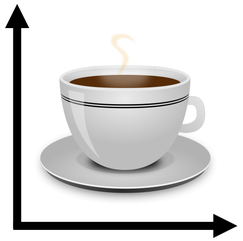
08 - The OpenGL Graphics Pipeline – Part 1

Acknowledgements: Daniele Panozzo

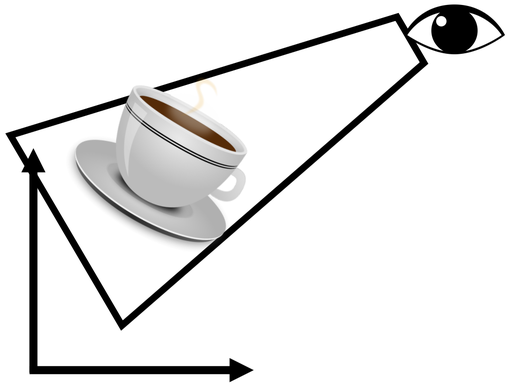
Viewing Transformation

object space



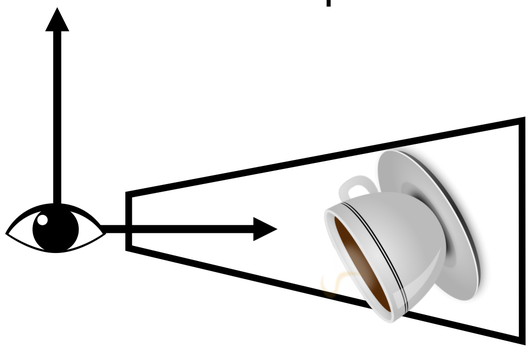
model

world space



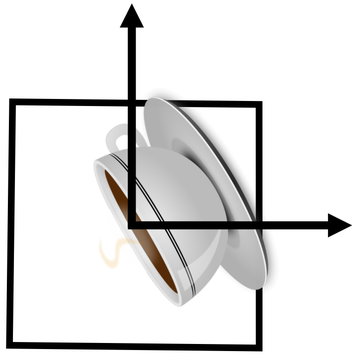
camera

camera space



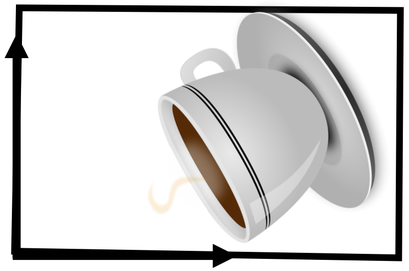
projection

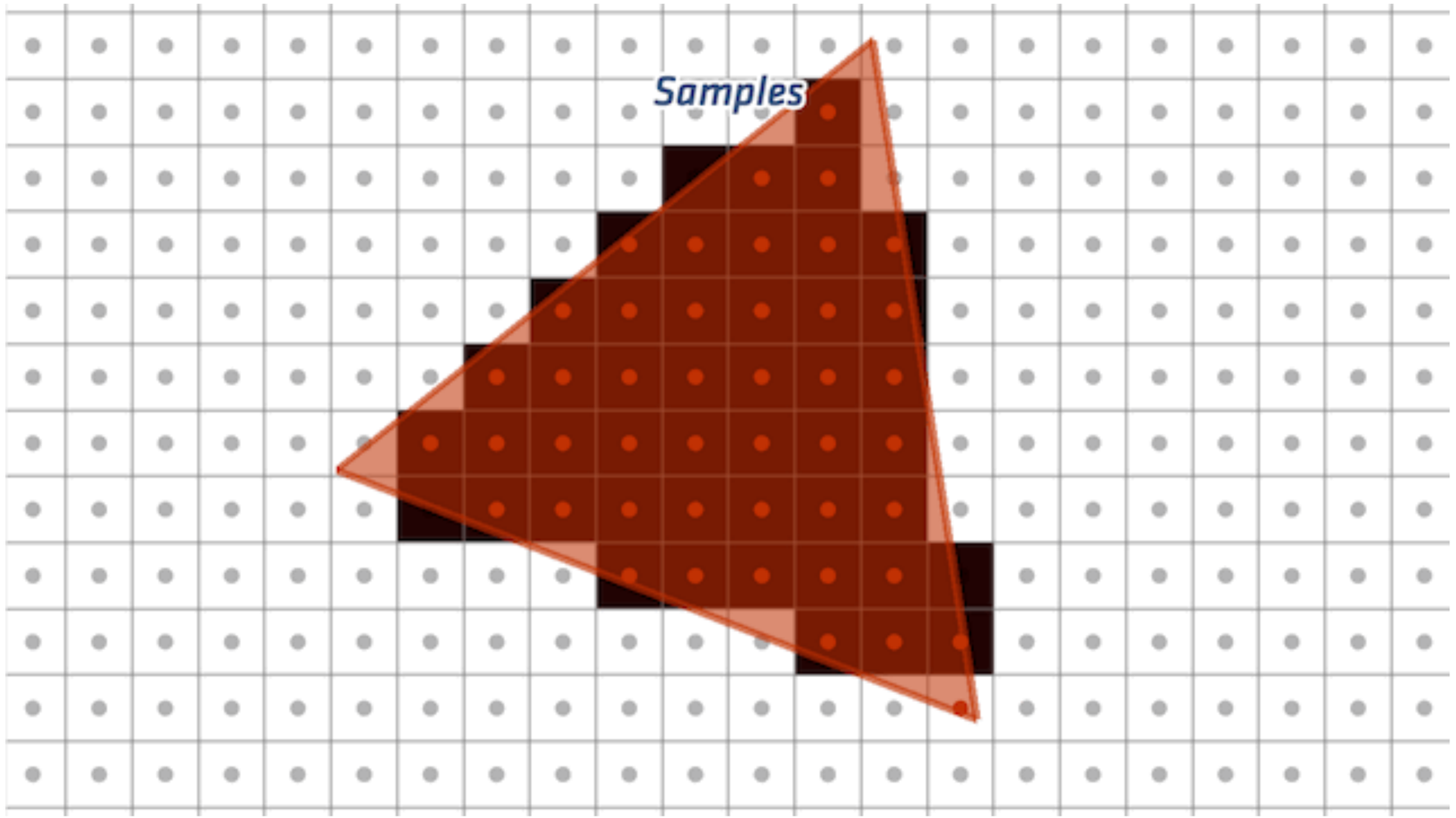
canonical
view volume



viewport

screen space




Samples

How to do it?

- Specialized hardware — Graphics Processing Unit (GPU)
- APIs to interact with the hardware
  - OpenGL
  - DirectX
  - Metal/Mantle/Vulkan

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

Context Creation

- Before you can draw anything you need to:
  - Open a window (i.e. ask the OS to give you access to a portion of the screen)
  - Initialize the OpenGL API and assign the available screen space to it
- This is a technical step and it is heavily dependent on the operating system and on the hardware

Window Manager

- There are many libraries that take care of this for you, hiding all the complexity and providing a cross-platform and cross-hardware interface
- We are going to use GLFW (http://www.glfw.org)
- A window manager usually provides an event management system

Skeleton

```
#include <libraryheaders>

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

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

        updateScene();

        drawGraphics();
        presentGraphics();
    }

    return 0;
}
```

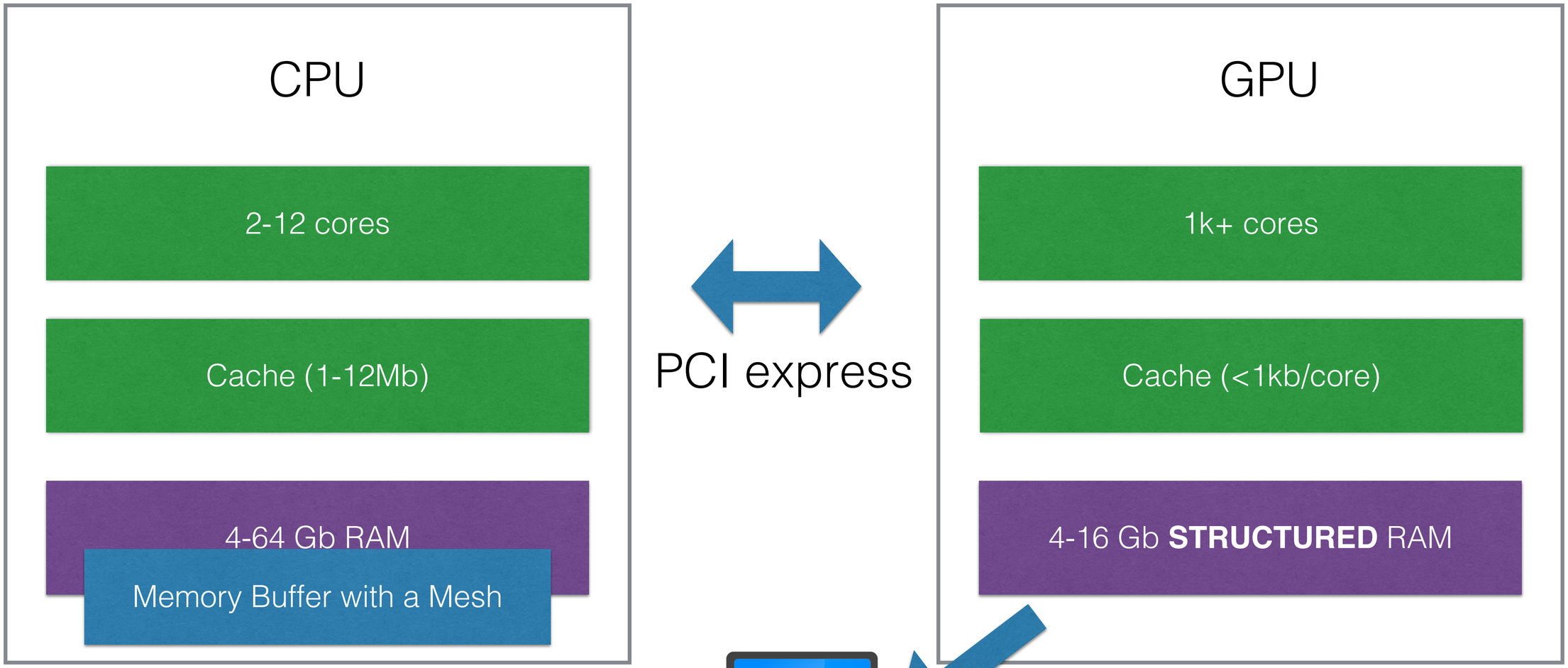
Event Processing

- Callback mechanisms
  - For every event (keypressed, mouse motion) you need to write a function that handles it
  - The functions are registered in glfw, that will call them whenever the corresponding event happens
  - We will see an example later on

CPU and GPU memory

You write code here …

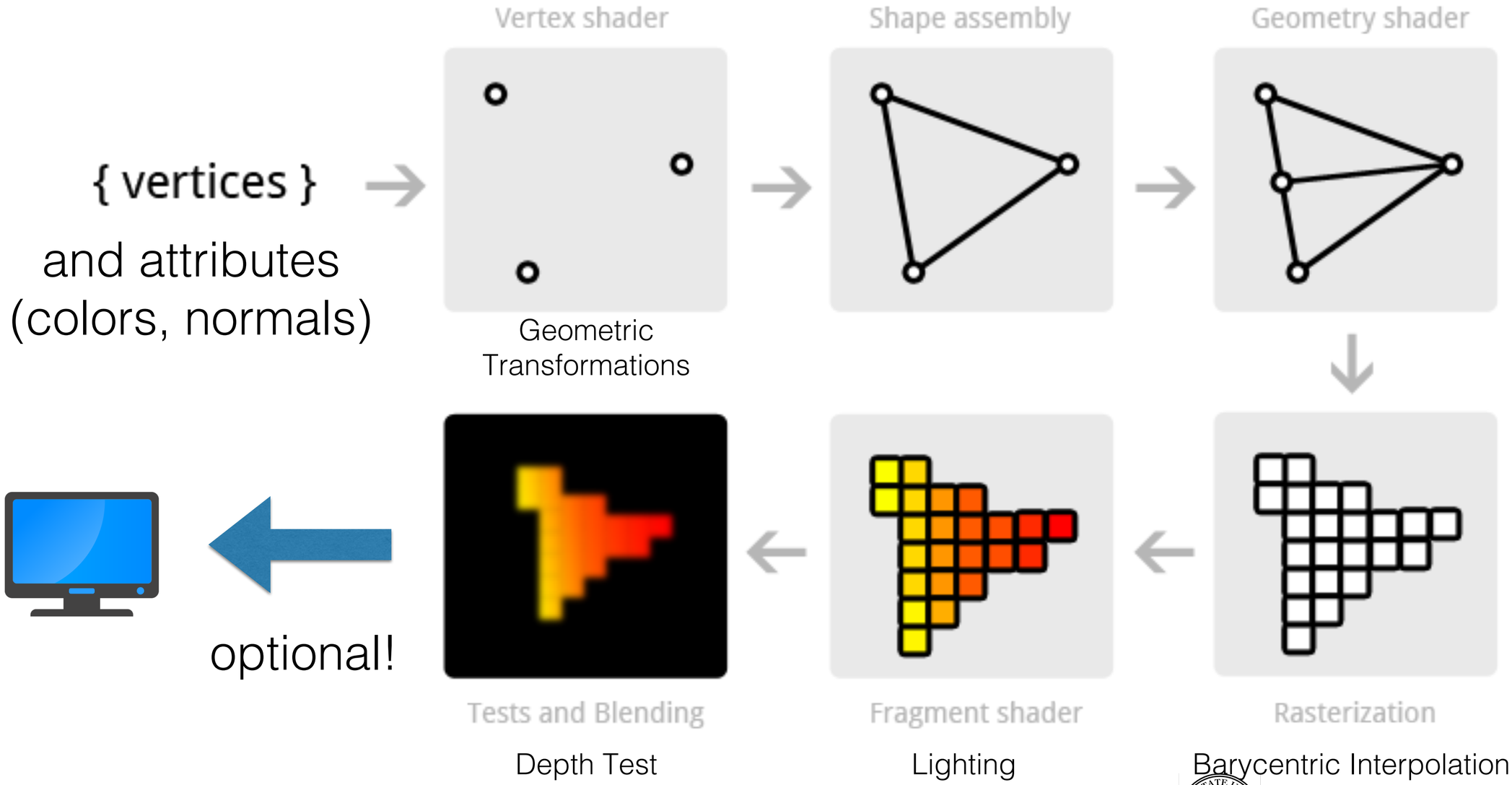
… which acts here.



“Modern” OpenGL

- OpenGL 1.0 was released in 1992 - historically very rigid and with a high level API that was dealing with all the pipeline steps
- “Modern” OpenGL usually refers to the lightweight OpenGL 3 (2008)
- Barebone, but adaptable: since each API call has a fixed and high overhead, the library was design to minimize them
- Much easier to use, but you need to understand how it works completely!

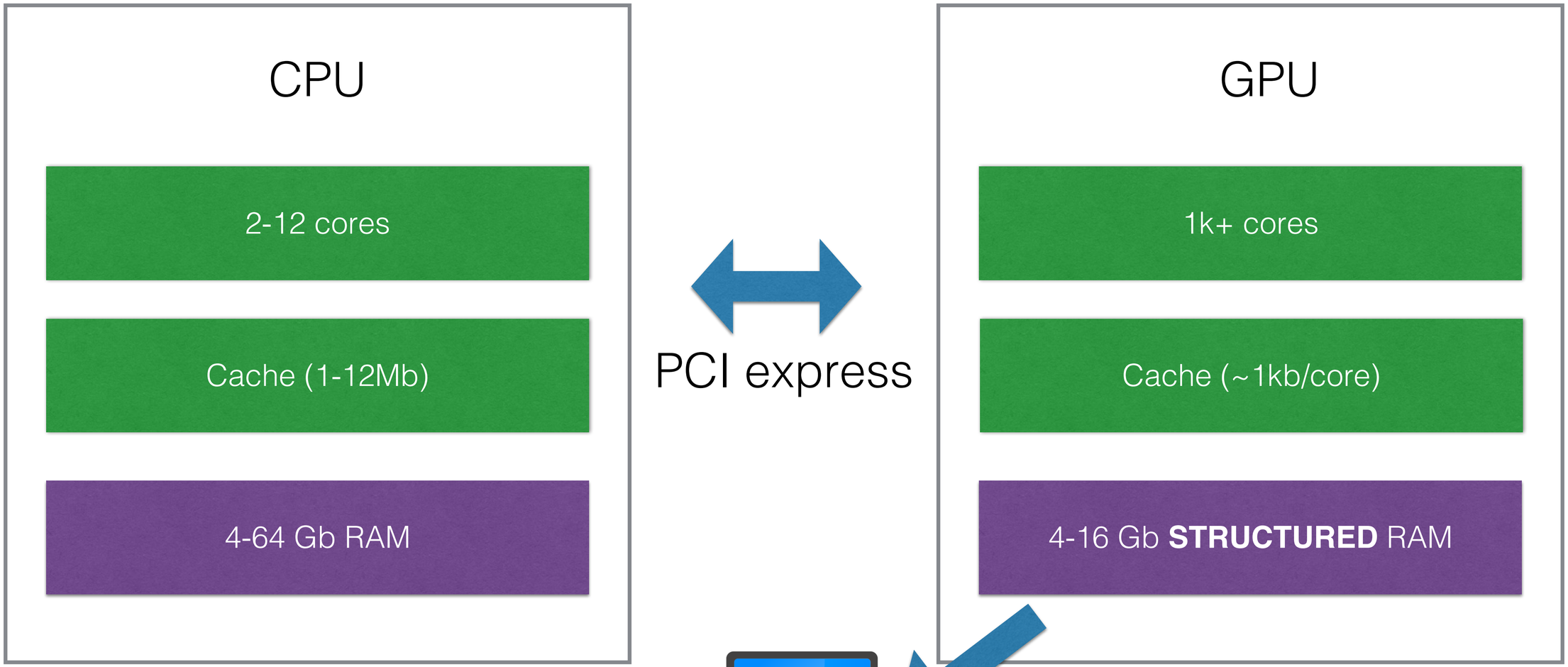
OpenGL pipeline



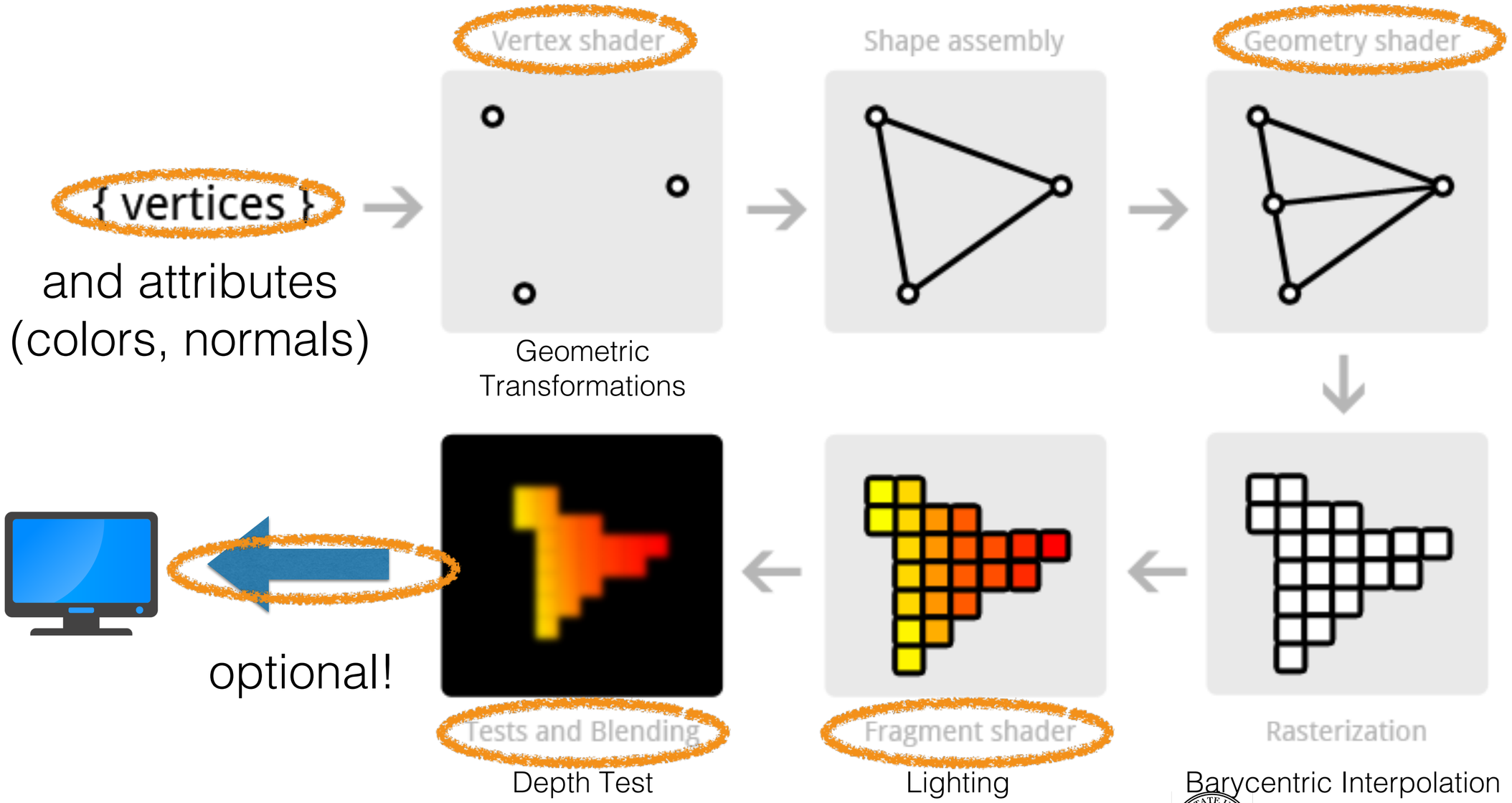
CPU and GPU memory

You write code here …

… which acts here.



OpenGL pipeline



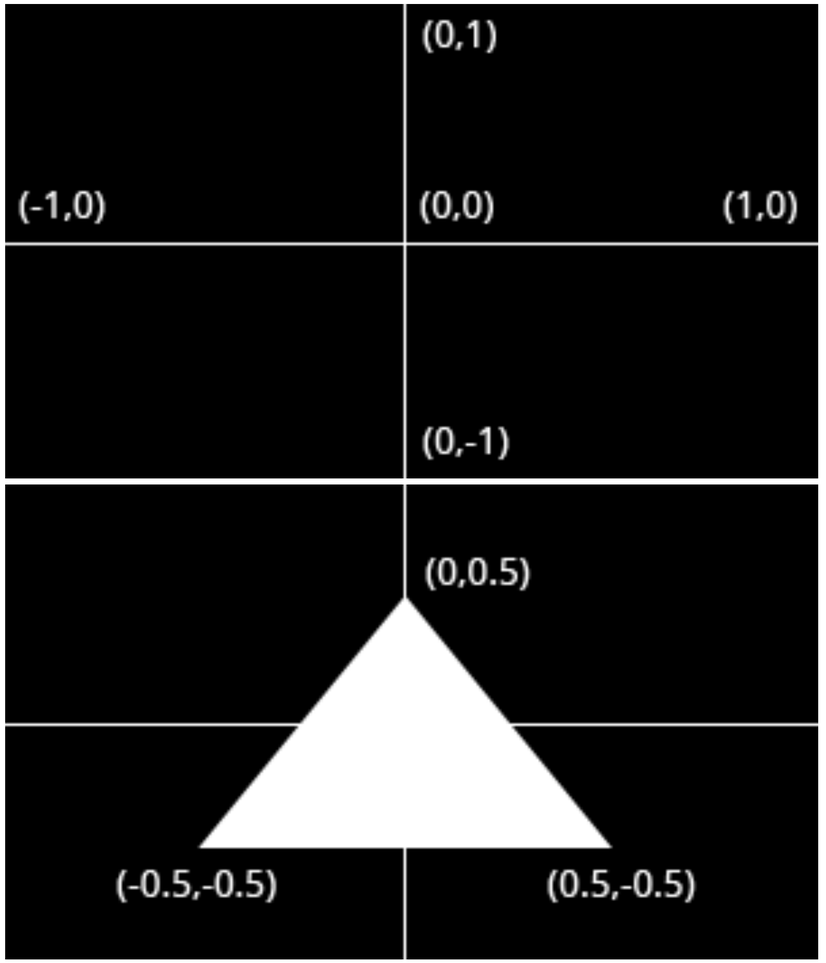
Vertex Input

- You have to send to the GPU a set of vertex attributes:
  - World Coordinates
  - Color
  - Normal
- You can pass as many as you want, but remember that bandwidth is precious, you want to send only what is needed

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)
};
```



GPU Memory

- It is faster and closer to the GPU cores
- You want to minimize the number of transfers, the more you can reuse between two frames the better
- The throughput is very high but unfortunately also the latency. Try to send as much data as possible with one single instruction
- You need to manually manage the GPU memory and you can have memory leaks on the GPU too, be careful

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

Buffer Types

- You can provide hints to the API depending on the type of data that you are uploading:

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

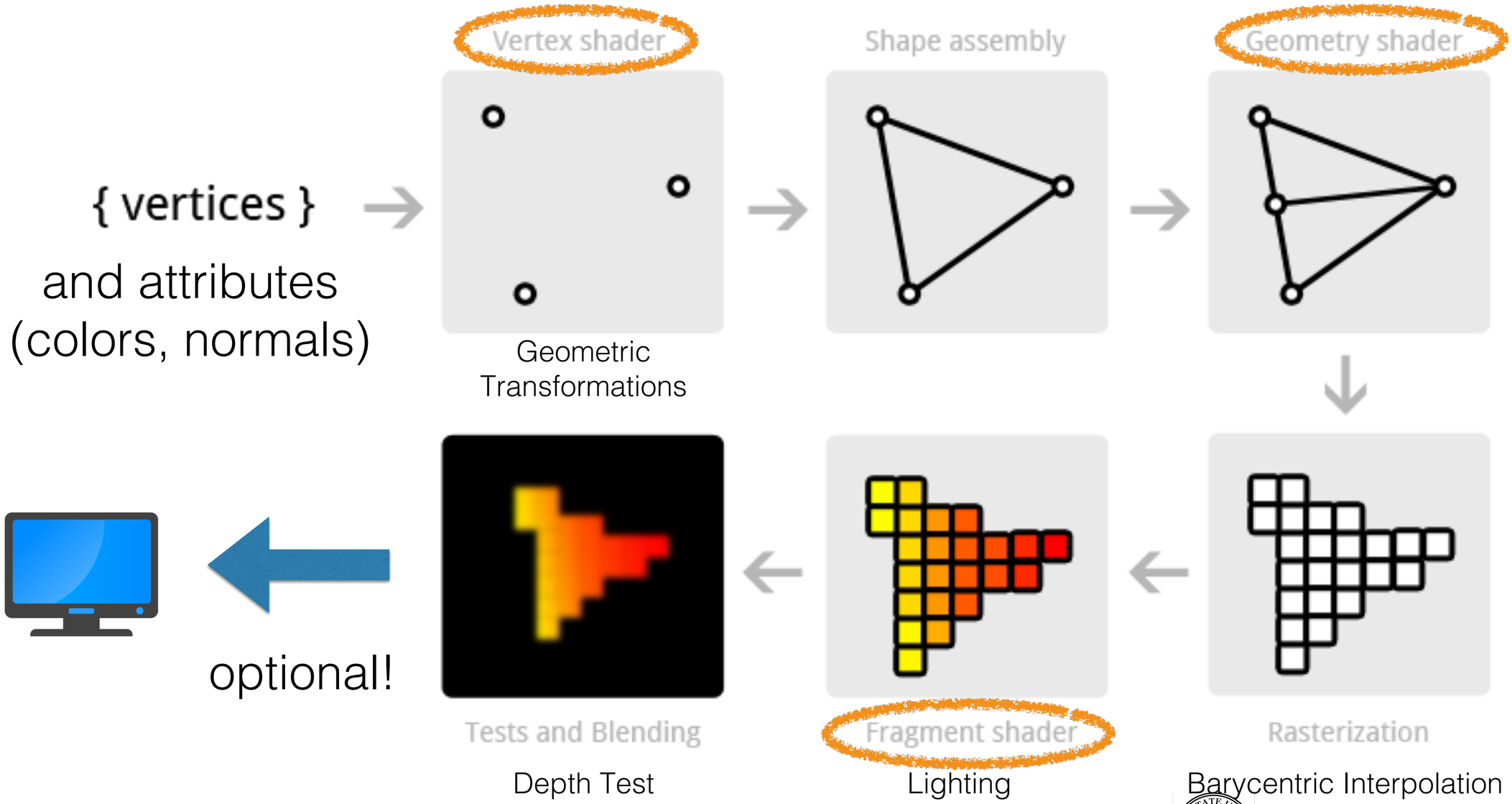
- For small applications this does not have a significant effect, but it is important to get it right, since the performance difference is massive

- `GL_STATIC_DRAW`: The vertex data will be uploaded once and drawn many times (e.g. the world).
- `GL_DYNAMIC_DRAW`: The vertex data will be changed from time to time, but drawn many times more than that.
- `GL_STREAM_DRAW`: The vertex data will change almost every time it's drawn (e.g. mouse cursor).

Shaders

- The name is historical — they were introduced to allow customization of the shading step of the traditional graphics pipeline
- In modern OpenGL, shaders are **general purpose functions** that will be executed in parallel on the GPU
- They are written in a special C-like language called GLSL. You send a string containing your program to the OpenGL API: the OpenGL driver compiles it on-the-fly and uploads it to the GPU
- Extremely powerful and simple to write!
- The only “annoyance” is that the graphic card vendors do not fully respect the standard, leading to possible compatibility problems (it is getting better, but problems still happen)

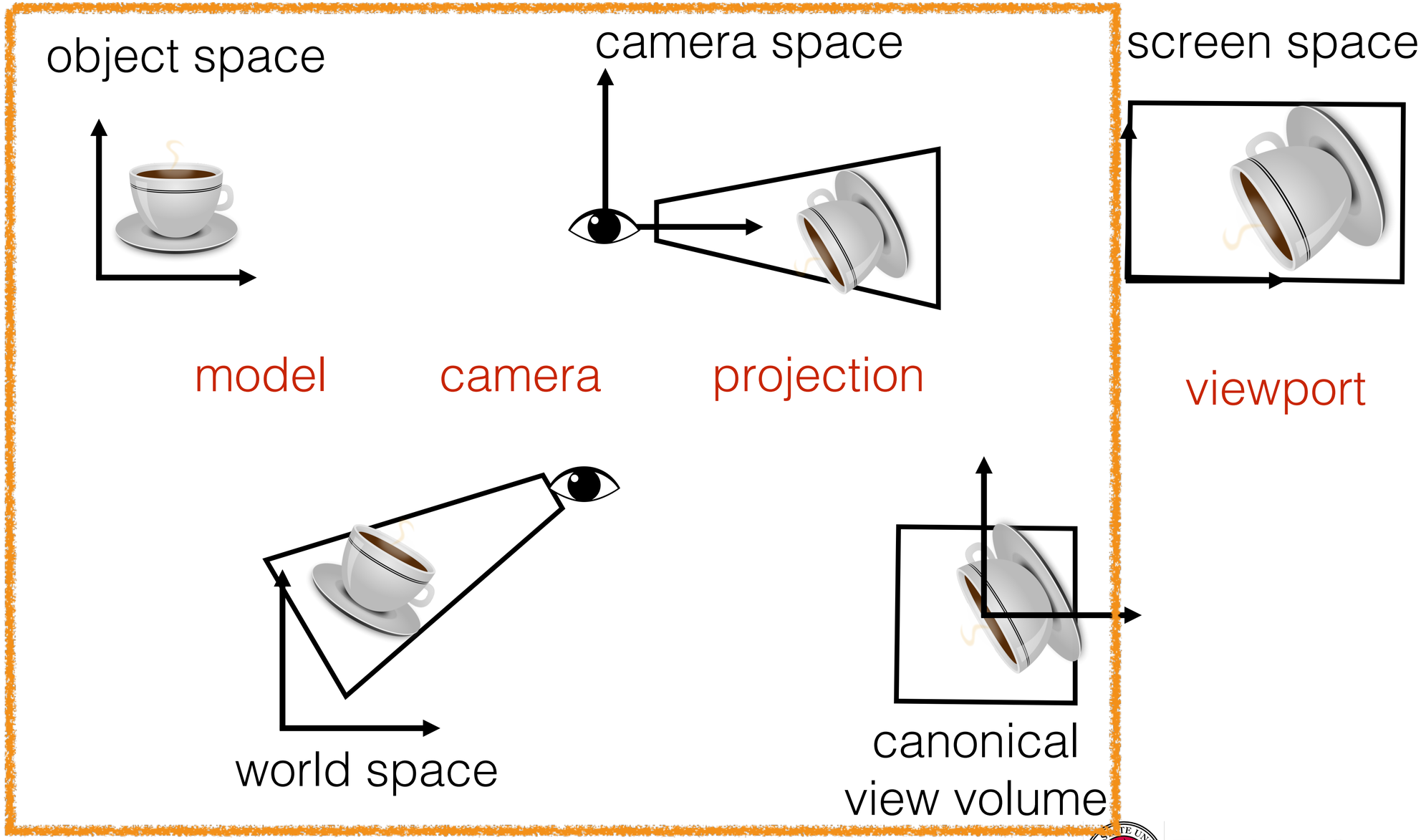
Shaders


{ vertices }
and attributes
(colors, normals)
Vertex shader
Geometric
Transformations
Shape assembly
Geometry shader
Rasterization
Barycentric Interpolation
Fragment shader
Lighting
Tests and Blending
Depth Test
optional!

Vertex Shader

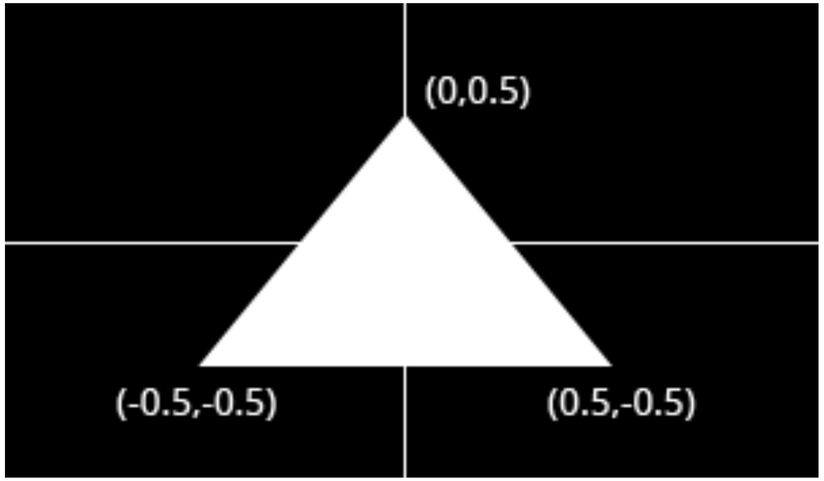
- The vertex shader is a program on the graphic card that processes each vertex and its attributes as they appear in the vertex array
- Its duty is to output the final vertex position in device coordinates and to output any data the fragment shader requires
- All transformations from **world to device coordinates** happen here

Vertex Shader



A simple vertex shader

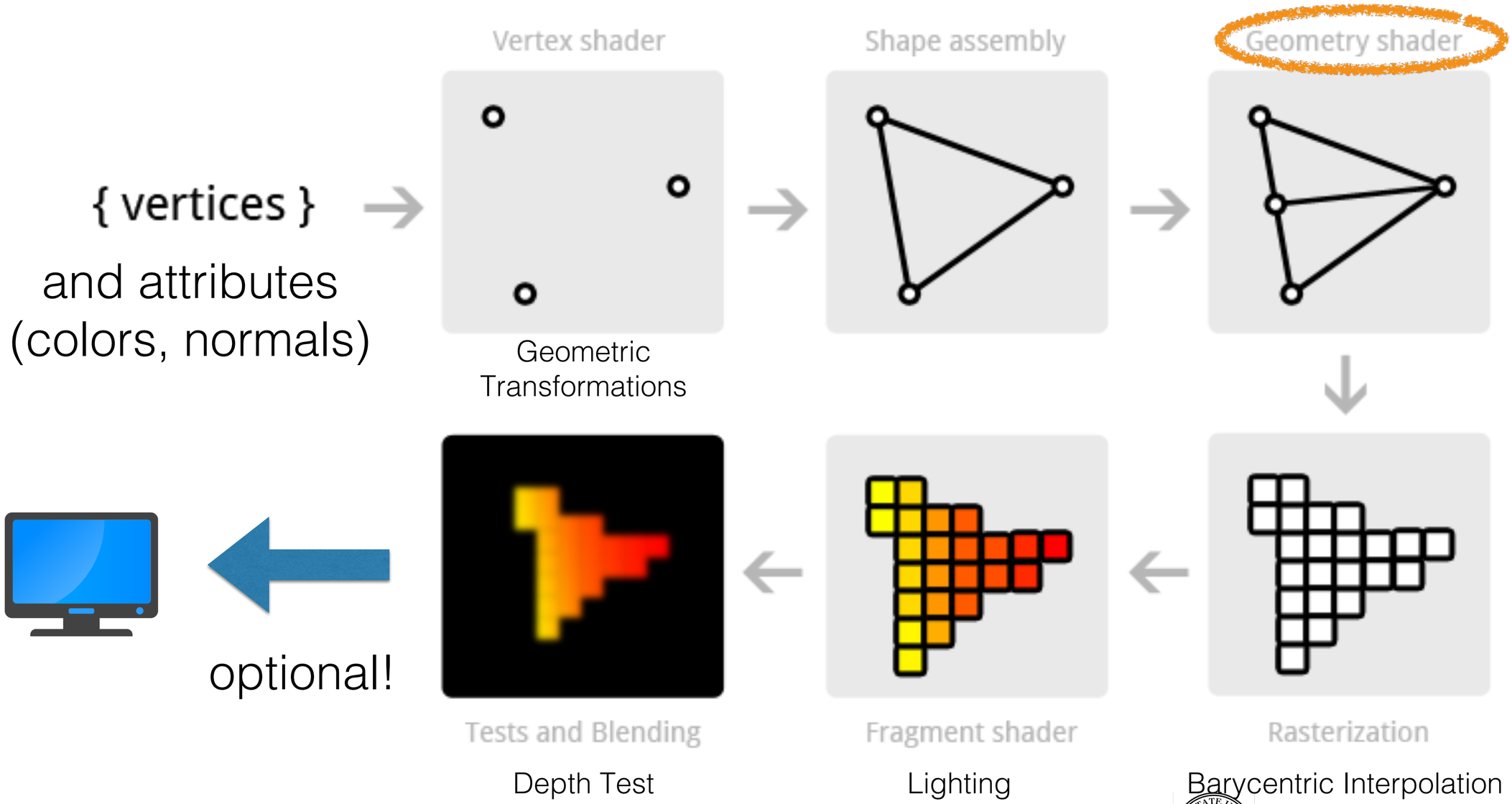
```
float vertices[] = {
     0.0f,  0.5f, // Vertex 1 (X, Y)
     0.5f, -0.5f, // Vertex 2 (X, Y)
    -0.5f, -0.5f  // Vertex 3 (X, Y)
};
```




```
#version 150

in vec2 position;

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

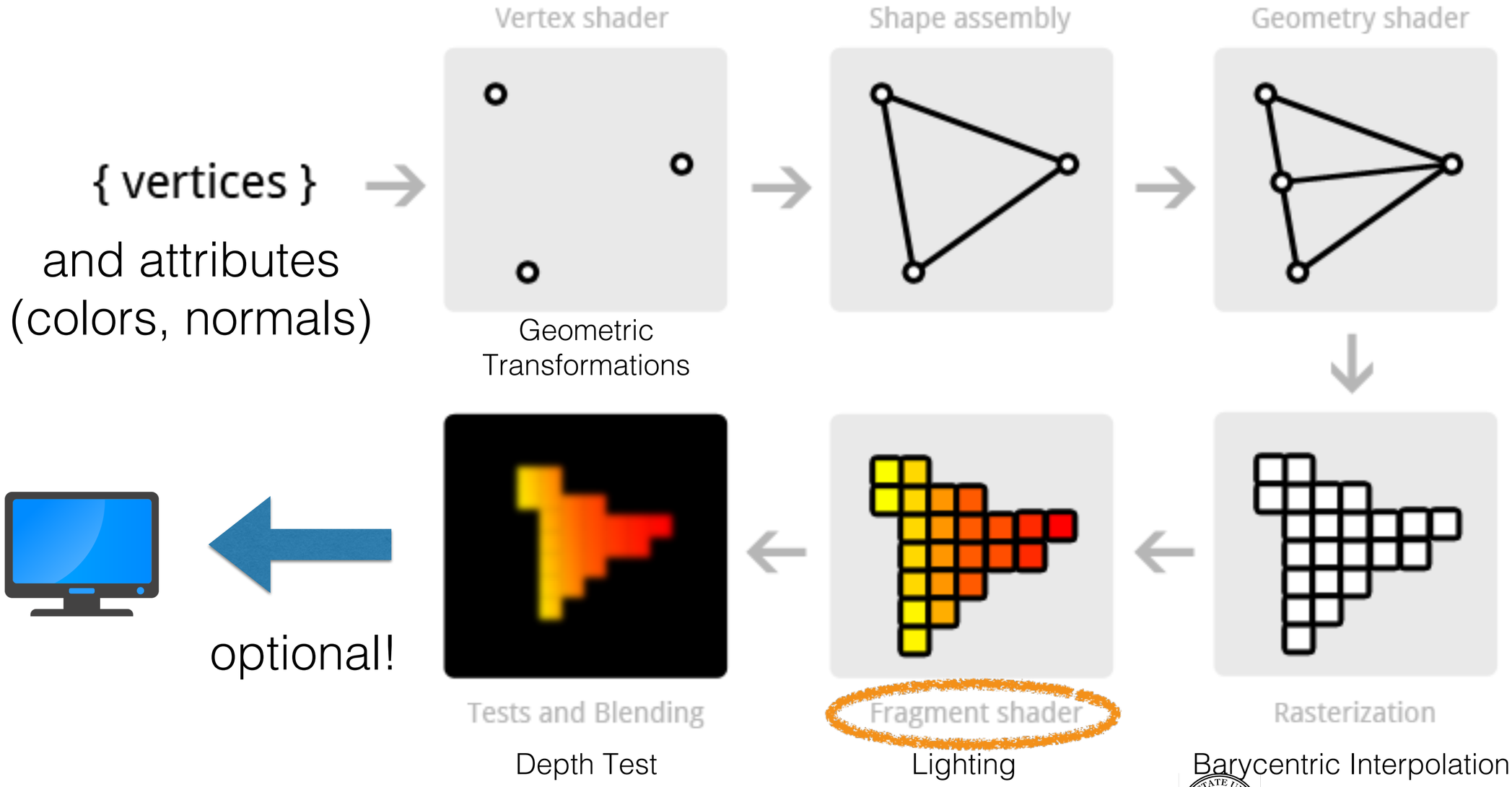
gl_Position is a special keyword

Shaders


{ vertices }
and attributes
(colors, normals)
Vertex shader
Geometric
Transformations
Shape assembly
Geometry shader
Rasterization
Barycentric Interpolation
Fragment shader
Lighting
Tests and Blending
Depth Test
optional!

Geometry Shader?

Shaders


Vertex shader
Shape assembly
Geometry shader
{ vertices }
and attributes
(colors, normals)
Geometric
Transformations
optional!
Tests and Blending
Depth Test
Fragment shader
Lighting
Rasterization
Barycentric Interpolation

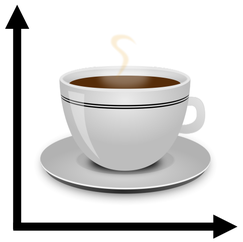
Fragment Shader

- The output from the vertex shader is interpolated over all the pixels on the screen covered by a primitive
- These pixels are called fragments and this is what the fragment shader operates on
- Just like the vertex shader it has one mandatory output, the final color of a fragment. It's up to you to write the code for computing this color from all the attributes that you attached to the vertices

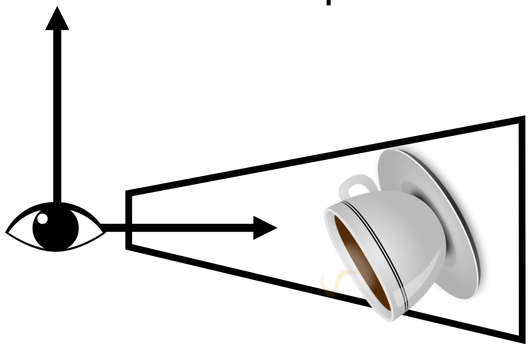
Vertex Shader

Fragment Shader

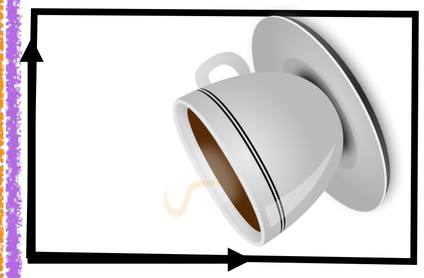
object space



camera space



screen space

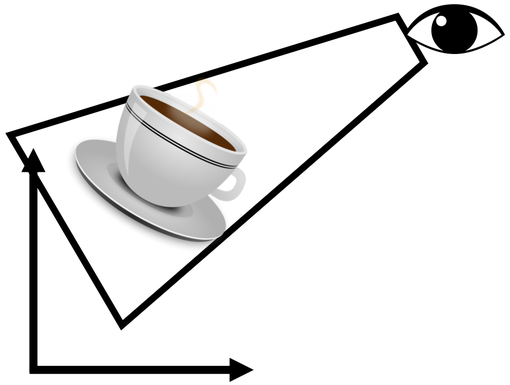


model

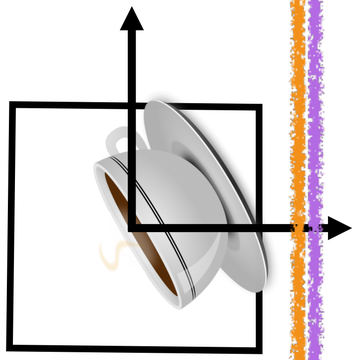
camera

projection

viewport



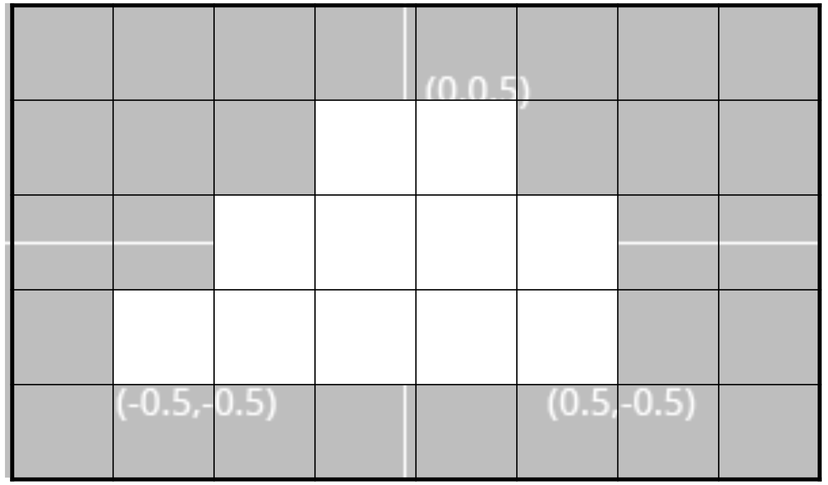
world space



canonical
view volume

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

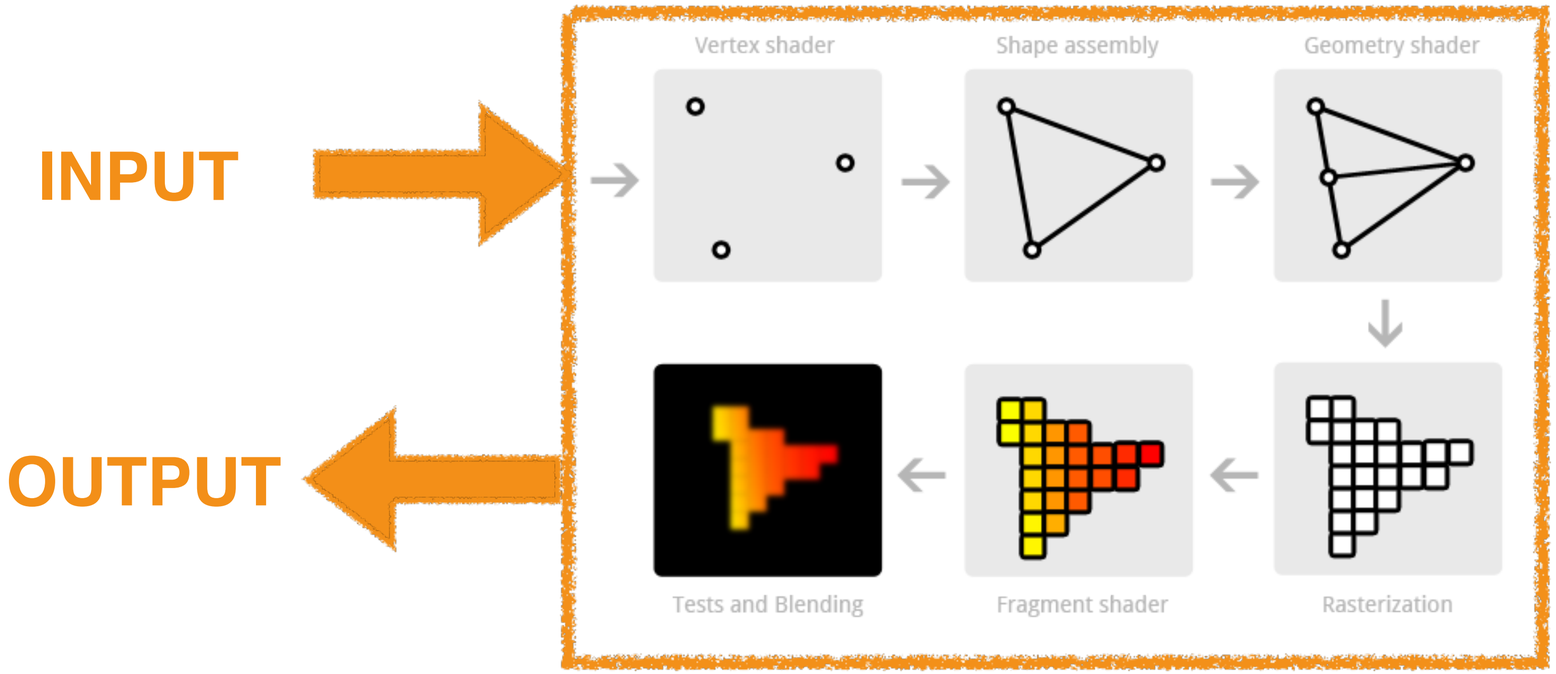
```
const GLchar* vertex_shader =
            "#version 150 core\n"
                    "in vec2 position;"
                    "void main()"
                    “{gl_Position = vec4(position, 0.0, 1.0);}";
const GLchar* fragment_shader =
            "#version 150 core\n"
                    "out vec4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



References

https://open.gl — Main reference

https://github.com/openglredbook/examples

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

Chapter 17