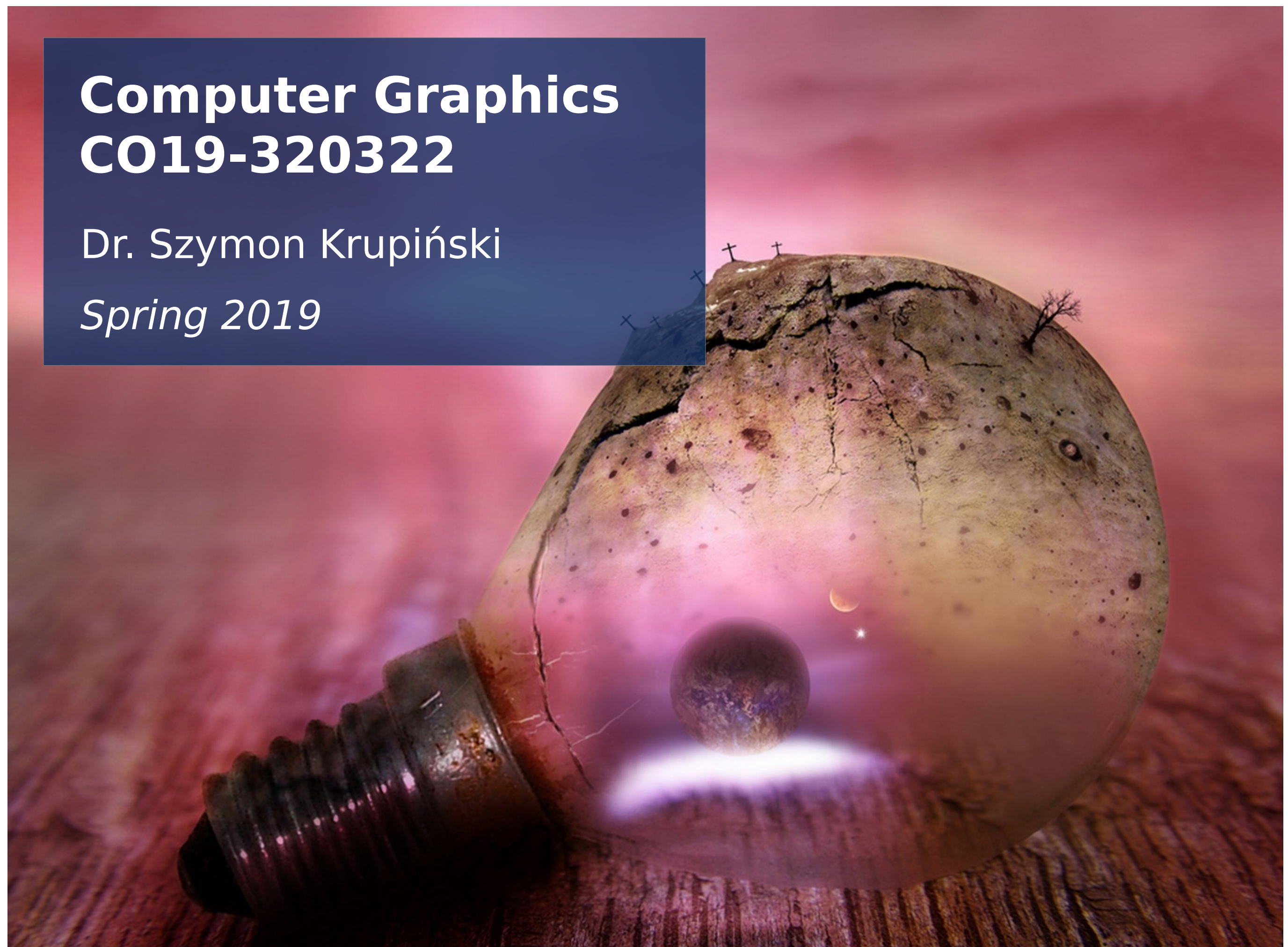
Lecture 17: Shading

JACOBS UNIVERSITY



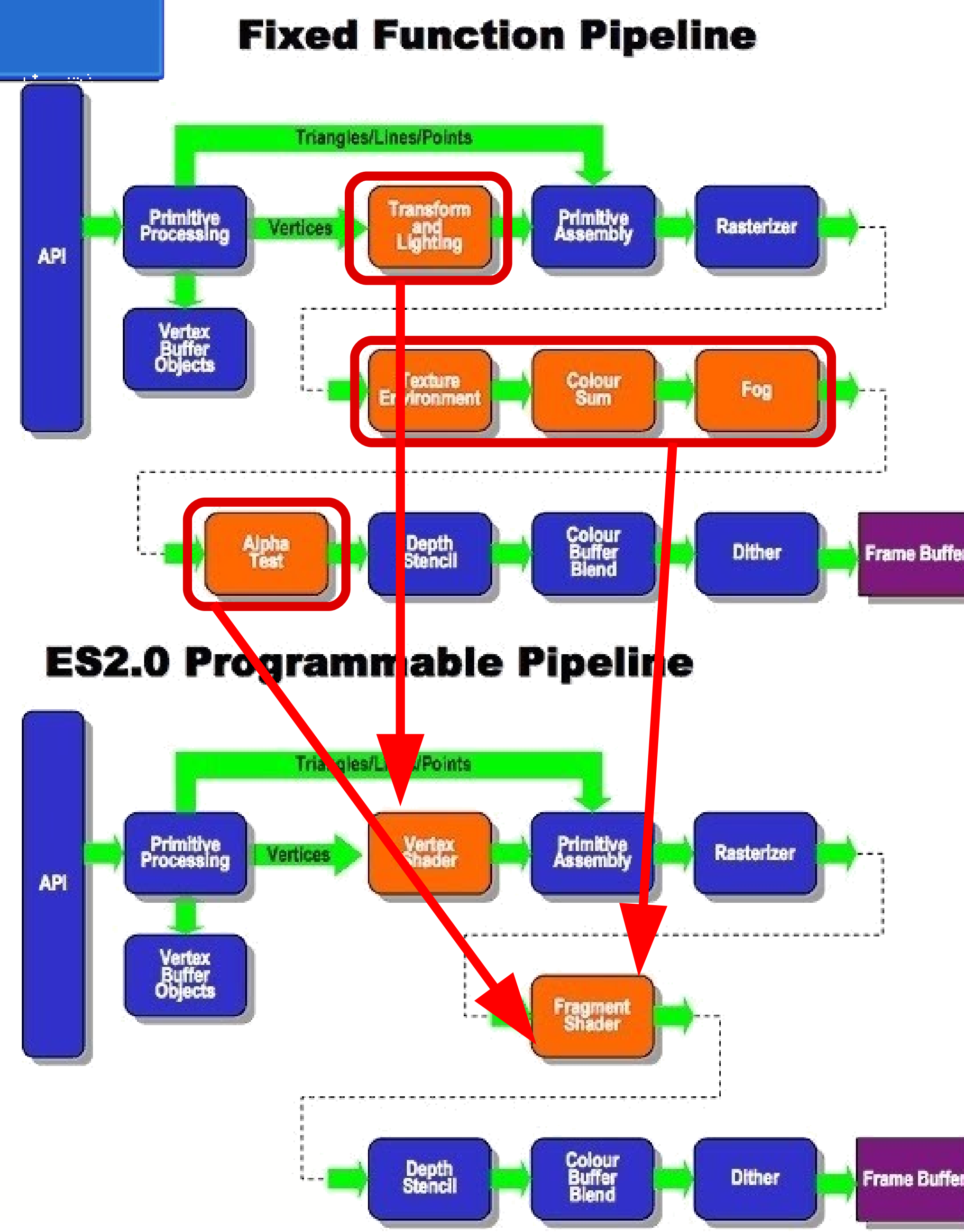
Computer Graphics
CO19-320322

Dr. Szymon Krupiński

Spring 2019

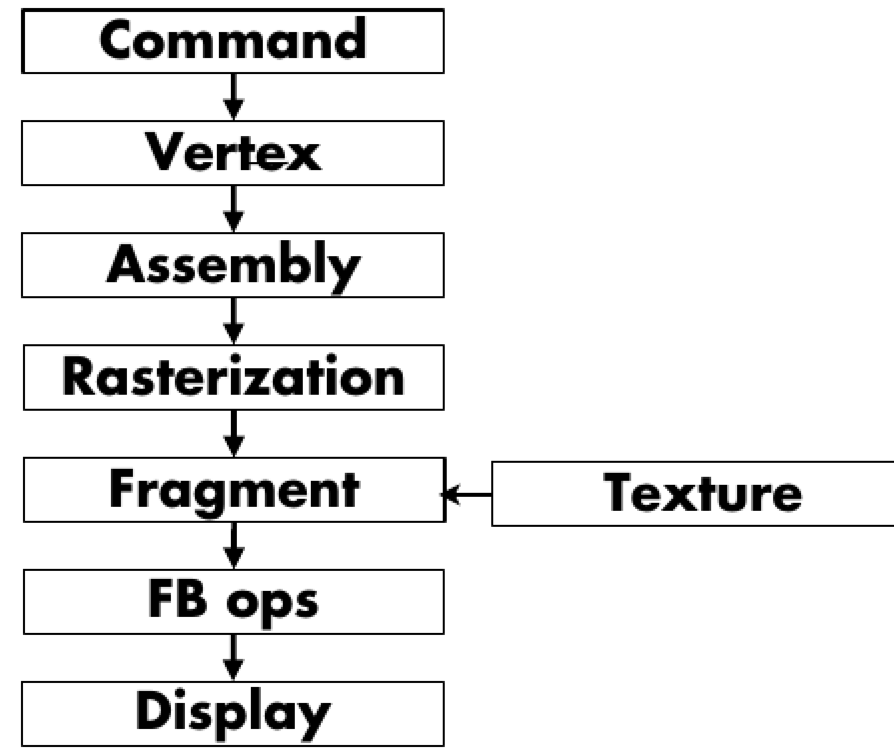
Programmable pipeline

- Fixed-function pipeline is replaced by programmable pipeline
  - More flexible use of the computational resources of the graphics card – GPU
  - Special programs have to be provided to do the (massively parallel) computation in there at certain stages of the pipeline



Switching between styles?

- Fixed pipeline was supposed to be removed in core OpenGL 3.1 and above
- Practically: it is still there in most implementations!
- On your installation, it is likely that OpenGL libraries will run the fixed function pipeline until one of your instruction will bring up the shaders
- Switching back and forth is possible – the shaders will have to be unloaded

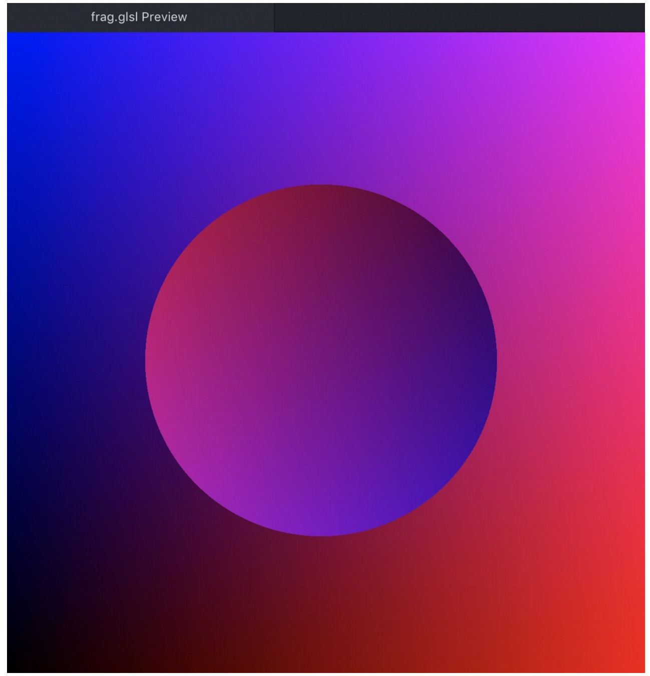


GLSL

- “C-like” minilanguage to process the information flowing in the CG pipeline at certain discrete stages
- Compiled to bytecode by OpenGL
- ...then to GPU-specific code by the GPU driver
- Has to be stitched together with the main program code by making appropriate links and using pre-defined structures

frag.glsl

```glsl
// The result of the frag.cson snippet example

precision mediump float;

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

float map(float value, float inMin, float inMax, float outMin, float outMax) {
  return outMin + (outMax - outMin) * (value - inMin) / (inMax - inMin);
}

void main() {
  vec2 uv = gl_FragCoord.xy / u_resolution;

  vec3 color = vec3(uv.x, 0.0, uv.y);

  float aspect = u_resolution.x / u_resolution.y;
  uv.x *= aspect;

  vec2 mouse = u_mouse;
  mouse.x *= aspect;

  float radius = map(sin(u_time), -1.0, 1.0, 0.25, 0.3);

  if (distance(uv, mouse) < radius){
    color.r = 1.0 - color.r;
    color.b = 1.0 - color.b;
  }

  gl_FragColor = vec4(color, 1.0);
}
```



Vertex shader anatomy

```
#version 120
```
Version and profile information to the compiler

```
attribute vec3 inputPosition;
attribute vec2 inputTexCoord;
attribute vec3 inputNormal;
```
The loaded data concerning this sample or vertex

```
uniform mat4 projection, modelview, normalMat;
```
Program parameters

```
varying vec3 normalInterp;
varying vec3 vertPos;
```
(Interpolated) data provided by the rasterizer

```
void main(){
    gl_Position = projection * modelview * vec4(inputPosition,
1.0);
    vec4 vertPos4 = modelview * vec4(inputPosition, 1.0);
    vertPos = vec3(vertPos4) / vertPos4.w;
    normalInterp = vec3(normalMat * vec4(inputNormal, 0.0));
}
```
Setting the outputs

Fragment shader example

```
precision mediump float;

varying vec3 normalInterp;
varying vec3 vertPos;

uniform int mode;

const vec3 lightPos = vec3(1.0,1.0,1.0);
const vec3 lightColor = vec3(1.0, 1.0, 1.0);
const float lightPower = 40.0;
const vec3 ambientColor = vec3(0.1, 0.0, 0.0);
const vec3 diffuseColor = vec3(0.5, 0.0, 0.0);
const vec3 specColor = vec3(1.0, 1.0, 1.0);
const float shininess = 16.0;
const float screenGamma = 2.2; // Assume the
monitor is calibrated to the sRGB color space

void main() {

  vec3 normal = normalize(normalInterp);
  vec3 lightDir = lightPos - vertPos;
  float distance = length(lightDir);
  distance = distance * distance;
  lightDir = normalize(lightDir);

  float lambertian = max(dot(lightDir,normal),
0.0);
  float specular = 0.0;

  if (lambertian > 0.0) {

    vec3 viewDir = normalize(-vertPos);

    // this is blinn phong
    vec3 halfDir = normalize(lightDir + viewDir);
    float specAngle = max(dot(halfDir, normal), 0.0);
    specular = pow(specAngle, shininess);

    // this is phong (for comparison)
    if(mode == 2) {
      vec3 reflectDir = reflect(-lightDir, normal);
      specAngle = max(dot(reflectDir, viewDir), 0.0);
      // note that the exponent is different here
      specular = pow(specAngle, shininess/4.0);
    }
  }
  vec3 colorLinear = ambientColor +
         diffuseColor * lambertian * lightColor * lightPower / distance +
         specColor * specular * lightColor * lightPower / distance;
  // apply gamma correction (assume ambientColor, diffuseColor and specColor
  // have been linearized, i.e. have no gamma correction in them)
  vec3 colorGammaCorrected = pow(colorLinear, vec3(1.0/screenGamma));
  // use the gamma corrected color in the fragment
  gl_FragColor = vec4(colorGammaCorrected, 1.0);
}
```

GLSL Shaders

- They could be as simple as this:
- Vertex shader

```
#version 330 core
    layout (location = 0) in vec3 aPos;
    void main()
    {
       gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);
    }
```

- Fragment shader

```
#version 120
    void main()
    {
       gl_FragColor = vec4(0.3f, 0.5f, 0.2f, 1.0f);
    }
```

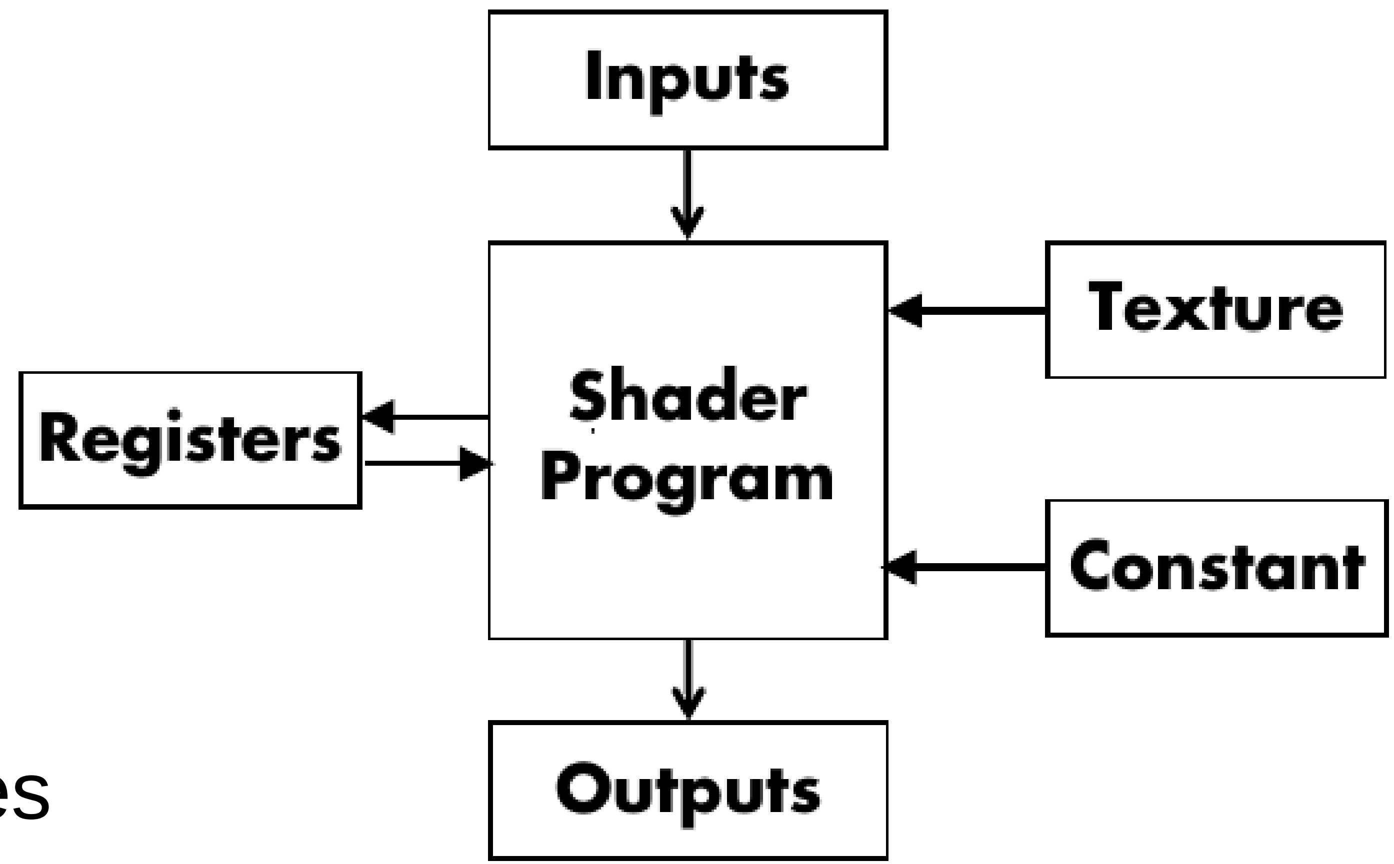
- They just have to set the essential information!
  - Here: the vertex position and the sample color

GLSL Shaders

- The body of the shader is provided in the program as a string function
  - It can be read from an external file – but the user has to provide the appropriate code

GLSL Shaders

- The environment in which a shader lives:
  - This data in the memory can be acce-ssed through the use of attributes, in/out variables and uniform variables



Shader types and roles

- Vertex (VS): handles the processing of individual vertices
- Geometry (GS): governs the processing of Primitives. Typical tasks: layered rendering or initial computations on data buffers
- Tesselation
  - Control (TCS): determines how much tessellation to do (it can also adjust the actual patch data, as well as feed additional patch data to later stages)
  - Evaluation (TES): takes the tessellated patch and computes the vertex values for each generated vertex
- Fragment (FS): similar to a Vertex Shader, but is used for calculating individual fragment colors. This is where lighting and bump-mapping effects are performed
- Compute (CS): anything!

Shader types and roles

- The defined shaders are compilied and then chained together into a Shader Program which can be executed on the GPU
- The shaders receive input data from a data buffer on card (VAO, explained later) through a process of attribute binding, allowing us to perform the needed computations to provide us with the desired results

Variables types

- Simple: bool, int, uint, float, double
- Matrices (3x3 and 4x4)
- Vectors (size 3 and 4)
- Example:

```
vec3   pos = vec3(vPos, 0.0);
float  angle = radians(rotation[0]);
vec3   axis = normalize(rotation.yzw);
mat3   I = mat3(1.0);
mat3   S = mat3(     0, -axis.z, axis.y,
                 axis.z,      0, -axis.x,
                -axis.y, axis.x,      0);
```

- Interface block (similar to structs)
- Data structures have automatic access labels like fields in a struct. The access convention is called swizzling
- x, y, z, or w, referring to the first, second, third, and fourth components
- there are 3 additional sets of swizzle masks. You can use x y z w, r g b a (for colors), or s t p q (for texture coordinates)
- Example:

```
vec2 someVec;
vec4 otherVec = someVec.xyxx;
vec3 thirdVec = otherVec.zyy;
```

Variables precision

- High (`highp`)
- Medium (`mediump`)
- Low (`lowp`)
  - Hopefully you won’t have to set them – their typical use relates to a technical issue with early GLSL versions: *In vertex shaders, if you do not explicitly set the default precision for floating-point types, it defaults to highp. However, if the fragment shader were to default to highp as well, that would cause issues since OpenGL ES 2.0 does not require support for high precision floating-point types in the fragment shader stage.*

Variable storage qualifiers

- **Uniform:** they do not change from one execution of a shader program to the next within a particular rendering call
- **Input/Attributes:** local CG data available to the shader of the given stage such as vertex coordinates, color, normal vector, etc.
- **Output/Varying:** provide an interface between Vertex and Fragment Shader. Vertex Shaders compute values per vertex and fragment shaders compute values per fragment. If you define a varying variable in a vertex shader, its value will be interpolated (perspective-correct) over the primitive being rendered and you can access the interpolated value in the fragment shader.
- `attribute` and `varying` might be deprecated in certain OpenGL versions. Instead, you simply use `in` and `out`

Vertex attributes

These canonical attributes are automatically made available:

- gl_Vertex Position (vec4)
- gl_Normal Normal (vec4)
- gl_Color Primary color of vertex (vec4)
- gl_MultiTexCoord0 Texture coordinate of texture unit 0 (vec4)
- gl_MultiTexCoord1 Texture coordinate of texture unit 1 (vec4)
- gl_MultiTexCoord2 Texture coordinate of texture unit 2 (vec4)
- gl_MultiTexCoord3 Texture coordinate of texture unit 3 (vec4)
- gl_MultiTexCoord4 Texture coordinate of texture unit 4 (vec4)
- gl_MultiTexCoord5 Texture coordinate of texture unit 5 (vec4)
- gl_MultiTexCoord6 Texture coordinate of texture unit 6 (vec4)
- gl_MultiTexCoord7 Texture coordinate of texture unit 7 (vec4)
- gl_FogCoord Fog Coord (float)

(Strange) Example: using color to do vertex specific shifting:

```
glBegin(GL_TRIANGLES)
   glVertex3f(0.0f, 0.0f, 0.0f);
   glColor3f(0.1,0.0,0.0);
   glVertex3f(1.0f, 0.0f, 0.0f);
   glColor3f(0.0,0.1,0.0);
   glVertex3f(1.0f, 1.0f, 0.0f);
   glColor3f(0.1,0.1,0.0);
glEnd();
```

Vertex Shader Source Code

```
void main(void)
{
   vec4 a = gl_Vertex + gl_Color;
   gl_Position =
gl_ModelViewProjectionMatrix * a;
}
```

Fragment Shader Source Code

```
void main (void)
{
   gl_FragColor =
vec4(0.0,0.0,1.0,1.0);
}
```

Layout qualifier

- Can precede variables and definitions
- Layout qualifiers affect where the storage for a variable comes from
- The most common ones: in and out to define input and output variables
- `layout(qualifier1, qualifier2 = value, ...) variable definition`

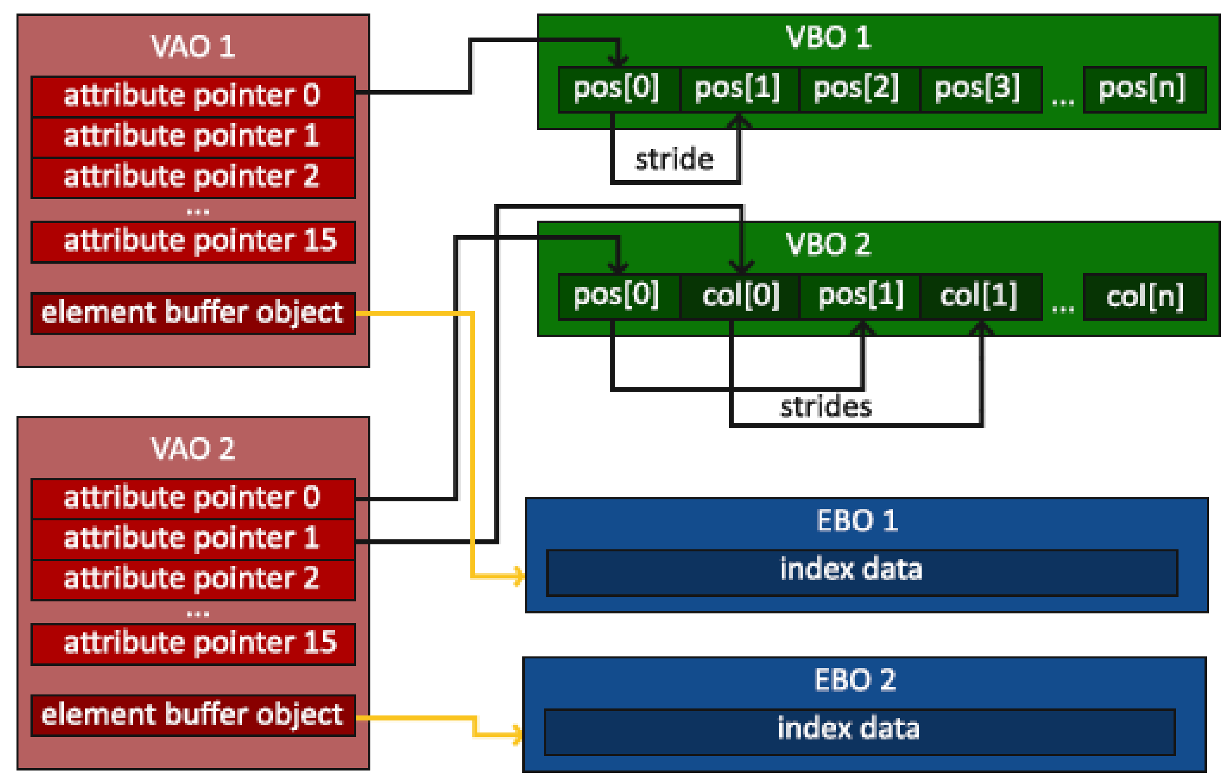
Layout qualifier, pointing to the shader where particular data is, replaced explicity binding of attributes at OpenGL level such as :
`glBindAttribLocation(shaderprogram, 0, "in_Position");`

Shaders in py-opengl

- Remark: it seems that the shaders defined in python do not need the initial section – instead of giving them variable names, just use the canonical names directly such as `gl_Vertex`, `gl_MultiTexCoord0`, etc.

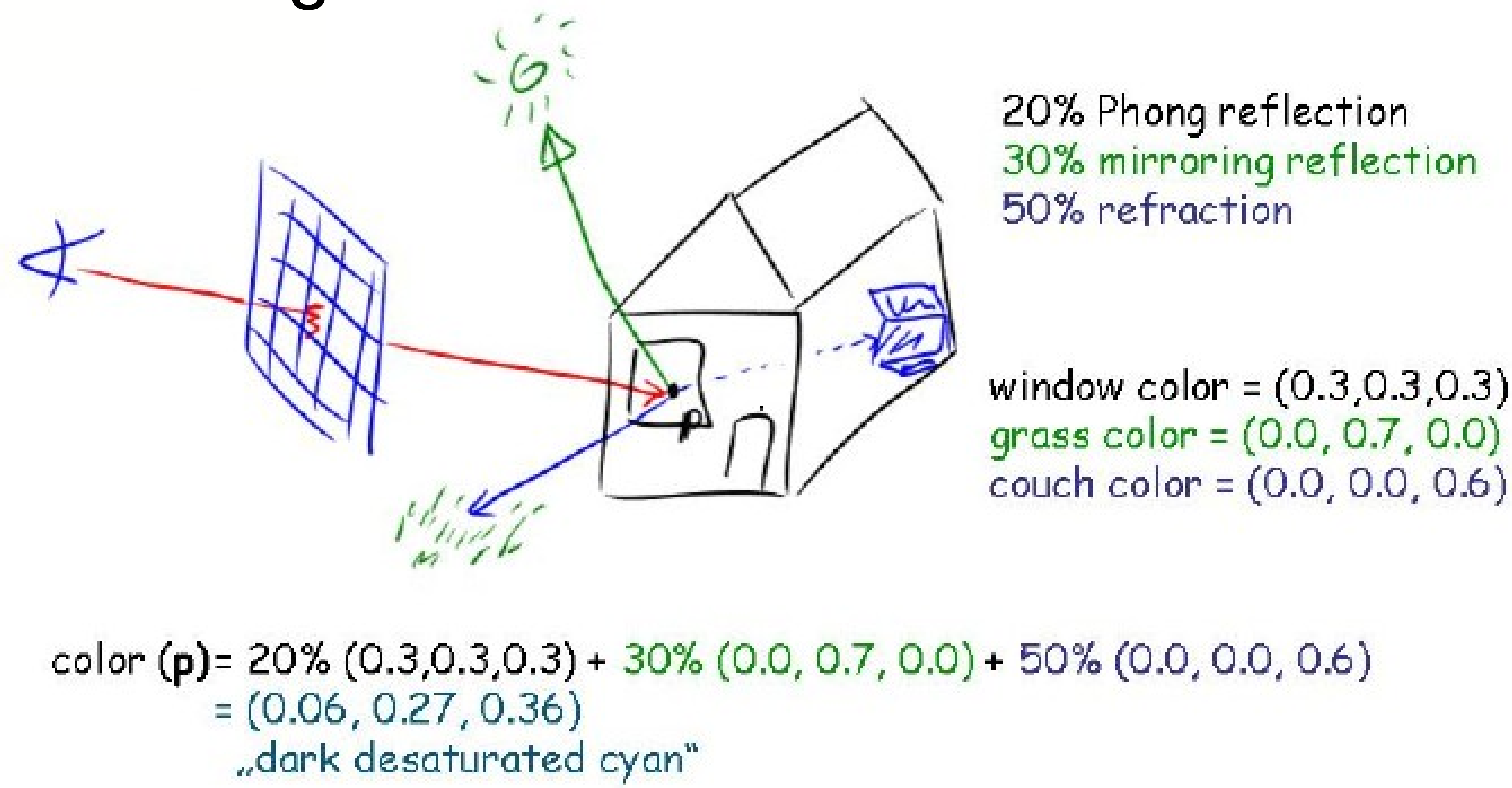
VAO and VBO

- A **Vertex Buffer Object** (VBO) is a memory buffer in the high speed memory of your video card designed to hold information about vertices such as the coordinates of vertices and the color associated with each vertex. VBOs can also store information such as normals, texcoords, indicies, etc
- A **Vertex Array Object** (VAO) is an object which contains one or more Vertex Buffer Objects and is designed to store the information for a complete rendered object.



What to do in a shader?

- Example: color mixing



Libraries

- GLEW performs important tasts for using the modern pipeline
- When you initialize it, it will check your platform and graphic card at run-time to know what functionality can be used in your program. Functions pointers will be set appropriately = context setting
  - Alternatives: GLAD

Libraries

- GLUT, despite being obsolete, can still be used for the “old style” high level operations like window management
- Alternative: GLFW is an Open Source, multi-platform library for OpenGL, OpenGL ES and Vulkan development on the desktop. It provides a simple API for creating windows, contexts and surfaces, receiving input and events
- Simple DirectMedia Layer (SDL) is a cross-platform development library designed to provide low level access to audio, keyboard, mouse, joystick, and graphics hardware
- Pygame for python

Thank you!

- Questions?