Department of Computer Science
Technical University of Cluj-Napoca

Basketball Simulation Application in OpenGL

Graphic Processing, Laboratory activity, 2021-2022

Name : Beáta Keresztes
Group: 30432
Email: keresztesbeata00@gmail.com

Contents

1 Subject specification **4**

2 Scenario **5**

2.1 Scene and object description 5

2.2 Functionalities 10

3 Implementation details **12**

3.1 Functions and special algorithms 12

3.1.1 Camera 12

3.1.2 Animation 14

3.1.3 Light source 16

3.1.3.0.1 Day cycle animation 17

4 Data structures **18**

5 Class hierarchy **19**

6 Graphical user interface **20**

6.1 View wireframe surfaces 20

6.2 Controlling the camera movement 21

6.3 Controlling the ball animation 22

6.4 Controlling the light sources 22

6.4.1 Control the day cycle animation: 22

6.4.2 Control the ambient light intensity: 22

6.4.3 Select the current shader: 22

6.4.4 Select the light source and control its movement (in case of the multiple point lights: 23

7 Conclusions and further developments **24**

8 References **25**

Chapter 1

Subject specification

The aim was to create a photo-realistic scene of 3D objects using the OpenGL Library, and to allow the user to inspect the scene and interact with the objects in it, using keyboard input or the mouse.
The main objectives targeted by this project were:

- Visualize the scene: scaling, translation, rotation, camera movement, animations
- Viewing solid, wireframe objects, polygonal and smooth surfaces
- Specification of at least two different light sources
- Texture mapping an materials
- Shadow computation
- Animation of objects
- Photo-realism, detailed modeling, scene complexity, algorithm development and implementation

The theme of the project is an application for a Basketball Game Simulation, which combines these objectives in order to build a photo-realistic environment, and provides the possibility for the user to view the created, move in the scene, as well as interact with the ball and the light sources that are present.

Chapter 2

Scenario

2.1 Scene and object description

The scene consists of a basketball court located outside, in a small field, in a natural environment. In order to build the scene and arrange, as well as edit the objects in it, the Blender application was used.

Figure 2.1: The basketball court scene

The scene is composed of many objects, arranged together using the Blender editor as well as scaling and rotation transformations applied to these objects after loading them in the application. The objects are texture mapped in order to create a more realistic view. Each object comes with its own material (.mtl) file, which describes the materials applied to each part of the given object, it's colour and light properties, and includes a reference to the texture image. The main objects composing the scene are:

- Basketball field: a simple plane, on which there is mapped a basketball court's image with the necesssary markings indicating the position of the player on the field.
- Two Basketball stands (goal) with a hoop: which represents the target that the player tries to hit when throwing the ball.
- Fence surrounding the court: which delimits the are of movement of the player.

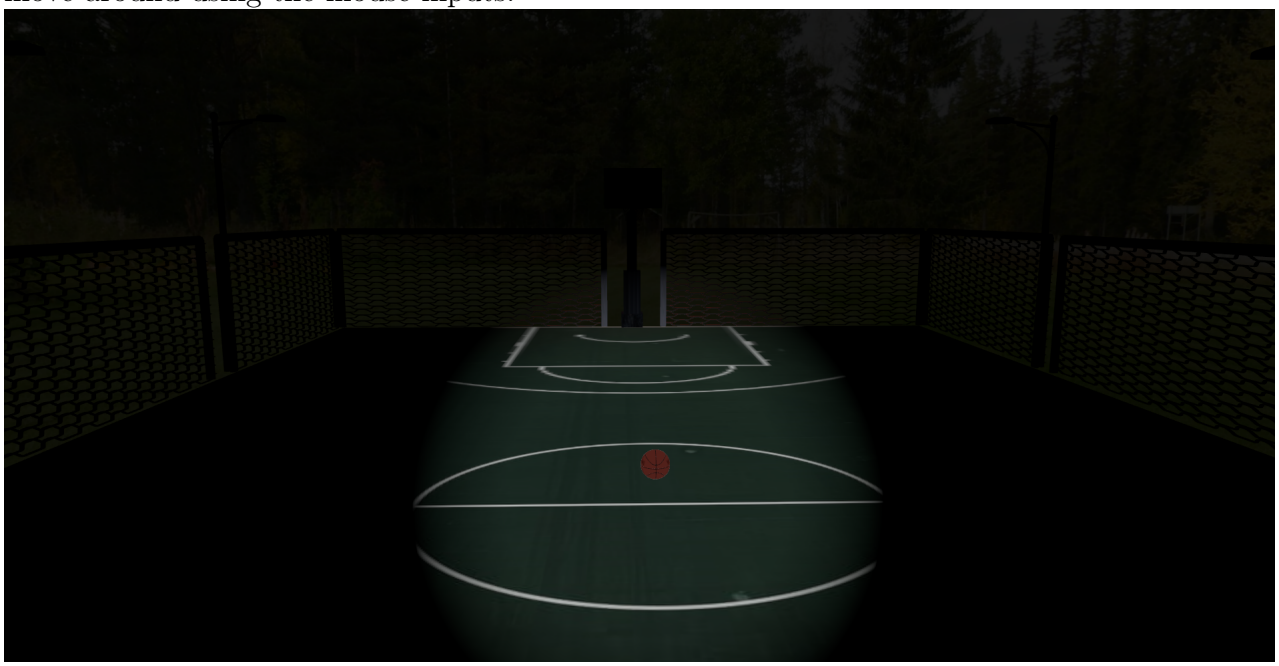
In addition, a skybox was added for a more realistic view, which gives the impression of being in a larger scene, an open field, with an extended field of view.
The skybox as well as the objects can be downloaded for free from the following sources .

To complete the scene, some light sources were added, as well as shadows to make the scene even more vivid. The lights sources can be changed by applying different filter on the scene:

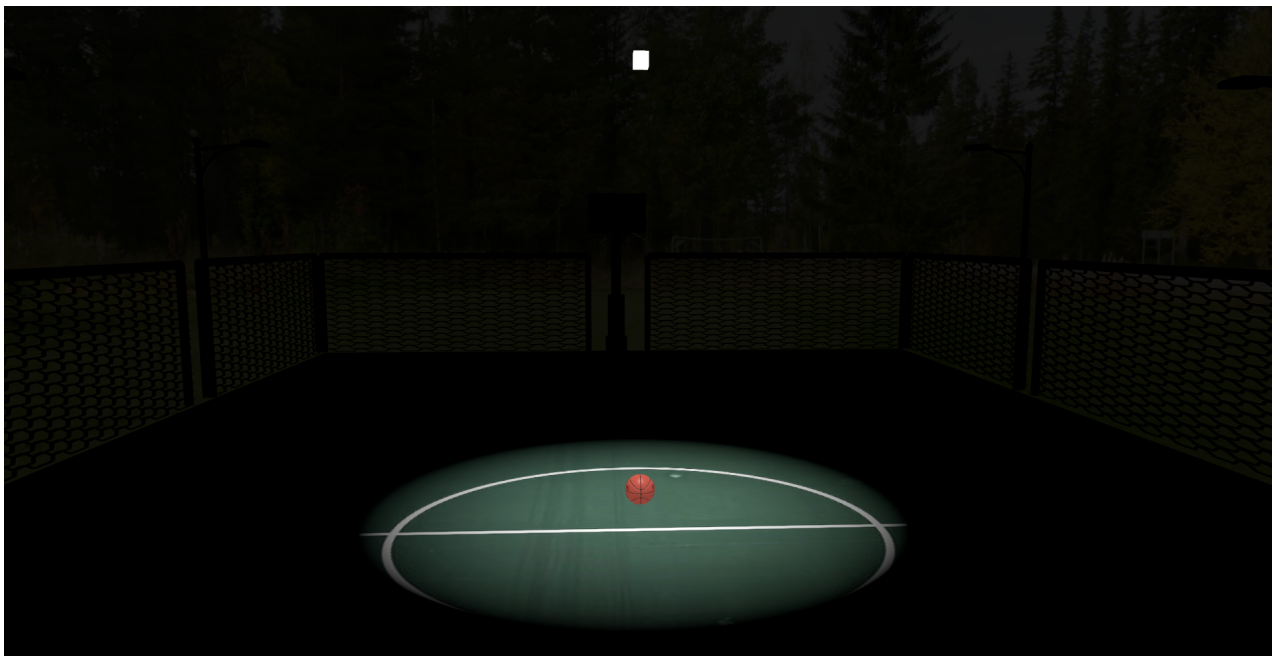
- Basic lighting: A directional light source with shadow mapping and a string ambient light intensity, which represents the day-light.

- Night lighting: Multiple spotlights mapped to some lamps and reflectors which illuminate the scene while the ambient component of the background's (skybox) lighting is very low to create the impression of darkness.

- Flashlight: A spotlight which is initially directed towards the basketball, and it can be move around using the mouse inputs.



- Spotlight: A stronger spotlight directed towards the ball, and it can also be moved around by the user using the same mouse inputs.



- Point lights: Three pointlights placed above the scene, which illustrate the way the point lights position influences which objects are illuminated in opposition with the directional light, where the light hits each object from the same direction, so each of the objects is illuminated uniformly.

For the directional light, shadows are also included, to illustrate the interaction between the objects in the scene and the given light source.

The light and shadow computations are performed in the fragment shader:

```
float computeShadow() {
// perform perspective divide
vec3 normalizedCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;

// Transform to [0,1] range
normalizedCoords = normalizedCoords * 0.5 + 0.5;

// Get closest depth value from light's perspective
float closestDepth = texture(shadowMap, normalizedCoords.xy).r;

// Get depth of current fragment from light's perspective
float currentDepth = normalizedCoords.z;

    if (currentDepth > 1.0f)
return 0.0f;

    float bias = max(0.05 * (1.0 - dot(fNormal, lightDir)), 0.005);

    // Check whether current frag pos is in shadow
float shadow = currentDepth - bias > closestDepth  ? 1.0 : 0.0;

return shadow;
}

void main()
{
    computeDirLight();

    float shadow = computeShadow();

    //compute final vertex color
```

```
    vec3 color = min((ambient + (1.0 - shadow) * diffuse) *
    texture(diffuseTexture, fTexCoords).rgb + (1.0 - shadow) * specular *
    texture(specularTexture, fTexCoords).rgb, 1.0f);

    fColor = vec4(color, 1.0f);
}
```

For the spotlight and flashlight the cut-off angle was considered to limit the range of visibility, the objects that fall into the light/shadow:

```
vec3 computeSpotLight(vec3 lightPosition, vec3 spotLightTarget, vec3 lightColor) {
//compute eye space coordinates
    vec4 fPosEye = view * model * vec4(fPosition, 1.0f);
vec4 lightPositionEye = view * model * vec4(lightPosition, 1.0f);
vec4 spotLightTargetEye = view * model * vec4(spotLightTarget, 1.0f);

 //normalize light direction
    vec4 lightDirN = normalize(lightPositionEye - fPosEye);

    float theta = dot(lightDirN, normalize(lightPositionEye - spotLightTargetEye));

vec3 color = computePointLight(lightPosition, lightColor);

 if(theta > cutOffAngle) {
        return color;
    }
else{
float epsilon = cutOffAngle - outerCone;
float intensity = (theta - outerCone)/epsilon;
return intensity * color;
}
}
```

2.2 Functionalities

The user is allowed to traverse the basketball field and view the scene, as if he/she was inside the court. He/She can interact with the ball object, and trigger certain animations which involve moving the ball, and it is also allowed to change the lighting filters presented above, as well as the position of certain light sources.

- Move around the scene with a camera
- Change the ambient light's intensity
- Change the light filters and move the light sources
- Start/Stop day cycle animation

- Play with the ball: pick it up, drop it, bounce, dribble or throw it in a given direction.
- View wireframe surfaces

Chapter 3

Implementation details

Starting from the core application provided in the laboratory, which involves loading a 3d model and basic lighting with a directional light source, some more functionalities were added to extend it.

3.1 Functions and special algorithms

3.1.1 Camera

In order to control the camera's movement, the methods provided by the Camera class were implemented so that the camera is able to move in all 6 directions: UP, DOWN, LEFT, RIGHT, FORWARD, BACKWARD, as well as it is possible to rotate the camera with certain pitch and yaw angles, as well roll the camera around the z axis (front direction). When moving the camera it was important to check that the camera remained within the boundaries of the basketball court, the player cannot leave the basketball field.

```
void Camera::move(MOVE_DIRECTION direction) {

    switch (direction) {
    case MOVE_LEFT: {
        this->cameraPosition -= this->cameraRightDirection * cameraSpeed;
        break;
    }
    case MOVE_RIGHT: {
        this->cameraPosition += this->cameraRightDirection * cameraSpeed;
        break;
    }
    case MOVE_FORWARD: {
        this->cameraPosition += this->cameraFrontDirection * cameraSpeed;
        break;
    }
    case MOVE_BACKWARD: {
        this->cameraPosition -= this->cameraFrontDirection * cameraSpeed;
```

```
            break;
        }
        case MOVE_UP: {
            this->cameraPosition += this->cameraUpDirection * cameraSpeed;
            break;
        }
        case MOVE_DOWN: {
            this->cameraPosition -= this->cameraUpDirection * cameraSpeed;
            break;
        }
        default: break;
        }

        checkBoundaries();
    }

    void Camera::checkBoundaries() {
        if (this->cameraPosition.y < 0.1f) {
            this->cameraPosition.y = 0.1;
        }
        if (this->cameraPosition.z > 55) {
            this->cameraPosition.z = 55;
        }
        else if (this->cameraPosition.z < -55) {
            this->cameraPosition.z = -55;
        }
        if (this->cameraPosition.x > 30) {
            this->cameraPosition.x = 30;
        }
        else if (this->cameraPosition.x < -30) {
            this->cameraPosition.x = -30;
        }
    }
```

When rotating the camera, the pitch and yaw value smust be considered:

```
     void Camera::rotate(float pitch, float yaw) {
        glm::vec3 direction;
        direction.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
        direction.y = sin(glm::radians(pitch));
        direction.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));
        this->cameraFrontDirection = glm::normalize(direction);
    }
```

The camera rolling option only takes into consideration the cameraAngle which can be adjusted by the user, and the effect is that the camera rotates around the z axis (front direction):

```
    void Camera::roll(float rollAngle) {
```

```
    glm::vec3 direction;
    direction.x = cos(glm::radians(rollAngle));
    direction.y = sin(glm::radians(rollAngle));
    direction.z = cameraRightDirection.z;
    this->cameraRightDirection = direction;
    this->cameraUpDirection = glm::normalize(
    glm::cross(cameraFrontDirection, this->cameraRightDirection));
}
```

3.1.2 Animation

The animation for the ball object is controlled using a separate Animation class, which keeps track of the object's current position and provides functionalities to move the object or animate a continuous movement.
The animations can be controlled entirely by the user, he/she can start/stop the animation using the correct keyboard inputs. Some of the animations will also automatically stop after a certain amount of time or after they reached a certain position.
While an animation is running, the user cannot start another one, unless it stops the previous one first.
The following animations are provided by this class:

- Bounce: bounce movement added to the ball when it is dropped down.
  It simulates the bounce effect using a damped oscillation function, where the damping factor depends on the object's elasticity, and the animation will stop after a given amount of time.

```
float dampedOscillation(float amplitude, float dampingFactor,
float oscillationFrequency,float time) {
 return amplitude * std::exp(-dampingFactor * time) *
 cos(PI * oscillationFrequency * glm::radians(time));
}

void Animation::bounce(float initialHeight) {
 float dampingFactor = 1.0 - elasticity;
 float elapsedTime = glfwGetTime() - animationStartTime;
 float amplitude = (initialHeight + 1) * BOUNCE_HEIGHT * elasticity;
 double oscillation = dampedOscillation(amplitude, dampingFactor,
 animationSpeed, elapsedTime);
 float posY = UNIT_STEP * abs(oscillation);
 if (posY <= MIN_BOUNCE) {
 stopAnimation();
 this->currentPosition.y = 0;
 this->ballPickedUp = false;
 return;
 }
 currentPosition.y = posY;
 transformationMatrix = glm::translate(glm::mat4(1.0),
```

```
 glm::vec3(0, -initialHeight + posY,0));
}
```

- Dribble: dribble the ball. It is similar to the bounce animation, only that it has no damping.

```
void Animation::dribble(float initialHeight) {
 float elapsedTime = glfwGetTime() - animationStartTime;
 float amplitude = (initialHeight + 1) * BOUNCE_HEIGHT;
 double oscillation = dampedOscillation(amplitude, 0, animationSpeed,
 elapsedTime);
 float posY = UNIT_STEP * abs(oscillation);
 currentPosition.y = posY;
 transformationMatrix = glm::translate(glm::mat4(1.0),
 glm::vec3(0, -initialHeight + posY, 0));
}
```

- Throw the ball: the ball is thrown in the direction indicated by the angles pitch and yaw. It mimics a parabolic movement, the higher the pitch the smaller the distance travelled by the ball in the air. When the ball falls down to the ground a bounce animation is started.

```
void Animation::throwBall(float pitch, float yaw) {
 // trajectory of the flying ball is a parabola
 float velocity = 25;
 float g = 9.8;
 float time = glfwGetTime() - animationStartTime;
 float x = - velocity * cos(glm::radians(yaw)) *
 sin(glm::radians(pitch)) * time;
 float z = - velocity * cos(glm::radians(pitch)) * time;
 float y = velocity * sin(glm::radians(pitch)) *
 sin(glm::radians(yaw)) * time -
 g * time * time / 2;

 this->transformationMatrix = glm::translate(glm::mat4(1.0),
 glm::vec3(0, y, z));
 this->currentPosition = glm::vec3(this->transformationMatrix *
 glm::vec4(this->initialPosition, 1.0));

 if (currentPosition.y < 0) {
 stopAnimation();
 currentPosition.y = initialPosition.y / 2;
 animateBounce();
 }
}
```

When the ball is thrown it is checked whether or not it hit a fence, and it will be thrown back in the reverse direction:

```
void Animation::hitAndBounce() {
 stopAnimation();
 animateThrow(180, -180);
}
```

- Spin the ball: spin the ball around its y axis. It also involves a damping, after a given amount of time, the spinning will slow down until it stops completely.

```
void Animation::spin(glm::vec3 axis) {
 float dampingFactor = 1.0 - this->weight;
 float elapsedTime = glfwGetTime() - animationStartTime;
 float dampingCoefficient = std::exp(-elasticity * elapsedTime);
 float angularVelocity = 2*this->animationSpeed * dampingCoefficient;
 GLfloat rotationAngle = elapsedTime * angularVelocity;
 if (dampingCoefficient <= MAX_DAMPING) {
 stopAnimation();
 return;
 }
 this->transformationMatrix = glm::rotate(this->transformationMatrix, glm::
 glm::mat4 spinTransformation = glm::translate(this->transformationMatrix,
}
```

3.1.3 Light source

In order to control the light sources added to the scene a separate class was created for it, namely LightSource. It holds important properties of the given light source, such as its position, its target and the direction it point to if it's the case, and it provides a move function which can reposition the light source using a rotation movement, that is the distance between the light source and the object remains the same:

```
void LightSource::move(gps::MOVE_DIRECTION direction) {
    switch (direction) {
    case gps::MOVE_LEFT: {
        rotate(1.0, glm::vec3(0.0f, 0.0f, 1.0f));
        break;
    }
    case gps::MOVE_RIGHT: {
        rotate(-1.0, glm::vec3(0.0f, 0.0f, 1.0f));
        break;
    }
    case gps::ROTATE_CLOCKWISE: {
        rotate(-1.0, glm::vec3(0.0f, 1.0f, 0.0f));
        break;
    }
    case gps::ROTATE_COUNTER_CLOCKWISE: {
```

```
            rotate(1.0, glm::vec3(0.0f, 1.0f, 0.0f));
            break;
        }
        case gps::MOVE_UP: {
            rotate(1.0, glm::vec3(1.0f, 0.0f, 0.0f));
            break;
        }
        case gps::MOVE_DOWN: {
            rotate(-1.0, glm::vec3(1.0f, 0.0f, 0.0f));
            break;
        }
        case gps::MOVE_FORWARD: {
            // increase intensity of light
            if (ambientStrength < 1.0) {
                ambientStrength += 0.1;
            }
            break;
        }
        case gps::MOVE_BACKWARD: {
            // decrease intensity of light
            if (ambientStrength > 0) {
                ambientStrength -= 0.1;
            }
            break;
        }
        default: break;
        }
    }
    void LightSource::rotate(float angle, glm::vec3 axis) {
        // vertical rotation (around y axis)
        setTransformationMatrix(glm::rotate(glm::mat4(1.0f),
        glm::radians(angle), axis));
    }
```

3.1.3.0.1 Day cycle animation

In order to simulate the changing light intensity during the day and night time, a day-cycle simulation was added, which runs continuously until it is stopped. It sends the updated value of the ambient light's intensity to the skybox shader, which will render the background with a darker color. In the same time, the shader for rendering the scene is changed automatically to the night light shader, when the ambient light decreases below half of its maximum value, and it is changed back to the basic shader, when it increases above it.

Chapter 4

Data structures

The main data structures used in the project were the ones defined in the OpenGL libraries, namely the vectors, matrices, int and float values, as well as the data structures for the uniform variable's locations:

- Vectors: glm::vec3, glm::vec4
- Matrices: glm::mat3, glm::mat4
- Primitive values: GLuint, GLfloat

Chapter 5

Class hierarchy

The project is structured around the following classes:

- Main
- Camera
- Animation
- LightSource
- Shader
- Model3D
- Mesh
- SkyBox

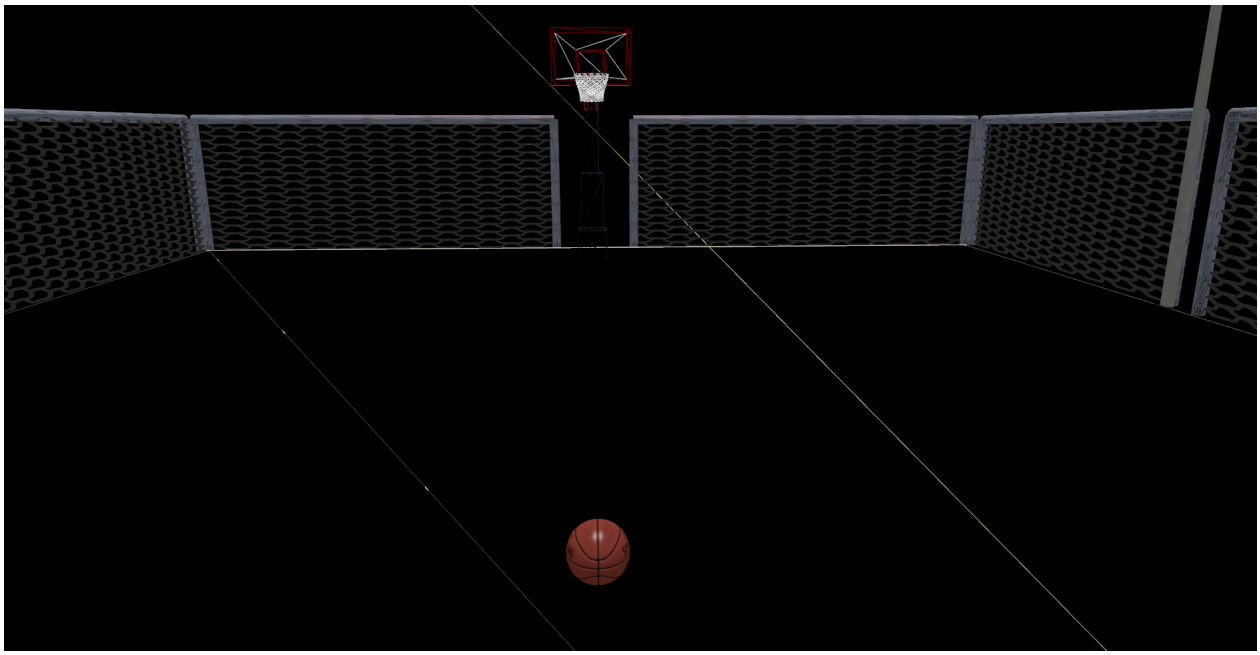
Chapter 6

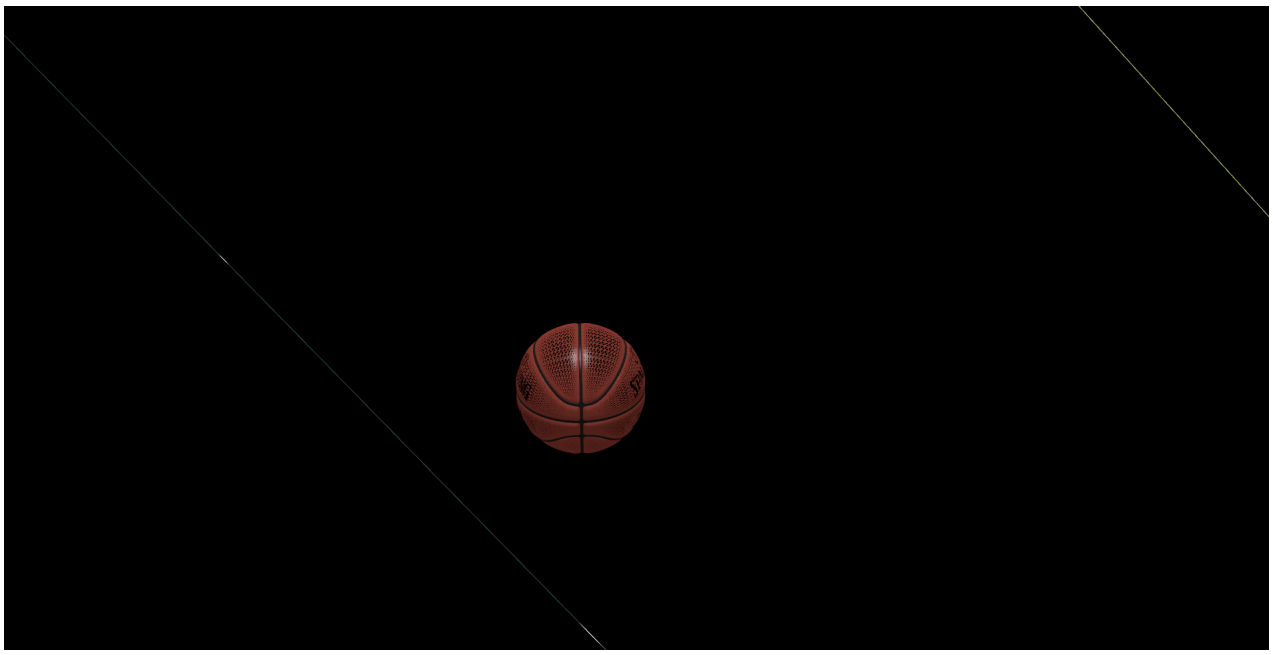
Graphical user interface

The user can manipulate the scene and interact with the object using the following key and mouse inputs:

6.1 View wireframe surfaces

1. View wireframe: **Left control** + **W**
2. Disable wireframe view: **Left control** + **F**





6.2 Controlling the camera movement

1. Move forward: **W**
2. Move backward: **S**
3. Move left: **A**
4. Move right: **D**
5. Move up: **F**
6. Move down: **C**
7. Roll camera: **R**
8. Rotate camera: Moving the mouse in the window: xoffset gives the yaw, yoffset gives the pitch values. The pitch must be between the values (-89,89) so that the user doesn't turn around completely, it would not be realistic.
9. Rotate the camera using only the yaw values: **Q** and **E**
10. Rotate the camera using only the pitch values: **UP** and **DOWN**
11. Move the camera to the left or right: **LEFT** and **RIGHT**
    This changes the position of the objects as well, therefore the ball also moves together with the camera.

6.3 Controlling the ball animation

1. Pick up the ball: **Left control + P**
2. Drop the ball: **Left control + D**
3. Stop animation: **Left control + Z**
4. Animate bounce: **Left shift + B**
5. Animate dribble: **Left shift + D**
6. Animate spin: **Left shift + S**
7. Animate throw: **Left shift + T**
   The position from which the ball is thrown is controlled by the camera's movements.
8. Rotate camera: Moving the mouse in the window: xoffset gives the yaw, yoffset gives the pitch values. The pitch must be between the values (-89,89) so that the user doesn't turn around completely, it would not be realistic.

6.4 Controlling the light sources

6.4.1 Control the day cycle animation:

1. Start animation: **Space bar**
2. Stop animation: **V**

6.4.2 Control the ambient light intensity:

1. Increase the intensity: **Right Alt**
2. Decrease the intensity: **Left Alt**

6.4.3 Select the current shader:

1. Day light shader: **1**
2. Night light shader: **2**

3. Spot light shader: **3**

4. Flash light shader: **4**

5. Point lights shader: **5**

6.4.4 Select the light source and control its movement (in case of the multiple point lights:

1. Select the one on the left: **6**

2. Select the one in the middle: **7**

3. Select the one on the right: **8**

4. Rotate to the left (z axis): **J**

5. Rotate to the right (z axis): **L**

6. Rotate upwards (x axis): **I**

7. Rotate downwards (x axis): **K**

8. Rotate clockwise (y axis): **U**

9. Rotate counter-clockwise (y axis): **O**

Chapter 7

Conclusions and further developments

Working with the OpenGl library I realised how many things can be done with simple transformations and mathematical functions, such as animating an object or simulating the movement of a camera. The texture mapping had conveyed a very realistic view for the objects, and the lighting techniques together with the shadow computations offered a vivid picture of how the objects interact with the light in the real world.

For further extending the application I could have added more control over the animations, such as displaying a message when the player has hit the goal, and the ball fell through the hoop. I could also add another scene, such as an indoor basketball court, which would require different lightings, point lights for example.

The application could be also extended by adding the shadow computation for point shadows, but I discarded that option, as it would be beyond the scope of a simple introductory application.

Chapter 8

References

1. Blender editor tool: `https://www.blender.org/`

2. Free 3D obj models can be dowloaded from:
   `https://www.cgtrader.com/`
   `https://free3d.com/`

3. Textures for the skybox can be downloaded from:
   `https://www.humus.name/index.php?page=Textures`

4. Reference for implementing the different types of lights and shadow mapping algorithms:
   `https://learnopengl.com/Lighting/Light-casters`