THE AVALANCHE

Katharina Pescoller (1526648)
Felix Ginzinger (1527276)

January 19, 2020

Imagine what it would be like, if, instead of being afraid of the avalanche – you were the avalanche! All those annoying alpine guides and tourists you usually meet during your ski tours, they will no longer be in your way. Actually, they will be – but this time, you don't try to avoid them – you want to hit them. The more people you bury under your growing snow masses, the better for your score. But beware! You are not immortal. People were smart enough to build up barriers in order to stop you. And also the glacier, your home, has turned against you. Due to the climate change, more and more crevasses lurk under the cover of snow awaiting you to pass innocently over them and drag parts of your precious snowy body down to the depth of the mountain...

1 Story

The demo starts with a plain snowy terrain and little by little snow particles are accumulated. They rise, spiraling upwards in the sky. To get a nice overview of the impressive glacier the camera shifts a little bit to the left, then a little bit to the right - stopping once again looking straight at a wild skier, who appears out of nothing. The skier seems furious, sliding fast, gaining speed, aiming straight at the poor avalanche who was about to form out of the snow spiral. And then they collide. The world seems to stop in time. A big explosion takes place, all particles being thrown into the sky, slowly remedying their direction, being forced by gravity back down on earth. The skier explodes into thousands of millions (only twenty, but there's nothing wrong with exaggerations) of smaller skiers, likewise being hurled in the sky, rotating wildly before they crash with the glacial ground. The world starts to rotate around us, we can not think or see clear anymore, everything blurs as we gain more and more rotational speed. Eventually the speed is too much. A black screen appears, telling us, that it is over.

2 Final Submission

Brief description of the implementation, in particular a short description of how the different aspects of the requirements (see above) were implemented - be specific, but keep it concise.

3 Additional Information

1. We use one single directional light source (the sun) which affects game objects (cook torrance model) and terrain (phong).
2. Additional library: PhysX 4.1 for collision detection (https://developer.nvidia.com/physx-sdk)
3. Implemented object loader inspired by (http://amin-ahmadi.com/2017/01/04/how-to-read-wavefront-obj-files-using-cqt/)
4. We downloaded the objects from https://www.yeggi.com.
5. With ESC key the demo window can be exited early
6. The demo was tested on an NVIDIA graphics card.

4 Our Effects

How you've implemented those Effects (Links/References to papers, books or other resources where the effect is described and a description of your extensions to it)

4.1 Motion Blur

The motion blur is implemented, by calculating the motion vector between two frames and sampling the color buffer along this vector. In our implementation it is possible to enable the motion blur for each type of object we are rendering individually (Background, terrain, Gameobjects). The motion vector is calculated in the shader of the objects and it is rendered into a 2D-buffer of the frame buffer. It is then passed as a texture to the upscale shader (used already for the lensflare effekt), where all the post-processing is handled and the processed color buffer is then render on a cube. We used the turtorial http://ogldev.atspace.co.uk/www/tutorial41/tutorial41.html as starting point for our implementation.

4.2 Omnidirectional Shadows

For the shadow map implementation we used the lecture slides (creating 6 view matrices for the sun, mapping that to textures, ...). We have a single sun. Shadows are only implemented for the skiers and the rocks, as it would require an additional render pass if we wanted to implement them for the particle system aswell.

4.3 GPU Particle System with Compute Shaders

The particle system is implemented using compute shaders. There are two different update shaders for updating and computing new positions of new particles and the rendering shader.

1. *Update Shader 1* – The upwards growing cylinder is computed with using a parametrized representation of a cylindric cone curve. Random particles are spawn within a defined spherical shell range of the emission center (0, 0).
2. *Update Shader 2* – The explosion is simply simulated taking the current position of each particle with respect to the emission point (0, 0) as its initial velocity and the application of gravity. Random particles are spawn within a defined spherical shell range of the emission center (0, 0).
3. *Render Shader* – The rendering is inspired by [1]. They introduce a very nice billboarding technique where the normals at each position on the billboard are computed such that the particles actually look like spheres.

For the randomness within that implementation (new particle positions, random initial directions, ...) we used random values generated by the CPU. The GPU accesses them via **Bindless Textures**.

4.4 Additional Effects

4.4.1 Lens Flare

We read the blog http://john-chapman-graphics.blogspot.com/2013/02/pseudo-lens-flare.html. Our implementation considers sun location as an extension to the one presented by John Chapman, i.e. halos and ghosts are formed around the sun location instead of the center of screen and also they are only shown when the viewer looks straight at the sun.

4.4.2 Tessellation of Terrain and

For the terrain-Tesselation we read the follwoing tutorials:

- http://codeflow.org/entries/2010/nov/07/opengl-4-tessellation/
- http://ogldev.atspace.co.uk/www/tutorial30/tutorial30.html and
- http://ogldev.atspace.co.uk/www/tutorial31/tutorial31.html

In our implementaion the amount of generated polygons depends on the distance to the camera. The Tesselation is done by two newly added shader stages: the Tessellation Control Shader (terrain.cs) and the Tessellation Evaluation Shader (terrain.es). We distinguish between three different distances to the camera / level detail which will be drawn. In order to change the height of the newly generated polygons, which is needed in order to gain a better impression of the terrain, a heigthmap-texture is loaded to the Tessellation Evaluation Shader and the new height value of the polygons is extracted out of it.

4.5 Procedural Texture:

Since we weren't really happy with the texture of the terrain we decided to use a procedural texture for the terrain in order to simulate a terrain which looks like an glacier environment. For this we added three functions to the Fragment shader of the terrain: one which generates random numbers, one that generates Perlin Noise (generates a texture out of adding multiple times Perlin Noise with different parameters and one time the noise function in order to apply a little bit the snow-like structure). The Perlin Noise is generated one time out of the xz-coordinates of the terrain to gain the nice cloudy structure and one time only with the height information in order to get fine lines along the steeper parts of the terrain hills. We read through the following webpages for help:

- http://www.science-and-fiction.org/rendering/noise.html for random numbers
- https://gpfault.net/posts/perlin-noise.txt.html for Perlin Noise

4.5.1 Physically Based Shading (Cook-Torrance)

We implemented the illumination model based on [2] for skiers and rocks. The web page http://www.codinglabs.net/article_ph helped us with it. We only consider a single directional light (our sun). The file material.dat is provided and contains materials which are used for rock and skier.

References

[1] Paolo Cignoni and Claudio Montani. Ambient occlusion and edge cueing to enhance real time molecular visualization. *IEEE transactions on visualization and computer graphics*, 12:1237–44, 09 2006.

[2] Robert Cook and Kenneth E. Torrance. A reflectance model for computer graphics. *ACM Trans. Graph.*, 1:7–24, 01 1982.