
2nd Submission :: Documentation

Anna Frühstück :: 0626930 :: 066 · 932

Stefanie Prast :: 0727540 :: 033 · 532

Markus Rumpold :: 0525647 :: 033 · 534

# 1) Implementation Description

## 1.1 Architecture

The architecture of Lumis is based on a multi-tier architecture concept, as dependencies on other components are only allowed in one direction, from top to bottom. This being the case, the components are split into two basic parts: Rendering and Game Logic.

Transformations and rendering are performed by the class `RenderingPipeline` and the classes it is depending on. Both parts are being called by a game class, which controls the workflow of the main game loop.

We split up the development of our game into several projects:

| | |
|---|---|
| `SimpleLogging`: | responsible for Logging |
| `Rendering`: | responsible for all the matrix transformations as well as the Rendering pipeline |
| `Model`: | responsible for Model loading & storage |
| `Lumis`: | responsible for Game play, game loop & initialisation |

Architecturally we have separated the game logic from the rendering pipeline using controller classes.

We have implemented rotations using rotation matrices or, in some cases, quaternions, wherever they were necessary.

Input and controls are caught with the input catching methods defined by glfw. Movement is calculated time independently.

Camera movement – especially during the jumping sequence – was quite tricky to realize, but rotating the player object with quaternions and orienting the camera dependent to the forward vector of the player did the trick.

## 1.2 Complex Objects:

Loading complex objects has been realized by loading collada models (vertices, normals and texture coordinates) using the *assimp* loader library.

## 1.3 Camera

The descriptive data (position, initial viewing direction, FOV,...) of the camera is stored internally in the Camera object. This object calculates the transformation matrix needed to transform models in world space to camera space.

## 1.4 Game Objects and Models

There are two classes separating the data storage needed to draw a game object.
While the `Model` class stores the vertices, normals, texture coordinates and indices of the object's visual representation, the `GameObject` class contains the position, rotation and various other attributes of the object in the world.

## 1.5 Moving Objects

The player is supposed to be able to navigate the main character freely around a spherical planet; hence the object describing it contains references to the planet the player is currently moving around on.
The planets, which are parts of a planetary system, are moving on elliptic planes in space. The calculations needed for these position changes are performed by game play components and the results are then stored in `GameObject` objects.

## 1.6 Texture Mapping

Texture mapping for the planets and the characters is calculated with simple UV Mapping. As the objects holding the mesh data for the simple game objects we currently use are generated programmatically in our code, the appropriate texture coordinates are being generated in the process.

## 1.7 Lighting and Materials

Lighting is implementing using gauss lighting for point lights. We have multiple light sources in our scene as the Lumis bugs are emitting light.
The light consists of an ambient, diffuse, and a specular portion which are calculated and applied to the objects in the fragment shader.
For the simulation of different materials, several fragment shaders have been implemented. These fragment shaders differ through their spectral material colour and their spectral coefficient.

## 1.8 Controls

Player Controls have been implemented using the provided functions from the *glfw* framework.
The W A S D - keys are used to move the player character. The mouse can be used to tilt the camera to a certain angle.

## 1.9 Game Play

Game Play is parted from the rendering code by the architectural structure.
The player can move around the planet freely, navigating with mouse movement and the WASD keys. Using the Space key, the player can jump to other planets that are within his range.

The Lumis bugs are placed onto the planets surfaces randomly and disappear and reappear at random.

### 1.10 View Frustrum Culling

The perspective projection matrix used for frustrum culling has been created by using the glm framework, specifically the glm::projection method.

### 1.11 Transparency

Transparency has been used for the post processing effects (glow and depth of field).

### 1.12 Experimenting with OpenGL

We are using VBOs and VAOs for transferring model data (vertices, normals, texture co-ordinates) to the shaders.
FBOs are used for the various effects (glow, depth of field, shadow mapping)
A UBO is used for transferring lighting data to the shaders.

## 2) Effects

### 2.1 Glow

Glow is realized by using 4 render steps. The first step renders the scene to the depth map and to a FBO color attachment. The second and third step blur the image by a two step blur. The fourth step renders the blurred glowing images to the screen and blends them with the original picture.

### 2.2 Depth Of Field

The depth of field effect is implemented by using a 4 pass rendering process with a two step blur. First the image data is rendered to a color attachment and the depth values are mapped to -1 to 1 space with player position at 0. And then the blur strength is determined by the distance to the player object. The blurring happens in step 2 and 3 just as seen in the glow effect, and the final step blends the glow image and the depth of field blurred image to one output image.

### 2.3 Shadow Mapping (coming soon)

Shadow Maps are created in a multi pass rendering process. First the shadow maps are created by rendering only to the depth attachment of the FBO of the viewpoint of the light source. Then the map is projected into the camera space and by comparing the depth values the shadow value is generated.

## 3) Features

### 3.1 In This Version

- Freely moveable main character
- Ability to jump for main character
- Ability to collect Lumis bugs for main character
- Textured and illuminated Game Objects

- Multiple light sources (Lumis bugs emit light)
- Detailed moving planetary system with multiple planets
- Depth of Field effects
- Glow Effect
- Shadow Mapping (coming very soon)
- Frame rate independent update loop
- Background music & sound effects

## 4) Additional Libraries

- *glfw* OpenGL framework library (*glfw.org*)
- *glm* OpenGL Mathematics (*glm.g-truc.net*)
- *FreeImage* (*freeimage.sourceforge.net*)
- *assimp* Open Asset Import Library (*assimp.sourceforge.net*)
- *fmodex* Audio Toolkit (*fmod.org*)

## 5) Links to Implementation Papers

Glow:

*http://http.developer.nvidia.com/GPUGems/gpugems_ch21.html*

Depth of Field:

*http://developer.amd.com/media/gpu_assets/Scheuermann_DepthOfField.pdf*

Shadow Mapping:

*http://http.developer.nvidia.com/GPUGems/gpugems_ch12.html*

## 6) Tools used for Model Creation

Autodesk Maya