Robots on Ice

2nd Submission Documentation

Gameplay

The goal of the game is killing all enemy robots by shooting them with the laser. The enemy robots are usually moving randomly between points. When they are near the player they start pursuing him. The enemies differ in their properties (speed, health, damage) to keep it interesting. The game is lost if the player robot has no more energy. Energy will be lost by driving in water or getting too close to the enemy robots. The amount of energy is indicated by brightly colored boxes. Green and glowing means full energy, red means dangerously low and grey means death. The laser can overheat when shooting for too long, which means it will be disabled for a short amount of time to cool down. To make it even more difficult the water level is slowly increasing over time.

Controls

The player controlls a robot that can be moved with WASD. W is forward, S backward and with A and D the gear can be rotated, changing the direction. The turret direction is controlled with the mouse by moving it around the robot. LMB shoots the laser; the camera is rotated by dragging the RMB. Quit the game by closing the window or pressing ESC. Toggle debugging info in the window title by pressing F1. An additional config.txt file in the root directory can be used for enabling fullscreen mode, changing the window size and hiding the cursor.

Effects

Water + Reflections (1.5P)

The scene is first rendered from a reflected viewpoint to an additional full-size FBO. All fragments beneath the water level are discarded in the shader, otherwise objects under water would be visible in the reflection. The reflelection map is then simply sampled in screen space. A normal map with a time dependent offset is used for the ripple effect. The transparency depends on the angle between view vector and water plane. Since our game has a top down view we didn't have a skydome inditially, but we added one in order to improve the water effect. No specific external resources were used for this effect.

Shadow Mapping with PCF (1.5P)

A single 2048*2048 pixel shadow map is used for the whole scene. Additionally the rendered section is moved with the camera and scaled according to the far plane distance. That way the shadow map resolution is distributed accross the visible area only, resulting in very crisp shadow edges. This makes the results of PCF actually quite hard to see. Shadow casting and receiving can be enabled or diabled per object. For example, the skydome does neither cast nor receive shadows. The terrain is also excluded from the shadow casters, since the light comes in quite steep and the resulting shadows had ugly artefacts.
http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/

Bloom (1P)

We implemented bloom as a four-pass post-processing effect: first the bright areas are extracted into a half-size FBO. This FBO is then blurred horizontally and vertically using a Gaussian kernel. After that the blurred brightspots are combined with the original scene with additional adjustments to saturation and brightness.

Bloom is clearly visible on the terrain on triangles directly facing the light. Even more so it is visible on emissive materials, like the health indicators of the robots. Move into a darker area for the clearest results.
http://digitalerr0r.wordpress.com/2009/10/04/xna-shader-programming-tutorial-24-bloom/

Other Requirements

Besides the points listed here the whole requirement set of the 2nd submission is met as well.

Complex Objects

All objects in the game are either generated procedurally or loaded from Wavefront OBJ files. Besides simple planes and boxes our game features these objects:

- skydome (loaded, visible only in the water reflections)
- enemy robot (loaded)
- terrain (procedural, generated from heighmap)

Animated Objects

Our game implements hierarchical animations, clearly visible on the player controlled robot. The turret and laser of the robot move relative to the gear.

View-Frustum-Culling

For all meshes we calculate bounding spheres for determining visibility. The frustum volume is extracted directly from the view-projection matrix and intersections are calculated in world space. Our culling system further supports any kind of translation and rotation and uniform scaling.

Transparency

Transparency is vital for the realism of our water effect. Similar to real water a steeper view angle results in higher transparency. In order to have correct visibility the water is rendered after every other object. For shadow mapping to work correctly with the semi-transparent plane an additional mechanism for disabling shadow casting and receiving had to be implemented.

Experimenting with OpenGL

We implemented all required debugging features. The output of frame time and current debugging states happens in the window title. We further experimented with VBOs, used for every object in the game and a VAO for the fullscreen quad for shadow mapping. Since VAOs required tight coupling of meshes and effects we decided to not use them extensively.

Lighting and Textures

Everything is lit by a single directional light source. Our materials feature an additional emissive channel for self-illumniation effects. Textures are used extensively for the terrain, robots and the sky.

External Libraries and Tools

GLM, GLFW, GLEW, EasyBMP (for image loading),
3ds Max (Student Edition), Photoshop CS4 (Student Edition)