Getting Started with OpenGL: Hello Triangle

Computer Graphics
Instructor: Sungkil Lee

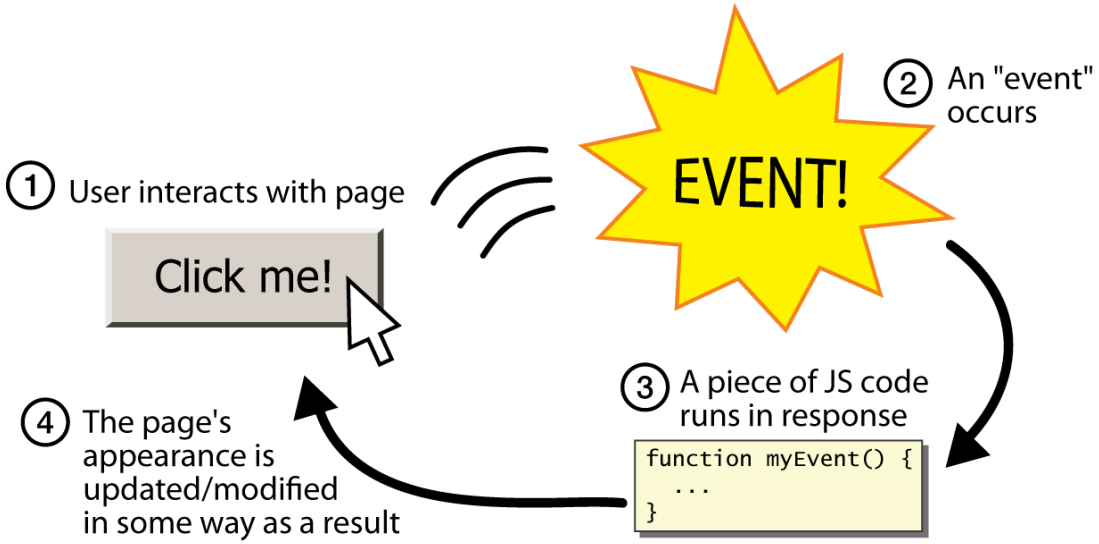
Today

- **Draw a colored triangle on the screen in a window.**

Background and Prerequisites

Event-Driven Programming

- **OpenGL rendering is performed event-driven way**
  - Similar to GUI programming (e.g., Windows API, QT, Web)
  - After initializations, the program enters an infinite event loop.
  - The program is terminated with a TERM signal that user sent.
  - See an example of Javascript-driven web programming



Event-Driven Programming

- **An OpenGL program has a similar structure.**
  - GLFW registers callback functions for various window-related events.
  - Responding to each event, GLFW tries to call its registered callback function.

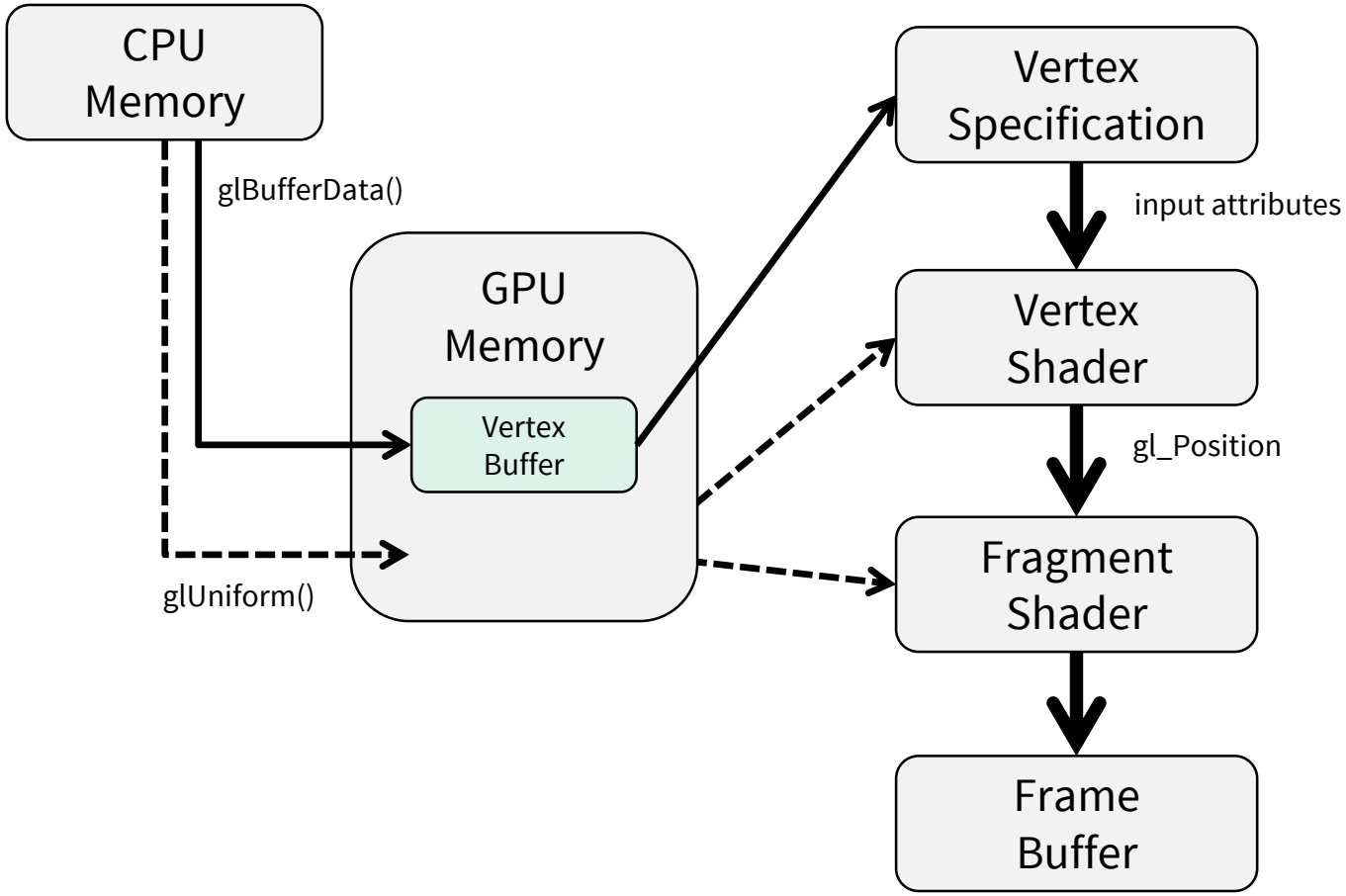
```
// event-driven callback functions
void update(){ ... }     // callback for per-frame update
void render(){ ... }     // callback for per-frame rendering
void reshape(){ ... }    // callback for window resizing
void keyboard(){ ... }   // callback for keyboard input
void mouse(){ ... }      // callback for mouse clicks
void motion(){ ... }     // callback for mouse movements

// init functions
void init_shaders(){ ... }
void user_init(){ ... }

// main function
int main(){ ... }
```

Rendering Modes for OpenGL

- **Retained mode (modern style: better in performance)**
  - Send arrays (e.g., vertex buffers) over and store on GPU
  - Reuse them for multiple renderings

- **Immediate mode (old style)**
  - Put all vertex and attribute data in arrays
  - Send the arrays to GPU to be rendered immediately
  - We would have to send the array over each time when we need another render of it.
  - ***Do not use the immediate mode for this course***.
    - If you use the old-style GL, you are supposed to do cheating!

OpenGL Pipeline in Retained Mode



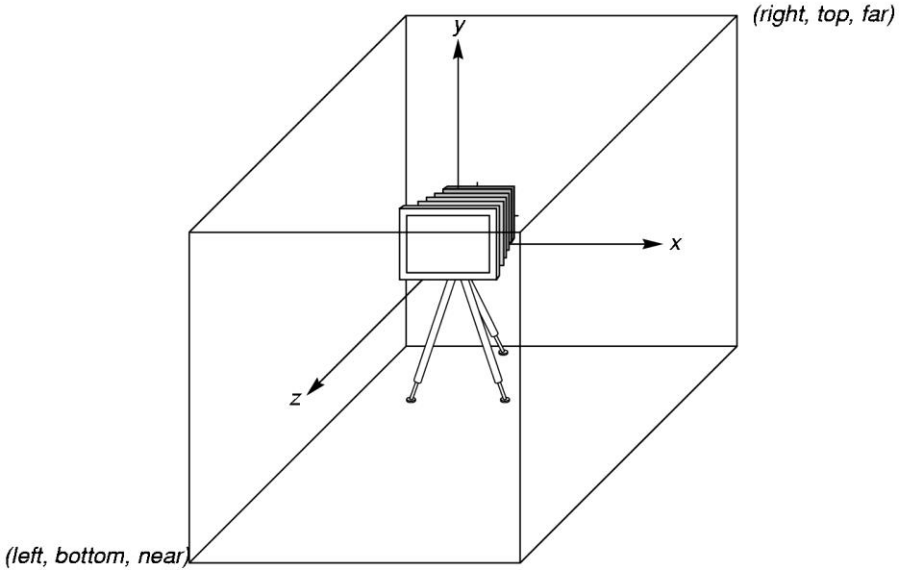
OpenGL as State Machine

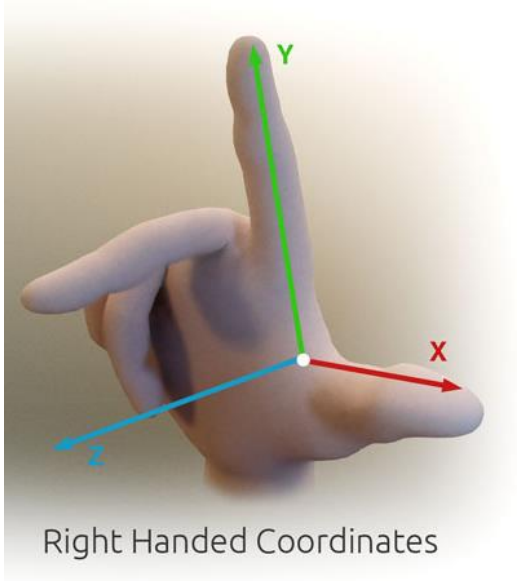
- **OpenGL maintains many global states (actually, variables) internally.**
  - OpenGL does not have object management, because it had been introduced before object-oriented scheme (e.g., from IrisGL).
  - So, many states associated with the current OpenGL context are internally managed by OpenGL drivers.
- **Programming for state machine**
  - When you use certain functions, you typically need bind the states.
  - For example, when you update the vertex buffer using glBufferData(), you find call glBindBuffer().
  - For another example, when you use a program, you need to first call glUseProgram().
  - This scheme applies to most of the OpenGL functions.

Coordinate Systems

Coordinate Systems in OpenGL

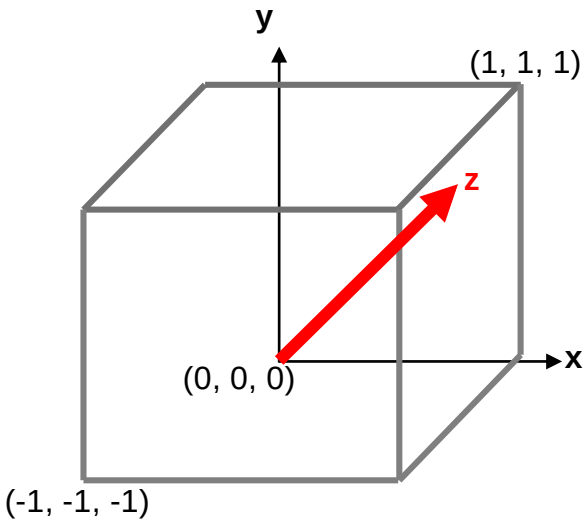
- **Camera (eye-space) coordinate system**
  - Use right-handed coordinate system (**RHS**).
  - OpenGL places a camera at the origin in object space pointing in the negative z direction.

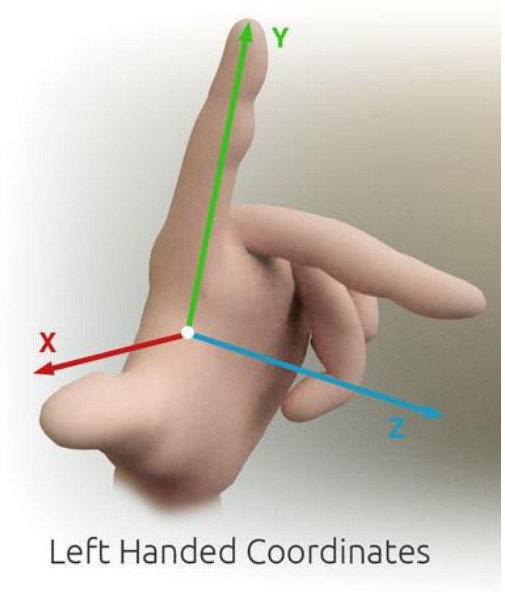




Coordinate Systems in OpenGL

- ***Normalized device coordinate*** **(NDC) system**
  - **In 2D or after a projection**, objects are located in the NDC system.
  - **Canonical view volume** is a box centered at the origin with sides of 2, in which objects are visible.
  - NDC uses **LHS convention** for depth test, which maintains objects with the **smallest** depths as visible.





Closer look into Code

Common Headers

- **cgmath.h (or cgmath-min.h)**
  - slee's math library

- **cgut.h**
  - slee's OpenGL utility library
  - defines common data structures

```
struct vertex;    // structure for vertices
struct mesh;      // collection of vertices and indices for rendering
```

  - defines many utility functions including:

```
cg_create_window();
cg_init_extensions();
cg_create_program();
cg_destroy_window();
```

Global Variable Definitions

- **All the examples will use variables similar to below.**

```
// global constants
const char* window_name= "cgbase - Hello triangle!";
const char* vert_shader_path = "../bin/shaders/hello.vert";
const char* frag_shader_path = "../bin/shaders/hello.frag";

// window objects
GLFWwindow* window = nullptr;           // default GLFW window
ivec2  window_size = ivec2( 512, 512 ); // initial window size

// OpenGL objects holding IDs generated from OpenGL
GLuint program = 0;        // ID holder for GPU program
GLuint vertex_array = 0;   // ID holder for vertex array object

// global variables
int  frame = 0; // index of rendering frames
vec4 solid_color = vec4( 1.0f, 0.5f, 0.5f, 1.0f );
bool b_solid_color = false;
```

main()

- **Initialization**
  - Window is created via cg_create_window(), which also initializes glfwInit().
  - To get function pointers of OpenGL extensions, call cg_init_extensions()
    - OpenGL extensions are dynamically obtained from driver (OpenGL32.dll)
  - Error checks are not shown here.

```
void main( int argc, char* argv[] )
{
   // create window and initialize OpenGL extensions
   window = cg_create_window( window_name, window_size.x, window_size.y );
   cg_init_extensions( window ); // init OpenGL extensions
```

main()

- **Initialization and validations of GLSL program**
  - cg_create_program() compiles vertex and fragment shaders and links them together for a single GLSL program.
  - user_init(): user-defined initialization
    - will be explained later

```
...

// initializations and validations of GLSL program
program = cg_create_program( vert_shader_path, frag_shader_path );
user_init();      // user initialization

...
```

main()

- **Registration of event callbacks**
  - glfwSet(*)Callback registers functions to the associated window events.
  - There are four major types of callbacks:
    - window resizing, keyboard press/release, mouse clicks, mouse movements

```
...

// register event callbacks
glfwSetWindowSizeCallback( window, reshape );  // window resizing events
glfwSetKeyCallback( window, keyboard );        // keyboard events
glfwSetMouseButtonCallback( window, mouse );   // mouse click inputs
glfwSetCursorPosCallback( window, motion );    // mouse movements

...
```

main()

- **(Infinite) Event loop**
  - glfwPollEvents() processs events and their registered callbacks.
  - User-defined update() and render() functions are called in a row.
    - These functions are for per-frame update and rendering.
  - When glfwWindowShouldClose() returns true, the loop is terminated.

```
...

// enters rendering/event loop
for( frame=0; !glfwWindowShouldClose(window); frame++ )
{
   glfwPollEvents(); // polling and processing of events
   update();         // per-frame update
   render();         // per-frame render
}

...
```

main()

- **Termination**
  - user_finalize() do user-defined clean-ups.
  - cg_destroy_window() calls glfwTerminate().

```
    ...

    // normal termination
    user_finalize();
    cg_destroy_window(window);
}
```

user_init()

- **User-defined initializations are placed here**
  - print the usage of the applications
  - initialize basic GL states (e.g., depth test, back-face culling)
  - define host-side vertex attributes (e.g., a triangle of three vertices)
  - create vertex buffers and vertex array objects
  - any other things that you need to initialize

user_init()

- **First, show the usage on console**
  - This is highly recommended when you do not have text rendering.

- **Initialize GL states**
  - This is nearly default for all the other examples

```
bool user_init()
{
   // log hotkeys
   print_help();

   // init GL states
   glClearColor( 39/255.0f, 40/255.0f, 34/255.0f, 1.0f ); // set clear color
   glEnable( GL_CULL_FACE );   // enable backface culling
   glEnable( GL_DEPTH_TEST );  // enable depth test

   ...
   return true;
}
```

user_init()

- **Vertex definition (declared in cgut.h)**

```
struct vertex // will be used for all the course examples
{
   vec3 pos;  // position
   vec3 norm; // normal vector; use this for vertex color for this example
   vec2 tex;  // texture coordinate; ignore this for the moment
};
```

- Here, we use `norm` to store vertex color (for compatibility with other examples), although it's intended for normal vectors.
- Create a vertex array on host (CPU) side.

```
// create a vertex array for triangles in NDC cube [-1~1]^3
vertex vertices[] =
{
   { vec3(0.433f,-0.25f,0), vec3(1,0,0), vec2(0.0f) },  // {pos, red, ...}
   { vec3(0.0f,0.5f,0), vec3(0,1,0), vec2(0.0f) },      // {pos, green, ...}
   { vec3(-0.433f,-0.25f,0), vec3(0,0,1), vec2(0.0f) }  // {pos, blue, ...}
};
```

user_init()

- **Create vertex buffers**
  - OpenGL buffer objects allow us to efficiently transfer large amounts of data to the GPU.
  - **Vertex buffers** are the input to the vertex shader of a GPU program.
  - You can access it only via buffer ID (e.g., `vertex_buffer` in the code).
    - There is no way of directly access (e.g., pointers) to GL objects.

```
bool user_init()
{
    ...

    // create and update vertex buffer
    glGenBuffers(1, &vertex_buffer ); // generate one buffer object
    glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer); // notify GL using the buffer

    ...
```

user_init()

- **Allocate GPU buffer memory**
  - glBufferData() allocates GPU memory.
  - A host-side vertex array is compiled to GPU *vertex buffers*.
    - When nullptr is given, there is no data copy to GPU.
    - Remember GL buffers are only the copies of host-side data.
  - We use GL_STATIC_DRAW, since our example has constant content.
    - You may use GL_DYNAMIC_DRAW for dynamic buffers.
    - GL_STATIC_DRAW is just a hint. So, even though you specify wrong, it still works.

```
// create and update vertex buffer
glBufferData( GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW );
```

user_init()

- **Generating vertex array object from vertex buffers**
  - OpenGL core profile (version ≥ 3.3) requires to use vertex array object (VAO), and not allows to use vertex buffers directly.
    - VAO is an abstraction of vertex (and index) buffers and their bindings.
  - So, you need to create vertex array objects

```
// generate vao, which is mandatory for OpenGL 3.3 and higher
vertex_array = cg_create_vertex_array( vertex_buffer );
```

  - The details are complicated, but are abstracted in cg_create_vertex_array()
  - If you want to know more, look over the details in the next pages.

(Advanced) cg_create_vertex_array()

- **Generate vertex array objects (VAOs)**
  - First simply generate a VAO
  - The, bind vertex (and index) buffers.
    - The details of index buffers will be covered in the next lecture.

```
// create and bind a vertex array object
GLuint vao = 0;
glGenVertexArrays( 1, &vao );
glBindVertexArray( vao );

// bind vertex/index buffer
glBindBuffer( GL_ARRAY_BUFFER, vertex_buffer );
if(index_buffer) glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, index_buffer );

...
```

(Advanced) cg_create_vertex_array()

- **Bind vertex attributes to GPU program**
  - We are now indicating that we are working on the n-th attributes.

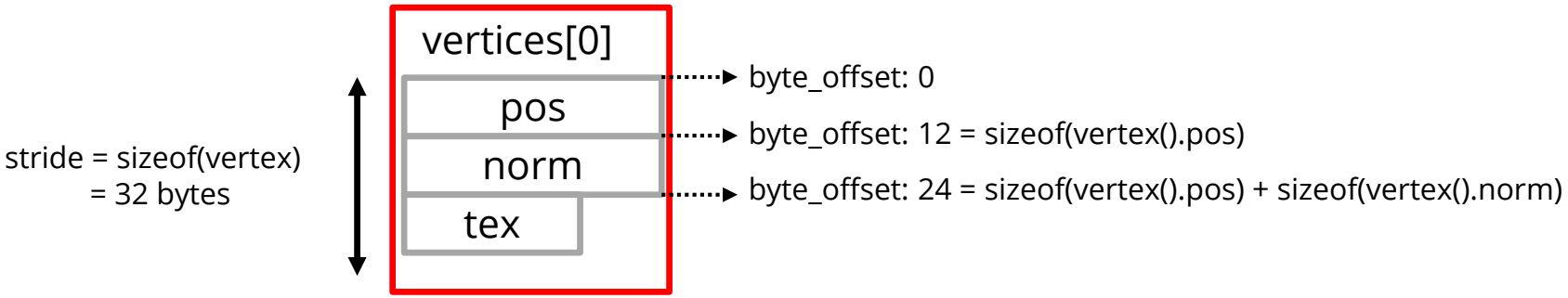
```
    // bind vertex attributes by interpreting the organization of struct vertex
    size_t attrib_size[] =
       {sizeof(vertex().pos),sizeof(vertex().norm),sizeof(vertex().tex)};
    for( size_t k=0,byte_offset=0; k<3; k++, byte_offset+=attrib_size[k-1] )
    {
       glEnableVertexAttribArray( GLuint(k) );
       ...
     }
}
```

(Advanced) cg_create_vertex_array()

- **Bind vertex attributes to GPU program**
  - We notify GL the memory layout of vertex buffer.

```
glVertexAttribPointer(
   GLuint(k),                          // attribute location
   attrib_size[k]/sizeof(GLfloat), // number of elements (e.g., 3 for vec3)
   GL_FLOAT,                           // type of elements (e.g., float for vec3)
   GL_FALSE,                           // never use fixed-point normalization
   sizeof(vertex),                     // stride: size of structure
   (GLvoid*) byte_offset );            // byte offset of each attribute
```

  - Memory layout of C/C++ structures



user_finalize ()

- **You can locate some clean-up, here.**
  - But, it's empty at present.

```
void user_finalize()
{
   // some clean-up code here
   if(vertex_array) glDeleteVertexArrays(1,&vertex_array);
};
```

update() and render()

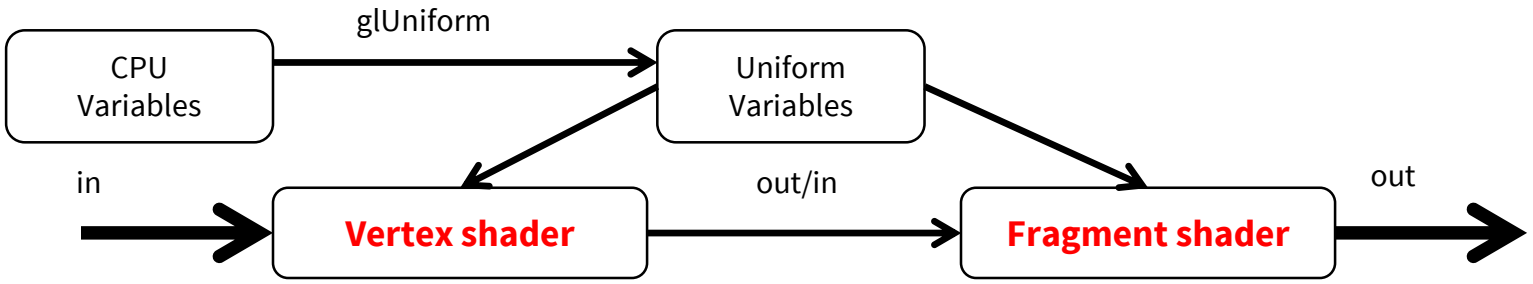
update()

- **Update simulation**
  - Here, we compute rotation based on elapsed time.
  - You may put more simulations and transformations here.

```
void update()
{
   // update simulation factor by time
   float theta = float(glfwGetTime())*0.5f;

   ...
}
```

update()

- **Uniform variables**
  - **Read-only global variables** in GPU program, which can be accessed when processing any vertices and fragments.
  - Uniform variables are similar to constant global variables in C.



update()

- **Update uniform variables**
  - First query the index of uniform variables using its name.
    - Returning -1 means that there is no such name or is not used in the program.
    - Declare but non-used variables also returns -1.
  - glUniform*() copies CPU variables to GPU's uniform variable objects.

```
void update()
{
    ...

    // update uniform variables in vertex/fragment shaders
    GLint uloc;

    uloc = glGetUniformLocation( program, "theta" );
    if(uloc>-1) glUniform1f( uloc, theta );
    uloc = glGetUniformLocation( program, "b_solid_color" );
    if(uloc>-1) glUniform1i( uloc, b_solid_color );
    uloc = glGetUniformLocation( program, "solid_color" );
    if(uloc>-1) glUniform4fv( uloc, 1, solid_color );
}
```

render()

- **First, clear the framebuffer (screen) with the clear color**
  - Clear color was configured in user_init()

- **Notify GL that we use our own program and vertex array**

```
void render()
{
   // clear screen (with background color) and clear depth buffer
   glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

   // notify GL that we use our own program
   glUseProgram( program );

   // bind vertex array object
   glBindVertexArray( vertex_array );

   ...
}
```

render()

- **Finally trigger GPU program by calling**
  - glDrawArrays(GL_TRIANGLES, 0, 3)

```
...
// render vertices: trigger shader programs to process vertex data
glDrawArrays(GL_TRIANGLES,0,3); // (topology, start offset, no. vertices)
```

- **Double-buffer swapping**
  - In double buffering, we are filling pixels in the back buffer, but we see the image of the front buffer.
  - glfwSwapBuffers() notifies to GLFW to swap back and front buffers.
  - At this point, the system waits for vertical refresh of a monitor.

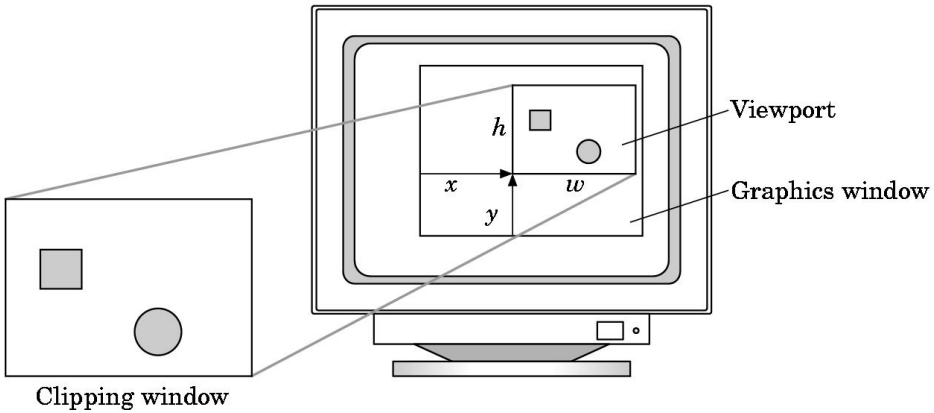
```
...
// swap front and back buffers, and display to screen
glfwSwapBuffers( window );
}
```

Callback Functions

reshape()

- **callback for when a window is resized.**

```
void reshape( GLFWwindow* window, int width, int height )
{
   // set current viewport in pixels (win_x, win_y, win_width, win_height)
   // viewport: the window area that are affected by rendering
   window_size = ivec2(width,height);
   glViewport( 0, 0, width, height );
}
```



keyboard()

- **callback for when a user pressed/released a key.**
  - You can handle all the low-level keyboard events here.
  - *key* is defined by GLFW and *action* indicates the press/release
  - *mods* are logical OR of ctrl/shift/alt modifier.

```
void keyboard( GLFWwindow* window, int key, int scancode, int action, int mods )
{
   if(action==GLFW_PRESS)
   {
     if(key==GLFW_KEY_ESCAPE||key==GLFW_KEY_Q) glfwSetWindowShouldClose(window,GL_TRUE);
     else if(key==GLFW_KEY_H||key==GLFW_KEY_F1) print_help();
     else if(key==GLFW_KEY_D) ...
   }
   else if(action==GLFW_RELEASE)
   {
      ...
   }
}
```

mouse()/motion()

- **mouse(): callback for mouse button clicks**
  - *button* indicates which button is pressed/released.
  - *action* can be either of GLFW_PRESS or GLFW_RELEASE
  - The mouse position can be found using glfwGetCursorPos()

```
void mouse( GLFWwindow* window, int button, int action, int mods )
{
   if( button==GLFW_MOUSE_BUTTON_LEFT && action==GLFW_PRESS )
   {
      dvec2 pos; glfwGetCursorPos( window, &pos.x, &pos.y );
      printf( "> Left mouse button pressed at (%d, %d)\n", int(pos.x), int(pos.y) );
   }
}
```

- **motion(): callback for mouse movements**

```
void motion( GLFWwindow* window, double x, double y )
{
}
```