Documentation

Advanced Computer Graphics IM16

Members

Antonia Christina Haslinger	S1610629007
Michael Alexander Staudinger	S1610629021
Jan Hanghofer	S1619002069

System

We started to set up our project using WebGL, but soon we realised major problems and switched to OpenGL. We used the editor Visual Studio 2015.
For our main-setup we used the tutorials from: learnopengl.com which was afterwards cleaned and edited by Michael and Antonia. Jan used a parallel working work-space.

Effects

Haslinger: Particle Effect Rain
Staudinger: Animated Water Surface
Hanghofer: Screen Space Ambient Occlusion

Problems

WebGL

For our chosen effects there were simply too less information about it.

System

Could not find terrain-model which contains expected data, included self-made terrain which won't use the texture we found.
Problems with shader-buffers (still not working for rain and animated water).

Particle Effect

The shader-buffer-problem is affecting this system so it can't be drawn, but the calculation of raindrop-positions works properly (see particle_data.mp4).

Animated Water Surface

Due to time-limits and heavy work of trying to fill and bind the shader-buffers there was no time for programming a wave-spread, but just a function for starting "swinging" certain sub-planes. The shader-buffer-problem is affecting this system so it can't be drawn, but the calculation of the "swinging" works properly (see water_swinging_data.mp4).

Screen Space Ambient Occlusion

At the beginning of the project phase, we have done a state of the art research about real time Global Illumination. All papers referenced screen space ambient occlusion as a base. It was hard to reimplement the papers because no detailed implementation were mentioned. There were problems with how to orient random vectors into the hemisphere for checking the occlusion factor. It was unclear how to handle FBOs and multiple output data form the fragment shader. We still do not have a solution for multiplying screen spaced ambient occlusion with the lightning model. Pixel coordinates were not the same as screen coordinates. Blurring the SSAO texture for smoother results.

Solutions

System (main.cpp)

The basic system was by LearnOpenGL. We added and changed functions to the base which we need:

- Function for Camera position/rotation setting
- Resulting function for smooth/animated/faded camera movements
- Simplified model and shader importing
- Simplified model drawing

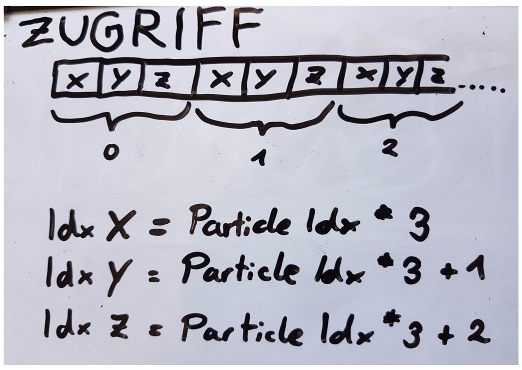
As initialization step

- GLFW is initialized
- GLEW is initialized
- Shaders are loaded
- Models are loaded
- RainSystem is initialized
- AnimatedWaterSystem is initialized and set for a simple swing which lasts 5 seconds (just for a technical check because rendering wouldn't work because of buffering-fail)

During the game-loop

- Keypress-events are checked (ESC for close)
- Scripted camera-pathing is calculated
- Models are drawn
    - The terrain-model is hand-made but still can't use the texture shader, so it's a simple green shader which we couldn't combine with SSAO in time
    - We used a texture-shader by LearnOpenGL
    - The other shaders for rain and animated water are just simple shaders as well, because we couldn't check them because of the mentioned buffering-fail
- Animated water behaviour is updated (not shaded though)
- Rain behaviour is updated (not shaded though)

Particle Effect (RainSystem.cpp)



To create our particles we initialized GLfloat array and filled it with position-attributes of the particles. For accessing the particles functions are defined which transform the particle-number to the array-index.

Before this approach we created a struct Particle which saves it's own position and velocity. Because we thought our shader-buffer don't accept this setup and it would lead to performance issues we changed this setup to the one described above which increased the frame rate quite noticeable.

Every frame the new y-position of the particles is calculated. Once a particle drops below a given y-coordinate the particle will be reset, so it will generate a new random position in the scene including a new start-height.

Animated Water Surface (AnimatedWaterSystem.cpp)

Parameters

The AnimatedWaterSystem is parameterized, so the following settings can easily be changed:

- Center of the plane
- Amount of subplanes which will "swing"
- Size of the subplanes
- Amplitude of water-"swing"
- Speed of water-"swing"
- Fade-factor for "swinging"

Initializing

When creating the AnimatedWaterSystem the grid of sub-planes is built. The resulting data is linked to the vertex-shader-buffer. The positions of the sub-lanes are manages in a similar way to the rain-system. To keep access to the sub-planes there are several methods who converts plane-coordinates into array-indices.

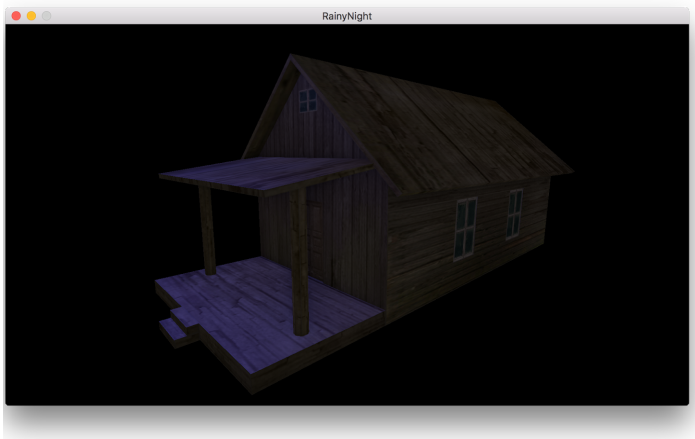
Updating

In each update updateSwinging(...) and drawWater(...) should be called.
UpdateSwinging(...) calculates the y-coordinates of several sub-planes which should "swing". To tell a sub-plane that it should "swing" there is a function startSwinging(int x, int z) which sets variables that way, that updateSwinging(...) will calculate and set the y-coordinate referring to the recent swinging-time.
For the calculation we worked with a sine which has the right frequency for the given speed-parameter. This sine will be multiplied with an amplitude-factor which decreases over

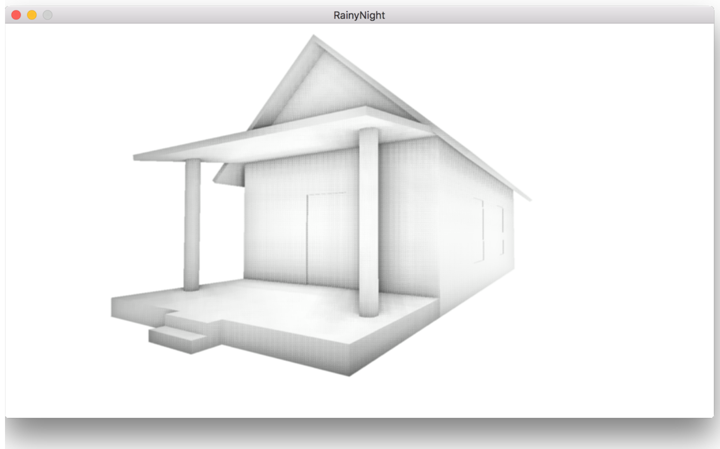
the swinging-time. The result will be added to the neutral y-coordinate-value. This results in a movement which should be achieved by throwing a stone into water - the water will swing up and down. But the affection of near sub-planes couldn’t be done in time because on heavy tries working with the shader-buffers.

Screen Space Ambient Occlusion

In order to get fundamental knowledge of screen spaced ambient occlusion we decided to do related online tutorials from https://learnopengl.com. First, we rendered our scene into depth and color buffers attached to an FBO. Along with this, we rendered the normal at each fragment and its linear depth in view space into a second color attachment on the same FBO. We rendered a full-screen quad with our ambient occlusion shader. The shader reads the depth value that calculates the occlusion value for each of the generated fragments. We generated a sample kernel in tangent space, with the normal vector pointing in the positive z direction. We added additional SSAO textures for blurring the result.



Lighting without SSAO



SSAO