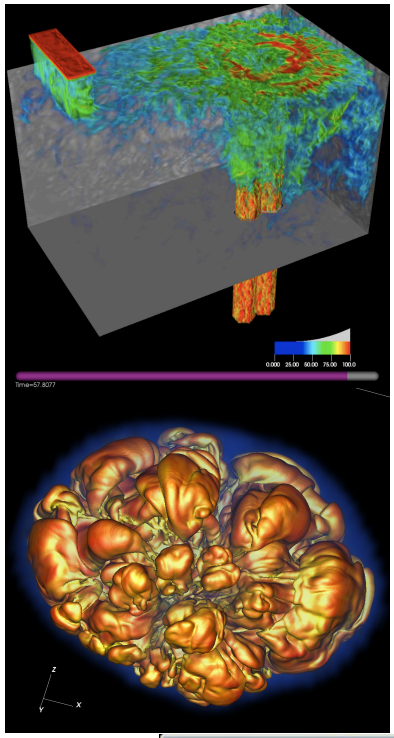
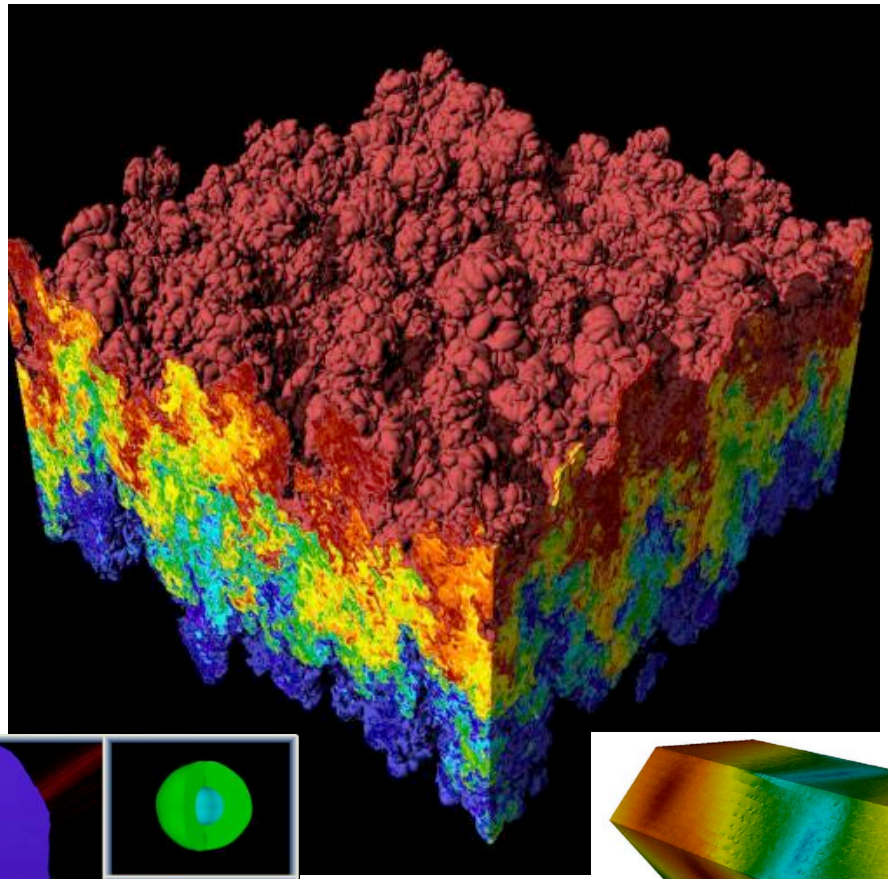
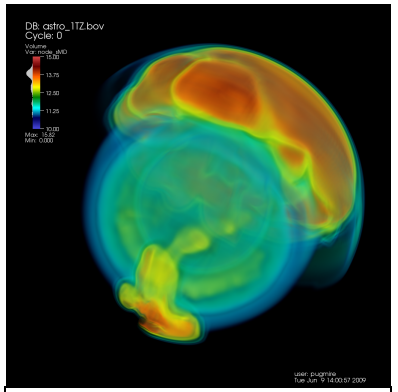
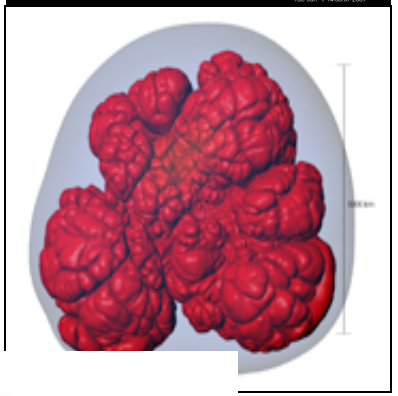
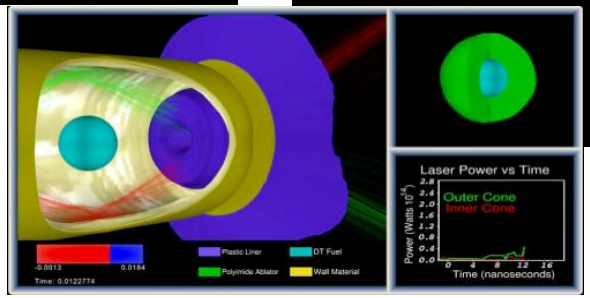
CIS 441/541: Intro to Computer Graphics
Lecture 10: OpenGL - Shaders

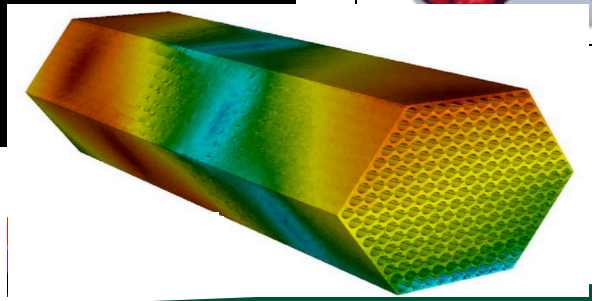












May 6, 2021

Hank Childs, University of Oregon

Office Hours

Published | Edit

How to access Office Hours

Hank Childs

Apr 4 at 2:02pm

All Sections

Hi Everyone,

We currently have an asymmetry for accessing Hank and Abhishek's Office Hours.

As of now, Abhishek's are always at: COVERED UP (THIS IS POSTED ONLINE)

And Hank's are accessible via the Zoom Meetings area in Canvas.

Let's chat on Tuesday about the most standard way to do this.

Finally, here is the OH schedule again:

Monday (Abhishek): 10am-11am
Tuesday (Abhishek): 945am-1045am
Wednesday (Hank): 230pm-330pm
Thursday (Abhishek): 945am-1045am

Best,
Hank

Layout of Simple OpenGL Program

- Set up windows
- Set up things to render (VBOs)
- Set up how to render (shaders)
- While (1)
  - Accept events, make changes
    - New camera positions, new geometry, etc.
  - Render

The remainder of this lecture and Thursday's lecture are made up of 4 parts

1) Set up windows
2) Doing a render
3) Set up things to render (VBOs)
4) Set up how to render (shaders) (Thursday)

The remainder of this lecture and Thursday's lecture are made up of 4 parts

1) Set up windows
2) Doing a render
3) Set up things to render (VBOs)
4) Set up how to render (shaders) (Thursday)

How to Use Shaders

- You write a shader program: a tiny C-like program
- You write C/C++ code for your application
- Your application loads the shader program from a text file (or just contains it as a string)
- Your application sends the shader program to the OpenGL library and directs the OpenGL library to compile the shader program
- If successful, the resulting GPU code can be attached to your (running) application and used
- It will then supplant the built-in GL operations

How to Use Shaders: Visual Version

Project2A' C++ code

g++

Project2A' binary

sends "char *" version of program to GL via function call

OpenGL library

reads text file when running

shader program

Program is used on GPU to support Project2A' binary

OpenGL compiles program, binary made just for the current execution

shader program is a binary

Compiling Shader

```
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
std::string vertexProgram = loadFileToString("vs.glsl");
const char *vertex_shader_source = vertexProgram.c_str();
GLint const vertex_shader_length = strlen(vertex_shader_source);
glShaderSource(vertexShader, 1, &vertex_shader_source, &vertex_shader_length);
glCompileShader(vertexShader);
GLint isCompiledVS = 0;
glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &isCompiledVS);
```

Compiling Shader: inspect if it works

```
if(isCompiledVS == GL_FALSE)
{
   cerr << "Did not compile VS" << endl;

   GLint maxLength = 0;
   glGetShaderiv(vertexShader, GL_INFO_LOG_LENGTH, &maxLength);

   // The maxLength includes the NULL character
   std::vector<GLchar> errorLog(maxLength);
   glGetShaderInfoLog(vertexShader, maxLength, &maxLength, &errorLog[0]);
   cerr << "Vertex shader log says " << &(errorLog[0]) << endl;
   exit(EXIT_FAILURE);
}
```

Compiling Multiple Shaders

```
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
std::string vertexProgram = loadFileToString("vs.glsl");
const char *vertex_shader_source = vertexProgram.c_str();
GLint const vertex_shader_length = strlen(vertex_shader_source);
glShaderSource(vertexShader, 1, &vertex_shader_source, &vertex_shader_length);
glCompileShader(vertexShader);
GLint isCompiledVS = 0;
glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &isCompiledVS);

if(isCompiledVS == GL_FALSE)
{
   cerr << "Did not compile VS" << endl;

   GLint maxLength = 0;
   glGetShaderiv(vertexShader, GL_INFO_LOG_LENGTH, &maxLength);

   // The maxLength includes the NULL character
   std::vector<GLchar> errorLog(maxLength);
   glGetShaderInfoLog(vertexShader, maxLength, &maxLength, &errorLog[0]);
   cerr << "Vertex shader log says " << &(errorLog[0]) << endl;
   exit(EXIT_FAILURE);
}

GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
std::string fragmentProgram = loadFileToString("fs.glsl");
const char *fragment_shader_source = fragmentProgram.c_str();
GLint const fragment_shader_length = strlen(fragment_shader_source);
glShaderSource(fragmentShader, 1, &fragment_shader_source, &fragment_shader_length);
glCompileShader(fragmentShader);
GLint isCompiledFS = 0;
glGetShaderiv(fragmentShader, GL_COMPILE_STATUS, &isCompiledFS);
```

Attaching Shaders to a Program

```
GLuint program = glCreateProgram();
glAttachShader(program, vertexShader);
glAttachShader(program, fragmentShader);

glLinkProgram(program);

glDetachShader(program, vertexShader);
glDetachShader(program, fragmentShader);
```

Inspecting if program link worked…

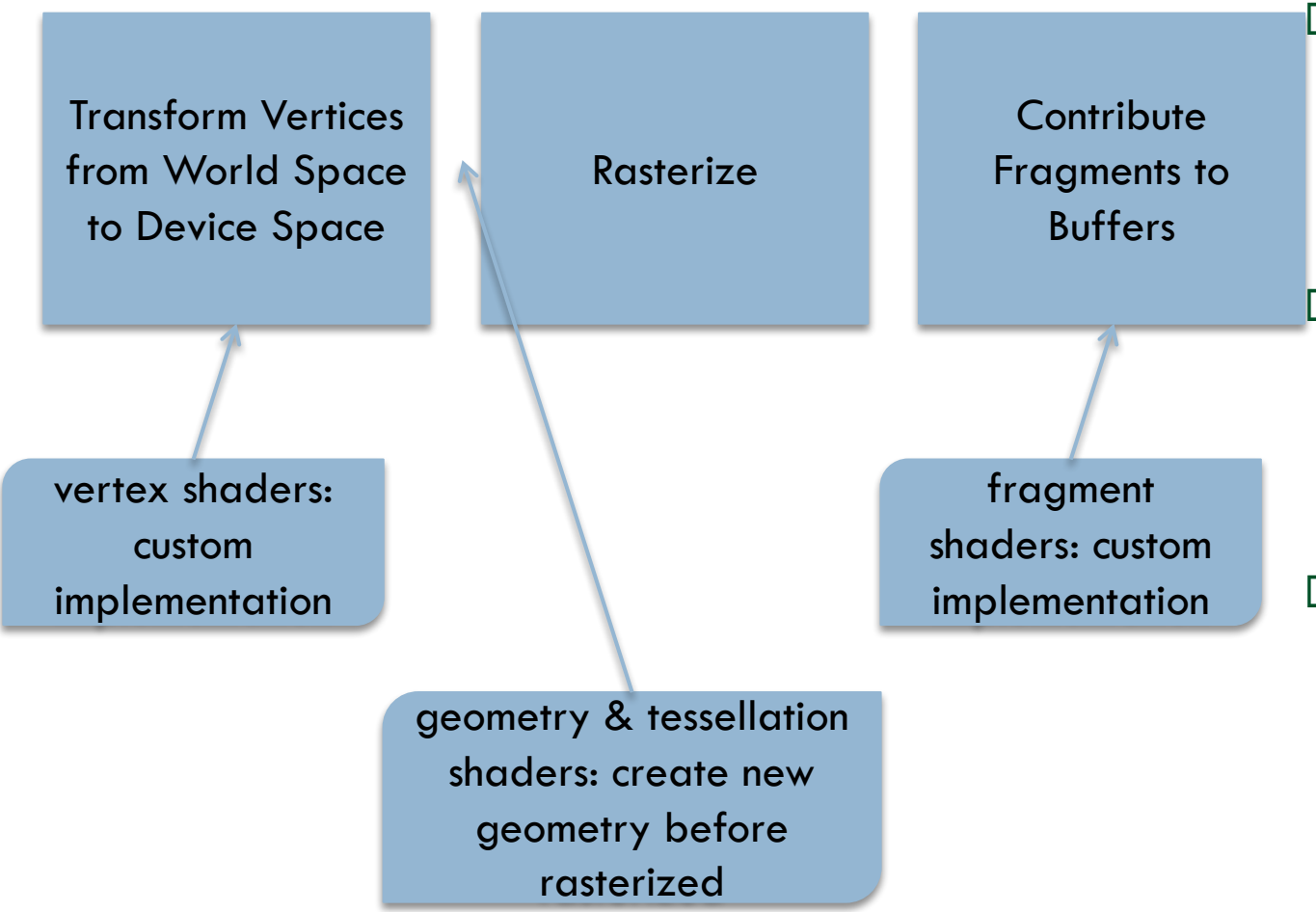
```
GLint isLinked = 0;
glGetProgramiv(program, GL_LINK_STATUS, (int *)&isLinked);
if(isLinked == GL_FALSE)
{
   GLint maxLength = 0;
   glGetProgramiv(program, GL_INFO_LOG_LENGTH, &maxLength);

   //The maxLength includes the NULL character
   std::vector<GLchar> infoLog(maxLength);
   glGetProgramInfoLog(program, maxLength, &maxLength, &infoLog[0]);
   cerr << "Couldn't link" << endl;
   cerr << "Log says " << &(infoLog[0]) << endl;

   exit(EXIT_FAILURE);
}
```

Starter Code Has 4 Shaders

- phase2VertexShader
- phase2FragmentShader
- phase345VertexShader
- phase345FragmentShader

- Phase 2 variants are complete and work
- Phase 345 variants are what you will implement

Reminder: How Shaders Fit Into the Graphics Pipeline



- You can have 0 or 1 of each shader type
- Vertex & fragment: very common (2A)
- Geometry & tessellation: less common
  - adaptive meshing

4 Elements to a Shader Program

Declare GLSL version (GL Shader Language)

Declare inputs to program

Declare outputs of program

```
void main() {
    C-like code that operates
    on inputs to make outputs
}
```

4 Elements to a Shader Program

Declare GLSL version (GL Shader Language)

Declare inputs to program

Declare outputs of program

```
void main() {
    C-like code that operates
    on inputs to make outputs
}
```

Declare GLSL version

- Syntax: #version 400
- Old versions: may be deprecated
- New versions: may not be available
- 400 is a good choice – works everywhere
  - And what we use for this class

OpenGL Version	GLSL Version
2.0	1.10
2.1	1.20
3.0	1.30
3.1	1.40
3.2	1.50

For all versions of OpenGL 3.3 and above, the corresponding GLSL version matches the OpenGL version. So GL 4.1 uses GLSL 4.10.

From: Khronos.org

4 Elements to a Shader Program

Declare GLSL version (GL Shader Language)

Declare inputs to program

Declare outputs of program

```
void main() {
    C-like code that operates
    on inputs to make outputs
}
```

Declare Inputs to Program

layout (location = 0) in vec3 vertex_position;

- In words:
  - The array that was placed in location 0 is a vector of 3 floats
  - In my code, I will refer to this array as vertex_position
  - Regarding placement:
    - The placement was already done before the shader program executes
    - The program must accept the placement made by the VAO or shader program that proceeded it

Type Names

Scalars

The basic non-vector types are:

- `bool`: conditional type, values may be either `true` or `false`
- `int`: a signed, two's complement, 32-bit integer
- `uint`: an unsigned 32-bit integer
- `float`: an IEEE-754 single-precision floating point number
- `double`: an IEEE-754 double-precision floating-point number

  Warning: The specific sizes and formats for integers and floats in GLSL are only for GLSL 1.30 and above. Lower versions of GLSL may not use these exact specifications.

Vectors

Each of the scalar types, including booleans, have 2, 3, and 4-component vector equivalents. The *n* digit below can be 2, 3, or 4:

- `bvecn`: a vector of booleans
- `ivecn`: a vector of signed integers
- `uvecn`: a vector of unsigned integers
- `vecn`: a vector of single-precision floating-point numbers
- `dvecn`: a vector of double-precision floating-point numbers

Vector values can have the same math operators applied to them that scalar values do. These all perform the component-wise operations on each component. However, in order for these operators to work on vectors, the two vectors must have the same number of components.

From: Khronos.org

Declare Inputs to Program

```
layout (location = 0) in vec3 vertex_position;
layout (location = 1) in vec3 vertex_color;
```

- In words:
  - The array that was placed in location 0 is a vector of 3 floats
  - In my code, I will refer to this array as vertex_position
  - The array that was placed in location 1 is also a vector of 3 floats
  - In my code, I will refer to this array as vertex_color

Declare Outputs of Program

```
out vec3 color;
```

- In words:
  - My program will create a vector of3 floats
  - I will refer to this vector as color
  - It is at location 0, since I declared this first
  - The next shader program needs to know that color is placed in location 0 and is a vec3

4 Elements to a Shader Program

Declare GLSL version (GL Shader Language)

Declare inputs to program

Declare outputs of program

```
void main() {
    C-like code that operates
    on inputs to make outputs
}
```

C-like code

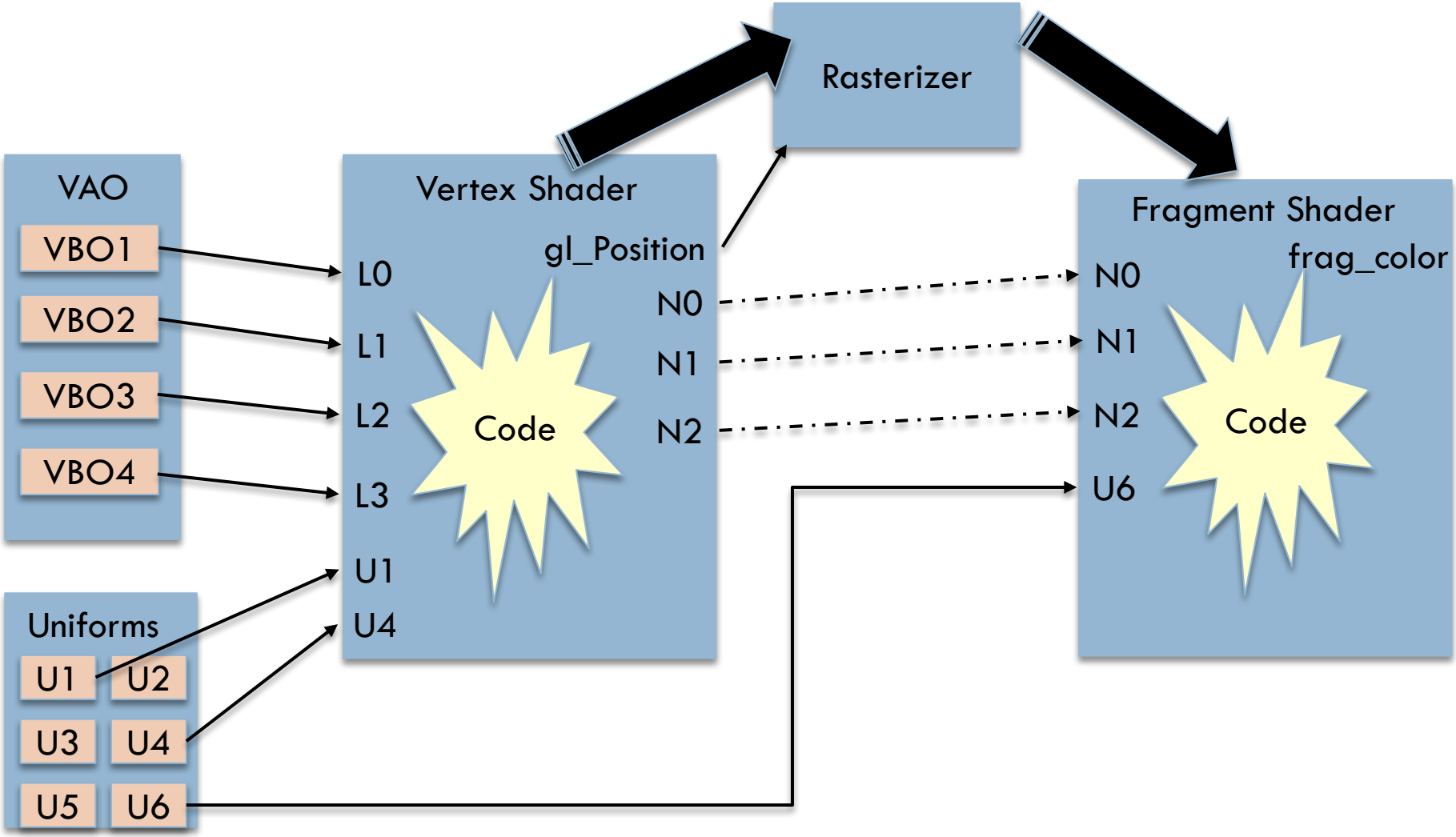
```
void main() {
  color = vertex_color;
  gl_Position = vec4(vertex_position, 1.0);
}
```

- gl_Position is a mandatory output of a vertex shader
  - And this did a bad job! – should have done matrix transform and did not
- Had to make a variable called color to send color info along to fragment shader

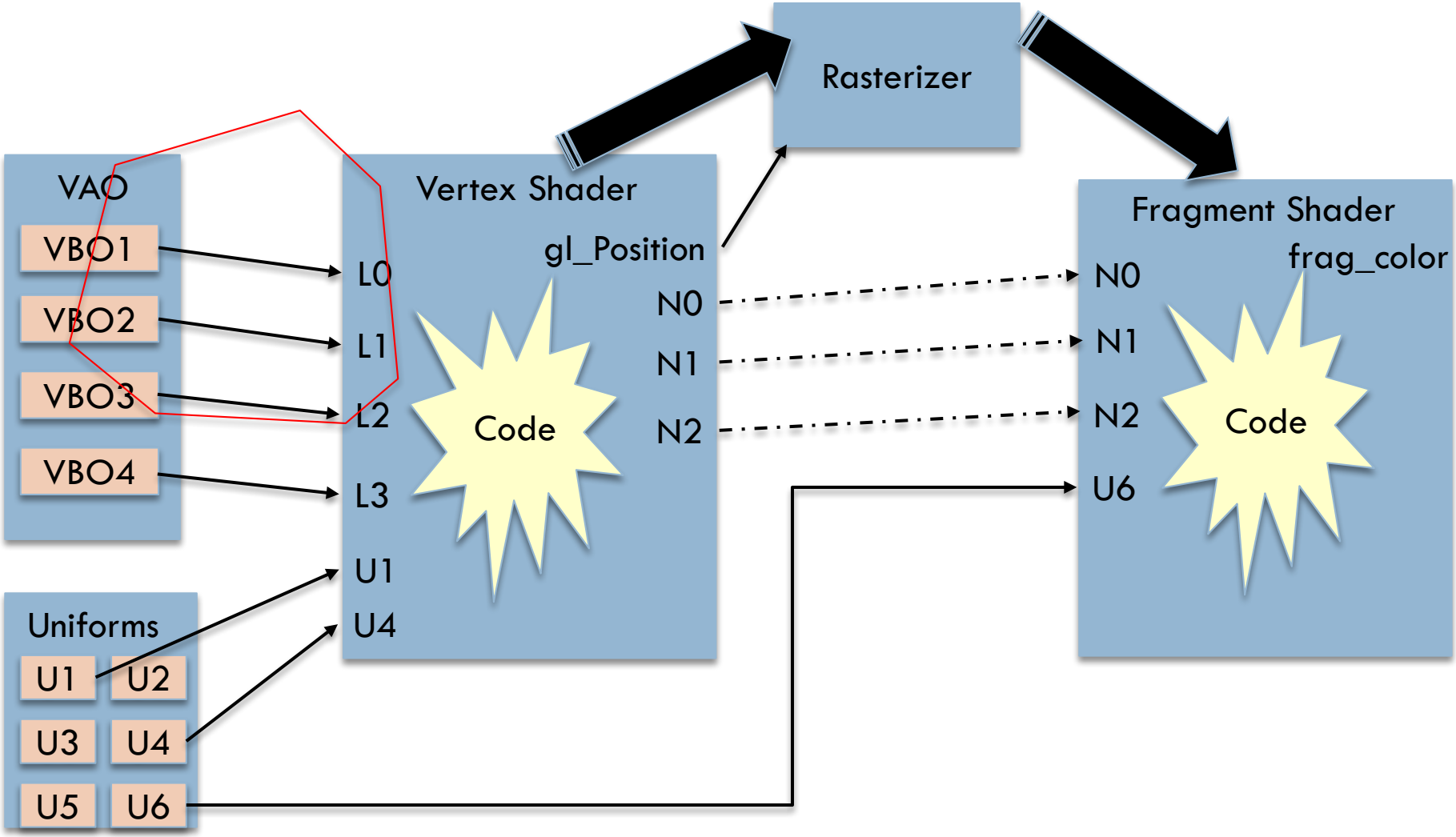
Vertex Shader From Starter Code

```
#version 400
layout (location = 0) in vec3 vertex_position;
layout (location = 1) in vec3 vertex_color;
out vec3 color;
void main() {
   color = vertex_color;
   gl_Position = vec4(vertex_position, 1.0);
}
```

Shader Overview



Shader Overview



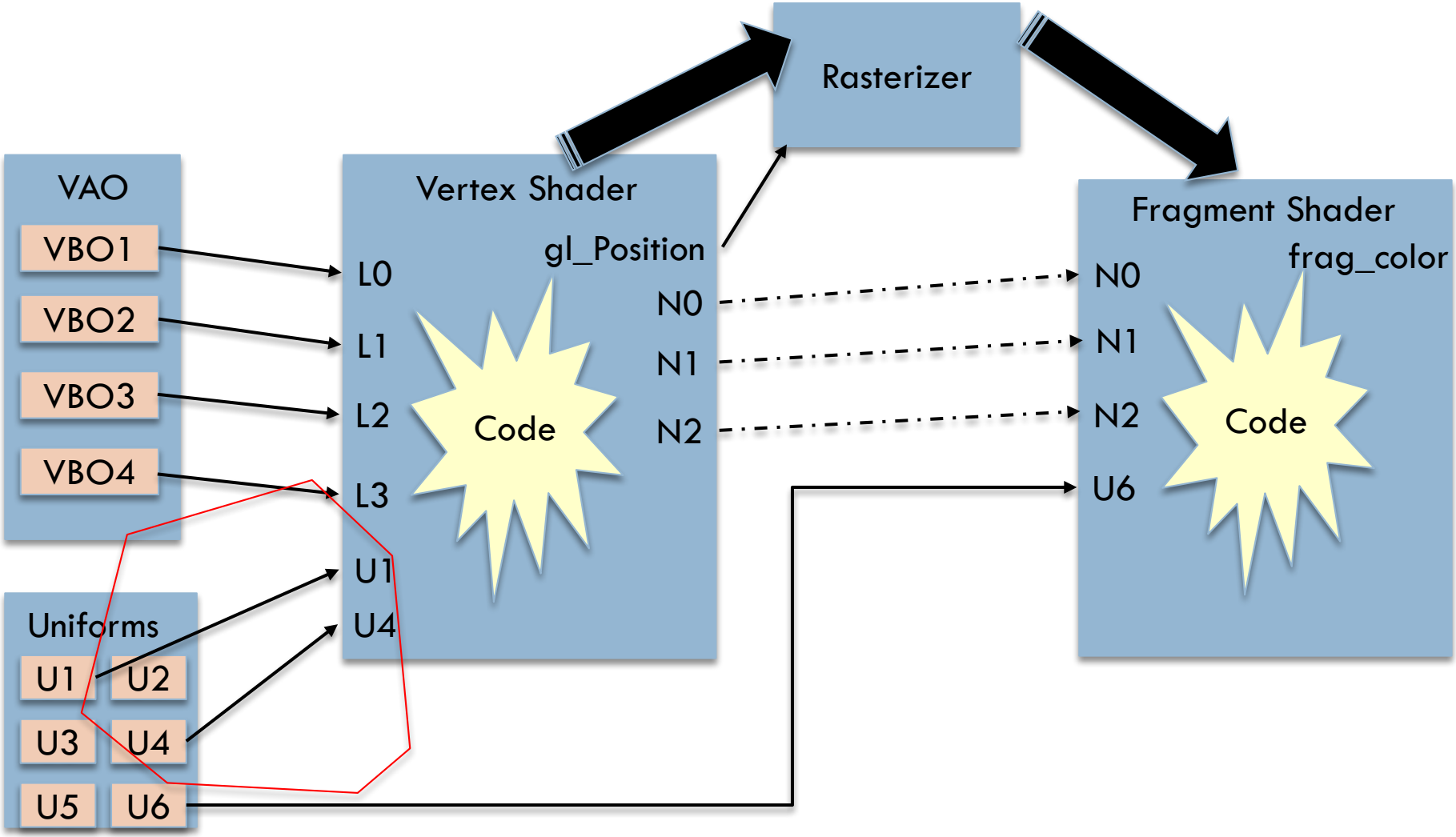
Vertex Shader From Starter Code

```
#version 400
layout (location = 0) in vec3 vertex_position;
layout (location = 1) in vec3 vertex_color;
out vec3 color;
void main() {
   color = vertex_color;
   gl_Position = vec4(vertex_position, 1.0);
}
```

These must match up. VAO is putting arrays in "locations."
Shader program must honor the VAO's ordering.
Location 0 contains "points_vbo" no matter what name it is given.

```
GLuint vao = 0;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);
glBindBuffer(GL_ARRAY_BUFFER, points_vbo);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
glBindBuffer(GL_ARRAY_BUFFER, colors_vbo);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, NULL);
glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, index_vbo );
```

Shader Overview



“Uniform” Means “Constant”

- You can set constants in your GL code
  - You set the name
  - You set the type
  - You set the value
- The shader program can then access those constants

Syntax for creating a uniform (in main GL code)

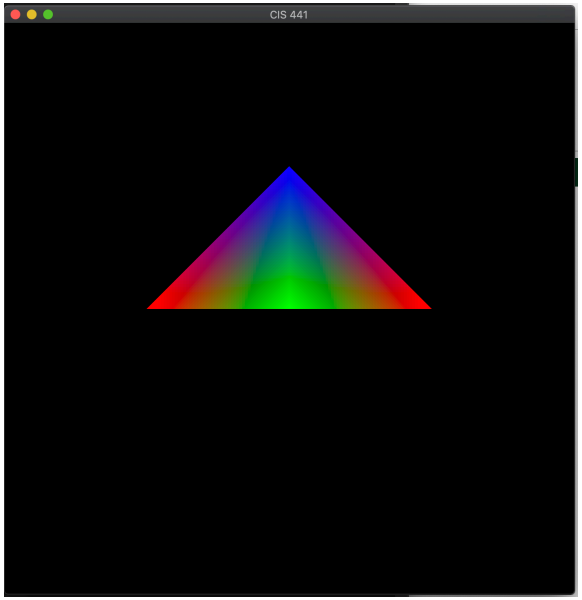
```
GLuint param = glGetUniformLocation(
                        shader_programme, "cis441"
                    );
glUniform1f(param, 0.5);
```

"Get Uniform Location" means "make a new uniform"

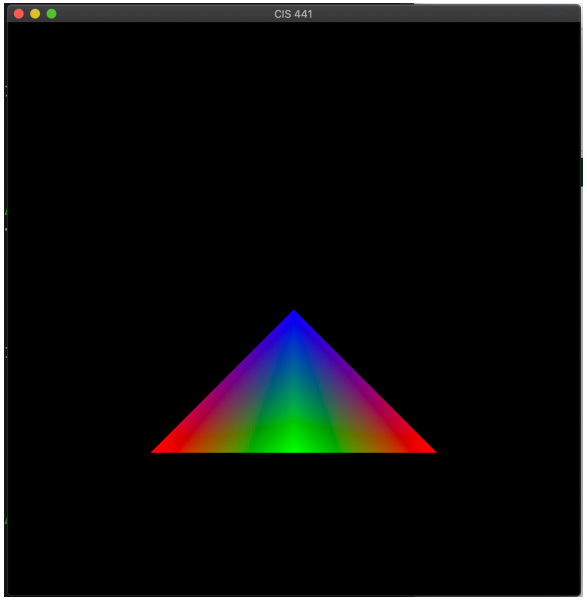
glUniform1f: the value of the constant will be a single float

Syntax for using a uniform

```
#version 400
layout (location = 0) in vec3 vertex_position;
layout (location = 1) in vec3 vertex_color;
uniform float cis441;
out vec3 color;
void main() {
   color = vertex_color;
   // gl_Position = vec4(vertex_position, 1.0);
   gl_Position = vec4(vertex_position.x,
vertex_position.y-cis441, vertex_position.z, 1.0);
}
```



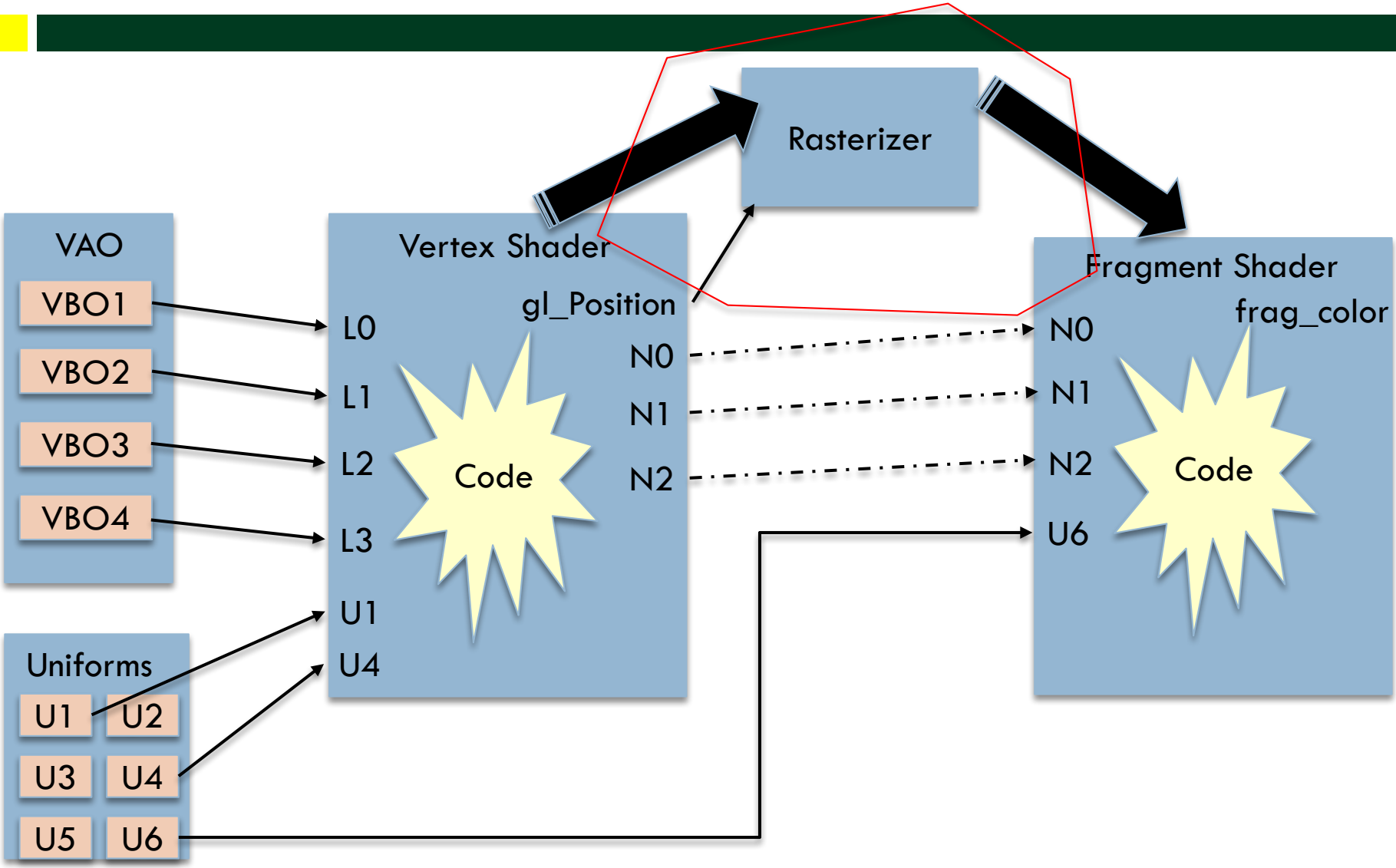
Original



New

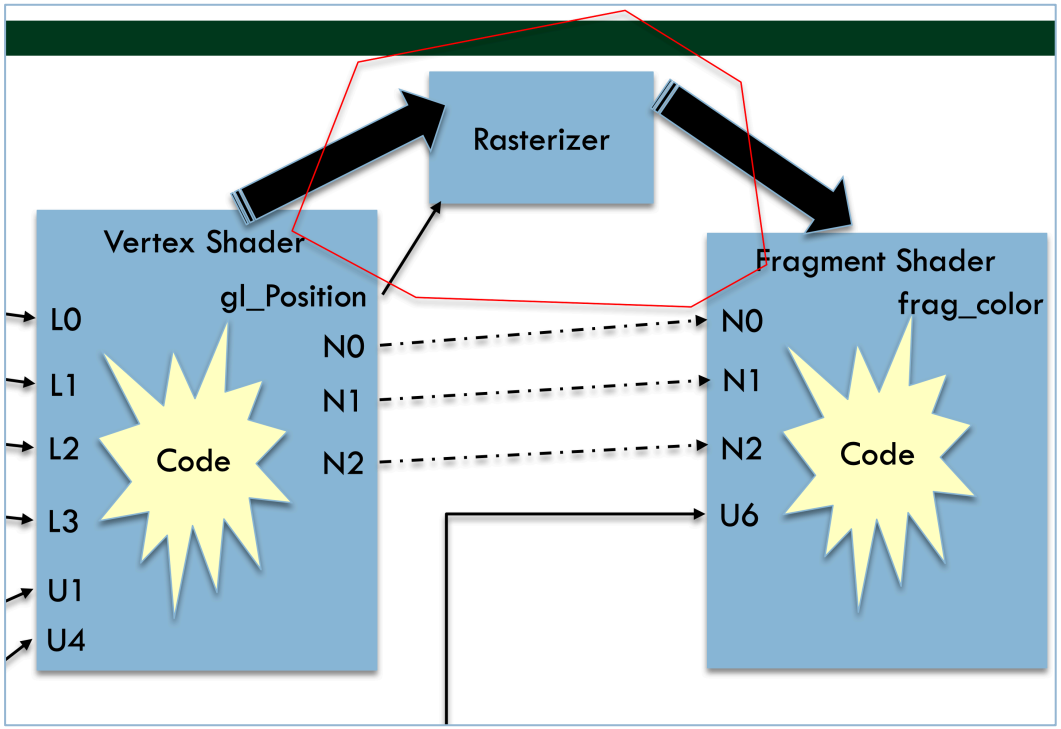
Note: this vertex shading is not typical.
Normal vertex shader: transform points from world space to image space
This vertex shader: assume they are already in image space

Shader Overview

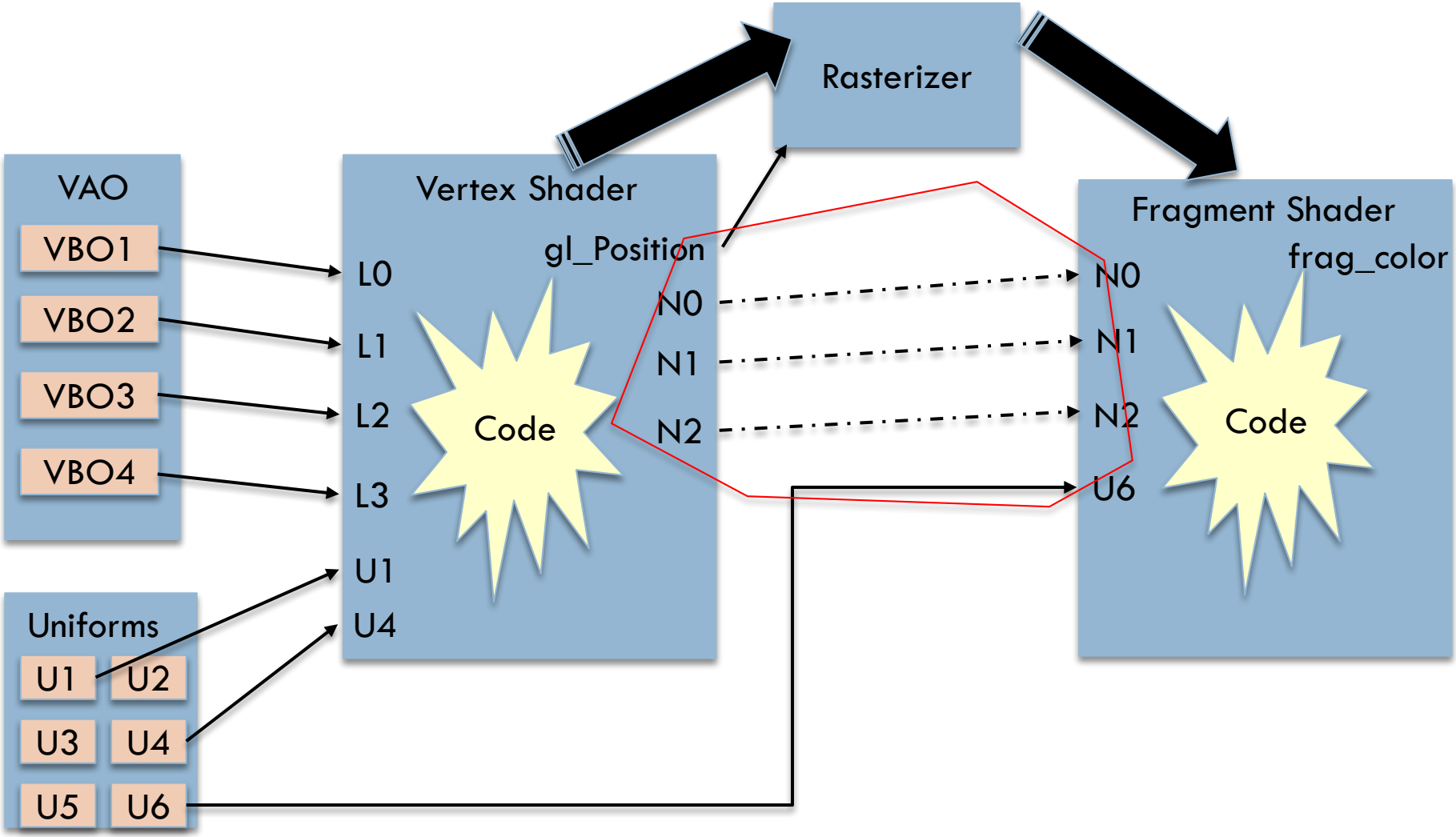


This picture is misleading

- Vertex shader called once per vertex
- Fragment shader called once per fragment
- One triangle has 3 vertices, but may have thousands of fragments
- Not one-to-one!



Shader Overview



One Shader's Output Is Another Shader's Input

- It is your job to arrange the output's of one shader to be the input's to the next
- Output of vertex shader is input to fragment shader
- If VAO sends in arrays that you want in the fragment shader, then the vertex shader needs to do work to pass them through (see next slide)

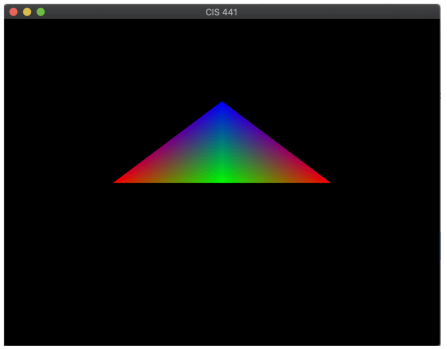
Vertex Shader → Fragment Shader

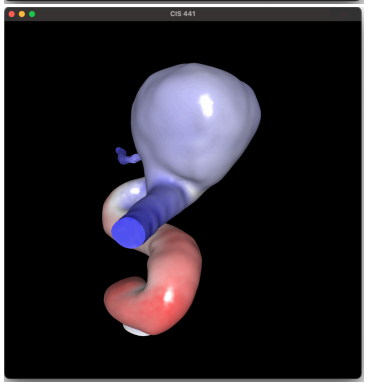
```
#version 400
layout (location = 0) in vec3 vertex_position;
layout (location = 1) in vec3 vertex_color;
out vec3 color;
void main() {
   color = vertex_color;
   gl_Position = vec4(vertex_position, 1.0);
}
```

```
#version 400
in vec3 color;
out vec4 frag_color;
void main() {
   frag_color = vec4(color, 1.0);
}
```

Project 2A

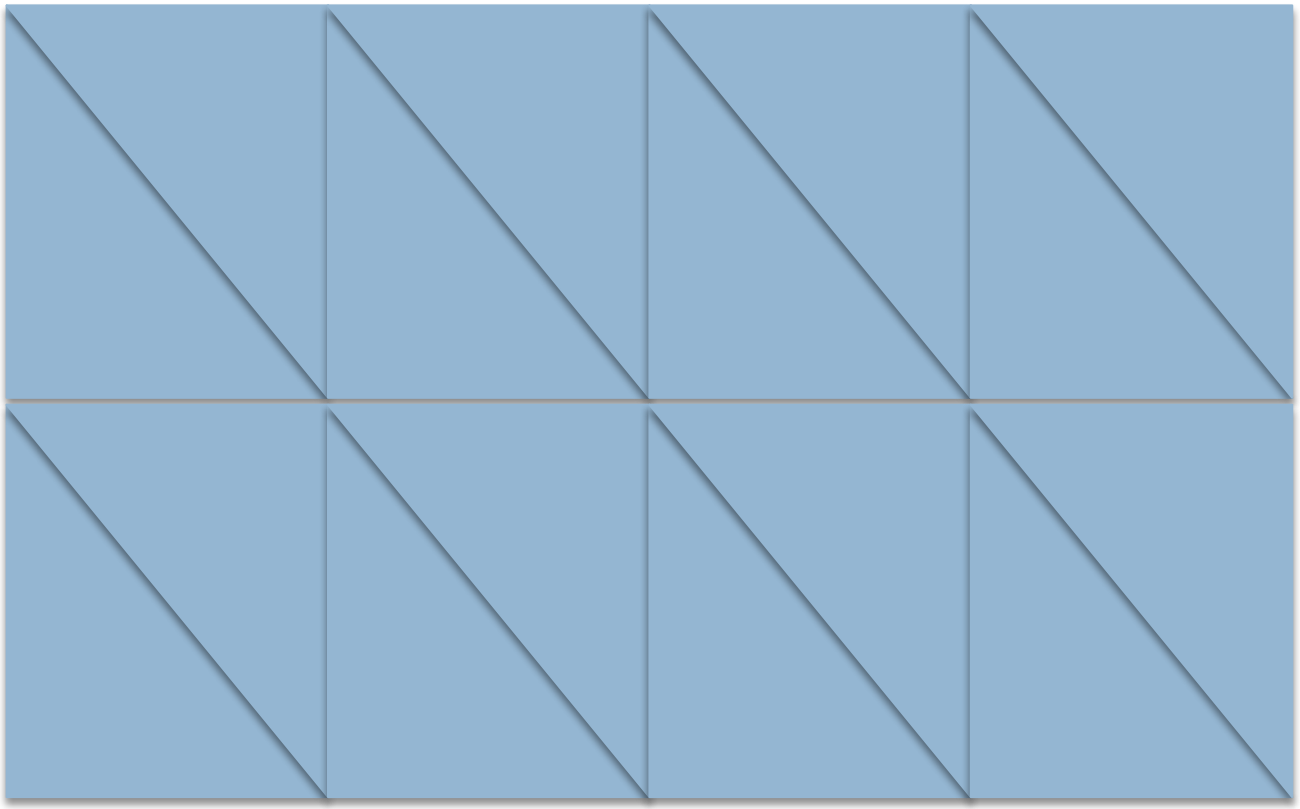
- Assigned today, due in one week (Tuesday May 11)
- Worth 8% of your grade
- Implementing Project 1 within OpenGL
- 5 phases
  - Phase 1: install GLFW
  - Phase 2: run example program
  - Phase 3: modify VBO/VAO
  - Phases 4 & 5: shader programs
- Please start ASAP on Phase 1-3
- Thursday's lecture will be on Phase 4 & 5





Rest of This Lecture

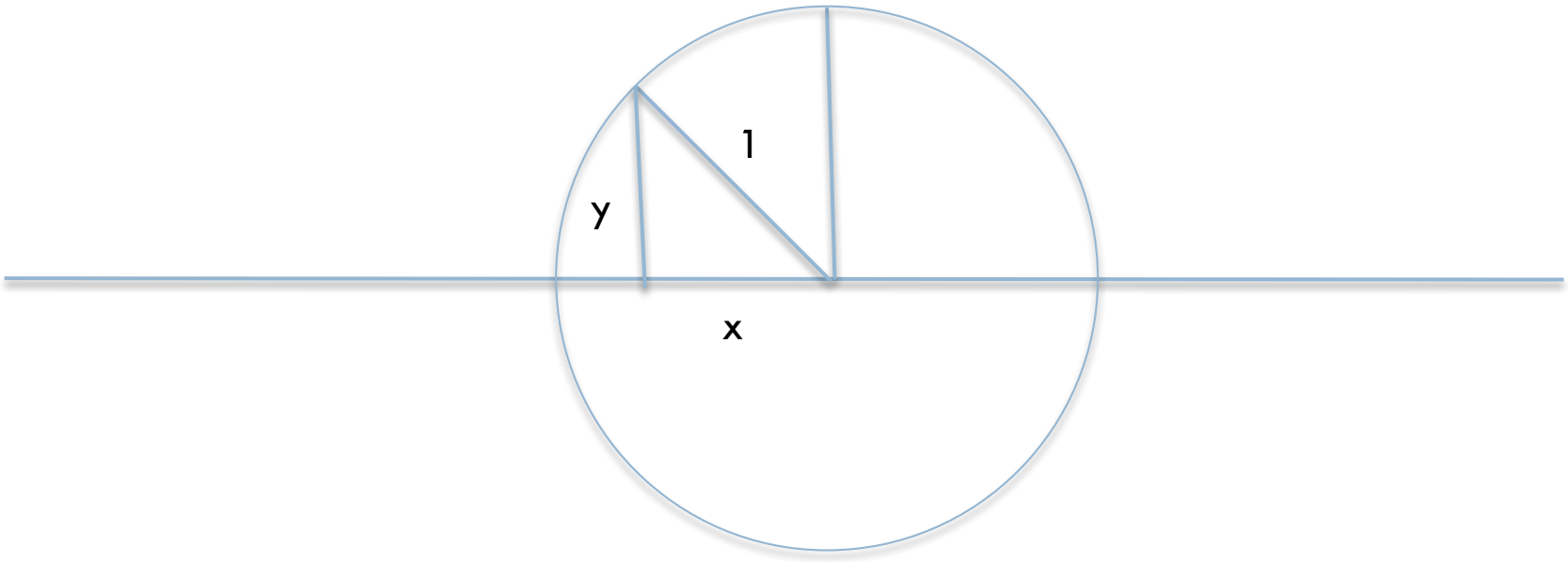
- Have fun with shaders
- Look at project 2A



RED

GREEN

BLUE


1
y
x