바다물결 모형의 합성 및 GPU를 이용한 시뮬레이션

이 동 민† · 이 성 기††

요 약

컴퓨터 그래픽스로 재현되는 많은 자연현상 중의 하나인 바다는 주변 환경에 의해 계속해서 움직이며 복잡한 형태를 나타낼 뿐만 아니라 그 규모가 거대하기 때문에 만족스러운 영상을 얻기 위해서는 많은 계산시간을 필요로 한다. 본 논문에서는 GPU를 연산유닛으로 활용하여 무한히 넓은 바다표면의 움직임을 실시간으로 빠르게 시뮬레이션하고 사실적으로 렌더링하기 위한 방법을 제안한다. 제안하는 방법은 Gerstner 모델에 의해 2차원 투사 격자에서 생성된 저해상도의 메쉬로 바다의 전체적인 구조와 큰 물결을 표현하고, 스펙트럼 모델에 의해 2차원 균일 격자에서 생성된 높이 맵과 법선 맵을 사용하여 작은 물결과 자세한 수면의 모습을 표현한다. 전체 과정이 GPU에 의해 처리되기 때문에 CPU 자원을 다른 연산에 양보할 수 있을 뿐만 아니라 시스템 메모리와 그래픽스 하드웨어 사이에 기하정보(geometry data)의 이동이 없어 보다 빠른 렌더링이 가능하다. 제안하는 방법은 컴퓨터 게임과 같이 계산량이 많고 빠른 처리가 요구되는 실시간 애플리케이션에 활용 가능성이 크다.

키워드 : 자연현상, 바다물결 모형, 그래픽스 프로세싱 유닛, 실시간 물결 시뮬레이션

Synthesis of Ocean Wave Models and Simulation Using GPU

DongMin, Lee† · Sungkee, Lee††

ABSTRACT

Among many other CG generated natural scenes, the representation of ocean surfaces is one of the most complicated and time-consuming problem because of its large extent and complex surface movement. We present a hybrid method to represent and animate unbound deep-water ocean surfaces by utilizing graphics processor as both simulation and rendering core. Our technique is mainly based on spectral approaches that generate a high-detailed height field using Fourier transform on a 2D regular grid. Additionally, we incorporate Gerstner model and generate low-detailed height field on a 2D projected grid in order to represent large waves and main structure of ocean surface. There is no interruption between CPU and GPU, and no need to transfer simulation results from the system memory to graphics hardware because the entire simulation and rending processes are done on graphics processor. As a result we can synthesize and render realistic water surfaces in real-time. Proposed techniques are readily adoptable to real-time applications such as computer games that have heavy work load on CPU but still demand plausible natural scenes.

Key Words : Natural phenomena, Ocean wave models, GPU(Graphics Processing Unit), Real-time wave simulation

1. 서 론

최근 영화에서 쉽게 볼 수 있는 컴퓨터 그래픽스 (CG)에 의해 만들어진 영상은 실사와 구분이 힘들 정도로 섬세하고 사실적으로 표현되고 있다. 특히 영화 'deep impact', 'the day after tomorrow'에서 보여준 해일, 구름, 눈보라와 같은 대규모 자연현상의 재현은 보는 사람으로 하여금 실제와 같은 착각에 빠지도록 하여 영화에 더욱 몰입할 수 있도록 한다. 하지만 이와 같은 자연 현상은 그 규모가 크고 관련된 물리법칙이 복잡하기 때문에 만족스러운 영상을 컴퓨터로 재현해 내기 위해서는 많은 계산시간과 노력을 필요로 한다. 일반적으로 영화에서는 고성능의 CPU와 많은 메모리 공간을 가진 컴퓨터를 사용하여 광선 추적 (ray-tracing) 등의 비실시간 렌더링 알고리즘으로 영상을 미리 만들어 영화에 이용한다.

하지만 영화가 아닌 컴퓨터 게임에서 이러한 높은 수준의 영상을 생성하기란 매우 어려운 일이다. 이는 컴퓨터 게임이 영화와 달리 사람과의 상호작용에 따라 실시간으로 영상을 만들어내야 하기 때문이다. 또한 영상의 렌더링 외에도 게임의 진행을 위한 다른 많은 작업들을 동시에 처리하여야 한다. 하지만 컴퓨터 게임이 주로 실행되는 개인용 컴퓨터는 불과 수GB의 시스템 메모리와 3GHz 내외의 CPU를 가지고 있어 영화에서 사용되는 컴퓨터에 비해 성능이 떨어지기 때문에 더욱 만족스러운 영상을 재현해 내기가 어렵다.

CG에 의해 재현되는 여러 대규모 자연현상 중에서도 바

† 정 회 원 : (주)KOG 연구원
†† 종신회원 : 경북대학교 전자전기컴퓨터학부 교수
논문접수 : 2007년 8월 14일, 심사완료 : 2007년 11월 12일

다는 시각적인 효과가 크기 때문에 제대로 재현되었을 때 사람들에게 큰 감동을 줄 수 있다. 오래 전부터 물과 같은 유체의 움직임을 사실적으로 표현하기 위해 유체역학을 기반으로 한 다양한 방법들이 연구되어 왔다[1, 2]. 이들은 유체의 움직임을 지배하는 Navier-Stokes 방정식을 풀이하여 유체의 압력과 속도, 밀도 등의 변화를 물리적으로 정확하게 계산하고 이를 통해 유체와 관련된 여러 가지 복잡한 현상을 표현할 수 있다. 최근에는 이러한 유체 시뮬레이션도 GPU (Graphics Processing Unit)를 이용하여 속도를 개선하고 있으나 여전히 많은 계산시간을 필요로 하기 때문에 비교적 작은 규모의 유체에 대해서만 한정적으로 사용되고 있다[3].

이와는 달리 바다는 그 규모가 거대하기 때문에 유체역학을 바탕으로 표현하기 어렵기 때문에 실험적 모델(empirical model)을 사용하여 바다의 모습을 그럴듯하게 모방하여 표현하는 방향으로 연구가 되고 있다. 전통적으로 바다의 모습을 표현하기 위한 연구는 크게 Gerstner 모델과 스펙트럼 모델을 기반으로 한 연구로 나누어진다.

스펙트럼 물결 모델은 바다를 정현파의 중첩으로 보고 실제 바다에서 측정된 스펙트럼으로부터 Fourier 변환을 통해 바다를 표현하는 높이장을 생성하기 때문에 사실적인 바다의 모습을 표현할 수 있어 주로 영화에 많이 사용되고 있다. Gerstner 모델은 스펙트럼 모델과 달리 바다를 정현파의 중첩이 아닌 트로코이드의 중첩으로 표현한다. 트로코이드는 정현파와 달리 뾰족한 마루와 평평한 골을 가지고 있어 실제 파도와 같은 날카로운 모습을 표현하기에 적합하다.

Fournier는 Gerstner 모델을 확장하여 수심에 따라 트로코이드의 주파수를 변형함으로써 해안에 접근하는 파면(wave front)이 해안선에 따라 굴절되는 현상 및 물마루가 구부러지는 현상 등을 표현하였다[4]. 또한 트로코이드의 속도를 이용하여 물보라와 물거품을 표현하기 위한 방법을 제시하였다. Thon은 실제와 같은 바다의 모습을 표현하기 위해서는 트로코이드의 특성 값, 즉, 진폭과 주파수 등을 올바르게 선택하여야 함을 지적하고 트로코이드의 스펙트럼이 정현파의 스펙트럼과 유사함을 이용하여 Pierson-Moskowitz 스펙트럼으로부터 이들 특성 값을 유도하였다[5]. 이와 함께 계산량을 줄이고 거친 바다의 모습을 효과적으로 나타내기 위해 Pierson-Moskowitz 스펙트럼으로 필터링된 Perlin 잡음[6]을 사용하여 작은 물결에 의한 복잡한 수면의 모습을 표현하였다.

Hinsinger는 시뮬레이션과 렌더링 과정 모두에서 적응 기법 (adaptive scheme)을 사용하여 Gerstner 모델을 통해 실시간으로 바다의 모습을 표현하고자 하였다[7]. 화면상의 2차원 균일 격자를 수면으로 투사하여 생성된 투사 격자(projected grid)를 사용하여 바다를 나타내는 메쉬를 생성하고 격자 간격에 따라 중첩되는 트로코이드의 수를 조절하여 전체적으로 계산시간을 단축하였다.

처음으로 스펙트럼 모델을 사용하여 바다를 표현한 Mastin은 공간 영역의 실수 (real-number) 백색 잡음 (white noise)을 Fourier 변환하여 주파수 영역의 복소수 백색 잡음으로 변환하고 이를 Pierson-Moskowitz 스펙트럼으로 필터링하여 바다를 구성하는 정현파들의 진폭 값을 계산하였다[8]. 하지만 바다를 마루가 둥근 정현파의 중첩으로 표현하였기 때문에 주로 잔잔한 바다를 표현하는데 그쳤다. Tessendorf는 Pierson-Moskowitz 스펙트럼 대신에 확장된 Phillips 스펙트럼을 사용하여 파도의 방향성과 속도 등을 임의로 조절할 수 있도록 하였고, 수면 물결의 분산성 전파특성을 이용하여 뾰족한 물마루를 표현할 수 있는 방법을 제시하였다[9]. Jensen은 처음으로 스펙트럼 모델을 실시간 애플리케이션에 활용하기 위한 시도를 하였다. 물보라, 물거품, 화선 (caustics)등의 효과를 추가하고 Stam의 Navier-Stokes 방정식 풀이방법[1]을 이용하여 배에 의한 물결의 움직임을 표현하였다[10]. 하지만 전체 과정이 CPU에서 처리되어 계산 속도가 느리기 때문에 Fourier 변환의 해상도에 많은 제약이 따른다. Lanza는 스펙트럼 모델에 의해 높이 맵과 법선 맵을 생성하고 넓은 영역의 바다를 표현하고 연속적인 LOD을 제공하는 메쉬를 생성하기 위해 쿼드 (quad) 트리를 사용하였다[11].

가장 최근에 Mitchell은 스펙트럼 모델을 기반으로 모든 과정이 그래픽스 하드웨어에서 처리되는 바다 표현 기법을 제시하였다[12]. 크기와 해상도가 서로 다른 2 개의 Fourier 변환을 사용하여 메쉬를 변형하기 위한 높이 맵과 세부적인 묘사를 위한 법선 맵을 각각 생성하여 바다를 표현하였다. 스펙트럼 모델과 GPU를 활용하였다는 점에서 본 논문의 연구와 유사하지만 높이 맵과 화면에 그려지는 메쉬가 1:1 대응이 되도록 하여 상대적으로 좁은 영역의 바다를 표현하는데 치중하였다.

Gerstner 물결 모델은 바다를 연속함수 (continuous function)로 정의하여 무한히 넓은 바다를 표현할 수 있지만 실제와 같은 거친 바다의 모습을 나타내기 위해서는 많은 수의 트로코이드를 중첩해야하기 때문에 실시간으로 처리하기가 쉽지 않다. 스펙트럼 물결 모델은 실제 바다의 스펙트럼을 바탕으로 Fourier 변환을 통해 다수의 정현파의 중첩을 빠르게 계산할 수 있지만 역시 실시간으로 처리되기 위해서는 시뮬레이션 해상도에 제약이 따른다. 또한 높은 파도를 표현하기 어렵고 수평선까지 펼쳐지는 넓은 영역의 바다를 표현하기 위해서는 제한된 크기의 높이장 (height filed)을 연속하여 이어 붙여야 하기 때문에 반복 패턴이 뚜렷하게 보이는 단점이 있다. 두 실험적 모델 모두 유체역학을 이용한 방법에 비해서는 적은 양이기는 하지만 여전히 많은 양의 계산을 필요로 하기 때문에 현재의 하드웨어에서 실시간으로 처리하기 위해서는 영상의 질을 희생할 수밖에 없었다.

하지만 그래픽스 하드웨어의 발전으로 개인용 컴퓨터에 GPU라고 하는 추가적인 연산 장치가 탑재됨에 따라 연산능력이 획기적으로 증가하였고 이러한 문제를 해결하기위한 실마리를 제공하고 있다. GPU는 CPU와 달리 많은 연산유닛을 탑재하여 다수의 데이터를 동시에 처리할 수 있는 병

렬 프로세서이고 CPU와 독립적으로 연산을 처리할 수 있다. 또한 모든 연산이 부동소수점 데이터를 대상으로 하고 기본적으로 내적, 외적 등의 벡터 연산을 제공하기 때문에 본 논문의 바다물결 시뮬레이션과 같이 많은 양의 데이터를 한꺼번에 처리하여야 하는 작업에 그 활용가능성이 크다.

본 논문에서는 시뮬레이션과 렌더링 과정에서 GPU의 빠른 연산능력을 활용함으로써 높은 수준의 바다 영상을 실시간으로 생성하기 위한 방법을 제안한다. Gerstner 물결 모델을 기반으로 바다의 전체적인 구조와 큰 파도를 나타내는 저해상도의 높이장을 생성하고 스펙트럼 물결 모델을 기반으로 작은 파도에 의한 수면의 거친 모습을 표현하는 고해상도의 높이장을 생성하여 이를 합성함으로써 바다 영상을 생성한다. 또한 영상을 생성하는 모든 과정을 병렬 프로세서인 GPU로 처리함으로써 빠른 시뮬레이션이 가능할 뿐만 아니라 시뮬레이션 결과를 시스템 메모리로부터 그래픽스 하드웨어로 전송할 필요가 없기 때문에 더욱 빠른 렌더링을 가능하게 한다.

본 논문에서 대상으로 하는 바다는 무한히 넓은 공간에 펼쳐져 있는 심해의 바다로 수면의 물결은 바람에 의해서만 발생하며 수심에 의해서는 영향을 받지 않는 것으로 가정한다. 배나 섬과 같은 물체에 의한 물결의 변화도 고려에서 제외하였다. 그리고 본 논문에서 제안하는 방법은 수면을 단일 값의 높이장으로 표현하여 부서지는 파도나 접히는 파도에 대해서는 고려하지 않았다.

본 논문의 구성은 다음과 같다. 2 장에서는 스펙트럼 물결 모델과 Gerstner 물결 모델에 대해 먼저 설명하고 이어서 제안하는 합성 모델에 대해서 설명한다. 3 장에서는 제안하는 바다물결 모델의 GPU 구현 방법에 대해서 설명한다. 4 장에서는 제안하는 모델을 성능을 알아보기 위해서 각각 CPU와 GPU를 사용하는 두 개의 시스템으로 구현하고 성능을 분석하였다. 마지막으로 5 장에서 본 논문의 결론을 내린다.

2. 물결 모델

2.1 스펙트럼 물결 모델

편의상 스펙트럼 모델에 대한 설명에 사용되는 수식과 표기법은 Tessendorf의 연구[9]를 따른다. 파도가 없는 해수면은 평면상에 펼쳐져있고 축이 수면 위쪽을 가리키는 것으로 가정한다. 스펙트럼 모델은 바다가 통계적으로 일정한 형태를 유지하기 때문에 바다의 물결을 여러 정현파의 중첩으로 표현할 수 있다는 원리를 기반으로 한다. 실제 바다에서 배나 부표에 부착된 측정 장비를 이용하여 파도에 의한 수면의 높이 변화를 측정하고, 이렇게 수집된 데이터를 분석하여 바다를 구성하고 있는 정현파 스펙트럼을 추출한다. 측정된 높이 데이터를 정현파로 분해하는 데에는 Fourier 변환이 사용되고 분해된 스펙트럼으로부터 다시 높이장(height field)을 재구성하는 데에는 역 Fourier 변환이 사용된다.

임의의 시간 t, 임의의 위치 $X=(x,z)$에서 수면의 높이 $h(X,t)$는 시간에 따라 바뀌는 진폭을 가진 다수의 정현파의 합으로 정의된다 :

$$h(X,t)=\sum_K \tilde{h}(K,t)\exp(jK\cdot X) \tag{1}$$

이때 $K=(k_x,k_z)$는 2차원 파형벡터 (wave vector)로서 $k_x=2\pi n/L_x$, $k_z=2\pi m/L_z$이다. L_x, L_z는 각각 시뮬레이션 영역의 가로, 세로 크기이고, N, M 을 각각 2차원 Fourier 변환 영역의 가로, 세로 해상도라고 할 때, n, m은 $-N/2\le n<N/2$, $-M/2\le m<M/2$ 의 범위에 속하는 정수이다. 이에 따라 높이장은 2차원 균일격자의 각 격자점 $X=(nL_x/N,\ mL_z/M)$에서 계산된다.

Fourier 변환에 의해 계산된 높이장은 넓은 영역의 바다를 표현하기 위해 반복해서 이어 붙일 수 있어야 하고, 계산된 높이장의 각 높이 값이 실수가 되어야 한다. 이를 위해서는 주파수 영역의 입력 값, 즉, 진폭 값이 복소켤레 특성 (complex conjugation property)을 만족하는 복소수이어야 한다. Tessendorf는 이러한 조건을 만족하는 입력 값을 만들기 위해서 임의의 시간 t에 주파수 영역의 입력 값 $\tilde{h}(K,t)$를 다음과 같이 정의하였다 :

$$\tilde{h}(K,t)=\tilde{h_0}(K)\exp(j\omega(k)t)+\tilde{h_0^*}(K)\exp(-j\omega(k)t) \tag{2}$$

이렇게 함으로써 복소켤레 특성, $\tilde{h^*}(K,t)=\tilde{h}(-K,t)$를 만족하는 입력 값을 만들 수 있고, 이전의 높이장에 관계없이 특정 시간의 높이장을 생성할 수 있다. 이때 $w(k)$는 분산관계 (dispersion relation)에 의해 정해지는 물결의 이동속도이다. 분산관계에 의해 심해의 바다에서 바닥까지의 깊이와 표면 장력을 고려하지 않을 때 수면을 따라 움직이는 물결의 속도는 그 물결의 파장에 의해 결정된다 :

$$w(k)=\sqrt{gk} \tag{3}$$

g는 중력상수이고, $k=2\pi/\lambda$는 파장 λ에 의해 정해지는 파수 (wave number)이다. 이러한 분산 관계에 의해 바다는 인위적인 정현파의 중첩이 아닌 실제 바다와 같은 모습으로 움직이게 된다. $\tilde{h_0}(K)$는 시간 $t=0$에서의 초기 진폭값으로서 다음과 같이 백색 잡음 (white noise)을 Phillips 스펙트럼으로 필터링하여 생성한다 :

$$\tilde{h_0}(K)=1/\sqrt{2}\,(\xi_r+\xi_i)\sqrt{P_h(K)} \tag{4}$$

이때 ξ_r과 ξ_i은 각각 평균이 0이고 표준편차가 1인 가우스 (Gaussian) 난수 생성기에 의한 난수로서 2차원 영역의 백색 잡음을 형성한다.

$P_h(K)$는 바다 스펙트럼의 대표적인 수학적 모델인 Phillips 스펙트럼으로서 다음과 같이 정의된다 :

$$P_h(K) = A\frac{\exp(-1/(kL)^2)}{k^4}|\hat{K} \cdot \hat{w}|^2 \quad (5)$$

이때 A는 전체적인 스펙트럼의 크기를 조절하는 상수이고, $L = V^2/g$는 중력 가속도가 g일 때, 속력이 V인 바람에 의해 일어날 수 있는 파도의 최대 높이를 의미한다. k는 파수이고, $\hat{K}$는 정규화된 (normalized) 파형 벡터, $\hat{w}$는 정규화된 바람 방향 벡터로서 $|\hat{K} \cdot \hat{w}|$에 의해 바람 방향에 수직으로 움직이는 물결을 제거하게 된다.

높이장뿐만 아니라 렌더링할 때 필요한 법선 벡터도 Fourier 변환을 통해 계산할 수 있다 :

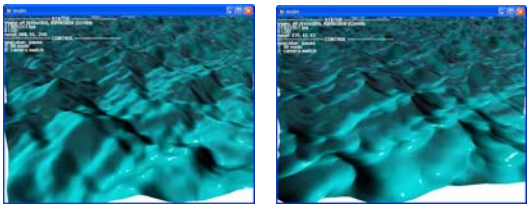
$$\epsilon(X,t) = \sum_K jK\tilde{h}(K,t)\exp(jK \cdot X) \quad (6)$$

수식 (6)은 수식 (1)을 x, z로 미분한 것으로서 x, z 방향의 접선의 기울기를 계산하게 된다. 이 두 기울기를 이용하여 높이장의 법선 벡터를 계산할 수 있다. 물론 이처럼 Fourier 변환을 통해서가 아니라 높이장을 계산한 후에 유한 차분법 (finite difference method)에 의해 법선 벡터를 계산할 수도 있으나 보다 좋은 렌더링 결과를 얻기 위해서는 Fourier 변환을 통해 계산된 해석학적 법선 벡터를 사용하는 것이 좋다.

또한 생성된 높이장은 정현파의 중첩이기 때문에 둥근 물마루를 가지는데 실제 바다는 파도가 거의 없이 잔잔한 경우에도 둥글기 보다는 뾰족한 물마루를 가진다. Tessendorf는 이를 표현하기 위해 다음과 같이 Fourier 변환에 의해 계산되는 변위벡터를 사용하였다 :

$$D(X,t) = \sum_K -j\frac{K}{k}\tilde{h}(K,t)\exp(jK \cdot X) \quad (7)$$

변위 벡터에 의해 변경된 격자의 위치는 $X + \alpha D(X,t)$가 된다. 이때 α는 변위 벡터에 의한 영향을 조절하는 상수이다. 이러한 변위 벡터를 통해 높이는 변화가 없지만 격자점의 수평 위치가 변경되어 (그림 1)에서처럼 뾰족한 물마루와 평평한 골을 표현할 수 있게 된다.



(a) (b)

(그림 1) 변위 벡터를 (a) 사용하지 않은 경우와 (b) 사용한 경우의 비교

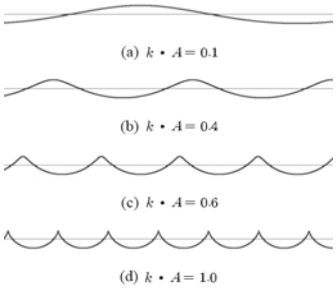
2.2 Gerstner 물결 모델

Gerstner 모델에서 바다의 물결은 안정시 위치를 중심으로 해수면에 수직으로 원운동하는 입자들의 움직임에 의한 곡선인 트로코이드의 중첩으로 표현된다. 안정시 위치가 $X_0 = (x_0, z_0)$인 입자는 임의의 시간 t에 진폭이 A인 물결들이 지나감에 따라 다음과 같이 움직인다 :

$$\begin{aligned} x &= x_0 + \sum A\frac{k_x}{|K|}\cos(K \cdot X_0 - \omega t), \\ y &= \sum A\sin(K \cdot X_0 - \omega t), \\ z &= z_0 + \sum A\frac{k_z}{|K|}\cos(K \cdot X_0 - \omega t). \end{aligned} \quad (8)$$

이때 $K = (k_x, k_z)$는 파형 벡터이고, ω는 앞서 설명한 분산 관계에 따라 물결의 이동속도를 결정한다.

Gerstner 모델에서 수면의 모양은 진폭과 파수에 의해 결정되는데, (그림 2)는 $k \cdot A$에 따른 파형의 변화를 보여준다. $k \cdot A > 1$인 경우에 물마루에서 수면이 서로 교차하는 부분이 생기기 때문에 이러한 경우가 생기지 않도록 주의하여야 한다.



(그림 2) $k \cdot A$에 따라 변하는 트로코이드의 모양

2.3 합성 물결 모델

실제 바다의 스펙트럼을 바탕으로 사실적인 바다를 나타낼 수 있어 영화에 자주 사용되고 있는 스펙트럼 모델은 많은 수의 정현파의 중첩을 빠르게 계산할 수 있지만 여전히 실시간으로 계산되기 위해서는 해상도에 많은 제약이 따른다. Tessendorf는 1+ GHz의 CPU를 사용하여 해상도 512x512의 높이 맵을 거의 실시간으로 생성할 수 있었다고 하였다. 이와 함께 영화 '워터 월드' 와 '타이타닉'에서는 해상도 2048x2048의 높이 맵을 사용하였고, 그 이상의 해상도에서는 부동소수점 표현범위의 한계 때문에 오차가 눈에 띄게 커질 수 있다고도 언급하고 있다[9].

이와 같은 해상도에의 제약은 사용되는 스펙트럼의 범위까지 제약한다. 수식 (1)에 의해 정해지는 스펙트럼의 범위는 시뮬레이션 영역의 크기와 격자의 해상도에 따라 변경된다. 즉, 파수 $k = \sqrt{k_x^2 + k_z^2} = \sqrt{(\frac{2\pi n}{L_x})^2 + (\frac{2\pi m}{L_z})^2}$는 시뮬레이션 영역의 크기 $L_x \times L_z$와 해상도 $N \times M$에 의해 $2\pi\sqrt{(\frac{1}{L_x})^2 + (\frac{1}{L_z})^2} \le k \le \sqrt{(\frac{\pi N}{L_x})^2 + (\frac{\pi M}{L_z})^2}$

로 제한되고, 파장 $\lambda = \frac{2\pi}{k}$의 범위도 $\frac{2}{\sqrt{(N/L_x)^2+(M/L_z)^2}} \le \lambda \le \frac{1}{\sqrt{(1/L_x)^2+(1/L_z)^2}}$로 제한된다. 수식을 간단히 하기 위해 $L_x = L_z = L$, $N = M$이라고 하면, 파장 λ의 범위는 $\sqrt{2}L/N \le \lambda < L$이 된다. 따라서 파장이 L보다 긴 파도는 표현할 수 없다. 앞서 말한 것처럼 현재의 하드웨어에서 실시간으로 높이장을 생성하기 위해서는 $N \le 512$로 제한되고, 사실적인 바다의 모습을 나타내기 위해서는 수m 정도의 짧은 파장을 지닌 정현파의 중첩이 다수 필요하기 때문에 시뮬레이션 영역의 크기도 $L \le 512$ 정도로 제한된다. 그렇기 때문에 짧은 파장의 물결들에 의한 거친 수면을 표현함과 동시에 큰 파장을 지닌 높은 파도를 표현하기가 어렵다.

물론 큰 파도를 표현하기 위해 격자간격과 해상도가 다른 또 하나의 Fourier 변환을 통해 추가로 높이장을 생성하여 합성하는 방법이 있겠지만 추가적인 Fourier 변환으로 인해 더 많은 계산시간을 필요하게 되고, 일반적으로 계산시간을 단축하기 위해 해상도를 낮추게 되는데 이 경우 넓은 격자간격 때문에 앨리어싱 (aliasing) 문제가 발생하여 영상의 질을 떨어뜨리게 된다. 더욱이 큰 파도에 의한 바다의 모습을 표현하기 위해서는 십여 개 정도의 소수의 물결만으로 충분히 사실적인 영상을 표현할 수 있기 때문에 이처럼 적은 수의 정현파의 중첩에 Fourier 변환을 사용하는 것은 적합하지 않다. 또한 스펙트럼 모델은 이처럼 제한된 크기의 높이장을 생성하기 때문에 보다 넓은 수면을 나타내기 위해 이를 연속하여 이어붙이게 되는데, 시야가 넓은 경우 반복패턴이 뚜렷하게 보이는 단점이 있다.

반면 Gerstner 모델은 스펙트럼 모델에 비해 계산량이 많아서 빠른 시간 내에 많은 수의 정현파의 중첩을 계산할 수는 없지만 Fourier 변환이 아닌 덧셈에 의해 계산되기 때문에 좁은 범위의 스펙트럼이 아닌 임의의 스펙트럼을 가진 정현파의 중첩을 쉽게 계산할 수 있다. 또한 주파수 영역이 아닌 공간 영역에서 계산이 이루어지기 때문에 시뮬레이션 도중에 새로운 스펙트럼을 추가하거나 변경할 수 있어 실시간으로 바다 표면의 특성을 변화시킬 수 있다. 뿐만 아니라 스펙트럼 모델에서 표현하기 어려운 한 방향으로 이동하는 평행한 물결, 빗방울에 의해 생기는 원형 물결 등을 표현할 수 있다. 또한 스펙트럼 모델과 달리 바다를 높이장이 아닌 연속 함수로 정의하기 때문에 반복패턴없이 넓은 바다를 나타낼 수 있다.

앞서 설명한 두 바다 모델을 적절하게 합성함으로써 각각의 단점을 보완하고 장점을 살릴 수 있다. 상대적으로 계산량이 많은 Gerstner 모델은 저해상도의 격자에서 소수의 큰 물결만을 중첩하여 바다의 전체적인 구조 및 큰 파도를 표현하고 스펙트럼 모델은 상대적으로 고해상도의 격자에서 작은 물결, 즉, 짧은 파장의 스펙트럼만을 사용하여 바다의 거칠고 세세한 모습을 표현한다. 이렇게 함으로써 잔잔한 바다뿐만 아니라 파도가 높고 거친 바다도 쉽게 표현할 수 있고, 스펙트럼 모델에 의해 생성되는 높이장을 이어 붙였을 때 보이는 반복패턴을 연속함수로 정의되는 Gerstner 모델에 의해 희석시킬 수 있다.

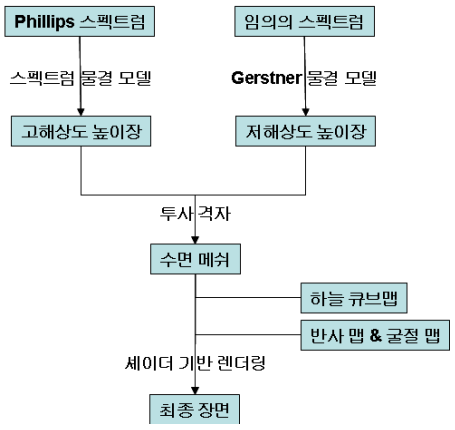
기존에도 이와 같이 합성 모델을 통해 바다를 시뮬레이션 하기 위한 시도들이 있었다. 앞서 말한 것처럼 두 개의 별도의 Fourier 변환에 의해 큰 파도를 나타내기 위한 저해상도의 높이 맵과 수면의 모습을 자세하게 나타내기 위한 고해상도의 높이 맵을 각각 생성하고 합성하여 바다를 표현할 수 있는데[9], 이 경우 추가적인 Fourier 변환으로 인해 계산량이 늘어나고 저해상도의 높이 맵에서 큰 격자간격으로 인한 앨리어싱 문제가 발생한다. 또한 시야가 넓은 경우 반복 패턴이 보이는 문제점을 해결할 수 없다. 하지만 제안하는 합성모델은 Gerstner 모델에 의해 바다의 전체적인 구조와 큰 파도를 표현하기 때문에 격자간격에 관계없이 반복패턴이 없는 연속적인 바다의 모습을 표현할 수 있고 적은 수의 물결만으로도 큰 파도를 표현할 수 있기 때문에 계산량도 크게 늘어나지 않는다.

Thon은 Gerstner 모델에 의해 저해상도의 바다 메쉬를 구성하고 그 위에 Perlin 잡음을 사용한 3차원 난류 함수(turbulence function)를 덧입히는 방식을 사용하였다. 하지만 작은 물결의 움직임을 단순히 Perlin 잡음을 횡이동하여 나타내었기 때문에 실제 바다와 같은 물결의 움직임을 표현할 수 없고 뾰족한 물마루를 표현하기 어렵다는 단점이 있다. 하지만 제안하는 합성모델에서는 정현파와 트로코이드의 위상을 수면 물결의 분산 특성을 이용하여 변화시킴으로써 수면의 움직임을 표현하기 때문에 매우 사실적인 바다의 움직임을 표현할 수 있다.

3. 바다물결 시뮬레이션 및 렌더링

3.1 시스템 개요

구현된 바다물결 시뮬레이션 및 렌더링 시스템의 구조는 (그림 3)과 같다. 전체 과정은 크게 높이 맵과 법선 맵을 생성하기 위한 시뮬레이션 과정과 생성된 높이 맵을 샘플링하여 메쉬를 생성하고 반사 맵과 굴절 맵을 생성하는 렌더링 준비과정, 그리고 생성된 메쉬와 맵을 사용하여 바다를 셰이딩하는 렌더링 과정으로 나누어진다.



(그림 3) 바다물결 시뮬레이션 및 렌더링 과정 전체 흐름도

먼저 시뮬레이션 과정에서는 Gerstner 모델에 의해 바다의 전체적인 형태와 큰 파도의 모습을 나타내기 위한 저해상도의 높이 맵과 법선 맵을, 스펙트럼 모델을 통해 바다의 거친 수면과 작은 파도를 자세히 표현하기 위한 고해상도의 높이 맵과 법선 맵을 생성한다. 모든 높이 맵과 법선 맵은 픽셀 셰이더를 이용하여 CPU와 독립적으로 GPU에 의해 생성되고 그래픽스 하드웨어상에 텍스쳐 형태로 저장되기 때문에 렌더링에 바로 사용이 가능하다.

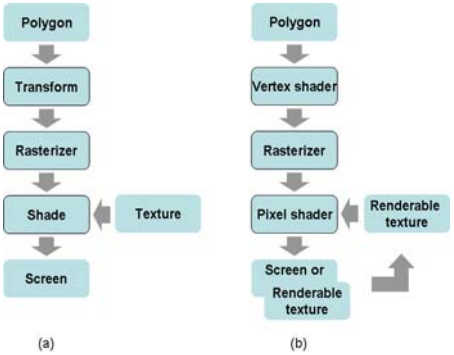
렌더링 준비과정에서는 화면상의 균일 격자를 수면으로 투사하여 만들어진 투사 격자의 각 격자점에서 앞서 생성된 높이 맵과 법선 맵을 샘플링하여 바다를 나타내는 메쉬를 구성한다. 이처럼 투사 격자를 사용함으로써 지형 렌더링에 사용되는 쿼드트리나 옥트리와 같은 다른 LOD 구조의 도움이 없이도 매우 넓은 영역의 바다를 표현할 수 있고, 실시간으로 자유롭게 시점을 이동하면서 바다를 렌더링 할 수 있다. 또한 배나 섬과 같이 수면에 가까이 있는 물체가 수면에 반사되는 모습을 표현하기 위한 반사 맵, 수면 밑의 물체가 굴절되어 보이는 모습을 나타내기 위한 굴절 맵을 매 프레임마다 실시간으로 생성하여 사용한다. 그리고 수면에 반사되는 하늘의 모습을 나타내기 위해 큐브 맵을 이용한 환경맵핑 기법을 사용하였다. 본 연구에서는 고정된 하늘 큐브 맵을 사용하여 시뮬레이션이 시작될 때 한번만 생성되도록 하였는데, 요즘에는 해나 구름의 움직임을 나타내기 위해 애니메이션되는 큐브 맵이 자주 사용되고 있는데 이처럼 실시간으로 큐브 맵을 생성하도록 할 수도 있다.

렌더링 과정에서는 이렇게 실시간으로 생성된 실시간 텍스쳐 맵들을 사용하여 바다 표면에 대한 셰이딩이 이루어진다. 수면에 의해 반사되어 보이는 빛과 굴절되어 보이는 빛의 양은 수면과 시선이 이루는 각도에 따라 크게 달라지는데, Fresnel 항으로 정의된 이러한 특성에 따라 반사 맵과 굴절 맵은 적절히 보간되어 바다의 색깔을 결정하게 된다. 정점 셰이더와 픽셀 셰이더를 이용하여 이러한 특수한 셰이딩 효과를 표현할 수 있다.

3.2 범용 GPU연산을 위한 프로그래밍 기법

과거의 그래픽스 하드웨어는 (그림 4(a))와 같이 고정된 파이프라인을 따라 폴리곤 데이터를 변환하고, 텍스쳐와 법선 벡터를 이용한 셰이딩에 의해 화면에 나타나는 최종 픽셀의 색상을 결정하는 역할을 하였다. 하지만 근래에 등장한 그래픽스 하드웨어는 (그림 4(b))에서 처럼 정점 셰이더와 픽셀 셰이더를 사용하여 폴리곤 데이터의 변환과 셰이딩을 자유롭게 프로그래밍 하는 것이 가능해졌다.

이와 더불어 32bit 부동 소수점 데이터를 저장할 수 있고 실시간으로 갱신이 가능한 렌더러블 텍스쳐의 등장으로 인해 GPU에 의해 계산된 결과를 텍스쳐 메모리에 저장하고, 다른 연산에서 이를 다시 사용할 수 있게 됨에 따라 GPU를 렌더링만이 아닌 범용연산에 활용할 수 있는 길이 열리게 되었다.

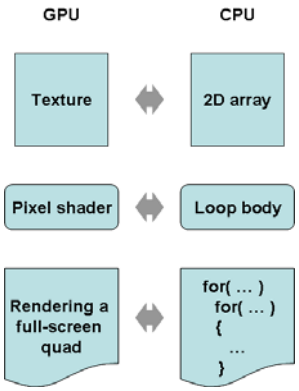


(그림 4) (a) 고정 그래픽스 파이프라인과 (b) 프로그래밍 가능한 그래픽스 파이프라인

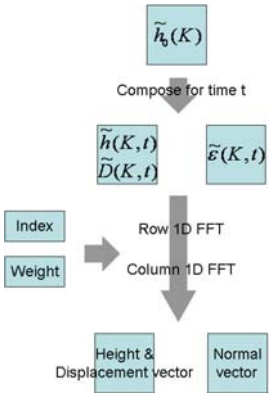
(그림 4(b))의 그래픽스 파이프라인에서 각각이 차지하는 위치를 보면 알 수 있듯이 정점 셰이더는 주로 3D 폴리곤 데이터를 스트림 형태로 전달받아 여러 가지 변환행렬과의 곱셈을 통해 화면상의 2D 폴리곤으로 변경하는 역할을 하고, 픽셀 셰이더는 정점 셰이더로 부터 전달받은 2D 폴리곤 내부의 색상을 텍스쳐 및 법선 벡터를 사용해 계산하는 역할을 한다. 정점 셰이더, 픽셀 셰이더 모두 자유로운 프로그래밍이 가능하기 때문에 범용연산의 구현에 활용할 수 있지만, 현재의 그래픽스 하드웨어는 픽셀 셰이더가 보다 많은 연산 유닛을 가지고 있고 연산 결과를 바로 텍스쳐에 저장할 수 있다는 장점이 있기 때문에 정점 셰이더보다 범용연산의 구현에 많이 사용되고 있다. 본 논문의 모든 시뮬레이션도 픽셀 셰이더에 의해 구현되었다.

렌더러블 텍스쳐는 일반적인 텍스쳐처럼 샘플링되어 폴리곤 내부의 색상을 결정하는데 사용될 수도 있고 프레임 버퍼를 대신한 렌더링 타겟 (rendering target)으로 사용되어 픽셀 셰이더에 의해 렌더링된 결과를 저장할 수도 있다. 렌더러블 텍스쳐의 각 텍셀 (texel)은 최대 4개의, 즉, RGBA 채널을 가지며, 각 채널은 8bit 정수형 데이터뿐만 아니라 32bit 부동소수점 데이터를 저장할 수도 있다.

이처럼 GPU가 새로운 범용연산장치로 각광받게 된 이유는 GPU는 한번에 하나의 연산만을 처리할 수 있는 CPU와 달리 많은 수의 연산유닛을 가지고 있어 다수의 데이터를 한꺼번에 처리하는 것이 가능한 병렬 프로세서이기 때문이다. 또한 기본적으로 벡터연산을 기반으로 하기 때문에 내적, 외적과 같은 다양한 벡터연산 인스트럭션을 제공하고, 다수의 스칼라 값을 벡터로 만들어 한번에 연산하는 것이 가능해 더욱 빠른 처리속도를 기대할 수 있다. 뿐만 아니라 모든 연산이 부동소수점 데이터를 대상으로 처리되기 때문에 유체 시뮬레이션과 같은 복잡한 수식의 계산에 적합하다. 게다가 GPU는 CPU와 독립적으로 실행되기 때문에 CPU에 걸리는 연산부하를 분산시키는 역할도 한다. 이러한 GPU를 이용한 범용연산은 비단 컴퓨터 그래픽스 분야뿐만 아니라 데이터베이스, 네트워크 등 여러 분야에서 폭넓게 활용되고 있다.



(그림 5) GPU와 CPU의 프로그래밍 모델 비교



(그림 6) 스펙트럼 모델의 시뮬레이션 과정

앞서 말한바와 같이 GPU는 벡터연산에 강하고 병렬처리가 가능하기 때문에 많은 양의 데이터에 대해 동일한 연산을 수행해야 하는 작업에서 CPU보다 빠른 처리속도를 기대할 수 있다. 이러한 작업을 예로 CPU와 GPU의 범용연산 프로그래밍 모델을 비교해보면 (그림 5)와 같다.

GPU에서는 많은 양의 데이터를 저장하고 빠르게 접근하기 위한 메모리로 텍스쳐를 사용한다. 일반적인 텍스쳐와 함께 계산결과의 갱신을 위해 렌더러블 텍스쳐를 데이터의 저장공간으로 활용한다. 이렇게 2차원 텍스쳐에 저장된 데이터에 대한 연산은 픽셀 셰이더를 사용한 사각형 그리기에 의해 실행된다. 텍스쳐의 크기와 동일한 사각형을 그림으로써 사각형 내부의 모든 픽셀이 렌더링되고 결과적으로 텍스쳐의 모든 텍셀에 대해 동일한 연산이 실행되게 된다. 즉, CPU 프로그래밍 모델에서 2차원 배열에 대한 이중 반복문은 GPU의 사각형 그리기에 대응되고 반복문내에서 실행되는 코드는 GPU 프로그래밍 모델의 픽셀 셰이더에 대응된다.

3.3 스펙트럼 모델의 구현

스펙트럼 모델에 의한 시뮬레이션 과정은 (그림 6)과 같고, 매 프레임마다 변화된 높이 맵과 법선 맵을 생성함으로써 이루어진다. 초기 진폭 데이터 $\tilde{h_0}(K)$가 저장되어 있는 텍스쳐를 입력으로 하여 매 프레임마다 분산 전파특성에 의해 변화된 진폭 값을 생성하고, 이를 Fourier 변환을 통해 높이 맵과 법선 맵으로 변환한다. 현재의 그래픽스 하드웨어는 MRT (Multiple Rendering Target) 기능을 지원하여 최대 4개의 렌더러블 텍스쳐에 동시에 서로 다른 결과 값을 출력하는 것이 가능하다. 이를 이용하여 높이 맵과 법선 맵, 변위 맵을 생성하기 위한 3개의 Fourier 변환을 한 번에 계산할 수 있다.

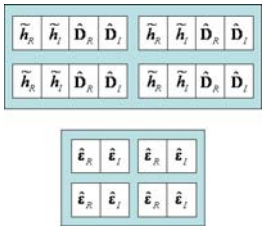
수식 (4)의 초기 진폭 값 $\tilde{h_0}(K)$는 시뮬레이션 도중에 값이 변경되지 않기 때문에 CPU에서 시뮬레이션의 시작 전에 미리 계산되어 텍스쳐의 형태로 저장된다. 이 텍스쳐를 이용하여 매 프레임마다 높이 맵, 법선 맵, 변위 맵을 생성하기 위한 Fourier 변환의 입력 값 $\tilde{h}(K,t)$, $\tilde{\epsilon}(K,t)$, $\tilde{D}(K,t)$를 GPU에서 계산한다.

앞에서 설명한 것처럼 이들 입력 값은 복소켤레 특성을 만족하기 때문에 Fourier 변환에 의해 계산된 결과는 실수값이 된다. 이러한 경우에 Fourier 변환의 선형성을 이용하여 2개의 Fourier 변환을 1개의 Fourier 변환으로 바꾸어 계산하는 것이 가능하다. 예를 들면 2개의 입력 데이터 $F(u)$, $G(u)$를 Fourier 변환하고자 할 때, 둘을 조합하여 하나의 새로운 입력 데이터 $H(u)=F(u)+jG(u)$를 만들어 Fourier 변환하면 $h(u)=f(u)+jg(u)$를 얻을 수 있다. 이때 계산된 $f(u)$, $g(u)$는 각각 실수이기 때문에 추가적인 처리과정이 없이도 $h(u)$로부터 $f(u)$와 $g(u)$를 추출할 수 있고 결과적으로 두 개의 Fourier 변환을 한 번에 계산할 수 있다.

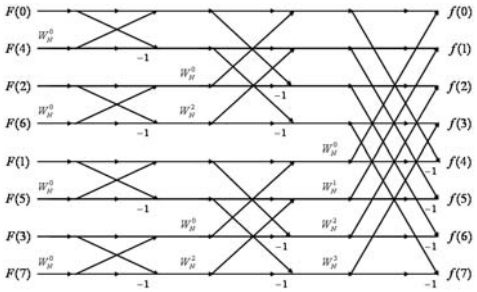
이를 이용하여 수식 (6), (7)의 법선 벡터와 변위 벡터의 x, z 성분을 수식 (9)에서처럼 각각 하나의 복소수로 만들어 Fourier 변환한다. 이렇게 두 복소수를 하나로 압축함으로써 높이 맵과 변위 맵을 (그림 7)과 같이 하나의 RGBA 텍스쳐에 저장할 수 있어 동시에 처리가 가능할 뿐만 아니라 높이 맵과 변위 맵은 렌더링할 때 정점 셰이더에서 같이 샘플링 되기 때문에 두 개의 맵에 나누어 저장하는 것 보다 샘플링 시간을 단축시킬 수 있다. 법선 맵은 픽셀 셰이더에서 따로 샘플링 되기 때문에 (그림 7)에서처럼 별도의 RG 텍스쳐에 저장된다.

$$\hat{D}=\widetilde{D}_x+j\widetilde{D}_z \qquad \hat{\epsilon}=\hat{\epsilon}_x+j\hat{\epsilon}_z \tag{9}$$

시뮬레이션의 가장 핵심이 되는 2차원 영역에 대한 Fourier 변환은 1차원 Fourier 변환을 각각 가로방향과 세로방향에 대해 연속적으로 적용함으로 처리된다.



(그림 7) 텍스쳐에 저장된 데이터의 배열 형태



(그림 8) 8개 데이터 radix-2 DIT FFT 흐름도

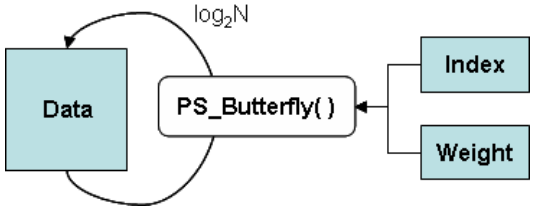
FFT (Fast Fourier Transform)의 구현에는 하나의 Fourier 변환을 크기가 절반인 두개의 Fourier 변환으로 나누어 계산할 수 있다는 Danielson-Lanzcos 정리에 따라 GPU의 병렬연산 구조에 적합하고 구현하기 쉬운 radix-2 DIT (Decimation-In-Time) FFT 알고리즘을 이용하였다[13]. (그림 8)은 8개의 입력 데이터에 대한 1차원 radix-2 DIT Fourier 변환의 흐름도이다. 이때 $W^x = \exp(j2\pi x/N)$이다.

N 개의 입력 데이터에 대해 DIT FFT는 $\log_2 N$ 개의 나비 단계 (butterfly stage)로 구성되고 각 나비단계 마다 이전 단계의 계산 결과를 입력으로 하여 W^x와 곱하여 더하는 방식으로 계산된다. 특히 첫 번째 나비 단계에서는 최종 계산된 결과의 순서를 입력 데이터의 순서와 같게 만들기 위해서 bit-reverse 순으로 입력 데이터를 재배열한다. 예를 들어 데이터를 가리키는 인덱스를 이진수로 표현하였을 때 $b_0b_1b_2$인 데이터는 bit-reverse 순으로 재배열하면 $b_2b_1b_0$의 위치로 옮겨진다.

(그림 9)는 GPU로 구현된 1차원 Fourier 변환의 흐름도입니다. 각 단계에서 곱해지는 W^x와 더해질 입력 데이터의 쌍을 정해주는 색인은 CPU에서 미리 계산되어 크기가 $N\times\log_2 N$ 인 2차원 텍스쳐에 저장되어 사용된다. 입력 데이터의 쌍을 가리키는 색인과 W^x를 이용하여 픽셀 셰이더에서 입력 데이터를 샘플링하여 각 나비 단계에서 필요한 계산을 수행한다.

3.4 Gerstner 모델의 구현

Gerstner 모델은 바다를 연속함수로 정의하지만 실제로 화면에 렌더링 되기 위해서는 높이 맵과 마찬가지로 이산 격자에서 샘플링 되어야 한다. 본 논문에서는 넓은 영역의 바다를 표현하기 위해 투사 격자를 사용하여 Gerstner 모델을 샘플링한다. 이렇게 함으로써 카메라의



(그림 9) GPU로 구현된 FFT의 흐름도

(그림 10) 투사 격자의 예. 참고문헌 [14].

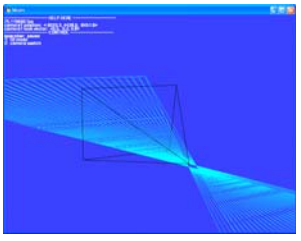
움직임에 제약없이 자유롭게 넓은 영역의 바다를 나타내는 메쉬를 생성할 수 있고 시점에 가까운 바다를 멀리 있는 바다보다 더 많은 수의 정점을 집중시켜 표현함으로써 자동으로 LOD를 제공할 수 있다.

투사 격자란 (그림 10)과 같이 화면상의 2차원 균일격자를 수면으로 투사하여 생성된 비균일 격자를 의미한다. 투사 격자 내부에 있는 격자점의 위치는 동차 좌표계에서 선형보간에 의해 계산된다 :

$$(x, y, z, 1) = (x_0, y_0, z_0, w_0) + s \bullet (\Delta x, \Delta y, \Delta z, \Delta w)$$
$$= \left(\frac{x_0 + s \bullet \Delta x}{w_0 + s \bullet \Delta w}, \frac{y_0 + s \bullet \Delta y}{w_0 + s \bullet \Delta w}, \frac{z_0 + s \bullet \Delta z}{w_0 + s \bullet \Delta w}, 1 \right). \quad (11)$$

Johanson은 시점이 수면에 가깝고 시선이 xz평면과 평행해짐에 따라 소위 역화 (back-firing)라고 하는 현상으로 잘못된 투사 격자가 생성된다고 지적하고 원래 시점보다 높은 곳에서 수평선이 아닌 수면 아래쪽을 향하는 새로운 투사 절두체를 만들어 사용함으로써 이러한 오류를 해결하고자 하였다[14].

하지만 실제로 이러한 문제가 생기는 원인은 컴퓨터의 부동소수점 표현 범위의 한계에 의해 발생한다. 투사 격자의 생성과정에서 4차원 동차 좌표를 3차원 공간의 좌표로 변환할 때 4차원 동차좌표의 마지막 성분의 값, w로 나머지 세 성분을 나누어주게 되는데 선형 보간에 의해 계산된 w의 값이 매우 작기 때문에 이것으로 나머지 성분의 큰 값을 나누었을때 오버플로우가 발생하여 (그림 11)과 같은 비정상적인 투사격자가 생성된다. 따라서 별도의 투사 절두체를 생성할 필요 없이 w의 값을 임의의 한계 값 (threshold)보다 작아지지 않도록 제약하는 것만으로도 이러한 오류의 발생을 막을 수 있다.



(그림 11) 오버플로(overflow)로 인해 오류가 생긴 투사격자

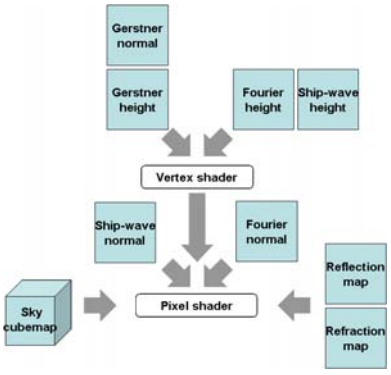
```
void PS_Gerstner(in half2 coord0 : TEXCOORD0,
                 out float4 OUT0 : COLOR0)
{
  float4 pos = lerp( lerp(corner0, corner1, coord0.x),
          lerp(corner3, corner2, coord0.x), coord0.y);
  pos = pos / pos.w;
  for( int i=0; i<WAVE_COUNT/4 ; i++ )
  {
    float4  theta  =  kx[i]*pos.x  +  kz[i]*pos.z  -
omega[i]*t;
    float4 SINE = mag[i]*sin(theta);
    float4 DX += dirx[i]*SINE;
    float4 DZ += dirz[i]*SINE;
    float4 Y  += mag[i]*cos(theta);
  }
  OUT0.x = pos.x - (DX.r+DX.g+DX.b+DX.a);
  OUT0.z = pos.z - (DZ.r+DZ.g+DZ.b+DZ.a);
  OUT0.y = Y.r + Y.g + Y.b + Y.a;
  OUT0.w = 1;
}
```

(그림 12) Gerstner 모델의 시뮬레이션 픽셀셰이더 코드

이렇게 생성된 투사 격자위의 각 격자점에서 Gerstner 모델에 의해 높이 맵과 법선맵을 생성한다. Gerstner 모델의 높이 맵을 생성하기 위한 코드는 다음과 같이 복잡한 알고리즘없이 단순한 덧셈으로 구현된다. CPU 코드와의 차이점은 (그림 12)의 픽셀 셰이더 코드에서 처럼 GPU의 벡터연산 기능을 활용하기 때문에 4개의 부동소수점 데이터에 대한 연산을 한 번에 할 수 있다는 점이다.

3.5 바다물결 렌더링

전체 렌더링 과정은 (그림 13)과 같다. 모든 높이 맵은 정점 셰이더에서 샘플링되어 메쉬를 변형하는데 사용되고 Gerstner 모델에 의한 법선 맵도 투사 격자의 각 격자점에서만 계산되었기 때문에 정점 셰이더에서 샘플링되고 이후 픽셀 셰이더에서 샘플링되는 나머지 법선 맵과의 합성을 위해 픽셀 셰이더로 전달된다. 이렇게 샘플링되어 합성된 법선 벡터를 이용하여 수면에 반사된 빛과 굴절된 빛의 양을 결정하는 Fresnel 항을 계산하고 수면에 반사된 하늘을 나타내는 큐브 맵과 실시간으로 생성된 반사 맵, 굴절 맵을 샘플링하여 Fresnel 항에 의해 선형보간함으로써 바다의 색상을 결정하게 된다.



(그림 13) 렌더링 과정의 전체 흐름도

정점 셰이더는 가장 먼저 동차 좌표계에서 정해진 투사 격자의 네 모서리 점을 선형보간하여 격자점의 위치를 다음과 같이 계산한다.

```
float4 pos = lerp( lerp(corner0, corner1, IN.coord0.x),
          lerp(corner3, corner2, IN.coord0.x), IN.coord0.y);
pos = pos / pos.w;
```

이들 격자점의 위치를 정점 셰이더에서 계산하지 않고 CPU에서 계산하여 정점 셰이더에 입력으로 전달할 수도 있지만 정점 버퍼를 매 프레임마다 갱신해야 하기 때문에 느리고 비효율적이다. 그리고 정점 셰이더는 픽셀 셰이더에 비해 렌더링시에 상대적으로 연산부하가 적게 걸리기 때문에 연산을 조금 추가하여도 전체적인 시스템 속도에 영향을 주지 않는다.

다음으로 Gerstner 모델과 스펙트럼 모델에 의해 생성된 높이 맵을 샘플링하여 수면을 나타내는 메쉬를 생성한다. 그리고 Gerstner 모델에 의해 생성된 법선 맵도 샘플링하여 픽셀 셰이더로 전달한다. 이렇게 함으로써 Gerstner의 법선 벡터는 래스터라이저 (rasterizer)에 의해 자동으로 선형 보간되기 때문에 앨리어싱 효과를 줄일 수 있다.

```
float3 GerstnerPos =
     tex2Dlod( SamplerGersterHeight, float4(IN.coord0,0,1));
float3 OUT.GerstnerNormal =
      tex2Dlod(SamplerGersterNormal, float4(IN.coord0,0,1));
float4 HD =
   tex2Dlod(SamplerFourierHD, float4(OUT.FourierCoord,0,1));
float3 FourierPos = pos + float3(lamda*HD.z,HD.x,lam*HD.w);
OUT.objpos = Gerstner_pos +Fourier_pos +{0,VerletHeight,0};
```

픽셀 셰이더에서는 정점 셰이더에 의해 계산된 메쉬와 텍스쳐 좌표들을 이용하여 텍스쳐들을 샘플링하고 법선 벡터를 이용하여 반사 벡터와 굴절 벡터를 계산하고 빛 벡터와의 내적 및 Fresnel 항의 계산을 통해 셰이딩을 하게 된다. 픽셀 셰이더에서는 가장 먼저 스펙트럼 모델과 항적파 모델의 법선 맵을 샘플링하여 Gerstner 모델의 법선과 합성한다.

```
float2 FourierNormal =
              tex2D(SamplerFourierNormal, IN.FourierCoord);
Normal.xz = IN.GerstnerNormal.xz + FourierNormal;
Normal.y = 1;
```

이처럼 법선 벡터의 합성도 변위의 합성과 같이 간단한 덧셈으로 계산이 가능하다. 그 이유는 법선벡터를 x, z 두 방향의 기울기로 표현할 수 있기 때문이다. 이렇게 얻은 법선 벡터를 이용하여 시선 벡터를 수면에 반사 혹은 굴절시켜 얻은 반사 벡터와 굴절 벡터를 계산한다. 픽셀 셰이더는 반사와 굴절 벡터를 계산하기 위한 내부 인스트럭션, reflect, refract를 제공한다. 굴절 벡터의 계산에서 시점은 항상 물위에 있는 것으로 가정하여 굴절율 1/1.33을 사용하

였다. 계산된 반사, 굴절 벡터를 이용하여 반사 맵과 굴절 맵을 샘플링할 텍스쳐 좌표의 위치를 적절히 변경하여 바다의 표면이 움직임에 따라 물 밑의 물체나 물 위의 물체가 일그러져 보이는 모습을 표현한다.

```
float3 vRefl = normalize( reflect(-IN.View, Normal) );
float3 vRefr =
          normalize( refract(-IN.View, Normal, 1/1.33) );
float2 refrcoord =IN.ReflCoord+vRefr.xz*IN.objpos.y *.1;
float2 reflcoord = IN.ReflCoord + vRefl.xz * 0.1;
```

수면에 반사되는 것은 전역적 (global) 반사와 지역적 (local) 반사의 두 가지 형태로 구분된다. 전역적 반사란 해, 구름, 하늘과 같이 수면에서 멀리 떨어져 있는 주위 환경이 수면에 반사되어 보이는 것을 의미하고 지역적 반사란 물 위에 떠 있는 배와 같이 수면에 가까이 있는 물체가 수면에 반사되어 보이는 것을 의미한다. 전역적 반사는 반사 벡터를 텍스쳐 좌표로 큐브 맵 텍스쳐를 샘플링하여 나타낸다.

```
float4 sky = texCUBE(SamplerSky, vRefl);
```

지역적 반사와 굴절은 추가적인 렌더링 시간이 필요하기는 하지만 렌더러블 텍스쳐를 이용하여 쉽게 표현이 가능하다. 지역적 반사는 수면을 제외한 나머지 물체들, 예를 들어, 배나 섬 등을 실제 화면이 아닌 렌더러블 텍스쳐에 렌더링한 후 화면에 실제로 바다를 렌더링할 때 이를 샘플링하여 물체가 수면에 반사된 것처럼 보이도록 한다. 반사 맵은 시점을 수면 아래로 옮기고 DirectX에서 제공하는 클리핑 평면 (clipping plane) 기능을 사용하여 수면 위쪽에 있는 물체만 그림으로써 생성되고 굴절 맵은 시점은 그대로 두고 역시 클리핑 평면을 사용하여 물 밑에 있는 물체만 렌더러블 텍스쳐에 그림으로써 생성된다. 이와 같이 반사 맵과 굴절 맵은 매 프레임마다 실제 화면에 렌더링을 하기 전에 생성되고 렌더링시에 샘플링된다.

```
float4 refl = tex2D(SamplerRefl, reflcoord);
float4 refr = tex2D(SamplerRefr, refrcoord);
```

지역적 반사와 전역적 반사를 선형 보간하여 반사에 의한 최종 색상을 결정한다.

```
refl = lerp(sky, refl, refl.a);
```

바다는 주위의 사물을 그 표면에 반사할 뿐만 아니라 어느 정도 투명하기 때문에 빛을 통과시켜 물 밑의 사물들을 볼 수도 있다. 물에 반사되거나 물속으로 투과되는 빛의 양은 수면과 시선 벡터가 이루는 각도에 따라 크게 달라지는데 Fresnel 항으로 정의되는 이러한 특성에 의해 반사에 의한 색상과 굴절에 의한 색상을 더하게 된다. Fresnel 항은 다음과 같이 계산된다.

```
float dotVN = dot(IN.View, Normal);
float fresnel = R0 + (1-R0)*pow(1-max(dotVN, -dotVN), 5);
```

추가로 햇빛이 수면에 반사되어 반짝거리는 효과를 나타내기 위해 아래와 같이 반사 벡터와 빛 벡터의 내적을 이용해 붉은 색을 추가하여 최종 바다의 색상을 결정한다.

```
float4 sun = pow( SunStrength * pow( saturate( dot(vRefl,
vLight)),SunShininess)*float4(1.2, 0.4, 0.1, 1), 1/2.2);
return lerp(refr, refl, saturate(fresnel)) + sun;
```

4. 실험 결과 및 분석

우리는 제안하는 바다물결 모델의 성능을 알아보기 위해 CPU와 GPU를 사용하는 시스템으로 각각 구현하고 시뮬레이션과 렌더링시에 초당 프레임수 (fps)를 비교하였다. 실험에 사용된 컴퓨터는 1GB의 시스템 메모리와 3GHz의 Pentium 4 CPU, 그리고 256MB의 비디오 메모리를 가진 NVIDIA GeForce 6800 Ultra 그래픽 카드를 사용하였다. 렌더링은 해상도가 1280 x 1024인 전체화면 (full-screen) 모드에서 이루어졌다.

GPU를 사용한 구현은 앞서 설명한 바와 같이 GPU에서 구현된 Fourier 변환을 사용하여 시뮬레이션 및 렌더링 전 과정이 GPU에서 처리된다. CPU를 사용한 구현에서 Fourier 변환은 FFTW 라이브러리[16]를 사용하였고 높이 맵과 법선 맵을 생성하는 모든 시뮬레이션 과정은 CPU에서 처리되고 투사 격자의 생성 및 샘플링 또한 CPU에서 처리된다. 단, 스펙트럼 모델에 의해 생성되는 법선 맵은 메쉬의 정점 간격에 관계없이 바다 표면을 자세하게 표현할 수 있도록 GPU 구현에서와 같이 픽셀 셰이더에서 샘플링된다.

<표 1>은 제안하는 합성모델에 의해 바다를 시뮬레이션하고 화면에 렌더링하는 경우의 초당 프레임 수를 CPU와 GPU로 구현된 시스템에서 각각 측정한 결과이다. 이때, Gerstner 모델의 해상도는 128x128 이고 각 격자점마다 중첩되는 트로코이드의 수는 16개 이다. 렌더링 과정에는 해상도가 256x256인 반사 맵과 굴절 맵을 실시간으로 생성하는 과정을 포함한다. 시점은 수면에서 1000m 정도 위에 있으며 렌더링된 바다는 수평선까지 펼쳐져 있고 화면 해상도는 1280x1024이다.

〈표 1〉 CPU와 GPU에서의 시뮬레이션+렌더링 속도 (fps)

스펙트럼 모델 해상도	GPU	CPU
64 x 64	48	14
128 x 128	37	14
256 x 256	28	12
512 x 512	14	2

전체적으로 CPU보다 GPU로 구현된 시스템의 성능이 우월하며 스펙트럼 모델과 Gerstner 모델의 해상도가 증가할수록 성능의 차이가 커진다. 이는 모든 시뮬레이션이 병렬 프로세서인 GPU에 의해 처리되기 때문에 CPU보다 빠른 처리가 가능하고, 시뮬레이션 결과로 생성된 높이 맵과 법선 맵이 이미 그래픽스 하드웨어상에 있기 때문에 렌더링시에 시스템 메모리로부터 이들 데이터를 전송할 필요 없이 바로 사용할 수 있기 때문이다. GPU에서 텍스쳐를 샘플링하는 속도는 CPU에서 캐쉬 메모리를 참조하는 속도에 비해 느리기 때문에 낮은 해상도에서는 GPU의 빠른 처리속도에 의한 성능향상 효과를 크게 볼 수 없지만, 보다 높은 해상도에서는 GPU의 병렬 처리에 의한 성능향상을 볼 수 있다.

<표 2>는 제안하는 합성 모델의 시뮬레이션 성능을 알아보기 위해, (그림 3)의 전체 시스템에서 항적파의 시뮬레이션을 제외하고 투사격자를 생성하기 전까지의 과정만 처리한 경우, 즉, 렌더링 없이 시뮬레이션만 했을 때의 초당 프레임수를 측정한 결과이다. 이때 GPU 구현에서 Gerstner 모델의 해상도는 256x256이고 트로코이드의 수는 16개 이다. CPU 구현에서 Gerstner 모델의 해상도는 64x64이고 트로코이드의 수는 16 개이다. <표 2>에서 보듯이 CPU 구현에서는 더 낮은 해상도의 Gerstner 모델을 사용하였음에도 불구하고 4~7배 빠른 속도를 보여준다.

스펙트럼 모델의 핵심이 되는 Fourier 변환의 성능을 알아보기 위해 FFTW 라이브러리의 복소수-복소수 Fourier 변환과 GPU로 구현된 복소수-복소수 DIT Fourier 변환을 비교하였다. <표 3>은 GPU로 구현된 Fourier 변환과 CPU의 FFTW 라이브러리의 Fourier 변환의 속도를 해상도를 변화시키면서 측정한 결과이다. <표 3>에서 보듯이 Fourier 변환의 속도는 낮은 해상도에서는 CPU가 빠르고, 512 x 512 이상의 높은 해상도에서는 GPU가 더 빠른 속도를 보인다. 이는 GPU가 병렬 프로세서로서 많은 데이터를 빨리 처리할 수 있기 때문이다. 특히 낮은 해상도에서 CPU가 더 빠른 처리속도를 보이는 것은 텍스쳐의 샘플링, 렌더링 타겟의 변경 등에 필요한 시간이 GPU의 계산 속도에 비해 느리기 때문에 GPU의 병렬 처리에 의한 속도향상 효과를 크게 볼 수 없기 때문이다.

Mitchell은 ATI Radeon X800을 사용하여 해상도가 64 x 64, 256 x 256 인 두 개의 Fourier 변환을 170 fps의 속도로

〈표 2〉 CPU와 GPU에서의 시뮬레이션 속도 (fps)

스펙트럼 모델 해상도	GPU	CPU
64 x 64	399	75
128 x 128	220	52
256 x 256	85	23
512 x 512	21	3
1024 x 1024	4	0.8

〈표 3〉 CPU와 GPU에서의 Fourier 변환 속도 (fps)

해상도	GPU	CPU
64 x 64	1925	3300
128 x 128	600	970
256 x 256	178	222
512 x 512	39	13
1024 x 1024	8	3

처리할 수 있었다고 하였는데[12], GPUbench[17]에 따르면 GeForce 6800 Ultra는 500 Mpix/sec, Radeon X800은 800 Mpix/sec의 처리속도를 보여준다.

그러므로 약 5:8의 하드웨어 성능의 차이를 감안하면 <표 3>의 결과는 Mitchell의 실험결과와도 일치되는 결과이다. 또한 GeForce 6800은 float4 형식의 데이터와 float2 형식의 데이터에 대한 처리속도가 같기 때문에 RGBA 텍스쳐를 사용하여 2 개의 Fourier 변환을 동시에 처리하는 속도와 RG 텍스쳐를 사용하여 하나의 Fourier 변환을 처리하는 속도가 같다. 따라서 본 연구의 시뮬레이션과 같이 두 개 이상의 Fourier 변환을 계산하여야 하는 경우에는 <표 3>에서의 GPU 실험 결과보다 2 배 정도의 속도를 기대할 수 있다.

Mitchell은 Radeon X800을 사용하여 스펙트럼 모델에 의해 해상도가 64 x 64, 256 x 256인 두 개의 높이 맵을 생성하고, 해상도가 192 x 192인 격자 메쉬를 사용하여 약 65 fps의 속도로 바다 영상을 렌더링할 수 있었다고 하였다[12]. <표 1>의 실험결과는 투사 격자를 사용하여 수평선까지 펼쳐진 넓은 영역의 바다 영상을 생성한 것이기 때문에 정해진 크기의 바다를 렌더링한 Mitchell의 실험결과와 바로 비교하기는 어렵다.

〈표 4〉 본 논문과 Mitchell의 연구의 실험조건 및 렌더링 속도 비교

		본 논문	Mitchell's
실험 조건	그래픽스 하드웨어	Geforce 6800 (500 Mpix/sec)	Radeon X800 (800 Mpix/sec)
	시뮬레이션 과정	3개의 256x256 FFT 1개의 192x192x16 덧셈	1개의 64x64 FFT 1개의 256x256 FFT
	렌더링 과정	2개의 높이 맵과 2개의 법선 맵을 샘플링	1개의 높이 맵과 1개의 법선 맵을 샘플링
		45 fps	65 fps

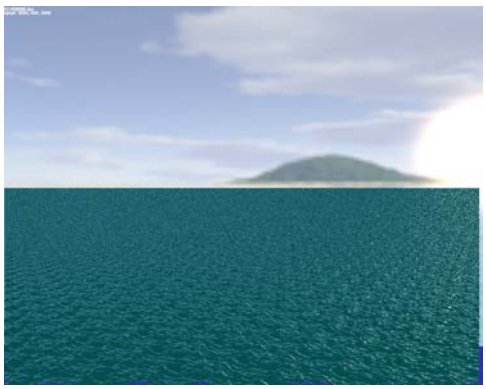
Mitchell의 실험결과와 본 논문의 실험결과의 정확한 비교를 위해서 Mitchell의 실험조건과 동일하게 화면 해상도를 768 x 768로 설정하고, 메쉬의 해상도, 즉, Gerstner 모델의 해상도를 192 x 192로 하여 화면에 바다가 가득 차도록 렌더링 했을 때 45 fps의 처리속도를 얻을 수 있었다. 이와 같은 처리속도의 차이는 표 4에 정리되어 있는 바와 같이 Mitchell은 해상도가 64 x 64 와 256 x 256인 Fourier 변환 각각 1개씩을 계산하였지만 본 논문에서는 해상도가 256 x 256인 Fourier 변환 3개를 동시에 계산하고, 이와 더불어 Gerstner 모델에 의한 높이 맵과 법선 맵도 생성하기 때문인 것으로 분석된다. 또한 이처럼 시뮬레이션 과정에서 보다 많은 수의 높이 맵을 생성할 뿐만 아니라 렌더링시에 보다 많은 수의 텍스쳐를 샘플링하여야 하기 때문에 렌더링 속도에 차이가 있다. 하지만 실험에 사용된 그래픽스 하드웨어의 성능 차이(5:8)를 고려하면 큰 속도차이가 아닌 것으로 분석된다.

(그림 14, 15)는 제안하는 합성 모델에 의해 렌더링된 바다 영상이다. (그림 14)와 같이 잔잔한 바다뿐만 아니라 (그림 15)처럼 높은 파도가 치는 거친 바다의 모습도 쉽게 표현할 수 있다.

(그림 16)은 스펙트럼 모델만을 사용하여 바다를 렌더링한 모습이다. (그림 16)에서는 반복패턴이 뚜렷하게 보이며 본 논문에서 제안하는 합성모델에 의해 렌더링된 (그림 14, 15)에서는 반복패턴이 나타나지 않는 것을 볼 수 있다.

(그림 14) 제안하는 방법으로 렌더링된 잔잔한 바다의 모습

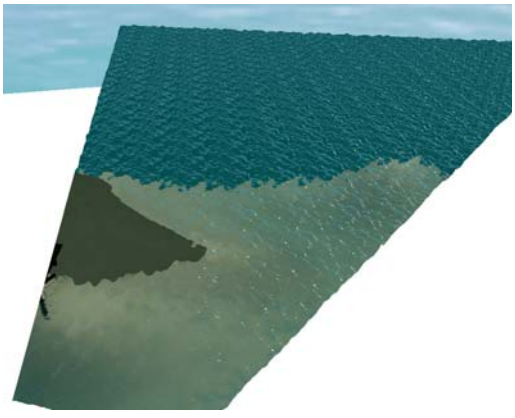
(그림 15) 제안하는 방법으로 렌더링된 거친 바다의 모습



(그림 16) 스펙트럼 모델에 의한 반복 패턴이 뚜렷하게 보이는 바다

(그림 17)은 본 실험에서 사용한 LOD가 적용된 투사격자의 모델이다. 그림에서 볼 수 있듯이 카메라 위치에 가까운 곳의 격자는 간격이 좁고 세밀하며 먼 곳의 격자는 간격이 넓게 표현되어 자동적으로 LOD가 적용되어 카메라의 위치에 관계없이 자연스러운 바다의 모습을 표현할 수 있다. (그림 18)은 이와 같은 투사격자에 의해 렌더링된 실제 바다의 모습이다.

(그림 17) LOD가 적용된 투사격자 모델



(그림 18) 투사격자로 모델링된 바다의 모습

5. 결 론

본 논문에서는 높은 수준의 바다 영상을 실시간으로 생성하기 위해 스펙트럼 모델과 Gerstner 모델을 합성한 모델을 제안하고 이를 GPU의 연산능력을 활용하는 시스템으로 구현하였다. 제안하는 방법은 Gerstner 모델을 이용하여 바다의 전체적인 형태와 큰 파도의 모습을 나타내는 저해상도의 높이장을 생성하고 스펙트럼 모델에 의해 작은 파도와 거친 수면을 자세하게 나타내기 위한 고해상도의 높이장을 생성하여 두 높이장을 합성함으로써 바다 영상을 생성한다. 이로써 잔잔한 바다뿐만 아니라 높은 파도가 치는 거친 바다도 쉽게 표현할 수 있다.

바다를 표현하는 메쉬는 화면상의 2차원 균일 격자를 안정상태의 수면으로 역 투사하여 얻은 투사 격자로 생성되기 때문에 좁은 영역의 바다뿐만 아니라 수평선까지 펼쳐지는 무한히 넓은 영역의 바다까지 렌더링 할 수 있고 모든 계산은 화면에 보이는 바다 영역에만 집중된다. 바다의 전체적인 형태를 표현하는 Gerstner 모델은 공간상에서 연속함수로 정의되기 때문에 시점에 따라 격자 간격이 변하는 투사 격자를 사용하면서도 앨리어싱 문제없이 연속적인 LOD를 제공할 수 있다.

바다 영상을 생성하기 위한 시뮬레이션 과정이 벡터 연산과 병렬 처리가 가능한 GPU에 의해 실행되기 때문에 CPU보다 빠른 속도로 처리가 가능하고, 시뮬레이션 결과인 높이 맵과 법선 맵이 그래픽스 하드웨어상에 텍스쳐의 형태로 저장되기 때문에 렌더링과정에서 이를 바로 사용할 수 있다. 따라서 CPU로 시뮬레이션 할 경우에 필요한 시스템 메모리로부터 그래픽스 하드웨어로의 데이터 전송이 필요없고 이에 따른 병목 현상이나 동기화 문제를 사전에 예방할 수 있어 보다 빠른 렌더링이 가능하다. 또한 구현된 시스템은 시점에 따라 계산량을 자동으로 조절하는 적응기법을 사용함으로써 시점에 관계없이 높은 수준의 영상과 빠른 처리속도를 유지할 수 있다.

향후 연구과제로서 앞으로 공개될 DirectX 10은 기하 셰이더 (geometry shader)가 추가되고 텍스쳐와 정점 버퍼의 구분이 없어지는 등 GPU 프로그래밍에 한층 더 높은 자유도를 부여할 것으로 기대되어 이를 활용하기 위한 연구가 필요하다. 특히 기하 셰이더는 실시간으로 새로운 정점을 생성할 수 있기 때문에 본 논문의 바다와 같이 2차원 격자 위에서 시뮬레이션되는 시스템에서 LOD를 제공하는 수단이나 투사 격자를 생성하는데 사용할 수도 있을 것이다. 또한 시뮬레이션 과정에서 높이 맵을 생성하고 이를 렌더링시에 다시 샘플링하여 메쉬를 변형하는 대신에 정점 버퍼의 데이터를 바로 변형할 수 있기 때문에 더욱 빠른 처리가 가능할 것으로 기대된다.

참 고 문 헌

[1] Jos Stam, “Stable Fluids,” Proceedings of the 26th Anual Conference on Computer Graphics and Interactive Techniques, pp 121-128, 1999.

[2] Nick Foster and Ronald Fedkiw, “Practical Animation of Lquids,” Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques, pp 23-30, 2001.

[3] Mark J. Harris, William V. Baxter III, Thorsten Scheuermann, Anselmo Lastra., “Simulation of Cloud Dynamics on Graphics Hardware,” Proceedings of Graphics Hardware 2003.

[4] Alain Fournier and William T. Reeves, “A Simple Model of Ocean Waves,” Computer Graphics, Vol. 20, No. 4, 1986, pp 75-84.

[5] Sebastien Thon, Jean-Michel Dischler and Djamchid Ghazanfarpour “Ocean Waves Synthesis Using a Spectrum-Based Turbulence Function,” Proceedings of the International Conference on Computer Graphics, 2000.

[6] Ken Perlin, “An Image Synthesizer,” SIGGRAPH, 1985.

[7] Damien Hinsinger, Fabrice Neyret and Marie- PauleCani, “Interactive Animation of Ocean Waves,” Proceedings of the 2002 ACM SIGGRAPH/ Eurographics Symposium on Computer Animation, 2002.

[8] Gary A. Mastin, Peter A. Watterger, and John F. Mareda, “Fourier Synthesis of Ocean Scenes,” IEEE Computer Graphics and Applications, pp 16-23, March 1987.

[9] Jerry Tessendorf, “Simulating Ocean Water,” In SIGGRAPH Course Notes, Addison-Wesley, 1999.

[10] Lasse Staff Jensen and Robert Golias, “Deep-Water Animation and Rendering,” Game Developers Conference Europe, 2001.

[11] Stefano Lanza, “Animation and Display of Water,” in Shader X3: Advanced Rendering with DirectX and OpenGL, Charles River Media, 2004.

[12] Jason L. Mitchell, “Real-Time Synthesis and Rendering of Ocean Water,” ATI Research Technical Report, 2005.

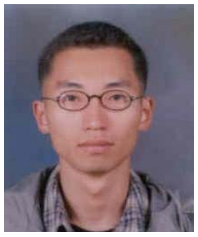
[13] Sumanaweera T. et al. “Medical Image Reconstruction with the FFT,” GPU GEMS 2, Addison-Wesley, 2005.

[14] Claes Johanson, “Real-time Water Rendering,” Master of Science Thesis in Computer Graphics, March 2004.

[15] Jeremy Zelsnack, “Vertex Texture Fetch Water,” NVIDIA SDK, 2004.

[16] “FFTW library,” http://www.fftw.org/

[17] “GPUbench,” http://graphics.stanford.edu/projects/]gpubench/



이 동 민

e-mail : closed96@yahoo.co.kr
2004년 경북대학교 컴퓨터과학과 (학사)
2006년 경북대학교 대학원 컴퓨터과학과 (석사)
2006년~현재 (주)KOG 연구원
관심분야 : 컴퓨터 게임

이 성 기

e-mail : sklee@knu.ac.kr
1979년 서울대학교 전기공학과 (학사)
1981년 서울대학교 대학원 전기공학과 (석사)
1985년 유타대학교 대학원 컴퓨터과학과 (박사)
1982년~1984년 울산대학교 전자계산학과 전임강사
1990년~현재 경북대학교 전자전기컴퓨터학부 교수
관심분야 : 컴퓨터게임, 영상처리, 의료용소프트웨어