The Lighthouse

Project in EDAN35 High Performance Computer Graphics

Anton Klarén* Valdemar Roxling•

Lund University
Sweden

Abstract

Graphical effects that affects the environment are very important in realistic real-time rendering, as they give the scene the ”right” feeling. In this project we have created scene consisting of a lighthouse at night surrounded by a big ocean. With only a few light sources and heavy rain, together with sporadic lightning strikes, realistic shadows, detailed terrain and a Christmas touch, the scene gets the dark and stormy-night feeling we were looking for.

1 Introduction

A single graphical effect can be very nice, but in most real applications they are combined together with several others to create the final image. So naturally we had to develop and combine multiple algorithms and position suiting models and lights to be able to compose the entire scene and get it right.

Along with fancy algorithms and detailed models a lot of the feeling comes from the details, like the rain and the waves moving in the same (wind) direction, the rain being illuminated by the light sources, and creating splashes on horizontal surfaces. Together with realistic rain falling rate, correct lighthouse spotlight rotation speed and much more.

2 Framework

To create this project we have developed an open-source graphics framework in `OpenGL` and `C++`, using a deferred-shading pipeline, called Smegod[1].

3 Effects & Algorithms

To compose the scene we implemented many different algorithms to create different effects, but also reused some from previous projects. Both the ocean, implemented using a shader-based approach, and the spotlights with shadows, created using deferred shading and calculating a shadow map per light source, are reused and will not be explained in more detail.

3.1 Rain

The rain in the scene is mostly based on the research in [Tariq 2007]. It is created using a billboard technique where single points are expanded to quads in a geometry shader and later textured in a fragment shader. The billboard quad is always facing the camera, to keep the rain visible at all times. This is accomplished by calculating the plane that is spanned by the vector from the point to the camera, and the vector along the animation direction of the rain. In our rain we sample the texture from several different texture based on a random value to give it a more natural look.

Another approach to render rain is to render it in screen space, this as one major disadvantage though; the rain will not interact with the lightning pass in any computational feasible way. By rendering the rain as particles in world-space we can make calculations based on the actual position of each rain drop in the scene.

To move the points a technique called transformation feedback[2] is used. Transformation feedback is the process of capturing the output from a vertex stage (vertex, geometry, tesselation etc.) and save the data to a buffer. This buffer can then later be used as input to a `OpenGL` draw function. Due to a limitation in the hardware you are not able to render from the data captured during the same frame. To circumvent this limitation we used two separate buffers, one for updating (animating), and one for rendering. After each frame the two buffers are swapped so that the updated data will be rendered the next iteration. This process is done entirely on the GPU and for a parallel operation such as move points along vector this method will be very efficient and able to handle well above a million points in real time.

The update of the rain moves the points one step along a vector based on the frame rate and this will resulting in a falling motion given the right parameters of the vector. In order to add some complexity to the scene the translation vectors are randomly generated with a set of constraints that will keep them in a downwards motion. Once the points are below a certain threshold the are moved to a random location above the current camera position; thus all the particles are reused without the involvement of the CPU.

To give it move realistic look we decided to add a splashing effect to all surfaces with normals primarily in the positive y-direction. This will create the feeling of the rain actually impacting the objects. This approximation works as long as no indoor environments are present, since it will rain on all surfaces regardless if there is a roof or not. A final image of the splashes can be seen in figure 3 with close inspection.

Splashes are added to the normal and specular buffers of the g-buffer and is thus added to the lightning pass of the pipeline. The splashes are also animated with the help of 3D textures where the additional axis in the texture represents time. The textures are mipmapped and then linear interpolated to smooth out the otherwise jerky animations. To remove repetition in the pattern, the splash bumps are randomly moved with an offset between each full animation cycle.

3.2 High Dynamic Range - HDR

HDR is a way to add better contrast to your scene. Regular lightning is often done in 8 bit RGB channels and will thus limit the number of discrete colors to 256^3. According to [Fabien Houlmann 2006] HDR expands the channels to be float values, typically 16 bit, and will greatly expand the color space during the light calculations.

Once the calculations are done, the lightning buffer must be mapped to the 8 bit RGB space in order to send the image to the display. This process is called tone mapping and in our scene we chose a tone mapping algorithm known as exposure mapping. Exposure mapping mimics the human eye by simulating the iris controlling the size of the pupil and is very intuitive to use.

To add extra light to specific areas such as the lamp posts and the Christmas-lights, we used the alpha channel of the images to specify emittance. This will reduce the amount of texture storage since

*fys09akl@student.lu.se, anton@klaren.it
•dat11vro@student.lu.se, valdemar.roxling@gmail.com

[1] https://github.com/Roxling/smegod

[2] https://www.opengl.org/wiki/Transform_Feedback

no new emittance-texture is loaded, but it removes the possibility of using transparent objects without changing the structure.

Everything that is over-exposed to light, with color values exceeding the normal range of visible colors will be added to the bloom buffer.

3.2.1 Lightning strikes

To generate the effect of lightning strikes we used the fact that our HDR tone mapping is a simplified model of the human eye. The exposure value is deciding how much the camera will react on light, and is set very low in this particular scene, to create the feeling of a late evening or night. By occasionally adding a sinus peak, increasing the sensitivity to light by a factor around 100 times bigger everything becomes very bright for a short amount of time as the camera gets ”blinded”. The result feels surprisingly natural.

3.3 Light-blur on distance

In order to add a glowing effect to the lights a bloom buffer were created. This buffer is filled with all the light from the light buffer with intensity greater than 1.0, and will then only contain the regions with that are over-exposed to light. To get the blur effect, a Gaussian kernel convolution is applied to the buffer and the result is added to the final image.

One problem with Guassian blur is the computational cost of performing a full convolution. Thankfully, the kernel can be separated into a column and a row vector, that can be applied separately; this is known as a separable convolution. According to [Eddins 2006], filtering a M-by-N texture with a P-by-Q kernel requires roughly *MNPQ* multiplications and additions. By using a separable version this complexity is reduced to one iteration with *MNP* and one with *MNQ*, resulting in the final cost of *MN(P + Q)*.

This optimization were not enough to get a decent frame rate so two additional methods had to be implemented. The bloom buffer is sampled at half of the resolution before the blur occurs and then interpolated back to the original resolution. Due to the smaller resolution, the size of the kernel can be reduced while still producing the same blur-factor. As a plus, the interpolation back to full resolution will add to the blur as well.

The last optimization exploits the texture hardware in the GPU to perform interpolations between texels. Since the texture lookup-calls allows one to specify a coordinate between two texels, the resulting value are the weighted values of the closest textels. By calculating offsets based on the Gaussian kernel the number of texture lookup-calls can be reduced by 50%.

Blurring the buffer with the first version of Gaussian blur took about 28 ms, while the optimized version took 2 ms on the same hardware, resulting in 14 times the performance.

3.4 Cobblestones

The heavy detailed cobblestones are created by a neat trick, called parallax occlusion mapping, where the depth between the stones are just a texture illusion. To produce this effect we need to perform ray tracing for each fragment and adjust the texture coordinates accordingly. Since we are not able to do actual ray tracing an iterative process called ray marching is used instead, as described by [Donnelly 2005].

The algorithm is illustrated by figure 1. We start by calculating the vector from the fragment to the camera and then move along that vector until we hit the desired surface. The final coordinates are based on that point and later used in the lookup-call for the diffuse, normal and specular textures. The number of iterations are based on heuristics and in our implementation will be limited between 20 to 40 iterations depending on the viewing angle. This is a reasonable heuristics since we need more iterations for shallow angels than step.

The desired surface is stored in a single 8 bit channel texture and is called a height map. Since its just a 2D texture one limitation is that you can not have overhang in the resulting geometry. The final result can be seen in figure 3.

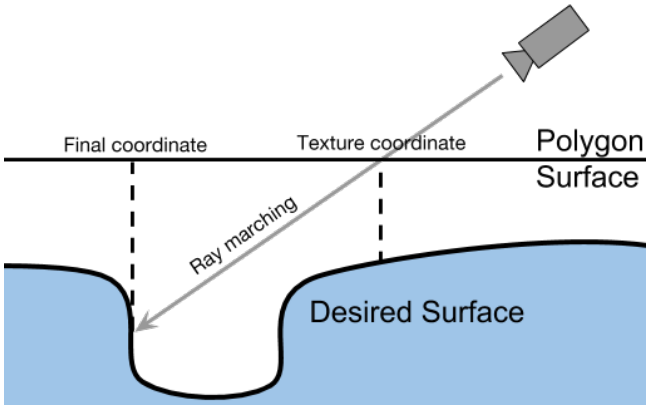

Figure 1: Illustration of the parallax occlusion mapping algorithm.

4 Results

The final result really captures the feeling we were looking for, with the geometry, rain and light effects all working together, as can be seen in figure 2. The entire scene runs perfectly fine in a reasonable frame-rate on any modern graphic chip at most a few years old. In this particular demo we render six lights with shadow maps, one million rain drops, use parallax occlusion mapping on all diffuse textures with 20 to 40 samples and post-process with HDR and bloom effects without any problem.

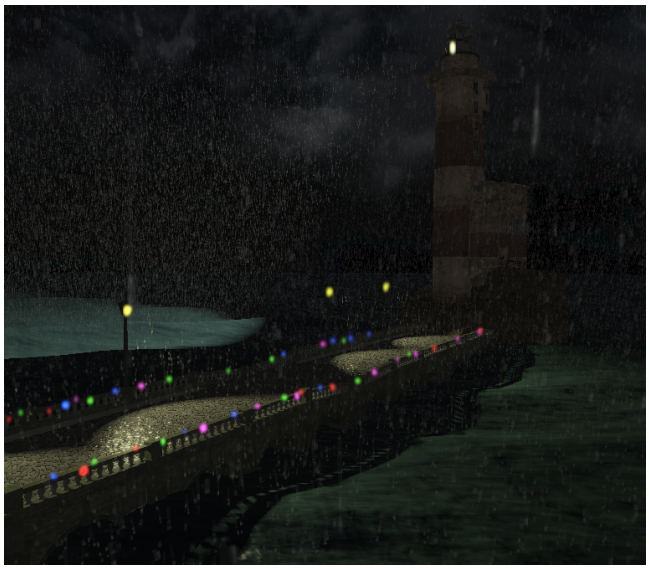

Figure 2: Overview of the scene.

A closer look on the cobblestones really shows the amount of details this algorithm can give to flat textures, as seen in figure 3, along with the animated rain-splashes also visible by closer inspection.

Using many different raindrop textures along with realistic animation and falling direction makes every raindrop unique, as can be partly be seen in figure4. And together with the nice illuminating effect caused by the light sources, and distance to them, the rain comes alive and feels natural.

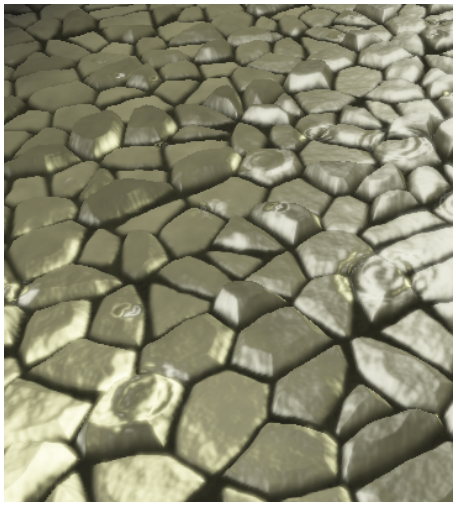

Figure 3: Parallax occlusion mapped cobblestone texture together with rain splashes

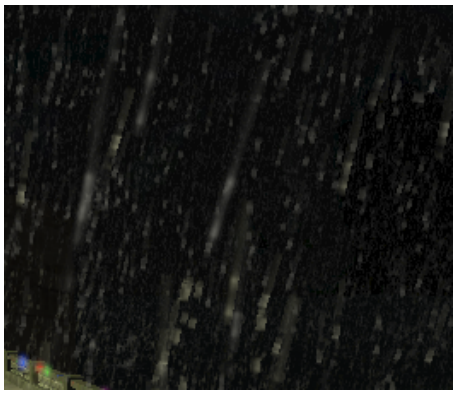

Figure 4: Raindrops affected by light.

To get all of these results combined into a final image we used several different buffers to hold different states and parts of the image, as illustrated in figure 5.

5 Discussion

We managed to get every algorithm working in the intended way, but not without some of them to cause trouble in one way or another. Some approaches were very inefficient and dropped the frame-rate to unacceptable levels, and had to be reworked to get a similar graphical result in a more efficient way.

We also struggled a lot to get every algorithm running on several different GPU's, from both AMD and Nvidia, where `OpenGL` often gave different graphical results, or did not work at all, and had to be rewritten. Almost every time it turned out that we had done something to cause undefined or unspecified behavior, giving us these problems.

6 Conclusion

Writing efficient `OpenGL` code that both works and looks good on every GPU is close to impossible, and fixing these problems takes away a lot of time from actually writing the algorithms, as most of the time is spend to get `OpenGL` to do what you want it to do.

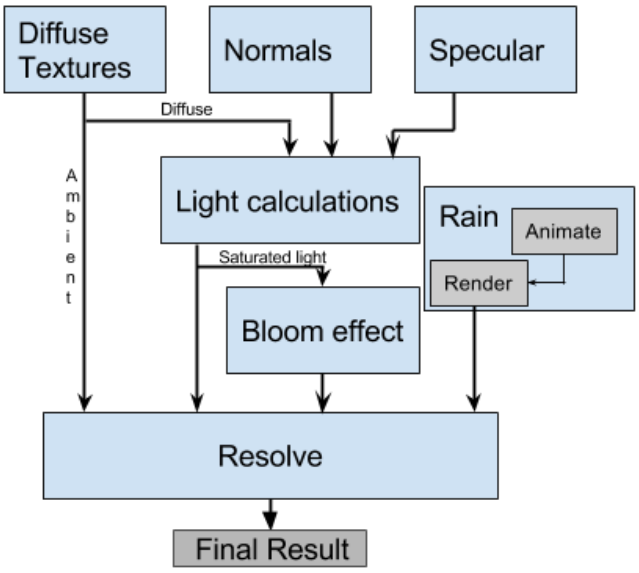

Figure 5: An illustration of some of the buffers used to create the final image.

Acknowledgments

- Joey de Vries for his amazing articles about modern `OpenGL`.[3]
- Etay Meiri for examples on transformating feedback and deferred shading.[4]
- Azlyirnizam for the lighthouse model.[5]

References

DONNELLY, W. 2005. Chapter 8. per-pixel displacement mapping with distance functions. In *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional.

EDDINS, S. 2006. *Separable convolution*. MathWorks, October.

FABIEN HOULMANN, S. M. 2006. *High Dynamic Range Rendering in OpenGL*. Université de technologie Belfort-Montbéliard, 19 June.

TARIQ, S. 2007. *Rain*. NVIDIA Corporation.

[3] http://www.learnopengl.com/
[4] http://ogldev.atspace.co.uk/
[5] http://tf3dm.com/3d-model/light-house-enterable-96732.html