186.831Computergrafik UE, SS16

DOCUMENTATION

A ROBOT'S PURPOSE

Second Submission

Philipp Klein, e1326251
Jakob Knapp, e1327386

1 DESCRIPTION OF OUR IMPLEMENTATION

Following requirements have been implemented in our game:

1.1 Gameplay

As the game is a combination between platformer and top-down shooter, the main gameplay-elements are moving around, jumping, aiming and shooting. The player can destroy the evil robot factory by reaching its last room and shooting the self-destruction button. The player is then notified about his success via console output. When the enemies are able to reach the characters position, the player loses and is notified about his defeat by console output. The game can also be lost by falling into the void when failing the jump-section after the second room.

Step-by-step instruction to beat the game:
First, shoot the two robots in the first room to eliminate them. Afterwards, go north to get to the first hallway, where you have to jump onto the two platforms to reach the next room. After passing through said room, you have to avoid falling into the abyss by using the platforms to get to the other side. After doing this, you enter another hallway, where you have to jump twice to get to the next floor. After reaching the last room, use the ego camera to shoot the destruction button on the wall to destroy the factory.

1.2 Controls

KEY	EFFECT
W, A, S, D	Move character
Space	Jump
Mouse Cursor	Aim
Left Mouse Button	Shoot
C	Change camera
F3	Wireframe Mode On/Off
F4	Textur-Sampling-Quality: Nearest Neighbor/Bilinear
F5	Mip Mapping-Quality: Off/Nearest Neighbor/Linear
F6	Decrease Ambient Light Strength
F7	Increase Ambient Light Strength
F8	View-Frustum Culling On/Off
F9	Transparency On/Off

The player can move his character along the x- and z- axis by using the 'W', 'A', 'S' and 'D' keys. When the 'SPACE' key is pressed, the character jumps, which is useful to get on platforms. Aiming is done using the mouse cursor. The character always faces the cursor. When the player presses the left mouse button, the character fires a projectile in his viewing direction. Pressing the 'C' key changes the camera perspective from standard to ego mode and back. Moving the mouse cursor in ego mode leads to the character looking around. Pressing the 'F'-keys enables, disables or changes various features of our game.

1.3 Freely Moveable Camera

The standard camera perspective is some kind of top down camera, which always remains focused on the main character, who is seen from an angle of about 45 degrees. This should allow the player

to always remain focused on the character, while also enabling simultaneous aiming and moving. Other games, which inspired us to use this kind of camera perspective, were top-down shooters like Nuclear Throne and MOBAs like League of Legends. But because this camera alone does not give too much flexibility to the player, we also implemented a second camera, the “ego camera”, which enables the player to see the world through the characters eyes and aim more accurately. When using this camera, the character can shoot projectiles in his viewing direction, which means bullets can be fired towards the ceiling, which is impossible when using the standard camera due to the bullets only moving along the x- and z-axis. The character can look up in a 90 degree angle, down in an about 60 degree angle and spin around for 360 degrees. This power has got a trade-off though, because we did not like the idea of players only using the first person perspective, which is not how the game is meant to be played. While using the “ego camera”, the player is rooted and therefore not able to move or jump, making him vulnerable while granting increased aiming accuracy.
When the camera is changed from standard to ego mode by pressing the 'C' key, the viewing direction remains unchanged. When changing back though, the character will face the direction in which he looked when the camera was initially changed from standard to ego mode.

1.4 Moving Objects

There are several moving objects in “A Robots Purpose”. The player can actively move the character by using the 'W', 'A', 'S', 'D' and 'SPACE' keys, as long as the standard camera is used. When pressing the left mouse button, the character fires a projectile towards the cursor, which travels in a certain direction by itself. The enemy robots, which are in the game, move towards the player by themselves. They get their moving direction by calculating a direction vector between the enemy’s position and the character’s position.

1.5 Animated Objects

To make our enemies even scarier they have a saw, which rotates as the enemy is moving toward the character. After loading the body of a robot we also load the saw separately. The enemy class now gets both models and inside this class we call the update function of the saw. There the rotation is calculated. First we calculate the new rotation matrix of the saw by simply rotating the current one. Now we can multiply it with the model matrix of the body and we get the model matrix of the saw. To render it correctly we then set the matrices in the shader and get some nicely rotating saws.

1.6 Complex Objects/Model Loading

We created our models with Blender and have decided to use .obj files for our models, because it is a very simple file format and blender supports the exportation in this format.
The modelmatrix and the filepath are set in the main-Class where we declare our models. In our Model-Class the forwarded filepath gets processed using the Assimp (Open Asset Import Library) library. The Assimp data structures are used to store all of the imported data. Assimp loads the entire model in a scene structure that contains all information of the imported file. For example the required data about the meshes and materials/textures of this model are stored in this scene structure. Each model is build by a different number of meshes. Our robot for example has currently five meshes (it will get more when we add more details soon). These meshes represent the data that is actually drawn, therefore each of the models meshes has to be processed and passed to the Mesh-Class. If the model has any textures they are also loaded in the Model class using the SOIL library and the required texture coordinates get forwarded to the responding mesh. If the model is successfully loaded we can interact with it. This is done with the update, move and rotate functions.

All of these manipulate the modelmatrix of the corresponding model by handling the input parameters, either by user input (keybord or mouse) or automatic calculation (moving enemies to character). The move functions e.g. translates the modelmatrix with the moving direction, which we get from Boolean values that are set according to the user input (W, A, S, D).

1.7 Collision Detection

Collision Detection has been implemented by us without using a library by using Axis-Aligned Bounding Boxes (AABB). Each object (character, enemy robots, bullets, the destruction button, walls, floors and other map-objects) has its own bounding box. Those are already calculated when the models are loaded. Therefore, the smallest and biggest x-, y- and z-coordinates of the model, which make up two opposing corners of the box, are stored. Whenever the objects move, the two points are multiplied with the model matrix (except in case of the character, where they are multiplied by the translation matrix to not be affected by rotating the character) to transform them into world coordinates and therefore update their position. Those coordinates are then used for detecting collision, which only requires some arithmetical checks.

1.8 Lighting and materials

All objects in the game, the main character, the enemies, the map, the bullets and the button have materials assigned to them. The material properties, ambient reflection, diffuse reflection, specular reflection and shininess define the objects appearance, to which the material is assigned.
All objects are lighted by spotlights, which shine from the ceiling towards the ground. Each room has its own light source, which is turned on when the character enters the room, to create the feeling of someone monitoring the player. Additionally, we used rather dark, reddish ambient light to give the scenery a creepy touch.
The map and the button are shaded using Blinn-Phong-Shading, which can be seen when looking at the specular highlights on the floor. Asimov, the players character, and the enemies are shaded using cel-shading, giving them a cartoony appearance. The bullets are rendered using a combination of cel-shading, to give them a black outline, and a special transparency shader, which makes them transparent and allows them to not be affected by the light sources.

1.9 Texture Mapping and Texture Quality

The main character, the two enemy robots, the map and the button at the end of the map are textured. The buttons texture was already prepared, while the other textures have been assigned to the models in Blender. They get loaded in the Model Class by usage of the SOIL (Simple OpenGL Image Library)-Library.
Texture Sampling Quality and Mip-Mapping Quality can be changed by using the F4 and F5 keys. The result can best be seen by looking at the brick walls or the floor when using the first person camera. Changing the Texture Sampling Quality results in the floor being drawn more or less detailed for example. The effects of changing the Mip-Mapping Quality can be noticed by looking at distant walls, which are being drawn more or less detailed.

1.10 View-Frustum Culling

View-Frustum Culling makes sure that objects, which can not even be seen from the players point of view, are not rendered because they cannot be seen anyway. This is done by calculating the

View-Frustum, basically a pyramid without its top, every frame by using the camera internals, like field of view, near and far plane etc., and the cameras position and viewing direction. After that it is checked, if the bounding boxes (see 1.8) are completely contained in or intersect with the View-Frustum. If they do, they can potentially be seen from the players perspective and are rendered. Potentially, because they could be concealed by another object, for example a wall, which is not taken care of in our case. The number of objects drawn in relation to all objects in the scene is displayed in the window title as performance output.

1.11 Transparency

The bullets, which are fired by the player, are transparent, which means you can see the walls or enemies behind them. This is achieved by passing an alpha value to the shader used to render the bullets and by enabling GL_BLEND. The transparency can be turned on and off by pressing F9.

1.12 Experimenting with OpenGL

VAOs and VBOs are used in our game to pass the vertices of meshes to the graphics card, which can be seen in the Mesh class. An FBO is used for calculation of the shadows, more specifically for drawing the shadow map. Changing Texture Sampling and Mip-Mapping Quality has already been described in 1.9.

2 EFFECTS

2.1 Spotlights with inner and outer cone (0.5 points)

The first effect, which has been implemented, are spotlights. We decided to use them because they fit the factory setting of our game very well and also go very well with shadow mapping. We had already implemented point lights for our first submission, so changing them to spotlights has not been that much of a problem. A spotlight is defined by a direction, in which the light is emitted, and the angle of two cones, the inner and outer cone. In our game, the spotlights appear to shine down from the ceiling. In our fragment shaders, we first calculate the angle between the point on the surface, which is to be illuminated. If the angle is bigger than the angle of the outer cone, the point is not illuminated by the spotlight and only affected by ambient light. If the angle is smaller then the angle of the inner cone, it means, that the surface is illuminated by the spotlights full power. If the angle is in between the inner and the outer cone angle, linear interpolation is used to calculate the strength of the light hitting the surface, which gives a smooth transition. The further away from the inner cone an object moves, the less it is illuminated.

2.2 Shadow Mapping with PCF (1.5 points)

Shadow Mapping has been implemented as it has been described in the lectures.
At first, the scene is rendered from the light sources point of view. Its view matrix has to be adjusted whenever the player enters the next room and another light source is activated. The scene is then rendered to a Frame Buffer Object using a special depth-shader, which only draws the depth to the texture attached to the FBO. This depth texture encodes the distance of various objects to the light source. When drawing the actual objects to the screen again, using blinn-phong shaders for example, said depth texture is then passed to the shader, which is used. Additionally, a matrix has to

be provided transforming the point, which is to be shaded, into light space to check if it is concealed by another object, which casts a shadow. This is checked for each point. When another point is closer to the light source than the point which is currently being evaluated, it is in the shadows. There are several artefacts though, which must somehow be countered. The most obvious and ugly of those is shadow acne, which can be avoided by enabling GL_POLYGON_OFFSET_FILL in OpenGL. It pushed the drawn polygons back by a certain distance which can be set by the user. It comes at a price though, because the actual shadows are also pushed a bit back, which results in them not being fully connected to their casting objects anymore. This can be resolved by increasing the resolution of the depth texture. To make the shadows look smoother, PCF can be used. This is done by setting certain parameters of the depth texture and by using a sampler2Dshadow instead of a simple sampler2D. The pixels are then interpolated bilinear automatically, which makes the shadows appear way smoother.

2.3 Cel Shading + Contours (backfaces) (0.5 + 0.5 points)

We decided to use cel shading to create a unique look of our game. But only the character and the enemies are drawn using cel shading. All other objects use different shaders. First of all we have to render a black silhouette of the model to create black contours. It is a simple shader which takes the position and the normals as input vectors and the model-, view- and projection-matrix as uniforms. Additionally we set an offset. This offset is used to create a bigger silhouette, so we can see it when we render our object with the actual cel shader. We simply multiply the normals of the model with the offset and get our new position. The fragment shader creates now an output with the specified color (in our case black). The actual cel shading adds up the intensity of all the lights affecting the point (in our case the ambient, specular and diffuse term). In the fragment shader we specify a number of levels that we want to use for cel shading. We have chosen 4 nuances because you can see the effects of cel shading very good and it looks nice. Now we can calculate the shade intensity of the current point by multiplying the brightness with the number of shades. The resulting value is round up and divided by the number of shades. The value we get from this calculation is used by multiplying it with the texture color of the corresponding point. With this technique we get a nice looking toon style for our models.

2.4 GPU-Particle System (+Transform Feedback,Instancing) (1 point)

To make our game more realistic we decided to implement the GPU-Particle System using transform feedback. We create different particle system for different collisions of a bullet. A collision with the map creates a small bluish particle system that is only alive for a short period of time. If a bullet hits an enemy the particle system is bright yellow and orange to simulate an explosion. This type is bigger and lives longer. To make winning the game good-looking we spawn a very big, orange particle system if the bullet hits the destruction button. It's alive for a long period of time to simulate the destruction of the factory. Afterwards the window closes as the player has won the game. We only want to create a particle system if a bullet hits something. So we have to check if this happens. If so we put a new particle system in our particleSystem vector. For a GPU-particle system we need two shaders – one to update our particle and one to render them. The update shader writes to the transform feedback buffer and the render shader draws the particles written in the transform feedback buffer. Because of this we need two buffers – one where the particles are written into and the other one where the particles are read from. After each frame we need to swap them. This means the former read buffer is now the write buffer and vice versa. To update and therefore create new particles in our particle system we need to set some uniforms in the shader. These are the spawn position of the particle system, the minimum and maximum velocity, the gravity force applied to the particles, the color, the minimum and maximum lifetime of a particle, the size of the particles and the threshold to create new particles. To get the spawn position

of the particle system we need the hit position of the bullet. This is just the last position of the bullet before it gets deleted. The other values are set manually and, as already mentioned, differ from the type of the collision. There are two different types of particles – one is a generator and the other is a normal particle. The generator as its name implies generates new particles and the normal particles are simply modified. All of this happens in the update geometry shader. There we check if a particle is a generator. If so we create the number of particles we set as a uniform. If it is a normal particle we modify the values and check if its lifetime is already over.
Particles are only 2D-Objects, so we have to ensure that they are always facing to the user. This is done using the billboarding technique where we create two vectors with the view, eye and up vector of the camera. These two vectors are then used in the render geometry shader to create a quad at the particle position that is always facing to the user. We now only have to apply the correct size to the quad and pass the color and lifetime to the fragment shader, where they are used to let our particles fade out as its lifetime grows.

3 ADDITIONAL FEATURES

We have also implemented features, which were not required:

3.1 Custom Cursor

We have decided that it would be a nice feature of our game to use a different cursor. The keynote of this decision was to provide better aiming, and of course a shooter looks better with a custom cursor. We have chosen a small red crosshair with 50x50 pixel. First of all the new cursor needs to be imported, which is done using the SOIL library. With the loaded image information we create a new cursor and use it in our current window. It is also necessary to set the center of the crosshair, so the bullets move toward the desired position. We also check if an error occurred, for example a bad filepath, and tell the user that something went wrong. If so, the standard cursor is used and we can play the game without any further restrictions.

4 IMPLEMENTATION DETAILS

At last, we want to give insight on some interesting implementation details.

4.1 Aiming / Ego Camera

When in standard mode, ray tracing is used to calculate the players viewing direction. Therefore, the cursors position is queried and then runs through the graphics pipeline in reverse direction, which means that the position has to be transformed to normalized device coordinates first and then to homogeneous clip coordinates. Next, the perspective is inverted by multiplying the vector by the inverse projection matrix. Afterwards, it is finally transformed from camera into world space by multiplying with the inverse view matrix of the standard camera, which gets us the vertex from the cameras eye point into the three-dimensional scene.
Lastly, the rays intersection with the plane, which is parallel to the x- and z-axis and put up by the formula , is calculated. This is the point, where the player is aiming and the vector from the character to that point can be calculated easily. Said vector is used as direction, in which bullets are moving, as well as for spinning the character around the y-axis to always face the point where the player is aiming.
To calculate the rotation, the angle between the viewing vector and the vector (0/0/-1) are used. The

angle is calculated by using the dot-product,stored and on the next iteration calculated again. The new angle is compared to the angle of the previous iteration and if it has changed, the character is rotated accordingly.
In ego mode, the calculations work slightly differently. Whenever the camera perspective is changed from standard to ego mode, the angle between the viewing vector and the vector (0/0/-1) is calculated to initialize the view correctly. The vertical angle is 0 at first, which means that the player always looks at the horizon first when changing perspective. From there on, simple trigonometry is used to calculate the angles, in which the mouse cursor has moved since the last frame. This information is used to calculate the new viewing direction, right hand vector and up vector, which are passed to the glm::lookAt function to calculate the new view matrix. The cursor always remains centred.

4.2 Shooting

When the player presses the left mouse button, a new bullet is created on the characters position. The characters current viewing direction will be used as direction, in which the bullet travels. This direction is passed to the bullet in its constructor and from there on, the bullet moves in said direction by itself by translating the model matrix each update-cycle.

5 ADDITIONAL LIBRARIES AND TUTORIALS

Following additional libraries have been used:

- Assimp, for loading .obj-files (Models)
  http://www.assimp.org/index.html
- SOIL, for loading images (textures and cursor)
  http://www.lonesock.net/soil.html

Following tutorials have helped us create our game:

- Recordings and slides provided in the CG Wiki
  https://lva.cg.tuwien.ac.at/cgue/wiki/doku.php?id=students:slides
- Tutorial for calculating characters viewing direction
  http://antongerdelan.net/opengl/raycasting.html
- Tutorial for rotating ego camera
  http://www.opengl-tutorial.org/beginners-tutorials/tutorial-6-keyboard-and-mouse/
- model and texture loading
  http://www.learnopengl.com/
- Shadow-Mapping
  http://learnopengl.com/#!Advanced-Lighting/Shadows/Shadow-Mapping
  http://www.opengl-tutorial.org/ru/intermediate-tutorials/tutorial-16-shadow-mapping/
  http://www.ogldev.org/www/tutorial47/tutorial47.html
- View-Frustum Culling
  http://www.lighthouse3d.com/tutorials/view-frustum-culling/
- Cel-Shading
  http://sunandblackcat.com/tipFullView.php?l=eng&topicid=15
- Particle System
  http://www.mbsoftworks.sk/index.php?page=tutorials&series=1&tutorial=26
  http://ogldev.atspace.co.uk/www/tutorial28/tutorial28.html

http://prideout.net/blog/?tag=opengl-transform-feedback

Tools to create our Models:

- Blender
- Paint
- Gimp 2