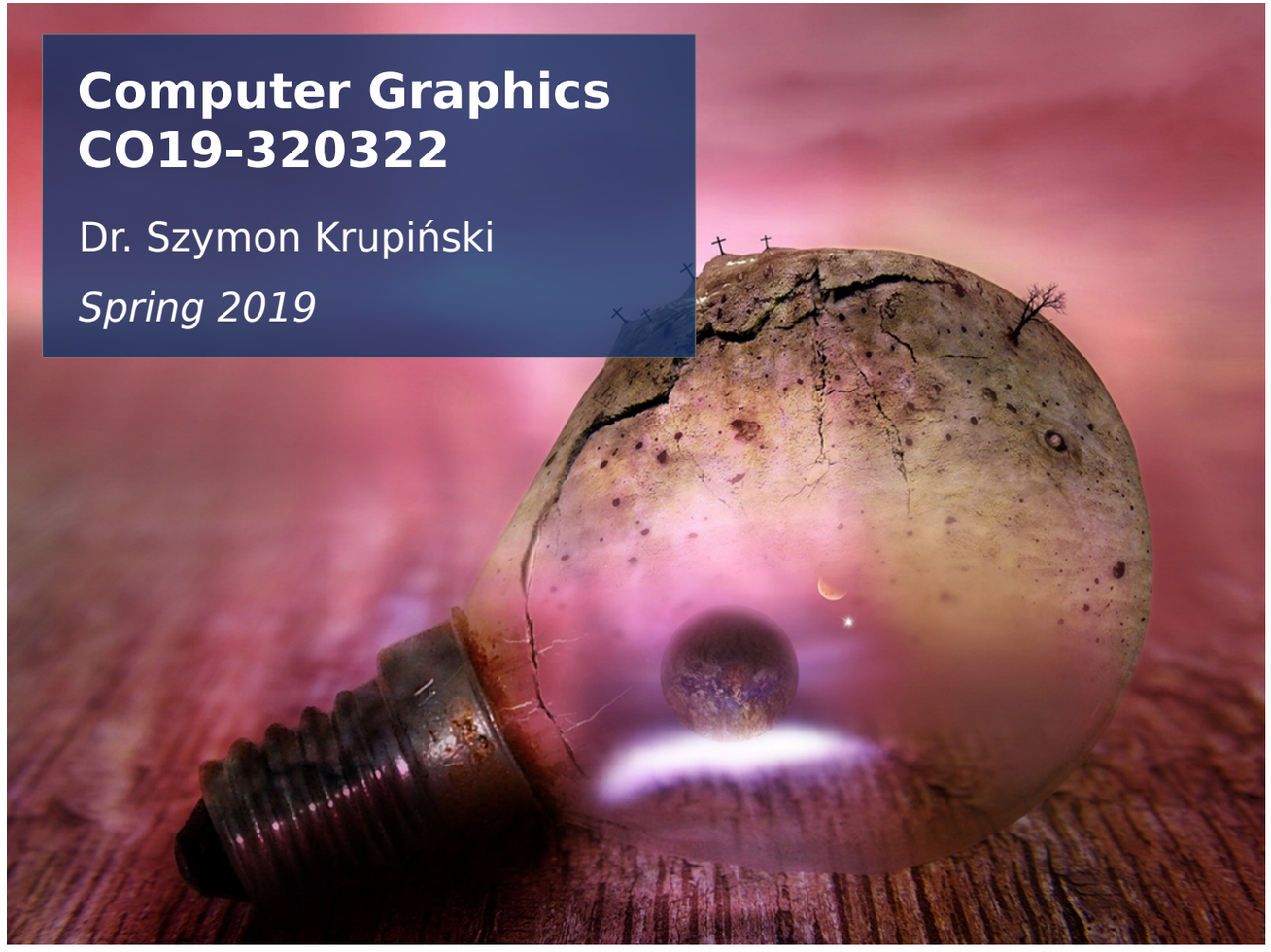
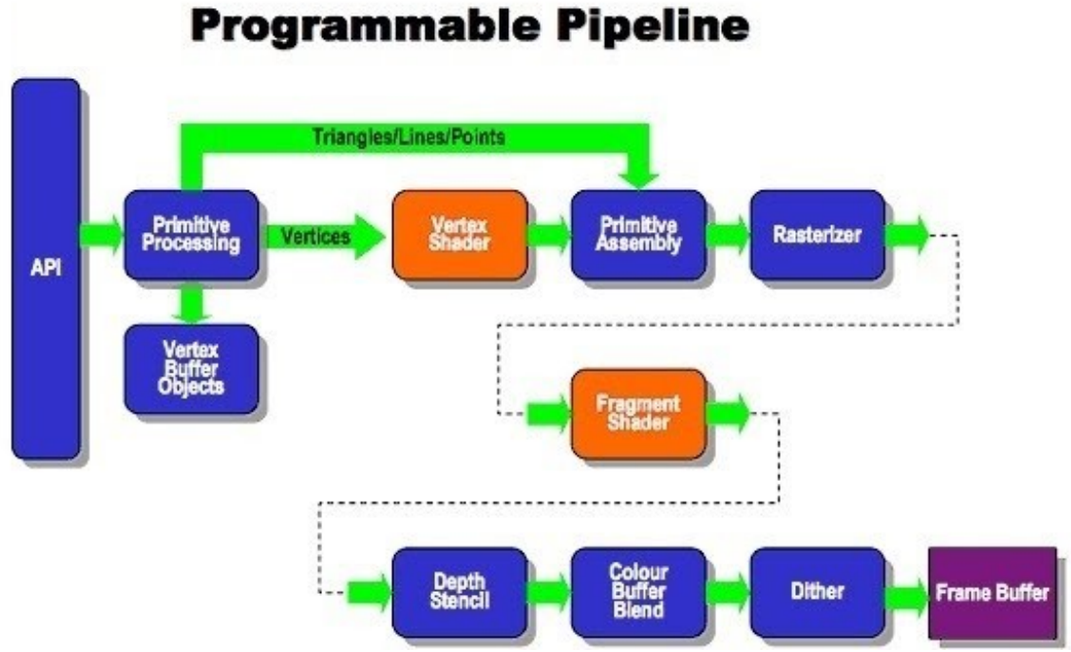
Lecture 18: Shading 2

JACOBS UNIVERSITY



Programmable pipeline

- Shaders are programs written to run on the GPU with specific tasks in mind
  - Coordinate and normal transformation (vertex shaders)
  - Assigning color values to samples (fragment shaders)
- Other types of shaders exist
- GLSL provides a number of pre-defined variables to use and to assign, corresponding to what the context of the shader is during execution
  - Attributes
  - Uniform variables
  - In/out variables / blocks (user-controlled)



Simplest GLSL Shaders

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

```
#version 120
    attribute vec3 aPos;
    void main()
    {
       gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);
    }
```

Just rewrite the user-provided coordinates for use of further stages of the CG pipeline

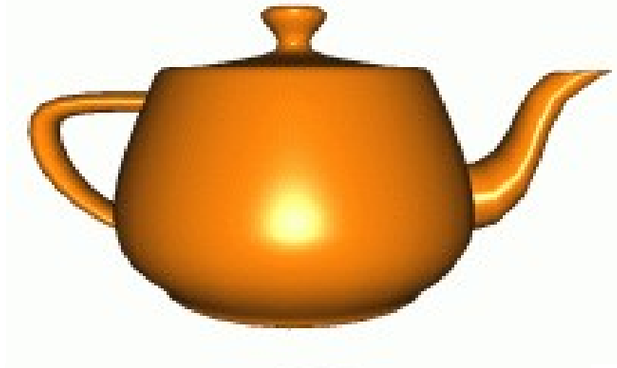
- Fragment shader

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

Assign a constant color and alpha to every sample

Shaders – use cases

- Toon, wireframe and alternative shading
- Tuning of the light reflection model
- Applying a procedural texture or transforming texture coordinates
- Making fancy geometry out of simple models

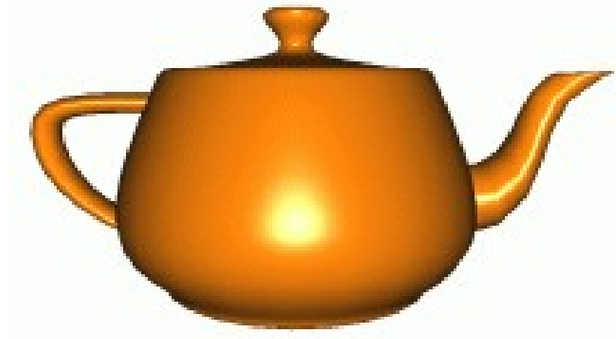


Good tutorials in introductions:

https://www.khronos.org/opengl/wiki/Core_Language_(GLSL)
https://www.lighthouse3d.com/tutorials/glsl-12-tutorial/shader-examples/

Shaders

- A number of built-in functions is available (here: not exclusive)



```
abs(genType α)
sign(α)
floor(α)
ceil(α)
mod(α, float β)
min(α, float β)
max(α, float β)
clamp(α, float β, float δ)
mix(α, float β, float δ)
step(float limit, α)
smoothstep(float α0, float α1, β)

float length(genType α)
float distance(genType α genType β)
float dot(genType α, genType β)
vec3 cross(genType α, genType β)
genType normalize(genType α)
vec4 ftransform()
genType normalize(genType α)
genType faceforward(genType N,
                    genType I, genType Nref)
```

```
bvec lessThan(vec α, vec β)
bvec lessThan(ivec α, ivec β)
bvec lessThanEqual(vec α, vec β)
bvec lessThanEqual(ivec α, ivec β)
bvec greaterThan(vec α, vec β)
bvec greaterThan(ivec α, ivec β)
bvec greaterThanEqual(vec α, vec β)
bvec greaterThanEqual(ivec α, ivec β)
bvec equal(vec α, vec β)
bvec equal(ivec α, ivec β)
bvec equal(bvec α, bvec β)
bvec notEqual(vec α, vec β)
bvec notEqual(ivec α, ivec β)
bvec notEqual(bvec α, bvec β)

bool any(bvec α)
bool all(bvec α)
bvec not(bvec α)

mat MatrixCompMult(mat x, mat y)
```

Binding and communicating values

- The shaders operate on the VAO/VBO

```
/* Create handles for our Vertex Array Object and two Vertex Buffer Objects */
GLuint vao, vbo[2];

const GLfloat diamond[4][2] = {
{  0.0,  1.0  }, /* Top point */
{  1.0,  0.0  }, /* Right point */
{  0.0, -1.0  }, /* Bottom point */
{ -1.0,  0.0  } }; /* Left point */

const GLfloat colors[4][3] = {
{  1.0,  0.0,  0.0  }, /* Red */
{  0.0,  1.0,  0.0  }, /* Green */
{  0.0,  0.0,  1.0  }, /* Blue */
{  1.0,  1.0,  1.0  } }; /* White */

/* Allocate and assign a Vertex Array Object to our handle */
glGenVertexArrays(1, &vao);

/* Bind our Vertex Array Object as the current used object */
glBindVertexArray(vao);

/* Allocate and assign two Vertex Buffer Objects to our handle */
glGenBuffers(2, vbo);
```

This is the code on the OpenGL program's side!

Binding and communicating values

```
/* Bind our first VBO as being the active buffer and storing vertex attributes (coordinates) */
glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);/* Copy the vertex data from diamond to our buffer */

/* 8 * sizeof(GLfloat) is the size of the diamond array, since it contains 8 GLfloat values */
glBufferData(GL_ARRAY_BUFFER, 8 * sizeof(GLfloat), diamond, GL_STATIC_DRAW);

/* Specify that our coordinate data is going into attribute index 0, and contains two floats per vertex */
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, 0);

/* Enable attribute index 0 as being used */
glEnableVertexAttribArray(0);

/* Bind our second VBO as being the active buffer and storing vertex attributes (colors) */
glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);

/* Copy the color data from colors to our buffer */
/* 12 * sizeof(GLfloat) is the size of the colors array, since it contains 12 GLfloat values */
glBufferData(GL_ARRAY_BUFFER, 12 * sizeof(GLfloat), colors, GL_STATIC_DRAW);

/* Specify that our color data is going into attribute index 1, and contains three floats per vertex */
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, 0);

/* Enable attribute index 1 as being used */
glEnableVertexAttribArray(1);
```

Binding and communicating values

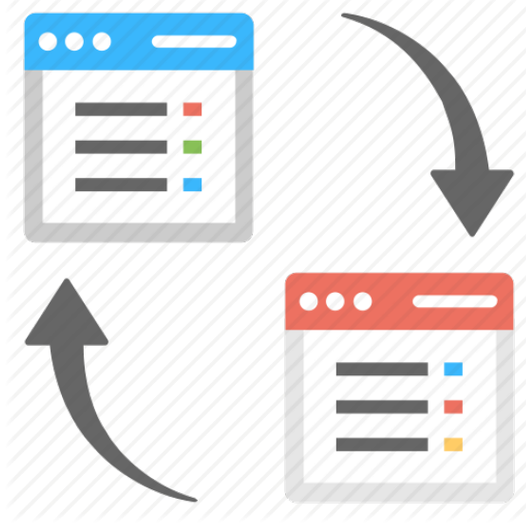
```
/* Bind attribute index 0 (coordinates) to in_Position and attribute index 1 (color) to in_Color */
/* Attribute locations must be setup before calling glLinkProgram. */
    glBindAttribLocation(shaderprogram, 0, "in_Position");
    glBindAttribLocation(shaderprogram, 1, "in_Color");
```

- The VAO/VBO data boud this way becomes accessible through the `in_Position` and `in_Color` attributes during every call
  - Specific vertex data for vertex shading
  - Interpolated values for sample shading
- `layout(location = XX)` is another (newer) way to point out to the shader program where the necessary variable is stored in the memory block of this shader
- Data can be passed to the shaders further down the pipeline by assigning it to the `varying` variables
- `uniform` variables cannot be modified

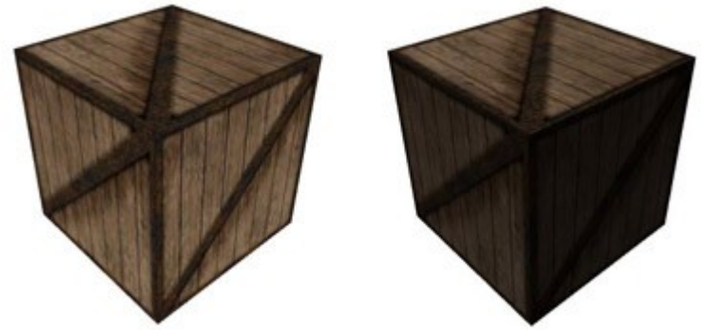
Layout qualifier is not available in early versions of OpenGL

Binding and communicating values

- Why did the situation become more complicated with the new style of pipeline programming?
- Just imagine: your OpenGL program and your shaders are separate programs running on separate CPUs
  - that’s why the code overhead must be there to assure they can exchange data smoothly and timely



Textures



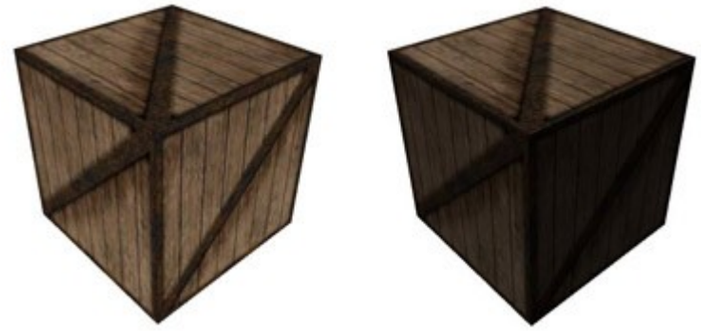
- On the OpenGL side:

```
GLint texUnitLoc = glGetUniformLocation(p, "texUnit");

/* Setup the variable texUnit to texture unit 0 */

glProgramUniform1i(p, texUnitLoc , 0);

/* Binding the texture to a texture unit is necessary as usual. Using the arbitrary
handle textureID as the texture name: */

GLuint textureID;

glGenTextures(GLuint textureID);

glActiveTexture(GL_TEXTURE0);

glBindTexture(GL_TEXTURE_2D, textureID);
```

Textures

- On the GLSL side:

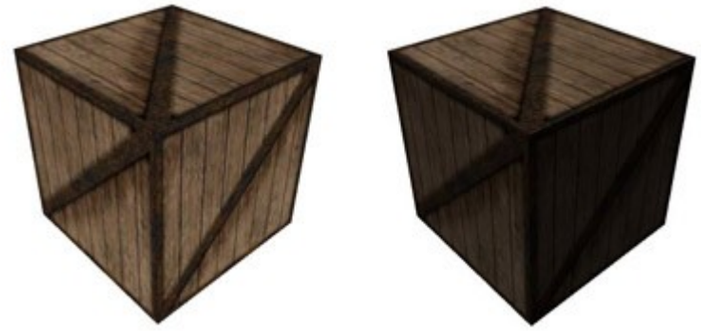
This code calculates the reflection model (using Material and Light information from outside and applies texture at the same time



```
#version 330

layout (std140) uniform Material {
    vec4 diffuse;
    vec4 ambient;
    vec4 specular;
    float shininess;
};

layout (std140) uniform Lights {
    vec3 l_dir;    // camera space
};

in Data {
    vec3 normal;
    vec4 eye;
    vec2 texCoord;
} DataIn;

uniform sampler2D texUnit;

out vec4 colorOut;
```

```
void main() {

    // set the specular term to black
    vec4 spec = vec4(0.0);

    // normalize both input vectors
    vec3 n = normalize(DataIn.normal);
    vec3 e = normalize(vec3(DataIn.eye));

    float intensity = max(dot(n,l_dir), 0.0);

    // if the vertex is lit compute the specular color
    if (intensity > 0.0) {
        // compute the half angle vector
        vec3 h = normalize(l_dir + e);
        // compute the specular term into spec
        float intSpec = max(dot(h,n), 0.0);
        spec = specular * pow(intSpec,shininess);
    }
    vec4 texColor = texture(texUnit, DataIn.texCoord);
    vec4 diffColor = intensity *  diffuse * texColor;
    vec4 ambColor = ambient * texColor;

    colorOut = max(diffColor + spec, ambColor);
}
```

Textures



- Remarks:
  - Attention to normal vectors and how they undergo projection:

    http://www.lighthouse3d.com/tutorials/glsl-12-tutorial/the-normal-matrix/
  - This is where we can use the level of detail logic to tune the shading:

```
void main()
{
    vec2 res = textureQueryLod(texUnit, VertexIn.texCoord.xy);

    if (res.x == 0)
        ...
    else if (res.x < 1)
        ...
}
```

  - One must not forget to pass the relevant data in the vertex shader

```
void main () {

    DataOut.normal = normalize(m_normal * normal);
    DataOut.eye = -(m_viewModel * position);
    DataOut.texCoord = texCoord;

    gl_Position = m_pvm * position;
}
```

Check the link above to see why the normal vector information is treated differently than the vertex coordinates

Texture / (u,v) coordinates provided by the user

Running the shaders

- The use of shaders starts when the user sets up everything correctly and calls the `glDraw()` function
  - It can be also done through use of `glDrawArrays()` or `glCallList()`
- The latter one uses the functionality of **display lists**:

Display list allow to compile a list of OpenGL primitives and call them to be displayed with one instruction.

```
// create one display list
GLuint index = glGenLists(1);
// compile the display list, store a triangle in it
glNewList(index, GL_COMPILE);
    glBegin(GL_TRIANGLES);
    glVertex3fv(v0);
    glVertex3fv(v1);
    glVertex3fv(v2);
    glEnd();
glEndList();
// draw the display list
glCallList(index);
// delete it if it is not used any more
glDeleteLists(index, 1);
```

Understanding GLSL

… will take time.

- Good resources:
  - http://www.lighthouse3d.com/tutorials/glsl-tutorial/
  - https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language
  - https://learnopengl.com/Advanced-OpenGL/Advanced-GLSL

Thank you!

- Questions?