TDT4230 Spring 2018, Final Project Report

Håkon Flatval

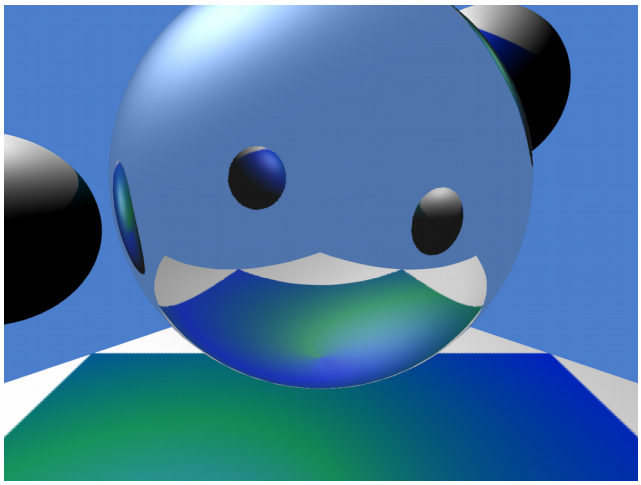
Introduction

For my final project, I have chosen to implement a modifiable reflective ball. The ball is placed in an environment of other objects and will reflect the environment to the user, creating an illusion of being made up of some mirror-like material.
In addition to this, the ball takes its surface normals from a normal map, which can be modified. The original map is stored within the delivered file tree, and may actually be modified by hand before program execution if so desirable. However, within the program itself, I have allowed for a point-and-click method of modifying the ball. That is, looking at a point at the ball's surface (keeping it in the middle of the screen) and pressing space will create a small valley (only on the texture, the mesh is unaltered) at the point of focus, making the normals bend inwards and creating a visible effect on the reflected image.

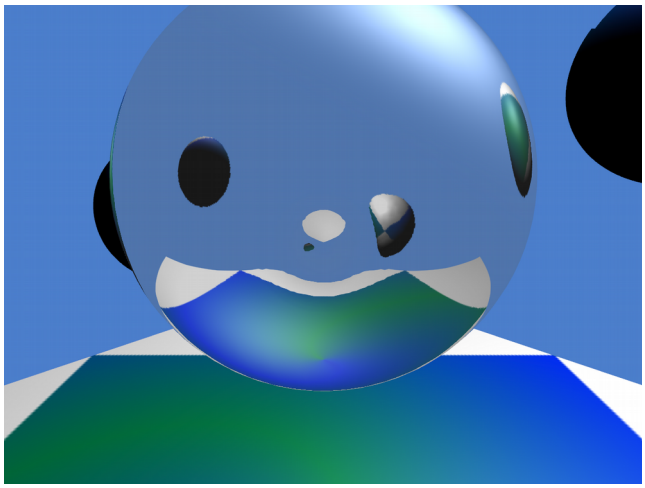
Implementation

There are numerous technical challenges to be overcome here. I will start with the actual reflections off of the ball.
To implement the reflections, I have used a cube map and an external framebuffer. Before anything else happens, the entire scene, without the ball, is rendered six times, with the camera looking along each positive and negative axis, staying in the center point of the reflective ball that is to be rendered. All that changes between each render is the view matrix and which side of the cube map we are rendering to. After this, we can render the ball itself. When applying the texture, we first find the direction of the ray from the eye, reflected off the surface on a given point. In this calculation, we use the normal, which we will alter later. Given this direction, we sample the point on the cube map which corresponds to this vector, and are left with a color reflected from the environment.
The next step is to allow for locally modifiable surface normals. As mentioned in the introduction, the surface normals of the ball are computed with a normal map. The original normal map contains only normals perpendicular to the triangles on which they reside, but this may be changed by hand with your favorite image editor. Normal mapping was already implemented in an exercise, and is done here in a similar fashion: Each vertex has a normal, tangent and bitangent associated with itself. The tangents and bitangents are parallel with the u- and v-texture coordinates at each vertex. A matrix is constructed with these three (ideally orthogonal) vectors, and this again is used on the normal values sampled from the normal map to compute a final normal vector.
The last interesting bit of the implementation is about the interactive modification of the texture map on runtime. When the user looks at a surface point on the ball and presses space, the program should distort the normals to create the illusion of a crater at the surface. This idea is a bit of food for thought, as one would want the resulting modification of surface normals should only affect triangles nearby in world-space,

which does not necessarily correspond nicely to UV-space, where the modifications are actually supposed to be made. I planned on creating this effect on the CPU at first, as I couldn’t imagine how the rasterizer of OpenGL could help me in this. However, it turned out it could.
The program first computes the point of impact on the ball on the CPU. Here, one could generalize the algorithm to take any shape into account, but given my limited scope (and implementation time), I have decided to assume that the target is round. Using the computed collision point, we set up a new framebuffer, distinct from the previous ones, and binds it to the normal map texture. Now, we execute a new shader, with the normal texture as the rendering target and the ball as mesh. In its vertex shader, we compute the distance to the collision point, and the length of this distance’s projection to the tangent and bitangent at the vertex. The gl_Position variable is set to the UV-coordinates of the mesh, as we want the rendered fragments to be put into corresponding pixel positions on the normal texture. Thus, in the fragment shader, we find whether the distance of the fragment is within a hard-coded limit. If it is, we compute a surface normal tilted towards the collision point and blend this with the original surface normal at this point, weighted with the distance from the collision.
Originally, I had hoped to create some kind of projectile, which, when thrusted at the ball, created the valley-effect described above. Alas, time has forbidden such adventures, and thus, the program achieves little more than demonstrating the theoretical principles involved.

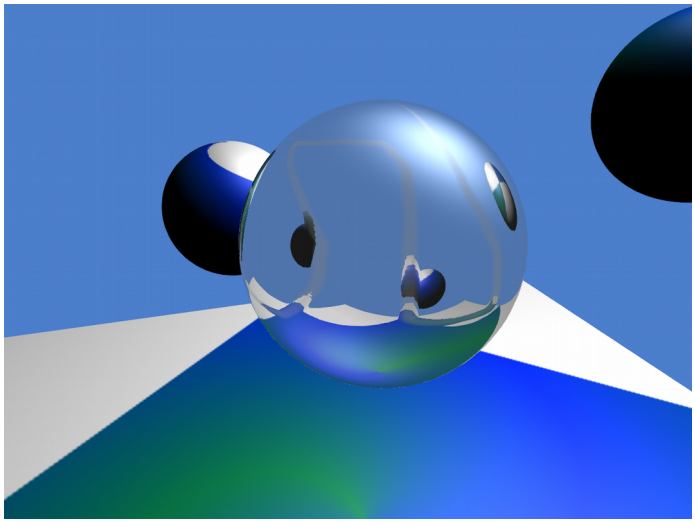


A ball reflecting its surroundings



A ball with distorted normals, still reflecting its surroundings

The original normal map, modified by hand with great artistic precision



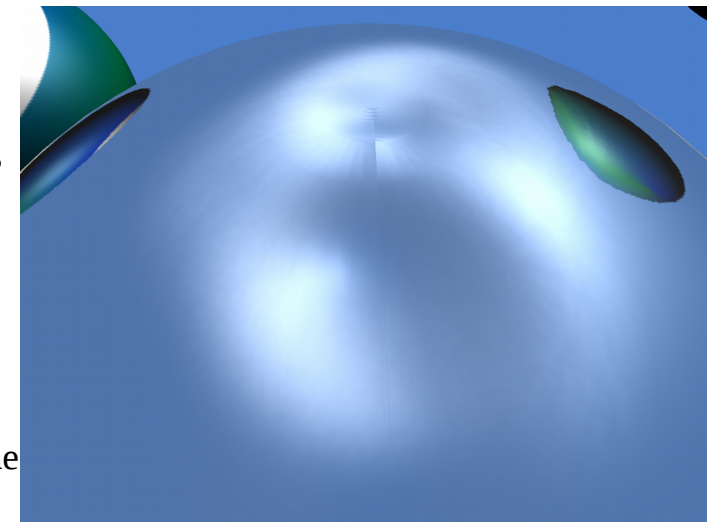
The result of modifying normals by hand. The gray color is applied by the shader and increases with the non-perpendicularity of the normals.

Restrictions

The program has a few approximations and restrictions that may impact the visual viability.

First and foremost, the reflection itself is a crude approximation to how the light would actually behave. This comes from the fact that the environment is rendered only from the *center* point of the ball, and we choose a sample point that lies in the same direction from the center as the ray from the eye that is reflected off of the surface points to. The illusion is believable, but you will find situations where you can see objects behind the ball as a reflection, even though you can't see the object itself.

Another limitation of my implementation, is that the tangents on the surface of the sphere are not entirely continuous on the point of the texture wrapping, which has to do with the fact that I compute the tangents dynamically, and not using the fact that we are dealing with a ball. This might be considered a disadvantage with the model rather than my implementation, but I wrote the model generating code, so anyway. The slight discontinuity creates visible seams in some cases.



Slightly visible seam at line of texture wrapping

Conclusion

The program successfully renders an approximation of a reflecting object, and gives interactive opportunities that, at least to some degree, corresponds to expected behavior, if I dare say so myself. The program runs nicely at a large framerate (I have capped it at about 100 FPS), but this may in turn come of the oversimplified environments. Clearly, when rendering something more complex than what I have currently, will impose a great load on the GPU, since we will have to render everything seven times each frame. Rendering a simplified version of the surroundings may prove to be the right choice in settings where performance is critical.

Still, I am impressed myself, to actually get this in a working shape despite the rather numerous things that could have (and have) gone wrong.