Abstract

In recent times the power of computer hardware, especially in the area of graphic-processing, has reached a very high level. All models used in current generation computer games have such a high amount of triangles, that it is barely possible to recognize single triangles. For the most types of assets throughout the world this method of using triangles works very well because usually every model has some planar areas, which are easily triangulated. The difficult parts are surfaces with a high complexity and lots of naturally curved parts.

One of the objects that fall under this category is water. Considering the wavy water of an ocean or a stream of water running through a riverbed there are no flat surfaces and lots of detail. While this is also true for objects like the human body, the advantage of water is, that it can be described by mathematical formulas. For example, every single wave is the form of a sine wave and thousands of these waves add up to form the final ocean. This mathematically description enables the possibility of rendering the water as a post-processing-effect while using no real geometry.

I did experiment with visualization of water supported by ray tracing some time ago. While the results of my past work were promising I never had the time to continue this work. Now I had the possibility to continue my work within this area in the form of my bachelor thesis. The aim was to push its quality on a level which can compete with current generation techniques in the area of water rendering and even have an advantage in some categories. Fulfilling this aim would mean that the described effect could possibly be used in current generation games.

Apart from the quality, my second personal aim was to acquire more knowledge within the area of shading. Especially physically correct visualization of natural phenomena was always interesting to me because PBR (Physically based Rendering) is the pinnacle of shader programming.

The aim of this thesis is to create such a water-effect, which uses ray tracing in order to avoid the usage of triangles. Without the triangles the final result will look much more realistic than classical implementations of water where u are able to see that the shape is not quite natural. Especially the interaction of the water and the shoreline is improved by being completely smooth without any straight lines caused by triangles.

The presented work in this thesis solves the problem of rendering water with ray tracing. First, the whole scene is rendered without the water. Afterwards, the water will be added in by a post processing effect on a pixel by pixel basis. For every pixel the algorithm reconstructs its position inside of the world. Then a ray will be created starting from the position of the pixel in world space and pointing toward the camera. This ray will be traced to the point where it hits the surface of the water (pixels being above the water surface are discarded). This position of the water surface is then used for all further computations like calculating the reflection or calculating the color of the water at that point.

In order for this algorithm to work the form of the water must be able to be described in a simple way. In this case the form of the water surface is sampled from a pregenerated texture which is layered five times in order to create the illusion of moving waves with standing water.

While this method of rendering is very complex and uncommon in computer graphics, I had tested it before and knew it was possible.
While this method of rendering is very complex and uncommon in computer graphics, I had tested

it before and knew it was possible.

While the ray tracing is the main part of the effect it also implements most of the features water has. The final color is a combination of the reflection (surroundings of the water) and the refraction (content of the water, mainly the riverbed). While the refraction can directly be sampled from the reflection map, the refraction must be further adjusted. Depending on the depth of the water the refracted color is changed so that deeper areas of the water appear in a dark blue. This effect is implemented very close to physical reality by calculating the light absorption of the water for the different colors. For example, red light is absorbed approximately after 4,5 meters of water while blue water is first absorbed after 300 meters in water. Since red light is absorbed first the water appears to the viewer as a naturally blue liquid. This effect is reproduced in the shader. The refracted color is also manipulated to contain the caustics. Caustics are the refractions of light on the ground of water because the waves act like lenses concentrating the incoming light onto a few areas.

In order to show the practical uses of this effect, it was evaluated in two different ways. First, a detailed analysis of the effect's performance was created and compared with other techniques. The main focus for this analysis was the usability in the area of computer games because this is the area in which water rendering is used most.

Secondly, the visual quality of the effect was evaluated with the help of a survey. This survey compares the presented effect with the Nvidia Island Demo, a Tech-Demo using tessellation to render water. The main focus in the survey was on the areas of the effect which are influenced most by the usage of ray tracing.

The general performance of the effect was really good (approximately 2ms of render time, resulting in ~10-15% of frame time if targeting 60 fps) but the interesting part here is the performance footprint. Nearly all of this time is spent inside of the Pixelshader where the ray tracing is implemented while the Vertexshader is literally doing nothing. This huge performance gain inside of the Vertexshader is the reason why this effect outperforms similar techniques, which rely heavily on geometry. Even when using Tesselation for generating the required geometry on the GPU the cost of the geometry transformation and rasterization takes way more time than the here presented ray tracing within the pixelshader.

On top of this, the low usage of the Vertexshader is a huge gain in performance for games, which are using the Vertexshader excessively in other areas of rendering. On the flip side, this effect is probably not preferably for games, which are mainly using the pixelshader in the other areas of rendering.

Considering Quality the survey shows that the presented effect has no deficits compared to the Nvidia Island Demo which is on a very high quality level. Actually, the survey shows the opposite with slightly better ratings across the board in favor of the presented effect. The only part being rated in favor of Nvidia are the Caustics, the refracted light on the ground of the water. This effect however is not directly linked to the usage of ray tracing itself and could be improved independently of the ray tracing.

Both of my personal aims were achieved with this work. The created effect was compared with the high level effect from Nvidia and has advantages in performance and quality. Especially the performance gain of ray tracing is an exceptional achievement considering that the visual quality of the effect does not suffer of its usage.

I could also acquire lots of new knowledge within the area of shader programming while writing the shader of the demo. Since shader programming is only a small part of my study, I had to learn most of the shader language myself. This was a great experience and I learned a lot in the area of shader programming and Physically based rendering.

Overall I am very happy with the developed shader but there are obviously parts of it, which could still be improved in order to strengthen its usability in the gaming industry.

While the performance is already on a very good level, it can still be improved by using more efficient algorithms in the part of the ray tracing. Especially the high usage of textures slows the algorithm down. This texture usage can be reduced by using a better tracing algorithm which in turn will cost more mathematics operations. Trading less texture usage for more mathematical operations is an overall performance gain since texture usage costs way more performance.

However, the more important future work would be improvements to the usability of the shader. Right now, the shader can only display one infinite surface of water and once the camera dives in the water the illusion brakes. While both of these problems can probably be solved quite easily they were left open for future work since they were not needed for the demo of this work.