Real Time Image Generation for Underwater Simulation

Milton T. S. Sakude, Edgar T. Yano
Instituto Tecnologico de Aeronautica – ITA
São Jose dos Campos, SP, Brazil
sakude@ita.br, yano@ita.br

Pedro S. C. R. Salles
Alpha Channel
São Paulo, SP, Brazil
pethrusx@gmail.com

ABSTRACT

Although graphics cards provide realistic real-time image generation for air-land scenarios, as built-in hardware, they are not appropriate for rendering underwater scenarios, which require special techniques to properly render the effects of liquid environment, loss of illumination and visibility, blur effects, spotlights, debris, caustics, and bubbles.

This paper presents some techniques for implementing real time underwater image generation that emulate loss of illumination and loss of visibility, spot light effects, and blur effects. The implementation is part of the development of an underwater visual simulation engine. OpenGL and OpenGL Shading Language (GLSL) are used in order to take advantage of GPU acceleration. The developed techniques use GLSL Shaders to perform the appropriate calculations, such as the computation of loss of visibility due to depth and distance. Spotlight Tyndall effects are emulated using an accumulated texture rendering technique. Loss of neatness (blur effect) is achieved seamlessly via a technique that blends the blurred image with the rendered image according to viewer-object distance.

Results show that the techniques presented here provide suitable image generation for underwater scenarios in real time by programming the GPU through GLSL.

ABOUT THE AUTHORS

Milton T. S. Sakude is an Assistant Professor at Instituto Tecnologico de Aeronautica-ITA. He received his Master's degree in Computer Science and Bachelor's degree in Mechanical Engineering from ITA. Research interests are in Computer Graphics, Simulation and Computer Security.

Edgar T. Yano is an Associate Professor at Instituto Tecnologico de Aeronautica-ITA. He received his Doctorate degree in Computer Engineering from ITA, his Master's degree in Computer Science from INPE, and Bachelor's degree in Mechanical Engineering from ITA. Research interests are in Software and Computer Security.

Pedro S. C. R. Salles is a Computer engineer at Alpha Channel. He has a Bachelor's degree in Computer Engineering from Instituto Tecnologico de Aeronautica. Research interests are in Computer Graphics, games and simulation.

Real Time Image Generation for Underwater Simulation

Milton T. S. Sakude, Edgar T. Yano
Instituto Tecnologico de Aeronautica – ITA
São Jose dos Campos, SP, Brazil
sakude@ita.br, yano@ita.br

Pedro S. C. R. Salles
Alpha Channel
São Paulo, SP, Brazil
pethrusx@gmail.com

INTRODUCTION

Although visual simulation technology has been employed extensively for military applications and games in land and air environments, little has been done for underwater simulation applications. Visual simulation technology generates real-time realistic images due to the implementation of Computer Graphics techniques through graphics cards. This technology has been spreading rapidly due to the low price of graphics cards used widely for gaming on personal computers.

Visual simulation technology is not provided by graphics cards only. Another key part is the visual simulation engine that runs on top of the graphics card and provides means to build simulation and to show visual effects. The simulation application, which establishes the entity behaviors and provides the user interface, runs on top of the simulation engine.

Since visual simulation technology has been developed mainly for land and air environments, it needs to be adapted through appropriate technology to generate underwater rendering. Such development is feasible now because graphics card can be programmed by using GLSL (OpenGL Shading Language) (Kessenish at al., 2010) or CUDA language (NVIDIA, 2010). This makes it possible to generate images in real time with different rendering from those hard coded in the graphics card.

Some work has been published in underwater image generation, for generating caustic and light ray (godrays) rendering mostly non real time (Iwasaki at al., 2002). Papadopoulos and Papaiannou (Papadopoulos and Papaiannou, 2009) developed a technique for generating caustics and godrays in real time, based on photon launching and image composition for blending shadow and filtering. Although these techniques generate realistic images, they spend significant amount of CPU processing power, so that they may compromise the processing of simulation behaviors and corresponding responses given by the visual.

THE UNDERWATER ENVIRONMENT

The underwater environment is somewhat similar to the air environment, because both water and air are fluids. Turbulence is more present in the water than in the air. In both environment, objects with less density tend to float, and those with heavier density tend to sink. Streams exist in both environments, but in water they are more noticeable.

The fog effect, due to the presence of particles in the environment, which absorb light, also exists in water; however, the loss of visibility occurs at much smaller distances than in air. Tiny particle similar to dust and debris can also be observed under water. Brightness diminishes with the distance in both mediums, but does much faster in water. Brightness also diminishes according to the depth in the underwater environment, which does not happen in air.

When a ray of light travels in water, a loss of intensity due to absorption by suspended particles occurs, and light can be diverted. Considering that a large number of rays can be diverted in a non-uniform manner, the result is the loss of sharpness of an object view. In water this sharpness can be lost easily, depending on how muddy water is, so that object appears blurred.

The caustic effect is unusual in the air environment, but it may be common in underwater settings such as under sun light.

In air, water falls as rain (drops) or as stream of water. Air has different behavior in water, where forms bubbles and rises. While bubbles may form in air, they are much more present in the water, such as when a diver breathes underwater.

In water an object moves at much lower speed than does in air or land. The visual effects of a moving

object in water may be quite different from that in air, such as a diver making turbulences and bubbles.

These similarities and differences must be taking into account to generate an effective simulation engine for an underwater scenario. Most of the known techniques for image generation can be used or adapted to do so. The challenge is to develop techniques that are not too CPU expensive.

This paper reports on the initial work done to develop a simulation engine to support underwater graphics imagery with features that do not overload the CPU and use GPU (Graphics Processing Unit) acceleration. This paper describes some techniques for implementing real time underwater image generation, including loss of illumination and loss of visibility, spot light effects, and blur effects.

UNDERWATER VISUAL TECHNIQUES

The first requirement for making underwater visual simulation is that Physics must apply. The Fresnel equation (Foley at al., 1996) must be considered when passing from air to the water environment. Loss in brightness must be considered both in depth and distance. The Navier Stokes equations (Teman, 2001) must be used with more weight because the behavior described by them is most noticeable in the water. However one must be aware that overloading the CPU/GPU with Physics computation may compromise the real time requirement. Although there is hope that the techniques for generating images and visual effects are made entirely on the GPU, they still consume a reasonable amount of CPU power.

In this paper an underwater visual simulation engine is described briefly. The first approach adopted in our research is to develop the appropriate techniques by using the GLSL for portability, and thereafter migrate them to GPU processing by using GPU parallel programming for better performance.

Underwater Engine

The Uw Engine (Underwater Engine) is an API (Application Programming Interface) prototype developed to implement the visual simulation process. The main classes that compose the API are: Camera, Effect, Element, Model, Scene, Light, Material, Shader, Input, and Primitives.

Element:
The Element class stores transformations, which can be translation, rotation, or scale. It also implements a hierarchy of transformations, where a child element inherits the transformations of its parent.

Model:
The Model class stores an imported 3D geometry file in *OBJ* format, as well as components, such as the Material and Element class instances.

Light:
The Light class represents the source of light that illuminates the scene. By default it is a point light. The class also supports spotlights. Ambient light, sun light and light parameters such as depth range and attenuation can be specified.

Scene:
The Scene class builds a complete scene with a list of models, elements, cameras, and lights. It also manages the creation of objects by loading and saving files. The handling of an instance of this class is made by the virtual class SceneControl.

Camera:
The Camera class represents a camera that captures the scene. There may have more than one camera, with possibility of camera switches. It extends the Element class. It can only be created by the Scene class.

Shader:
The Shader class manages the image generation by GLSL Shader algorithms. This class creates interfaces and abstracts the parameters passed for the GLSL Shader.

Material:
This class applies specified features to a model, and stores the basic parameters such as color and brightness.

Effect:
This class specifies a visual effect technique. It is an abstract class that must be implemented to create an effect that is applied to the underwater scenario.

Input:
This class manages user input, such as keyboard or mouse.

Primitives:
The Geometry class contains some procedural-ready, CSG (Constructive Solid Geometry) such as Cube and Sphere.

Some of these classes deserve detailed descriptions of implementation and techniques, such as Shader computation, the fog effect, the depth illumination effect, the sharpness loss effect and the Tyndall effect.

Shader Computation

A shader is written in GLSL for the implementation to render the scene. Shader computation is composed of three steps: Vertex Shader, Geometry Shader, and Fragment Shader.

The Vertex Shader step receives the vertex information that composes the primitives of the scene as input. Algorithms written for this step are applied on each of these vertices. Also the transformations of the vertices of world coordinates into view coordinates are performed, projecting the vertices of a 3D environment onto the 2D plane of the screen, with the necessary perspective corrections. The result of this step is passed on to the Geometry Shader to continue processing.

The Geometry-Shader step is optional. Connectivity information among vertices is also passed, which allows for the generation of new primitives and new vertices. Some of the most common applications include the refinement of geometries and the generation of Point Sprites. The result of this step, with vertices and primitives transformed, is then passed on to the Fragment Shader.

The Fragment-Shader step must contain an algorithm that deals with each pixel. It automatically performs the interpolation between the information contained in each vertex by the GPU. The algorithm in this step may modify the painting of the pixels. The result of this step is saved in the frame buffer. Once filled with paintings of all the primitives of the scene, the frame buffer is displayed on the screen by using the double buffering technique (SwapBuffer).

Illumination Computation

For sun light and light above the water surface, Fresnel's equations are applied to compute the refraction on surface water. The specified sunlight, ambient light, and point light are computed by using the Gouroud model or Phong model (Foley at al., 1996).

Fog Effect

In Computer Graphics fog effect is treated as a simplification of the actual physical phenomena. Commonly it is taken that the fog intensity depends only on the distance between the observer and the observed point. This effect results in loss of light visibility and in the transition from the object color to the fog color represented by the color of the muddy water.
The relationship between the color variation and distance to the observer can be linear or exponential. This work implements the linear approach.

The algorithm consists of the following steps:

- In the Vertex Shader:
  Compute the distances from each vertex to the observer, as z value (from DepthMap)
  Calculate the fog intensity as a function of distance and maximum fog distance (fogDist), limited between 0 and 1. The linear function is:

$$fogIntens = \frac{|z|}{\text{fogDist}}$$

- In the Fragment Shader:
  Calculate the new pixel color as a linear interpolation between actual pixel color and fog color, with the *fogIntens* parameter:

$$color = mix(color, fogColor, fogIntens)$$

Where mix is a linear interpolation between 1 and 2 using parameter 3 as the weight.

Figure 1 shows the effect of blue fog.

Figure 1. Fog effect in blue.

Depth Illumination Effect

Loss of illumination due to depth is similar to fog calculation:

- In the Vertex Shader:
  Compute the distance of the vertex and the water surface or light source position.
- In the Fragment Shader:

Compute the loss of light intensity according to the distance using linear interpolation (or other specified function). At the established threshold (light range) the intensity is null.
Correct the fogColor by blending with black color according to depth distance.
Blend the color with underwater color (blue).
Compute the color intensity according to the Gouroud (or the Phong) model.

Figure 2 shows the difference between an outside-water rendering and an underwater rendering. Figure 3 illustrates the underwater depth rendering effects.

(a) (b)

Figure 2. (a) Outside water rendering and (b) underwater rendering.

Figure 3. Underwater depth rendering.

Loss of Sharpness Effect

The Blur effect varies with distance from the viewed object to the observer, so that the farther the object is the more blurred it appears.
Generally, a blurred image is obtained by applying a filter, such as a Gaussian filter, to each pixel of the original image. The direct application of the filter fails because it is a discrete technique, which means that a seamless variation is hard to obtain. Yet, the size of the filter depends on the observer's distance from the object.
The solution to this problem is to blend a blurred image and the original image weighted by the object-observer distance, as proposed in (Rigues, Tatarchuck, Isidore, 2004).

The Blur Algorithm can be divided in four steps:

- Render the scene
  The first step is the normal rendering of the scene by using the Shader algorithms.
  Save the frame-buffer in Frame Buffer Objet (FBO).
  Save the z-buffer to compute distance between the observer and objects (DepthMap).

- Down sample the frame buffer
  Creates a new FBO, with half the resolution in X and Y, that is ¼ of the full resolution. To do this, the FBO from the previous step is simply copied with ¼ of the resolution. The purpose of this step is to optimize the algorithm: performing the interpolation of pixels in a quarter of the resolution is four times faster without an apparent loss of quality in the result.

- Apply Gaussian filters to the downsampled FBO.
  In this step the Gaussian filter is applied along the axis X and along the axis Y. The Gaussian filter is a method that calculates the weighted average of pixels around the pixel in question with the weights following a Gaussian curve. In this implementation it is used Gaussian filter of size 7 for calculating the color average. It is applied in X and Y direction using OpenGL operation.

- Composition
  This step blends the original frame buffer and the blurred one with the alpha value calculated as a function of object-observer distance saved in the DepthMap. This is performed by applying the following formulas:

$$alpha = mix(\min Dist, \max Dist, z)$$
$$newColor = fullColor + alpha * (blurColor - fullColor)$$

  fullColor and *blurColor* are pixel color from original frame buffer and blurred frame buffer respectively.

In Figure 4 an example of original image and blurred image can be seen. Figure 5 shows the effects of composition of original image and blurred image by applying the distance weight blending.

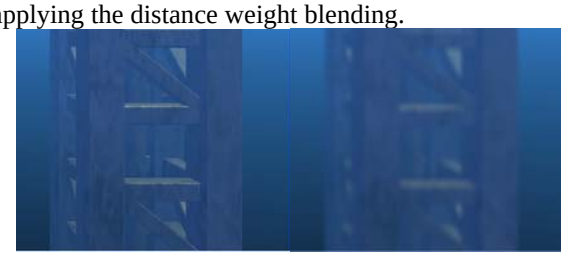

Figure 4. Original image and blurred image.

Figure 5. Composition of blurred image with original image by using distance weight blending

Tyndall Effect

The Tyndall effect is the name given to the scattering of a beam of light in a medium containing suspended particle, in this case water. In practice, the result in water is a greater brightness around a light point or in the case of a spotlight, the presence of greater brightness in the middle of the cone.

A way to obtain this effect is to use Physics to model it; however, the computational cost is prohibitive for a real-time application.

A simplified and efficient method of using OpenGL Point Sprites was chosen to model this effect. The end result is quite adequate as it may cover the most common situations of this effect in the water, such as the spotlight attached to the observer.
Point Sprite in OpenGL is a textured Quad facing the observer, positioned so that its center is the vertex set.

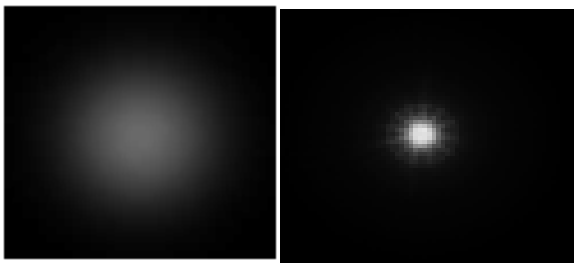

Figure 6. Point Sprites Textures.

The idea is to drag the Point Sprites along the spotlight cone with increasing radius and spacing, and decreasing opacity. Figures 7 and 8 illustrate the idea of the approach, where each circle is a Point Sprites.

For every new Point Sprites created, the pixel colors of the Point Sprites texture are added to the existing pixels in the scene by using alpha blending.

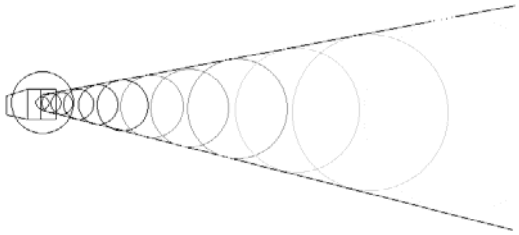

***Figure 7* - Idea of the algorithm of Tyndall effect**

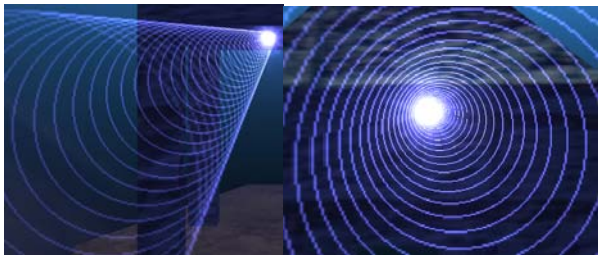

***Figure 7* – Point Sprites of the algorithm**

Spacing of the Point Stripes locations may grow with a quadratic function. The Point Sprites radius increase according to the defined spacing. There is also the calculation of attenuation that varies according to the distance from light source to the Point Sprites, thereby defining an increasing degree of transparency for Point Sprites far from the light source.

The algorithm parameters are adjusted empirically, but they also depend on the distance of the spotlight cone from the observer. The greater the cone is, the more Point Sprites must to be drawn.

Since overlapping pixels with transparency are added, there is an increase in brightness when there are several overlapping Point Sprites.

The texture of the Point Sprites texture can be changed for a better representation of the effect, as shown in Figure 8.

Although the focus of the algorithm is the volume of particles illuminated in the water, it is necessary to illuminate the objects contained within the cone of light. For this purpose Phong illumination model is used, because it produces smooth shading rendering.

Figure 8. Use of different texture for Point Sprites

Occlusion Improvement

One of the problems of the previous algorithm is that it ignores the light occlusion of the object. In order to use the Point Sprites idea, an improvement of the algorithm was implemented by using the OpenGL Stencil Buffer much as it is used to generate shadow (Crow, 1977; McGuire at al., 2003). The idea is to discount the object shadow portion from the Point Sprites drawing. The previous algorithm is used until there is no intersection between a sphere representing the Point Sprite and an object. When an intersection occurs, it passes to another algorithm that computes a stencil shadow mask, as illustrated in Figure 9. The object shadow (in black) is projected to the cone base. This shadow is stored in the stencil shadow.

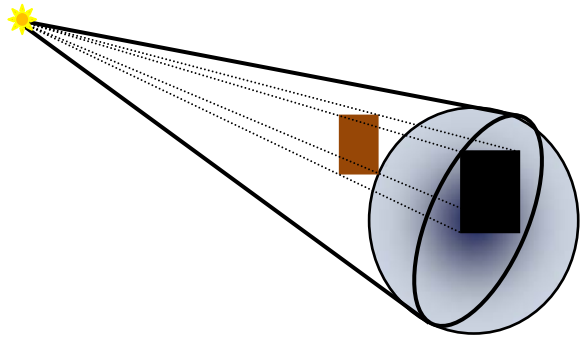

Figure 9. Illustration of Stencil mask generation

Then the Point Sprites are drawn with Stencil Mask. Shadow lines information (dashed in Figure 9) is held in a data structure for fast shadow calculation by using an incremental method in the next step.

Partial blocking the Tyndall Effect can be seen in Figure 10.

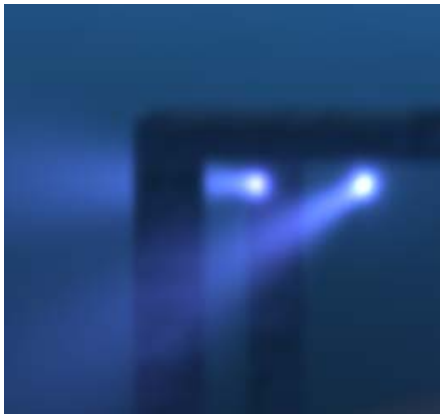

Figure 10. Partial blocking of Tyndall Effect

RESULTS

More image results can be seen in Figures 11, 12, 13, 14 and 15. Figure 11 shows an above water scene. Figure 12 shows an underwater scene, where the blue color mixing effect can be observed. Figure 13 illustrates loss of illumination due to depth. Figure 14 shows loss of sharpness due to distance that occurs with the fog effect. Figure 15 shows a scene with several spotlights.

Running in a computer powered by Intel Core 2 at 3.2 GHz and a GeForce GT 240 graphics card with 112 Cores, the frame rate exceeds 30 Hz, reaching more than 60 Hz for a resolution of 800 x 600 pixels.

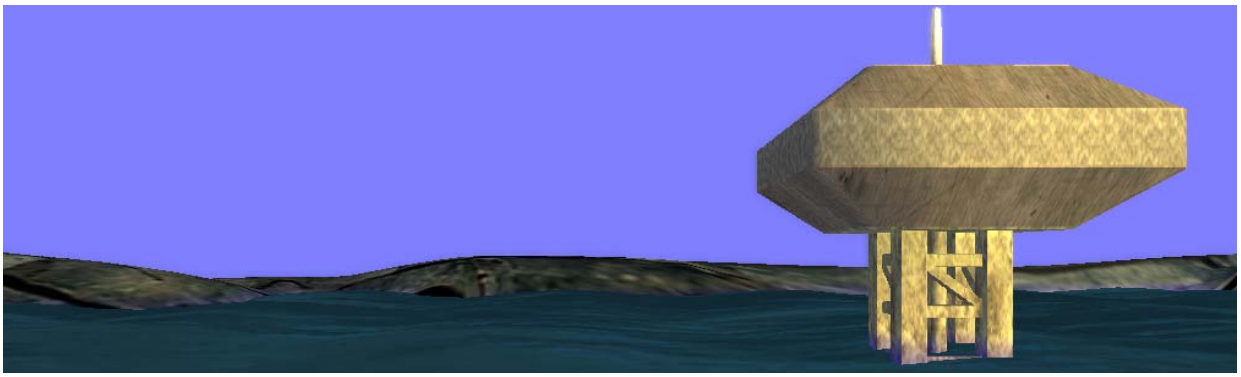

Figure 11. Above surface scene.

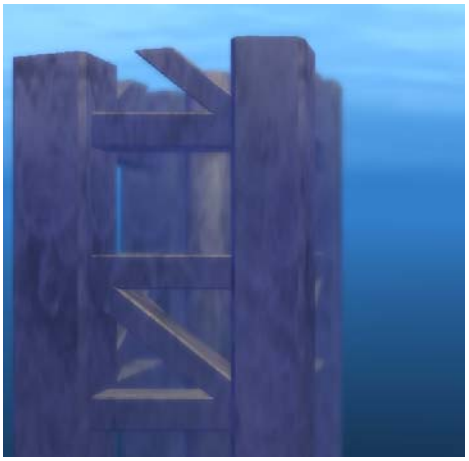

Figure 12. Near object image.

Figure 13. Loss of illumination due to depth

Figure 14. Far object image

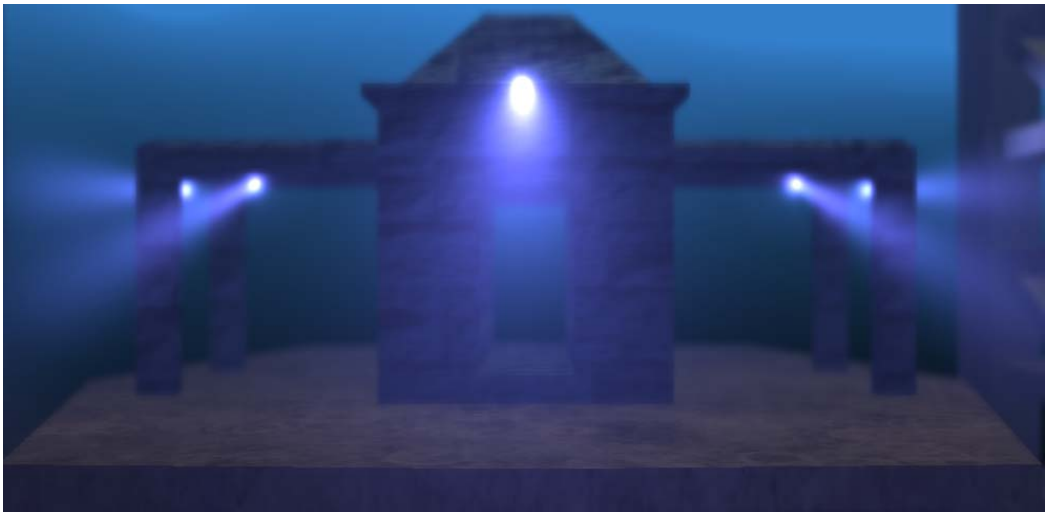

Figure 15. Scene with several spot lights

CONCLUSIONS AND FUTURE WORK

Some algorithms were developed for rendering underwater images in real time by using GLSL to take advantage of GPU acceleration. They compute fog effects, loss of illumination due to distance and depth, and blur effects. The blur effect appears seamless according to distance, without any flicking effects. Spotlight Tyndall effects are generated efficiently by using an approximate approach. Results show that the techniques presented here provide suitable image generation for underwater scenarios in real time by programming the GPU through GLSL.

The Underwater Engine framework presented in this work was developed for supporting underwater simulation. Further development may contemplate image generation for dealing with caustics, bubbles, particles, flow dynamics and underwater object dynamics.

REFERENCES

Crow, F. C (1977) "Shadow Algorithms for Computer Graphics", *Computer Graphics* (SIGGRAPH '77 Proceedings), vol. 11, no. 2, pp. 242-248.

Foley, J. D, van Dam, A., Feiner, S. K. and Highes, J. F. (1996). *Computer Graphics. Principles and Practice*. Addison-Wesley, Reading. 4th ed.

Iwasaki K., Dobashi, Y and Nishita, T. (2002). An Efficient Method for Rendering Underwater Optical Effects Using Graphics Hardware. *Computer Graphics Forum*. Vol. 21, No.4, pp. 1-11.

Kessenish, J. Baldwin, D, Rost, R. (2010). *OpenGL Shading Language*, version 4.10. Retrieved November, 2010, from http://www.opengl.org/documentation/glsl/

McGuire, M., Hughes, J.F., Egan, K.T, Kilgard, M.J. and Everitt, C. (2003) *Fast, Practical and Robust Shadows.* Retrieved November, 2010 from http://graphics.cs.brown.edu/games/FastShadows/index.html

NVIDIA, (2010) *CUDA Programming Guide*. Retrieved November, 2010 from http://developer.nvidia.com/category/zone/cuda-zone

Riguer, G., Tatarchuk, N.; Isidoro, J. (2004) Real-Time Depth of Field Simulation. In ENGEL, Wolfgangm *ShaderX2: Shader Programming Tips & Tricks with DirectX9*, pp. 529-556.

Temam, R. (2001), *Navier-Stokes Equations, Theory and Numerical Analysis*, AMS Chelsea, pp. 107-112.