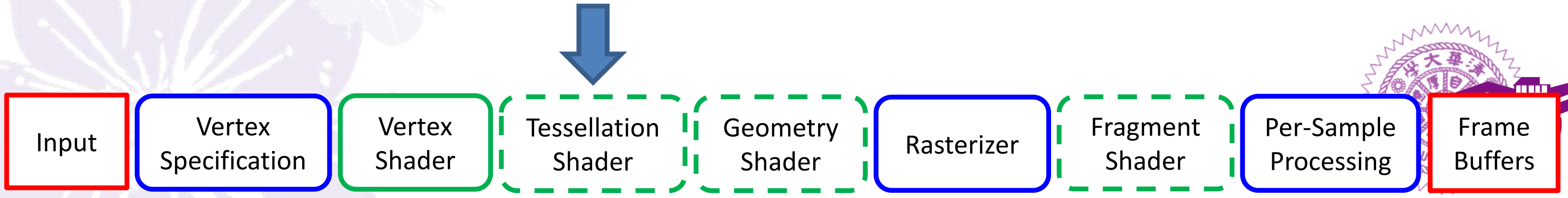
Tessellation Shader Stage

- Optional stage. Can be turned off
- Three sub-stages:
  1. Tessellation Control Shader Stage
  2. Tessellation Engine (fixed function stage)
  3. Tessellation Evaluation Shader Stage
- Three sub-stages must be used together, or none at all



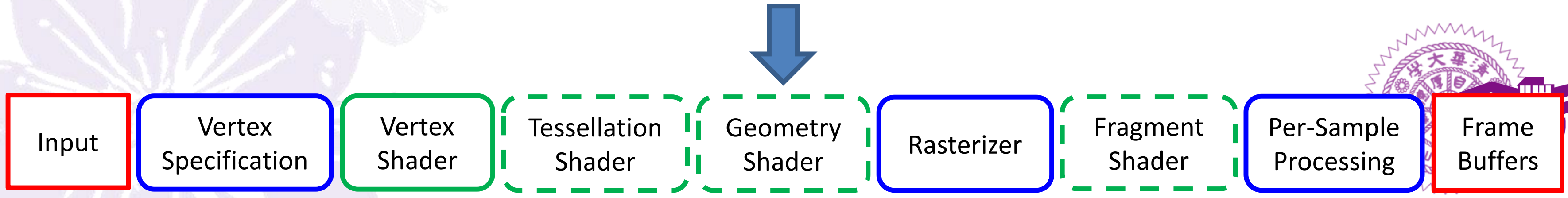
Tessellation Shader Stage

- Map ***higher-order patches*** that represent smooth, curved surfaces to conventional ***triangle-based*** raster hardware
  - Common misunderstanding: tessellation = triangulation (wrong)
- Demand for increasing image quality
  - High resolution models (1 character = 100K vertices)
  - Greatly increases on-disk, in-memory storage, I/O bandwidth and calculations
  - Scale geometry detail between different hardware from a fixed input
  - Relief the artists from participation in triangle-based mesh implementation details



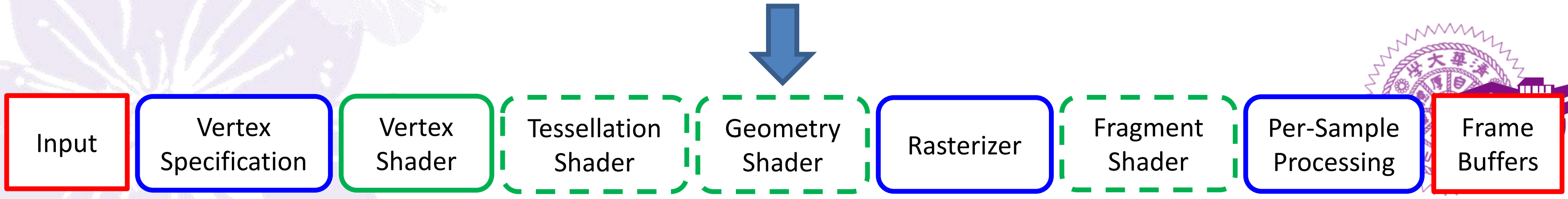
Geometry Shader Stage

- The last stage that can ***manipulate the geometry property*** of the rendered object
- ***Invoked once for each primitive*** from the previous stages
- Less common in algorithm designs, probably due to performance issues



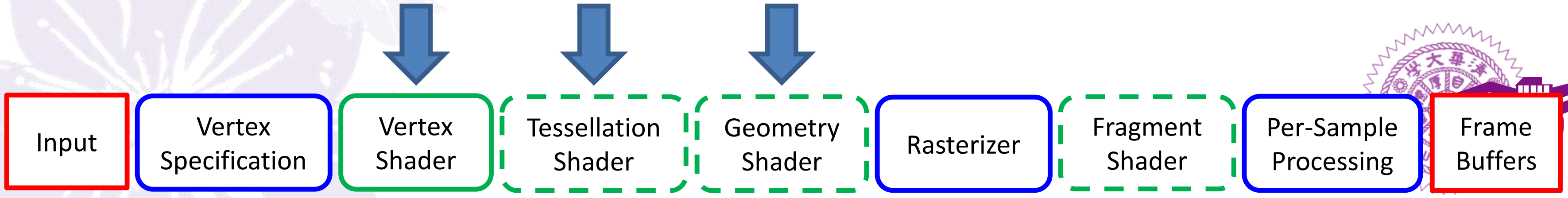
Geometry Shader Stage: Operation

- Programmatically insert/remove geometry
- Produce a different primitive type than is passed to it
- Pass geometry information to buffers through the ***transform feedback*** function
  - ***Transform Feedback***: save geometry to a buffer object from the geometry shader stage
  - Save results generated by expensive vertex/tessellation shader algorithms for cheap reuse
  - Useful when the GPU is used as a co-processor to the CPU. The GPU can process the geometry, then save the result for the CPU to read back



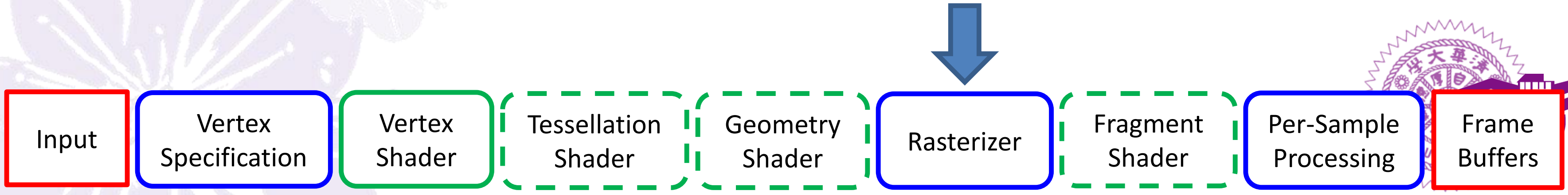
Vertex Processing

- Vertex, Tessellation and Geometry Shader stages are also called **Vertex Processing**
- They all operate on vertices and geometries
- The last active stage of Vertex Processing stages should output a **clip space coordinate** to the Rasterizer stage



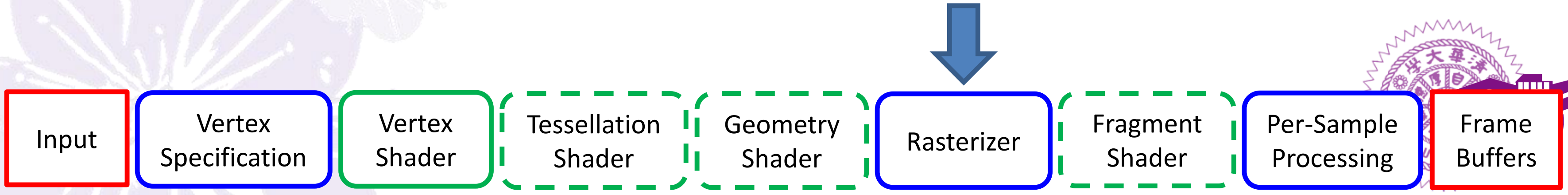
Rasterizer Stage

- Fixed function stage
- Consist of three sub-stages:
  1. Vertex Post-Processing Stage
  2. Primitive Assembly Stage
  3. Rasterization Stage
- Since they are not really separated from one another, and the hardware implementation may differ from this order, for convenience we will just discuss the functions they perform



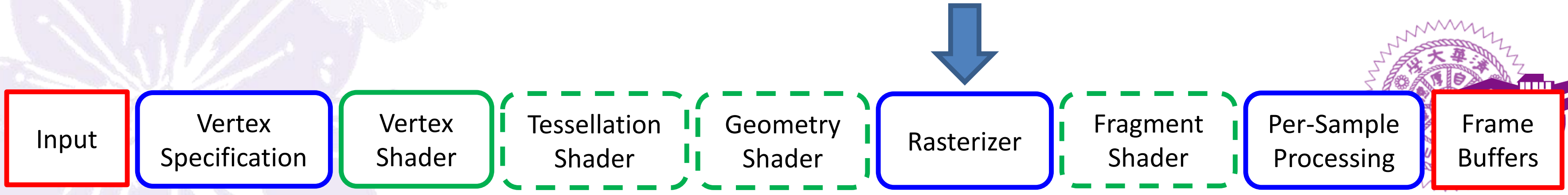
Rasterizer Stage: Operations

- Operations performed, in logical order:
  1. Face Culling
  2. Primitive Culling
  3. Primitive Clipping (or Frustum Clipping)
  4. Perspective Division (or Homogeneous Division)
  5. Viewport Transformation
  6. Rasterization
  7. Multisampling



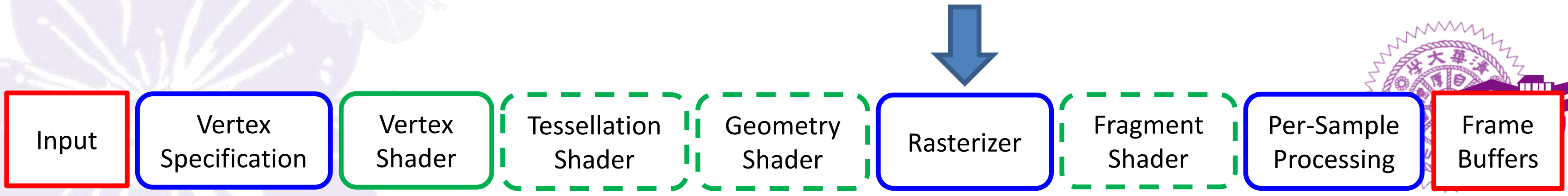
Rasterizer Stage: Operations

- Face Culling
  - Discard a primitive if it's facing towards or away the camera
  - Implement back face culling: for a watertight geometry, it is not possible to see its "inside", so back face will never contribute to the final rendering
- Primitive Culling
  - Discard a primitive if it's completely outside of the view volume
- Primitive Clipping (or Frustum Clipping)
  - If a primitive is partially inside the view volume, then it is split into new primitives that reside completely inside
  - Clip space coordinate make this process very efficient



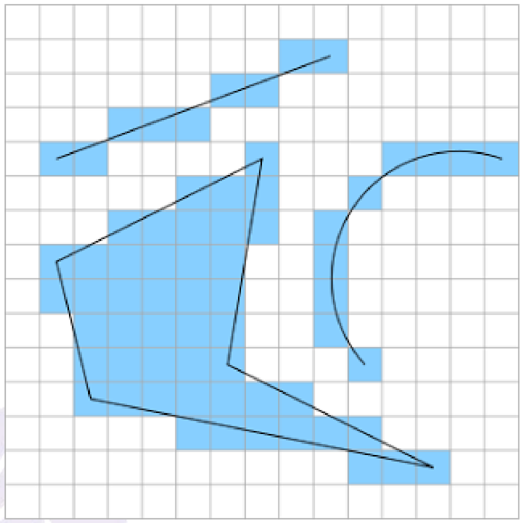
Rasterizer Stage: Operations

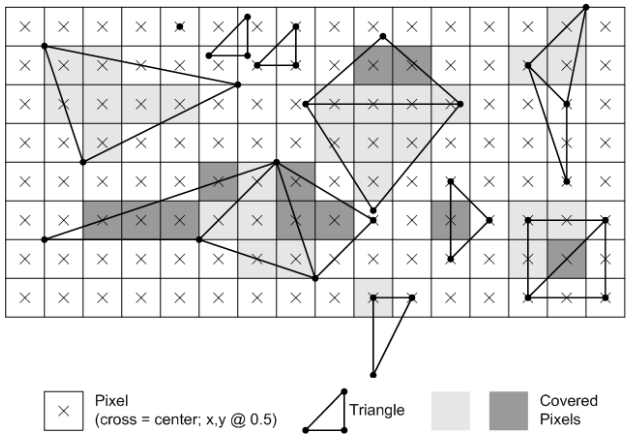
- Perspective Division (or Homogeneous Division)
  - Transform the coordinate from clipping space to NDC
  - Covered in the previous lecture ☺
- Viewport Transformation
  - Transform the coordinate from NDC to screen space
  - Covered in the previous lecture ☺

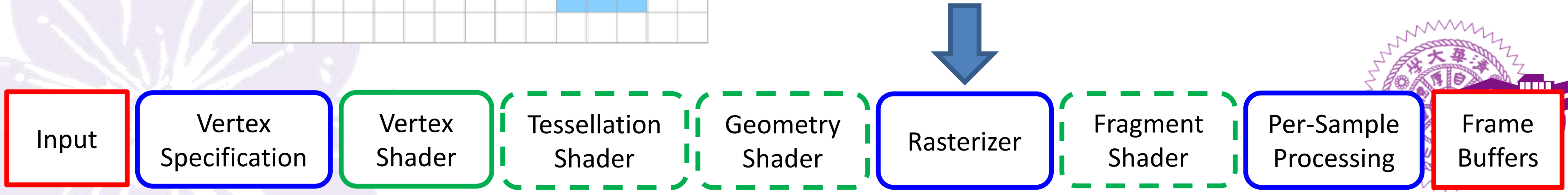


Rasterizer Stage: Operations

- Rasterization
  - Convert the ***geometric*** data into a regularly sampled representation that can be written to ***pixel-based images***, or be displayed on ***pixel-based screens***
  - Please refer to ***Computer Graphics CS 5500*** ☺

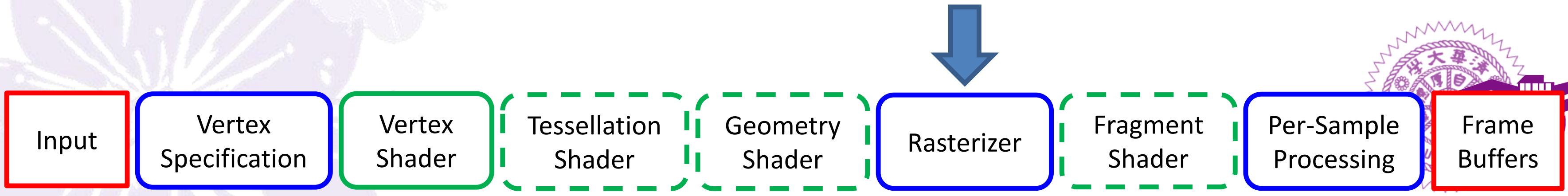






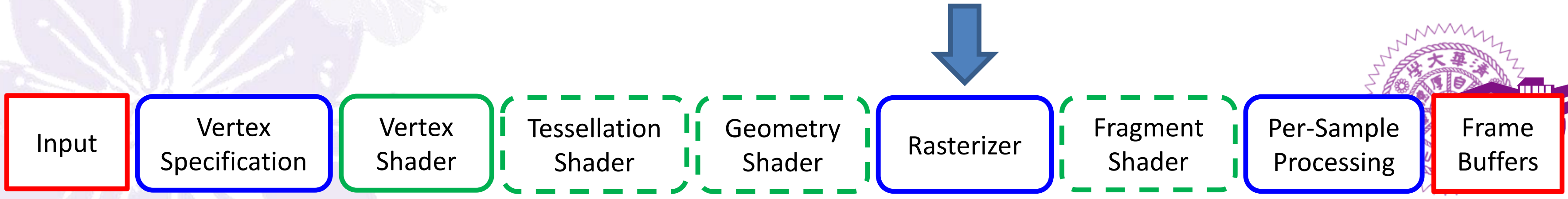
Rasterizer Stage: Operations

- Multisampling
  - A simplified, optimized and hardware-accelerated form of ***anti-aliasing algorithm***
  - We will explain this in a future lecture ☺



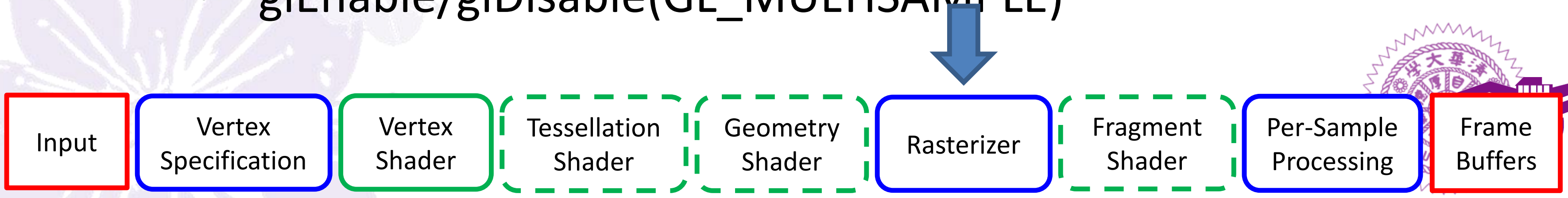
Rasterizer Stage: APIs

- This stage is mostly configured, but not all, by:
  - Primitive Clipping
    - glEnable/glDisable(GL_CLIP_DISTANCEi)
  - Face Culling & Primitive Culling
    - glEnable/glDisable(GL_CULL_FACE)
    - glFrontFace
    - glCullFace
  - Perspective Division
    - None, cannot be controlled



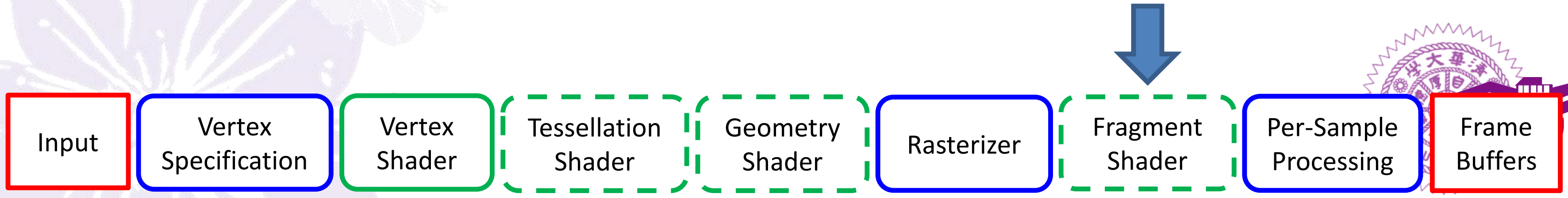
Rasterizer Stage: APIs

- This stage is mostly configured, but not all, by:
  - Viewport Transformation
    - glViewport
    - glDepthRange
  - Rasterization
    - glPolygonMode
    - glEnable/glDisable(GL_POLYGON_OFFSET)
    - glPolygonOffset
    - glPointSize
    - glLineWidth
  - Multisampling
    - glEnable/glDisable(GL_MULTISAMPLE)



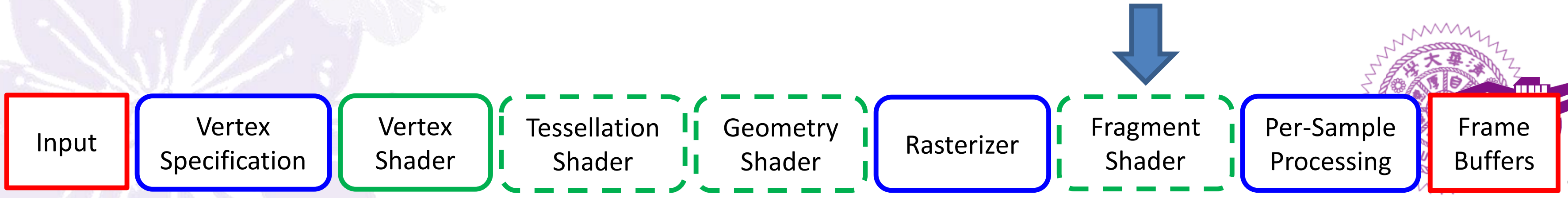
Fragment Shader Stage

- The last programmable stage
- ***Invoked once for each fragment*** generated by the rasterizer
- Unaware of neighboring fragments. Runs in parallel



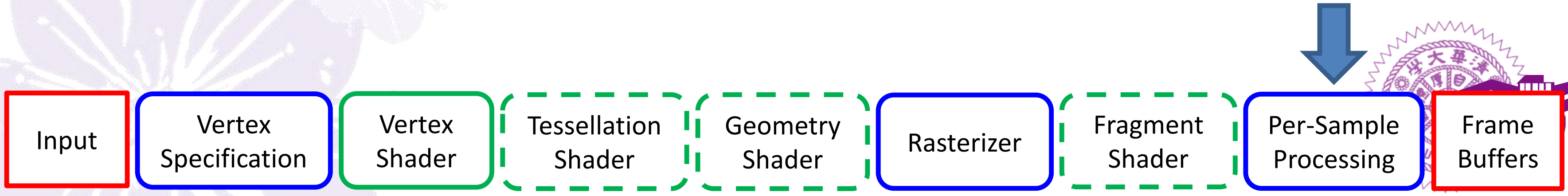
Fragment Shader Stage: Operations

- In plain English: decide the color of a pixel
- Operate at a much higher frequency than the previous stages (1080p=2M pixels)
- Commonly used to add all the details to a rendering
- Perform lighting calculation and texturing operations



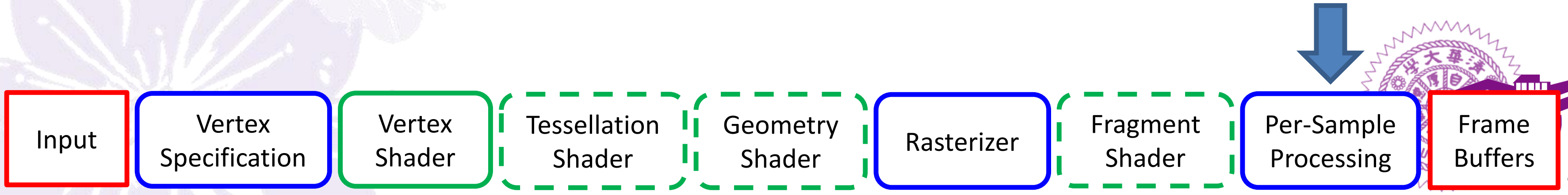
Per-Sample Processing Stage

- The last stage before a fragment is written to the framebuffer
- Fixed function stage
- Perform three kind of tests:
  - **Depth Test**
  - **Stencil Test**
  - **Scissor Test**
- If a fragment passes all the tests, it will be written to the framebuffer. If **Blending** is enabled, blending is performed before the fragment is written



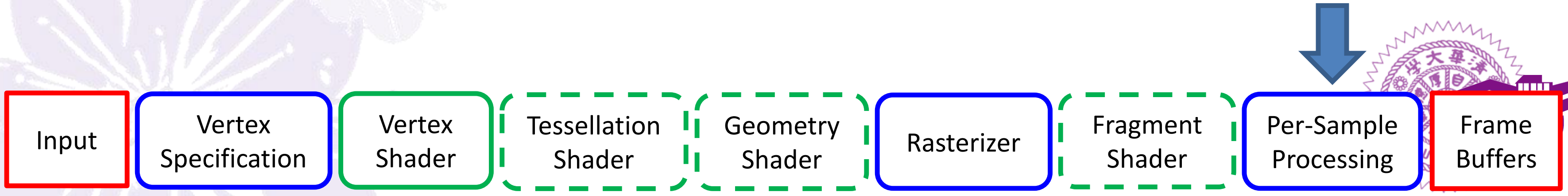
Per-Sample Processing Stage: Operations

- Scissor Test
  - Discard the fragments that are outside a user-specified rectangle
  - Useful for masking the output to a specific area
- Stencil Test
  - Compare a reference value of incoming fragment with the contents of the ***stencil buffer***
  - The content of the stencil buffer can also be updated dynamically
  - Like scissor test, useful for masking the output, but with more flexibility in algorithm design



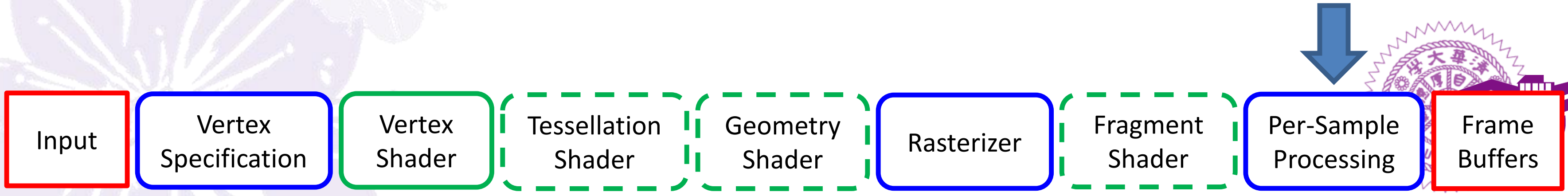
Per-Sample Processing Stage: Operations

- Depth Test
  - Hardware implementation of the classical Z-buffer algorithm [Williams, 1978] with ***depth buffer*** to give the scene a correct depth order
  - Determines whether an incoming fragment can override or blend with the contents of the color buffer
- Blending
  - Traditionally used for alpha-transparency rendering
  - Blend two color sources with a blending function. Sources and blending function are configurable

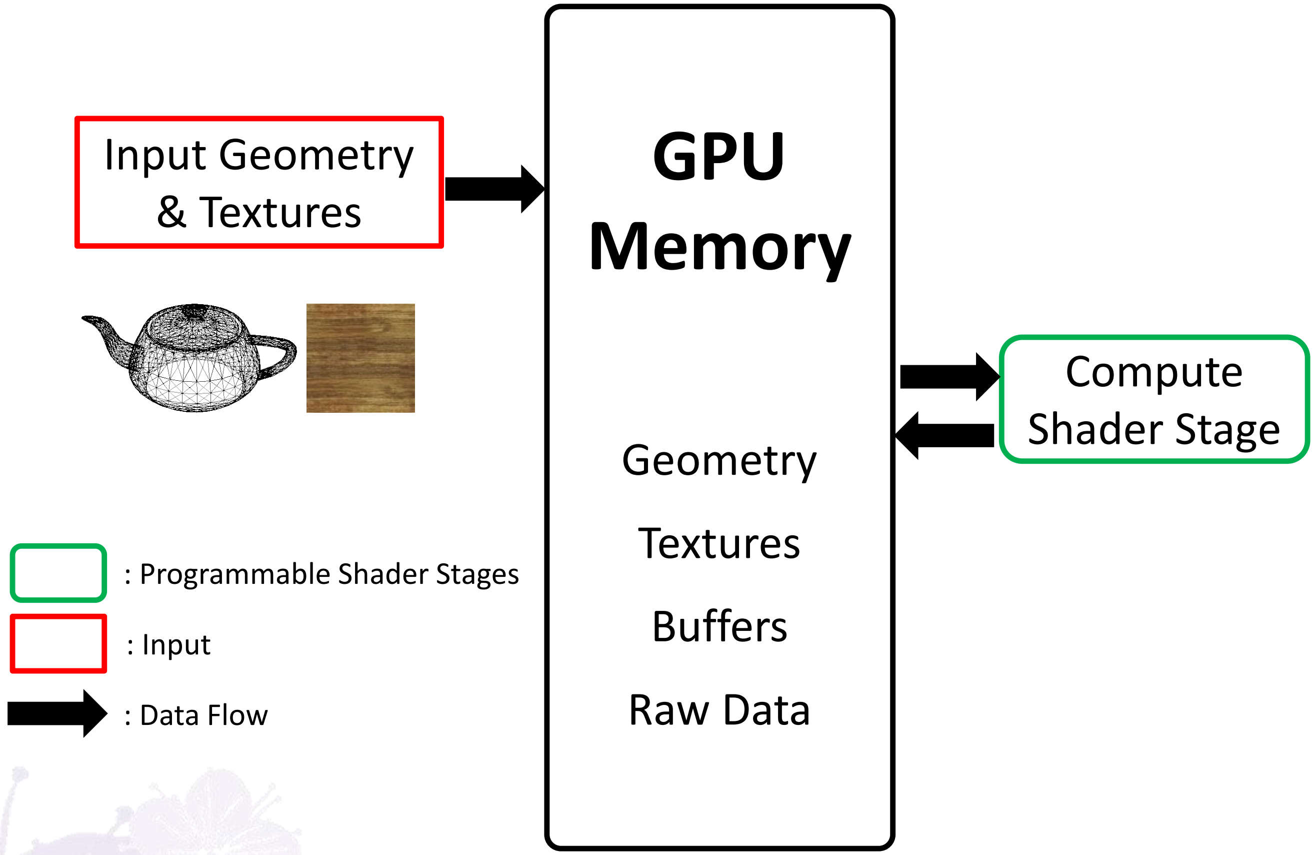


Per-Sample Processing Stage: APIs

- This stage is mostly configured, but not all, by:
  - Depth Test
    - glEnable/glDisable(GL_DEPTH_TEST)
    - glDepthMask
    - glDepthFunc
  - Stencil Test
    - glEnable/glDisable (GL_STENCIL_TEST)
    - glStencilOp
    - glStencilFunc
    - glStencilMask
  - Scissor Test
    - glEnable/glDisable (GL_SCISSOR_TEST)
    - glScissor
  - Blend
    - glEnable/glDisable (GL_BLEND)
    - glBlendFunc



The OpenGL Computation Pipeline



Compute Shader

- Available since OpenGL 4.3
- Compute shader pipeline only has one programmable stage
- There are no predefined input or output in compute shader stage. However, Compute shaders can ***read/write buffers and textures asynchronously***, allowing for highly customized, parallel algorithm designs
- Idea close to CUDA or DirectCompute
- Compute shader is built upon the same foundation as the rendering pipeline stages, meaning it can ***benefit from hardware texture filtering and GLSL intrinsic functions***

Compute Shader: Operations

- Compute shaders are becoming increasingly important in modern rendering algorithms
- ***Physic-based simulations***
  - Water surface or cloth simulation
  - Particle system simulation
  - Audio reverb zone simulation
- Procedural texture generation
- ***Image processing operations***
  - Separated 2D convolution (Gaussian blur, …)
- GPGPU operations
  - Act as a co-processor to the CPU
  - General parallel algorithm designs

The First OpenGL Shader Program Example: Single Point

Example Programs

- You can find the example programs of this course on the ILMS!
- For example, find Example: Single Point here:
  - Lecture Programs/VC10/Lecture 01/Lecture01.sln
- All the examples today are in Lecture 01 ☺

A Simple Vertex Shader

```
// tell shader compiler to use version 4.1 of GLSL
// and use only features from the core profile.
#version 410 core

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

A Simple Fragment Shader

```
#version 410 core

// color is the output value of this fragment shader
out vec4 color;

void main()
{
    color = vec4(0.0, 0.8, 1.0, 1.0);
}
```

gl_Position

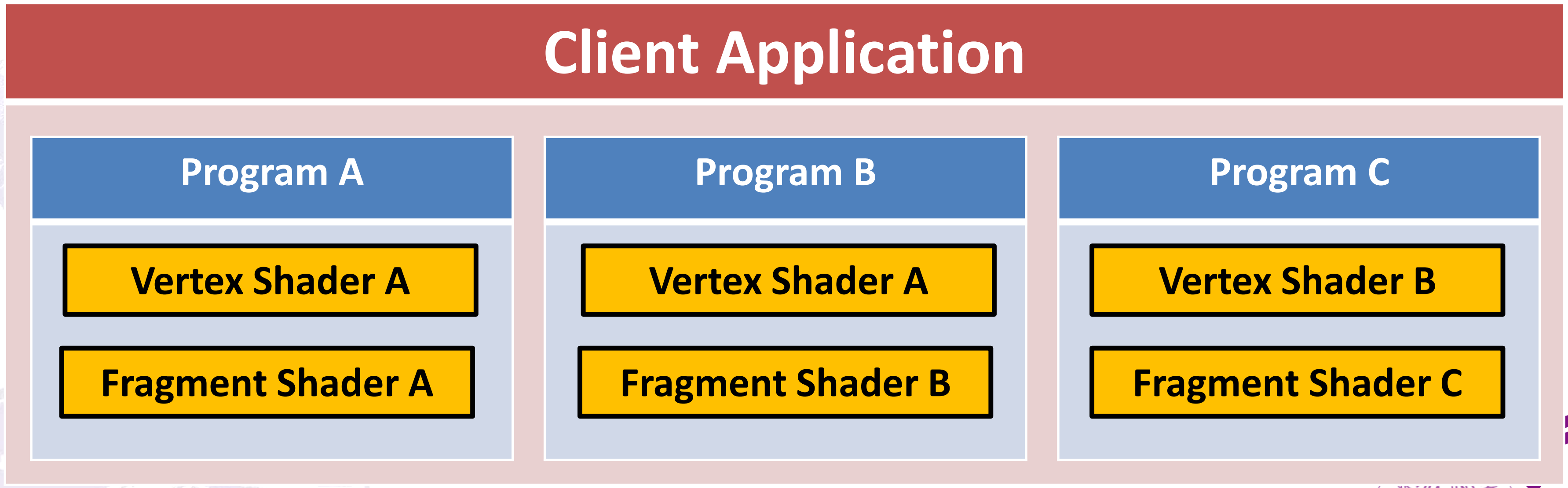
- Represent the position of the current vertex.
- A member of the gl_PerVertex named block:

```
out gl_PerVertex {
    vec4 gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[];
};
```

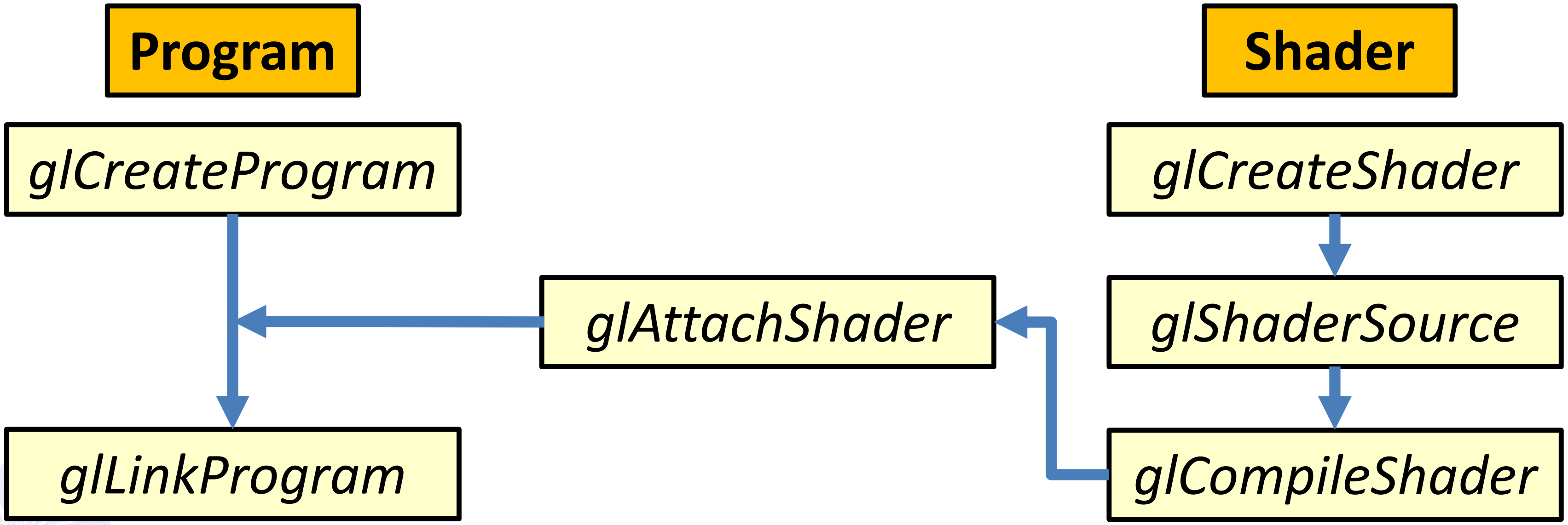
- **Description**: These variables only take on the meanings below if this shader is the last active Vertex Processing (from vertex shader to geometry shader) stage and is an output that receives the homogeneous vertex position. It may be written at any time during shader execution.

Shader and Shader Program

- A ***shader program object*** is an object to which ***shader objects*** can be attached
- An application would contain more than one programs for rendering operations



GLSL Architecture



If you want to use two or more different programs, you have to create and link the separated programs

glShaderSource

```
void glShaderSource( GLuint shader, GLsizei count,
                                   const GLchar **string, const GLint *length);
```

- **Shader**: Specifies the handle of the shader object whose source code is to be replaced.
- **Count**: Specifies the number of elements in the string and length arrays.
- **String**: Specifies an array of pointers to strings containing the source code to be loaded into the shader.
- **Length**: Specifies an array of string lengths.

glShaderSource (Cont'd)

void glShaderSource(GLuint shader, GLsizei count,
*const GLchar **string, const GLint *length);*

- **Description**: glShaderSource sets the source code in shader to the source code in the array of strings specified by string. Any source code previously stored in the shader object is completely replaced. The number of strings in the array is specified by count. If length is NULL, each string is assumed to be **null terminated.**

glCreateProgram

```
GLuint glCreateProgram(void);
```

- **Description**: glCreateProgram creates an empty program object and returns a non-zero value by which it can be referenced. A program object is an object to which shader objects can be attached. When no longer needed as part of a program object, shader objects can be detached.

glAttachShader

```
void glAttachShader (GLuint program, GLuint shader);
```

- **program**: Specifies the program object to which a shader object will be attached.
- **shader**: Specifies the shader object that is to be attached.
- **Description**: Shaders that are to be linked together in a program object must first be attached to that program object. ***glAttachShader*** attaches the shader object specified by shader to the program object specified by program. This indicates that shader will be included in link operations that will be performed on program.

glLinkProgram

```
void glLinkProgram (GLuint program);
```

- **program**: Specifies the handle of the program object to be linked.
- **Description**: glLinkProgram links the program object specified by program. If any shader objects of type are attached to program, they will be used to create an executable that will run on the programmable fragment processor.

Compiling Shader Code

```
static const char *vs_source[] =
{
    "#version 410 core                                   \n"
    "                                                    \n"
    "void main()                                         \n"
    "{                                                   \n"
    "     gl_Position = vec4(0.0, 0.0, 0.0, 1.0);        \n"
    "}                                                   \n"
};

static const char *fs_source[] =
{
    "#version 410 core                                   \n"
    "                                                    \n"
    "out vec4 color;                                     \n"
    "                                                    \n"
    "void main()                                         \n"
    "{                                                   \n"
    "     color = vec4(0.0, 0.8, 1.0, 1.0);              \n"
    "}                                                   \n"
};
```

Compiling Shader Code (Cont’d)

```
GLuint setup_shaders()
{
    GLuint    vertex_shader;
    GLuint    fragment_shader;
    GLuint    program;

    // create and compile the vertex shader
    vertex_shader = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vertex_shader, 1, vs_source, NULL);
    glCompileShader(vertex_shader);

    // create and compile the fragment shader
    fragment_shader = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(fragment_shader, 1, fs_source, NULL);
    glCompileShader(fragment_shader);
```

Compiling Shader Code (Cont'd)

```
    // create a program, attach shaders to it, and link it
    program = glCreateProgram();
    glAttachShader(program, vertex_shader);
    glAttachShader(program, fragment_shader);
    glLinkProgram(program);

    // delete the shaders as the program has them now
    glDeleteShader(vertex_shader);
    glDeleteShader(fragment_shader);

    return program;
}
```

Shader and Shader Program

- Compiling shaders is a CPU-intensive operation. In modern game engines, there are usually **hundreds or thousands** of shaders to compile
- A shader object can be **reused** in different shader programs, as long as the input/output/globals matches between every programmable stages, to reduce the required shader objects
- Unless necessary, **all shaders and shader programs should be loaded at start up**

Vertex Array Object (VAO)

- Represents the **input configuration** of the Vertex Specification stage
- Defines a **input attribute layout** for the Vertex Specification stage to assembly Vertex Shader stage inputs from buffers
- VAOs should be pre-created for each input configurations, and bound to the pipeline before draw command calls
- **Required for rendering operations, even if there is no input at all**

glGenVertexArrays

```
void glGenVertexArrays(GLsizei n, GLuint *array);
```

- **Description**: glGenVertexArrays generates “*n*” vertex array objects and stores their identities (or names) in the “*array*”.

glBindVertexArray

```
void glBindVertexArray(GLuint array);
```

- **Description**: glBindVertexArray binds the vertex array object with the object with identity “array”. Set the value of “array” to zero to break the existing vertex array object binding.

glUseProgram

```
void glUseProgram(Gluint program);
```

- **Description**: glUseProgram installs the program object specified by "*program*" as part of current rendering state. One or more executables are created in a program object by attaching shader objects to it with glAttachShader, compiling the shader objects with glCompileShader, and linking the program object with glLinkProgram.

glDrawArrays

```
void glDrawArrays(GLenum mode, GLint first, GLsizei count);
```

- **mode**: Specifies what kind of primitives to render. Symbolic constants include GL_POINTS, GL_LINE_STRIP, GL_LINE_LOOP, GL_LINES, GL_LINE_STRIP_ADJACENCY, GL_LINES_ADJACENCY, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_TRIANGLES, GL_TRIANGLE_STRIP_ADJACENCY, GL_TRIANGLES_ADJACENCY and GL_PATCHES.
- **first**: Specifies the starting index in the enabled arrays.
- **count**: Specifies the number of vertices to be rendered.
- **There is NO GL_QUADS after GLSL 3.0!**

glDrawArrays (Cont’d)

```
void glDrawArrays(GLenum mode, GLint first, GLsizei count);
```

- **Description**: *glDrawArrays* constructs a sequence of geometric primitives using array elements starting at “*first*” and ending at *“first + count – 1”* of each enabled array. Mode specifies what kinds of primitives are constructed (e.g., GL_TRIANGLES, GL_LINES, GL_LINE_LOOP, GL_POINTS, etc).

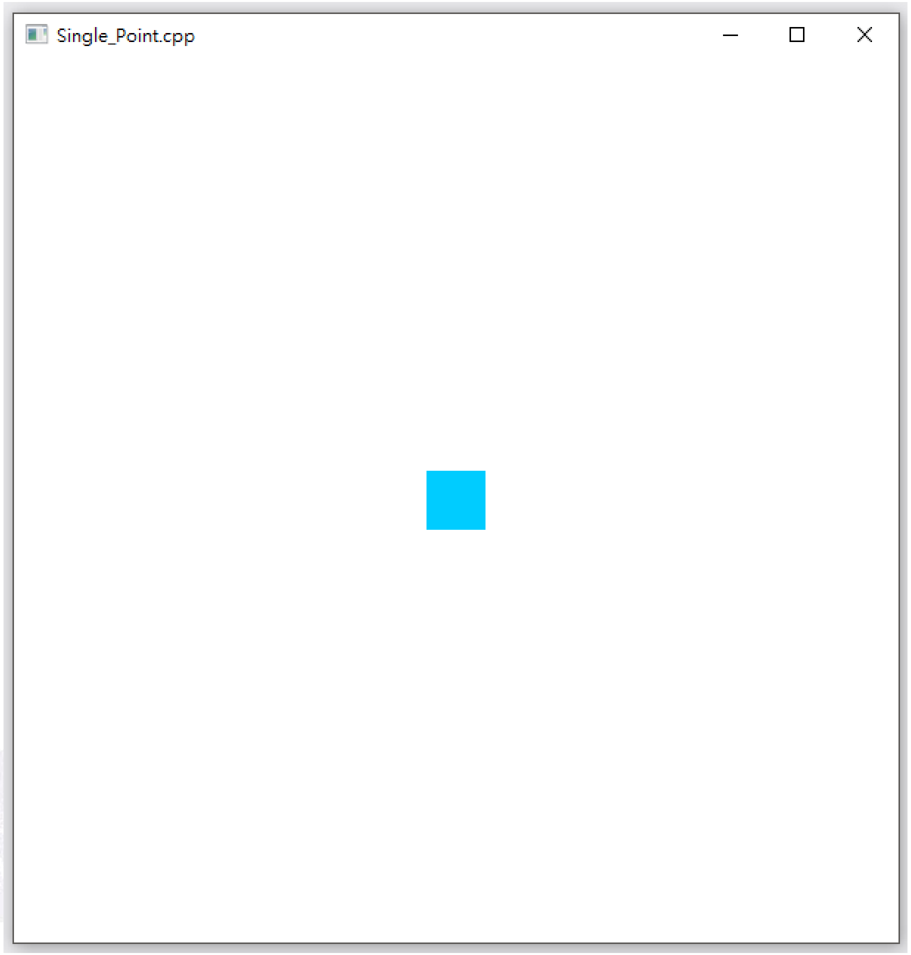
Code: Main Program

```
GLuint rendering_program;
GLuint vertex_array_object;

void startup()
{
    rendering_program = setup_shaders();
    glGenVertexArrays(1, &vertex_array_object);
    glBindVertexArray(vertex_array_object);
}

void shutdown()
{
    glDeleteVertexArrays(1, &vertex_array_object);
    glDeleteProgram(rendering_program);
}
```

Code: Render Function

```
void render()
{
    static const GLfloat white[] = { 1.0f, 1.0f, 1.0f, 1.0f };
    glClearBufferfv(GL_COLOR, 0, white);
    glUseProgram(rendering_program);
    glBindVertexArray(vertex_array_object);
    glPointSize(40.0f);
    glDrawArrays(GL_POINTS, 0, 1);
}
```

Result



Example: Simple Triangle

Code: Vertex Shader

```
#version 410 core

void main()
{
    const vec4 vertices[3] = vec4[3](
        vec4( 0.25, -0.25, 0.5, 1.0),
        vec4(-0.25, -0.25, 0.5, 1.0),
        vec4( 0.25,  0.25, 0.5, 1.0));
    gl_Position = vertices[gl_VertexID];
}
```

gl_VertexID

- A built-in variable of the vertex shader
- Contains the index of the current vertex
- **Declaration**:
  - in int gl_VertexID ;
- **Description**: gl_VertexID is a built-in input variable that holds an integer index for the vertex. The index is implicitly generated by glDrawArrays and other commands

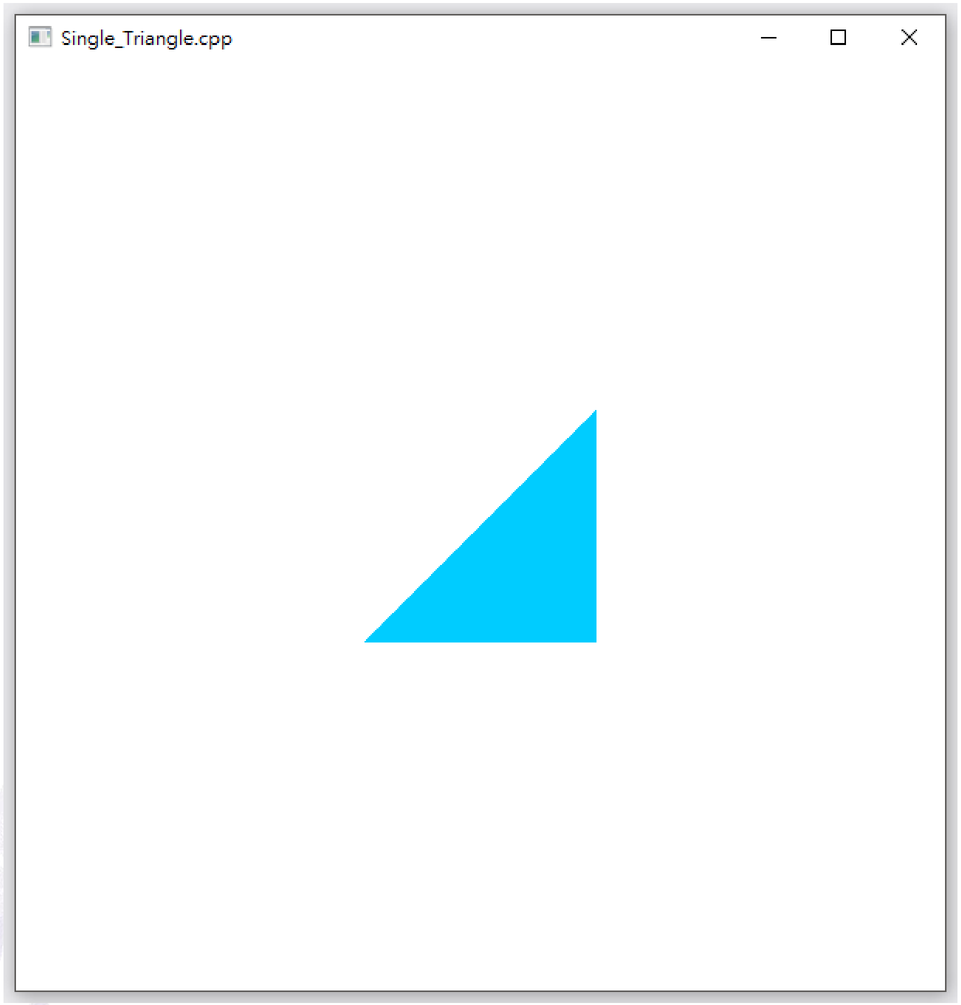
Code: Fragment Shader

```
#version 410 core

// output of this fragment shader
out vec4 color;

void main()
{
    color = vec4(0.0, 0.8, 1.0, 1.0);
}
```

Code: Render Function

```
void render()
{
    static const GLfloat color[] =
        { 1.0f, 1.0f, 1.0f, 1.0f };
    glClearBufferfv(GL_COLOR, 0, color);
    glUseProgram(rendering_program);
    glDrawArrays(GL_TRIANGLES, 0, 3);
}
```

Result



Example: Simple Triangle Sin Colored

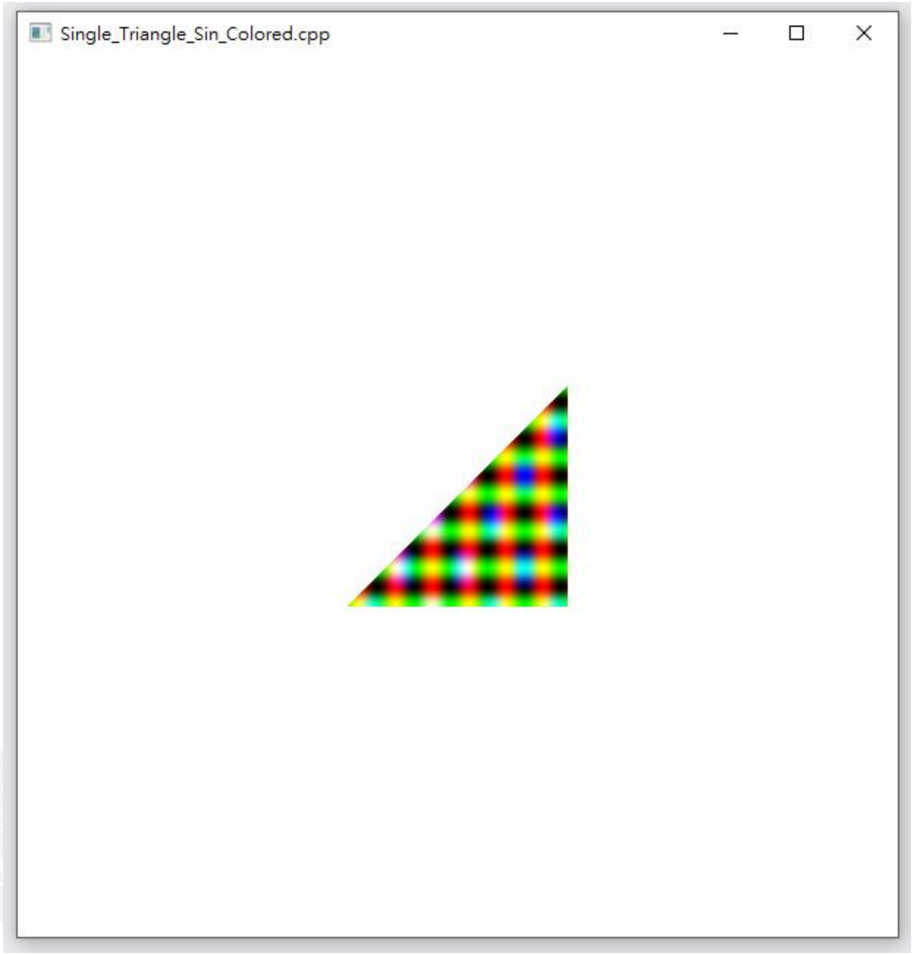
Code: Fragment Shader

```
#version 410 core

// color is the output color of this fragment shader
out vec4 color;

void main()
{
    // compute color based on the fragment’s screen coordinate
    color = vec4(
        sin(gl_FragCoord.x * 0.25) * 0.5 + 0.5,
        cos(gl_FragCoord.y * 0.25) * 0.5 + 0.5,
        sin(gl_FragCoord.x * 0.15) * cos(gl_FragCoord.y * 0.15),
        1.0);
}
```

gl_FragCoord

- Contains the window-relative coordinates of the current fragment
- **Declaration**:
  - in vec4 gl_FragCoord ;
- **Description**: Available only in the fragment language, gl_FragCoord is an input variable that contains the window relative coordinate (x, y, z, 1/w) values for the fragment

Result



Example: Vertex Attribute Variable

In/Out Storage Qualifiers

- In GLSL, the mechanism for getting data in and out of shaders is to declare global variables with the ***in*** and ***out*** storage qualifiers.
- Between stages, ***in*** and ***out*** can be used to form conduits from shader to shader and pass data between them.

Vertex Attributes

- A variable declared with ***in*** storage qualifier in the vertex shader. It's value is automatically filled in by the ***vertex specification*** stage.

```
#version 410 core

// 'offset' is an input vertex attribute
layout (location = 0) in vec4 offset;

void main(void)
{
     const vec4 vertices[3] = vec4[3](
          vec4( 0.25, -0.25, 0.5, 1.0),
          vec4(-0.25, -0.25, 0.5, 1.0),
          vec4( 0.25,  0.25, 0.5, 1.0));
   // Add 'offset' to our hard-coded vertex position
   gl_Position = vertices[gl_VertexID] + offset;
}
```

glVertexAttrib

```
void glVertexAttrib{1234}{fds}(GLuint index, TYPE values);
void glVertexAttrib{1234}{fds}v(GLuint index,const TYPE *values);
void glVertexAttrib4{bsifd ub us ui}v(GLuint index,const TYPE
*values);
```

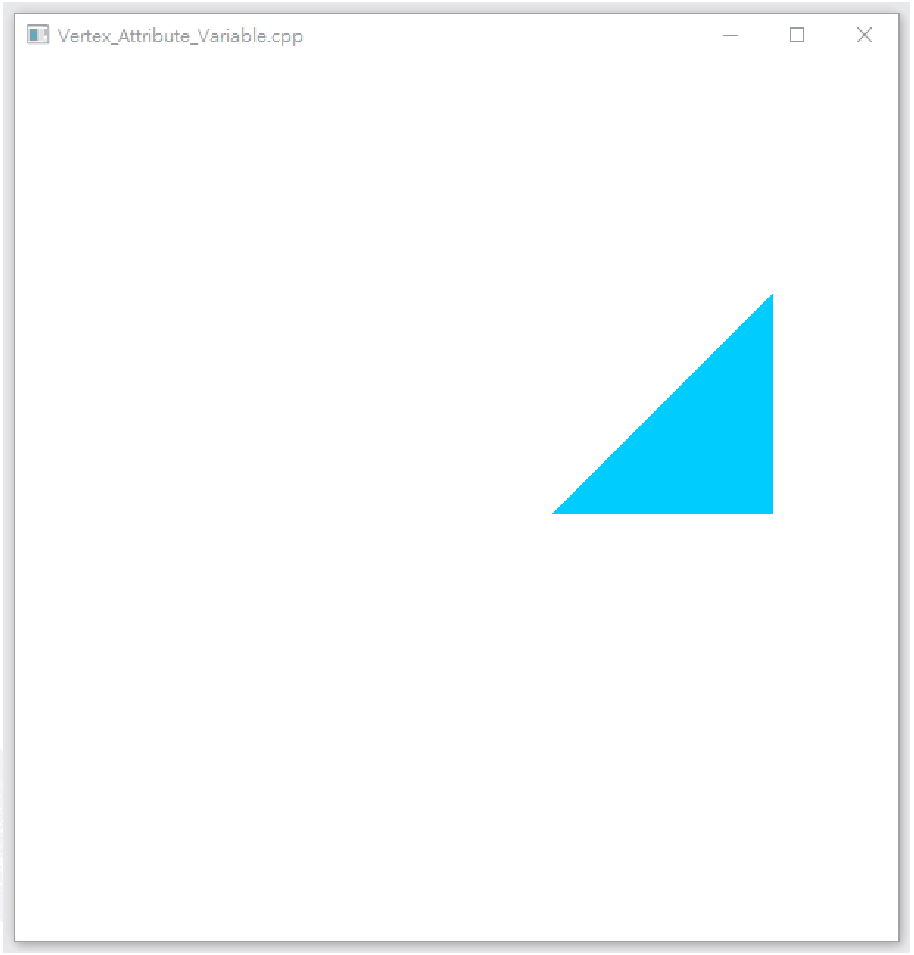
- **index**: Specifies the index of the generic vertex attribute to be modified.
- **values**: For the packed commands, specifies the new packed value to be used for the specified vertex attribute.

Vertex Attributes: Vertex Fetching

```
void render(double currentTime)
{
    static const GLfloat color[] = { 1.0f, 1.0f, 1.0f, 1.0f };
    glClearBufferfv(GL_COLOR, 0, color);
    glUseProgram(rendering_program);

    GLfloat attrib[] = {  (float)sin(currentTime) * 0.5f,
                          (float)cos(currentTime) * 0.6f,
                          0.0f, 0.0f};
   // Update the value of input attribute 0
   glVertexAttrib4fv(0, attrib);

   glDrawArrays(GL_TRIANGLES, 0, 3);
}
```

```
layout (location = 0) in vec4 offset;
```

Results



Passing Data Between Stages

User-Defined Attributes

- Declare two variables with similar names and different in/out storage qualifiers in different stages. These are called ***user-defined attributes***
- Declare a variable ***s1_v*** with ***out*** keyword in the stage s1
- Declare a variable ***s1_v*** with ***in*** keyword in the stage s2
- Anything you write to ***s1_v*** stage s1 will be sent to the next stage s2

Rasterizer and User-Defined Attributes

- The inputs to the fragment shader are somewhat unlike inputs to other shader stages, which is vertex-based
- The rasterizer stage ***interpolates*** the user attributes across the primitive that's being rasterized

Code: Vertex Shader

```
#version 410 core

// “offset” and “color” are two input vertex attributes
layout (location = 0) in vec4 offset;
layout (location = 1) in vec4 color;

// “vs_color” is an output that will be sent to the next shader stage
out vec4 vs_color;

void main()
{
    const vec4 vertices[3] = vec4[3](
        vec4( 0.25, -0.25, 0.5, 1.0),
        vec4(-0.25, -0.25, 0.5, 1.0),
        vec4( 0.25,  0.25, 0.5, 1.0));
    gl_Position = vertices[gl_VertexID] + offset;
    // output the input “color” attribute
    vs_color = color;
}
```

Code: Fragment Shader

```
#version 410 core

// interpolated input from the rasterizer
in vec4 vs_color;

// color is the output color of this fragment shader
out vec4 color;

void main()
{
    // simply output the interpolated value
    color = vs_color;
}
```

Interface Blocks

- Declaring interface variables one at a time is possibly the simplest way to communicate data between shader stages
- In most scenarios, we want to communicate a number of different data between stages
- The declaration of an ***interface block*** looks a lot like a structure declaration to group together a number of variables

Code: Vertex Shader

```
#version 410 core

// “offset” and “color” are two input vertex attributes
layout (location = 0) in vec4 offset;
layout (location = 1) in vec4 color;

// declare “VS_OUT” as an output interface block
out VS_OUT
{
    vec4 color; // send color to the next stage (rasterizer)
} vs_out;

void main(void)
{
    const vec4 vertices[3] = vec4[3](
        vec4( 0.25, -0.25, 0.5, 1.0),
        vec4(-0.25, -0.25, 0.5, 1.0),
        vec4( 0.25,  0.25, 0.5, 1.0));
    gl_Position = vertices[gl_VertexID] + offset;
    vs_out.color = color;
}
```

Code: Fragment Shader

```
#version 410 core

// same “VS_OUT” interface block as in the vertex shader
in VS_OUT
{
    vec4 color; // value received from the rasterizer
} fs_in;

// color is the output color of this fragment shader
out vec4 color;

void main()
{
    // output the interpolated color value
    color = fs_in.color;
}
```

Example: Simple Triangle RGB

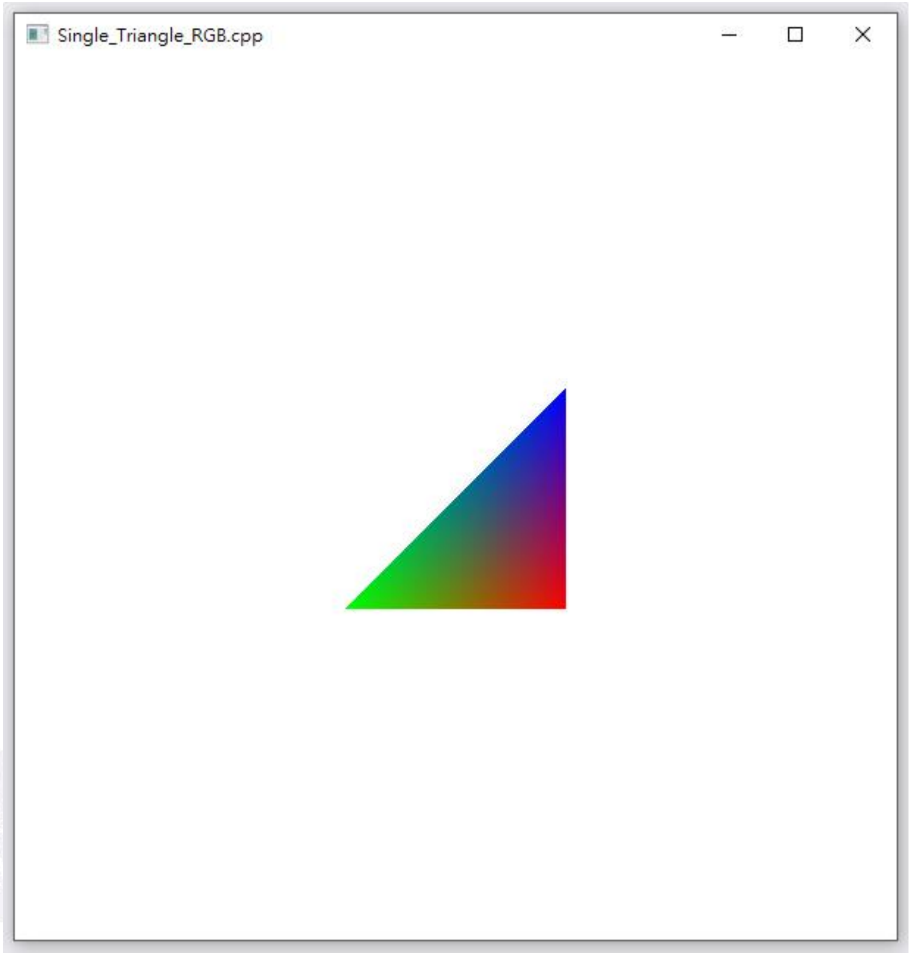
Code: Vertex Shader

```
#version 410 core

// vs_color is an output user-defined attribute
out vec4 vs_color;

void main()
{
    const vec4 vertices[3] = vec4[3](
        vec4( 0.25, -0.25, 0.5, 1.0),
        vec4(-0.25, -0.25, 0.5, 1.0),
        vec4( 0.25,  0.25, 0.5, 1.0));
    const vec4 colors[3] = vec4[3](
        vec4(1.0, 0.0, 0.0, 1.0),
        vec4(0.0, 1.0, 0.0, 1.0),
        vec4(0.0, 0.0, 1.0, 1.0));

    // output vertex data with gl_VertexID = 0,1,2
    gl_Position = vertices[gl_VertexID] ;
    vs_color = colors[gl_VertexID];
}
```

Code: Fragment Shader

```
#version 410 core

// vs_color is the user-defined attribute
// interpolated by the rasterizer
in vec4 vs_color;

// color is the output color of this fragment shader
out vec4 color;

void main()
{
    color = vs_color;
}
```

Result


Single_Triangle_RGB.cpp