Implementation of deferred snow deformation technique used in the Rise of the Tomb Raider

OpenGL context setup and camera

Setting up the context and simple rendering pipeline was rather non-trivial task, mainly because small mistakes often don't crash the program, but result in a black screen. The problems I have probably spent most time were two: one was an VertexArrayObject - OpenGL is a state machine, and this object can actually stores the current state of the context-bind buffers, vertex attribute pointers and other stuff. It is quite easy to switch them all by just swapping the VAO. However, I didn't know that it is required to create at least one such vertex array object, and this requirement come with the version 4.0, so it is not mentioned in older tutorials. Even worse, not setting this object up results in black screen, with no error message.

Second problem was camera. Due to the way I implemented it, the camera can never look exactly down (which is { 0, -1, 0 }). In this orientation, there is some singularity in the math, and the result is again black screen. However, this orientation was also my initial orientation, and it took while to spot it. After the struggle, movable camera, simple shader program and nice (so far) code structure layout were implemented.

Terrain

I was choosing between procedurally generated terrain (some simple pesudorandom generator with smoothing to generate vertex height in vertex shader) or a heightmap loaded from the disk. I have decided to use the heightmap, because it is arguably easier to implement, I can use the heightmap texture in multiple shader programs without having to copy the code and I have more control over what the scene will look like.

After deciding that, I soon figured out that loading images into OpenGL textures is not so simple task as I thought. I also didn't like the idea to link more external tools for this into the project, only to start learning how to use them. Luckily, I discovered that GIMP program has an option to export image into .c source file, which contains structure with a long string of image data in char format. While it is an odd solution, it is super simple to just include the file and load the image data into OpenGL texture.

I also created a grid mesh for the terrain vertices, and in the vertex shader I sampled the heightmap and set the vertex height accordingly. Then I improved the shader program with tesselation. This was quite easy to do, since there are plenty of tutorials and the tesselation evaluation shader is quite similar to the vertex shader I already had.

Dynamic objects

There were several options for the next step: I needed a snow mesh on top of the terrain, dynamic objects that will deform the snow and deformation texture created from the objects.

I have decided to create the objects first, since their form might have consequences on the other parts. For the shape of the objects, I found simple code snippet that uses latitude and longitude iterations over the rings to create a sphere.

I used instanced rendering, so I created SHADER_STORAGE_BUFFER_OBJECT (SSBO) that contains info about each sphere - its position in world, size and color so far. Then I rendered single sphere of unit size many times. During instanced rendering, I sampled the bound SSBO based on the gl_InstaceID variable, and change the properties of the vertex based on the sphere it belongs to.

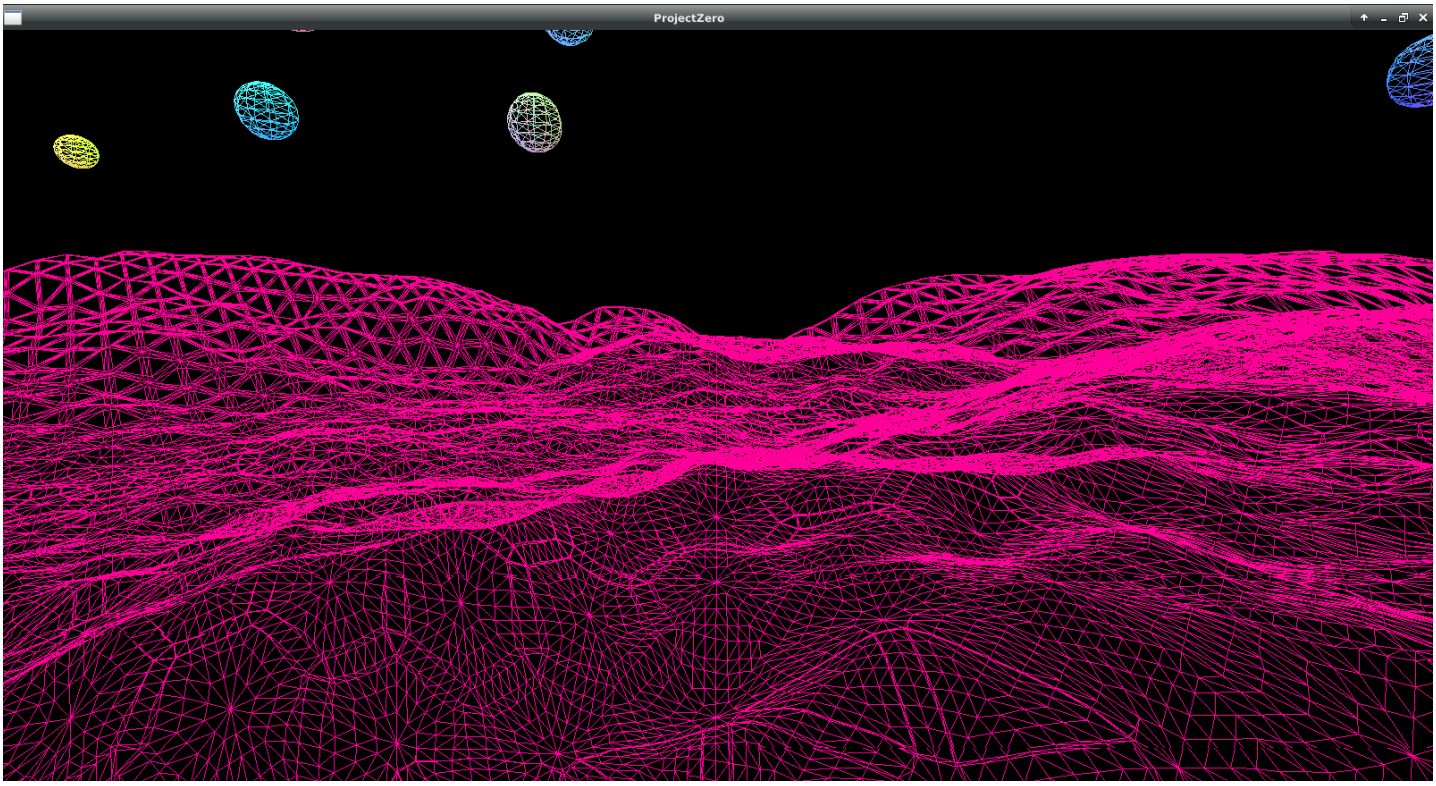
There I had another quite ugly bug. The glm math library I use for vectors and math should be configured to work with OpenGL flawlesly, yet when I loaded the data into SSBO and then sampled them in vertex shader, I got very strange results. After a quite long time of searching in the code and on internet, I found out that there is a problem with memory alignment. Finally I solved it by changing the structure definition in the vertex shader that was supposed to represent the data in the buffer, to contain only floats instead of vec3 types. I have read somewhere that OpenGL requires alignment to positions divisible by largest structure type, so for floats it was only 4, which finally worked with glm::vec3 type.

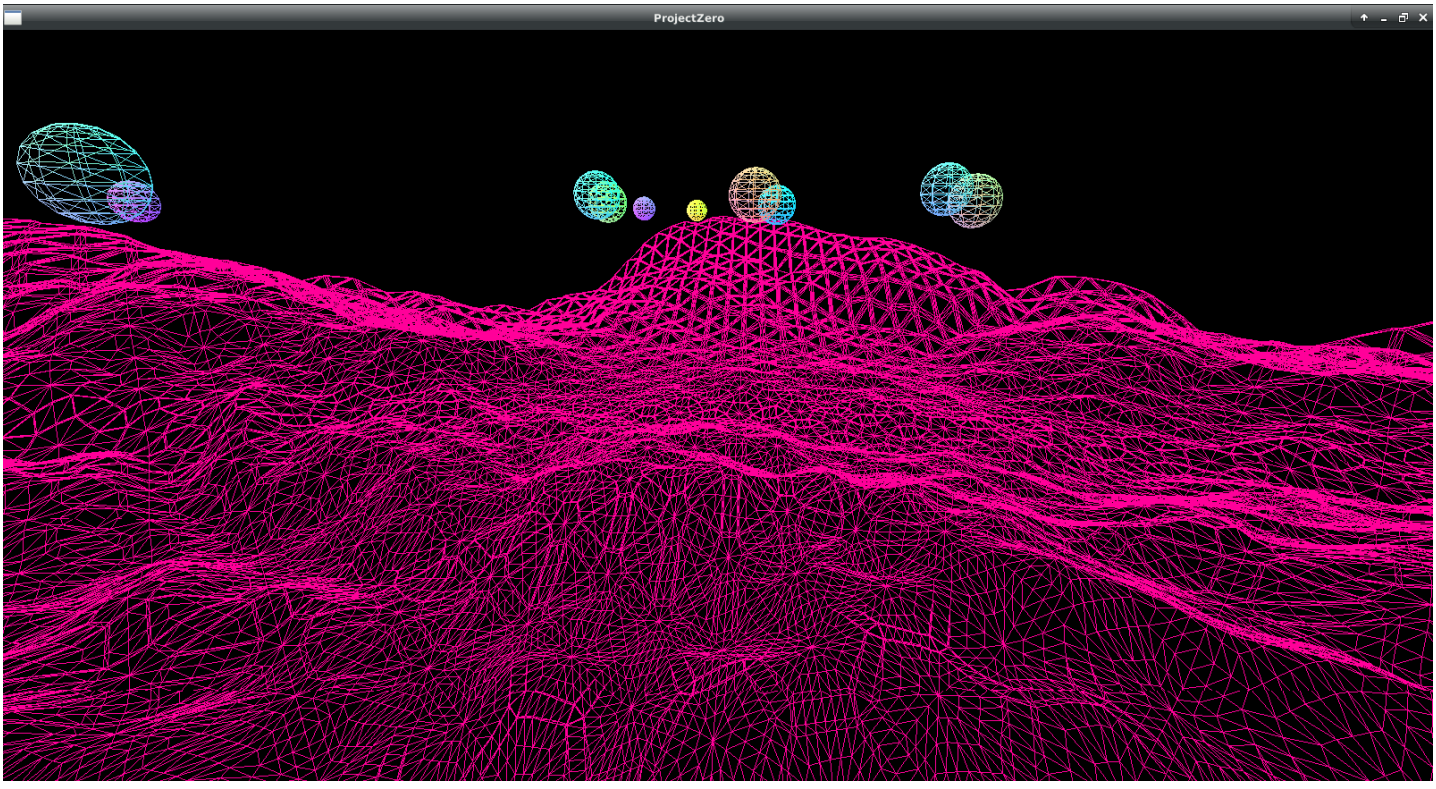
Compute shader

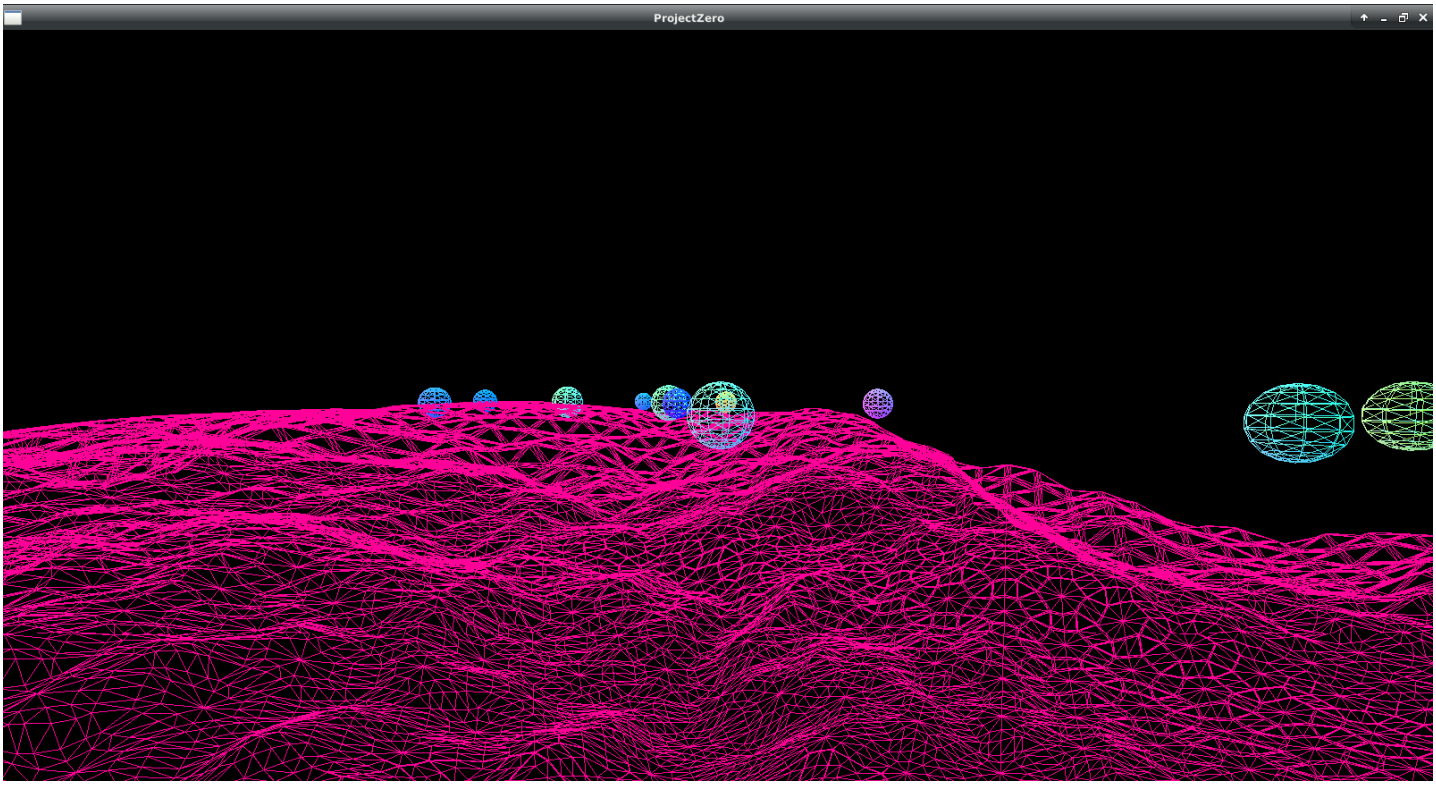
The next thing to do is the compute shader, that will compute the deformation texture from the dynamic objects. With my approach, I just need to bind the SSBO with the sphere data into the compute shader and I will get access to all relevant info about dynamic objects almost for free. I have also decided to implement the simple physics stuff in the compute shader. Since SSBO can be also written into, I can compute the gravity and terrain clamping for each sphere there.

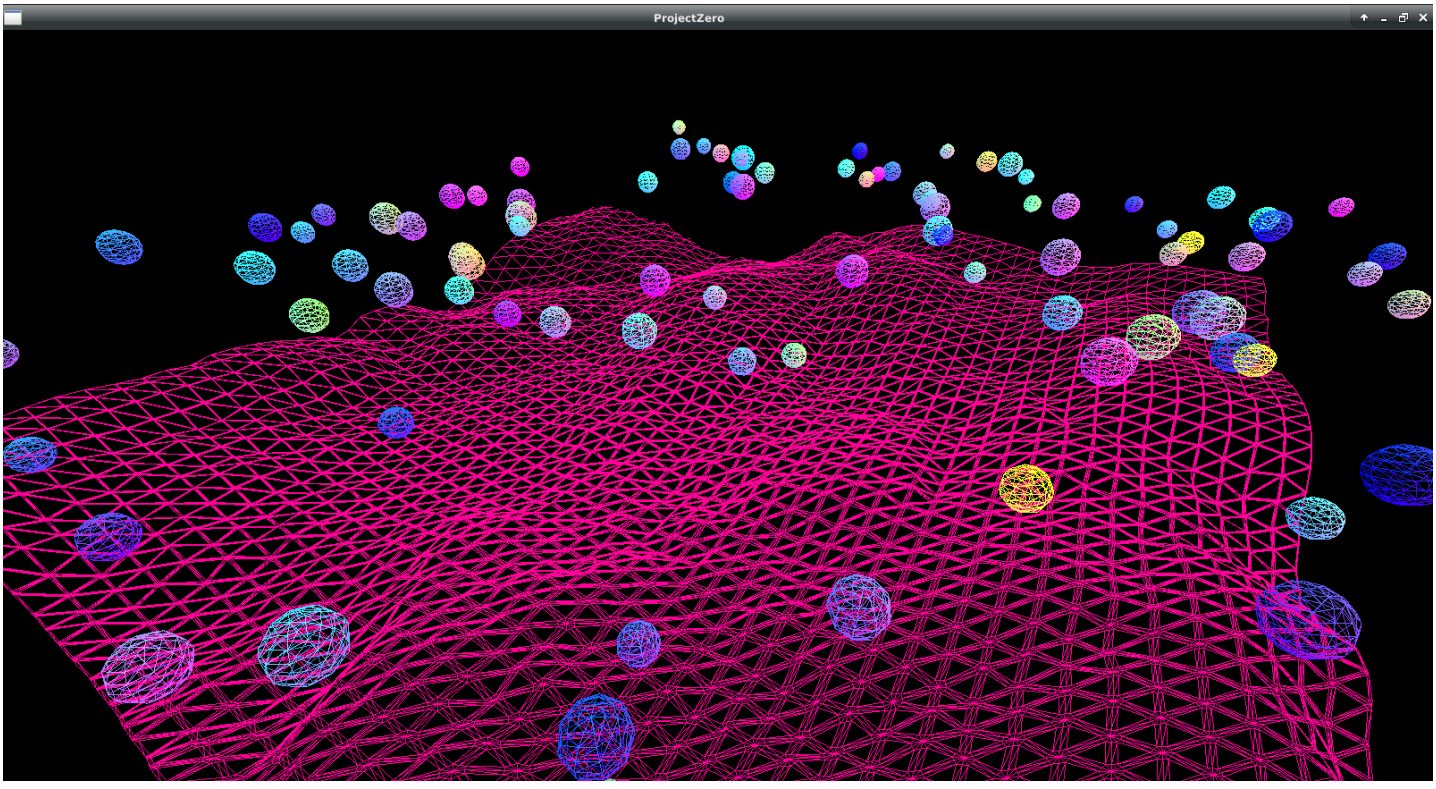
Before that however, I had to learn how the compute shaders work and how writing to texture works, since I knew there will be problems. First thing I decided to implement was gravity effect on the objects - just simple falling downwards. I got it working finally, after some struggle with the compute shader compilation - it requires its own shader program, compilation and linking.

The next step will be to implement the computation of the deformation texture and use it for snow mesh rendering, which are the core features of my chosen technique. There might be pitfalls with the write access to the texture, because the technique requires atomic access. The technique also actually stores two values into single uint64 type, which can be another problem.


ProjectZero


ProjectZero


ProjectZero


ProjectZero