COSC342: OpenGL Lab 2

Materials, Model loading, Render to textures

Objectives

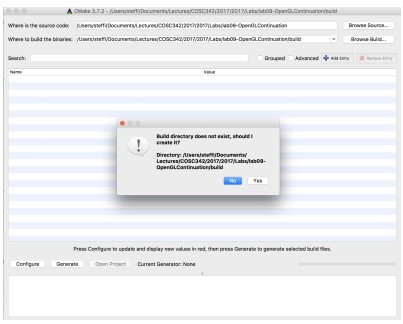
- Load a model file and render a textured model.
- Implement basic shading using the Phong reflection model.
- Render a scene into a texture

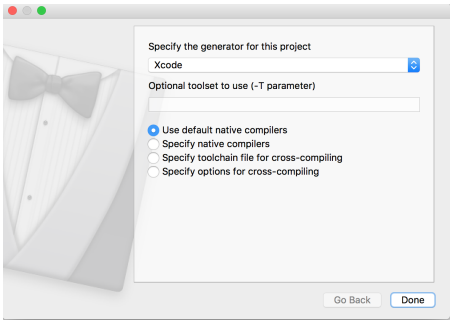
Introduction

In the last lab, we got started with OpenGL, we created a first output window and rendered primitives, such as triangles, quads and cubes. We also started to use textures. In this lab, we will go further and load more complex model files and apply the Phong shading model.

Make sure that you copy the source files from the `/home/cshome/coursework/342/pickup/labs/lab09-OpenGLContinuation` directory into a directory within your home directory. Otherwise you may not be able to compile the code. The process of generating the project build files is the same as for the last lab using CMake.

The first step is to create the development project files. Open CMake.app from the Applications folder. Put the source code folder location for lab09-OpenGLContinuation into source directory and add a location to where CMake should build the project files (e.g. lab09-OpenGLContinuation/build). Click **Configure** and allow to create a new directory if the directory did not exist before.





Select the IDE you want to use, e.g. XCode and wait until configuring is finished. CMake will show settings and path relevant to project (we will leave them to the default values). Press **Generate** to create the project files.

In the following we will explain how to compile the project with XCode, but you can also go to the Terminal and use make to compile the project. However, please note that now the project files are getting more complex with multiple applications and several header and source files to edit during the lab, so XCode is recommended.

XCode Use Finder navigate to the build directory (e.g /build). Open XCode project file: `COSC_Lab_OpenGL2.xcodeproj`. In XCode click on "project navigator" and navigate to Part04. Have a look at the source files. To compile and execute the code, go to "Product" and "Scheme" and select Part04 as scheme in the list. Then go to "Product" and click on "Build". Finally run the app by selecting "Product" - "Run".

MAKE (advanced) Change into the `lab09-OpenGLContinuation/build` directory and type "`make`" to build the code. Part04, Part05 and Part06 require shaders and textures from the corresponding subfolders, so you need to navigate to the subfolder (e.g. Part04) and call *../build/Part*04 from there. If you want to avoid that, you can also use the shell scripts within the build directory (e.g. launch-Part04.sh. This script sets the correct working directories automatically. To run the shell script use "`./launch-Part04.sh`".

Part 04: Model Loading

Go to the source code for Part04. The first part of the application will load a model (stored in suzanne.obj) and put all the corresponding indices, vertices, texture coordinates and normals into std::vectors for later usage. For loading models we use an external library called assimp. Have a look into the function loadAssImp in SimpleObjectLoader.cpp and see how the assimp reads the data from the obj file.

```
std::vector<unsigned short> indices;
std::vector<glm::vec3> indexed_vertices;
std::vector<glm::vec2> indexed_uvs;
std::vector<glm::vec3> indexed_normals;
// Read our .obj file
bool res = loadAssImp("suzanne.obj", indices, indexed_vertices,
  indexed_uvs, indexed_normals);
```

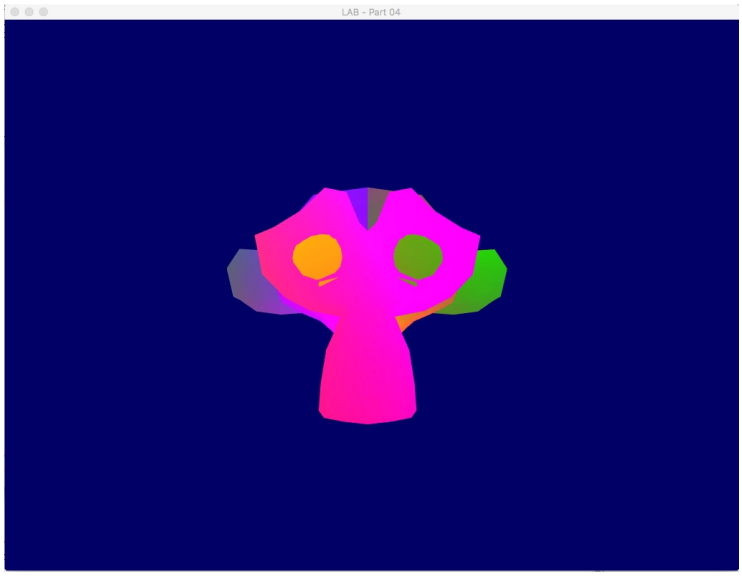
Similar like creating our cube geometry in the last lab, we create a geometry and set the vertices, the uv coordinates and the normals using the respective methods.

```
  //create geometry
Mesh* myGeom = new Mesh();
myGeom->setVertices(indexed_vertices);
myGeom->setUVs(indexed_uvs);
myGeom->setNormals(indexed_normals);
```

We also want to texture this model so the last step is to create again a TextureShader, load a texture, pass it to the shader and finally attach the shader to our geometry.

```
TextureShader* shader = new TextureShader( "textureShader");
Texture* texture = new Texture("uvmap.DDS");
shader->setTexture(texture);
myGeom->setShader(shader);
myScene->addObject(myGeom);
```

Run the application and the results should look like this:



In the render loop we will then again call the render method on our scene.

```
//Render loop
while( glfwGetKey(window, GLFW_KEY_ESCAPE ) != GLFW_PRESS &&
  glfwWindowShouldClose(window) == 0 ){
  // Clear the screen
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Also clear
  the depth buffer!!!
  // update camera controls with mouse input
  myControls->update();
  myScene->render(myCamera);
  // Swap buffers
  glfwSwapBuffers(window);
  glfwPollEvents();
}
```

You will notice a difference compared to last lab. We now also call an update method on controls. The controls receive mouse and keyboard input and allow us to navigate within the scene. You can use the arrow keys for moving around and the mouse input for changing the orientation of the camera.

Exercise

- Change the fragment shader (textureShader.frag) in such a way that it outputs the texture coordinates as colour values instead of the colour of the texture (Red showing UV.x and green showing UV.y).

- Create a simple animation by adding 0.01 units to x coordinate of the position of the model in each render step (hint using the method setTranslate).

Part 5: Basic shading

So far we did not take lighting and illumination into account. In this part of the lab, we will integrate diffuse, ambient and specular lighting for implementing the Phong reflection model. We will compute the illumination on a per-fragment basis to implement the Phong shading. If you look into Part05/basicShading.cpp, you will see that there is a lot of similarity to the last examples, but there is also a new Shader used, the BasicMaterialShader.

```
Geometry* myGeom = new Geometry();
myGeom->setVertices(indexed_vertices);
myGeom->setUVs(indexed_uvs);
myGeom->setNormals(indexed_normals);
myGeom->setIndices(indices);
BasicMaterialShader* shader = new BasicMaterialShader("
  basicMaterialShader");
Texture* texture = new Texture("uvmap.DDS");
shader->setTexture(texture);
myGeom->setShader(shader);
myScene->addObject(myGeom);
```

The rest of the code is basically the same. So let's have a look into the BasicMaterialShader class.

```
private:
        Texture* m_texture;
        GLuint m_lightPosID;
```

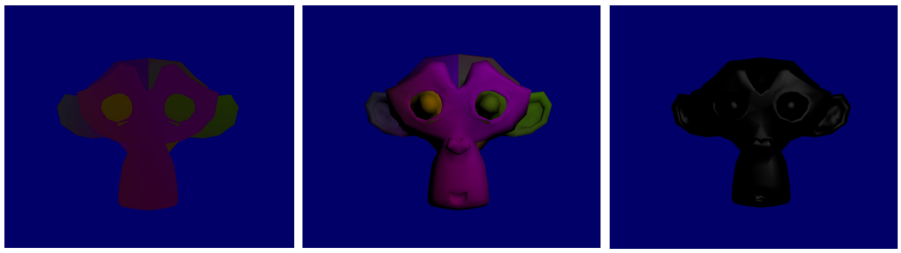
Here we have additional member variables describing a light source position and the location of a uniform variable that will be used to pass the light's position to the shader. This will be set up in the init method of the shader using glGetUniformLocation.

```
void init(){
        glUseProgram(programID);
        m_lightPos = glm::vec3(4,4,4);
        m_lightPosID = glGetUniformLocation(programID, "
    lightPosWorldspace");
        glUniform3f(m_lightPosID, m_lightPos.x, m_lightPos
    .z);
        m_TextureID  = glGetUniformLocation(programID, "myTextureSampler
    ");
}
```

We use the vertex shader and the fragment shader (basicMaterialShader.vert and basicMaterialShader.frag) for computing the illumination. At first, we will have a look into the vertex shader. If we compare it to the previous vertex shader, we see that there is much more computation happening. In addition to the texture coordinates and the vertex position in clip space, the normal of the vertex, the eye direction, and the light direction in camera space will be passed to the fragment shader. For computation, we use the model matrix (M), the view matrix (V) and the model-view-projection matrix (MVP) to transform into camera space and clip space.

```
// Output position of the vertex, in clip space : MVP * position
gl_Position =  MVP * vec4(vertexPosition_modelspace,1);

// Position of the vertex, in worldspace : M * position
posWorldspace = (M * vec4(vertexPosition_modelspace,1)).xyz;
// Vector that goes from the vertex to the camera, in camera space.
// In camera space, the camera is at the origin (0,0,0).
vec3 vertexPosCameraspace = ( V * M * vec4(vertexPosition_modelspace
  ,1)).xyz;
eyeDirectionCameraspace = vec3(0,0,0) - vertexPosCameraspace;

// Vector that goes from the vertex to the light, in camera space. M
  is ommited because it's identity.
vec3 lightPositionCameraspace = ( V * vec4(lightPosWorldspace,1)).xyz;
lightDirectionCameraspace = lightPositionCameraspace +
  eyeDirectionCameraspace;

// Normal of the the vertex, in camera space
normalCameraspace = ( V * M * vec4(vertexNormal_modelspace,0)).xyz;

// UV of the vertex. No special space for this one.
UV = vertexUV;
```

The fragment shader computes the Phong reflection model (explained in the lecture) describing the reflection of light from a surface as a combination of diffuse reflection, ambient reflection and specular reflection.



Ambient Diffuse Specular

Diffuse Component

The diffuse component describes all the light that is reflected in all directions when light hits a surface. The reflected light depends on the angle θ between the incoming light ray (L) and the surface normal (N), as well as on the diffuse material colour (kd), the diffuse texture map and the colour of the light (Id). When computing the angle between the light direction and the normal, we have to clamp the angle to 0, to avoid light that comes from behind the surface.

```
// Cosine of the angle between the normal n and the light direction l,
// clamped above 0
float cosTheta = clamp( dot( N,L ), 0.0,1.0 );
vec3 diffuseComponent = diffuseLightColor* diffuseMatColor *
  textureVal * cosTheta;
```

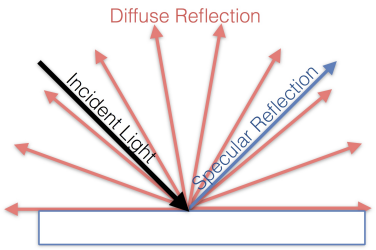
Ambient Component

The ambient component of the phong reflection model defines a minimum brightness and prevents the surface from being completely black. It creates an effect that let the surface simply emit light. We multiply the color value from the texture as well.

```
vec3 ambientComponent = ambientLightColor*ambientMatColor*textureVal;
```

Specular Component

The specular component is used to describe the behaviour of shiny surfaces. In contrast to the diffuse component where light is reflected in all directions, the specular component describes the amount of light that is reflected from a surface in the direction of the reflection vector. According to the law of reflection, the reflection vector has the same angle to the surface normal as the incident ray. An example for an ideal reflection surface is a mirror reflecting all light in the direction of reflection vector.

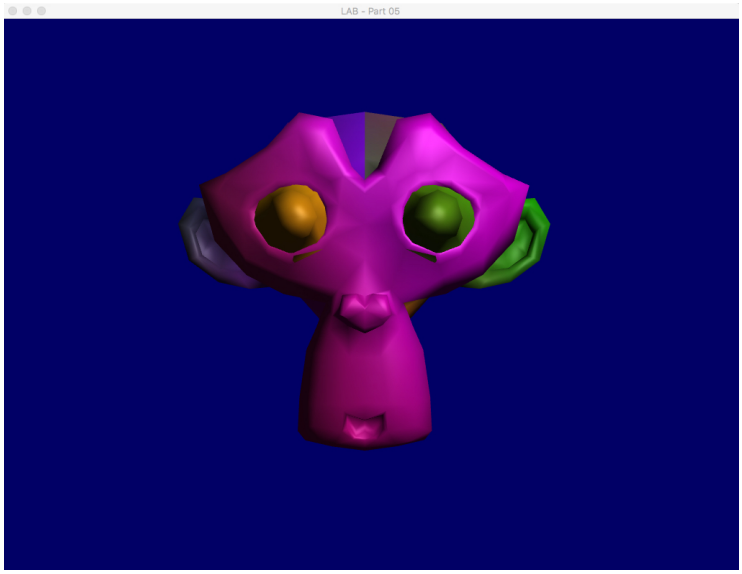


To compute the specular component we use the angle between the eye vector and the reflected ray to create the effect that for small angles between the reflected ray and the eye direction the specular component will be larger. In addition we use the specular exponent to adjust the focus of the specular highlight. A high exponent creates a concentrated highlight. Values of ns range between 0-1000.

```
1  // Eye vector (towards the camera)
   vec3 E = normalize(eyeDirectionCameraspace);
3  // Direction in which the triangle reflects the light
   vec3 R = reflect(-L,N);
5  // Cosine of the angle between the Eye vector and the Reflect vector,
   // clamped to 0
7  float cosAlpha = clamp( dot( E,R ), 0,1 );
   vec3 specularComponent = specularLightColor * specularMatColor  * pow(
     cosAlpha,ns);
9
```

Result

Finally, the combination of all three components will create the following output:




```
color =
// Ambient : simulates indirect lighting
ambientComponent +
// Diffuse : "color" of the object
diffuseComponent +
// Specular : reflective highlight, like a mirror
specularComponent;
```

Exercise

- Create a setLightColour method in BasicMaterialShader that sets a new diffuse light colour using glm::vec3. Add a new uniform to the fragment shader to pass that light colour and use it.
- Use glfwGetKey in the rendering loop to capture keyboard input of 1,2,3 and create three different light colours and switch between different light colours using keyboard input.

Part 6: Render to Texture (RTT)

The basic idea of RTT is to render the scene into a texture and to apply a specific effect on this texture afterwards. This allows us to create different post-processing effects on a rendered scene (e.g. image filters, blur or edges), but also for implementing shadows mapping. In Part06, we will use RTT to create a time-dependent dynamic glass effect to our rendering. Have a look in to `Part06/renderToTexture.cpp`.

RTT consists of 3 main steps:

1. The creation of a texture to render to (the render target),
2. Rendering something into the texture.
3. Using the created texture and apply a certain post-processing step to it.

Creating the Render Target

The OpenGL rendering pipeline uses geometry data and textures to render output as 2D pixels to the screen. The final output destination for the render output is specified framebuffer. By default, OpenGL renders to a framebuffer that is set up by the window system. For RTT, we want to create a specific framebuffer to capture the rendering output for later usage. For this purpose we use the method $glGenFramebuffers$.

```
GLuint FramebufferName = 0;
glGenFramebuffers(1, &FramebufferName);
glBindFramebuffer(GL_FRAMEBUFFER, FramebufferName);
```

In order to access the content of the famebuffer, we need to attach a texture. For this purpose, we create a texture object and use the method $glFramebufferTexture$ that attaches the texture object as a buffer to the currently bound framebuffer object.

```
// The texture we're going to render to
GLuint renderedTexture;
glGenTextures(1, &renderedTexture);
// Set "renderedTexture" as our colour attachement #0
glFramebufferTexture(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
  renderedTexture, 0);
```

Render to the texture

For rendering our scene into the textures, we use $glBindFramebuffer$ to bind our framebuffer and then render just as in the previous examples using the scene's render function.

```
// Render to our framebuffer
glBindFramebuffer(GL_FRAMEBUFFER, FramebufferName);
...
myScene->render(myCamera);
```

Using the render texture

In order to apply the post-processing steps to our render texture, we will render it to the screen using a simple quad and our postprocessing shaders.

```
    // the quad that we use to render the framebuffer texture to the
    screen
    Quad* outputQuad = new Quad();
PostProcessingShader* passThroughShader = new PostProcessingShader("
    Passthrough.vert", "PostEffect.frag" );

```

For the actual rendering to screen, we need to set the output framebuffer back to default:

```
// Render to the screen
glBindFramebuffer(GL_FRAMEBUFFER, 0);

```

And render the quad with the render texture to the screen using the postprocessing shaders (Passthrough.vert and PostEffect.frag).

```
// Use our shader
postEffectShader->bind();
// Bind our texture in Texture Unit 0
//the one from the famebuffer = render texture the one used in the
    shader texid
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, renderedTexture);
// Set our "renderedTexture" sampler to user Texture Unit 0
postEffectShader->bindTexture();
postEffectShader->setTime((float)(glfwGetTime()*10.0f)); //set time to
    get animated effect in shader
outputQuad->directRender(); //call render directly to render quad only
    without transformations

```

To create the dynamic glass effect, we use the time in the fragment shader and offset the pixels depending on the time.

```
    color = texture( renderedTexture, UV + 0.005*vec2( sin(time+1024.0*
    UV.x),cos(time+768.0*UV.y)) ).xyz;
```

Exercise

- Change the PostEffectShader.frag so that it creates a box blur effect (using a 9x9 window) as output:

1/81

1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1

9x9 box filter

The result should like this:

- Change the PostEffectShader.frag so that the post effect extracts the edges of the rendered scene using the Sobel operator: $S = \sqrt{S_x^2 + S_y^2}$

-1	0	1
-2	0	2
-1	0	1

S_x

1	2	1
0	0	0
-1	-2	-1

S_y

The result should like this: