Towards Real-Time Visual Simulation of Water Surfaces

Jens Schneider* and Rüdiger Westermann†

Scientific Visualization & Imaging Group
Aachen University of Technology
Seffenter Weg 23
52056 Aachen,Germany

Abstract

In this paper we demonstrate the benefits of the most current nVidia graphics chip set for realistic simulation and rendering of dynamic water surfaces in real-time. In particular we employ programmable vertex shaders for the simulation of the water surface height field and for the displacement, transformation and lighting of vertices. To keep bandwidth requirements low a superposition of NURBS surfaces generated by the hardware through OpenGL evaluators and gradient noise implemented in the shader program itself is proposed. Furthermore, environment cube-mapping is used to enable realistic rendering. Our method can efficiently simulate effects such as refraction, reflection and the Fresnel term to model the optical characteristics of water. Although in this work only the simulation of a certain class of natural phenomena is exemplified, the proposed method is suitable for visually representing other effects as well.

1 Introduction and related work

The realistic simulation and rendering of water-like phenomena in real-time is still a challenge to the computer graphics community. Although the modeling and rendering of animated water surfaces has been studied extensively during the last decades and the underlying physical models and optical properties are well understood, the complexity of the numerical solution methods in general prohibits real-time visual simulation.

The proposed techniques can be classified into different categories with respect to the algorithms used to simulate the desired effects. In its most general form, wave propagation is described by the linear hyperbolic wave equation that governs the displacement of a surface under tension during small vibrations:

$$\frac{\partial^2 y}{\partial t^2} - c^2\left(\frac{\partial^2 y}{\partial x^2} + \frac{\partial^2 y}{\partial z^2}\right) = 0$$

Here, c specifies wave speed and the displacement of surface points is along the y-direction.

A closed form solution to this equation can be found by specifying initial conditions on the surface boundaries and on the displacement of the surface. The general solution involves expensive computations of trigonometric function series but can be simplified by using only the dominant frequencies. Even more efficiently, the continuous equation can be approximated by means of finite differences, which results in a simple and local scheme to be solved at each surface point [8].

Other approaches rely on results from oceanography, where it has been shown that ocean waves can be simulated using statistical models. The water surface height field is composed of a finite sum of complex sinusoidals with different phases and amplitudes that are computed by utilizing an FFT algorithm. A variety of different shapes can be simulated quite efficiently using this approach (see [3] for a good overview), but the complexity of the employed algorithm doesn't allow for interactive frame rates in general.

Particle models [5, 11], on the other hand, are based upon the separate displacement of particles with respect to physics based constraints. By simulating the circular motion of particles during wave propagation effects like shoreline waves or wave loops can be modeled. Beta splines were employed for wave simulation in [16]. Using only a small set of control points the shape of smooth waves that doesn't exhibit turbulent features can be modeled quite realistically.

A very general procedural technique for the simulation of water surfaces by means of stochastic fractals was proposed in [9]. Although it has been shown that this particular kind of simulation process is only well suited for a very limited kind of wave phenomena, its ease and efficiency in implementation and the possibility to use this process to simulate other phenomena make it a very appealing alternative. Particularly in games industries where the demand for physical realism is often over-balanced by the demand for real-time simulation, simplified models that still exhibit physical realism but also guarantee for the desired frame rates are favored.

However, due to the huge amount of vertices to be generated and processed to generate realistic water surfaces *and* the numerical complexity of the operations to be performed water simulation techniques in general are still far from real-time. Furthermore, even if a practical approach would be available, the number of graphical primitives to be send through the graphics pipeline make real-time rendering of realistically sized domains difficult to achieve.

Considerable effort has been spent on the development of rendering techniques for water including refraction, reflection and Fresnel terms [4]. Refraction and reflection can be easily simulated taking into account ray optics based on Snell's law. A less accurate approximation of the Fresnel term which models the dependence of the reflectance from the incoming light direction and the surface's index of refraction was given in [14]. This approximation to the physically accurate BRDF will be used in our approach as well. In addition, caustics that are based on diverging refractions of wavefront at water surfaces have been simulated quite pleasantly in [17]. An approximative simulation of ray beams refracted through water that illuminates the ground will also be addressed in this work.

The development of dedicated graphics hardware, on the other hand, has spawned a completely new class of rendering algorithms particularly designed to exploit advanced graphics opera-

*e-mail: schneide@glHint.de
†e-mail: westermann@sc.rwth-aachen.de

tions through standard APIs, *i.e.*, to simulate realistic lighting effects and the Fresnel term [7] and caustics [15] . Anticipating the same trend in other application areas we are actively investigating the possibility to accelerate simulation algorithms by means of these extensions.

Recently, a number of first attempts to exploit graphics hardware for the real-time rendering of water surfaces have been proposed (see [10, 1]). Here, the water is assumed to stay in a closed and opaque container, which makes environment mapping into precomputed texture maps capturing the upper and the lower surrounding scene the appropriate choice. These techniques are of particular use in games and entertainment industries and can be run at interactive speed on consumer market graphics cards. In general they rely on hardware support for computing reflection and refraction vectors and for the automatic generation of texture coordinates used to index into the environment maps. In comparison, our technique takes full advantage of vertex shader programs for the simulation of water, it uses only one cube-map for the reflective and refractive part and it achieves improved visual quality by adding effects such as attenuation, color shifts and caustics.

In this paper we address the problem of simulating *and* rendering water surfaces in real-time on graphics cards suitable for the consumer market, i.e. the nVidia GeForce3. We outline an unique approach that allows for the simultaneous simulation and rendering utilizing hardware supported graphics operations. In this respect our approach is different to others in that we employ graphics hardware to entirely perform visual simulation on chip.

The goal of our approach is twofold: to emphasize the impact of affordable state-of-the-art graphics hardware on current simulation techniques *and* to demonstrate how it can be used to accelerate the realistic rendering of water surface height fields. In particular we show that vertex shaders can be exploited to interactively generate non-stationary stochastic fractals, which are used to simulate the dynamics of water in this work. Furthermore, we demonstrate that OpenGL evaluators can be used effectively to limit bandwidth requirements usually inherent to approaches that rely on the displacement of huge numbers of vertices.

The simulation result is then rendered including physically motivated optical models. For both effects, refractions and reflections, we employ environment cube-maps and automatic texture coordinate generation to appropriately map into the pre-computed textures. Both the reflecting and the underwater environment is represented by one cube-map in our approach.

The reminder of this paper is organized as follows. First, we summarize the fractal Brownian motion (fBm) method used to generate the water surface height field and we demonstrate how it can be computed on current graphics hardware. In particular we demonstrate the benefits of evaluators to avoid the setup of a huge number of vertices through the API. Next, we present our technique for rendering the water surface, which incorporates the refraction, reflection and Fresnel term. We finally conclude our paper with a detailed discussion of our results and we propose further improvements and applications.

2 Simulation of water surfaces

As summarized in the introduction many different techniques exist to simulate realistic water surfaces. In the current work we shift emphasis towards the development of a method suitable for the implementation on available graphics hardware thus enabling real-time generation of water surface height fields. To reach this goal we utilize hardware supported vertex shaders available on the most current nVidia chip set. Vertex shaders are supported in OpenGL via the NV_vertex_program extension. We will use the terms *vertex shader* and *vertex program* synonymously.

Vertex shaders can be seen as small fragments of micro code that can be loaded into the GPU. Every vertex that is sent through the geometry pipeline bypasses the traditional transform-&-lighting stage and undergoes the operations implemented in the shader program. A limited number of input and temporary registers are available that can be accessed to code and store variables that are needed by the shader. Some constraints, however, make the use of vertex shaders quite restrictive:

- Vertex shader programs have a maximal length of 128 statements. Thus only a very limited number of operations can be implemented.
- The available instruction set is restricted to a few simple operations including additions, multiplications, dot product calculations and comparisons among others. More complex instructions like modulo operations, trigonometric function evaluations or branches are not supported.
- Vertices can only be transformed but new vertices can not be generated. Furthermore, access to neighboring vertices is not possible in general.

Due to these limitations a method for the simulation of water surfaces is needed that can be evaluated locally by using only a few simple operations. Functional approaches to the simulation of stochastic fractals satisfy these requirements.

2.1 A functional approach

The use of time-varying fractals for the visual simulation of natural phenomena like clouds, fire, smoke or water has been studied extensively during the last decades. Perlin [12, 13] first introduced a functional synthetic turbulence model as the basis for the realistic simulation of many different effects. It can be seen as a particular kind of stochastic fractal that is generated as a summation of several appropriately scaled and dilated copies of a continuous noise function. For an excellent and comprehensive survey of various approaches to generate fractals and how to use them in natural image synthesis let us refer to [2].

In [9] the suitability of fractal displacements for the visual simulation of water has been demonstrated. Surprisingly, only two octaves of noise were added to generate quite pleasant and realistic results. Although it is well known that water surfaces do not exhibit a fractal shape in general, in many particular applications, e.g. the simulation of calm water or water in narrow containers, convincing results can be achieved using this approach.

The simplest noise function (value noise) is generated by means of random values positioned at each grid point of the underlying domain. To minimize memory requirements a hash-table is usually employed for indexing the grid. The discrete scalar field is extended to a continuous or even smooth function using linear or higher order interpolation. A more sophisticated technique (gradient noise) for the generation of a stochastic function was proposed in [12]. In this technique, pseudo-random gradients are stored at each grid point and dot products between the fractional parts of the sample position with respect to the integer lattice and these gradients are interpolated. As a consequence, the noise function becomes zero at every grid point, and it thus exhibits a more turbulent shape due to increasing high-frequency parts.

In this work the displacement of vertices simulating the water surface height field is computed by means of the well-known turbulence function [12], where the fractal exponent H_i is used to model the roughness of the height field and t specifies time in order to simulate dynamic behavior:

$$y = \sum_{i=0}^{1} \frac{1}{2^{H_i}} noise(2^i x, 2^i z, 2^i t)$$

Particularly for the simulation of indoor scenarios or water in containers where large wave valleys are not likely to occur the use of gradient noise leads to more realistic results compared to simple value noise. Furthermore, as will be described below, the normal to any point on a gradient noise function can be efficiently estimated without the need to evaluate additional function samples.

2.2 Bandwidth requirements

Before we start with a detailed description of the procedure we employ to generate the water surface height field using vertex shaders let us briefly outline an efficient technique to minimize the bandwidth requirement inherent to the rendering of large scale dynamic height fields. Usually, the displacement of every grid point is computed on the CPU and the displaced vertices are sent to the geometry processing unit. This, however, imposes some limitations on the number of vertices that can be rendered in real-time.

In order to circumvent this drawback we exploit the hardware to generate vertices by means of OpenGL evaluators. Evaluators provide a mechanism to specify a surface using only a few control points. Roughly speaking, the user specifies the degree of the polynomial surface representation to be approximated, the control points and the level-of-detail of the approximating grid. On the graphics unit surface points are then going to be generated at the desired resolution and the appropriately tesselated surface approximation is rendered. As a consequence the number of primitives to be set up by the user is considerably decreased.

In our approach we start by generating a regular mesh using evaluators. On the CPU a low frequency octave of gradient noise is evaluated on a 8x8 grid that serves as control points. Using these 64 control points the grid can be generated in any finer resolution by the graphics hardware. All remaining computations, i.e. the generation of the fBm, the superposition of the NURBS surface with the fBm and the computation of per-vertex normals, are then going to be performed in the shader program itself.

2.3 Vertex shaders for gradient noise

Each vertex that is generated by the evaluators finally passes through the vertex shader program before it is sent to the fragment processing unit. Vertex shaders are programmed using micro code that only supports a limited instruction set.

In the current implementation one additional high frequency octave of gradient noise is calculated inside the vertex program and superimposed with the appropriately tesselated low frequency NURBS surface. Indexing is performed using a permutation field. Both the gradient lattice and the permutation field are stored as linear arrays in constant registers each of length 64 that can be accessed in the shader program. To both tables the first entry is appended in order to save two modulo operations that require three instructions each. The entire code necessary to generate one octave of 2D gradient noise already requires 48 instructions. This includes the gradient lookup, the computation of the fractional parts of each sample point with respect to the integer lattice and the dot product calculation between the gradient and these fractions.

Unfortunately only up to 128 instructions can be coded in a single shader program. This imposes very strong limitations on our approach, since it prohibits the generation of additional octaves and also restricts the method to the simulation of stationary effects in the shader. We can easily overcome the latter problem by simulating dynamic effects on the CPU. Therefore, the time varying gradient field is generated by linearly interpolating the current values between pre-computed values at integer lattices. It is then issued to the shader program where indexing is always performed using the static permutation field. Advection by wind or similar effects can also be simulated by means of this technique.

Once the fractal displacement has been computed in the shader program the current vertex position is disturbed using this displacement. It is just used as a shift towards the y-axis thus resulting in the water surface height field (see color plate 2).

2.4 Normal estimation

In order to calculate refractions, reflections and the Fresnel term the normal of the water surface height field at any grid point is needed. The normal at any point of the gradient noise field is the cross product of the partial derivatives with respect to x and z. The partial derivatives, however, can be estimated quite efficiently using forward differences. Therefore, let us recall that the contribution of the gradient (∇) at a particular lattice point to the interpolated noise sample is

$$noise = \nabla_x \cdot fx + \nabla_y \cdot fy + \nabla_z \cdot fz$$

where fx, fy and fz are the fractional parts of the sample position with respect to the integer lattice. The partial derivatives can then be approximated by forward differencing as follows:

$$\frac{\partial noise}{\partial x} = \frac{noise(x+ds,z,t) - noise(x,z,t)}{ds}$$

$$\frac{\partial noise}{\partial z} = \frac{noise(x,z+ds,t) - noise(x,z,t)}{ds}$$

Instead of calculating three different noise-values, we perform the gradient lookup for $gnoise(x,z,t)$ as usual and we shift the interpolation weight by ds along the x and z-directions. This, in general, yields a good approximation, and as a consequence it is sufficient to access the gradient lattice once to derive the partial derivatives. At each sample point the contributions from adjacent grid points involved in the interpolation procedure are finally summarized and used to derive the normal vector. Note that discontinuities in the surface normals are introduced if different lattice patches are accessed to compute forward differences.

The question that remains to be answered is how to reconstruct the normals of the surface that is generated by the superposition of the fBm and OpenGL evaluators. Recognizing that for Bernstein-polynomials

$$\frac{\partial}{\partial t} B_i^n(t) = n(B_i^{n-1}(t) - B_{i-1}^{n-1}(t))$$

the partial derivatives of the NURBS surface can be obtained by using a 7x7 grid of discrete differences as control points for two new evaluators that generate the partial derivatives with respect to u and v. All three evaluators can be issued simultaneously, and the results can be accessed in the shader program from different registers. The initial evaluator that is responsible for generating the surface triggers the execution of the shader program. The vertex shader interface takes care of the evaluation order, such that all other evaluators are processed in advance to the initial evaluator. Since all evaluators are requested to tesselate the same grid, the partial derivatives exactly correspond to the derivatives at the vertex positions. Vertex normals are computed by superposition of the partial derivatives of the evaluated mesh and the perturbed mesh. A cross product is finally performed along with a normalization to yield an unit-length normal N.

Up to this point the shader program performs all necessary tasks to compute the water surface height field and its normal at every grid point. What remains to be done is the rendering of the water surface incorporating realistic optical effects.

3 Efficient Rendering using OpenGL

In the shader program we proceed by calculating the view vector V at every vertex point using the current modelview matrix which can be tracked by the vertex program. The reflection vector is calculated as

$$R = 2 \cdot < V, N > \cdot N - V$$

and forced to the upper hemicube by performing a re-mapping along the y-component:

$$R'_y = \frac{1}{2} \cdot R_y + \frac{1}{2}$$

In this way we only need to use one cube-map as illustrated in Figure 1 that represents both the reflective and the refractive part.

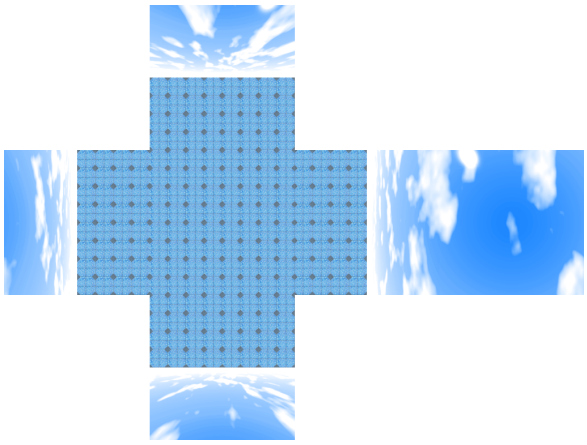

Figure 1: *The cube-map we use for the reflective and the refractive part of the environment.*

This step effectively halves the sampling ratio along the sides of the cube-map thus adding distortions. While this effect can be reduced by prefiltering it is hardly noticeable in practice since liquid surfaces in motion do not show sharp reflections. The modified reflection vector is then issued as texture coordinate used to fetch values from texture unit 1. This unit stores the environment map containing the reflective and the underwater environment, but only the reflective part is accessed.

The transmission vector is a lot more expensive to compute than the reflection vector. According to Snell's law (see [4]) it can be calculated as follows:

$$T = \frac{1}{n_r} \cdot I - \frac{1}{n_r} \cdot < I, N > \cdot N + \sqrt{1 - \frac{1}{n_r^2} \cdot (1 - < I, N >^2)} \cdot N$$

where:

I: Incidence vector from camera to vertex. Notice that $I = -V$

n_r: Relative refraction index. Water has a refraction index of ≈ 1.33

For each refraction vector a ray-hemicube intersection test is performed that exploits the geometry of a hemicube corresponding to the water container. If we assume only single refractions and reflections, and if n_r is assumed to be greater than one, all rays are going to intersect the base plane if they are not occluded by one of the other faces. This, however, is only true for a viewer above the surface. The base plane intersection is calculated separately, the intersection tests with all other faces can be performed in parallel using a single register. From all possible intersections we choose the one with the minimum distance greater than zero and we reconstruct the intersection point. A new vector pointing from the origin of the cube to that point is then issued as texture coordinate into texture unit 0. This unit stores the same environment map as texture unit 1, but now only the underwater environment is accessed.

To produce a more natural look texture values from unit 0 and unit 1 are blended using a reflectivity term ρ. An accurate approximation of ρ can be obtained as a function of the surface normal and the view vector as proposed in [14]:

$$\rho(f_0, N, V) \approx f_0 + (1 - f_0) \cdot < N, V >^5$$

where

$$f_0 = \frac{(1 - \frac{1}{n_r})^2}{(1 + \frac{1}{n_r})^2}$$

f_0 is the reflectivity at incident angle, which is $\approx 2\%$ for water. However, by choosing this value the surface might look less reflective than real water because the refractive index is essentially a complex valued entity. The complex part is called the *extinction coefficient* and will be used later on to model absorption. By adjusting f_0 other effects like dispersing minerals or other particles can be simulated as well. Notice that the refractive index is also a function of wavelength, but we treat it as a constant when computing reflectivity in order to fit our program into 128 instructions. The reflectivity is then set up as the primary color's alpha value.

Outside the vertex program we use the EXT_texture_env_combine extension to interpolate linearly between the texture values fetched from texture units 0 and 1 using the alpha value as interpolation coefficient.

While on the GeForce3 chip per-vertex computation of the Fresnel term could be replaced by a 1D table lookup, using three texture units at a time results in a significant drop in performance. Moreover, only three additional instructions are needed to calculate the reflectivity inside the vertex shader.

Finally, an additional lighting calculation is performed. We try to approximate the visual sensation produced by caustics even though we know that a correct simulation of this phenomenon is rather difficult to achieve and would not fit into our vertex program. To simulate this effect a modified Blinn-Phong lighting model is applied as proposed in [6]:

$$I' = k_a + k_d \cdot < L, N > + k_t \cdot < H_t, N >^{tr}$$

where

- I' is the intensity.
- k_a, k_d, k_t are coefficients for ambient, diffuse and transmissive lighting.
- L is the normalized light vector.
- H_t is the normalized transmissive half vector, see below.
- tr is the transmissive exponent.

The transmissive half vector H_t can be calculated by:

$$H_t = L - \frac{1}{n_r} \cdot V$$

where V is the view vector.

As mentioned before, the absorption of light beams due to

the physical properties of water should be simulated as well. We do so by computing a e^{-cx} drop-off of the light intensity when a beam is traveling the distance x underwater. The problem in doing so is to accurately calculate x, and as a consequence we once again benefit from a compact approximation.

Since we have already calculated the length of the transmitted ray we can treat it as if it was reflected perpendicular to the mesh surface at the intersection point. However, we still have to integrate along the way back to the surface, which is too costly to be performed in the shader program. Thus the second length is simply approximated by the depth of the pool, which is not correct in general but produces a good visual appearance in practice. The approximated length x of the underwater ray is then used to simulate wavelength dependent absorption:

$$I'_{result} = I' \cdot \begin{pmatrix} e^{-c_r x} \\ e^{-c_g x} \\ e^{-c_b x} \end{pmatrix}$$

For the purpose of absorption the extinction coefficient is split into three color-bands according to the rgb-paradigma to simulate the blue shift generally expected when looking into water. The resulting color is then set as the primary color's RGB components.

With texture unit 0's environment set to *modulate* and texture unit 1's environment set to *interpolate*, we finally obtain the following blending equation:

$$C^{rgb}_{result} = (1 - C^{\alpha}_{0}) \cdot C^{rgb}_{Tex0} \cdot C^{rgb}_{0} + C^{\alpha}_{0} \cdot C^{rgb}_{Tex1}$$

where

- C^{rgb}_{Tex0} the fragment's color from texture unit 0 (transmission)
- C^{rgb}_{Tex1} the fragment's color from texture unit 1 (reflection)
- C^{α}_{0} alpha component of primary color (reflectivity)
- C^{rgb}_{0} RGB-component of primary color (lighting result)

4 Results and Conclusion

In this work we have emphasized a general approach for the visual simulation of dynamic water surface height fields. In particular we have shown that both simulation *and* rendering can entirely be performed on the GPU thus minimizing bandwidth requirements and reducing less efficient RAM access. The major contribution here is that dynamic water surfaces can be rendered in real-time by means of hardware accelerated vertex generation, vertex perturbation and texture mapping.

We have demonstrated that evaluators and vertex shader programs can be used efficiently for the generation and the displacement of vertex positions according to an approximation for water surfaces. Multi-textures and environment maps have been incorporated to correctly model refractions and reflections thus exploiting the rasterization performance of current graphics hardware architectures. We spent some effort on the visual improvement by integrating caustics, attenuation and color effects. Quite realistic rendering of water surface could be achieved in this way.

By exploiting the functionality of the nVidia GeForce2 chip set, which emulates vertex shader in software, we were able to render all examples shown in the color plates below at 20 fps. The grid resolution was 64x64 in all our examples, dynamic effects were computed as described and additional movement due to wind was added on the CPU. On the GeForce3, vertex shaders are hardware accelerated. Here we achieved up to 100 fps, which allows us to add additional effects or to increase the grid resolution. In this context it is quite interesting to note that the performance of our technique is not bound by the rasterization unit but only by the vertex shader program.

However, due to the limitations of the shader program with respect to functionality and the number of instructions that can be coded only two octaves of gradient noise could be directly implemented. In this respect we are currently investigating new methods for the generation of realistic water movement, which can be implemented using the provided functionality.

References

[1] M. DeLoura, editor. *Game Programming Gems.* Charles River Media, 2000.

[2] D. Ebert, K. Musgrave, D. Peachey, K. Perlin, and S. Worley. *Texturing and Modeling: A Procedural Approach*. Academic Press, 1994.

[3] D. Ebert, K. Musgrave, P. Prusinkiewics, J. Stam, and J. Tessendorf. Simulating nature: From theory to practice. ACM Siggraph '00 Course Note, September 2000.

[4] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics — Principles and Practice (2nd Ed.)*. Addison-Wesley, 1990.

[5] A. Fournier and W.T. Reeves. A simple model of ocean waves. *Computer Graphics*, 20(4):75–84, 1986.

[6] R. A. Hall and D. P. Greenberg. A testbed for realistic image synthesis. *IEEE Computer Graphics and Applications*, 3(8):10–20, November 1983.

[7] W. Heidrich and H.-P. Seidel. Realistic, hardware-accelerated shading and lighting. *Computer Graphics, Proc. SIGGRAPH '99*, August 1999.

[8] M. Kass and G. Miller. Rapid, stable fluid dynamics for computer graphics. *Computer Graphics*, 24(4):49–57, August 1990.

[9] K. Musgrave. *Methods for realistic landscape imaging*. PhD thesis, Ann Arbor, Michigan, 1994.

[10] nVidia. nVidia Developer relations & Whitepapers. http://www.nvidia.com/Developer.

[11] D. Peachey. Modeling waves and surf. *Computer Graphics*, 20(4):65–74, 1986.

[12] K. Perlin. An image synthesizer. *Computer Graphics*, 19(3):287–296, 1985.

[13] K. Perlin and E. M. Hoffert. Hypertexture. *Computer Graphics, Proc. SIGGRAPH '89*, pages 253–262, July 1989.

[14] C. Schlick. An inexpensive BRDF model for physically-based rendering. *Computer Graphics Forum*, 13(3):C/233–C/246, 1994.

[15] C. Trendall and J. A. Stewart. General calculations using graphics hardware, with application to interactive caustics. In *Eurographics Workshop on Rendering*, pages 287–298. Springer, June 2000.

[16] P. Y. Ts'o and B. A. Barsky. Modeling and rendering waves: Wave-tracing using beta-splines and reflective and refractive texture mapping. *ACM Transactions on Graphics*, 6(3):191–214, 1987.

[17] A. Watt and M. Watt. *Advanced Animation and Rendering Techniques - Theory and Practice*. Addison-Wesley, New York, 1992.

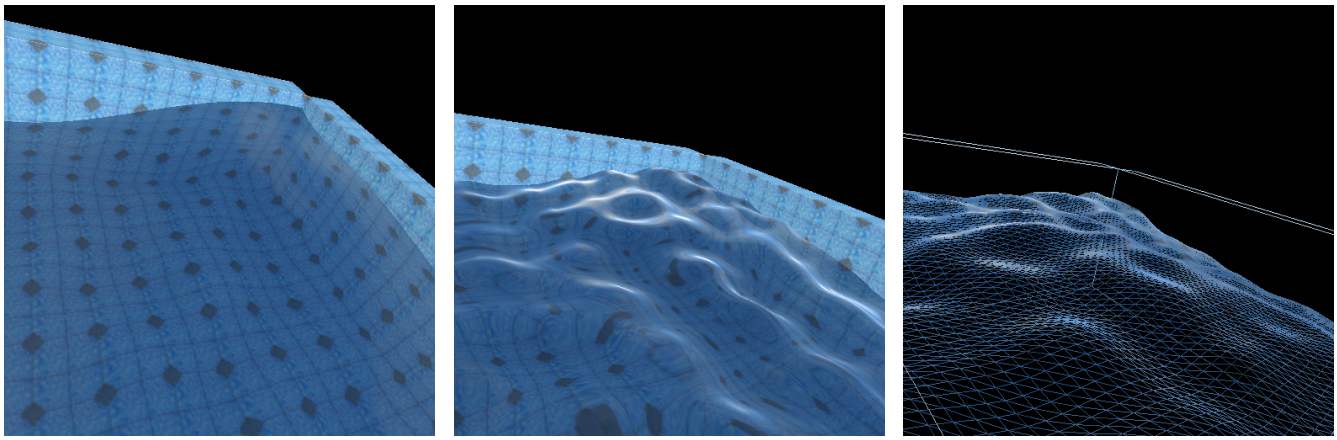

Figure 2: *First, the water surface generated by means of OpenGL evaluators is shown. Next, one octave of a gradient noise fBm is superimposed. On the right the wire frame representation is shown.*

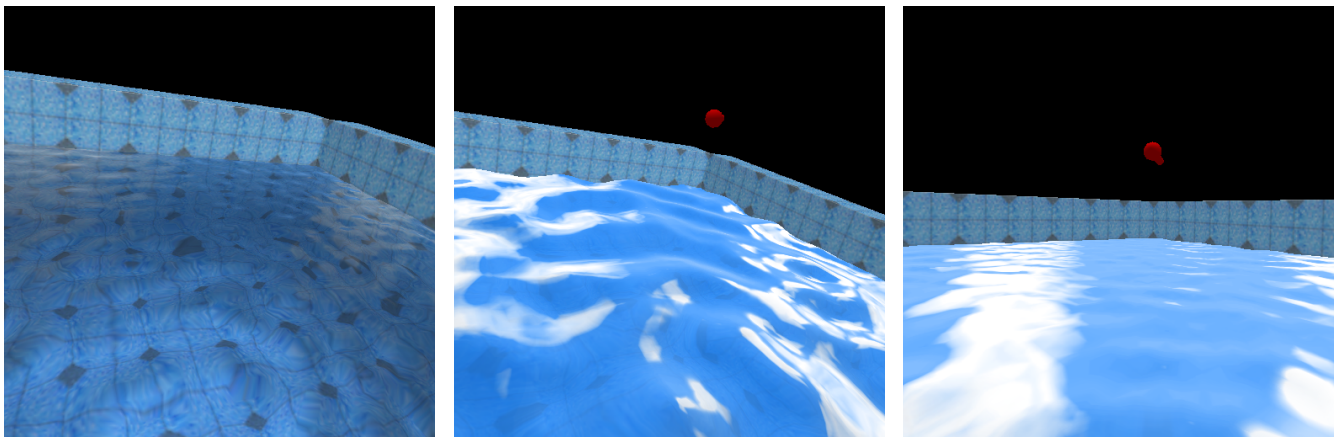

Figure 3: *These images demonstrate the simulation of reflection and refraction.*

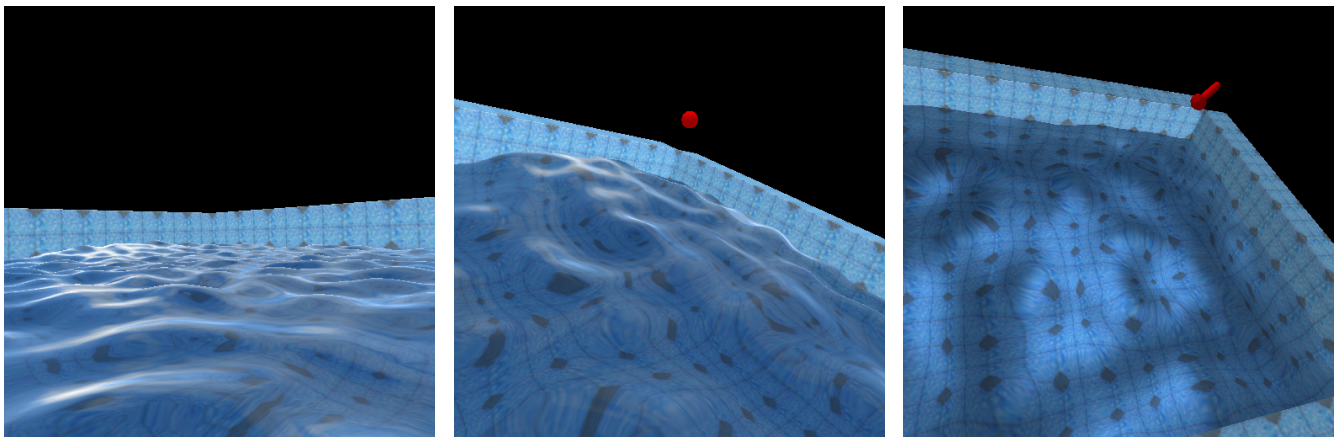

Figure 4: *The leftmost image show the fBm with high magnitude. In the middle image deep valleys were simulated by superimposing an appropriately scaled NURBS surface. The rightmost image illustrates the caustics approximation we implemented.*