Diana Vysoka 01633081
Michael Landauer 11778912

RESURGENCE

The ultimate fight for interstellar survival.

Instructions

Your goal is to collect all the spaceship parts that are distributed randomly on the Mars surface, in order to rebuild the spaceship. However, it is not that easy because your space scaffander only has a limited level over energy that decreases with the time. On Mars, there are different types of minerals that can to a certain extent increase your battery level so you do not die. These are fluorophore and phosphorus, each having different values. Collect them but be cautious; the maximum level of energy is 100, that means if you collect all of them in the beginning, you will waste them and therefore ensure your death before getting back to Earth.

While you collect the spaceship parts, the day and night changes. It can happen, that you will end up in a shadow or in complete darkness due to the sunset. You can turn on the flashlight that is embedded in your scaffander. Everything has its perks and corks; having the flashlight wastes your battery faster. Use it wisely.

General

All objects 3D-Objects are made with Blender except the terrain and have a texture. For the illumination of the scene we use a directional light which is the sun. We also implemented a spotlight for the character to lighten up the scene when its dark.

Playable (11)

The game is fully playable. This includes that the submission can be started via the .exe file, it runs on 60FPS in the Vislab (AMD RX580) and there is an advanced gameplay.

3D Objects (6)

We only use advanced 3D shapes in the game that we have modelled in Blender. The only exception is the Sun that orbits around Mars, it is represented as a sphere. We have implemented a 3D geometry importer based on Assimp library[1]. There is plenty of different tutorials, for example at learnopengl.com[2], on how to approach this challenge, but none of them showed (for us) suitable way to implement it. After reading the documentation and understanding how assimp works, we wrote the importer that works under the assumption that the 3D objects we want to import consist only of one mesh. We decided to implement it this way because for us, it is not necessary to have multimesh objects.

Win/lose condition (3)

The player won the game after he/she has collected all the generated spaceship parts. The number of the generated parts is written on heads-up display, as well as the number of already collected parts. We currently generate 10 objects.

The player lost if he/she lost all the energy of the spacesuit before collecting all the parts. Each mineral adds a certain amount of energy to the spacesuit, after it's collected with E key. The battery level is also written on the HUD. The maximum battery level is 100, the game starts with 50. Please, note the special trick with the battery level deduction based on day time.

Intuitive Controls (2)

The movement and actions of the player are controlled in a standard way - **WASD** (front, left, back, right) + **Space** (jumping). Additionally, the player can pick up the spaceship parts with **E** and build a spaceship with **B**. It is important that the player brings the parts back to the spaceship and as soon as he/she is near to the spaceship, the spaceship can be re-build. The instructions for building are shown on HUD. Also the yaw and pitch of the viewing angle can be controlled with the **mouse**.

When the night and day changes, or the player stands in shadow, it comes handy that a player

can actually see what is in front of him. To solve this problem, we implemented a lamp that the player can turn it on and off with **F** key.

Intuitive Camera (2)

We decided to implement the first person camera. The movement of the camera is described in the part “Intuitive Controls”.

Textures (2)

All the 3D objects are textured. We used the standard way that we learned in ECG to solve this task. Some of the textures were converted from JPEG to DDS. Most of the textures come from the “Total Textures Repository”.

Moving Objects (2)

The spaceship parts and the minerals are dynamic objects, that means, the player can displace them. The player can either push them or, after the player picked up an object with E key, the object’s position changes with the position of the camera/player. Another moving object is the orbiting Sun around Mars (in reality it’s vice versa, but for the sake of the game we implemented it the other way around). The Sun simulates the daytime.

Adjustable Parameters (1)

We implemented the adjustable parameters using .ini file. There we defined a boolean value for full-screen mode, width and height of the screen, brightness multiplier which is then passed to the fragment shader and the refresh rate, which is currently set to 60 FPS.

HUD (4)

For heads-up display we decided to use FreeType library[3]. Our implementation corresponds to the restrictions given in TUWEL Forum, that requires the GPU part to

be written manually (meaning not to apply FreeType methods to pass data to GPU). To implement the feature, the learnopengl.com tutorial[4] was of great use. HUD shows the battery level and the amount of the collected part vs. generated parts.

PhysX (12)

Proper interaction and collision detection are of high necessity in our game. That's why we decided to implement physics using PhysX library from NVIDIA[5]. We have two types of PhysX game objects: static actors and dynamic actors . Our camera is defined as a character controller. The interaction between the camera/player and the dynamic objects (e.g. Spaceship Parts) is done through raycasting. With the help of raycasting, we are able to detect so called "hit" and pick up the object. This applies for building the spaceship as well - we figured out a simple workaround of triggers - we detect a hit with a static object (spaceship) instead. The terrain (heightmap texture) is also imported as heightfield using the PhysX cooking process.

Video textures (8)

We implemented the video textures on a monitor spaceship part which is supposed to be collected by the player. We currently load 100 frames that simulates a broken monitor screen. The current implementation changes the video frame after each 30 render frames. After one cycle is done, the video texture starts from the beginning. The frames come from the site GeekTyper[6], we recorded the screen and then converted the single frames to a *.dds file.

Cel shading (4)

Cel shader is applied on all the objects except for the Mars surface. We calculated the dot product of the fragment normal and the light direction, to figure out what is the

intensity of the light. Then we compared the actual value of the intensity with different thresholds and changed the color with different multipliers.

The Mars surface is illuminated with the basic Phong shader. The reason behind that is aesthetics, as it did not look as good as expected with our basic Mars surface.

The only light source is the Sun. There is a white directional light placed in the center of it, that orbits together with the sphere geometry. When the night is simulated, we made use of the Sun being under the surface to illuminate the game objects so the player can see them better.

Tessellation from height map (12)

For our scene we decided to implement a terrain which is loaded in runtime. For this we generated a terrain via “Blender” and then captured these mesh as an texture/image. Our character has to walk on this surface, so we had to import the heightmap into PhysX, for this we used the heightmap.bmp, this image is rotated 90° CW and flipped vertically to the opengl heightmap. For the opengl scene we use a subdivided plane and then calculate the height from the heightmap_border.dds texture. This texture has an improvement that the outer pixel are black, so when the sun in goin up/down correct shadow values will be calculated. For the normals on the terrain we use a precalculated normal map which we generated with [8]. Most of the code is covered in MapObject and the shaders are marked with map. We were not able to match both terrain scene completed ident, so on some places of the map the collision detection is not optimal, these results that there are problems in the shadow calculation and this looks like “peter panning”. We wrote a post in the forum about this problem [9].

Shadow maps with PCF (16)

For our dynamic light source the Sun, which is a moving directional light, it is necessary to have shadows, to make the scene look authentic. To achieve this we implemented shadow maps which captures the whole scene with all objects

excluding sun, lens flares, particles and skybox. In the first step of the draw-function of the class “Scene” we capture all depth values for the specific points. For this process we wrote 3 different optimized shaders one for normal meshes, one for the

vertex animation object and one for the terrain. All 3 shaders for calculating depth are marked with the word “shadow”. After this process we adapted all relevant shaders to implement shadow calculation. We used the Tutorial on learnopengl.com [7] as a template for the calculation in the draw shaders.

Vertex shader animation (8)

First we wanted to implement aliens which attack the player, but we haven't enough time to implement the game logic for this, then we decided to generate and new spaceship-part which is “fuel”. For this we begin with a sphere geometry, which is then passed through a vertex shader, then to a geometry shader where the animation happens. We use a geometry shader to calculate the new normals with the cross product of the edges of one triangle. The problem is that this results in flat shading and doesn’t work fine with cel shading, so we didn’t touch normals in our process. The calculation for flat-shading is still in the geometry shader, but is not used. In the Labor one of the tutors said it is ok for the submission. The shaders which covers vertex shader animation is named with “animation”.

Particle System with Compute Shader (12)

We applied particle systems with compute shader on a sand storm. Following the repetitorium 2018, we had a solid fundamentals to implement it. For geometry shader, we also read learnopengl.com tutorial[12], which helped us understand how to use this shader program. For our implementation, we did not need velocities, as our particles (texture quads with dust) had fixed positions in the beginning and eventually we computed the new position based on delta time and radius, to achieve that these particles actually cycle. The count of particles in our case does not increase/decrease.

Lens Flares (8)

In order to implement lens flares, we first rendered the whole scene in a texture using frame-buffer object and color attachment. After this texture is rendered on

screen, additional draw function is called; the one for lens flares. We render a quad on the screen with a texture of lens flare, but before, we compute its position on CPU in every frame. As we know the position of the sun, we normalize the position to normalized screen coordinates and check whether the sun is on the screen. In case its position is in interval of -1 to -1 for x and y, the direction from the center of the screen to the sun is computed. Then, we compute the actual texture coordinates, as with the sun position change, the lens flare texture should flip either vertically or horizontally. Eventually we draw it with these texture coordinates. For this we used two tutorials[10] [11], but eventually we figured it out ourselves, as they were not solving the problems we had.

Resources

[1]http://www.assimp.org/ last access on 6.5.2019

[2]https://learnopengl.com/Model-Loading/Assimp last access on 6.5.2019

[3]https://www.freetype.org/ last access on 6.5.2019

[4] https://learnopengl.com/In-Practice/Text-Rendering last access on 6.5.2019

[5]http://gameworksdocs.nvidia.com/PhysX/4.0/documentation/PhysXGuide/Manual/Index.html last access on 6.5.2019

[6] http://geektyper.com/ last access on 06.05.2019

[7] https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping last access on 19.06.2019

[8] https://cpetry.github.io/NormalMap-Online/ last access on 19.06.2019

[9] https://tuwel.tuwien.ac.at/mod/forum/discuss.php?d=143701 last access on 19.06.2019

[10] http://john-chapman-graphics.blogspot.com/2013/02/pseudo-lens-flare.html last access on 19.6.2019

[11] https://www.informatik-forum.at/forum/index.php?thread/112939-lens-flares-effekte-allgemein/&postID=841376#post841376 last access on 19.6.2019

[12] https://learnopengl.com/Advanced-OpenGL/Geometry-Shader last access on 19.6.2019