YATCER

Description

YATCER is an Endless Runner set in Space. The player controls a Moon rolling along the rings of Saturn. Points are gained by collecting shiny cogwheels, but colliding with a rock ends the game.

Start the game by double-clicking on **YATCER.bat** You can also edit the 'YATCER.bat' file, to adjust screen size by changing the parameters after 'YATCER.exe' accordingly:

```
width height fullscreen
```

Controls

A, D	Move left/right
Space	Jump
Esc	End Game
P	Pause Game
R	Rotate Obstacles
F2	Frametime toggle
F3	Wireframe toggle
F6	Camera switch
F7	Toggle Normal Mapping
F9	Transparency toggle

Features

- endless gameplay
- reel-feel rocky obstacles are randomly placed
- shiny cogwheels are randomly placed
- additional free camera mode

- space feeling

Illumination

Only one directional light, coming from the sun, is used to illuminate the scene. All objects are lit. The cogwheels use a shiny material.

We use the Phong Illumination model, all lightening calculations happen in the Shader stage and depend on the fragment normals. Specular factors are determined by object materials and diffuse and ambient values set for the light source.

Effects

Normal Mapping

Normal Mapping is used to make the rocky surfaces of our obstacles look rockier and its edges sharper. First of all we needed to generate a normal Map with per-Fragment normals given in the RGB channel in Mudbox for our low-resolution mesh and a Collada File where next to UV-, Vertex-, and Normal-coords, all our tangets and bitangents reside. (The Tangents and Bitangents are calculated by ASSIMP!)

Since we only use Normal Mapping for the Obstacles, the stone-normal Map texture was generated directly in our Renderer class. Also a new Shader "NormalMap" has been written, which takes this normal map texture as an uniform. At the Vertex Shader Stage the provided vertex normals, tangents and bitangents were transformed into world coordinates and packed together to the notorious TBN-Matrix (Tangent, Bitangent, Normal) which is given to the frament shader.

If normal mapping is enabled, the normal (which is in tangent space) will be grabbed from the texture and by applying the TBN Matrix, is transformed into world space, where we can perform our usual lighting.

Shadow Maps

We implemented the use of one Shadow Map, since we only have one directional light source. For that we created a FBO with a depth texture (with the texture comparison Settings enabled for PCF) attached. In the first shader pass, with a little Polygon Offset factor of 4, we draw into this depthFBO the depth Values as seen from our light source with the LightSpace-Matrix – an orthogonal projection.

The depth texture is now correctly drawn and can be used for the second shader pass, where we draw our scene as always and actually calculate shadows. We use the previous LightSpace Matrix, with a little Bias of 0.5 (for UV lookup) and apply it to the vertex at the vertex shader stage, to get our vertex position in this light space and pass this light space position to the fragment shader.

While shading the fragment, we can use the texture which is sampled with sampler2DShadow (with this OGL calculates PCF automatically thank god) and perform a comparison of the depth in the Texture with the depth of our fragment in light space. From this comparison we get an output between 0 and 1. If it is 1, the fragment lies completely in shadows and only gets ambient lighting.

Bloom

To create a bloom effect, the scene is rendered into a FBO with a smaller texture size to speed up the process. The image is first split into two textures, the scene itself and an image of only the parts exceeding a brightness threshold. The texture containing only the bright parts is blurred horizontally and vertically by rendering it repeatedly in two FBOs. The blurred texture is then added onto the scene image to create a bloom effect.

Lens Flare

To create the lens flare effect, we first render only the sun and the saturn with a black texture into a seperate FBO. This creates a texture which is mostly black, except for the sun (if it is visible). By doing this, we avoid having to determine whether the sun is visible or occluded by another object, since a lens flare created while the sun is not visible will be black and therefore not visible on the final scene.

In the lensflare shader, we partly invert texture coordinates and use a few preset variables to create ghosts along a vector from the sun to the camera. We also create halo effects and use chromatic distortion (i.e. multiply the texture rgb-values with modifiers < 1) to make our flare effects prettier.

Submission Requirements

Complex Objects

Both the obstacles (rocks) and the goodies (cogwheels) are complex objects. There is an ObstacleLoader class that generates a VAO from the loaded collada files. Seperate obstacles and goodies are structs with a model matrix and a pointer to their vao.

Animated Objects

To implement animated objects, we added a small moon orbiting the player controlled moon, called Moonmoon. Moonmoon's position is calculated by first translating by a set distance (so that the moonmoon is not drawn inside the moon) and then rotating to create an orbiting effect. Then, we get the moon's position and translate the moonmoon accordingly, so that moonmoon correctly orbits the moon.

Transparency

The rings of Saturn the moon is rolling on are transparent. To achieve this, we created a material with an alpha value of 0.5. We avoided most possible issues with transparency by ensuring that the transparent rings are always rendered after all other objects in the scene.

Tools used to create Models

We used Blender, Mudbox and Maya to create our object models.

Additional Libraries and References

- assimp – for loading object models
- freeImage – for loading images
- glm – for handling matrices and vectors
- freetype – for displaying text
- glfw – for window creation and input handling
- https://www.opengl.org/wiki/Framebuffer_Object_Examples - fbos
- http://learnopengl.com/ - for a lot of stuff: displaying text, fbos, normal mapping, shadow mapping, bloom, skybox
- http://www.c-jump.com/bcc/common/Talk3/OpenGLlabs/c262_lab10/c262_lab10.html - skybox
- http://www.tomdalling.com/blog/modern-opengl/05-model-assets-and-instances/
- http://john-chapman-graphics.blogspot.co.at/2013/02/pseudo-lens-flare.html – lens flare
- https://github.com/daanvanhasselt/lensflare – lens flare
- http://www.tomdalling.com/blog/modern-opengl/06-diffuse-point-lighting/ - for lighting

- http://www.mbsoftworks.sk/index.php?page=tutorials&series=1&tutorial=11 – for lighting