

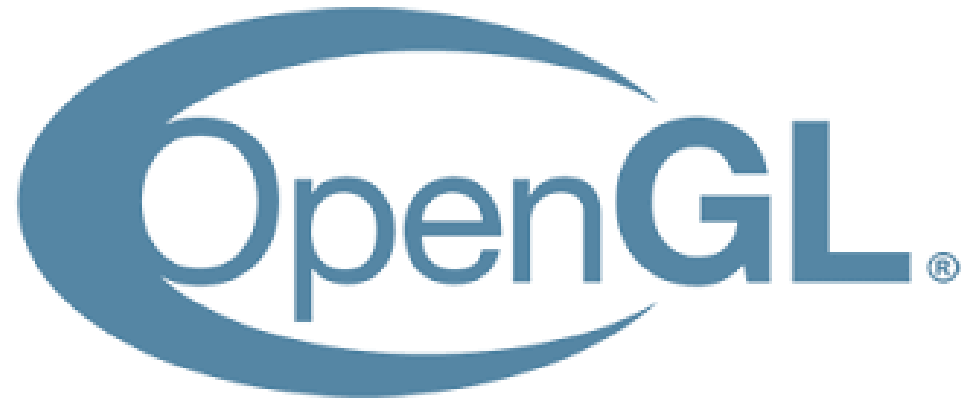
Computer Graphics (COMP0027) 2023/24

OpenGL

Tobias Ritschel

Overview

- OpenGL idea
- Pipeline overview
- Individual stages
- Example code
- Deprecated OpenGL



OpenGL: Hard or easy?

- Old-fashioned (deprecated) GL is didactical
- Easy to explain
- But
 - This is not what is out there
 - This is not what we will do here
 - We will try to see how it is in 2023

OpenGL philosophy

- Platform and language-independent
- Rendering-only
- Aims to be real-time
- Supports Graphics Hardware (GPUs)
- State system
- Client-server system
- Extendable (OpenGL Extensions)

OpenGL

- OpenGL is
 - .. a specification to create images in a frame buffer
 - .. an API to access that mechanism
 - .. well specified

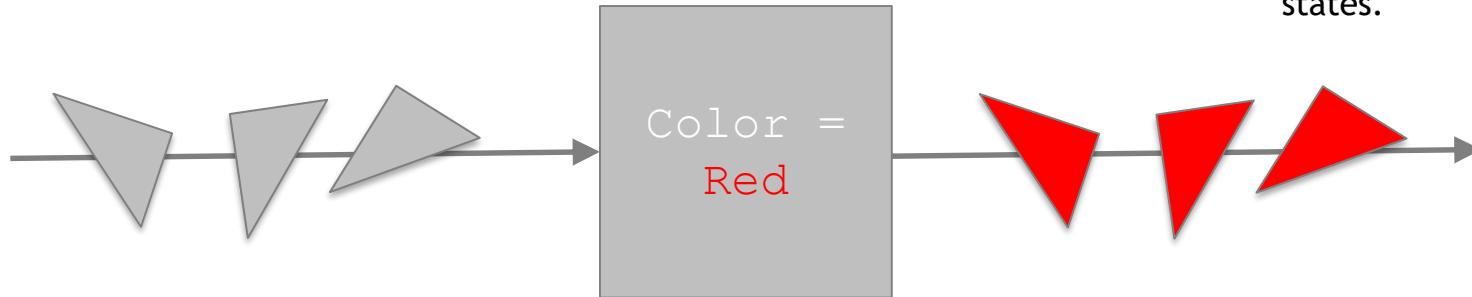
OpenGL

- OpenGL is not
 - .. a window system
 - .. a user interface
 - .. a display mechanism
 - .. a library
 - .. modeling cameras, materials or lights

What is a “state system”?



If this looks too simple: There are likely hundreds of such states.

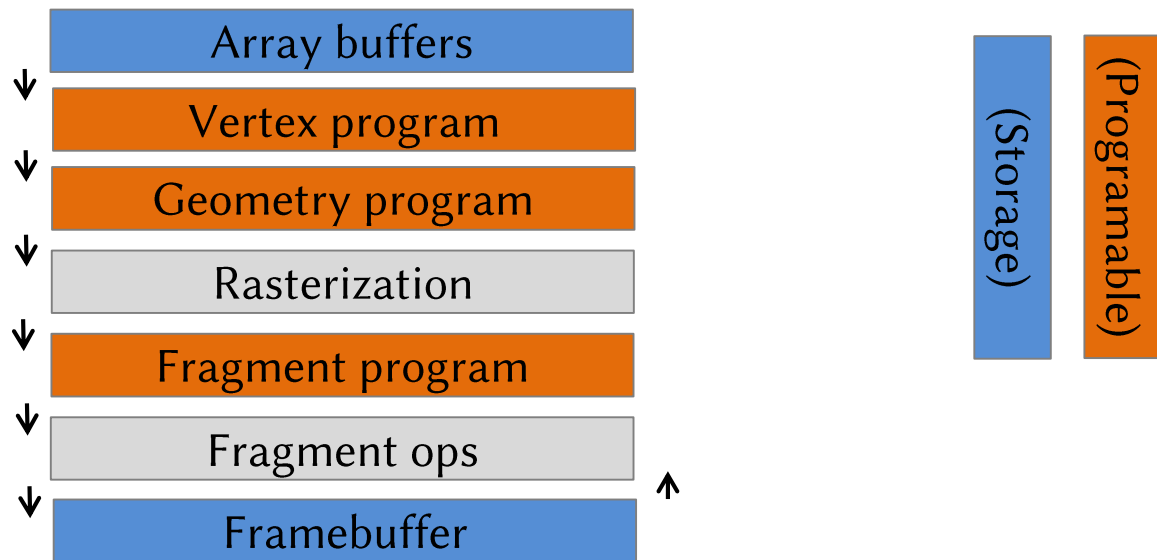


How to program OpenGL

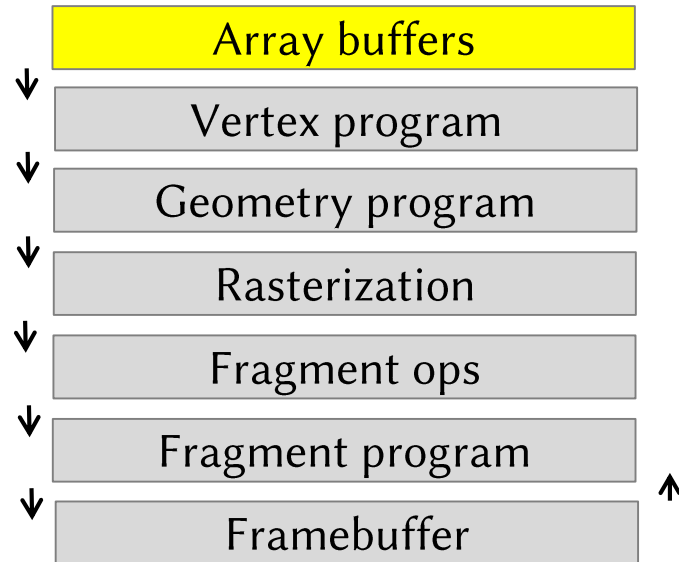
- We will here use WebGL in JavaScript
- A subset of the specification
 - More restricted (-)
 - More modern (+)
 - Runs on all devices (+)
 - Easy to run and compile (+)



OpenGL Pipeline



The OpenGL Pipeline



Array Buffers

- Just arrays of memory on the GPU
- It will take batches of data from those arrays and feed them into the pipeline
- Instead of using `new()` or `malloc()` we use gl functions like `bufferData()` to manage these.
- For now, assume they define vertices of a polygonal mesh
- Don't upload every frame. That is slow.

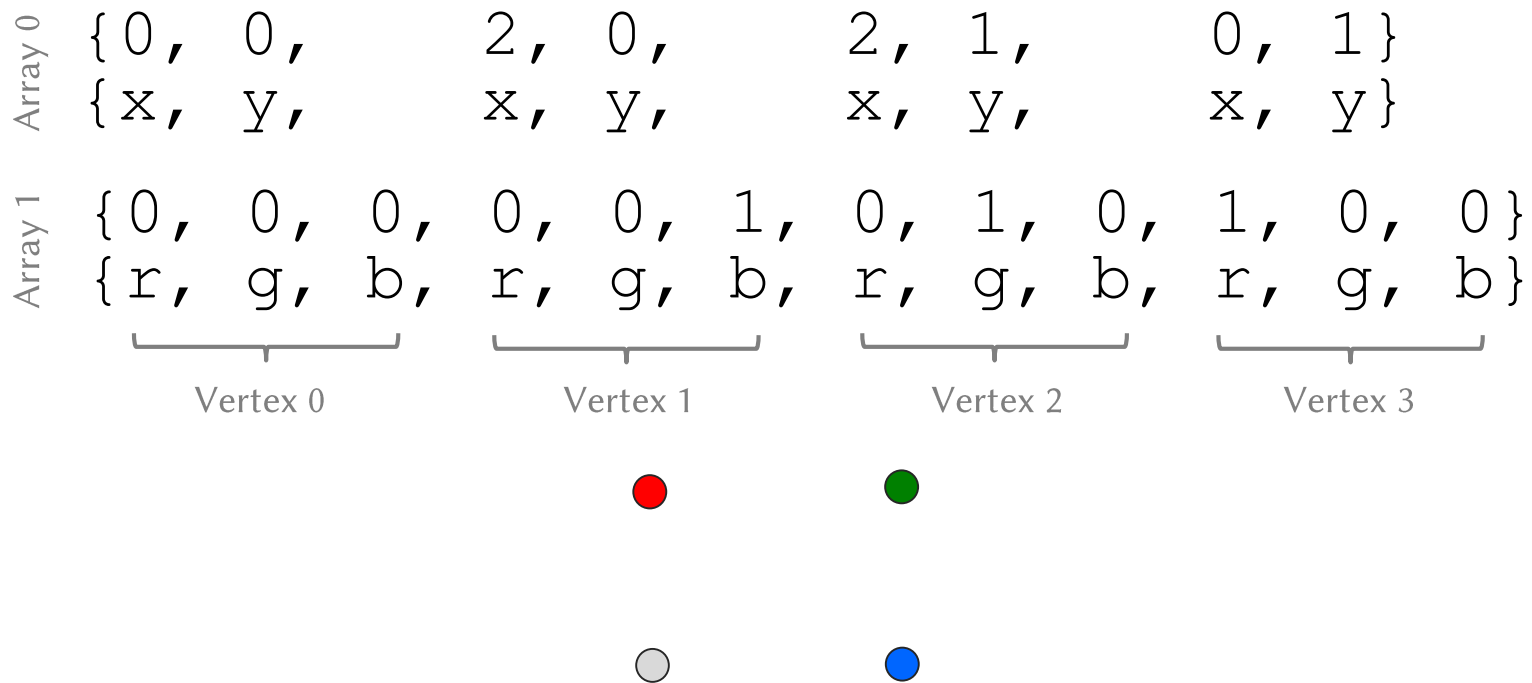
Array Buffer: Example 1

Array 0

{ 0, 0, 2, 0, 2, 1, 0, 1 }
{ x, y, x, y, x, y, x, y }
<div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div>
Vertex 0 Vertex 1 Vertex 2 Vertex 3



Array Buffer: Example 2



Array Buffer Example Uses

- Positions
- Colors
- Normals
- (Multiple) u , v texture coords
- Motion flow
- Tangent spaces for bump mapping
- Fully flexible .. no semantic!

Array Buffer Code

```
var positionBuffer = gl.createBuffer();  
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);  
gl.bufferData(  
    gl.ARRAY_BUFFER,  
    new Float32Array(getVertices()),  
    gl.STATIC_DRAW);
```


Element Arrays

- Special type of arrays
- They are used to compose vertices into polygonal primitives
- Only triangles in modern GL

Element Arrays: Example

$\{0, 0, 2, 0, 2, 1, 0, 1\}$
 $\{x, y, x, y, x, y, x, y\}$

$\{0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0\}$
 $\{r, g, b, r, g, b, r, g, b, r, g, b\}$

$\{0, 1, 2, 1, 2, 3\}$
 $\{a, b, c, a, b, c\}$

Element array: Code

```
var indexBuffer = gl.createBuffer();  
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);  
gl.bufferData(  
    gl.ELEMENT_ARRAY_BUFFER,  
    new Uint16Array(getIndices()),  
    gl.STATIC_DRAW);
```

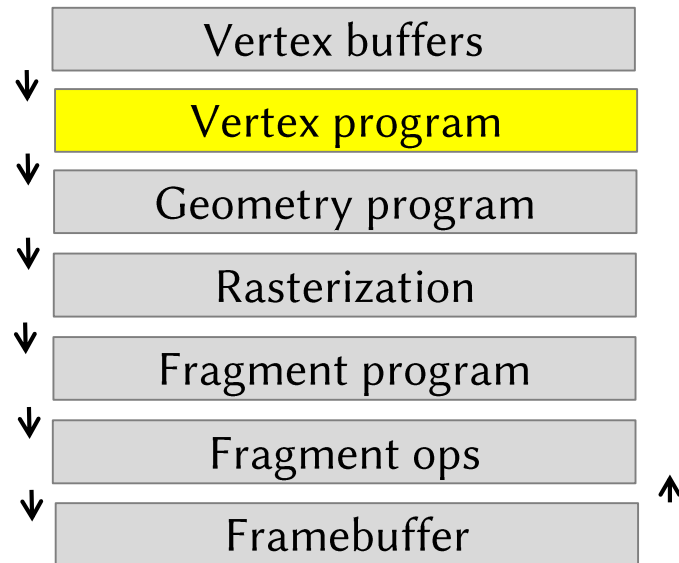
Issuing a draw call

- Issue draw call with an element array buffers

```
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, tri);  
gl.drawElements(gl.TRIANGLES,  
    tri.numItems,  
    gl.UNSIGNED_SHORT,  
    0);
```

DEMO

Definitions and changing a few array values



Vertex program

- Also called “shader”
- There are so many things that can happen to a vertex, that this needs a full programming language: GLSL
- Executed
 - at every vertex
 - in parallel

Vertex program

- Input:
 - Vertices from array buffers now called **attributes**
 - Some **uniforms** that are the same for all vertices
- Output:
 - The clip space **coordinate** of that vertex (before division with w)
 - Every VS output you plan to use per-pixel called **varyings**

What **array** goes into which **attribute**?

- Everything possible
- Programmable mapping
- Example for 6 positions of cube:

```
gl.bindBuffer(gl.ARRAY_BUFFER, buffer);  
var location = gl.getAttribLocation(shaderProgram, "position");  
gl.enableVertexAttribArray(location);  
gl.vertexAttribPointer(location, 6, gl.FLOAT, false, 0, 0);
```

Vertex program: Example

- Input:

- attribute position
- uniform direction
- uniform matrix

```
uniform vec3 direction;  
uniform mat4 matrix;  
attribute vec3 position;  
varying color;
```

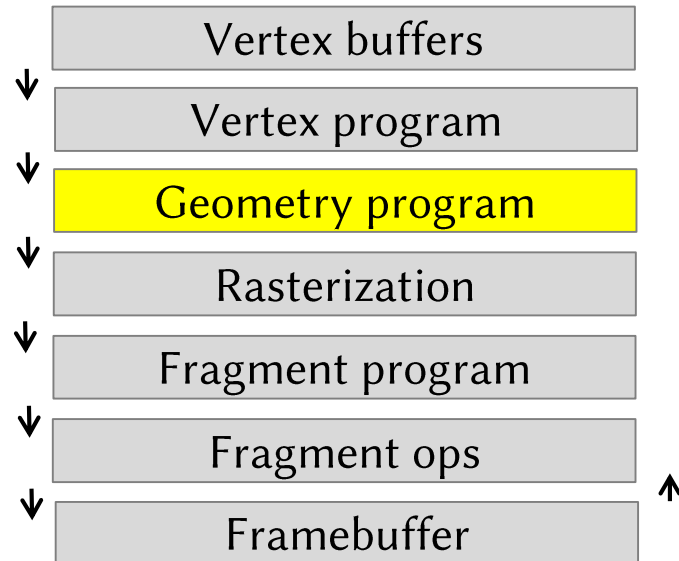
- Output:

- Clip-space coordinate
- varying color

```
void main() {  
    gl_Position = matrix * position  
    color = vec3(dot(position, direction));  
}
```

DEMO

Code-walk VS, adding a bit of animation via uniform



Geometry programs

- So far have processed single **vertices**, not **primitives**
- Remember: Use element array to turn vertices into primitives
- **Geometry** programs can change entire primitives
- Not yet in WebGL implementations. So no demo.

Geometry program

- Input:
 - All attributes (e.g. 3) that form a primitive
- Output:
 - One or multiple new primitives

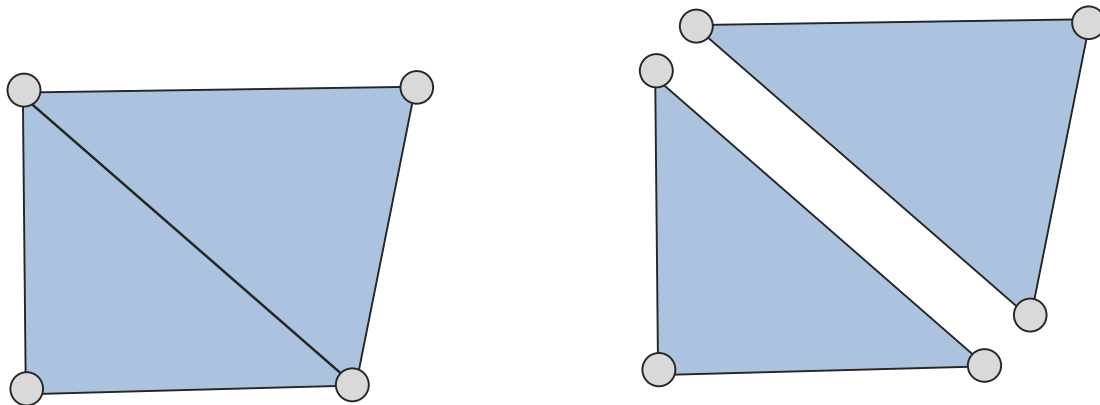
Geometry program: Example

- Input:
 - Tri positions
 - Tri normals
 - A magic jump
- Output:
 - The same, just changed

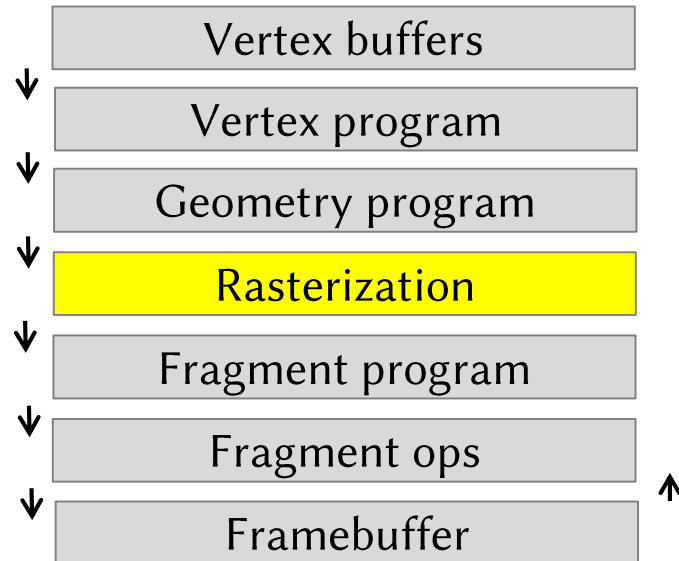
```
in vec3 ip[3]; in vec3 in[3];  
out vec3 op[3]; out vec3 on[3];  
uniform float jump;
```

```
void main() {  
    for(int i = 0; i<3; i++) {  
        op[i] = ip[i] + jump * in[i];  
    }  
    on = in;  
}
```

Geometry program: Example result



Quiz: Why is this so interesting, could a VS do that?

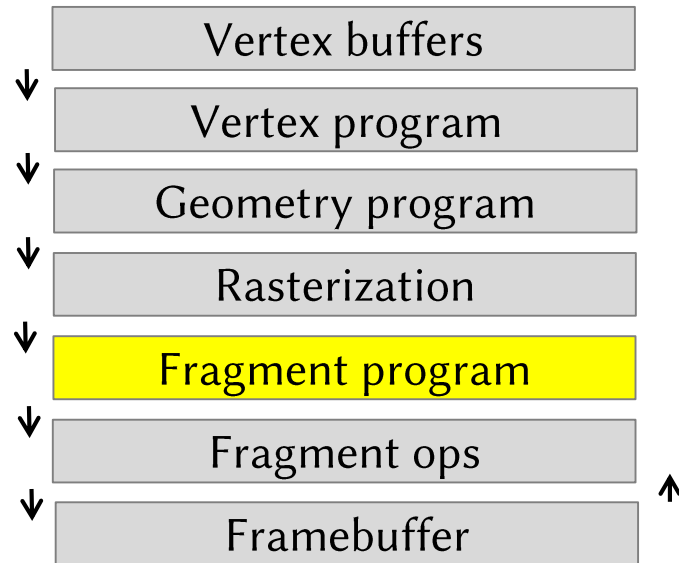


Rasterization

- Does what you learned
 - Clipping
 - Projective division
 - Culling
 - (perspective) Interpolation
- Can turn on and off culling and some other things

DEMO

Culling on and off (without depth buffering)



Fragment program

- The most important one
- Executed for every “fragment” (lots of)
- Without antialiasing a fragment is a pixel
- With anti-aliasing, multiple fragments go into a pixel

Fragment program

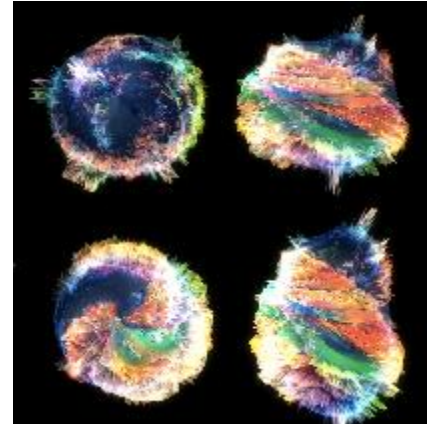
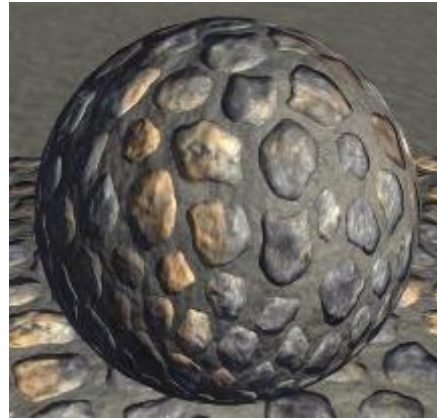
- Input
 - All the varyings the geometry shader outputs
 - The fragment coordinate
 - Uniforms
 - Samplers (GL name for textures)
- Output
 - A fragment color `gl_FragColor`
 - Optionally, depth `gl_FragDepth`
 - Can also discard fragments

Fragment program: Example

- Input:
 - varying texCoord
 - Sampler texture
- Output:
 - Color
 - Depth

```
varying vec2 texCoord;  
uniform sampler2d testSampler;  
  
void main() {  
    gl_FragColor.rgb = texture(  
        testSampler,  
        texCoord).rgb;  
    gl_FragDepth = texture(  
        testSampler,  
        texCoord).a;  
}
```

Shader Examples



DEMO

Pixel shading code, replace with some overly simplified ones

Textures

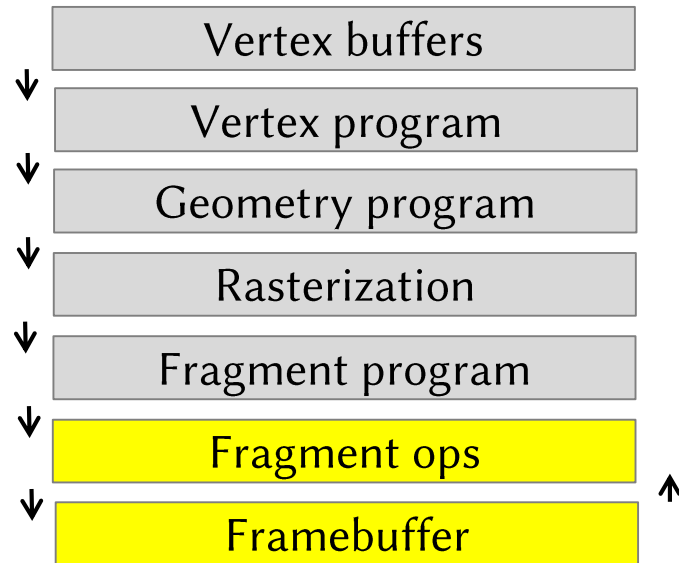
- Textures are storage objects like array buffers
- Have to allocate and fill them with specific calls
- Have the filtering modes explained in last lecture
- Read in every program stage using the function

```
texture(sampler, vec2 texCoord)    w.texCoord 0..1
```

```
texelFetch(sampler, ivec2 texCoord)    w.texCoord 0..N
```

DEMO

Texture allocation & turning it on in fragment program



Framebuffer

- How fragment colors affects the frame buffer color
- Configurable, but not programmable
 - Depth test
 - Blending
 - Multiple render targets / render-to-texture

Depth test

- As you would expect
- Need to allocate and clear the depth buffer to use it
- *Off* by default

DEMO

Turing on and off depth test (with culling off, ideally)

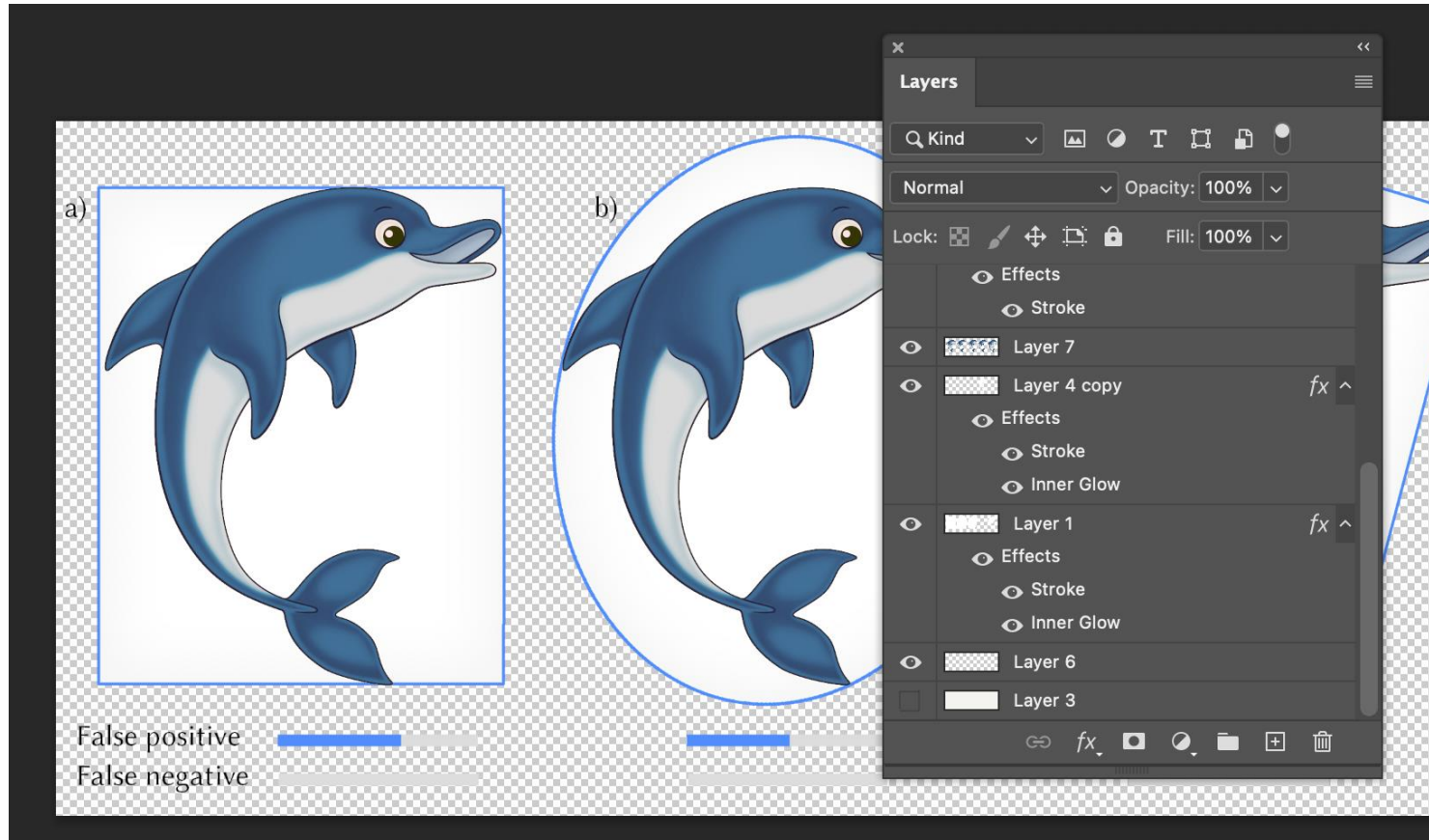
Alpha



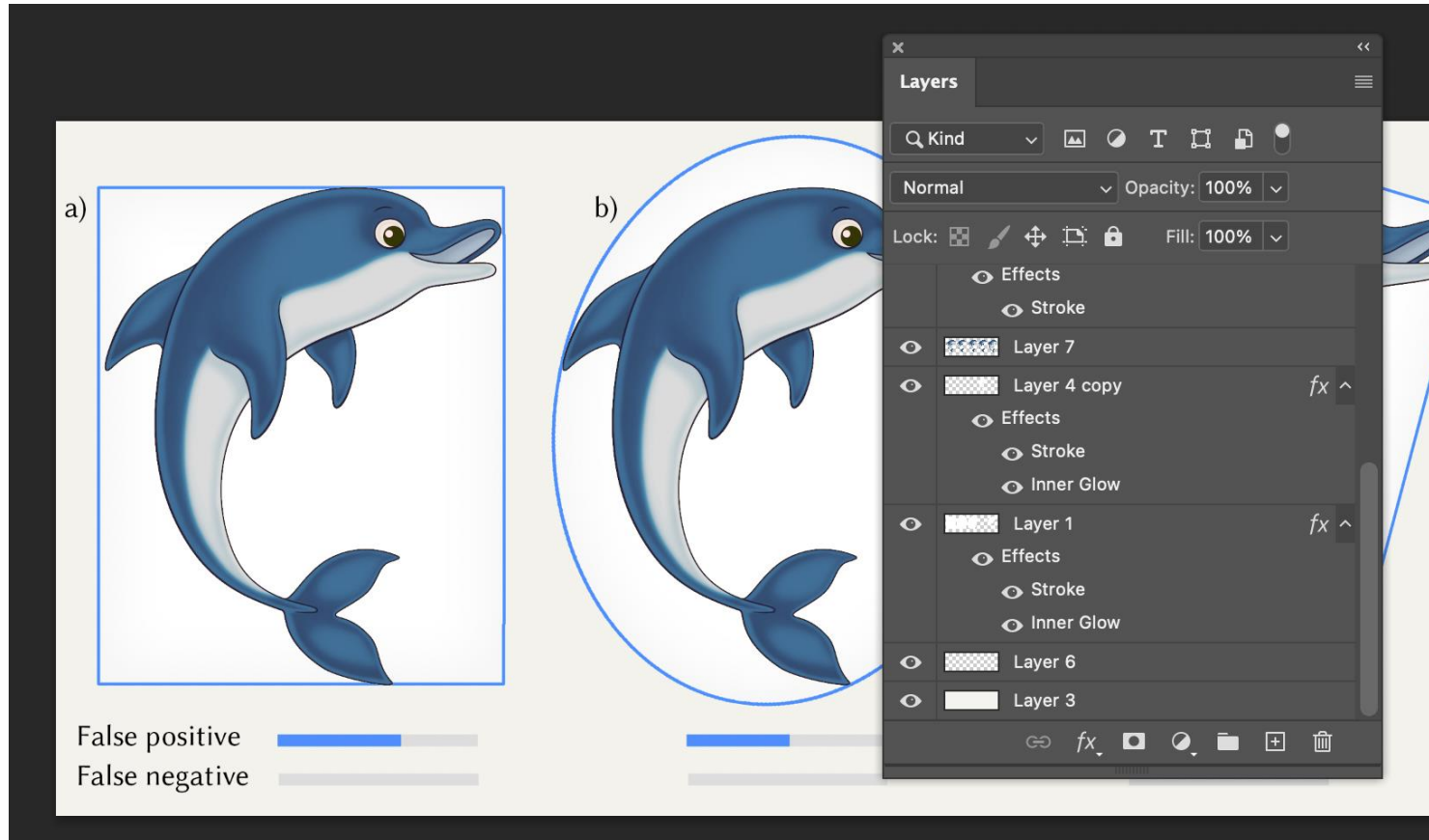
Alpha



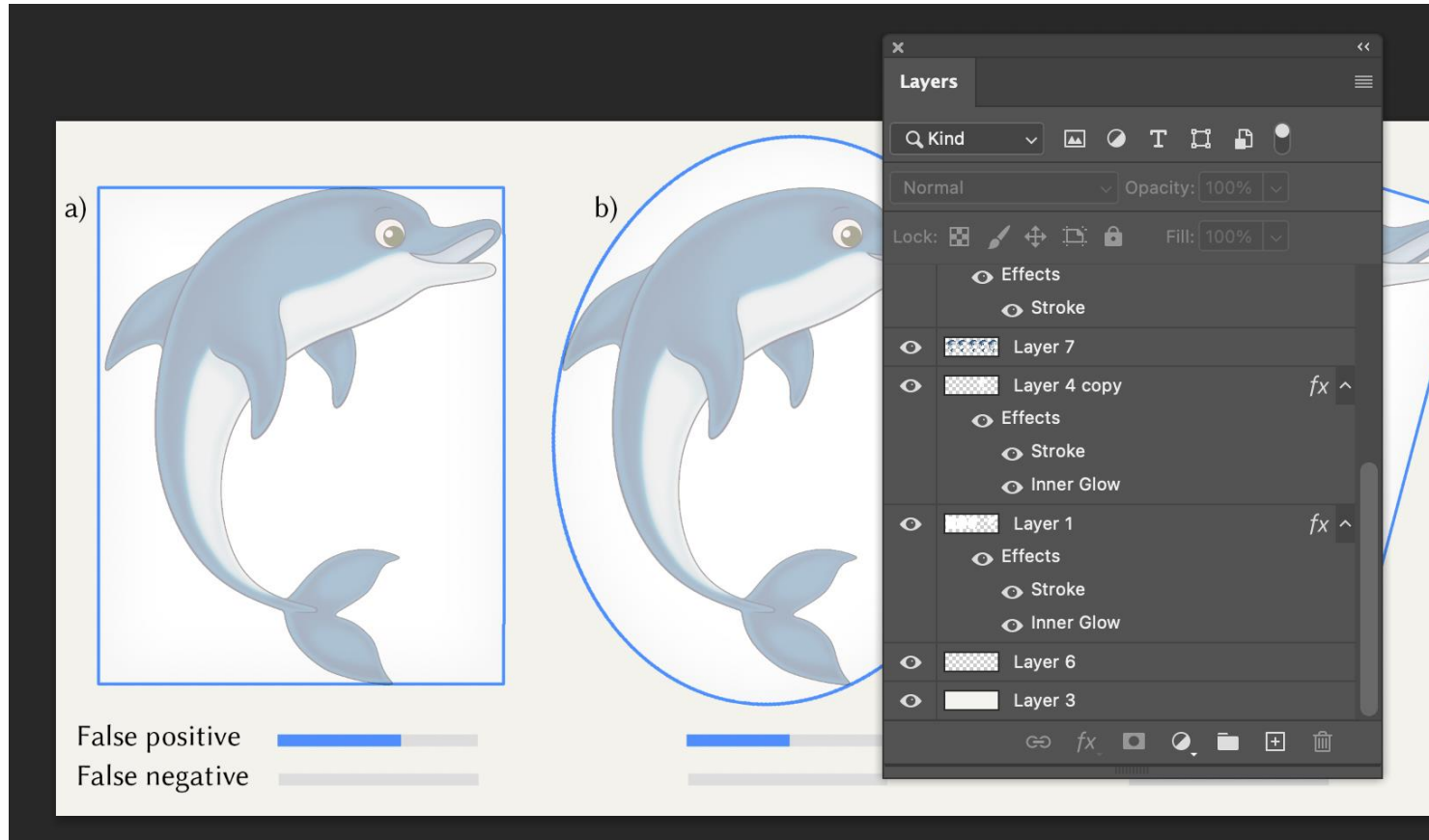
Photoshop



Photoshop



Photoshop



Alpha

- Alpha models

- Opacity or

- If 0 it is transparent,
 - if 1 it is opaque

- Transparency

- If 0 it is opaque,
 - if 1 it is transparent

- Both are in use

不透明度

透明度

不透明 / opaque /

RGB vs RGBA

- Instead of using RGB, three-dimensional color
- We will use RGBA, four-dimensional color
- Interpolate in exactly the same way
- Even the framebuffer you draw to is supposed to have alpha, to be RGBA
- You may think about this as a homogeneous space, but I will not insist and it is a debated view

但这 diffuse 啥
不考虑 alpha 通道

Alpha textures

- Not only storing RGB, but also alpha
- Rest is as everything else
 - Filtering
 - Interpolation
 - Texture coords

Blending

- If depth test passes, GL does not just replace the color such as we do in our coursework
- Instead, it evaluates an equation involving old and new color, including alpha, a notion of transparency/opacity

Old code

depth of
pixel which is
already there

```
if(src_depth < dest_depth) {  
    color_buffer = src_color  
    depth_buffer = src_detph  
}
```

New code

```
if(src_depth < dest_depth) {  
    color_buffer = magic(src_color, more)  
    depth_buffer = src_detph  
}
```

OpenGL's Blending Equation

$$x_{out} = x_{op}(x_{src} * x_{src_fac}, x_{dest} * x_{dest_fac})$$

- x can be RGB or ALPHA
- x_{out} result
- x_{src} fragment program output
- x_{dest} current frame buffer content
- Can configure x_{op} , x_{src_fac} and x_{dest_fac}

factor
↓
factor

配置

`x_op`

- Can be, either
 - `GL_ADD`
 - `GL_MINUS`
 - `GL_MULTIPLY`

x_src ^{func} and x_dest ^{fac}

- Both can be, either
 - GL_ZERO
 - GL_ONE
 - GL_SRC_ALPHA
 - GL_ONE_MINUS_SRC_ALPHA

时 效 好
在 OpenGL 性能
设置

DEMO

Additive blending

You have to start somewhere

- The initial state matters
- glClear()
- Also used to reset the depth or stencil buffer
- So far it was fine with black, but maybe you want something else
- Example: Adding some clouds to a blue sky



Blending may imply ordering 顺序



Blending may imply ordering



$$(R * 0.9 + G * 0.1) * 0.8 + B * 0.2 \neq (B * 0.9 + G * 0.1) * 0.8 + R * 0.2$$

order matters

~~src-factor~~ 0.9 0.1 \Rightarrow $\frac{0.8}{\text{src-factor}} \frac{0.2}{\text{dst-factor}}$

So what will this code look like? part of CW!

```
// Fosure some setup of states

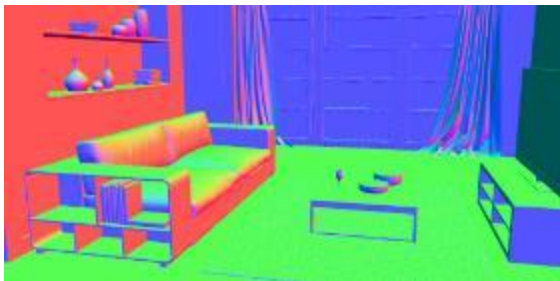
vec4 magic(...) {
    // Using the state
}

if(src_depth < dest_depth) {
    color_buffer = magic(src_color, dest_color, ?)
    depth_buffer = src_detph
}
```

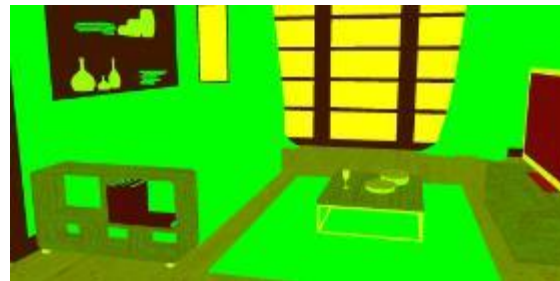
Multiple render targets

- No reason to output only a single color
- Can output multiple colors
- Each goes into its own framebuffer
- This frame buffer can then again be used as input texture in a new shader
- Example: Deferred shading
- First fill framebuffer, then only shade what you see

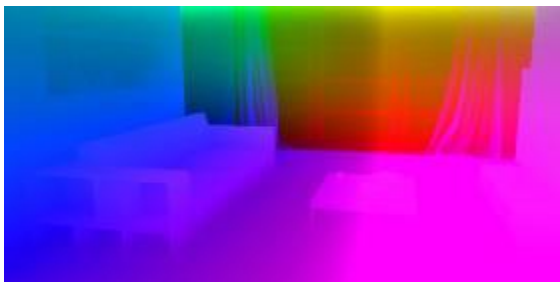
Multiple render targets



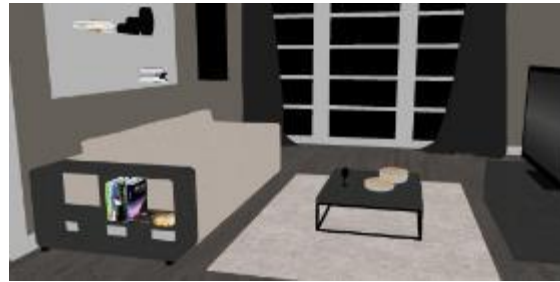
Position



Material



Normal



Reflectance

Deprecated OpenGL

- Modern GL does not do some things old GL did:
 - `glVertex` / `glColor` / ...
 - `glMatrix` & Matrix stack
 - `glLight`
 - `glMaterial`
 - `GL_QUAD` / `GL_POLYGON`
- Will make many tutorials or courses out there break

Conclusion

- OpenGL is an specification
- Implements the rasterization pipeline we know
- Allows to configure many stages
- Allows to program some stages
- It turns primitives from buffers into a frame buffer using other buffers and programs