Spacescape Documentation

Gameplay:

3D Geometry:

There are different 3D-Objects in the scene which have all been created and uv-unwraped using Blender. The models are loaded by the class OBJLoader.cpp which we implemented ourselves. It reads the vertex positions, normals and uv coordinates and returns an GeometryData object which is a struct we used from the ECG-Framework. Further the GeometryData is then given to a Geometry object which is also from the ECG-Framework and then rendered. Other Geometry like the quad of the post-processing or the cubes of the particle system are generated separately in their respective classes without using anything from the ECG-Framework.

Playable/Advanced Gameplay:

The Game is completely playable. The game logic is handled by game.h which calculates positions and velocities of the objects as well as handles player input. The goal of the game is to reach the open space and pass through the packages without colliding with them. Further there is a fuel resource which must not be 0 otherwise it is game over. Fuel can be restored by collecting stars.

60 FPS and Framerate Independence:

The current frame duration is around 7ms and all logic calculations are done in respect do the dt value which lets us achieve Framerate Independence.

Win/Lose Condition:

A Player either loses if he runs out of life or if he runs out of fuel. Once a life is lost it cannot be restored. However, fuel has to be restored by collecting stars in order to make it to the end. The player wins the game if he manages it to maneuver through the objects to the end.

Intuitive Controls:

The controls are more detailed in the controls section of this document, but are very straight forward. They are simple WASD controls with shift for boost and the mouse to turn the camera depending on the angle the player wants.

Intuitive Camera:

There are two cameras implemented in the game. The player camera is a standard 3rd person camera which can be adjusted via the mouse and it is handled by cameraPlayer.cpp. The other camera is the debug camera which can be accessed via F3. Here it is possible to fly freely through space. The debug camera is handled by cameraDebug.cpp and has slightly different controls which are explained in the controls section of this document.

Illumination Model:

For the illumination model we used the light parts from the ECG-Framework. There is currently one point light source and one directional light in the scene. Both light sources are passed to those shaders which are sensitive for light, for example the pbr shader.

Textures:

The player rocket, packages, and stars have all images textures on them with mipmapping and trilinear filtering enabled. The skybox also has a cubemap image texture on it in order to function. Further we implemented a video texture on the barriers. For image textures we used the texture.h class from the ECG-Framework, however, for the cubemap texture and the video texture we implemented separate classes which are called cubemapTexture.cpp and videoTexture.cpp respectively. Further we used the material class from the ECG-Framework as a starting point for our development and implemented subclasses with slight changes regarding the texture they receive. There are now the subclasses textureMaterial which handles the image textures (e.g., for the rocket, packages and stars), cubemapMaterial which handles the skybox and videoMaterial which handles the video texture.

Moving Objects:

Moving objects in our game are the packages and the rocket. Technically every object could move, but because of gameplay reasons the barriers and the stars do not move. Movement vectors and positions are calculated by game.h according to the dt value for framerate independence.

Collision Detection:

The collision detection is implemented very simplified by just using bounding spheres, i.e. a radius around each objects position which represents the object. Hereby we also used a variable called _boundingsphereCenter which moves the center of the bounding sphere locally. Every bounding sphere was drawn and exactly adjusted to the models. Collision detection is handled in the update function of game.h. A player can collide with packages or stars or if he runs in one of the borders (top, down, left, right).

Heads-up Display:

There are two HUD elements which can be toggled by F3: hearts, which represent the remaining life of the player and fuel, which represents the remaining fuel (each fuel barrel represents 25 fuel). We used two textures, one with a heart and one with a barrel. These textures are then given on a cube which has no perspective transformations. We then blended the background away so that we are only left with the heart and barrel part of the texture. Shaders used for this are hud.vert and hud.frag.

Controls:

F1 – toggle wireframe mode (only possible if no post-processing is applied)

F2 – toggle backface culling

F3 – toggle between game and debug mode

F4 – change resolution to 1600x900

F5 – change resolution to 1920x1080

F6 – change resolution to default (according to value in settings.ini)

F9 – toggle HUD

If in game mode:

WASD - rocket movement (up, down, right, left)

Shift – for activating the rocket's boost

Mouse wheel – zoom in and out

Mouse input – move camera around rocket if left button is down

If in debug mode:

WASD – camera movement (forward, backward, right, left)

EQ – additional camera movement (up, down)

Shift – fast movement

Effects:

CPU Particle System:

A CPU particle system is implemented for the exhaust of the spaceship. The particle system uses instancing and is implemented in ParticleSystem.cpp. The according shaders to it are particle.vert and particle.frag. The particles positions are also changed according to the dt value and color is interpolated over lifetime. The ParticleData struct represents a default particle and the Particle struct represents each individual particle at any given time.

Video Texture:

A video texture was implemented on the barriers. The videos duration is about 4 seconds and has 100 frames. The texture is implemented in videoTexture.cpp. In main the dt time between each frame is summed and if the sum is greater than a certain number the texture loads the next frame with glTexSubImage2D in order to prevent a complete reinitialization of the texture.

Physically Based Shading:

Since none of the team members implemented a PBR illumination model in ECG, we implemented the Cook-Torrance model in pbrShader.vert and pbrShader.vert. The shader was mostly implemented using this and this reference. The metallic and roughness parameters can be adjusted via uniforms, currently they are adjusted in Spacescape::setSpecificObjectUniforms(). Also, when creating the material there is a material parameters vector (like in the ECG-Framework) given to the shader. Here the x coordinate of the vector adjusts the amount of ambient light, however the y and z coordinate do not adjust anything. This could definitely be implemented cleaner but we decided to use the same material for each object and just alter the material parameters in the drawing loop.

Bloom/Glow:

A post processing Shader was implemented which gets a framebuffer of the scene as texture and is then drawn on the main framebuffer. Further we used two output buffers: one buffer for the regular image and one buffer for the bright parts of the image which are then blurred. Both are then combined in postProcessing.frag in order to achieve the bloom effect. With using two output buffers on each object shader, we can define what elements of the scene shall have bloom. Currently only the particle system and the barriers have bloomed since they are the only emissive materials in the scene.

Music and SFX:

Added FMOD sound library. FMOD dll has been added to the project. Class was created using certain features of FMOD library. Continuous playback for music. The once-play feature has been set for sounds. Music and Sounds are used with a single 5-layer array and an enum set for sound. It is used by calling the getter function from Game.h.