09 - The OpenGL Graphics Pipeline– Part 2

Acknowledgements: Daniele Panozzo

https://open.gl

- This slide set is based on the excellent tutorial available at https://open.gl
- Many thanks to:
  - Toby Rufinus
  - Eric Engeström
  - Elliott Sales de Andrade
  - Aaron Hamilton

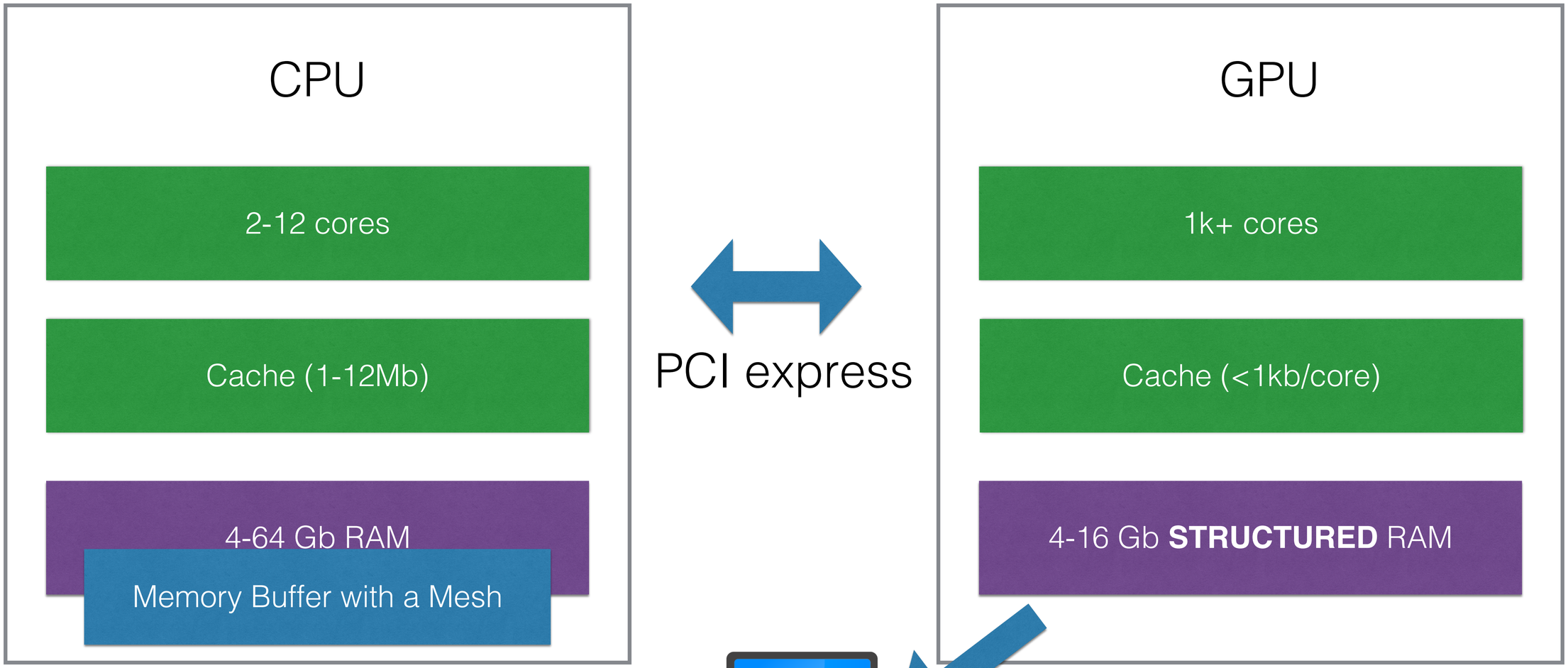
Skeleton

```
#include <libraryheaders>

int main()
{
    createWindow(title, width, height);
    createOpenGLContext(settings);

    while (windowOpen)
    {
        while (event = newEvent())
            handleEvent(event);

        updateScene();

        drawGraphics();
        presentGraphics();
    }

    return 0;
}
```

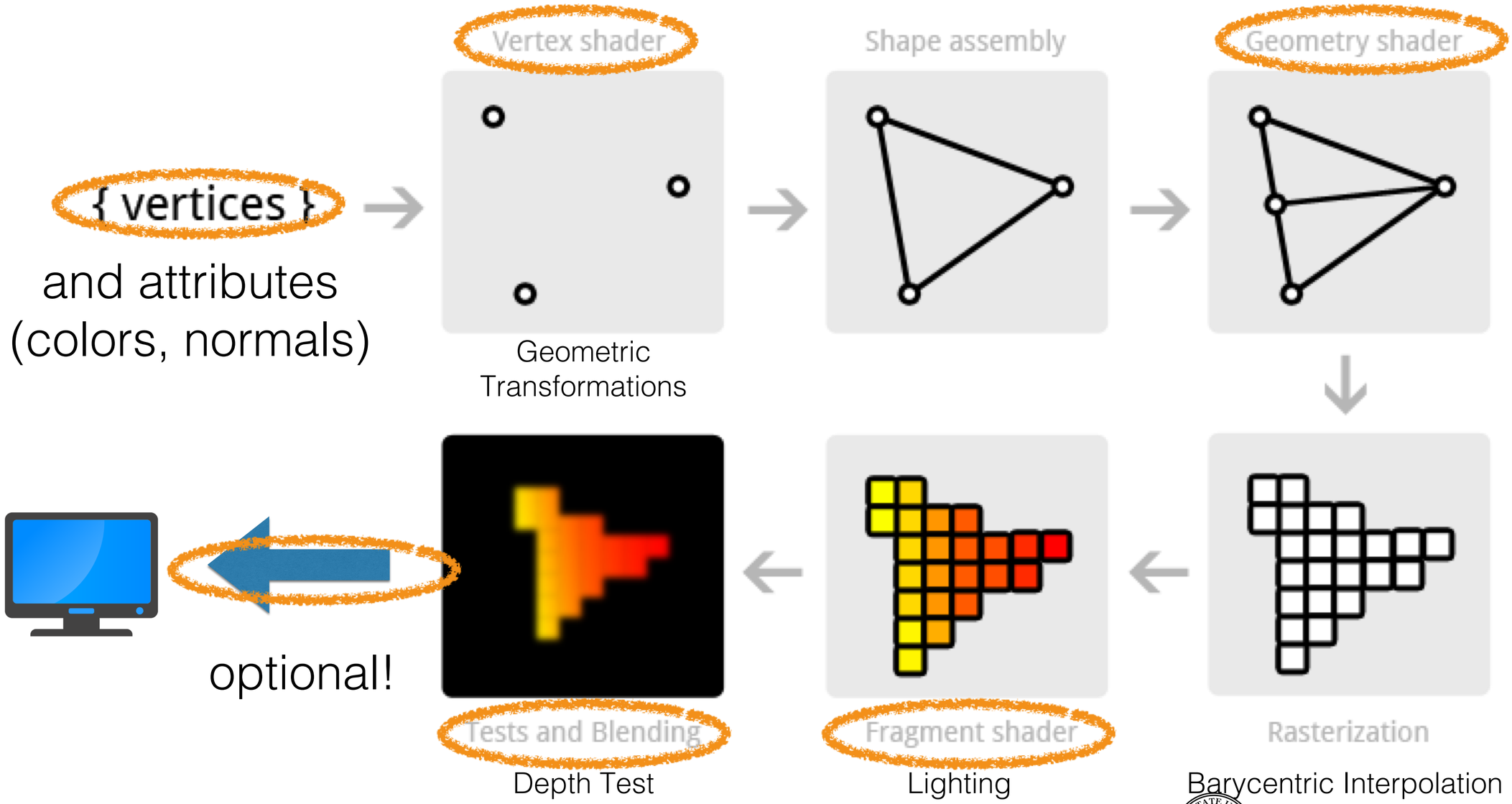
CPU and GPU memory

You write code here …

… which acts here.



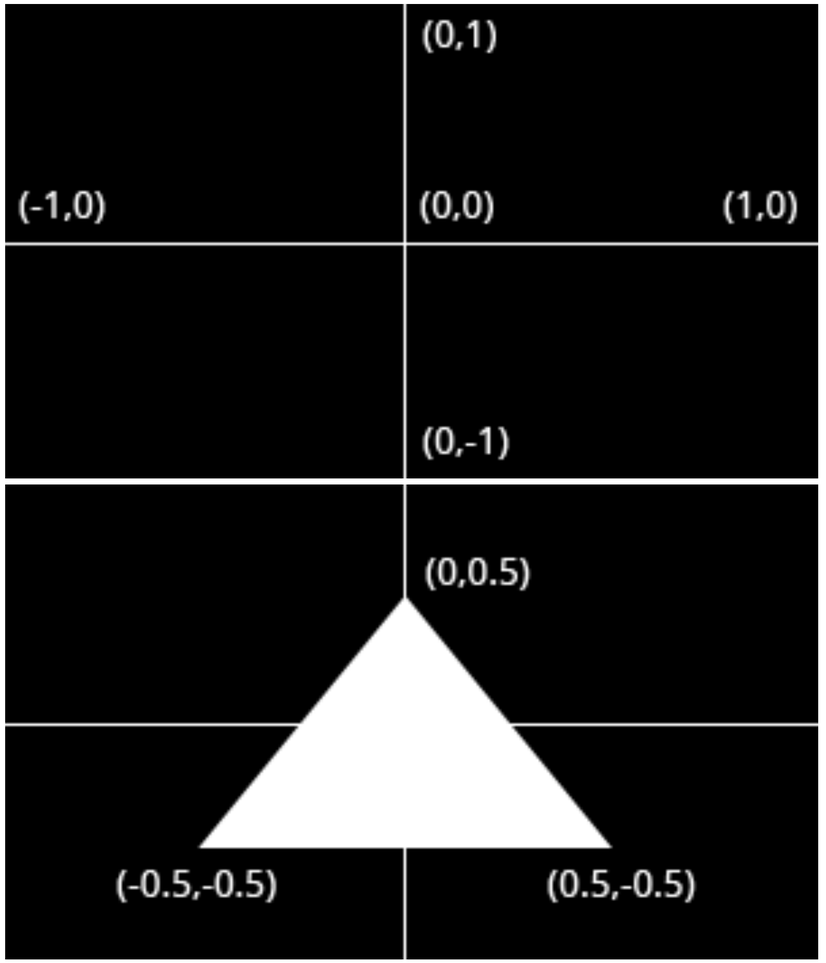
OpenGL pipeline



Device Coordinates

- Only the vertices in the canonical cube will be shown. The cube will be stretched to fill all available screen space.
- All vertices must be passed in one instruction, in a single contiguous block of memory
- This matrix resides in CPU memory!

```
float vertices[] = {
     0.0f,  0.5f, // Vertex 1 (X, Y)
     0.5f, -0.5f, // Vertex 2 (X, Y)
    -0.5f, -0.5f  // Vertex 3 (X, Y)
};
```



Vertex Buffer Object

- Strange name for something simple, a chunk of memory in the GPU space

```
GLuint vbo;
glGenBuffers(1, &vbo); // Generate 1 buffer
```

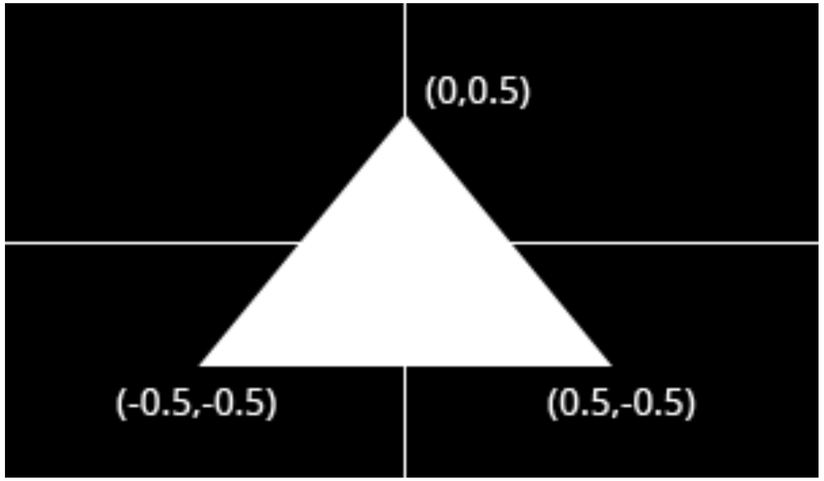
- vbo contains a “opengl reference” to the buffer, which is simply an integer

```
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```

- Now the content of vertices is uploaded to the GPU memory

A simple vertex shader

```
float vertices[] = {
     0.0f,  0.5f, // Vertex 1 (X, Y)
     0.5f, -0.5f, // Vertex 2 (X, Y)
    -0.5f, -0.5f  // Vertex 3 (X, Y)
};
```

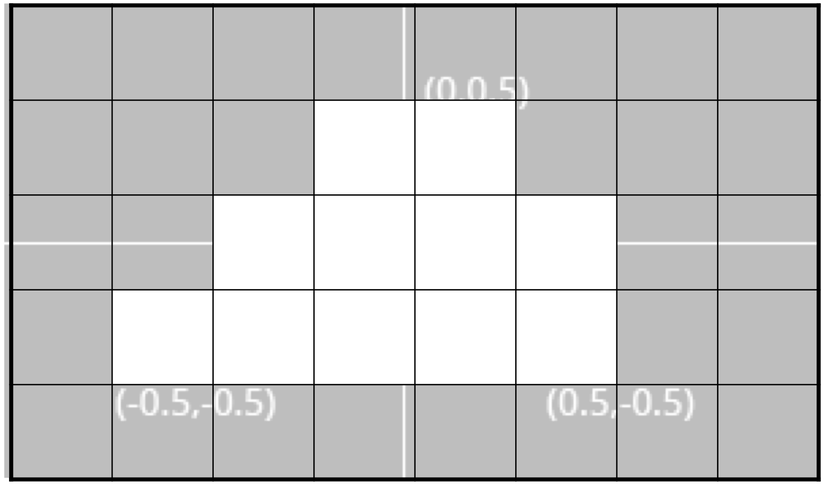



```
#version 150

in vec2 position;

void main()
{
    gl_Position = vec4(position.x, position.y, 0.0, 1.0);
}
```

gl_Position is a special keyword

A simple fragment shader

```
float vertices[] = {
     0.0f,  0.5f, // Vertex 1 (X, Y)
     0.5f, -0.5f, // Vertex 2 (X, Y)
    -0.5f, -0.5f  // Vertex 3 (X, Y)
};
```




```
#version 150

out vec4 outColor;

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

The colors inside a shader are between 0.0 and 1.0

Compiling Shaders

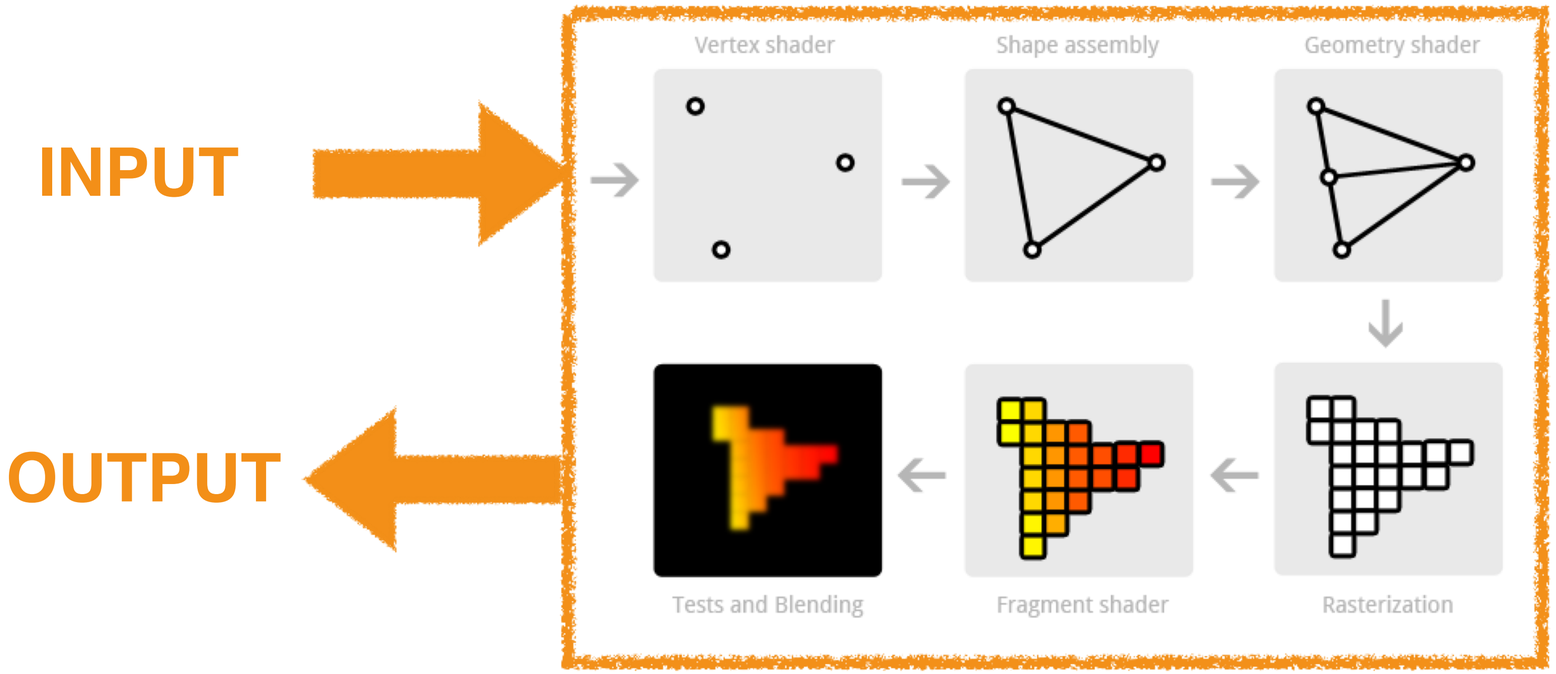
```
const GLchar* vertex_shader =
            "#version 150 core\n"
                    "in vec2 position;"
                    "void main()"
                    “{gl_Position = vec4(position, 0.0, 1.0);}";
const GLchar* fragment_shader =
            "#version 150 core\n"
                    "out vec4 outColor;"
                    "uniform vec3 triangleColor;"
                    "void main()"
                    “{ outColor = vec4(triangleColor, 1.0); }”;

GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, &vertex_shader, NULL);
glCompileShader(vertexShader);

GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fragment_shader, NULL);
glCompileShader(fragmentShader);
```


Almost there, OpenGL program

- The two shaders needs to be combined into a program

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

We need to connect the program with our input data and map the output to a memory buffer or to the screen



Almost there, OpenGL program

- The two shaders needs to be combined into a program

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

- The output of the fragment shader connected with the default frame buffer (“0”)

```
glBindFragDataLocation(shaderProgram, 0, “outColor");
```

- The program is now ready to be linked and activated

```
glLinkProgram(shaderProgram);
glUseProgram(shaderProgram);
```


Connecting VBOs to the program

```
// This is our set of vertices
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

// This is our vertex shader
#version 150
in vec2 position;
void main()
{
    gl_Position = vec4(position.x, position.y, 0.0, 1.0);
}
```

- We want to connect vbo to the “position” slot, so that the program will know where to find the vertex positions

Connecting VBOs to the program

- Bind the program

```
glUseProgram(shaderProgram);
```

- Query the program to find the address of “position”

```
GLint posAttrib = glGetAttribLocation(shaderProgram, “position");
```

- Bind the vbo we want to connect

```
glBindBuffer(GL_ARRAY_BUFFER, vbo);
```

- Connect the VBO to the “position” slot

```
glVertexAttribPointer(posAttrib, 2, GL_FLOAT, GL_FALSE, 0, 0);
```

- Activate the attribute

```
glEnableVertexAttribArray(posAttrib);
```

glVertexAttributePointer

```
glVertexAttribPointer(posAttrib, 2, GL_FLOAT, GL_FALSE, 0, 0);
```

- posAttrib is the index of the attribute
- 2 indicates that the VBOs has two attributes per vertex (x and y)
- GL_FLOAT indicates that the VBO stores single precision floating point numbers
- 0 is the ***stride***, or how many bytes are between each position attribute in the array
- the last 0 is the ***offset***, or how many bytes from the start of the array the attribute occurs
- **There is no type checking, if you do a mistake here you will render whatever is in the memory address that you specified and possibly crash the graphic driver**

Vertex Array Objects - VAO

- Changing OpenGL program is easy, but it would be annoying to have to redo all the connections between VBOs and input every time
- A VAO stores these connections and allow to easily switch between different configurations

```
GLuint vao;
glGenVertexArrays(1, &vao);

glBindVertexArray(vao);
```

- After the bind, all the subsequent calls to glVertexAttribPointer will be stored in the VAO
- Tip: You must have a binded VAO at all times. If you do not have it, glVertexAttribPointer will ignore the bind without giving you an error :-)

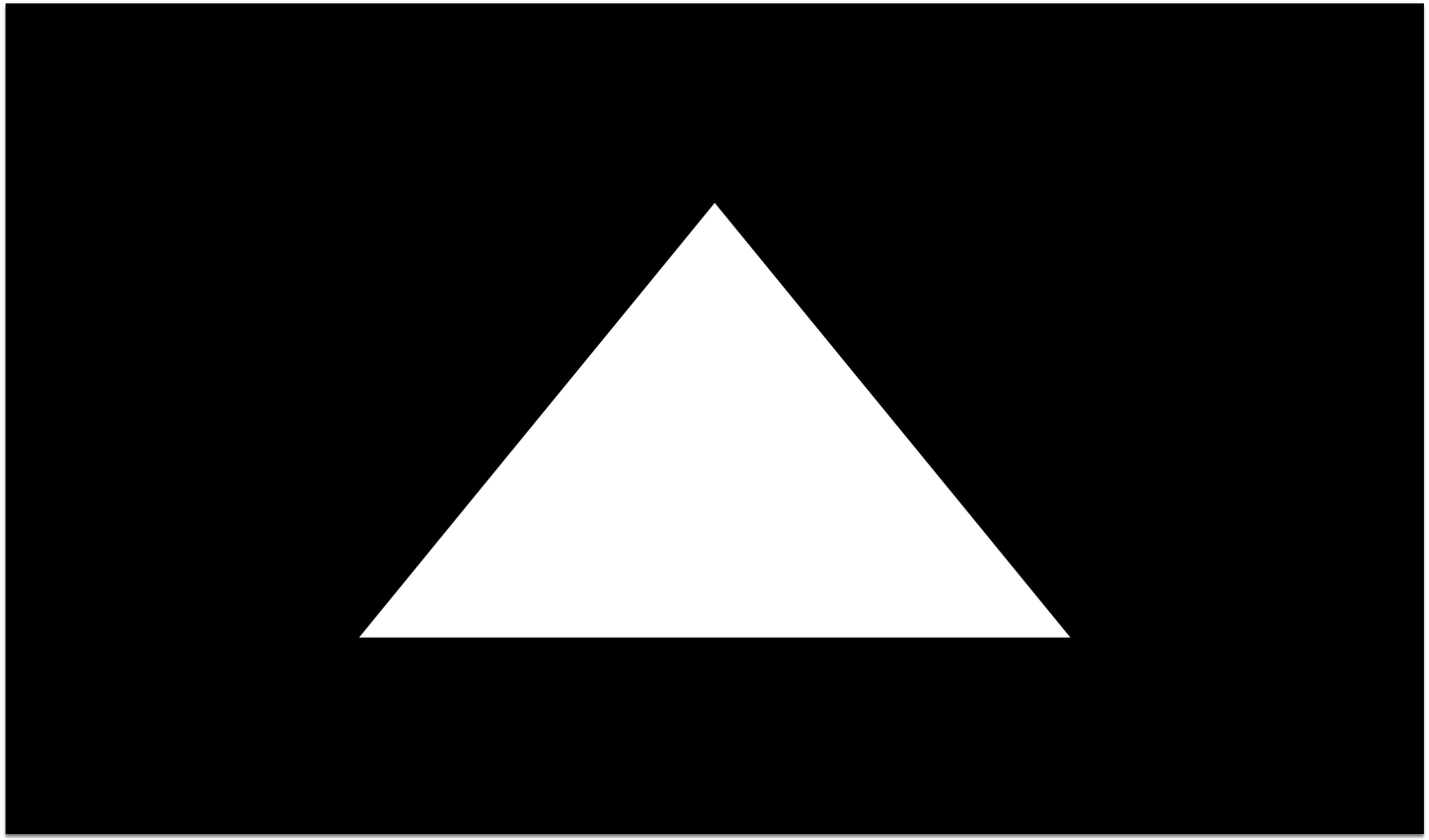
We are all set!

- Now that we prepared the program, linked the input array and set the output to the frame buffer we are ready to draw our triangle

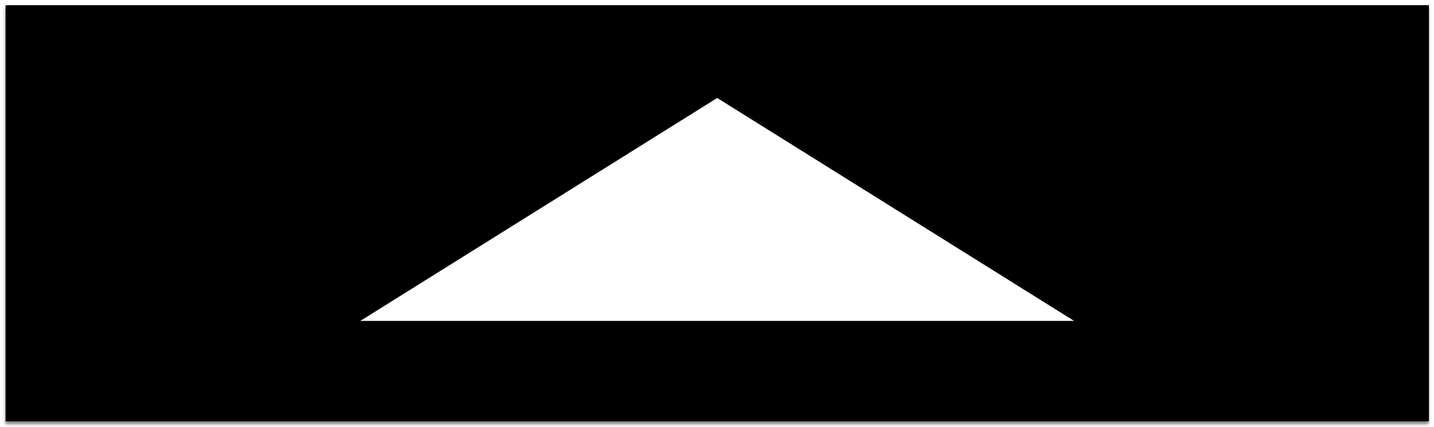
```
glDrawArrays(GL_TRIANGLES, 0, 3);
```

- The first parameter is the type of primitive (in this case, it draws a triangle after each set of three vertices is processed)
- 0 is the ***offset,*** i.e. how many vertices it has to skip from the beginning of the VBOs
- 3 is the number of **VERTICES** (not primitives) that it should process

If everything was done correctly…



If everything was done correctly…



Our current pipeline does not consider view transformations!

(Recommended Exercise: how would you change the current shaders to take the size of the window into account and always draw a equilateral triangle?)

Uniforms

- Uniform are values that are constant for the entire **scene**, i.e. they are not attached to vertices
- They correspond to global variables in C/C++
- All vertices and all fragments will see the same value for the same uniform
- For example, let's use a uniform to store the triangle color (which is now hardcoded in our shader)

Uniforms

Old Fragment Shader

```
#version 150

out vec4 outColor;

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

New Fragment Shader

```
#version 150

uniform vec3 triangleColor;

out vec4 outColor;

void main()
{
    outColor = vec4(triangleColor, 1.0);
}
```

```
GLint uniColor = glGetUniformLocation(shaderProgram, “triangleColor”);
glUniform3f(uniColor, 1.0f, 0.0f, 0.0f);
```

Recap

- Compile, link and enable an OpenGL program (vertex + fragment shader)
- Connects the output of the fragment shader to the frame buffer
- Connect the VBOs to the input slots of the vertex shader using a VAO
- Assign the uniform parameters (if you use them in the shaders)
- Clear the framebuffer
- Draw the primitives
- Swap the framebuffer to make the newly rendered frame visible

Complete code Example

References

https://open.gl — Main reference

Fundamentals of Computer Graphics, Fourth Edition
4th Edition **by Steve Marschner, Peter Shirley**

Chapter 17