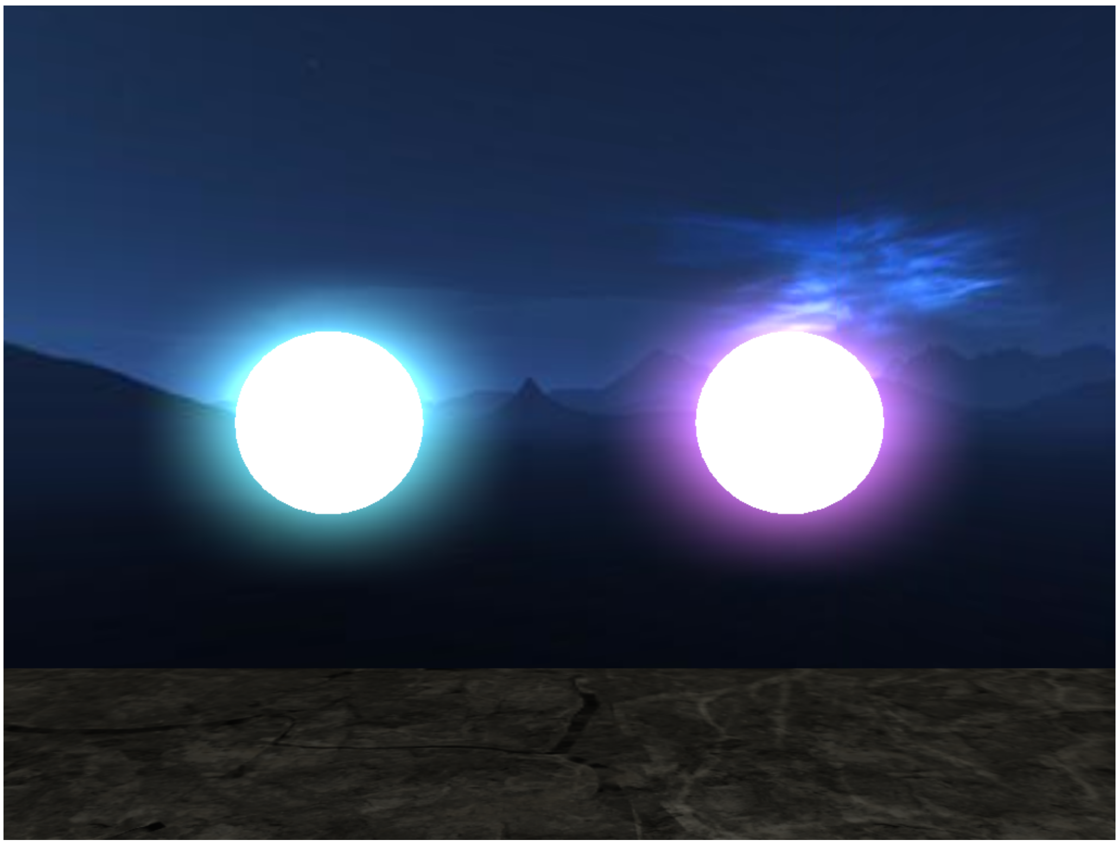

Still a better love story than twilight

RTR WS2017

Ulrik Schremser (00728034)
Patrick Mayr (01226745)

January 15, 2018

1 Remarks

The demo should work on AMD as well as on NVIDIA GPU. Just execute 'Journey.exe'. Settings can be adjusted in 'Journey.exe.config'.

Development Status: Plan.

2 Implementation

For the implementation, `C#` with `OpenTK` was used. For storing and using models, `GameObject` instances are used that hold instances of `Model` as well as key-frame animations baked into the `.dae` files the engine uses. A `Model` can have multiple meshes and is rendered by our rendering pipeline. To further instruct the shaders on how to process the meshes, each model has an integer used for bit flags, enabling the shader to decide on how to process a specific model (e.g. only render the texture instead of doing full shading).

2.1 Cameras & Movement

As mentioned earlier, camera and object movements can be baked into the scene via keyframe animations. This enables us to model the camera- and object paths directly in Blender. Depending on the scene we use either a single camera or two cameras and split screen to render the scene simultaneously for every one of the protagonists of our demo.

2.2 Texture Mapping

Every `GameObject` stores the `Model` which got imported from a `.dae-file` by using `AssimpNet`. `Model` is the representation of an imported model and holds e.g. the mesh data in `Mesh` for a regular game object. `Mesh` stores the textures and its vertex data using `Texture` respectively `Vertex`.

`RenderingPipeline` respectively its passes, which are subtypes of `AbstractRenderPass`, will finally pass the texture and vertex data to the shader to do the texture mapping. The vertex shader passes the UV data straight through to the fragment shader, which will get the color data from the texture per fragment using the `GLSL` function `texture(Texture, UV)`.

2.3 Simple lighting and materials

See Illumination and Texturing [4]

2.4 Controls

The controls for the demo are the following.

Key	**Effect**
ESC	quit demo

2.5 Effects

2.5.1 Glow

1. `RenderPass1` with its shaders `pass1.vert` and `pass1.frag` is creating the glow source texture and renders the scene in another color texture.

2. `RenderPass2` with its shaders `pass2.vert` and `pass2.frag` filters the glow source texture vertically using a 1x64 gauss kernel.

3. `RenderPass3` with its shaders `pass3.vert` and `pass3.frag` filters the glow source texture horizontally using a 64x1 gauss kernel.

4. `RenderPass4` with its shaders `pass4.vert` and `pass4.frag` is finally blending the color texture with the blurred glow source texture scaled by a glow intensity factor. Furthermore we using a max function on this effect to avoid that the objects center is brighter than the border. Finally we spread the excess of the color before clamping it.

2.5.2 Environment Mapping

In `RenderPass0` the scene is rendered using a so-called *CubeMapCamera*. That means, for each object which should get an environment mapping, the scene is rendered for every one of the principal axis +X, -X, +Y, -Y, +Z, -Z, by changing the view matrix according to the axis. For performance reasons the shaders `pass0.vert` and `pass0.frag` for creating the environment map are using only the diffuse texture information and no further effects or illumination. The resulting cube map is then used in *RenderPass1* with its shaders `pass1.vert` and `pass1.frag` to texture the objects, which should get an environment mapping, by calculating an reflection vector and using it to get the proper texture information.

2.5.3 Parallax Occlusion Mapping

For `Parallax Occlusion Mapping`, we need to transform the vertex, camera and light position to tangent space. Therefore, normals and tangents are provided by the model and the bitangent vector is calculated on the fly by using the cross product in `pass1.vert`. These vectors, then can be used to build the `TBN` matrix which is able to transform the mentioned positions to tangent space. In `pass1.frag` these positions are used to alter the original `UV` coordinates by performing parallax occlusion mapping. The diffuse color and normals from the provided normal map are then sampled from these new `UV` coordinates, which boosts a textured surface's detail and gives it a sense of depth.

In a nutshell parallax occlusion mapping works in the following way. The provided height map is subdivided in a number of layers. The view direction is then divided by this number of layers to get a delta to shift the `UV` coordinates every iteration. As long as the current layer depth is smaller than the sampled depth at this position, we move along the view direction and check for each layer if this condition is true. If the current layer depth is finally larger than the sampled depth at this position, we get the current and previous `UV` coordinates and also the depth before and after. These two depth values are now used to calculate a weight to interpolate the previous and current `UV` coordinates and finally return the altered `UV` coordinates.

2.5.4 References

1.	**Glow**	CGUE-Slides - Bloom & Glow GPUGems - Chapter 21
2.	**Environment Mapping**	RTR-Slides - Shading Learn OpenGL - Cubemaps Anton's OpenGL 4 Tutorials - Cubemaps
3.	**Parallax Occlusion Mapping**	RTR-Slides - Shading Learn OpenGL - Parallax Mapping gamedev.net - A Closer Look At Parallax Occlusion Mapping Sun & Black Cat - Parallax Occlusion Mapping in GLSL

3 Features

- glow effect (spheres)
- environment mapping (discord)
- parallax occlusion mapping (maze, floor)
- split screen
- multiple cameras
- multiple perspectives

4 Illumination and Texturing

Our game is illuminated by a simple Phong shading. The ambient light originates from one light source that stays and moves according to the location of the camera (one ambient light per camera).

Every GameObject stores the Model which got imported from a .dae-file by using AssimpNet. Model is the representation of an imported model and holds e.g. the mesh data in Mesh for a regular game object. Mesh stores the textures and its vertex data using Texture respectively Vertex.

If the imported model is a light source, there are no mesh data to store, but light data. This is done by using `Light` as a container. All data that are necessary for the rendering come together in `RenderingPipeline`, which will pass the data to the subtypes of `AbstractRenderPass` with its vertex and fragment shader `passX.vert, passX.frag` (X means shader 0 to 4) , which will do the actual shading.

5 Additional Libraries

1.	**Object-loader:**	AssimpNet (v.3.2)
2.	**OpenGL/OpenAL Wrapper:**	OpenTK (v.1.1.15)

6 Modeling

1.	**Modeling Tools:**	Blender
2.	**Models:**	Discord
		other models are self-made