Hello Triangle World

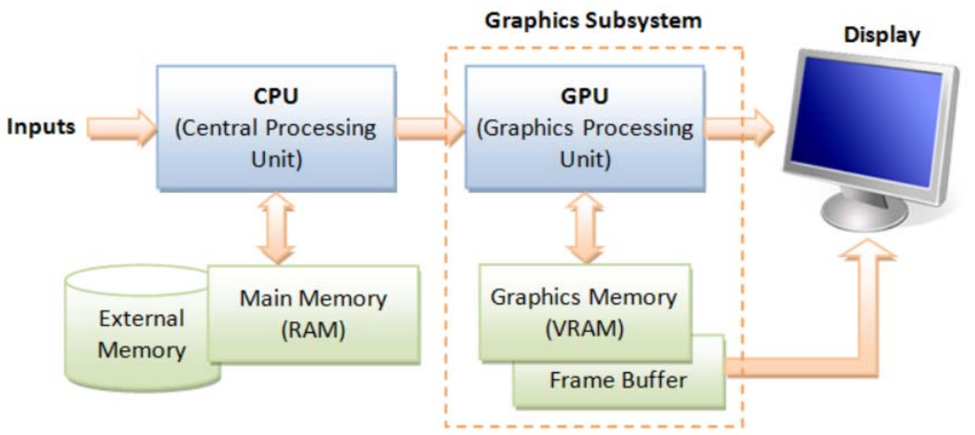
Supplement for
OpenGL Introduction

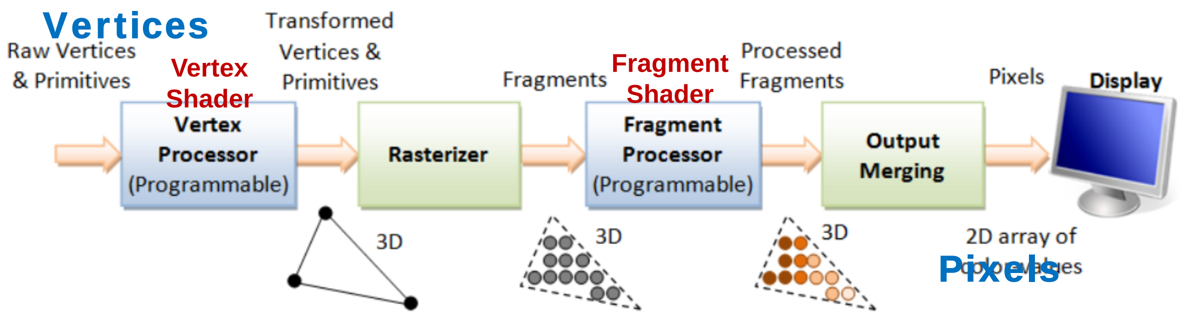
March 10, 2016

Last Lecture

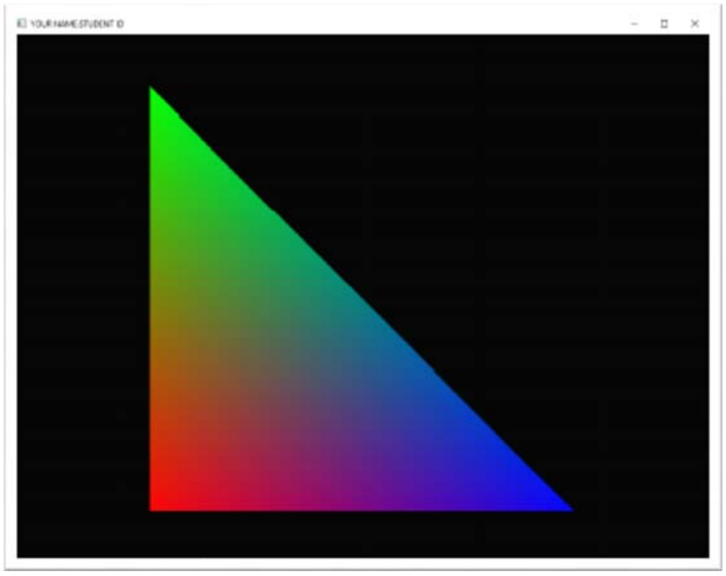
- Computer Graphics
  - All aspects of generating pictures using a computer
  - → Images
    - arrays of pixels
- The Programmer's Interface
  - Graphics API
    - OpenGL
      - State machine
        - Configurable
- Graphics Systems
  - Pixels and the Frame Buffer
  - Output devices
  - Input devices
    - Input mode
      - Event-driven
  - CPU / GPU
- Graphics Architectures
  - Graphics Pipeline
    - Vertex processing
    - Clipping and primitive assembly
    - Rasterization
    - Fragment processing
  - Programmable Pipelines

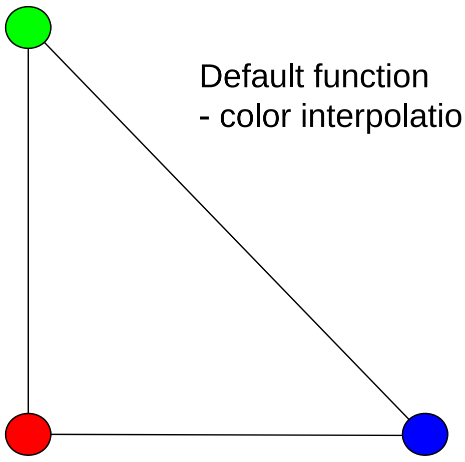
https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html





HW#0

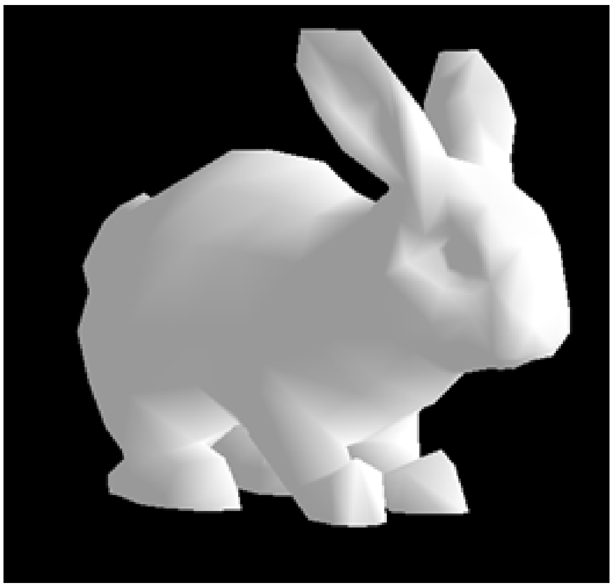




Default function
- color interpolation

Geometric Objects

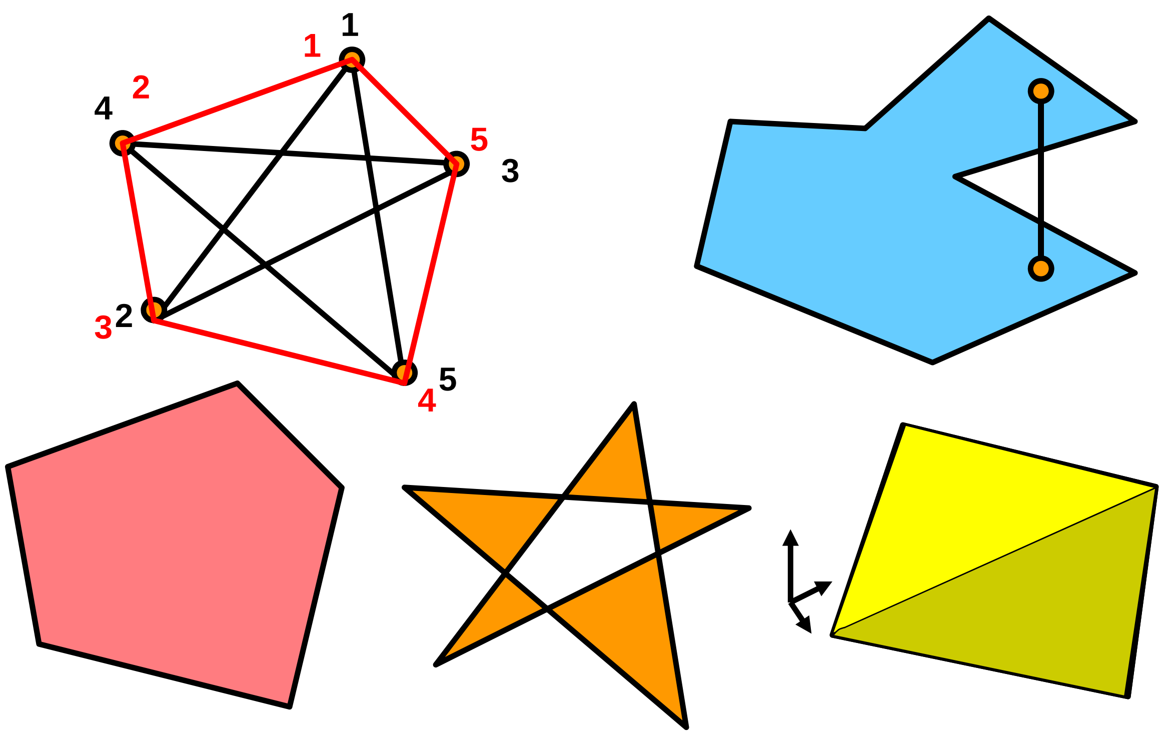
- Made of (defined by) a set of vertices (faces)
  - Usually triangles
  - Flat polygons
- Vertex has **position** information (coordinates)
- Objects
  - Point : 1 vertex
  - Triangle : 3 vertices
  - Rabbit : 251 triangles
  - …



Polygon Basic

- Very special role in computer graphics
  - Performance of the graphics system is measured by number of polygons per second that can be displayed.
- Outer edges of the polygon are defined by an **ordered** **list of vertices.**
- Three properties that ensure that the interior of a polygon is <u>well defined</u> (so the polygon be displayed correctly):
  - Simple
    - No pairs of edges cross each other
  - Convex
    - If all points on the line segment between any two points inside the object, or on its boundary, are inside the object
  - Flat
    - Triangle
- Most graphics systems require that the application program do any necessary testing.

polygons: simple, convex, flat → triangles



(2D) graphics programming

- We do it in 3D world
  - Regard 2D systems as special cases
  - A point in the z=0 plane as p=(x,y,0) in 3D world, or p=(x,y) in 2D plane.
- Vertex vs. Point
  - Vertex: a position in space
    - We use vertices to define the geometric **primitive**
  - Point: a simplest geometry primitive
    - Specified by a single vertex
- The basic OpenGL primitives are specified by sets of vertices.
- An application starts by computing vertex data (positions and other attributes) and putting the results into arrays that are sent to the GPU for display.
  - **glDrawArrays(GL_POINTS, 0, NumPoints);**

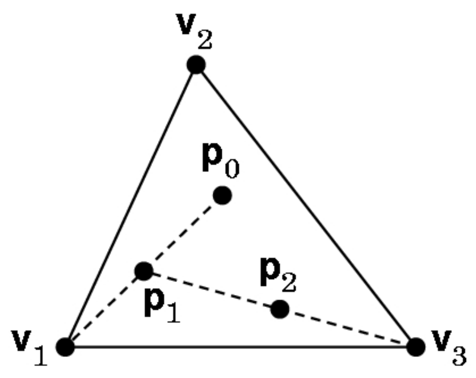
glDrawArrays — render primitives from array data

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

- *mode*
  - Specifies what kind of primitives to render.
  - Symbolic constants 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 are accepted.
- *first*
  - Specifies the starting index in the enabled arrays.
- *count*
  - Specifies the number of indices to be rendered.

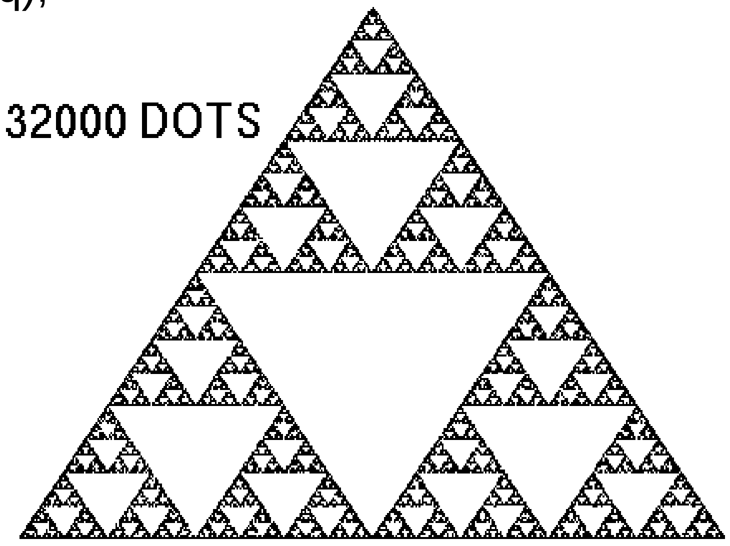
Sierpinski Gasket

1. Pick an initial point at *random* inside the triangle (p_0)
2. Select one of the 3 vertices at random (v_1)
3. Find the point halfway (p_1)
4. Display this new point
5. Replace the initial point with this new point
6. Return to step 2.



- Random process leading to regular behavior
- Fractal

```
main ( ) {
    initialize_the_system();
    p = find_initial point();

    for  (some_number_of_points) {
       q = generate_a_point(p);
       display_the_point(q);
       p = q;
    }
}
```



```
main ( ) {
    initialize_the_system();
    p = find_initial point();

    for  (some_number_of_points) {
       q = generate_a_point(p);
       display_the_point(q);
       p = q;
    }
}
```

Immediate mode graphics

There is no memory of the geometric data.
If we want to display the points again, we should go through the entire creation.

```
main ( ) {
    initialize_the_system();
    p = find_initial point();

    for  (some_number_of_points) {
       q = generate_a_point(p);
       store_the_point(q);
       p = q;
    }
    display_all_points();
}
```

retained mode graphics

Avoids the overhead of sending small amounts of data to the graphics processor for each point we generate at the cost of having to store all the data

How about in animation?

```
main ( ) {
    initialize_the_system();
    p = find_initial point();

    for  (some_number_of_points) {
      q = generate_a_point(p);
      store_the_point(q);
      p = q;
    }
    display_all_points();
}
```

glGenVertexArrays
(find unused name for the buffer)
glBindVertexArray
(object is created given the name)

```
main ( ) {
    initialize_the_system();
    p = find_initial point();

    for  (some_number_of_points) {
      q = generate_a_point(p);
      store_the_point(q);
      p = q;
    }
    send_all_points_to_GPU();
    display_data_on_GPU();
}
```

Vertex Array Object

OpenGL Common Syntax

- OpenGL functions start with gl… glSomeFunction*()
  - glClear(), glFlush()
  - * : nt or ntv (n: 2,3,4,matrix)
    (t: type i f d) (v: pointer to an array list of the data)
- Constant: GL_ …
  - GL_POINTS, GL_TRIANGLES ,,,
- Type: GL…
  - GLfloat, GLinteger, GLuint. GLboolean
- GLFW functions start with glfw…
  - glfwCreateWindow(….)
- Shaders are in
  OpenGL Shading Language (GLSL)

OpenGL is a state-machine

- Set OpenGL states:
  - glEnable(…);
  - glDisable(…);
  - gl*(…);
    // several call depending
    // on purpose

- Query OpenGL states
  - glGet*(….);

- Conceptual Model

OpenGL defined many state variables and contained separate functions for setting the values of individual variables.

The latest version eliminated most of these variables and functions. Instead, **the application program can define its own state variables** and use them or send their values to the **shaders.**

Learned from a lab exercise ..

In initialization

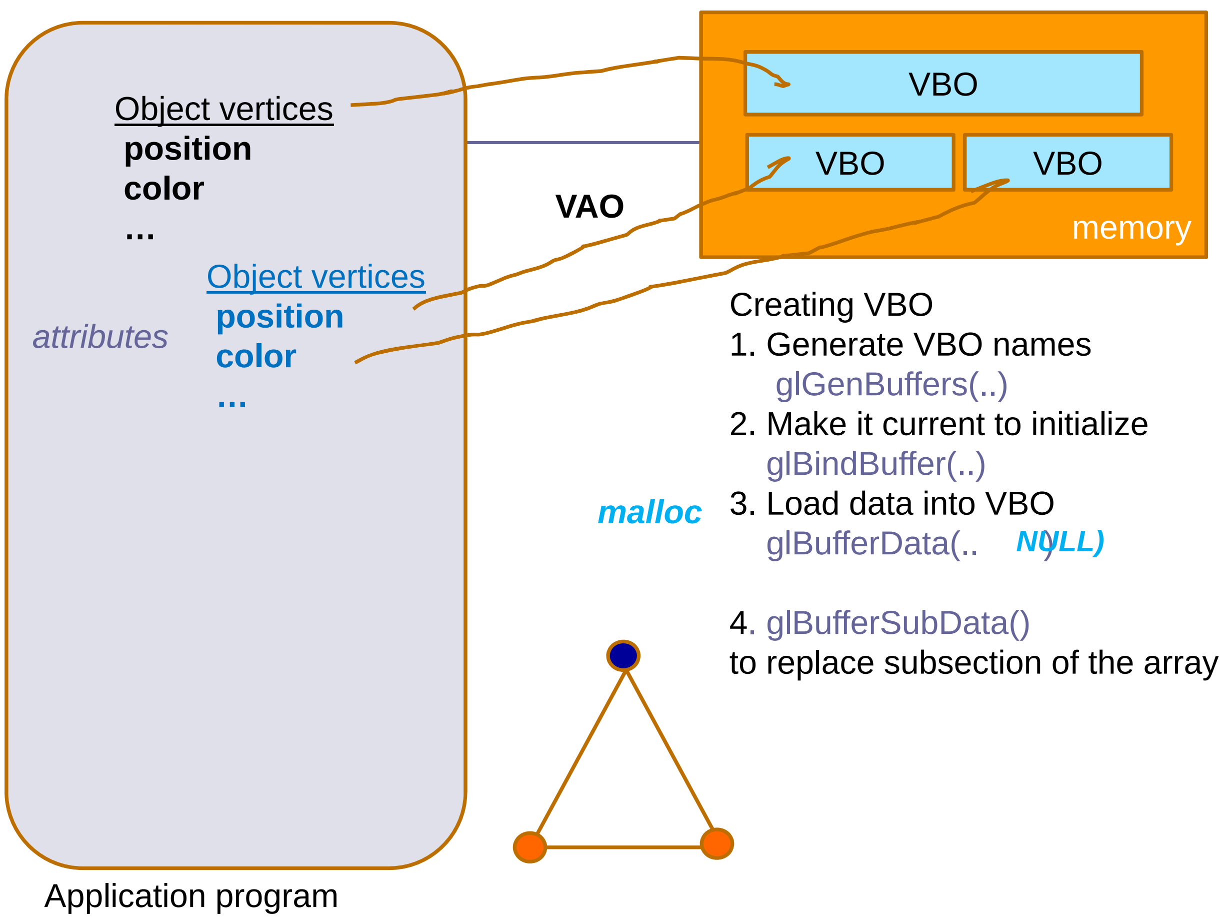
- glGenVertexArray (1,&vao) // Create a vertex array object
  glBindVertexArray(vao)

- glGenBuffer(1, &buffer_id) // Initialize a buffer object on **GPU**
  glBindBuffer(GL_ARRAY_BUFFER, buffer_id)
  // the data in the buffer will be vertex attribute data
  glBufferData(GL_ARRAY_BUFFER, sizeof(points), points,
  GL_STATIC_DRAW)
  // allocate sufficient memory on GPU

In display callback function

- **glDrawArrays**(GL_POINTS, 0, N) // rendering the points

primitive_type, *first_index*, *count*)



Creating VBO

1. Generate VBO names
   glGenBuffers(..)
2. Make it current to initialize
   glBindBuffer(..)
3. Load data into VBO
   glBufferData(.. NULL)

4. glBufferSubData()
to replace subsection of the array

Connecting Vertex Shaders with Geometric Data

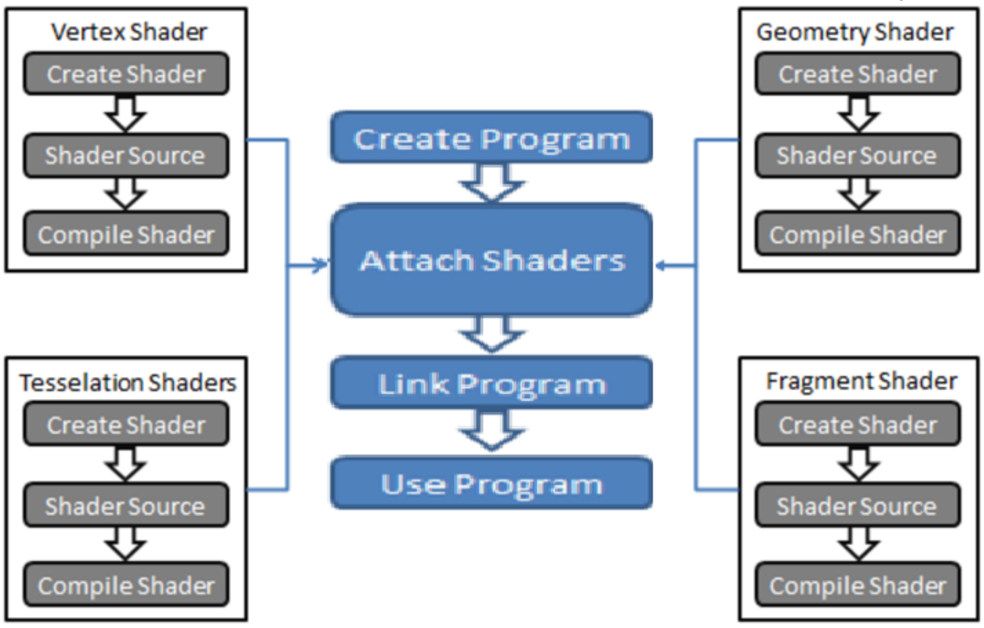
- Application vertex data enters the OpenGL pipeline through the **vertex shader**
- Each of the attributes that we enable must be associated with an "**in**" variable of the currently bound vertex shader.
- Need to connect vertex data to shader variables
  - requires knowing the attribute location
- Attribute location can either be queried by calling `glGetVertexAttribLocation()`

OpenGL Programming in a Nutshell

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

SIGGRAPH2012

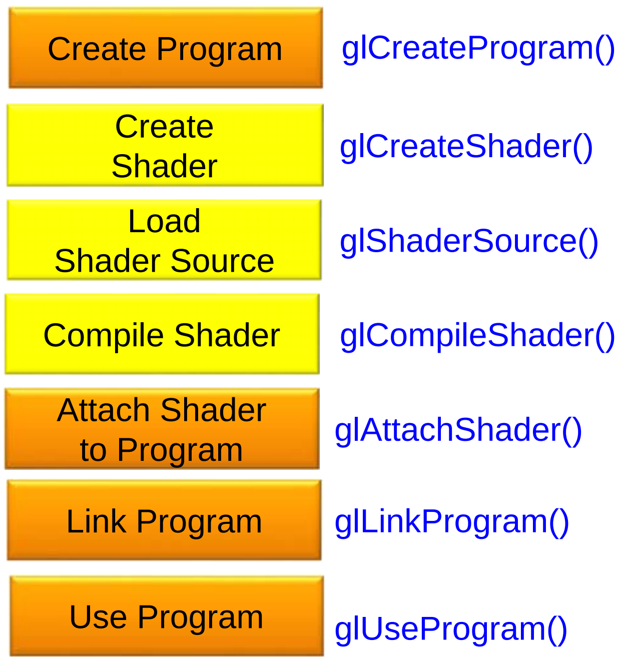
http://www.lighthouse3d.com /



Getting Your Shaders into OpenGL

- Shaders need to be compiled and linked to form an executable shader program
- OpenGL provides the compiler and linker
- **A program must contain vertex and fragment shaders**
  - other shaders are optional

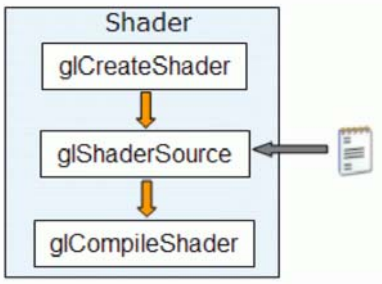
These steps need to be repeated for each type of shader in the shader program



Creating a shader

shaderType:
GL_VERTEX_SHADER or
GL_FRAGMENT_SHADER.

- The first step is creating an object which will act as a shader container. The function glCreateShader(GLenum shaderType) returns a handle for the container.
  - You can create as many shaders as you want to add to a program, but remember that there can only be a *main* function for the set of vertex shaders and one *main* function for the set of fragment shaders in each single program.
- void glShaderSource(GLuint shader, int numOfStrings, const char **strings, int *lenOfStrings); to add some source code. The source code for a shader is a string array.
- Finally, the shader must be compiled glCompileShader(GLuint shader);

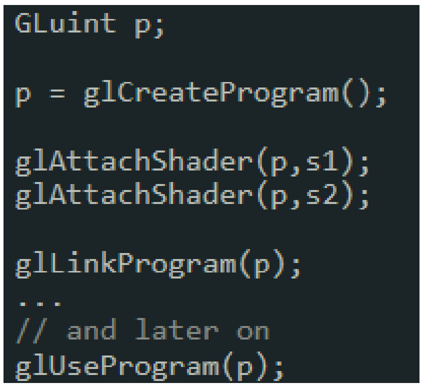


Creating a program and

- The first step is creating an object which will act as a program container. The function glCreateProgram(void) returns a handle for the container.
- We can create as many programs as we want.
  - Once rendering, we can switch from program to program during a single frame. For instance you may want to draw a teapot with refraction and reflection shaders, while having a cube map displayed for background using another shader.
- Attach a shader to a program glAttachShader (prog, sh).
  - We can have many shaders of the same type attached to the same program, just like a C program can have many modules. **For each shader type there can only be one shader with a *main* function**.
  - We can also attach a shader to multiple programs, for instance if we plan to use the same vertex shader in several programs.

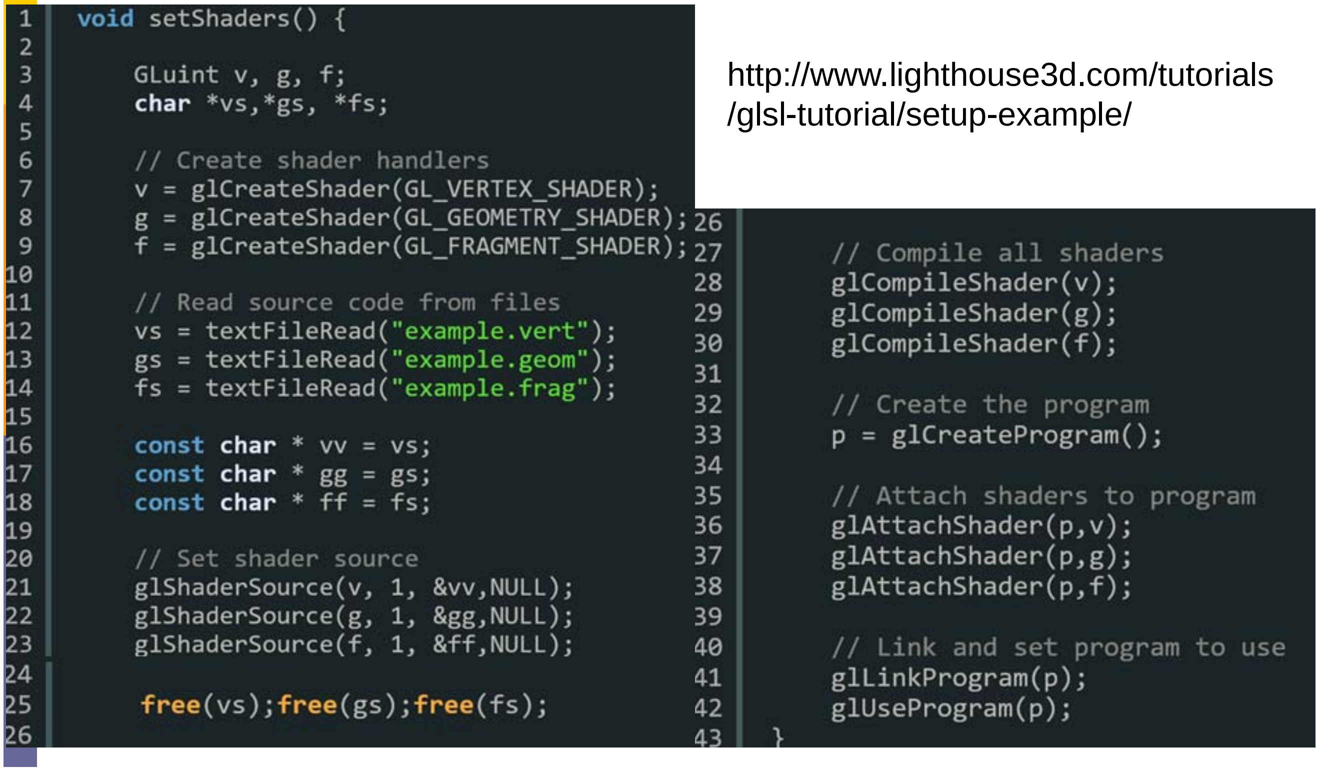
Link program and use …

- The final step is to link the program glLinkProgram(prog). In order to carry out this step the shaders must be compiled first.
  - After the link operation the shader’s source can be modified, and the shaders recompiled without affecting the program. For the modifications to take effect we must link the program again.
  - Each program is assigned an handler, and we can have as many programs linked and ready to use as we want (and our hardware allows), but only one can be active.
- Function to actually load and use the program: glUseProgram(prog)



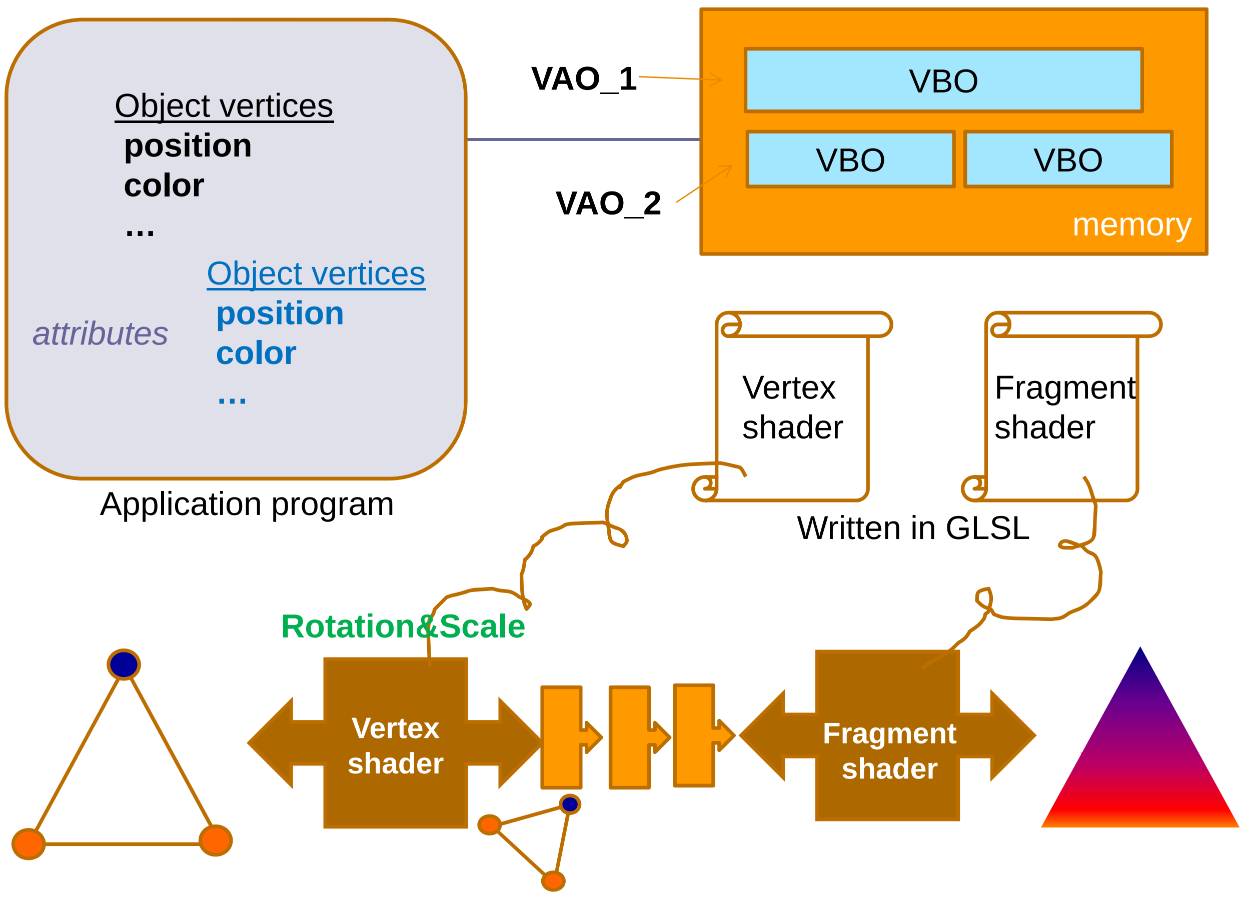
```
GLuint p;

p = glCreateProgram();

glAttachShader(p,s1);
glAttachShader(p,s2);

glLinkProgram(p);
...
// and later on
glUseProgram(p);
```

Setup example



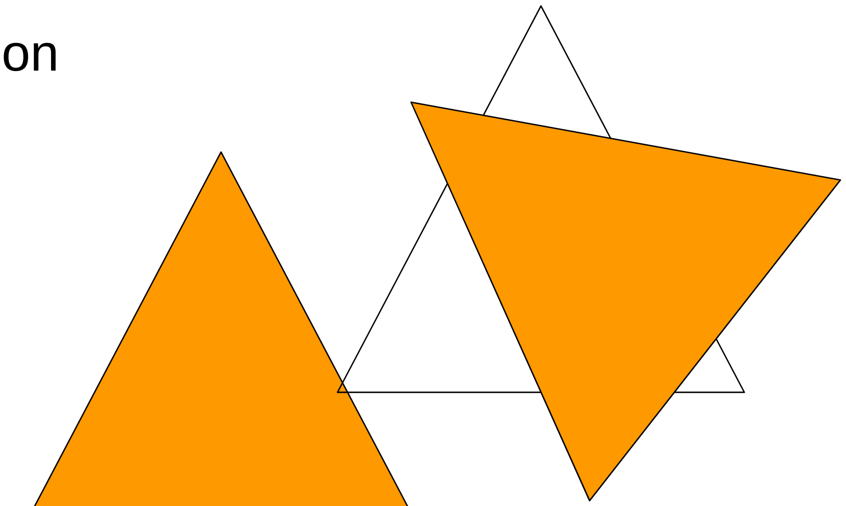
```
void setShaders() {

    GLuint v, g, f;
    char *vs,*gs, *fs;

    // Create shader handlers
    v = glCreateShader(GL_VERTEX_SHADER);
    g = glCreateShader(GL_GEOMETRY_SHADER);
    f = glCreateShader(GL_FRAGMENT_SHADER);

    // Read source code from files
    vs = textFileRead("example.vert");
    gs = textFileRead("example.geom");
    fs = textFileRead("example.frag");

    const char * vv = vs;
    const char * gg = gs;
    const char * ff = fs;

    // Set shader source
    glShaderSource(v, 1, &vv,NULL);
    glShaderSource(g, 1, &gg,NULL);
    glShaderSource(f, 1, &ff,NULL);

    free(vs);free(gs);free(fs);

    // Compile all shaders
    glCompileShader(v);
    glCompileShader(g);
    glCompileShader(f);

    // Create the program
    p = glCreateProgram();

    // Attach shaders to program
    glAttachShader(p,v);
    glAttachShader(p,g);
    glAttachShader(p,f);

    // Link and set program to use
    glLinkProgram(p);
    glUseProgram(p);
}
```

http://www.lighthouse3d.com/tutorials/glsl-tutorial/setup-example/



Rigid-body Transformation

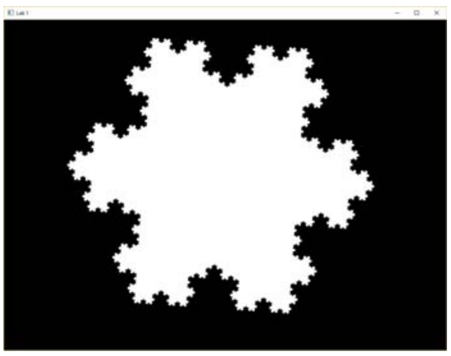
- Translation
- Rotation



→ Read Chapter 2 & 3 for next week !

Lab #1

- Complete your lab exercise by Tuesday NOON (before lunch).
  - Updated handout for the lab will be posted tonight.
    - Koch Snowflake algorithm
    - Transformation (optional)
    - Tips from TA



- Will have a optional Lab Next Wednesday
  - Transformation