PyOpenGL workshop progression

Over my graphics module at university, I completed 7 workshops over 7 weeks, where I progressively Implemented new features to a scene that was given to me. These workshops were assessed as coursework, so while not all the code in this project is mine, the features mentioned in this document were created solely by me.

An easy way to get the code running in python is to create an anaconda environment:

```
> conda create --name <yourNameHere> python=3.5
> source activate <yourNameHere>
> conda install -c anaconda pyopengl
> conda install -c cogsci pygame
```

The list of things implemented by me are:

- Implementation of a prospective projection.
- A moveable camera.
- Method to calculate normals from a given mesh.
- Generation of a sphere model.
- Implementation of Gourad, Phong and Blinn shading.
- Rendering of a cube map and sampling for reflection texture.
- Rendering of a shadow map and shader to implement said map.

The final project code after all workshops can be found at:
https://github.com/Reubunbun/UniOfExeterGraphicsWorkshops

While the rest of the code outside these implementations was not created by me, after working with it for so long I built a very solid understanding of how everything works. The rest of this document shows specific code that was implemented by me if you do not wish to search through the project files:

Implementation of a prospective projection

In matutils.py, a frustrum matrix function

```
def frustumMatrix(l,r,t,b,n,f):
    '''
    Returns perspective projection matrix
    :param l: left clip plane
    :param r: right clip plane
    :param t: top clip plane
    :param b: bottom clip plane
    :param n: near clip plane
    :param f: far clip plane
    :return: A 4x4 frustrum projection matrix
    '''
    return np.array(
        [
            [ 2*n/(r-l),    0,            (r+l)/(r-l),    0           ],
            [ 0,            -2*n/(t-b),   (t+b)/(t-b),    0           ],
            [ 0,            0,            -(f+n)/(f-n),   -2*f*n/(f-n) ],
            [ 0,            0,            -1,             0           ]
        ]
    )
```

In scene.py, in the constructor for, use the method to create the projection matrix variable. This along with the view (camera) matrix is used for each model matrix to bind shaders.

```
# Use the frustrum matrix
self.P = frustumMatrix(left,right,top,bottom,near,far)
```

Implementation of a movable camera (view matrix)

In camera.py

```python
# import a bunch of useful matrix functions (for translation, scaling etc)
from matutils import *


class Camera:
    '''
    Base class for handling the camera.
    TODO WS2: Implement this class to allow moving the mouse
    '''

    def __init__(self):
        self.V = np.identity(4)      # Start with the identity matrix.
        self.phi = 0.                # Azimuth angle (around the y axis)
        self.psi = 0.                # Zenith angle (around the x axis)
        self.distance = 10.          # Distance of the camera to the centre point
        self.center = [0., 0., 0.]   # Position of the centre
        self.update()                # Calculate the view matrix

    def update(self):
        '''
        Function to update the camera view matrix from parameters.
        Starts by setting the point the camera looks at as the center of the coordinate system.
        This coordinate system is then moved and rotated by changing this class' attributes.
        '''
        # Calculate the translation matrix for the view center (the point we look at)
        T0 = translationMatrix(self.center)

        # Calculate the rotation matrix from the angles phi and psi angles.
        R = np.matmul( rotationMatrixX(self.psi), rotationMatrixY(self.phi) )

        # Calculate translation for the camera distance to the center point.
        T = translationMatrix( [0., 0., -self.distance] )

        # Finally we calculate the view matrix by combining the three matrices.
        self.V = np.matmul( np.matmul(T, R), T0 )
```

In scene.py, in the pygameEvents method, changing the Camera attributes when mouse events are detected

```
# Controls for moving the camera
elif event.type == pygame.MOUSEMOTION:
    if pygame.mouse.get_pressed()[0]:
        if self.mouse_mvt is not None:
            self.mouse_mvt = pygame.mouse.get_rel()
            #TODO: WS2
            self.camera.center[0] += ( float(self.mouse_mvt[0]) / self.window_size[0] )
            self.camera.center[1] -= ( float(self.mouse_mvt[1]) / self.window_size[1] )
        else:
            self.mouse_mvt = pygame.mouse.get_rel()

    elif pygame.mouse.get_pressed()[2]:
        if self.mouse_mvt is not None:
            self.mouse_mvt = pygame.mouse.get_rel()
            #TODO: WS2
            self.camera.phi -= ( float(self.mouse_mvt[0]) / self.window_size[0] )
            self.camera.psi -= ( float(self.mouse_mvt[1]) / self.window_size[1] )
        else:
            self.mouse_mvt = pygame.mouse.get_rel()
    else:
        self.mouse_mvt = None
```

Method to calculate normals from a given mesh

Note this method was later changed when textures were implemented (in mesh.py), this was my version:

```
    def calculate_normals(self):
        '''
        TODO WS3: calculate the correct normals
        Creates an array of normals and assigns them to self.normals
        1. calculate normal for each face using cross product
        2. set each vertex normal as the average of the normals over all faces it belongs to.
        '''

        self.normals = np.zeros( (self.vertices.shape[0], 3), dtype='f' )

        #For all the faces in the model.
        for i in range( self.faces.shape[0] ):
            # Calculate the normal of the face by taking the cross product of two of the triangles sides.
            a = self.vertices[ self.faces[i, 2] ] - self.vertices[ self.faces[i, 0] ]
            b = self.vertices[ self.faces[i, 1] ] - self.vertices[ self.faces[i, 0] ]
            faceNormal = np.cross(a, b)

            # Normalise.
            faceNormal /= np.linalg.norm(faceNormal)

            # Add the new normals for each of the verices on the current face.
            for j in range(3):
                self.normals[ self.faces[i, j] ] += faceNormal

        # Normalise the vectors.
        self.normals /= np.linalg.norm(self.normals, axis=1, keepdims=True)
```

Generation of a sphere model

In sphereModel.py, the constructor for the Sphere class

```
class Sphere(Mesh):
    def __init__(self, nvert=10, nhoriz=20):
        n = nvert * nhoriz + 2
        vertex_colors = np.zeros((n, 3), 'f')
        # texture coordinates
        textureCoords = np.zeros((n, 2), 'f')

        circles, vertices = self.generate_points(nvert, nhoriz)
        vertices = np.array(vertices, dtype='f')
        faces = self.generate_faces(circles)
        indices = np.array(faces, dtype=np.uint32)

        Mesh.__init__(self,
                      vertices=vertices,
                      faces=indices,
                      textureCoords=textureCoords,
                      material=Material(Ka=[0.5,0.5,0.5], Kd=[0.6,0.6,0.9], Ks=[1.,1.,0.9], Ns=15.0)
                      )
```

generate_points helper method

```
def generate_points(self, horizontal, vertical):
    '''
    Generates a list of vertices for a sphere of given number of horizontal and vertical points you wish to have.
    :param horizontal: How many horizontal segments the sphere will have
    :param vertical: How many vertical segments the sphere will have
    :return points: A list of all the vertices that the sphere will have
    :return circles: A list of the vertices separated by which vertical segment you are on
    '''

    currentPoint = [0, 1, 0]
    # A list that will contain separate lists of each circle as you move down the sphere
    circles = []
    # A list that will contain vertices of a circle at a given y coordinate
    yCircles = []
    for i in range(vertical - 1):
        # Rotate around the z axis for every vertical point over pi radians
        rotation = self.rotZ(np.pi / vertical)
        currentPoint = np.matmul(rotation, currentPoint)
        yCircles.append(currentPoint)
        for j in range(horizontal):
            # Rotate around the y axis for every horizontal point over 2 pi radians
            rotation = self.rotY(2 * np.pi / horizontal)
            currentPoint = np.matmul(rotation, currentPoint)
            # Don't append to the list if we are at the last point
            if j != horizontal - 1:
                yCircles.append(currentPoint)
        circles.append(yCircles)
        yCircles = []

    # The very top point of the cirlce
    points = [[0, 1, 0]]
    # Create the raw list of vertices
    for circle in circles:
        for point in circle:
            points.append(point)

    # Append the very bottom point of the circle
    circles.append([0, -1, 0])
    points.append([0, -1, 0])

    return circles, points
```

generate_faces helper method

```python
def generate_faces(self, points):
    '''
    Creates a list of all faces in a sphere, given we have the vertices
    :param points: A list of all the points on the sphere
    :return: A list of all the faces on the sphere
    '''

    faces = []
    for i in range(len(points)):
        width = len(points[0])
        # Calculate the index of the first point on the current circle
        currentCircle = (i - 1) * width
        # Calculate teh index of the first point on the next circle
        nextCircle = i * width
        # Create the faces for the top fan
        if i == 0:
            for j in range(1, len(points[0]) + 1):
                if j != len(points[0]):
                    faces.append([0, j, j + 1])
                else:
                    faces.append([0, j, 1])
        # Create the faces of the bottom fan
        elif i == len(points) - 1:
            for j in range(1, len(points[0]) + 1):
                if j != len(points[0]):
                    faces.append([currentCircle + j, nextCircle + 1, currentCircle + j + 1])
                else:
                    faces.append([currentCircle + j, nextCircle + 1, currentCircle + 1])
        # Create all other faces
        else:
            for j in range(1, len(points[i]) + 1):

                if j != len(points[i]):
                    faces.append([currentCircle + j, nextCircle + j, nextCircle + j + 1])
                    faces.append([currentCircle + j, nextCircle + j + 1, currentCircle + j + 1])
                else:
                    faces.append([currentCircle + j, nextCircle + j, nextCircle + 1])
                    faces.append([currentCircle + j, nextCircle + 1, currentCircle + 1])

    return faces
```

Helper methods for getting rotation matrices for rotating specific angles

```python
def rotZ(self, ang):
    '''
    :param ang: The angle to rotate by
    :return: The rotation matrix around the z axis
    '''

    rotation = np.identity(3, dtype='f')
    rotation[0][0] = np.cos(ang)
    rotation[0][1] = np.sin(ang)
    rotation[1][0] = -1 * np.sin(ang)
    rotation[1][1] = np.cos(ang)

    return rotation

def rotY(self, ang):
    '''
    :param ang: The angle to rotate by
    :return: The rotation matrix around the y axis
    '''
    rotation = np.identity(3, dtype='f')
    rotation[0][0] = np.cos(ang)
    rotation[0][2] = np.sin(ang)
    rotation[2][0] = -1 * np.sin(ang)
    rotation[2][2] = np.cos(ang)

    return rotation
```

Implementation of Gourad, Phong and Blinn shading

Gourad vertex_shader.glsl

```
#version 130

in vec3 position;
in vec3 normal;
in vec3 color;

out vec3 fragment_color;

uniform mat4 PVM;
uniform mat4 VM;
uniform mat3 VMiT;
uniform int mode;

uniform vec3 Ka;
uniform vec3 Kd;
uniform vec3 Ks;
uniform float Ns;

uniform vec3 light;
uniform vec3 Ia;
uniform vec3 Id;
uniform vec3 Is;


void main() {
  // transform the position using PVM matrix.
  gl_Position = PVM * vec4(position, 1.0f);

  // calculate vectors used for shading calculations
  vec3 position_view_space = vec3(VM*vec4(position,1.0f));
  vec3 normal_view_space = normalize(VMiT*normal);
  vec3 camera_direction = -normalize(position_view_space);
  vec3 light_direction = normalize(light-position_view_space);

  // calculate light components
  vec3 ambient = Ia*Ka;
  vec3 diffuse = Id*Kd*max(0.0f,dot(light_direction, normal_view_space));
  vec3 specular = Is*Ks*pow(max(0.0f, dot(reflect(light_direction, normal_view_space), -camera_direction)), Ns);

  // calculate the attenuation function
  float dist = length(light - position_view_space);
  float attenuation =  min(1.0/(dist*dist*0.005) + 1.0/(dist*0.05), 1.0);

  // combine the shading components
  fragment_color = ambient + attenuation*(diffuse + specular);
}
```

Gourad fragment_shader.glsl

```
# version 130

in vec3 fragment_color;

out vec3 final_color;

void main() {
  final_color = fragment_color;
}
```

Phong vertex_shader.glsl

```glsl
#version 130

in vec3 position;
in vec3 normal;
in vec3 color;

out vec3 fragment_color;
out vec3 position_view_space;
out vec3 normal_view_space;

uniform mat4 PVM;
uniform mat4 VM;
uniform mat3 VMiT;
uniform int mode;


void main() {
    // transform the position using PVM matrix.
    gl_Position = PVM * vec4(position, 1.0f);

    // calculate vectors used for shading calculations
    position_view_space = vec3(VM*vec4(position,1.0f));
    normal_view_space = normalize(VMiT*normal);

    // pass on the color from the data array
    fragment_color = color;
}
```

Phong fragment_shader.glsl

```glsl
# version 130

in vec3 fragment_color;
in vec3 position_view_space;
in vec3 normal_view_space;
out vec3 final_color;

uniform int mode;

// material uniforms
uniform vec3 Ka;
uniform vec3 Kd;
uniform vec3 Ks;
uniform float Ns;

// light source
uniform vec3 light;
uniform vec3 Ia;
uniform vec3 Id;
uniform vec3 Is;

void main() {
  // calculate vectors used for shading calculations
  vec3 camera_direction = -normalize(position_view_space);
  vec3 light_direction = normalize(light-position_view_space);

  // calculate light components
  vec3 ambient = Ia*Ka;
  vec3 diffuse = Id*Kd*max(0.0f,dot(light_direction, normal_view_space));
  vec3 specular = Is*Ks*pow(max(0.0f, dot(reflect(light_direction, normal_view_space), -camera_direction)), Ns);

  // calculate the attenuation function
  float dist = length(light - position_view_space);
  float attenuation =  min(1.0/(dist*dist*0.005) + 1.0/(dist*0.05), 1.0);

  // combine the shading components
  final_color = ambient + attenuation*(diffuse + specular);
}
```

Blinn vertex_shader.glsl

```glsl
#version 130

in vec3 position;
in vec3 normal;
in vec3 color;

out vec3 fragment_color;
out vec3 position_view_space;
out vec3 normal_view_space;

uniform mat4 PVM;
uniform mat4 VM;
uniform mat3 VMiT;
uniform int mode;


void main() {
  // transform the position using PVM matrix.
  gl_Position = PVM * vec4(position, 1.0f);

  // calculate vectors used for shading calculations
  position_view_space = vec3(VM*vec4(position,1.0f));
  normal_view_space = normalize(VMiT*normal);

  // pass on the color from the data array
  fragment_color = color;
}
```

Blinn fragment_shader.glsl

```glsl
# version 130

in vec3 fragment_color;
in vec3 position_view_space;
in vec3 normal_view_space;


out vec3 final_color;

uniform int mode;

// material uniforms
uniform vec3 Ka;
uniform vec3 Kd;
uniform vec3 Ks;
uniform float Ns;

// light source
uniform vec3 light;
uniform vec3 Ia;
uniform vec3 Id;
uniform vec3 Is;

void main() {
  // calculate vectors used for shading calculations
  vec3 camera_direction = -normalize(position_view_space);
  vec3 light_direction = normalize(light-position_view_space);
  vec3 halfway = normalize(light_direction+camera_direction);

  // calculate light components
  vec3 ambient = Ia*Ka;
  vec3 diffuse = Id*Kd*max(0.0f,dot(light_direction, normal_view_space));
  vec3 specular = Is*Ks*pow(max(0.0f, dot(halfway, normal_view_space)), 4*Ns);

  // calculate the attenuation function
  float dist = length(light - position_view_space);
  float attenuation =  min(1.0/(dist*dist*0.005) + 1.0/(dist*0.05), 1.0);

  // combine the shading components
  final_color = ambient + attenuation*(diffuse + specular);
}
```

In ecm3423_ws7.py, in the draw method, code for switching between shaders for the displayed model

```
def keyboard(self, event):
    '''
    Process additional keyboard events for this demo.
    '''
    Scene.keyboard(self, event)

    if event.key == pygame.K_1:
        print('--> using Flat shading without texture')
        self.modelToDraw.use_textures = False
        self.modelToDraw.bind_shader('flat')
    elif event.key == pygame.K_2:
        print('--> using gouraud')
        self.modelToDraw.use_textures = False
        self.modelToDraw.bind_shader('gouraud')
    elif event.key == pygame.K_3:
        print('--> using phong')
        self.modelToDraw.use_textures = False
        self.modelToDraw.bind_shader('phong')
    elif event.key == pygame.K_4:
        print('--> using blinn')
        self.modelToDraw.use_textures = False
        self.modelToDraw.bind_shader('blinn')
```

Rendering of a cube map and sampling for reflection texture

In cubeMap.py, an update method that renders each face of the cube map

```
def update(self, scene):

    near, far, left, right, top, bottom = (1.0, 10, -1.0, 1.0, -1.0, 1.0)
    store_cam = (scene.camera.center, scene.camera.psi, scene.camera.phi, scene.camera.distance)

    scene.P = frustumMatrix(left,right,top,bottom,near,far)
    scene.camera.V = np.identity
    scene.camera.distance = 0
    scene.camera.psi = 0
    scene.camera.phi = 0
    scene.camera.phi += (np.pi / 2)
    glViewport(0, 0, 512, 512)
    scene.camera.update()

    iter = 0
    for (key, value) in self.files.items():
        fbo = glGenFramebuffers(1)
        glBindFramebuffer(GL_FRAMEBUFFER, fbo)
        glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, key, self.textureid, 0)
        scene.draw_reflections()

        if iter == 0:
            scene.camera.phi -= np.pi
        elif iter == 1:
            scene.camera.phi += np.pi / 2
            scene.camera.psi -= np.pi / 2
        elif iter == 2:
            scene.camera.psi += np.pi
        elif iter == 3:
            scene.camera.psi -= np.pi / 2
        elif iter == 4:
            scene.camera.phi += np.pi
        iter += 1
        scene.camera.update()

    glBindFramebuffer(GL_FRAMEBUFFER, 0)

    scene.camera.center, scene.camera.psi, scene.camera.phi, scene.camera.distance = store_cam
    scene.camera.update()
    near, far = 1.5, 50

    glViewport(0, 0, scene.window_size[0], scene.window_size[1])
    scene.P = frustumMatrix(left,right,top,bottom,near,far)
```

In ecm3423_ws7.py, a method that renders what will be reflected

```
def draw_reflections(self):
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
    self.shadow_map.bind_texture()

    for model in self.meshes:
        model.draw()
    for model in self.table:
        model.draw()
    for model in self.box:
        model.draw()
```

In shaders.py, a class for the cubemap shader

```
class CubeShader(PhongShader):
    def __init__(self):
        PhongShader.__init__(self, name='cube')

        self.add_uniform('sampler_cube')
        self.add_uniform('VT')


    def bind(self, model, M):
        PhongShader.bind(self, model, M)
        glUseProgram(self.program)
        location = glGetUniformLocation(self.program, 'sampler_cube')
        glUniform1i(location, 0)

        V = model.scene.camera.V
        self.uniforms['VT'].bind(V.transpose())
```

Cube vertex_shader.glsl

```
#version 130

in vec3 position;
in vec3 normal;
in vec3 color;
in vec2 texCoord;

out vec3 fragment_color;
out vec3 position_view_space;
out vec2 fragment_texCoord;
out vec3 normal_view_space;

uniform mat4 PVM;
uniform mat4 VM;
uniform mat3 VMiT;
uniform int mode;

void main(){
    // transform the position using PVM matrix.
    gl_Position = PVM * vec4(position, 1.0f);

    // calculate vectors used for shading calculations
    position_view_space = vec3(VM*vec4(position, 1.0f));
    normal_view_space = normalize(VMiT*normal);

    // forward the texture coordinates.
    fragment_texCoord = texCoord;

    // pass on the color from the data array.
    fragment_color = color;
}
```

Cube fragment_shader.glsl

```glsl
#version 130

in vec3 fragment_texCoord;
in vec3 position_view_space;
in vec3 normal_view_space;

out vec4 final_color;

uniform samplerCube sampler_cube; // the cube map texture
uniform mat4 VT;

void main(void) {

  vec3 camera_direction = -normalize(position_view_space);          // incident vector from viewpoint
  vec3 reflect = reflect(camera_direction, normal_view_space);       // reflect around the normal
  vec3 world_coord = vec3(VT*vec4(reflect, 1.0f));                   // back to world coordinates
  vec3 flipped = vec3(-world_coord.x, world_coord.y, world_coord.z); // flip x values

  // sample from the cube map texture
  final_color = textureCube(sampler_cube, flipped);
}
```

Rendering of a shadow map and shader to implement said map

In showTexture.py, a ShadowMap class constructor

```python
class ShadowMap(Texture):

    def __init__(self, scene, width=2048, height=2048, wrap=GL_CLAMP_TO_EDGE, format=GL_DEPTH_COMPONENT, sample=GL_NEAREST, type=GL_FLOAT):
        self.width = width
        self.height = height
        self.target = GL_TEXTURE_2D

        glEnable(GL_DEPTH_TEST)

        # generate and bind texture
        self.textureid = glGenTextures(1)
        self.bind()

        # set texture
        glTexImage2D(self.target, 0, format, self.width, self.height, 0, format, type, None)

        # for texture coordinates outside [0, 1]
        glTexParameter(self.target, GL_TEXTURE_WRAP_S, wrap)
        glTexParameter(self.target, GL_TEXTURE_WRAP_T, wrap)
        # how sampling is done
        glTexParameter(self.target, GL_TEXTURE_MAG_FILTER, sample)
        glTexParameter(self.target, GL_TEXTURE_MIN_FILTER, sample)

        # update texture
        self.update(scene)

        # unbind
        self.unbind()
```

Methods in ShadowMap class, the first binds the shadowmap texture and the second updates the camera to generate the shadowmap

```python
def bind_texture(self):
    glActiveTexture(GL_TEXTURE1)
    self.bind()

def update(self, scene):

    # set viewport and matrix
    glViewport(0, 0, self.width, self.height)
    left, right, top, bottom, near, far = (-1.0, 1.0, -1.0, 1.0, 1.0, 20)
    scene.P = frustumMatrix(left, right, top, bottom, near, far)

    store_cam = (scene.camera.center, scene.camera.psi, scene.camera.phi, scene.camera.distance)
    scene.camera.V = lookAt(scene.light.position, np.array([0., 0., 0.], dtype='f'))

    # generate framebuffer
    fbo = glGenFramebuffers(1)
    glBindFramebuffer(GL_FRAMEBUFFER, fbo)
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, self.target, self.textureid, 0)
    # draw
    scene.depth_draw()
    # unbind
    glBindFramebuffer(GL_FRAMEBUFFER, 0)

    # reset matrix and viewport
    glViewport(0, 0, scene.window_size[0], scene.window_size[1])
    near, far = 1.5, 50
    scene.P = frustumMatrix(left, right, top, bottom, near, far)
    scene.camera.center, scene.camera.psi, scene.camera.phi, scene.camera.distance = store_cam
    scene.camera.update()
```

In ecm3423_ws7.py, a method that renders what will be included in the shadow map

```python
def depth_draw(self):
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
    for model in self.table:
        model.draw()
    for model in self.box:
        model.draw()
    for model in self.meshes:
        model.draw()
    self.modelToDraw.draw()
```

In shaders.py, a class for the shadow shader

```
class ShadowShader(PhongShader):
    def __init__(self, Vs):
        PhongShader.__init__(self, name='shadow')

        self.Vs = Vs
        self.add_uniform('sampler_shadow')
        self.add_uniform('Ps')
        self.add_uniform('Vs')
        self.add_uniform('Vic')
        self.add_uniform('x')

    def bind(self, model, M):
        PhongShader.bind(self, model, M)
        Vc = model.scene.camera.V
        Ps = frustumMatrix(-1.0, 1.0, -1.0, 1.0, 1.0, 20)
        x = np.array([
                [0.5, 0.,  0.,  0.5],
                [0.,  0.5, 0.,  0.5],
                [0.,  0.,  0.5, 0.5],
                [0.,  0.,  0.,   1.]
            ], dtype='f')
        self.uniforms['Ps'].bind(Ps)
        self.uniforms['Vs'].bind(self.Vs)
        self.uniforms['Vic'].bind(np.linalg.inv(Vc))
        self.uniforms['x'].bind(x)

        glUseProgram(self.program)
        location = glGetUniformLocation(self.program, 'sampler_shadow')
        glUniform1i(location, 1)
```

In scene.py, pygameEvents method, code for moving the light source and updating the shadow+cube maps

```
# Controls for moving the light source
elif event.type == pygame.MOUSEBUTTONDOWN:
    mods = pygame.key.get_mods()
    if event.button == 4:
        if mods & pygame.KMOD_CTRL:
            self.light.position *= 1.05
            self.light.update()
            self.shadow_map.update(self)
            self.cube.update(self)
        else:
            self.camera.distance = max(1, self.camera.distance - 1)

    elif event.button == 5:
        if mods & pygame.KMOD_CTRL:
            self.light.position *= 0.95
            self.light.update()
            self.shadow_map.update(self)
            self.cube.update(self)
        else:
            self.camera.distance += 1
```

Shadow vertex_shader.glsl

```
#version 130

in vec3 position;

in vec3 color;
in vec2 texCoord;

out vec3 fragment_color;
out vec3 position_view_space;
out vec2 fragment_texCoord;
out mat4 S;

uniform mat4 PVM;
uniform mat4 VM;
uniform mat3 VMiT;
uniform int mode;

uniform mat4 Ps;
uniform mat4 Vs;
uniform mat4 Vic;
uniform mat4 x;

void main(){
  gl_Position = PVM * vec4(position, 1.0f);
  position_view_space = vec3(VM*vec4(position, 1.0f));
  fragment_texCoord = texCoord;
  fragment_color = color;

  S = x * Ps * Vs * Vic;
}
```

Shadow fragment_shader.glsl variables

```
#version 130

in vec3 position_view_space;
in vec2 fragment_texCoord;
in mat4 S;

out vec4 final_color;

uniform int mode;

uniform int has_texture;

uniform sampler2D textureObject;
uniform sampler2D sampler_shadow;

uniform vec3 Ka;
uniform vec3 Kd;
uniform vec3 Ks;
uniform float Ns;

uniform vec3 light;
uniform vec3 Ia;
uniform vec3 Id;
uniform vec3 Is;
```

Shadow fragment_shader.glsl main function

```glsl
void main() {
  // Calculate vectors used for shading calculations
  vec3 camera_direction = -normalize(position_view_space);
  vec3 light_direction = normalize(light-position_view_space);

  // Calculate the normal to the fragment using position of its neighbours
  vec3 xTangent = dFdx( position_view_space );
  vec3 yTangent = dFdy( position_view_space );
  vec3 normal_view_space = normalize( cross( xTangent, yTangent ) );

  // calculate light components
  vec4 ambient = vec4(Ia*Ka,1.0f);
  vec4 diffuse = vec4(Id*Kd*max(0.0f,dot(light_direction, normal_view_space)),1.0f);
  vec4 specular = vec4(Is*Ks*pow(max(0.0f, dot(reflect(light_direction, normal_view_space), -camera_direction)), Ns), 1.0f);

  // calculate the attenuation function
  float dist = length(light - position_view_space);
  float attenuation =  min(1.0/(dist*dist*0.005) + 1.0/(dist*0.05), 1.0);

  vec4 texval = vec4(1.0f);

  if (has_texture == 1){
    texval = texture2D(textureObject, fragment_texCoord);
  }

  // Store the tempory value for the final colour
  vec4 temp = texval*ambient + attenuation*(texval*diffuse + specular);

  // Convert coordinates to light space
  vec4 position_light_space = S * vec4(position_view_space, 1.0f);

  // If the last coordinate is negative, we can ignore this point and render in the usual way as its behind the rendered view.
  if (position_light_space.w >= 0){

    // Divide by the fourth coordinate to remove the homogenous coordinate. Each coordinate is moved very slightly to improve appearance.
    vec3 position = vec3((position_light_space.x/position_light_space.w)-0.001f, (position_light_space.y/position_light_space.w)-0.001f, (position_light_space.z/position_light_space.w)-0.001f);

    // If the texture depth is smaller than position's z value, then it is occluded and we should use ambient lighting. Otherwise render the usual way.
    if (texture(sampler_shadow, position.xy).z < position.z){
      final_color = texval * ambient;
    } else {
      final_color = temp;
    }

  } else {
    final_color = temp;
  }

}
```