2. Submission

— Deserter —

Bálint István Kovács 1227520, 033 532
Lilla Fésüs 1226017, 033 532

Implementation

Gameplay

Deserter is a semi-rail shooter game with scripted camera movement, but with a free movable first person view. The camera can be turned in an FPS mode and in a true rail shooter mode. The difference is, that in the latter the aiming is based on the cursor's location and the camera can be turned with the WASD key combination.

5	FPS mode
6	rail shooter mode

The aim of the game is to shoot down a given number of enemies within the given time. On each level the number of needed kills and the speed are higher, but the time is shorter. These variables are displayed in the top left corner before a level starts. If the time is up, the player loses.

Another feature is that the player can either be on an aircraft and shoot from the air or can drive a tank a shoot from the ground. These two modes are called air mode and ground mode and they can be set in the `configuration.txt` file. Please notice, that the air mode is only available with random player path generation. This random challenge can also be set in the `configuration.txt` file and in this case the predefined player path will be random generated in the `main.cpp`, before the level starts. We manipulate the `x` and `z` coordinates and the height is obtained from the ground with the help of a terrain collider (`mainTerrainCollider`).

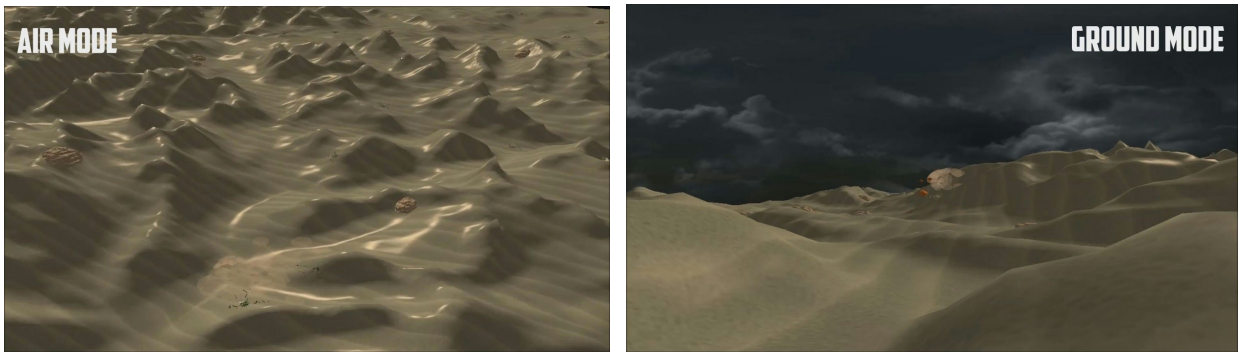


Fig.1.: Two gaming modes; in the air or on the ground

The terrain collider is mainly used for preventing to shoot through the sand. If the projectile meets the ground a semi-transparent dust cloud is rendered to enhance its effect.

Effects

Shadow Mapping with PCF (1.5)

Shadow Mapping is implemented through a simple depth map rendered from the point of view of the assigned light source. A separate frame buffer object is used to hold the depth texture, which is then bound for reading at the default render stage. Both the shadow map as well as the default render stages are provided information about the light position in order to calculate the correct values. Percentage closer filtering is used to create softer shadow edges. The implementation is based on OGLdev Modern OpenGL Tutorials Nr. 23, 24 and 42.

Since the huge size of the game world, shadows are not of uniformly good quality with a fixed light. To avoid this we implemented a permanently moving spotlight, which moves with the player and is by `20.0f` below the camera, so the shadows are not exactly behind the objects.

Fig.2.: Shadow mapping with PCF

Normal Mapping (1)

Normal mapping is implemented in world space based on the tutorial Gamasutra: Robert Basler - Three Normal Mapping Techniques Explained For the Mathematically Uninclined. The tangents and bitangents are post processed by the Assimp Loader Library and a TBN (tangent, bitangent, normal) matrix, as well as the light directions, are calculated in the vertex shader. The effect needs a normal map, for which the path is the second attribute of the method `void ModelLoader::load(std::string file, std::string normalMap, std::string reflectionMap, Scene::Mesh &m)`. The shader does not take the

specular attribute into account and the output color is based only on the ambient and diffuse lights.

Normal mapping is used to render the rocks and the drones above the enemies. To exaggerate the effect we placed a huge rock in one of the corners of the gamefield, around what a drone and a blue light source circles.

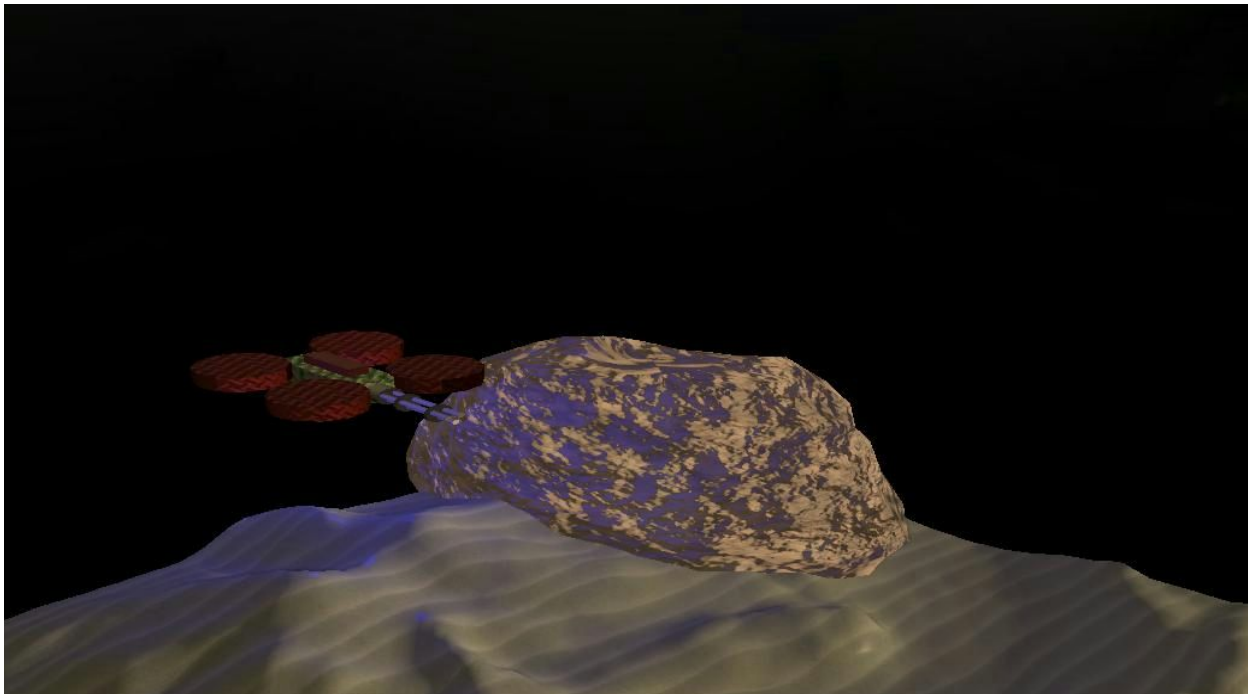


Fig.3.: Visualizing normal mapping

Environment Mapping (1)

Environment mapping is implemented with a reflection map, which is the third attribute of the method `void ModelLoader::load(std::string file, std::string normalMap, std::string reflectionMap, Scene::Mesh &m)`, however the default is with a fix reflexion coefficient. To enable the reflection map code must be uncommented in the `environment.frag` fragment shader.

Fig.4.: Environment mapping with a fix reflection coefficient *(left)* and with a reflection map *(right)*

The sky box is a separate class with its own texture loading, since we need to generate cube maps instead of 2D textures. The vertices of the skybox are hard coded in the class `SkyBox.cpp` and the depth test is disabled before its rendering. For the implementation of the sky box, as well as for the effect, following two tutorials were used: Learn OpenGL - Cubemaps, Anton Gerdelan - Cube Maps: Sky Boxes and Environment Mapping

Motion Blur (1.5)

The effect is a per object motion blur. At each render loop, a velocity texture is rendered in a separate step, containing the motion vectors of the rendered objects. These calculated from the current and the previous MVP matrices, which the camera and the model objects have to keep track of and provide in a correct manner. From the velocity texture, the speed and direction of moving fragments is calculated in screen space. This information is used to sample and blur the content of the default render stage (stored in a separate texture). Finally the results are rendered onto a fullscreen quad. This approach is outlined here: john-chapman-graphics - Per-Object Motion Blur.

The effect's strength can be adjusted in the `configuration.txt` file by setting the `fMotionblurFactor` to another float.

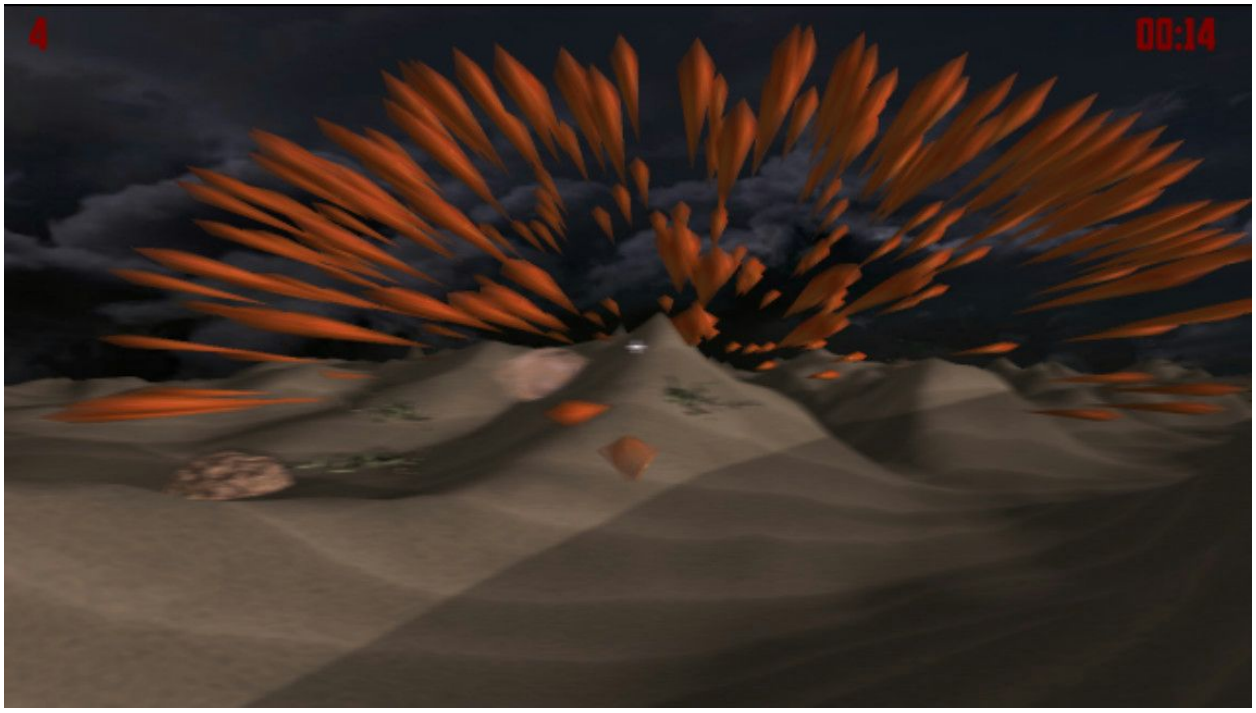


Fig.5.: Motion blur at the explosion of the enemy

Complex objects

For the import of complex objects we use the Assimp Loader Library and the FreeImage Library for their textures. The models were created by Bálint István Kovács with 3D Studio Max and each of them has texture and material, too.

To see the complexity of the object we provide a wireframe render mode. For turning it on/off press F3.

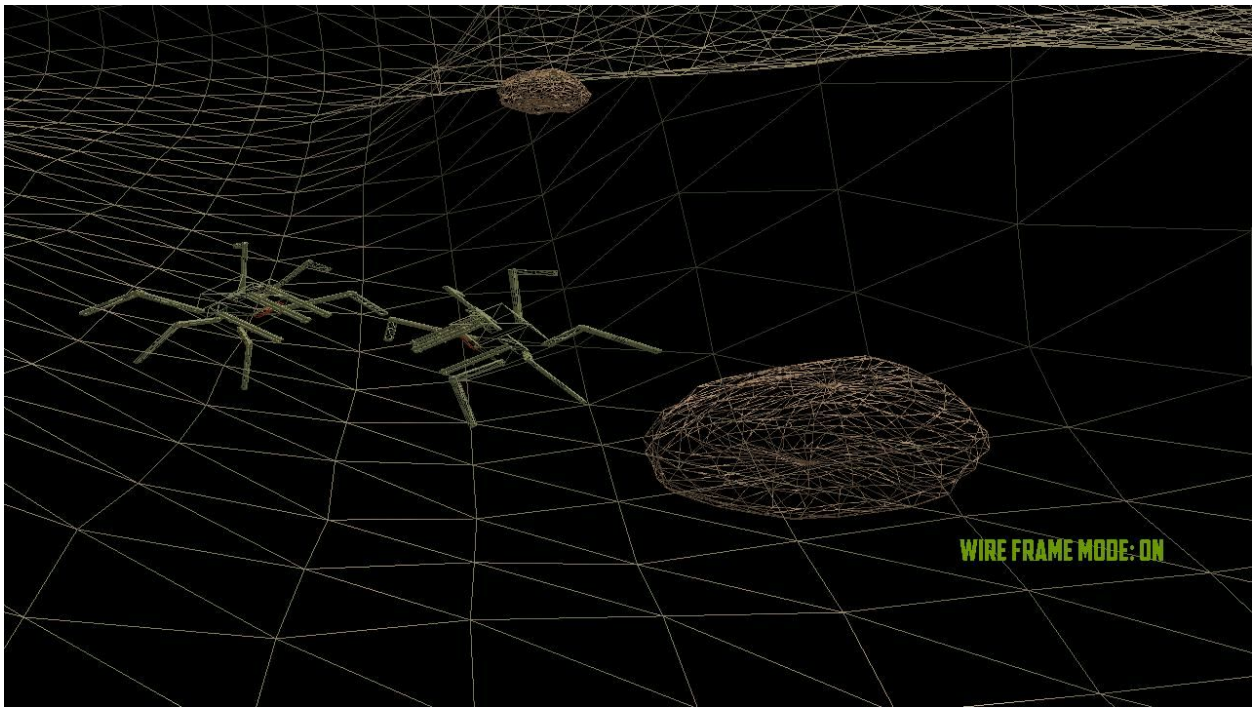


Fig.6.: Enemies and rocks in wireframe mode

Animated objects

Our animated objects are the enemies, which are composed of independent parts. (Currently: body, base and drone.) Every enemy has a battle drone attached, which rotates around the y-axis and is dependent from the crawler's position. The base of the crawler has three animation frames, which are loop around based on the current time.

View frustum culling

To check whether an object is within the view frustum of a camera the methods `bool Scene::Camera::isInViewFrustum(glm::vec3 point)` and `bool Scene::Camera::isInViewFrustum(glm::vec3 point, float radiusLength)` are used. The former tests a point and the latter tests a sphere. To test a cube we use the second method and generate a sphere with the middle and maximal point of its bounding box. The frustum planes are extracted from the view-projection matrix of the camera based on the tutorial Adventures in Game Development World - View Frustum. The distance between the planes and the sphere is the signed distance and it is out of the frustum if its center is on the wrong side of at least one plane and the distance to the plane is greater than its radius. To understand view frustum culling the tutorial Lighthouse3d.com - View Frustum Culling was very useful.

To see the effect of frustum culling press F2 to see the performance and press F8 to turn the culling off/on. When frustum culling is on less objects are rendered. Please notice, that the ground is not added to the number of rendered objects.

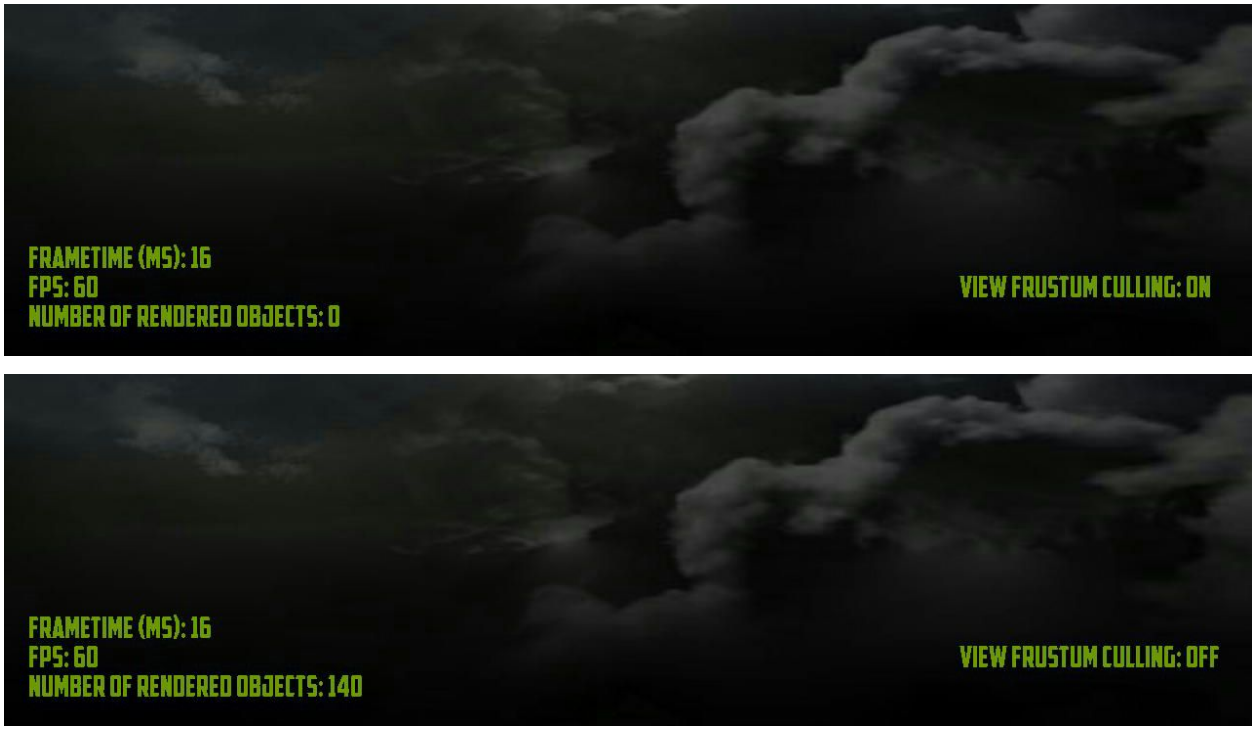


Fig.7.: View frustum culling on *(top)* and off *(bottom)*

Transparency

Transparency is applied on the sand dust, when the player shoots into the ground. The dust consists of several objects, that have different alpha values. Transparent object have a dedicated shader with fixed opacity values. To turn transparency on/off press F9.

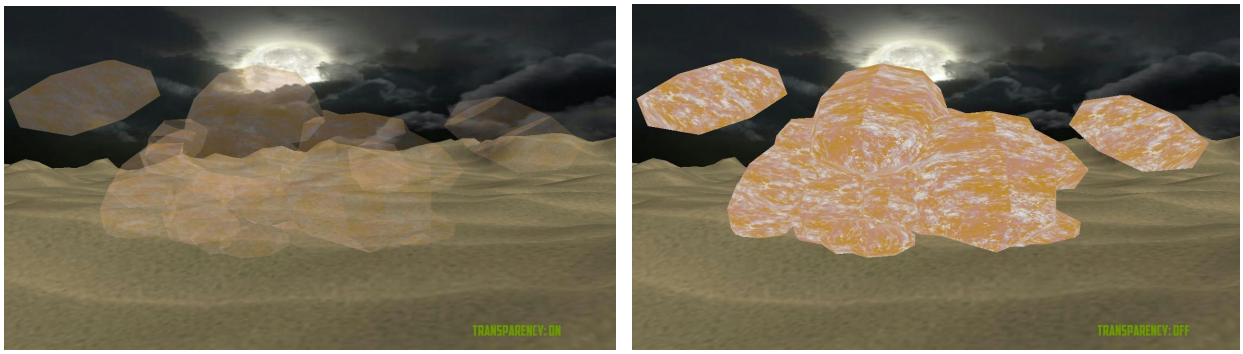
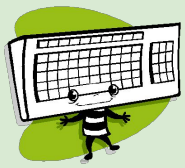


Fig.8.: Transparency on *(left)* and off *(right)*

Controls



WASD *(rail shooter mode)*	turn turret
WASD *(debug mode)*	move camera on horizontal axes
XY *(debug mode)*	move camera on vertical axis
MOUSE *(rail shooter mode)*	move crosshair
MOUSE *(FPS and debug mode)*	turn camera
LEFT MOUSE BUTTON	shoot
LEFT MOUSE BUTTON HOLD	continuous shooting
RIGHT MOUSE BUTTON HOLD	zoom + bullet time
MOUSEWHEEL *(debug mode)*	zoom in/out
P	*pause*
1	setup spotlight
2	setup directional light
3	-
4	-
5	FPS mode
6	rail shooter mode
7	debug mode
8	game mode
F1	-
F2	performance on/off
F3	wireframe mode on/off
F4	texture sampling quality nearest neighbour/bilinear
F5	mip mapping quality off/nearest neighbour/linear
F6	-
F7	-
F8	view frustum culling on/off
F9	transparency on/off

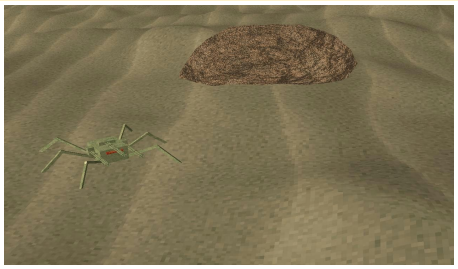
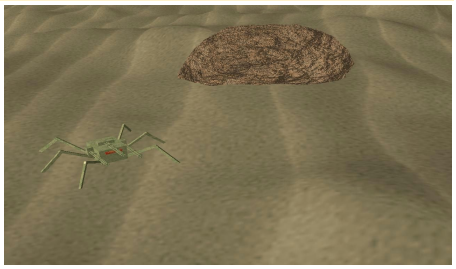
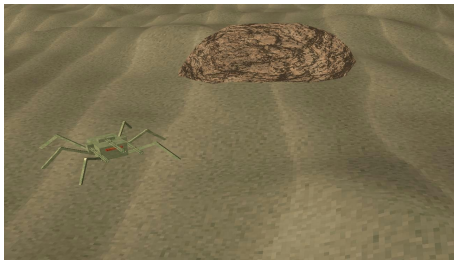
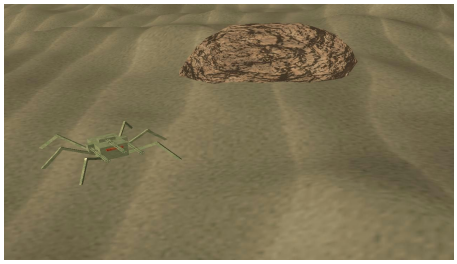
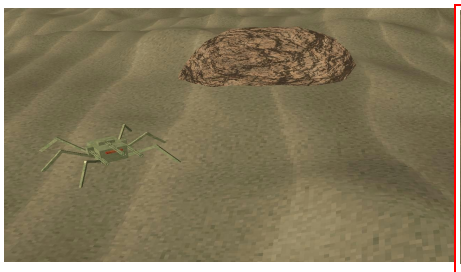
Tab.1.: Controls

Experimenting with OpenGL

In our implementation, different render targets are set by creating different types of framebuffer object to hold the configuration details (required texture objects and their settings, etc.) for the

task at hand. These classes supply the functionality for binding the right textures for reading/writing at the specialized render passes.

OpenGL Textures can be interpolated in two ways: scaling the image down with the keyword `GL_TEXTURE_MIN_FILTER` and scaling the picture up with the keyword `GL_TEXTURE_MAG_FILTER`. The former can have six parameters, namely `GL_NEAREST`, `GL_LINEAR`, `GL_NEAREST_MIPMAP_NEAREST`, `GL_LINEAR_MIPMAP_NEAREST`, `GL_NEAREST_MIPMAP_LINEAR`, `GL_LINEAR_MIPMAP_LINEAR`, but the latter can have only the first two. The table below visualises how the different filterings are implemented in our game. In order to change the settings press F4 for texture quality and F5 for mip mapping quality.

Texture sampling quality / **Mip mapping quality**	nearest neighbour `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);`	bilinear `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);`
off		
nearest neighbour		
linear		 default

Tab.2.: Filter qualities at different settings

Lightning, textures and materials

`Light.h` represents our lightning object. We provide implementation for spotlight, point light and directional light and we do have both in our initial game. An array in the `main.cpp` contains the loaded light sources, where `gLights[0]` is a spotlight, `gLights[1]` is a directional one and `gLights[2]` is the directional muzzle flash. We differentiate directional light by setting the `w` coordinate of the position `0.0f`. The initialization happens also in the `main.cpp` in the `SetupLights()` method. A spotlight moves with the player permanently and provides the moving light source for normal and shadow mapping.

A light source has several attributes (with their default values):

- position (`0.0f, 0.0f, 0.0f, 1.0f`)
- color (`1.0f, 1.0f, 1.0f`)
- attenuation (`0.1f`)
- ambient coefficient (`0.1f`)
- spotlight angle (`180.0f`)
- spotlight direction (`0.0f, 0.0f, 0.0f`)

Interesting feature is our muzzle flash light, that occurs at firing and illuminates the environment.

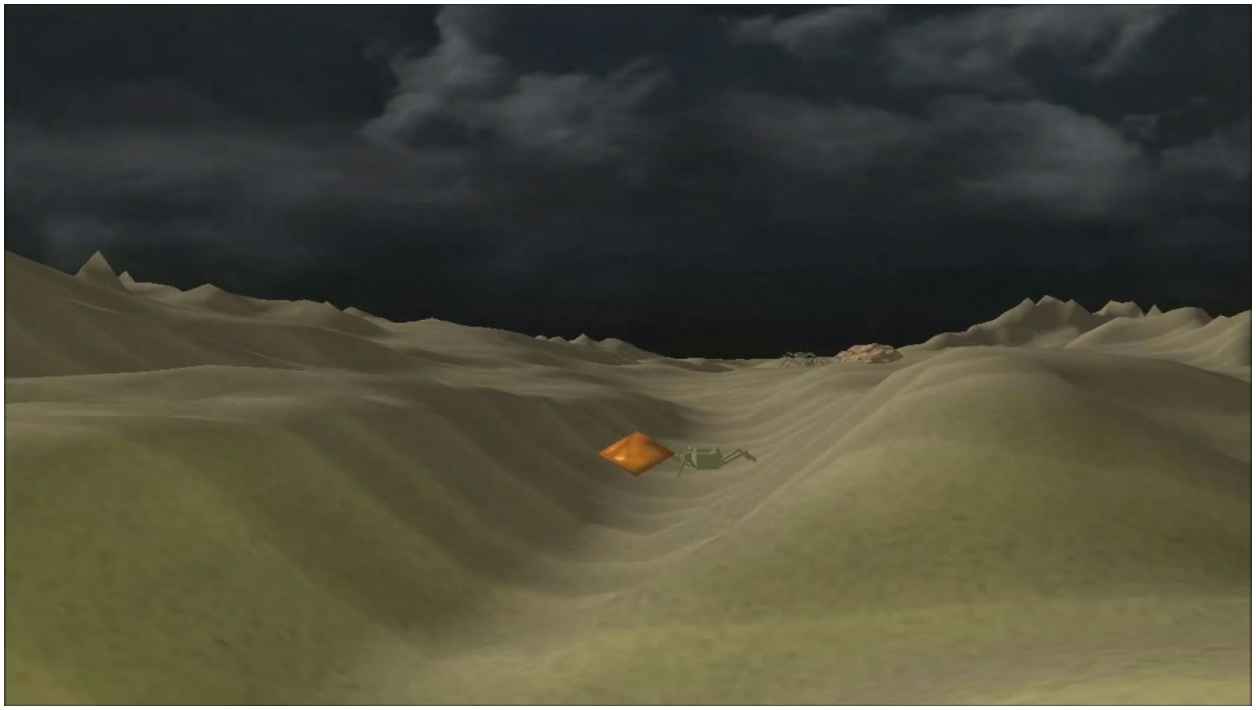


Fig.9.: The yellowish light is the muzzle flash

Every object in the game is textured. The color map is loaded automatically in the `ModelLoader` class and the file's location is encoded in the material (`.mtl`) file of the object.

The actual texture loading happens with the FreeImage library also in the `ModelLoader` class, but for example the skybox has it own texture loading and binding function.

Since only one texture can be saved in the material file, additional textures, for example the normal map's and reflection map's path, must be provided explicit. If the model does not have one, an empty string will be passed and the loading of these maps are skipped.

The texture binding is performed in the `ModelAsset` class. The mesh has two booleans, called `hasNormalMap` and `hasReflectionMap`, which indicates, whether a normal or reflection map was loaded. If first is the case, than the normal or reflection map will be bound as well and assigned to the shader.

The material is a struct in the `Mesh` class. The attributes are loaded in the `ModelLoader` class from the material file with the help of the Assimp library functions. A material has following properties:

- diffuse
- ambient
- specular
- emissive
- shininess

Not all of these properties are considered in every shader, for example the normal mapping shader does not use the attributes specular, emissive and shininess.

Features

We tried to implement realistic features, that imitate real physical occlusions. These are for example the dust cloud, if the player hits the sand, the muzzle flash at every shot and the shaking at the explosion of an enemy.

Another remarkable feature is the random and automatic path generation. We do not need to predefine every level, but they are still different. In this way unlimited levels are possible and we spare space, since we do not need to store the path in an `.obj` file.

At the beginning of the game the player must travel through a wormhole, which is an unattended feature of our motion blur initialization.

Additional libraries

The following libraries were used for development:

- GLFW http://www.glfw.org/
- GLEW http://glew.sourceforge.net/
- glm http://glm.g-truc.net/
- FreeImage http://freeimage.sourceforge.net/

- Assimp Loader Library http://assimp.sourceforge.net/
- FreeType http://www.freetype.org/index.html
- DevIL http://openil.sourceforge.net/

Tools used to create the models

Game assets have been created in 3D Studio Max and Maya from multiple primitives with separate materials, which then were combined into one texture and mesh and exported as `.obj` model data.