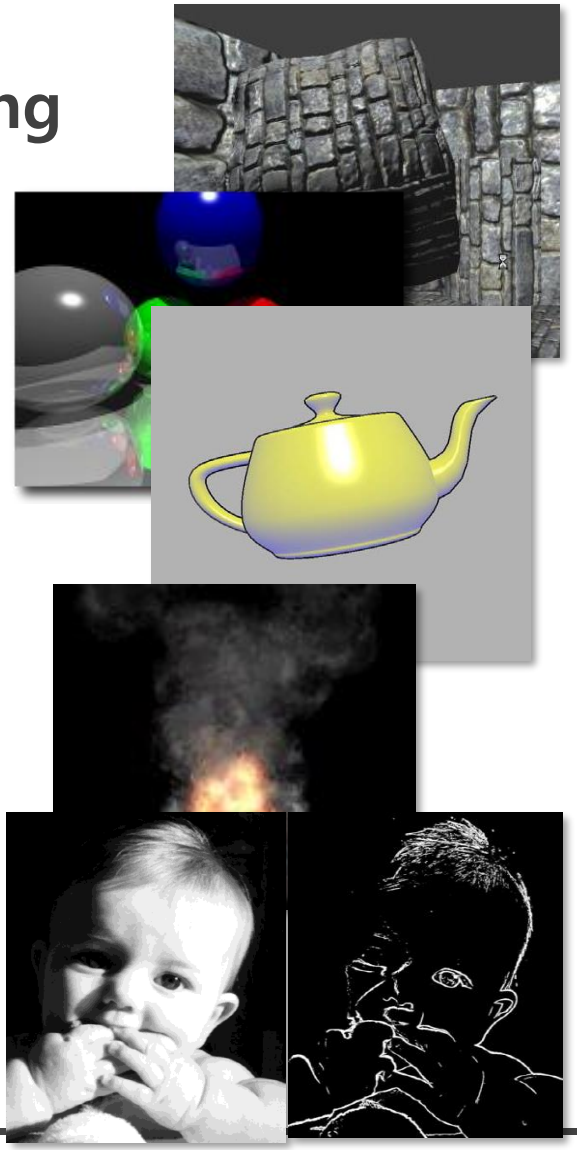
# GLSL Basic

# History of OpenGL

# Application Area of Shaders

- **Realistic materials, mapping & lighting**
- **Advanced rendering effects**
  - Raytracing, NPR, Global Illuminations,…
- **Animation effects**
  - Natural phenomena, particle systems,…
- **Image processing**
  - Filtering, anti-aliasing, matting,…



# Why Shaders?

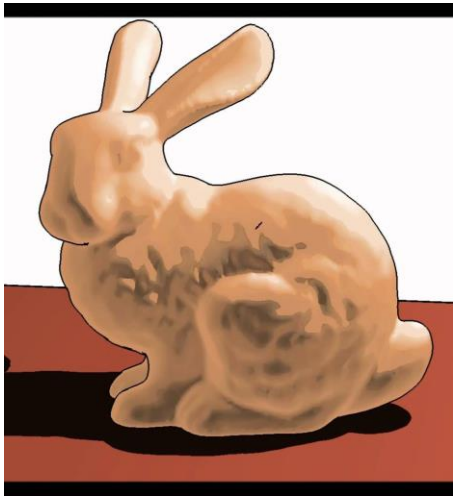
With OpenGL Shading
(GL_SMOOTH)

With GLSL Shading

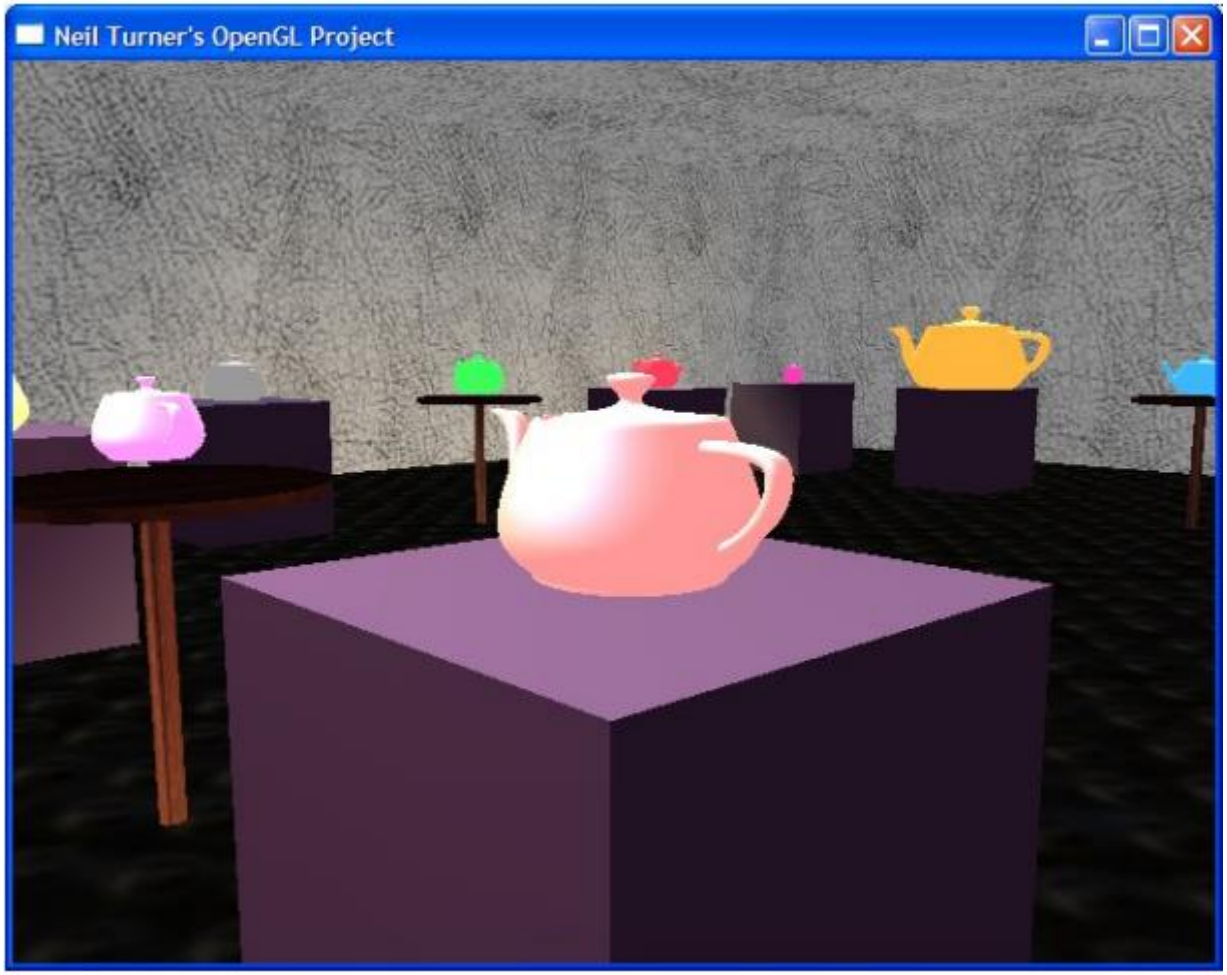
# OpenGL vs. Shader Example

- OpenGL Application

- With Shader

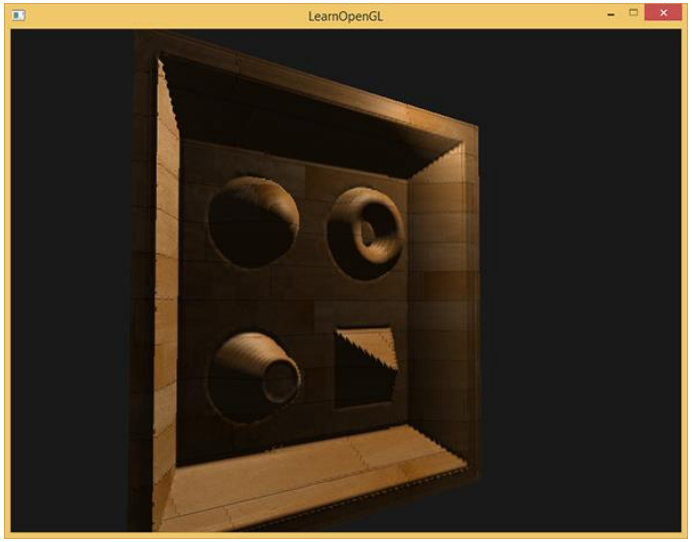


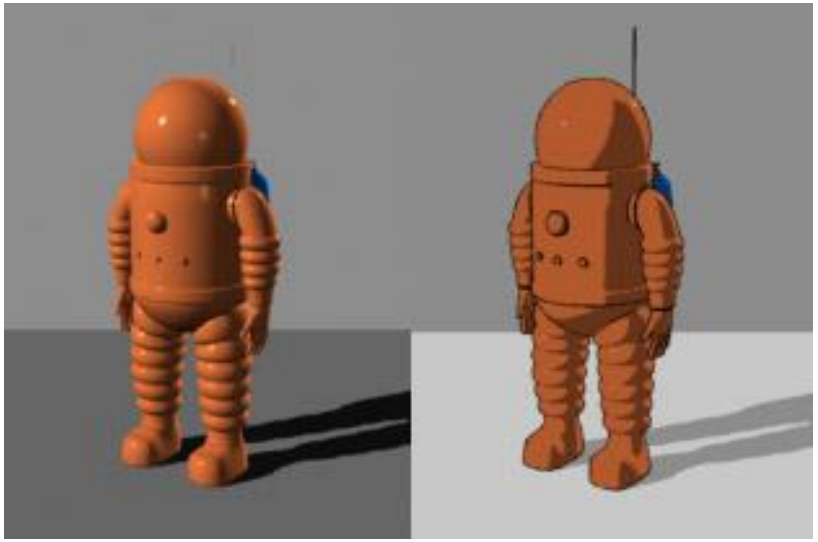
# What could OpenGL do 25 years ago

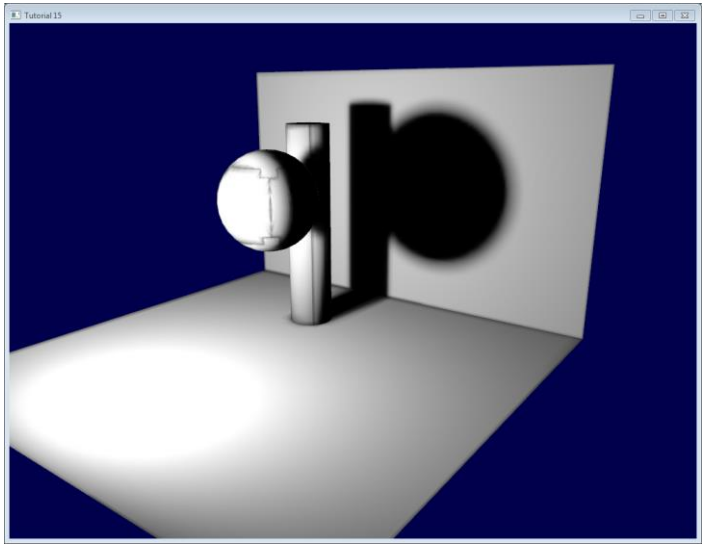


# What can  do now

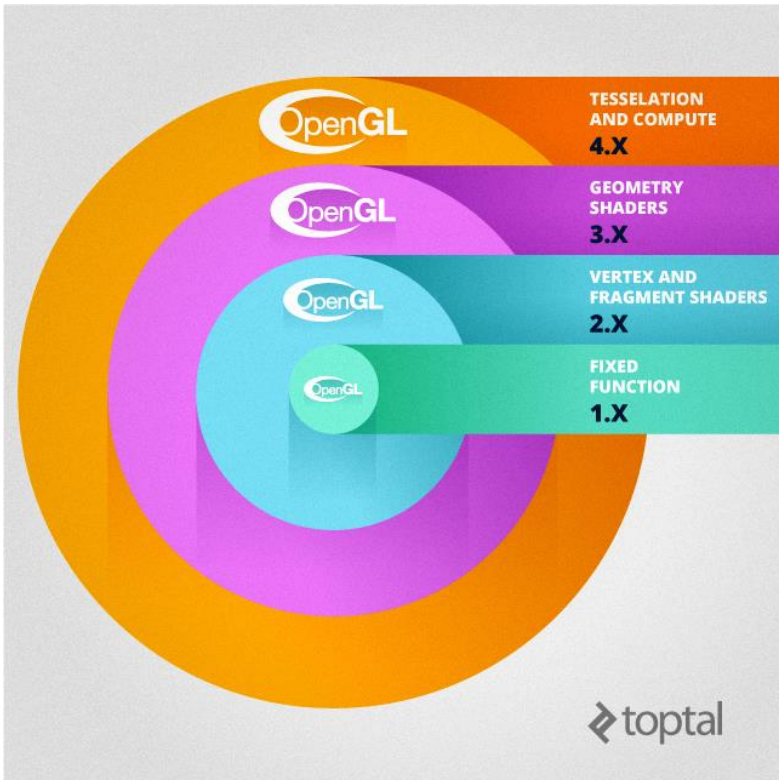
# Other Shader Applications





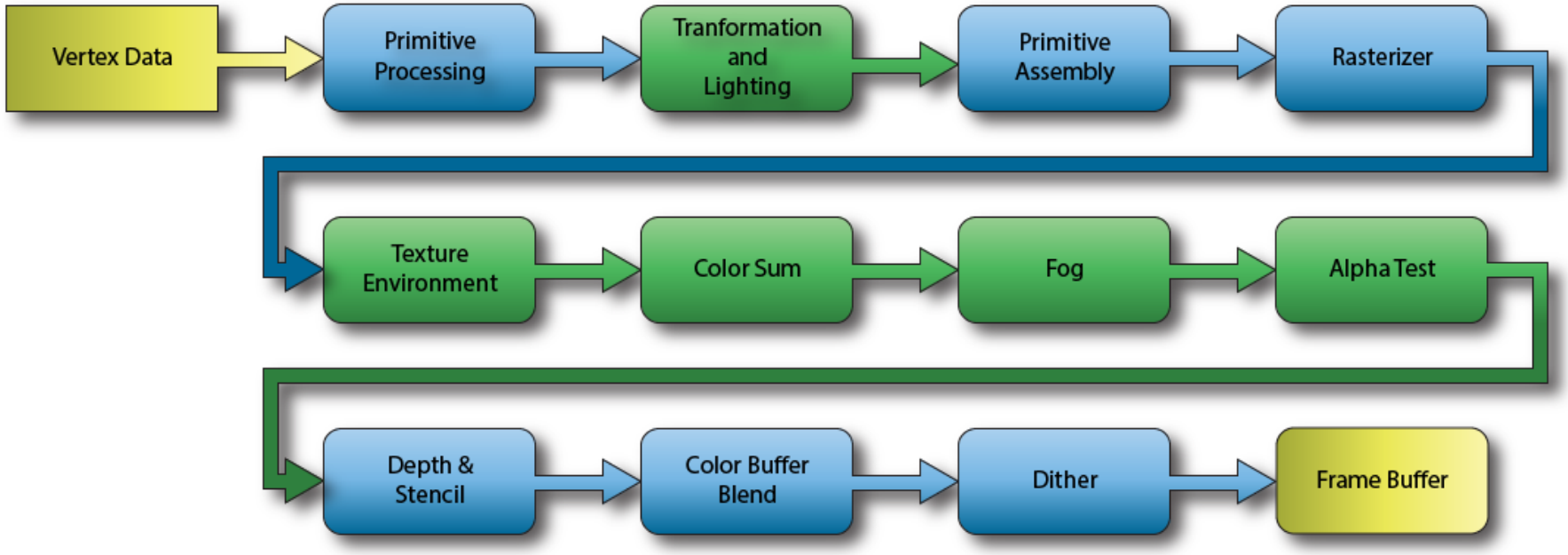

# What's Changed? - Overview




- **25 years ago:**
  - Transform vertices with modelview/projection matrices.
  - Shade with Simple lighting model only.
- **Now:**
  - Custom vertex transformation.
  - Custom lighting model.
  - More complicated visual effects.
  - Shadows
  - Displaced and detailed surfaces
  - Simple reflections and refractions
  - Etc.

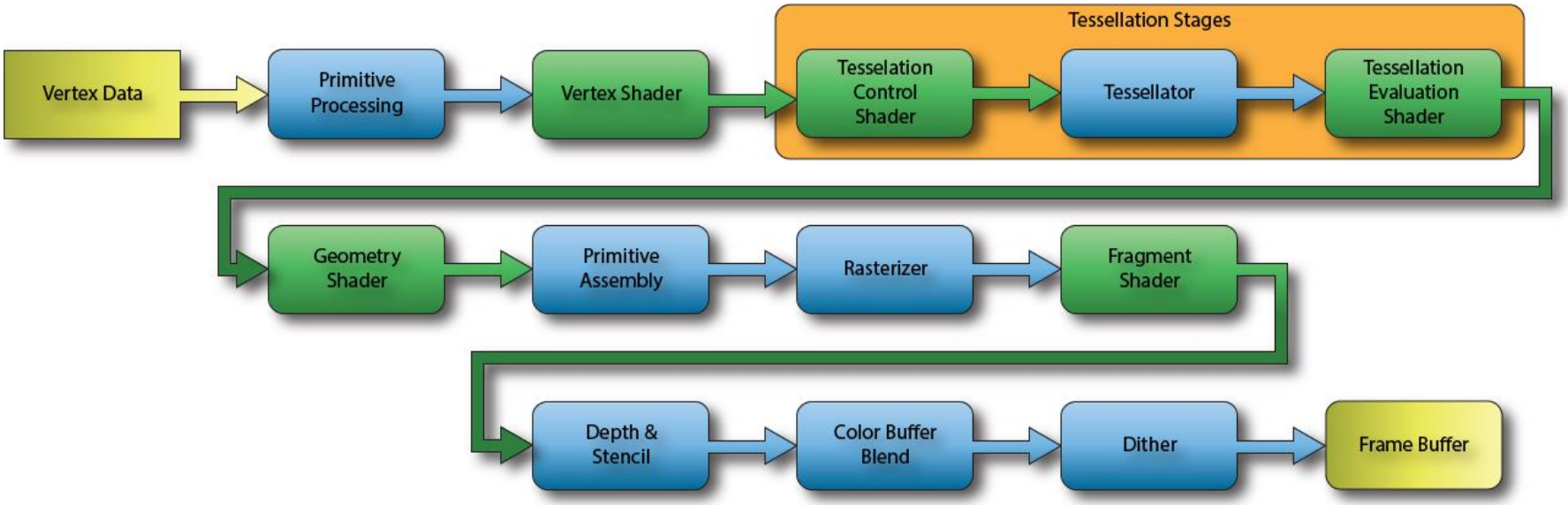
# What's Changed? – 25 years ago

- **25 years ago:**
  - Vertex transformation/fragment shading hardcoded into GPUs.



# What's Changed? - Now

- **Now:**
  - More parts of the GPU are programmable.(GLSL)




What do you need for using GLSL?

GLEW

Project setting is the same as when using OpenGL Extensions

# Vertex Shader

- **Transform vertices from object space to clip space.**
  - Conventionally modelview followed by projection
  - Can define custom transformation to clip space

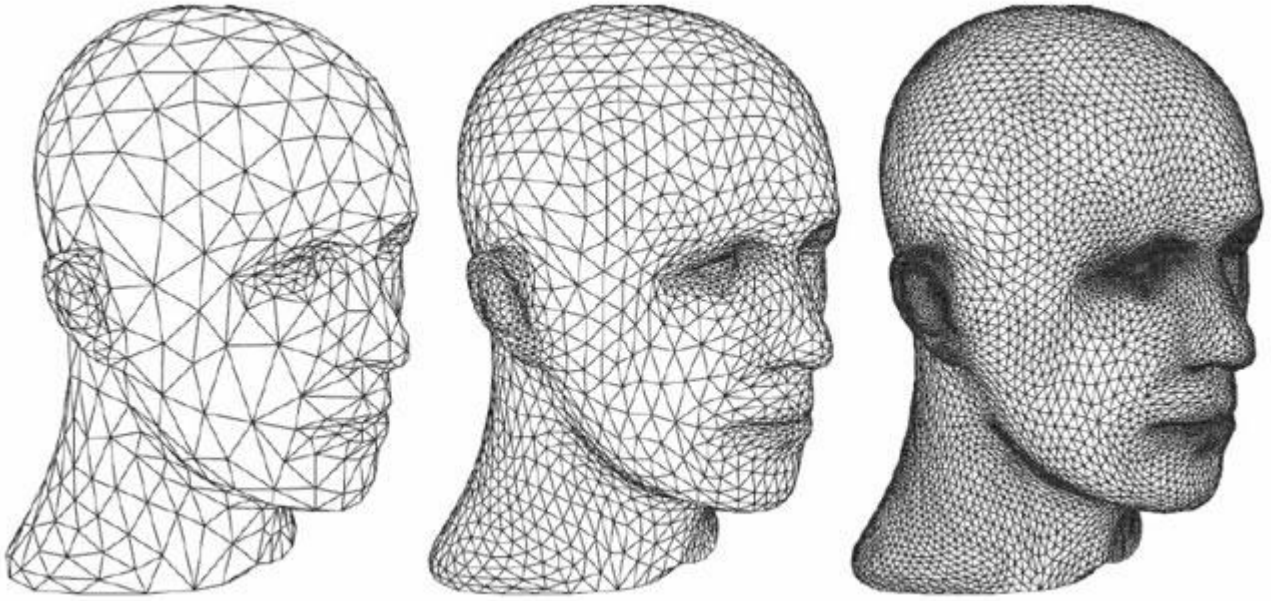
- **Corresponding other data that are related with vertices.**
  - Color
  - Normals
  - Texture coordinates
  - Etc.

# Tessellation Stage

- **Tessellation Control Shader(TCS)**
  - Defines how the subdivision should be done for each primitive.
  - Sets TessLevelOuter, TessLevelInner
- **Tesselator**
  - This stage is a fixed-function stage responsible for creating a set of new primitives from the input patch.
- **Tessellation Evaluation Shader(TES)**
  - TES is responsible for taking the abstract coordinates generated by the primitive generator, along with the outputs from the TCS and using them to compute the actual values for the vertices.
  - This is where you code the algorithm that you actually use to compute the new positions, normal, texcoords etc.

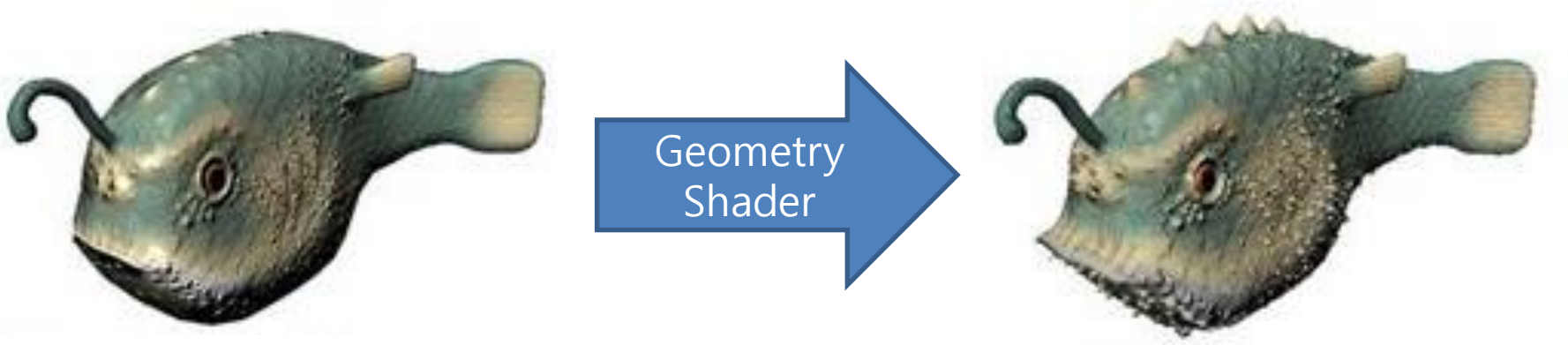
# Tessellator

- Perform adaptive subdivision based on a variety of criteria
  (size, curvature, etc.)
- You can provide coarser models, but have finer ones displayed



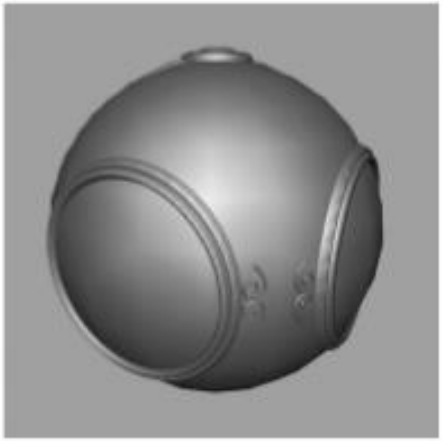
# Geometry Shader

- **Geometry shader invocations take a single Primitive as input and may output zero or more primitives.**
  - Triangles, lines, points, etc.
- **A geometry shader is optional and does not have to be used.**
- **Usually be used to make surface details**



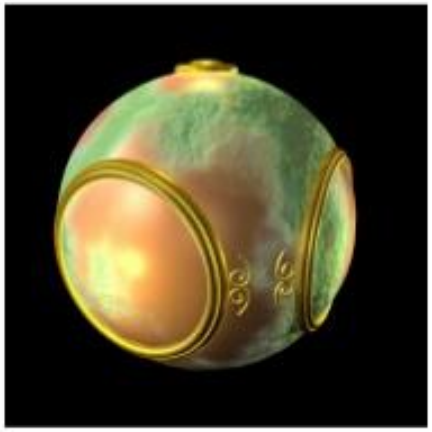
# Fragment Shader

- **Compute the color of a fragment (i.e. a pixel).**
- **Take interpolated data from other shaders.**
- **Can apply more data from:**
  - Textures
  - User specified values



smooth shading

environment mapping

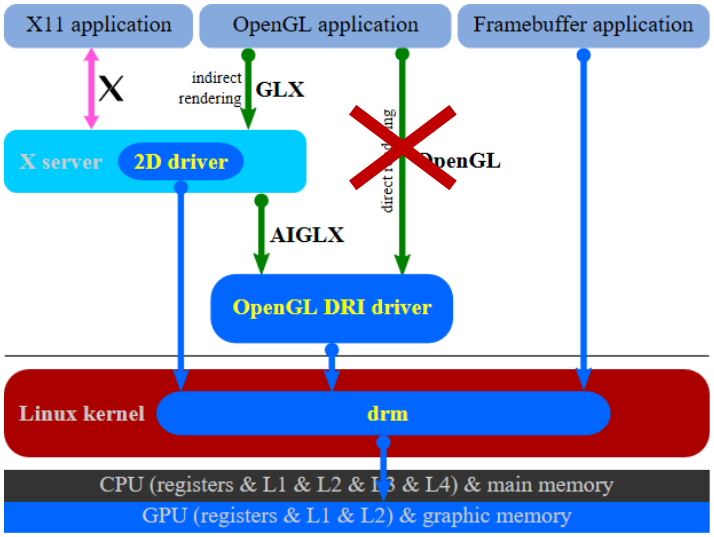


bump mapping

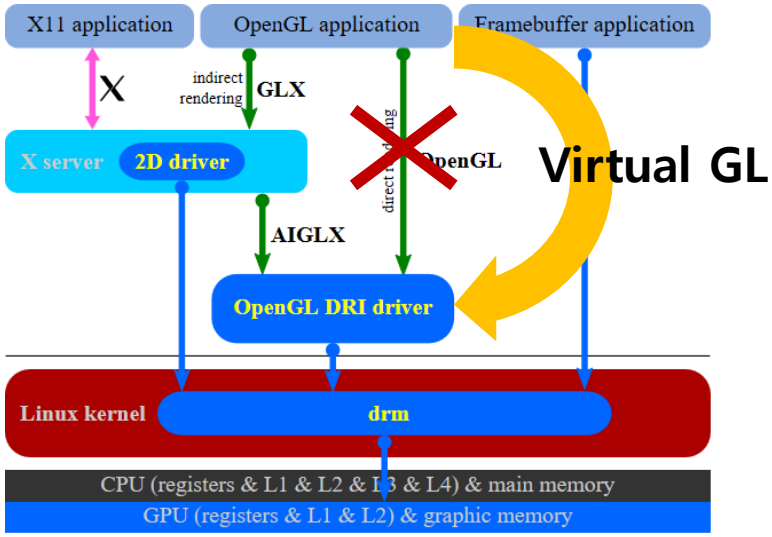
# Virtual GL@GPU Server

# What is Virtual GL?

**Virtual GL:** Open source program that redirects the 3D rendering commands from Unix and Linux OpenGL applications to GPU




<Rendering in remote access>

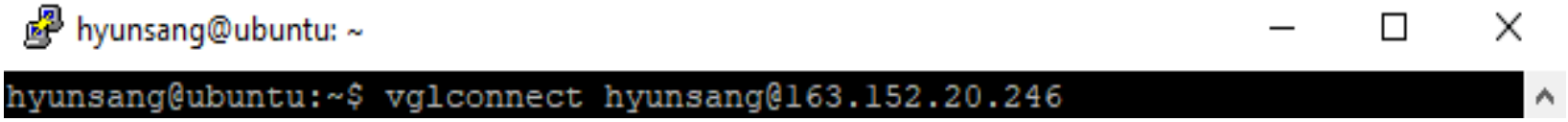



<Direct rendering with Virtual GL>

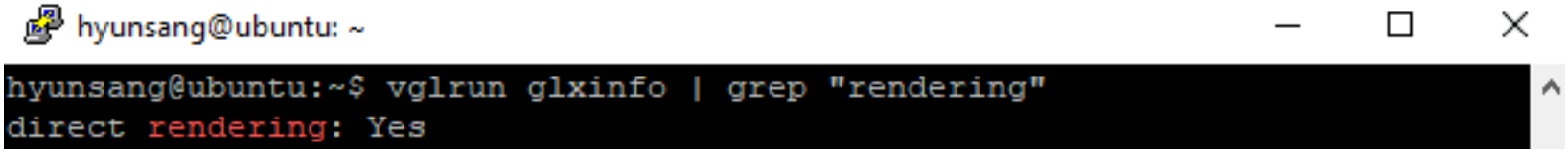
- Direct rendering is **restricted** when connecting **remotely like SSH.**
- **Virtual GL makes it possible for remote rendering to use direct rendering.**

# Usage of Virtual GL

- **Connect to the VirtualGL Server**
  - vglconnect [Client's ID]@[Server IP]




- **Direct Rendering enabled**
  - vglrun glxinfo | grep "rendering"



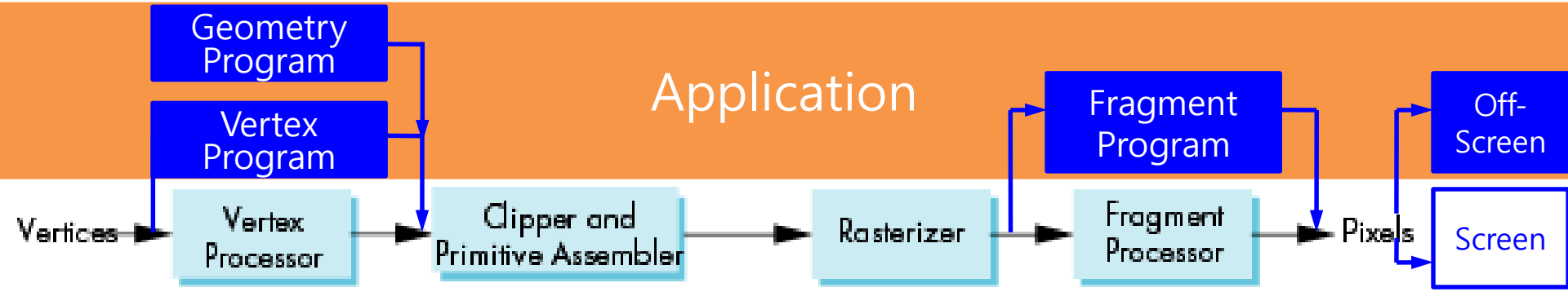

- **Run the program.**
  - vglrun ./[Execution file]

# Shader Programming Preview

# Shader Programming

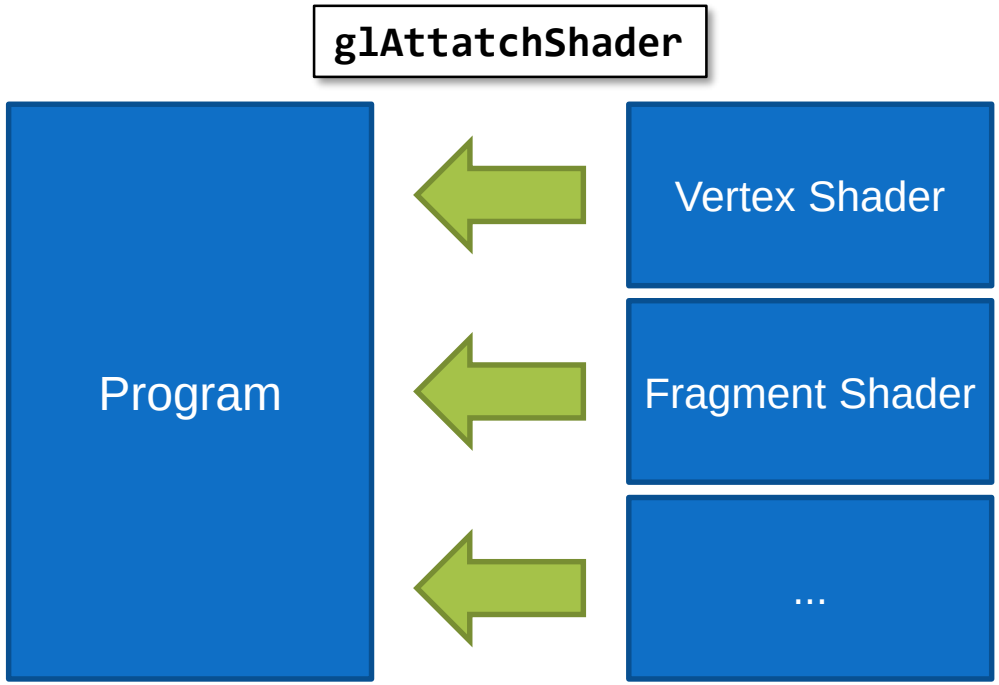
- **Shading languages:**
  - Cg (NVidia's C for graphics)
  - GLSL (OpenGL Shading Language)
  - HLSL (Microsoft's High Level Shading Language)

- **Vertex/Geometry/Fragment programs:**
  - Exploit massive processing capabilities of GPU

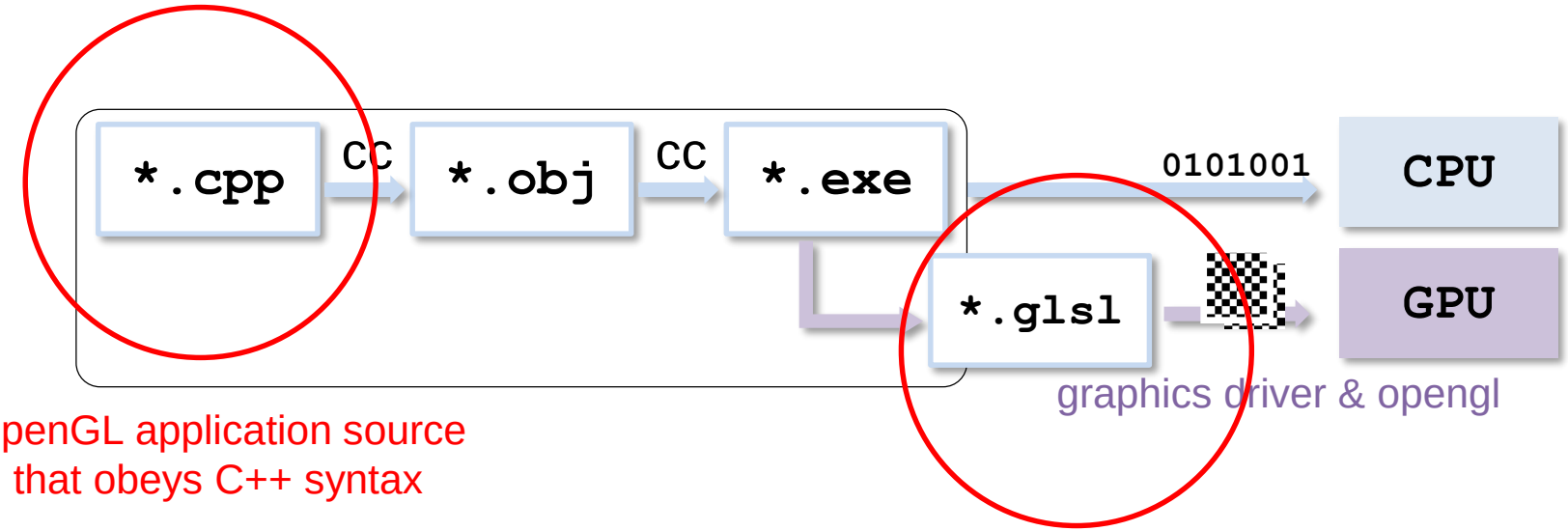


# Shader and Program

- Each **shader** (vertex & fragment) is like **a C module**, and it must be compiled separately, as in C.
- The set of compiled shaders, is then linked into **a program**.

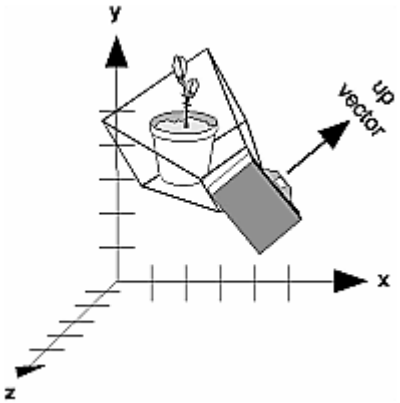


# How to work with GLSL?

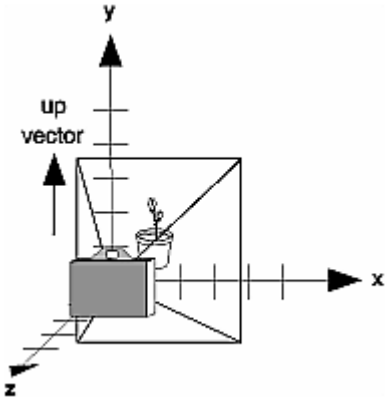


# Vertex Procesing

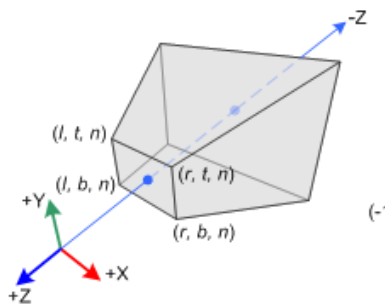
- **One vertex for Input/Output**
- **Each vertex is transformed into "screen space" independently.**
- **Programmable stage.**

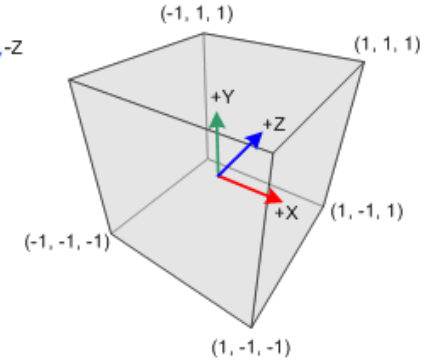



World coordinate



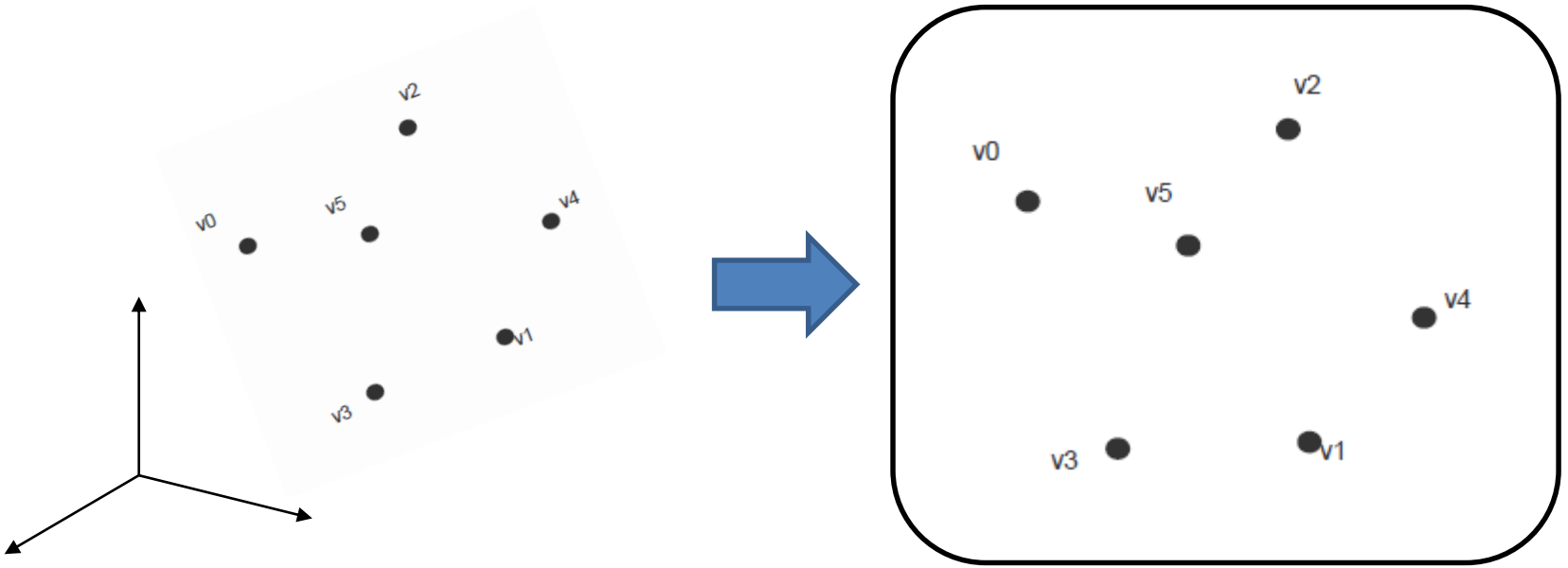

Camera(Eye) coordinate



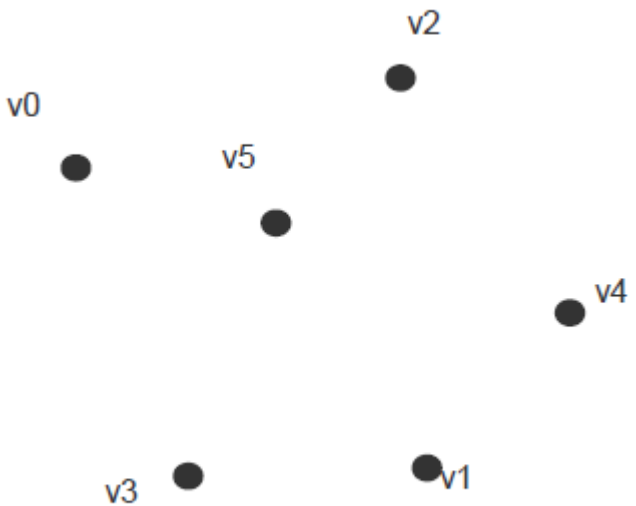

Screen coordinate

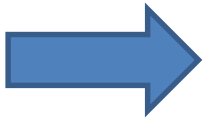
# Vertex Processing

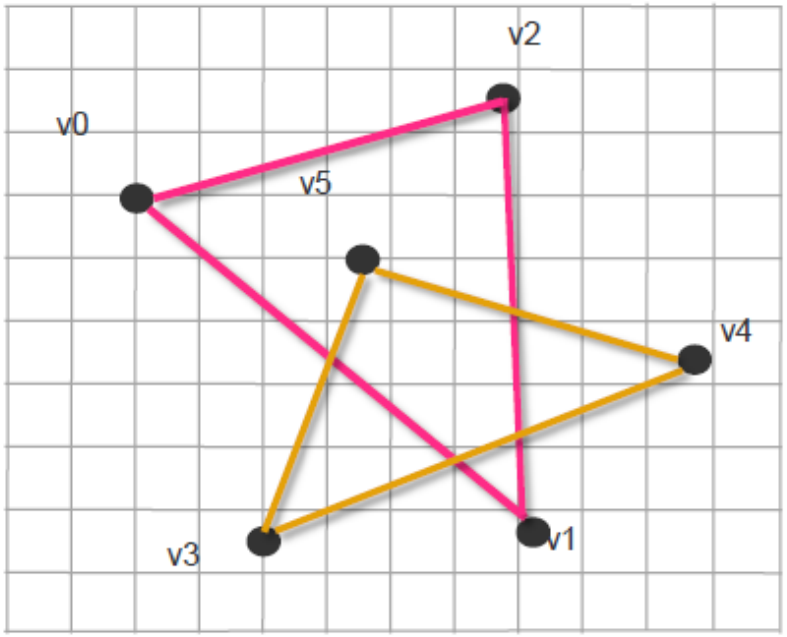



$$S_{v0} = M_p * M_{mv} * v_0$$

# Primitive Processing



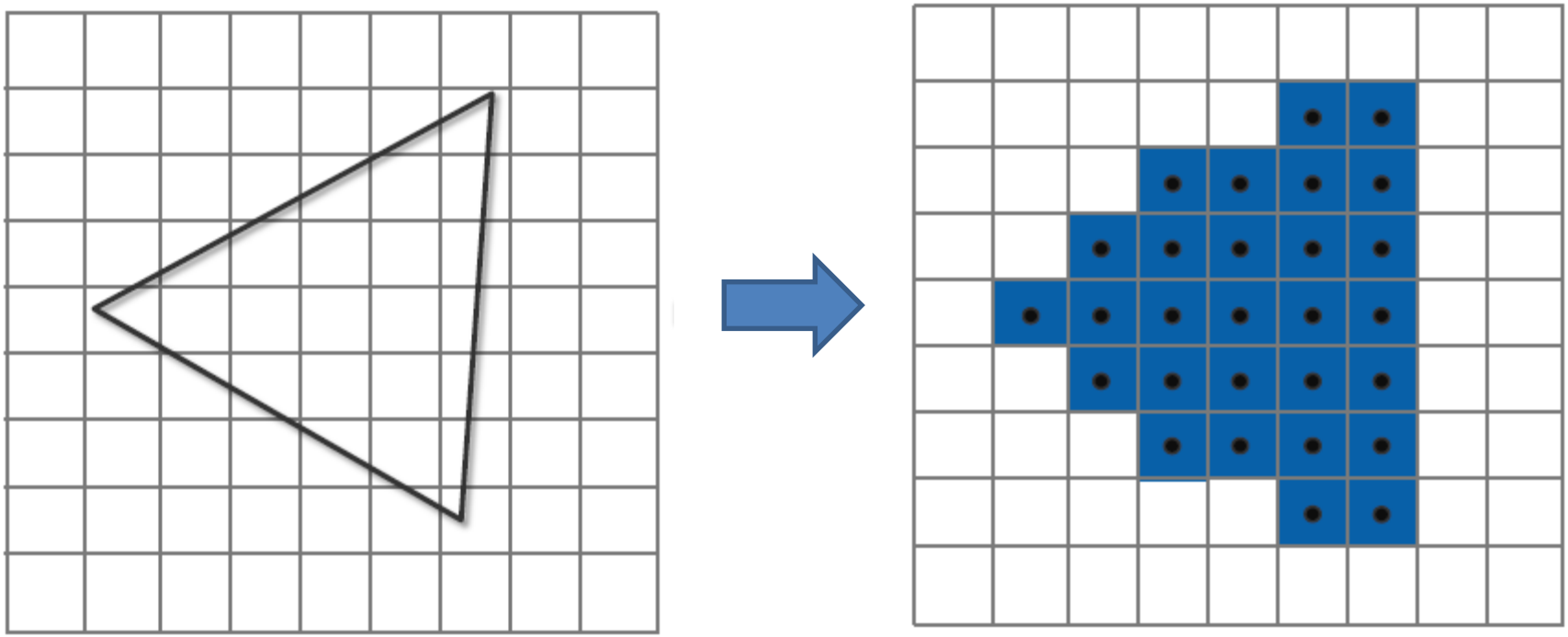



vertices

primitive

# Rasterization

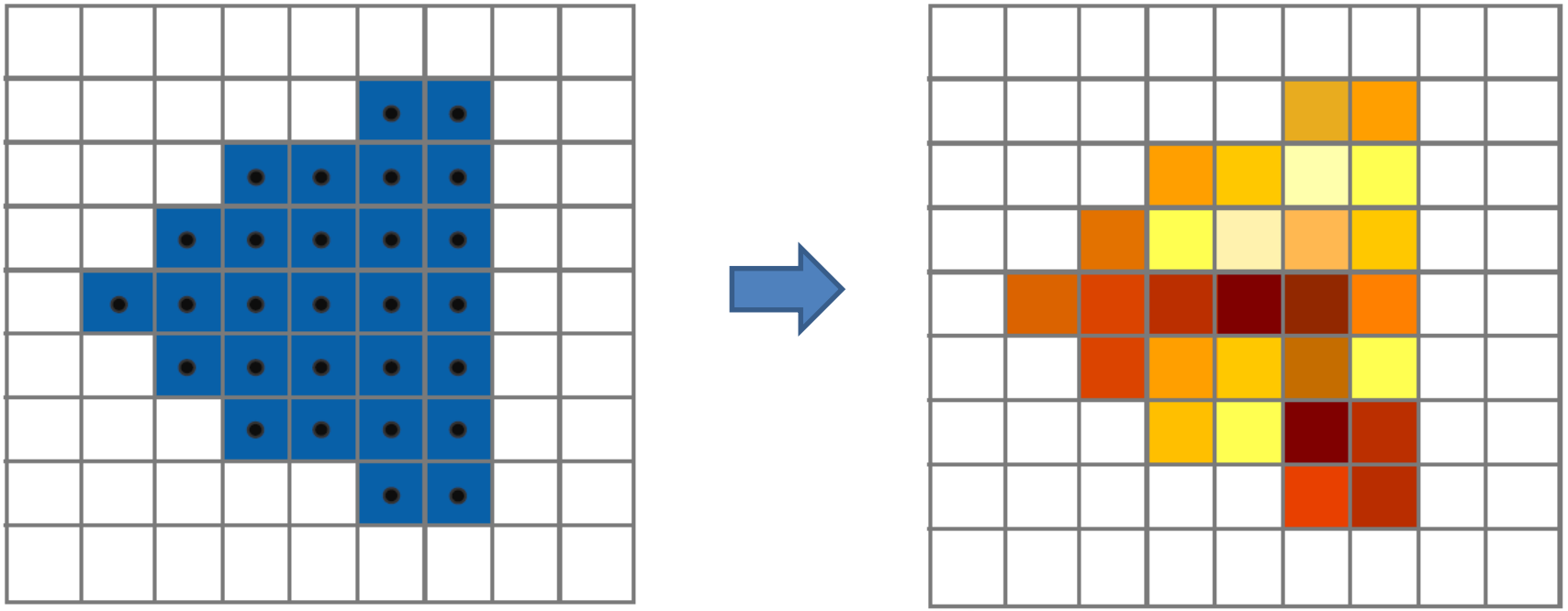
- **Converting vector format into pixel format**



# Fragment Processing

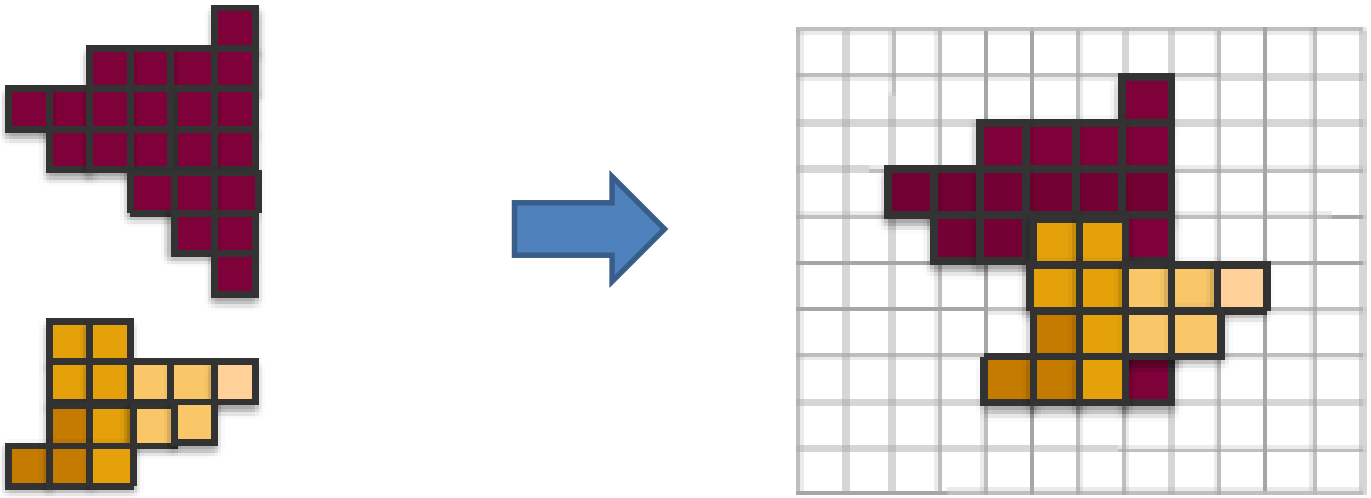
- **Generated by each primitive**
- **Input : rasterization of each primitive**
- **Output : each fragment (pixel) color**
- **Programmable stage**

# Fragment Processing



# Frame Buffer

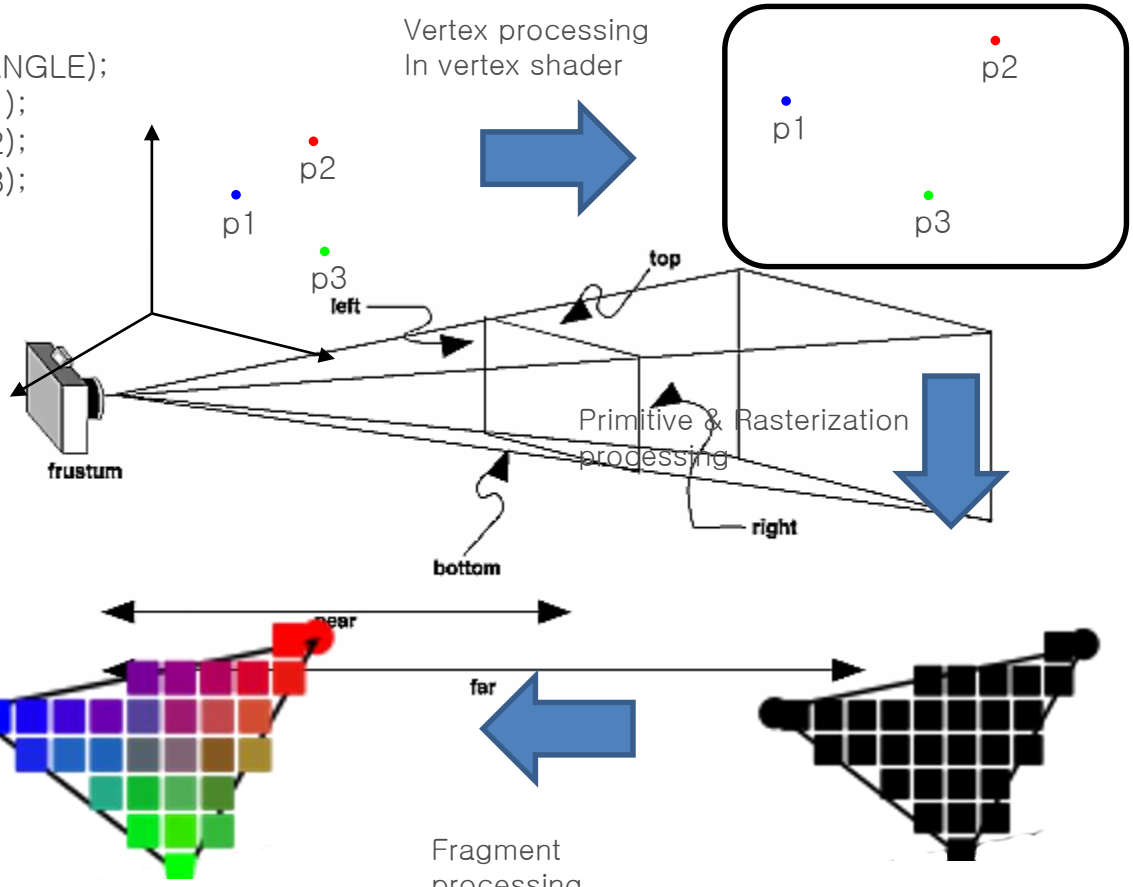
- Locate the pixels using z-buffer



# Overview of Process pipeline

- In cpu

```
glBegin(GL_TRIANGLE);
glVertex(x1,y1,z1);
glVertex(x2,y2,z2);
glVertex(x3,y3,z3);
glEnd();
```



# Shader Programming Start

# VGL Connect guide

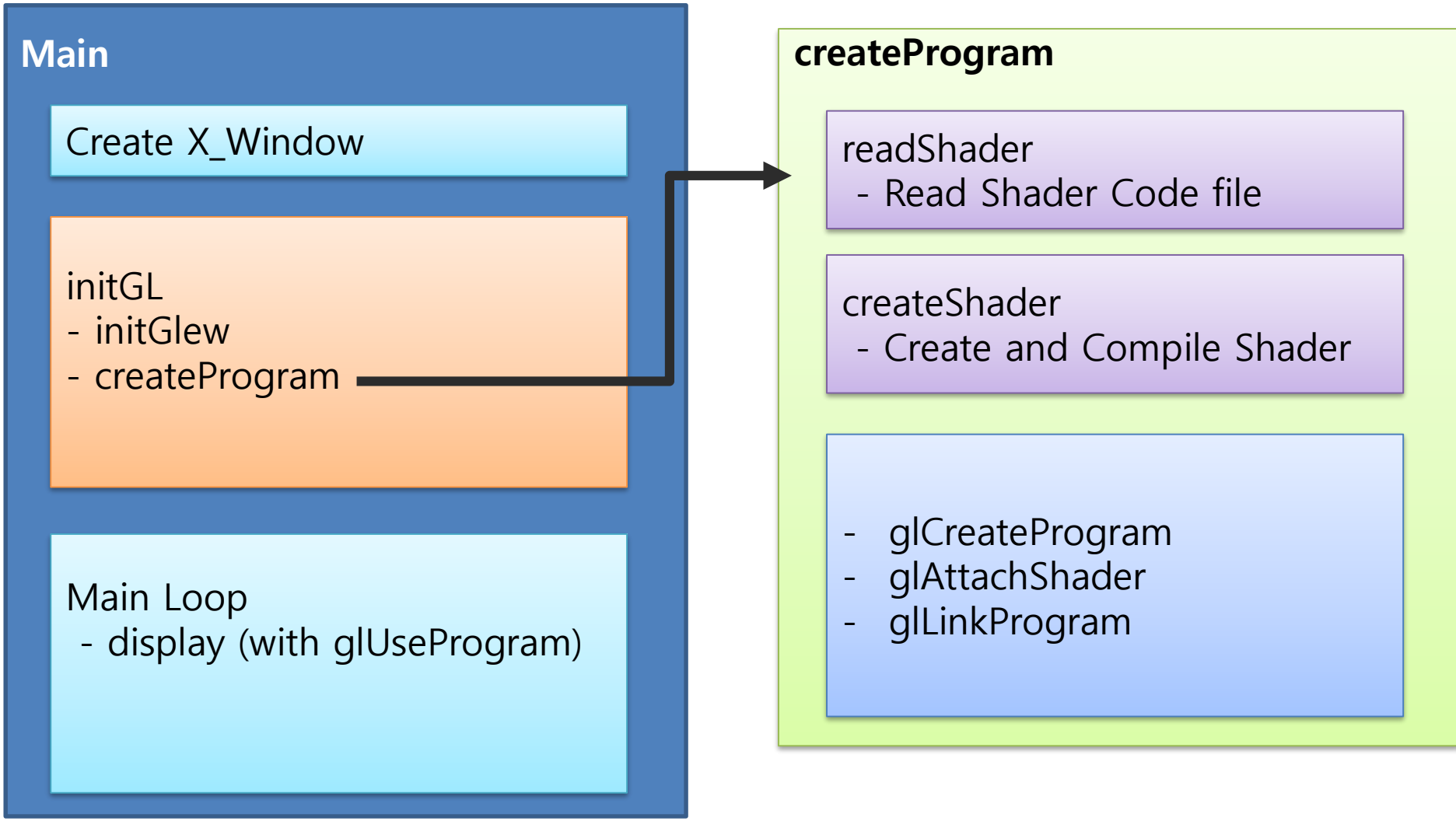
- If you have some error with VGL, follow this order.

1. Terminate Putty and Xming
2. Launch Xming
3. Launch PuTTY and connect GPU Server
   -IP: 163.152.20.246
4. Connect vgl
   -Command: vglconnect [User ID]@163.152.20.246
5. Graphical execution launch with “vglrun”
   Ex. vglrun xclock, vglrun ./EXE

# Shader Basic Coding Exercise

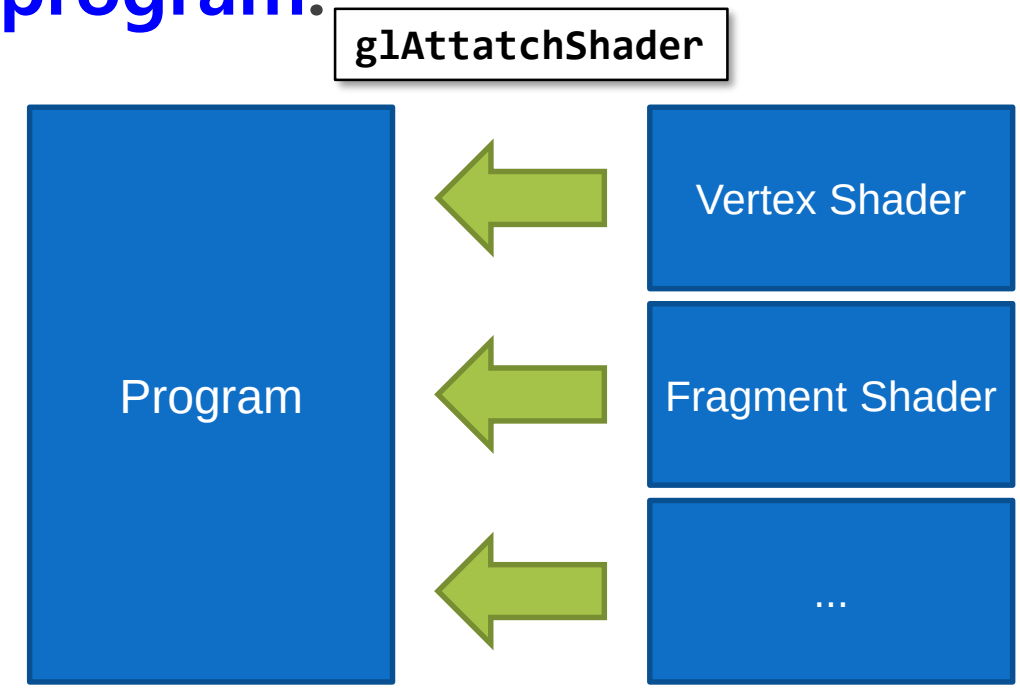
- **Copy Sample Skeleton Code**
  - vglconnect [Uesr ID]@163.152.20.246
  - cp –r /home/share/GLSLBasic ./
  - cd GLSLBasic
- **Notepad: shader code 작성**
  - Fragment.glsl
  - Vertex.glsl
- **Compile program**
  - make
  - vglrun ./EXE

# Basic Pattern



# Shader and Program

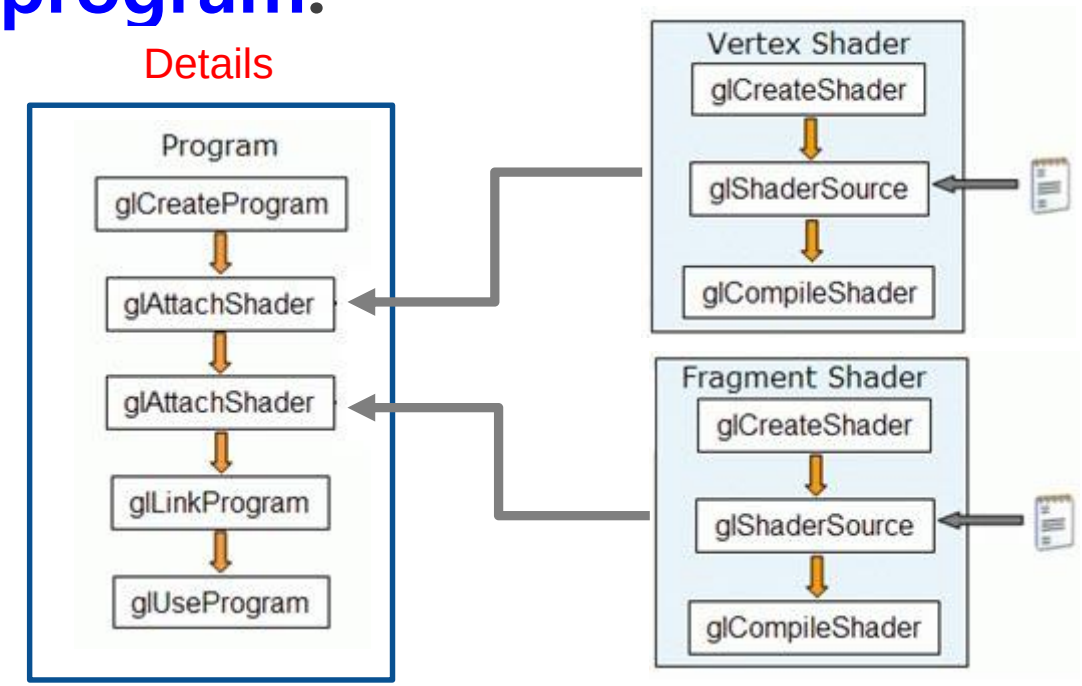
- Each **shader** (vertex & fragment) is like **a C module**, and it must be compiled separately, as in C.
- The set of compiled shaders, is then linked into **a program**.



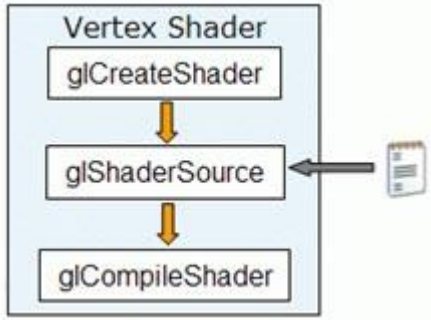
# Shader and Program

- Each shader (vertex & fragment) is like a C module, and it must be compiled separately, as in C.
- The set of compiled shaders, is then linked into a program.

Details



# Creating Shader




```
GLuint shader;
shader = glCreateShader(type);
glShaderSource(shader, 1, (const char**)src, NULL);
glCompileShader(shader);
printShaderInfoLog(shader);
```

Type can be one of

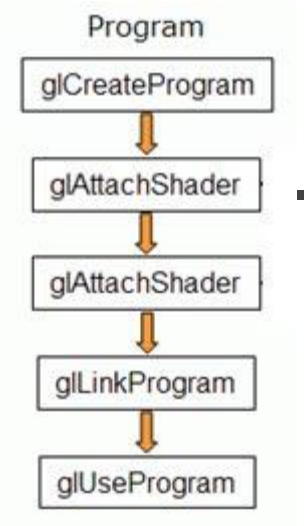
- **GL_VERTEX_SHADER**, **GL_FRAGMENT_SHADER**
- GL_GEOMETRY_SHADER
- GL_TESS_CONTROL_SHADER, GL_TESS_EVALUATION_SHADER

# Printing Shader Log

- If you want to print compilation warning/errors,

```
void printShaderInfoLog(GLuint shader) {
    int len = 0;
    int charsWritten = 0;
    char* infoLog;
    glGetShaderiv(shader, GL_INFO_LOG_LENGTH, & len);
    if (len > 0) {
        infoLog = (char*)malloc(len);
        glGetShaderInfoLog(shader, len, &charWritten, infoLog);
        printf("%s\n", infoLog);
        free(infoLog);
    }
}
```

# Creating Program




```
GLuint program = glCreateProgram();
glAttachShader(program, vertShader);
glAttachShader(program, fragShader);
glLinkProgram(program);
printProgramInfoLog(program);
```

# Example Program

```
/*Global Variables*/
GLuint program, vertShader, fragShader = 0;

void main(int argc, char *argv[]) {
        /*Create Window*/
        initGL();
        while(1){
                display();
                /*Event Handle*/
        }
}
```

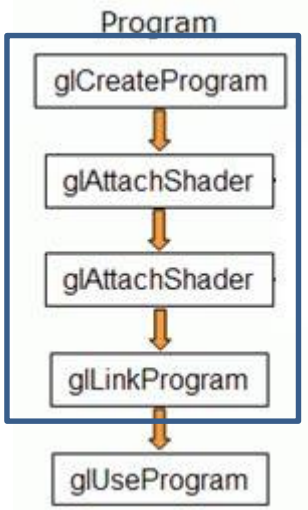
# initGL@Main

```
void initGL()
{
        glewInit(); //glew Initialize Function;
        createProgram(); //Create Shader Program
}
```

Run-time compilation w/ shader source file

# createProgram@initGL

```
void createProgram(){
        char* vert;
        char* frag;
        vert = readShader("Vertex.glsl");
        frag = readShader("Fragment.glsl");
        vertShader = createShader(vert, GL_FRAGMENT_SHADER);
        fragShader = createShader(frag, GL_FRAGMENT_SHADER);
        GLuint p = glCreateProgram();
        glAttachShader(p, vertShader);
        glAttachShader(p, fragShader);
        glLinkProgram(p);
        program = p;
}
```



# Read Shader Code@createProgram

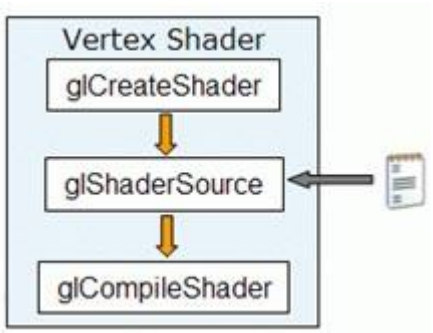
```
char* readShader(char *filename){
   char *buffer = NULL;
   int string_size, read_size;
   FILE *handler = fopen(filename, "r");
   if (handler){
        fseek(handler, 0, SEEK_END); // Seek the last byte of the file
        string_size = ftell(handler); // filesize
        rewind(handler); // go back to the start of the file
       // Allocate a string that can hold it all
       buffer = (char*) malloc(sizeof(char) * (string_size + 1) );
       // Read it all in one operation
       read_size = fread(buffer, sizeof(char), string_size, handler);
       /*Continued*/
```

# Read Shader Code Cont.

```
        /*Continued*/
        // fread doesn't set it so put a \0 in the last position
      buffer[string_size] = '\0';
      if (string_size != read_size){
          // Something went wrong, throw away the memory and set NULL
          free(buffer);
          buffer = NULL;
      }
      fclose(handler);
   }
   return buffer;
}
```

# createShader@createProgram

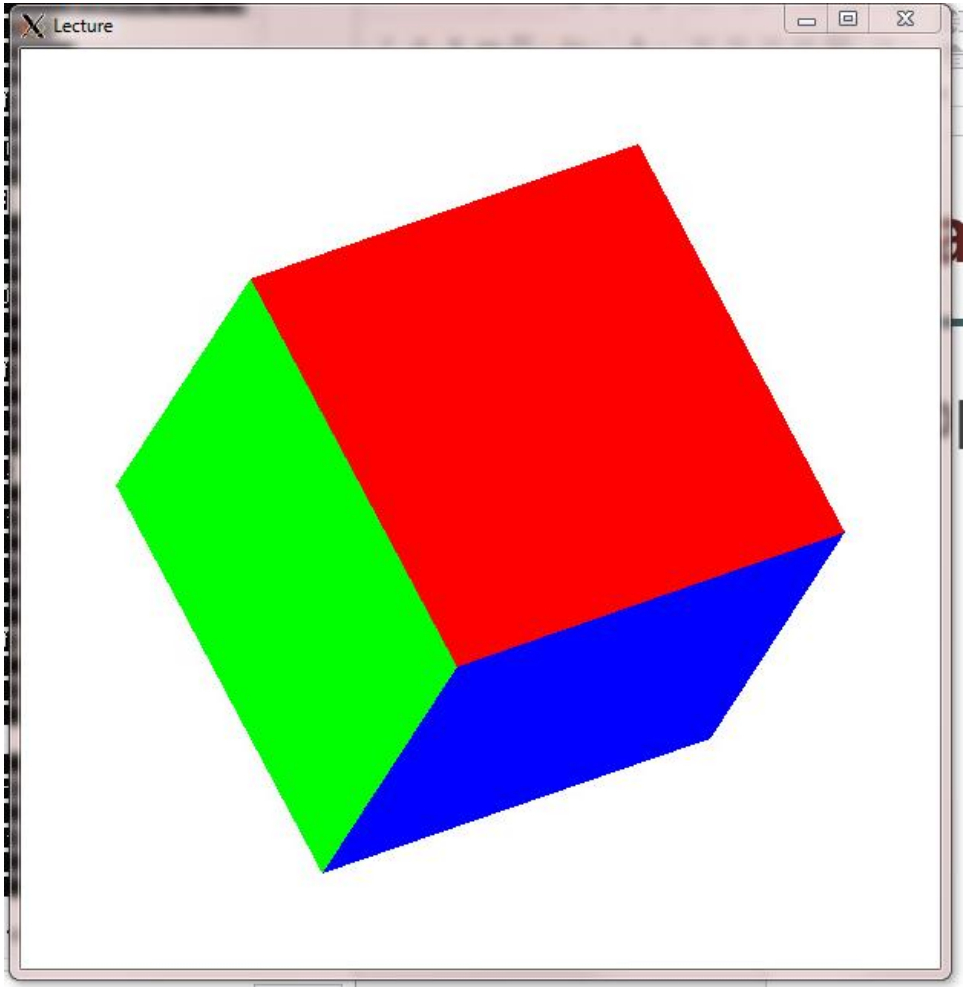
```
GLuint createShader(char* src, GLenum type)
{
        GLuint shader;
        shader = glCreateShader(type);
        glShaderSource(shader, 1, &src, NULL);
        glCompileShader(shader);
        return shader;
}
```



# display@main

```
void display()
{
        glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
        glUseProgram(program);
        /*Draw Call*/
        glUseProgram(0);
        glXSwapBuffers(dpy, win);
}
```

# Draw Cube with GLSL



# Draw Cube – Draw Call

```
void display(){
        /*init display*/
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        glRotatef(40.0, 1.0, -1.0, 1.0);
        glUseProgram(program);
        glBegin(GL_QUADS);
        glColor3f(1.0f, 0.0f, 0.0f);
        glVertex3f(-0.5f, -0.5f, -0.5f); glVertex3f( 0.5f, -0.5f, -0.5f);
        glVertex3f( 0.5f,  0.5f, -0.5f); glVertex3f(-0.5f,  0.5f, -0.5f);
        glColor3f(0.0f, 1.0f, 0.0f);
        glVertex3f(0.5f, -0.5f, -0.5f); glVertex3f(0.5f, -0.5f,  0.5f);
        glVertex3f(0.5f,  0.5f,  0.5f); glVertex3f(0.5f,  0.5f, -0.5f);
        /*Draw Other Faces*/
        glEnd();
        glUseProgram(0);
        glXSwapBuffers(dpy, win);
}
```

# Draw Cube - Shader Codes: Vertex

```
#version 130

void main(){
	gl_Position = gl_ModelViewMatrix*gl_Vertex;
	gl_FrontColor = gl_Color;
}
```

# Draw Cube - Shader Codes: Fragment

```
#version 130

void main(){
	gl_FragColor = gl_Color;
}
```

# VGL Connect guide

- If you have some error with VGL, follow this order.

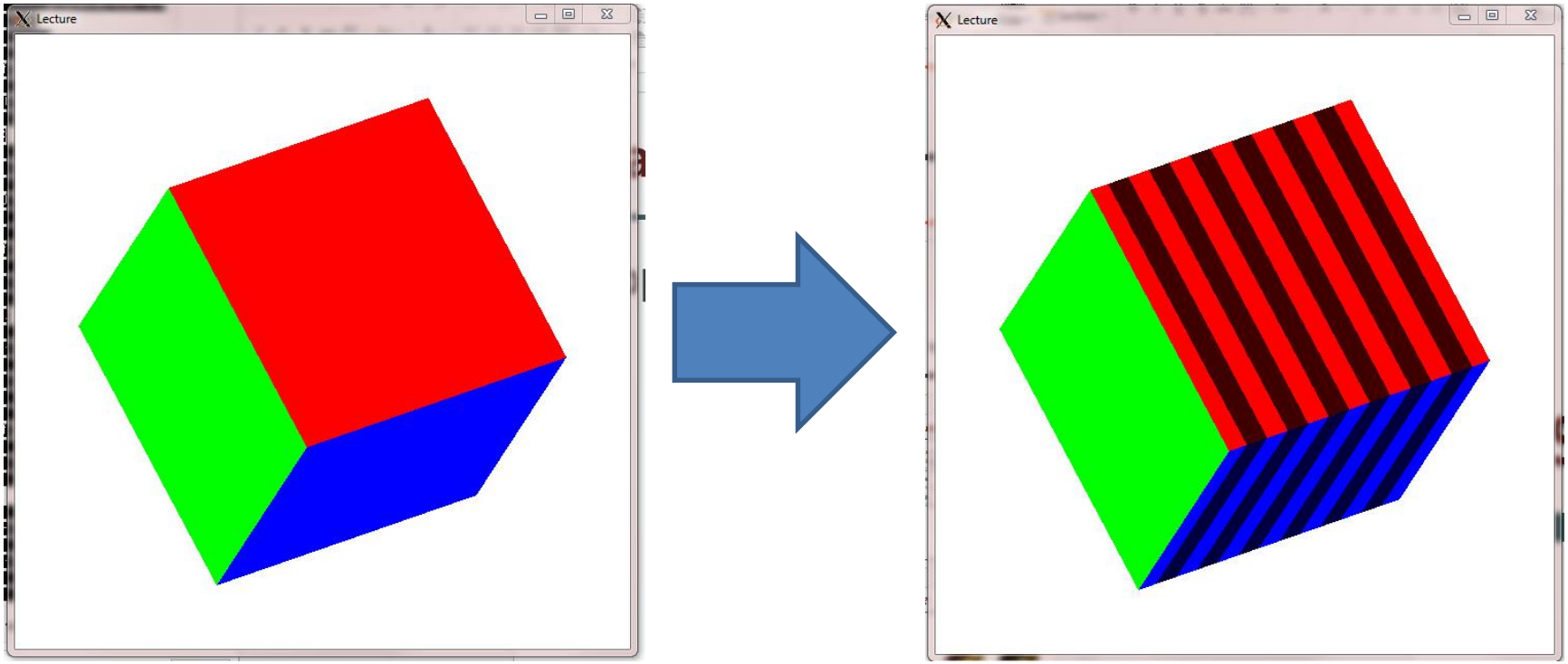
0. disconnect vgl, Xming out & in, reconnect vgl
1. Terminate Putty and Xming
2. Launch Xming
3. Launch PuTTY and connect GPU Server
   -IP: 163.152.20.246
4. Connect vgl
   -Command: vglconnect [User ID]@163.152.20.246
5. Graphical execution launch with “vglrun”
   Ex. vglrun xclock, vglrun ./EXE

# Striped Cube Exarcise

- **Notepad: shader code 수정**
  - Fragment.glsl
  - Vertex.glsl

- **Don’t need to compile again**
  - vglrun ./ExE

# Single Striped Cube Drawing

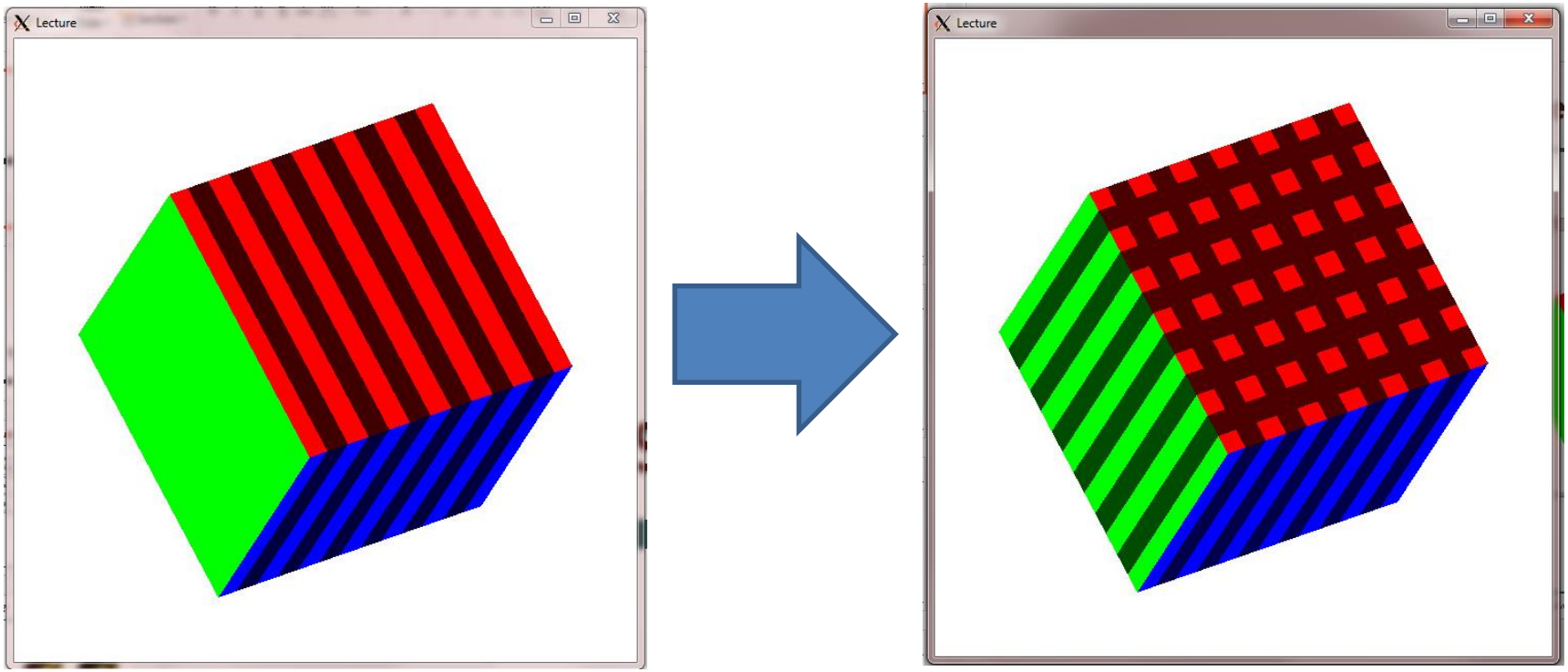


# Draw Striped Cube - Shader Codes

```
// Vertex shader
#version 130
varying float x;
void main(){
        gl_Position = gl_ModelViewMatrix*gl_Vertex;
        gl_FrontColor = gl_Color;
        x = gl_Vertex.xyz;
}
// fragment shader
varying float x;
void main(){
        float stripe = cos(40.0*x);
        if (stripe < 0.0) stripe = 0.25;
        else stripe = 1.0;

        gl_FragColor = stripe*gl_Color;
}
```

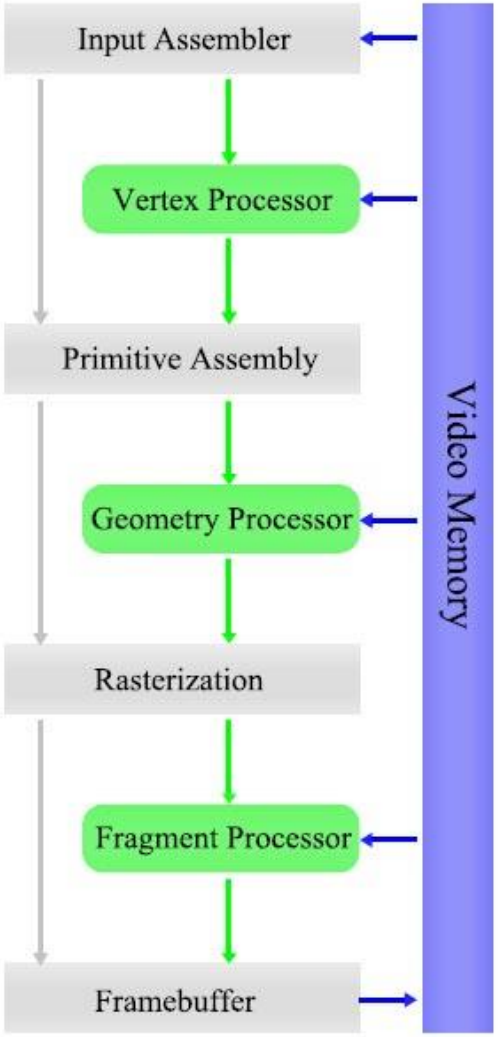
# Double Striped Cube



# Draw Striped Cube - Shader Codes

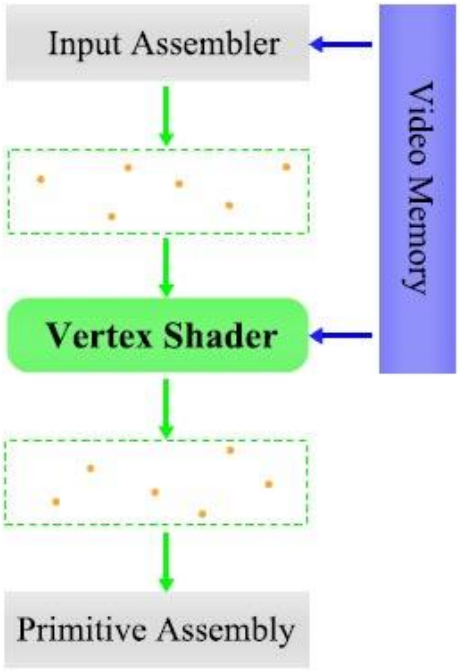
```
// Vertex shader
#version 130
varying vec2 pos;
void main(){
	gl_Position = gl_ModelViewMatrix*gl_Vertex;
	gl_FrontColor = gl_Color;
	pos = gl_Vertex.xy;
}
// fragment shader
varying vec2 pos;
void main(){
	if(cos(pos.x*40.0f)>0.0f&&cos(pos.y*40.0f)>0
.0f)
		gl_FragColor = gl_Color;
	else
		gl_FragColor = gl_Color*0.3f;
}
```

# GLSL Syntax relies on the Pipeline



# Input and Output of Vertex Shader

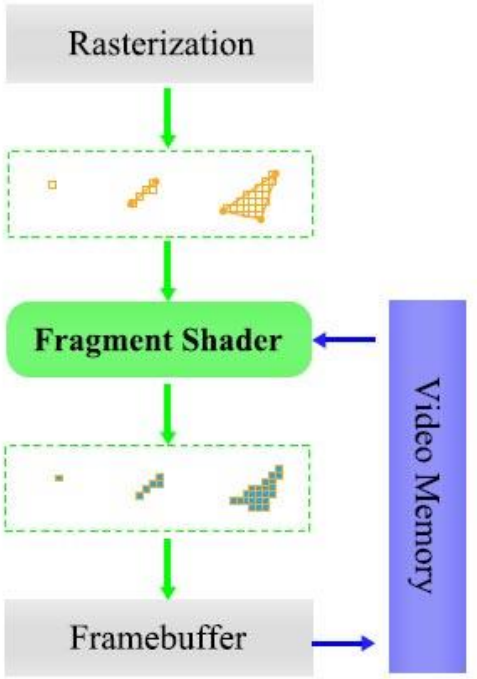
- **Input**
  - Per vertex attributes.
    - Examples: **`gl_Vertex`** , **`gl_Color`** , **`gl_Normal`**
  - OpenGL states and user defined uniform variable such as **`gl_ModelViewMatrix`**
- **Output**
  - **`gl_Position`**: coordinates in the canonical space
  - Other values to be interpolated for each fragment during the rasterization
  - **`gl_FrontColor, gl_BackColor,,, etc`**



# Input and Output of Fragment Shader

- **Input**
  - Interpolated values from the rasterizer
    (Position, Normal, Color,,, etc)
  - OpenGL states and user defined uniform variables(Light, Texture,,,, etc)

- **Output**
  - Pixel values to be processed by pixel tests and written to the framebuffer
    - Examples: `gl_FragColor`, `glFragDepth`



# GLSL Scope of Variables

- **Global**
  - The variables defined outside the shader function.
  - Vertex and fragment shaders can share global variables that must have the same types and names as long as they are in the same program.
- **Local**
  - The variables defined within the shader function.
  - The scope limited inside of the function.

# GLSL Qualifiers

- **Storage Qualifiers**
  - **`uniform:`** value does not change across the primitive (defined with a **`glBegin`** and **`glEnd`** block) being processed
  - **`attribute:`** values which are input to the vertex shader.
  - **`varying:`** Output of vertex shader, input of fragment shader.

  - Instead of attribute/varying, the following are used in later versions
    - **`in:`** input to the shader (GLSL1.3 or above)
    - **`out:`** output from the shader (GLSL1.3 or above)
    - **`layout:`** variable for which its attribute location of grouped interface can be specified. (GLSL 1.5 or above)
- **Parameter Qualifiers (used for function calls)**
  - **`in, out, inout`**

# `uniform` Qualifier

- **Global variables that change less often, e.g. per primitive or object**
  - May be used in both vertex and fragment shader
  - Read-only
  - Passed from the OpenGL application to the shaders.
  - Can not be set inside **`glBegin`** and **`glEnd`** block
  - Number of these variables is limited, but a lot more than attribute variables
- **Built-in:**
  - `uniform mat4 gl_ModelViewMatrix;`
  - `uniform mat4 gl_ProjectionMatrix;`
  - `uniform mat4 gl_ModelViewProjectionMatrix;`
  - `uniform mat4 gl_TextureMatrix[n];`
  - `...`
- **User-defined uniform variables (e.g.)**
  - `uniform float myCurrentTime;`

# Usage of Uniform Variables

- Uniform variables work as an interface between C/C++ code and the shaders.

```
// in C/C++ code
float frequency =40;
GLint loc = glGetUniformLocation(program, “freq”);
glUniform1f(loc, frequency);
    // put the value of C++ var frequency to the
    // location pointed by loc, i.e., to freq.
    /*draw Call*/
```

```
// in fragment shader code
uniform float freq;
void main() {
    ...
     if(cos(pos.x*freq)>0.0f&&cos(pos.y*freq)>0.0f)
    ...
}
```

# `attribute` Qualifier

- **Global variables that change per vertex**
  - Can be used only in the vertex shader, and read-only.
  - Passes values from the application to vertex shaders
  - Number of them is limited, e.g. 32 (hardware-dependent)
- **Built-in:**
  - `gl_Vertex`
  - `gl_Color`
  - `gl_Normal`
  - `gl_MultiTexCoord0~7`
  - `...`
- **User can define them in the vertex shader (e.g.)**
  - `attribute float temperature;`
  - `attribute vec3 velocity;`

# Usage of Attribute Variables

- Attribute variables also work as an interface between C/C++ code and the shaders.

```
// in C/C++ code
Vert[]= {……}
glUseProgram(program);
GLint loc = glGetAttribLocation(program, “InVertex”);
glEnableVertexAttribArray(loc);
glVertexAttribPointer(loc, 4, GL_FLOAT, 0, 0, verts);
...
```

```
// in vertex shader code
attribute vec4 InVertex;
void main() {
    ...
     gl_Position = gl_ModelviewProjectionMatrix *
                   InVertex;
    ...
}
```

# `varying` Qualifier

- **Used for passing data from the vertex shader to the fragment shader (for interpolation).**
  - Writable in the vertex shader.
  - Read-only in the fragment shader.
- **Built-in:**
  - `gl_FrontColor`
  - `gl_BackColor`
  - `gl_TexCoord[]`
  - `...`
- **User-defined varying variables**
  - Requires matching definitions in the vertex and fragment shader.
  - For example, `varying vec3 normal` should appear in both shaders.

# Usage of Varying Variables

```
[Vertex Shader]
#version 130

varying vec2 pos;
void main(){
        /*……*/
        pos = gl_Vertex.xy;
}
```

"pos" is transferred to fragment shader with interpolation

```
[Fragment Shader]
#version 130

varying vec2 pos;
void main(){
        if(cos(pos.x*40.0f)>0.0f&&cos(pos.y*40.0f)>0.0f)
        /*……*/
}
```

# GLSL Built-In Data Types

- **Scalar types**
  - **`bool, int, float`**
  - No implicit conversion between types
  - **`uint`** and **`double`** are supported in GLSL 1.3 and 4.0, respectively.
- **Vectors**
  - Float: **`vec2, vec3, vec4`**
  - Int: **`ivec`**, Bool: **`bvec`** Also **`(uvec, dvec)`**
  - C++ style constructors

```
vec3 a = vec3(1.0, 2.0, 3.0);
vec2 b = vec2(a);
```

Red parts are not available in GLSL 1.2

- **Matrices**
  - **`mat2,mat3,mat4 (dmat2,dmat3,dmat4,matnxm,dmatnxm)`**
  - Stored by columns order (OpenGL convention)

```
mat2 m = mat2(1.0, 2.0, 3.0, 4.0);
```

# GLSL Operators

- Matrix multiplication is done with * operator.

```
mat4 m4, n4;
vec4 v4;
m4 * v4; // a vec4
v4 * m4; // a vec4
m4 * n4; // a mat4
```

# Selection and Swizzling

- Can refer to vector or matrix elements by using [] operator or selection(.) operator with

```
x, y, z, w
r, g, b, a
s, t, p, q
vec4 v4 = vec4(1.0, 2.0, 3.0, 4.0);
```

`v4[2], v4.b, v4.z, v4.p` are the same

- Swizzling operator lets us manipulate components in the following way:

```
vec4 a;
a.yz = vec2(1.0, 2.0);
a.zw = vec2(2.0, 4.0);
```

# Arrays and Structs

- **GLSL can have arrays**
  - Only one-dimensional array is allowed.

```
float x[4];
vec3 colors[4]; colors[0] = vec3(1.0, 1.0, 0.0);
mat4 matrices[3];
```

  - Arrays know the number of elements they contain

```
array_name.length();
```

- **GLSL can also have C-like structs**

```
struct light {
  vec4 position;
  vec3 color;
};
```

- **There are no pointers in GLSL**

# Built-in Functions

- Trigonometry
  - `sin, cos, tan, asin, acos, atan,...`
- Exponential
  - `pow, exp2, log2, sqrt, inversesqrt,...`
- Common
  - `abs, floor, sign, ceil, min, max, clamp,...`
- Geometric
  - `length, dot, cross, normalize, reflect, distance,...`
- Texture lookup
  - `texture1D, texture2D, texture3D, textureCube,...`
- Noise functions
  - `noise1, noise2, noise3, noise4`
- Misc.
  - `ftransform`: transform vertex coordinates to clipping space by modelview and projection matrix (deprecated in GLSL1.3)