Journal of Korea Multimedia Society Vol. 9, No. 11, November 2006(pp. 1515-1528)

키-프레임 기반 실시간 유체 시뮬레이션

유지현†, 박상훈††

요 약

물리기반 유체 애니메이션 시스템들이 시각 특수효과 산업계에서 빠르게 발전하고 있고, 이를 이용해 매우 높은 화질의 영상을 제작하는 것이 가능하게 되었다. 그러나 컴퓨터 게임과 같은 실시간 응용 분야의 경우, 화질보다는 시뮬레이션 속도가 더 중요한 문제이다. 본 논문은 프로그래머블 그래픽스 파이프라인을 이용하여 유체에 대한 애니메이션을 수행하는 실시간 기법에 대해 설명한다. 두 개의 키-프레임이 주어졌을 때, 본 기법을 이용하여 원시 프레임으로부터 목적 프레임으로 변하는 연속 영상을 대화식으로 생성할 수 있음을 보인다.

Key-Frame Based Real-Time Fluid Simulations

Jihyun Ryu†, Sanghun Park††

ABSTRACT

Systems for physically based fluid animation have developed rapidly in the visual special effects industry and can make very high quality images. However, in the real-time application fields such as computer game, the simulation speed is more critical issue than image quality. This paper presents a real-time method for animating fluid using programmable graphics pipeline. We show that once two key-frames are given, the technique can interactively generate a sequence of images changing from the source key-frame to the target.

Key words: Fluid Simulation(유체 시뮬레이션), Programmable Shader(프로그래머블 쉐이더), Real-Time Rendering(실시간 렌더링)

1. 서 론

연기, 화염, 물 등의 불규칙한 유체의 운동을 자연스럽게 표현함으로써 자연현상을 사실적으로 시뮬레이션 하는 것을 목표로 하는 물리기반 모델링(physical-based modeling) 관련 연구가 최근 몇 년간 컴퓨터 그래픽스와 컴퓨터 애니메이션에서 매우 중요한 분야로 활발하게 연구되고 있다. 현재까지 수행된 대부분의 연구 결과들은 유체의 움직임을 잘 표현해 주는 Navier-Stokes 방정식에 각 유체마다의 특성을 고려한 함수의 적절한 결합과 변형을 통해 새로운 미분방정식을 모델링하고 효율적인 풀이 기법을 개발하는 것을 기본으로 한다. 그러나 방정식의 비선형성으로 인해 해를 구하는데 시간적 경제적 비용의 문제가 있기 때문에, 대부분의 연구는 수치적 정확성의 추구보다는 실제적이고 자연스러운 시뮬레이션의 제작을 목표로 진행되어 왔다.

대표적인 유체 시뮬레이션 관련 연구들은 다음과

※ 교신저자(Corresponding Author) : 박상훈, 주소 : 서울시 중구 필동3가 26번지(100-715), 전화 : 02)2260-3765, FAX : 02)2260-3764, E-mail : mshpark@dongguk.edu
접수일 : 2006년 8월 21일, 완료일 : 2006년 9월 20일
† 정회원, 세종대학교 응용수학과 초빙교수
(E-mail : ajhryu@sogang.ac.kr)
†† 종신회원, 동국대학교 영상대학원 멀티미디어학과 조교수
※본 논문은 2003년도 한국학술진흥재단의 지원에 의하여 연구되었으며 (KRF-2003-037-D00012), 연구의 일부는 2005년도 서울시 신기술 연구개발 사업의 지원을 받았음 (10672).

같다. Foster 등은 유한 차분법(finite differencing)과 Successive Over Relaxation (SOR) 방법을 사용하여 미분방정식을 풀이 하였고, 유체의 속성을 제어하는 매개변수들을 조절하여 다양한 성질의 유체 움직임을 생성하였다[1]. 하지만, 그들은 인위적인 형상 제어가 아닌 유체의 역학적 성격을 고려한 움직임만을 표현하였다. 이후 물리학에 기초한 유체 시뮬레이션은 1999년 Stam에 의해 안정적인(stable) 유체 알고리즘이 소개되면서 빠른 발전을 보여 왔다[2]. 그는 세미-라그랑주 방법(semi-lagrangian method)에 의한 효과적인 풀이 과정을 소개하였다. 특히, 시간 간격(time-step)이 큰 경우에도 안정적인 해를 얻을 수 있음을 보임으로써, 유체 시뮬레이션에서 수치적인 안정성 증대 및 사용의 용이성 향상 등의 획기적인 기여를 하였다. 이후 Foster 등은 SOR 대신에 더욱 효과적인 선형 풀이인 켤레구배법(conjugate gradient method)을 사용하고, 처음으로 음대수곡면 레벨셋 방법(implicit level-set surface method)을 사용하여 경계면을 표현하였다[3]. 레벨셋을 이용함으로써 액체 경계면을 매우 부드럽게 표현할 수 있고 위상의 변화도 쉽게 나타낼 수 있었다. 레벨셋 방법이 발전하면서, Enright 등은 물의 경계면을 표현하기에 용이한 파티클 레벨셋 방법(particle level-set method)을 제안하였다[4].

2003년에 Treuille 등은 유체의 궤적, 최종 모양 등을 사용자가 임의로 제어할 수 있는 시뮬레이션 방법을 소개하였고[5], 이 연구 결과는 사용자가 제어 가능한 유체 시뮬레이션 연구의 계기가 되었다. 그들은 사용자에 의해 지정된 밀도와 속도를 갖는 키-프레임(key-frame)을 정의하고, 이 키-프레임으로부터 시뮬레이션에 영향을 주는 힘을 제어변수라는 매개변수를 이용하여 정의하였다. 그리고 정의된 힘을 최소화하면서 시뮬레이션의 결과가 목표하는 키-프레임에 수렴할 수 있도록 목적함수(objective function)를 만들고 이에 대한 최적화 문제를 해결하였다. 목적함수를 정의하고 최적화 하는 문제가 쉽지 않을 뿐만 아니라, 매우 방대한 양의 계산을 필요로 하기 때문에 실제 프로그램 구현에서는 사용하기에 적합하지 않다는 단점을 가지고 있었지만, 이것은 최초로 발표된 제어 가능한 유체 시뮬레이션 연구 결과였다. 이를 시작으로 2004년 Fattal 등이 일반 시뮬레이션과 비슷한 비용, 속도로 수행 가능한 사용자 정의의 연기 시뮬레이션 연구 결과를 발표하였다[6].

이러한 물리기반 시뮬레이션은 현재 영화, 애니메이션과 같은 디지털 콘텐츠 분야에서 다양하게 이용되어 그 효용성이 실제로 입증되고 있다. 그러나 앞에서 언급한 바와 같이, 각 프레임을 계산하는데 필요한 계산 비용이 너무 크고 시뮬레이션 결과를 실시간 또는 대화식으로 예측할 수 없으며, 따라서 원하는 최종 시뮬레이션 결과 영상들을 얻기까지 반복되는 모델링에 대한 검증과 제어 파라미터들의 설정에 많은 시행착오와 시간을 필요로 한다.

본 논문에서는 [6]의 연구 결과를 발전시켜 사용자가 의도하는 연기 시뮬레이션을 실시간으로 구현하는 것을 목표로 한다. 앞에서 설명한 관련 연구들이 모두 오프라인 렌더링(offline rendering) 기반의 접근 방법을 선택하고 있지만, 이와 별개로 효과적인 실시간 시뮬레이션 기법을 개발하기 위한 연구가 고성능 GPU(Graphics Processing Unit)를 기반으로 하는 실시간 렌더링(real-time rendering) 분야에서 최근 활발하게 수행되고 있다[7-9]. 이 기법들은 프로그래머블 그래픽스 파이프라인(programmable graphics pipeline)의 버텍스 쉐이더(vertex shader)와 프래그먼트 쉐이더(fragment shader)를 이용하여 복잡한 시뮬레이션 계산을 CPU가 아닌 GPU상에서 수행함으로써 연산 속도를 향상시키는 것을 목표로 한다. 실시간 시뮬레이션 결과 영상이 앞서 설명한 오프라인 렌더링 기법들에 의해 생성된 영상들에 비해 좋지 않은 것이 사실이지만, 최신 실시간 렌더링 핵심 기술을 기반으로 하는 컴퓨터 게임과 같은 실시간 응용 분야에서 본 논문의 연구 결과는 매우 효과적으로 활용될 수 있다.

2. 유체 시뮬레이션

2.1 문제 정의

앞 절에서 언급한 바와 같이, 연기, 물, 불 등의 자연현상을 모델링하는 대부분의 연구 결과들은 유체의 움직임에 대한 수학적인 표현으로 다음과 같이 정의된 Navier-Stokes 방정식을 이용해 왔다.

$$u_t = -u \cdot \nabla u - \frac{1}{\rho}\nabla p + \nu \nabla \cdot \nabla u + f$$
$$\nabla \cdot u = 0$$

여기서, $u(x,t)$는 위치벡터 x의 시간 t에서의 유

체의 속도장(velocity field)이고, 이를 시간에 대해 미분한 것이 $u_t\ (\equiv \frac{\partial u}{\partial t})$ 이다. 또한, $p=p(x,t)$ 는 유체의 압력, f 는 유체의 움직임에 영향을 주는 외부로부터의 힘(중력, 부력 등)이다. 그리고 ν는 유체의 점도로서, 직관적으로 끈적이는 정도를 의미한다. 첫 번째 방정식을 운동 방정식(momentum equation)이라 하는데, 이는 유체를 구성하는 입자의 힘은 그 질량과 가속도의 곱에 비례한다는 뉴턴의 운동방정식으로부터 유도된다. 두 번째 방정식은 비압축성 유체의 조건인 질량보존 방정식(conservative equation, continuity equation)이다.

위의 Navier-Stokes 방정식은 표현하려는 대상이 기체인 경우 (전혀 끈적임이 없는 경우, 즉 점도가 0인 경우) 다음과 같은 방정식으로 변형 가능하다.

$$u_t = -u \cdot \nabla u - \frac{1}{\rho}\nabla p + f$$
$$\nabla \cdot u = 0$$

이 비점성 유체방정식을 오일러 방정식(Euler equation)이라 부른다. 이 때, 밀도 ρ가 각 점에서 상수라는 것을 고려하면 위 편미분 방정식의 변수는 속도 벡터 u와 압력 p임을 알 수 있다.

본 논문에서는 외부입력 f를 유체 제어를 위한 유도 힘(driving force)[6] F로 정의하고, 방정식의 해를 구하는 각 연산 단계의 속도를 향상시키기 위해 GPU의 프로그래머블 쉐이더를 이용하였으며, 이를 통해 사용자가 유체 시뮬레이션을 실시간에 제어할 수 있도록 구현하였다.

2.2 수학적 기본개념

본 절에서는 모델링된 미분방정식의 풀이를 위해 필요한 몇 가지 수학적 개념을 정리한다. 다음의 미분연산자(differential operator)들은 벡터 혹은 스칼라에 정의되는 벡터해석(vector calculus)의 기본연산들이다.

2.2.1 그레디언트 (Gradient: ∇)

그레디언트 연산은 이변수 이상의 스칼라함수에 정의되며 연산의 결과는 벡터이다. 예를 들어 3차원의 경우 $\nabla f(x,y,z) = (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z})$ 이고, 이를 $\nabla = (\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z})$ 으로 나타내기도 한다. 그레디어트 연산은 종종 벡터를 정의역의 원소로 갖는 함수의 방향도함수(directional derivatives)를 나타내는데 쓰인다. 즉, $\vec{x}=(x,y,z)$ 라 할 때, 충분히 작은 값 $\triangle\vec{x}$에 대하여 $f(\vec{x}+\triangle\vec{x}) \approx f(\vec{x}) + \nabla f(\vec{x})\triangle\vec{x}$ 이 성립한다. 따라서 그레디언트 연산의 유한차분(finite difference)형식은 다음과 같다.

$$\nabla f = (\frac{f_{i+1,j,k} - f_{i-1,j,k}}{2\triangle x}, \frac{f_{i,j+1,k} - f_{i,j-1,k}}{2\triangle y}, \frac{f_{i,j,k+1} - f_{i,j,k-1}}{2\triangle z})$$

2.2.2 발산 (Divergence: $\nabla \cdot$)

발산 연산자는 벡터에만 정의되는 것으로 연산 결과는 스칼라이다. 이는 어떤 지점의 모든 방향에 대한 물리량의 합을 의미하는 것으로, 발산 연산자를 속도벡터에 적용하면 특정 지점을 통과하는 모든 속도벡터의 합을 뜻한다. 본 논문에서 다루려는 비압축성(incompressible) 유체의 경우 속도벡터에 대한 발산 연산의 결과가 0이 된다는 질량보존 방정식을 전제로 한다. 연산의 정의는 3차원 속도벡터 $\vec{u}=(u,v,w)$ 의 경우, $\nabla \cdot \vec{u} = \nabla \cdot (u,v,w) = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}$ 이고, 이는 그레디언트 연산자와 벡터함수의 내적이므로 다음과 같이 표현하기도 한다.

$$\nabla \cdot \vec{u} = (\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}) \cdot (u,v,w) = \frac{\partial}{\partial x}u + \frac{\partial}{\partial y}v + \frac{\partial}{\partial z}w$$

이 경우, 유한차분형식은 다음과 같다.

$$\nabla \cdot \vec{u} = \frac{u_{i+1,j,k} - u_{i-1,j,k}}{2\triangle x} + \frac{v_{i,j+1,k} - v_{i,j-1,k}}{2\triangle y} + \frac{w_{i,j,k+1} - w_{i,j,k-1}}{2\triangle z}$$

2.2.3 회전 (Curl: $\nabla \times$)

회전 연산자는 특정 지점에서 얼마나 많은 회전이 일어나느냐에 대한 척도이다. 이는 일반적으로 벡터함수에 정의되는 연산으로 그 결과는 다음과 같이 표현되는 벡터이다.

$$\nabla \times \vec{u} = \nabla \times (u,v,w) = (\frac{\partial w}{\partial y} - \frac{\partial v}{\partial z}, \frac{\partial u}{\partial z} - \frac{\partial w}{\partial x}, \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y})$$

다만, 2차원 함수인 경우 벡터함수와 스칼라함수인 경우에 해당되는 두 가지 방법으로 회전 연산을 정의한다. 먼저, 벡터함수 $\vec{u}=(u,v)$ 에 회전 연산을 정의할 경우 외적 $\times$을 계산할 수 없으므로 $\vec{u}=(u,v,0)$ 으로 확장하여 정의한다. 즉, $\nabla \times \vec{u} = \nabla \times (u,v,0) = \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}$ 이 되어 연산의 결과가 스칼라임을 알 수 있다. 다음으로 2차원 스칼라함수 $w=w(x,y)$ 에 대한 회전 연산

은 $w=(0,0,w)$ 에 대하여 $\nabla \times w=\nabla \times (0,0,w)=(\frac{\partial w}{\partial y},-\frac{\partial w}{\partial x})$로 정의한다.

2.2.4 라플라시안 (Laplacian: ∇^2 ($\nabla \cdot \nabla$, Δ))

일반적으로 그레디언트 연산의 결과에 발산 연산을 적용한 결과를 말한다. 즉, 3차원의 경우 스칼라함수 $f(x,y,z)$ 에 대하여, 다음의 결과를 얻는다.

$$\nabla \cdot \nabla f(x,y,z)=\nabla \cdot (\frac{\partial f}{\partial x},\frac{\partial f}{\partial y},\frac{\partial f}{\partial z})=\frac{\partial^2 f}{\partial x^2}+\frac{\partial^2 f}{\partial y^2}+\frac{\partial^2 f}{\partial z^2}$$

이 경우, 유한차분형식은 다음과 같다.

$$\nabla^2 f=\frac{f_{i+1,j,k}-2f_{i,j,k}+f_{i-1,j,k}}{2\Delta x}+\frac{f_{i,j+1,k}-2f_{i,j,k}+f_{i,j-1,k}}{2\Delta y}+\frac{f_{i,j,k+1}-2f_{i,j,k}+f_{i,j,k-1}}{2\Delta z}$$

특히, $\Delta x=\Delta y=\Delta z$ 인 경우는 위 수식은 아래와 같이 간단히 정리된다.

$$\nabla^2 f=\frac{f_{i+1,j,k}+f_{i-1,j,k}+f_{i,j+1,k}+f_{i,j-1,k}+f_{i,j,k+1}+f_{i,j,k-1}-6f_{i,j,k}}{2\Delta x}$$

임의의 상수 a, b에 대하여, 라플라시안 연산이 포함된 방정식 $a\nabla^2 f(x,y,z)=b$ 를 포아송 방정식(Poisson equation)이라 부른다. 또한 $b=0$ 인 경우, 즉 $a\nabla^2 f(x,y,z)=0$ 인 방정식을 라플라스 방정식(Laplace equation)이라 한다. 이 방정식은 비압축성 유체의 가장 중요한 요소인 압력을 계산하는데 쓰이며, 해를 구하기 위한 수치 해석적 방법으로 자코비 반복법(Jacobi iteration)이 일반적으로 이용되며, 속도 향상을 위하여 켤레구배법(conjugate gradient method)을 사용하기도 한다.

2.3 제어 유체 방정식 설계 및 풀이

일반적으로 기체를 표현하는 경우 앞에서 소개한 비점성, 비압축성 유체방정식인 오일러 방정식(Euler equation)이 사용된다.

$$u_t=-u \cdot \nabla u-\frac{1}{\rho}\nabla p+F$$
$$\nabla \cdot u=0$$

먼저 적절한 변형을 통하여 각 단계별 방정식의 풀이 과정을 소개한다.

2.3.1 Helmholtz-Hodge 분해(decomposition)과 유체방정식의 변형

Helmholtz-Hodge 분해란, 임의의 속도장(velocity field) w가 질량보존장(mass conserving field) u와 압력장(pressure field) p의 그레디언트 벡터의 합으로 분해된다는 정리이다. 즉,

$$u=w-\nabla p$$

여기서 벡터 u가 질량보존의 성질을 갖고 있어 식 $\nabla \cdot u=0$ 을 만족시킨다는 것을 이용하면 다음의 식을 얻는다.

$$\nabla \cdot w=\nabla \cdot u+\nabla^2 p=\nabla^2 p, \quad \text{즉,} \quad \nabla^2 p=\nabla \cdot w$$

이 방정식으로부터 임의의 벡터 w와 압력 p에 대한 포아송 방정식을 풀고, 계산된 w를 이용하여 무발산(divergence free) 벡터 u를 얻을 수 있다.

한편, 벡터 u는 벡터 w의 사영(projection) 벡터이므로 $Pw=u$ 이고, 무발산 벡터 u는 $Pu=u$ 를 만족하므로, $P(u)=P(w-\nabla p)$ 로부터 $P(\nabla p)=0$ 를 얻는다.[1] 즉,

$$u_t=P(-u \cdot \nabla u+F)$$
$$\nabla \cdot u=0$$

2.3.2 유도 힘(driving force) 적용 단계

직관적으로 어떤 유체를 특정한 방향으로 움직이도록 하는 힘은 초기에 주어진 밀도장에서 각 시간간격마다 계산되어 목표 지점의 밀도장으로 향하도록 제어하는 힘이다. 기체의 진행 방향을 제어하기 위하여 각 단계에서의 밀도는 목표 지점의 밀도의 그레디언트 방향을 향해야 하므로 비례식 $F(\rho,\rho^*)\propto\nabla\rho^*$ 을 만족해야 한다. 한편, 목표형상에 도달했을 때의 유도 힘 F는 그 지점에서의 그레디언트 벡터로 주어질 것이므로 다음의 식을 얻는다.

$$F(\rho,\rho^*)=\rho\frac{\nabla\tilde{\rho}^*}{\tilde{\rho}^*}$$

이제 속도벡터(일반적으로 무발산 벡터가 아닐 수도 있는) u에 F를 더한다.

$$u=u+F(\rho,\rho^*)$$

2.3.3 속도에 대한 확산(advection) 단계

유체의 속도벡터는 속도 자신과 밀도, 온도 등의 물리량을 이동시키는데, 이를 확산(advection, con-

1) 여기서 P는 사영 연산자(projection operator)를 의미한다.

vection)이라 부른다. 유체 시뮬레이션에서 확산의 개념은 현재 격자(grid)에 위치한 파티클을 역 추적하여 이전에 있던 격자로부터 속도, 밀도, 온도 등의 정보를 얻어 현재 격자의 속도(혹은 밀도, 온도 등)를 새로 계산할 수 있다는 것이다. 속도 u에 대한 확산을 표현하는 수식은 다음과 같다.

$$u(\vec{x}, t+\Delta t) = u(\vec{x} - u(\vec{x}, t)\Delta t.\ t)$$

실제로 위의 식을 시간 t에 관하여 양변 편미분하면 다음과 같은 확산방정식을 얻는다.

$$u_t = -u \cdot \nabla u$$

2.3.4 사영(projection) 단계

외부 힘, 이동, 확산 등에 의해 계산된 값의 발산을 막기 위한 보정 단계로 오일러 방정식의 질량보존식을 Helmholtz-Hodge 분해를 이용해 속도에 관한 다음과 같은 하나의 식으로 표현할 수 있다.

$$\nabla^2 p = \nabla \cdot u$$

일반적으로 이러한 형태의 방정식을 포아송 방정식이라 하며, 본 논문에서는 풀이가 쉽고 간편한 자코비 반복법으로 방정식의 해를 구하였다. 우선 확산 단계에서 얻은 속도를 이용하여 위 방정식을 풀어 압력을 구한다. 그리고 각 격자의 속도와 압력을 갱신하고 이웃한 격자 사이의 발산을 막기 위한 보정 과정을 되풀이 한다.

2.3.5 밀도에 대한 확산(advection) 단계

일반적으로 유체의 상태는 속도벡터 u와 밀도 ρ에 의해 결정된다. 밀도의 경우에도 속도에 관한 확산 과정에서와 같은 방법으로 아래 같은 밀도에 관한 확산 수식을 얻는다.

$$\rho(\vec{x}, t+\Delta t) = \rho(\vec{x} - u(\vec{x}, t)\Delta t,\ t)$$

실제로, 위의 식을 시간 t에 관하여 양변 편미분하면 다음의 확산방정식을 얻게 된다.

$$\rho_t = -u \cdot \nabla \rho$$

이 방정식은 밀도가 속도장을 따라 움직인다는 것을 보여준다. (예를 들어, 연기에 바람을 불어주면 바람의 방향을 따라 연기가 자연스럽게 움직이는 것을 볼 수 있다.) 전체 밀도장을 동일한 간격의 격자로 나누고 각 격자의 중앙에 주어진 밀도벡터와 속도벡터를 이용하여 각 시간간격에서 위의 선형 미분방정식을 풀어 밀도벡터를 갱신한다.

3. 실시간 시뮬레이션 구현

3.1 프로그래머블 쉐이더

최근 몇 년간 GPU의 성능과 기능에 많은 발전이 이뤄졌으며, 더욱이 최근 상용화된 최신 그래픽스 프로세서들은 32비트 부동소수점 연산 기능뿐만 아니라 고급 프로그래밍 언어를 위한 컴파일러까지 지원하고 있는 수준이다. 전통적인 컴퓨터 그래픽스에서 GPU의 역할은 안티-에일리어싱(anti-aliasing), 텍스춰 매핑(texture mapping), 다각형 데이터의 실시간 렌더링 등에 국한되어 있었으나, 지속적인 기술발전에 힘입어 GPU들의 성능과 기능들은 CPU가 담당해야만 했던 일반적인 계산들을 CPU보다 더 빠르고 효율적으로 수행할 수 있는 수준에 이르게 되었다. 최근에 이르러 소프트웨어 개발자들은 GPU를 속도가 빠른 하나의 보조 프로세서로서 고려하고 이를 이용하여 (그래픽스와 관련되지 않는) 복잡한 문제들의 효과적으로 해결에 응용하고 있는데, 이를 GPGPU(General Purpose GPU)라 부른다. 이러한 응용들의 대부분은 GPU가 내재적으로 가지고 있는 병렬성과 벡터 처리 능력을 이용하는 것이다[10]. 기존의 고정된 그래픽스 파이프라인 구조와는 달리, 최신 GPU가 제공하는 프로그래머블 그래픽스 파이프라인은 기본적으로 프로그래머블 버텍스 프로세서과 프로그래머블 프래그먼트 프로세서를 포함하고 있으며, SIMD(Single Instruction, Multiple Data) 방식의 병렬 수행이 가능하도록 설계되었다.

프로그래머블 쉐이더 개발에 이용되는 대표적인 프로그래밍 언어는 OpenGL GLSL[11], NVIDIA의 Cg[12], Microsoft의 HLSL[13], University of Waterloo의 Sh[14]이며, 언어 마다 문법적인 차이가 있지만 구현 가능한 버텍스 쉐이더와 프래그먼트 쉐이더의 기능과 형태는 매우 유사하다. 본 구현에서는 Cg를 이용하였으며 시뮬레이션 프로그램의 특성상 버텍스 쉐이더 사용없이 프래그먼트 쉐이더만으로 시스템을 구현하였다. 실제로 물리기반 시뮬레이션에서 필요로 하는 (행렬 연산, 격자기반 연산과 같은) 다양한 일반적인 수학 연산들이 쉐이더를 사용하여

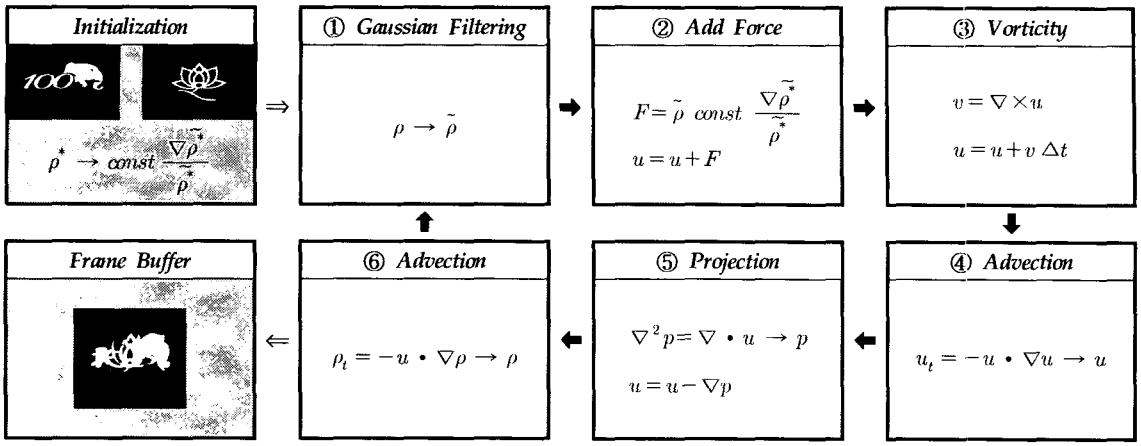


그림 1. 구현된 실시간 시뮬레이션 알고리즘의 단계별 연산 (프레임 버퍼의 영상은 초기화 과정에서 주어진 원시 키-프레임에서 목적 키-프레임으로 변해가는 중간 영상을 보여준다. 그림 9 참고)

GPU에서 빠르게 수행 가능하며, 이와 관련된 GPGPU 연구가 최근 활발하게 수행되고 있다[15-18]. 특히, 실시간 유체 시뮬레이션을 위한 Cg 코드 구현은 [8]에 자세히 설명되어 있다.

3.2 실시간 알고리즘

그림 1은 본 논문에서 구현된 프래그먼트 쉐이더로 설계된 실시간 시뮬레이션 알고리즘의 단계별 연산 과정을 보여준다.[2] 원시 키-프레임(source key-frame)과 목적 키-프레임(target key-frame) 두 개를 입력으로 받아 원시 키-프레임부터 목적 키-프레임으로 변화해가는 중간 애니메이션 프레임을 생성하는 기본 알고리즘은 [6]에서 소개된 CPU 기반의 오프라인 시뮬레이션 기법과 유사한 계산 과정으로 구성된다. 각 단계의 연산들의 수행속도 향상을 위해 적절한 프래그먼트 쉐이더를 설계하였고, 쉐이더 구현 과정에서 해결해야 할 다양한 문제들을 효과적으로 해결하여 오프라인 시뮬레이션에서는 불가능했던 실시간 제어 기능을 추가할 수 있었다.

우선, 그림 1의 ①에서 ⑥까지의 연산 과정은 시뮬레이션 종료 전까지, 각 애니메이션 프레임에 대해서 반복적으로 수행되는 부분이다. 따라서 GPU 기반 프로그램에서 병목현상(bottleneck)의 가장 심각한 원인이 되는 데이터의 메모리간 복사 비용을 최소화하는 것은 전체적인 성능 향상을 위한 매우 중요한 고려사항이다. 본 시스템의 구현에서는 렌더링 결과를 직접 텍스춰 메모리로 저장하는 RTT(Render-To-Texture) 기법을 이용하여 이러한 메모리 복사 문제를 최소화 하였다[19]. 기존의 CTT(Copy-To-Texture) 기법에 기반을 둔 프로그램들은 텍스춰 매핑을 통해 수행된 프래그먼트 쉐이더의 연산 결과를 프레임 버퍼에 저장했다가 다시 텍스춰 메모리로 복사하는 비효율적인 메모리 복사를 내부적으로 수행해야만 했다. 최신 GPU들이 RTT를 지원하게 됨으로써 일반 그래픽스 프로그램에서 렌더링 속도를 향상시킬 수 있게 된 것은 물론이고, 특히 GPGPU 분야에서 연산 속도의 증대 효과를 얻을 수 있게 되었다.

본 논문에서는 Cg 1.4.4과 NVIDIA SDK 9.5를 사용하여 프래그먼트 쉐이더를 구현하였으며, 전체 시뮬레이션 연산 가운데 스텝 ①과 ②에 대한 Cg 프래그먼트 쉐이더는 그림 2와 3에서 확인할 수 있다. 특히, 예전에 구현할 수 없었던 프래그먼트 쉐이더 내에서의 반복문사용이 최신 GPU에서는 가능해졌기 때문에, 스텝 ①의 가우시안 필터링을 프래그먼트 쉐이더로 구현할 수 있었다(그림 1). 각 프래그먼트에 대해 이중 반복문(nested loop)을 수행해야만 하는 필터링 스텝을 CPU에서 수행할 수밖에 없다면, 본 시뮬레이션을 실시간으로 처리하는 것은 불가능하다.

4. 실험 결과

본 실시간 유체 애니메이션 프로그램은 Intel Pentium D (3.0 GHz), 1GB RAM을 탑재한 범용 PC

2) 참고로, 그림1의 수식에서 $A \rightarrow B$는 A로부터 B를 계산하여 구한다는 것을 의미한다.

```
//// Step ①: Gaussian Filtering ( ρ → ρ̃ )
void GaussFilt( half2 coords: WPOS,        // grid coordinates
out half4 dNew: COLOR,   // gaussian filtered density
uniform half radius,        // radius of gaussian filter
uniform samplerRECT d,   // density field ρ
uniform samplerRECT k )  // gaussian kernel
{
        half4      vSum = half4(0, 0, 0, 0);
        int        i, j;

        for (i = -radius; i <= radius; i++)
            for (j = -radius; j <= radius; j++)
                vSum += h4texRECT(d, coords+half2(i, j)) * h4texRECT(k, half2(r+i, r+j));

        dNew.xy = vSum.xy;
        dNew.zw = 0.0;
}
```

그림 2. 스텝 ① *Gaussian Filtering* 프래그먼트 프로그램

```
//// Step ②-1: Add Force (F = ρ̃ const ∇ρ̃*/ρ*)
void MultDen( half2 coords : WPOS, // grid coordinates
out half4 f : COLOR,        // driving force F
uniform samplerRECT d,   // filtered density ρ̃
uniform samplerRECT c )  // pre-calculated const ∇ρ̃*/ρ*
{
        half       dd = h1texRECT(d, coords);
        half2      cc = h2texRECT(c, coords);

        f.xy = dd * cc.xy;
        f.zw = 0.0;
}

//// Step ②-2: Add Force (u = u + F)
void AddForce( half2 coords : WPOS,  // grid coordinates
out half4 uNew : COLOR,      // new velocity
uniform half timestep,
uniform samplerRECT u,       // the velocity field
uniform samplerRECT f )      // force field to be added
                             )
{
        half4      vel   = h4texRECT(u, coords);
        half4      force = h4texRECT(f, coords);

        uNew.xy = vel + timestep * force;
        uNew.zw = 0.0;
}
```

그림 3. 스텝 ② *Add Force* 프래그먼트 프로그램

에서 구현되었다. GPU로는 NVIDIA GeForce 6800 GT (256MB)가 사용되었다.

4.1 기본 실시간 유체 시뮬레이션 프로그램

그림 4는 키-프레임 기반 유체 시뮬레이션 기법 개발의 선행 연구로 구현된 실시간 유체 시뮬레이션 프로그램의 수행과정을 캡쳐한 영상이다. 이 프로그램은 유도 힘에 대한 계산과 적용 없이 단순히 사용자가 마우스로 윈도우 화면을 클릭하고 드래그 하는 순간 마우스의 움직임 벡터로부터 속도장을 생성하고, [2]에서 소개된 기본 연산 과정을 적용하여 유체의 흐름에 대한 밀도장의 변화를 반복적으로 디스플레이 하도록 설계되었다. 본 프로그램은 256×256의 해상도에서 30 fps(frames per second) 이상의 속도로 렌더링 가능하다.

그림 4. 기본 실시간 유체 시뮬레이션 프로그램의 캡쳐 영상

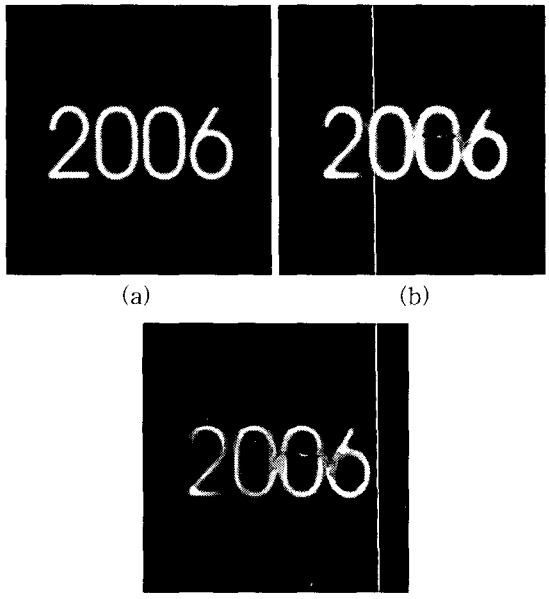
4.2 키-프레임 기반 시뮬레이션

그림 6, 7, 8은 본 논문에서 제안한 기법을 통해 수행된 256×256 해상도의 애니메이션 샘플 프레임을 보여주며, 해상도의 변화에 따른 시뮬레이션 속도 변화의 실험 결과는 표 1과 같다. 이 실험 결과로부터 영상의 해상도가 시뮬레이션 성능에 매우 중요한 요소로 작용한다는 사실을 확인할 수 있으며, 원시 키-프레임과 목적 키-프레임의 복잡도(image complexity)는 시뮬레이션 속도에 전혀 영향을 주지 않는다는 사실을 확인할 수 있다. 이것은 구현된 시뮬레이션 알고리즘의 모든 단계별 연산과 각 애니메이션 프레임이 GPU의 텍스춰 매핑을 통한 SIMD 방식의 병렬 수행 구조를 기본으로 한다는 사실에 기인한다.

시스템 성능에 영향을 미치는 파라미터로 가우시안 커널의 반지름 크기를 생각할 수 있다. 이것은 시뮬레이션 결과 영상에 매우 민감하게 작용하는 파라미터이며, 본 실험에서는 최적의 결과를 얻을 수 있도록 반복적으로 수치를 조절을 통해 최적의 값을 선택하였다.

시뮬레이션 알고리즘에서 단계 ⑤의 사영(projection) 연산에서 $\nabla^2 p = \nabla \cdot u$로부터 p를 구하기 위한 수치해석적인 방법으로 자코비 반복법을 사용하였다. 사용자가 설정하는 자코비 반복 회수의 변화에 따른 시뮬레이션 속도를 측정한 결과는 표 2와 같다.

고화질 프레임 영상 보다 렌더링 속도가 더 중요한 응용의 경우라면 반복 회수로 50 내외의 값을 사용하는 것으로 충분하지만, 속도가 조금 느려지더라도 더 나은 시뮬레이션 화질을 원한다면 반복 회수를 150이상으로 설정해야 한다. 자코비 반복 회수가 애니메이션 프레임의 화질에 미치는 영향은 그림 5에서 확인할 수 있다. (a)는 스텝 ①의 초기화 과정에서 입력으로 주어진 목적 키-프레임이고, (b)와 (c)는 원시 키-프레임에서 출발해 알고리즘의 반복 스텝을 거쳐 목적 키-프레임과 거의 가까운 상태로 변화된 프레임을 캡춰 한 것이다. 이 때, 사용된 자코비 반복 회수는 (b)와 (c) 각각에 대해 150과 50이며, 반복 횟수가 많을수록 더 선명하고 좋은 화질의 영상을 얻을 수 있음을 확인 할 수 있다.



(a) (b)

(c)

그림 5. 자코비 반복 회수의 변화에 따른 애니메이션 프레임 화질: (a) 입력된 목적 영상, (b) 반복 회수 = 150, (c) 반복 회수 = 50

표 1. 프레임 해상도 변화에 따른 시뮬레이션 속도 (단위: fps) (자코비 반복 회수 = 50)

해상도	*Graphics to Wave* (그림 6)	*Wave to Siggraph* (그림 7)	*Siggraph to 2006* (그림 8)	*Elephant to Flower* (그림 9)
128×128	31.18	31.13	30.96	30.28
256×256	13.28	13.28	12.86	12.28
512×512	3.24	3.24	3.23	3.22

표 2. 자코비 반복 회수 변화에 따른 시뮬레이션 속도 (단위: fps) (프레임 해상도 = 256×256)

반복 회수	*Graphics to Wave* (그림 6)	*Wave to Siggraph* (그림 7)	*Siggraph to 2006* (그림 8)	*Elephant to Flower* (그림 9)
50	13.28	13.28	12.86	12.28
100	12.38	12.16	12.20	11.94
150	10.73	10.77	10.16	110.21

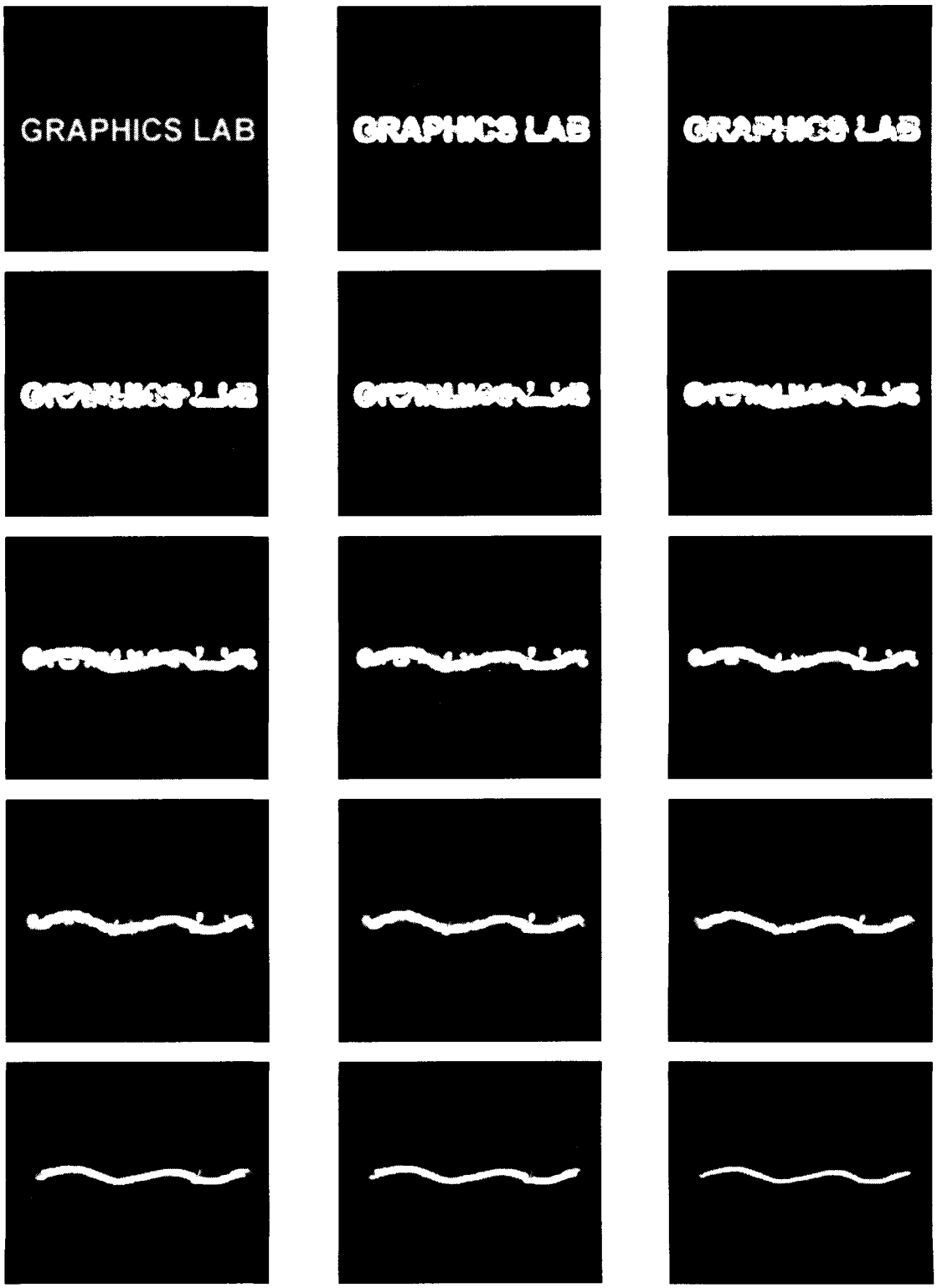


그림 6. 애니메이션 프레임 (1): *Graphics to Wave*

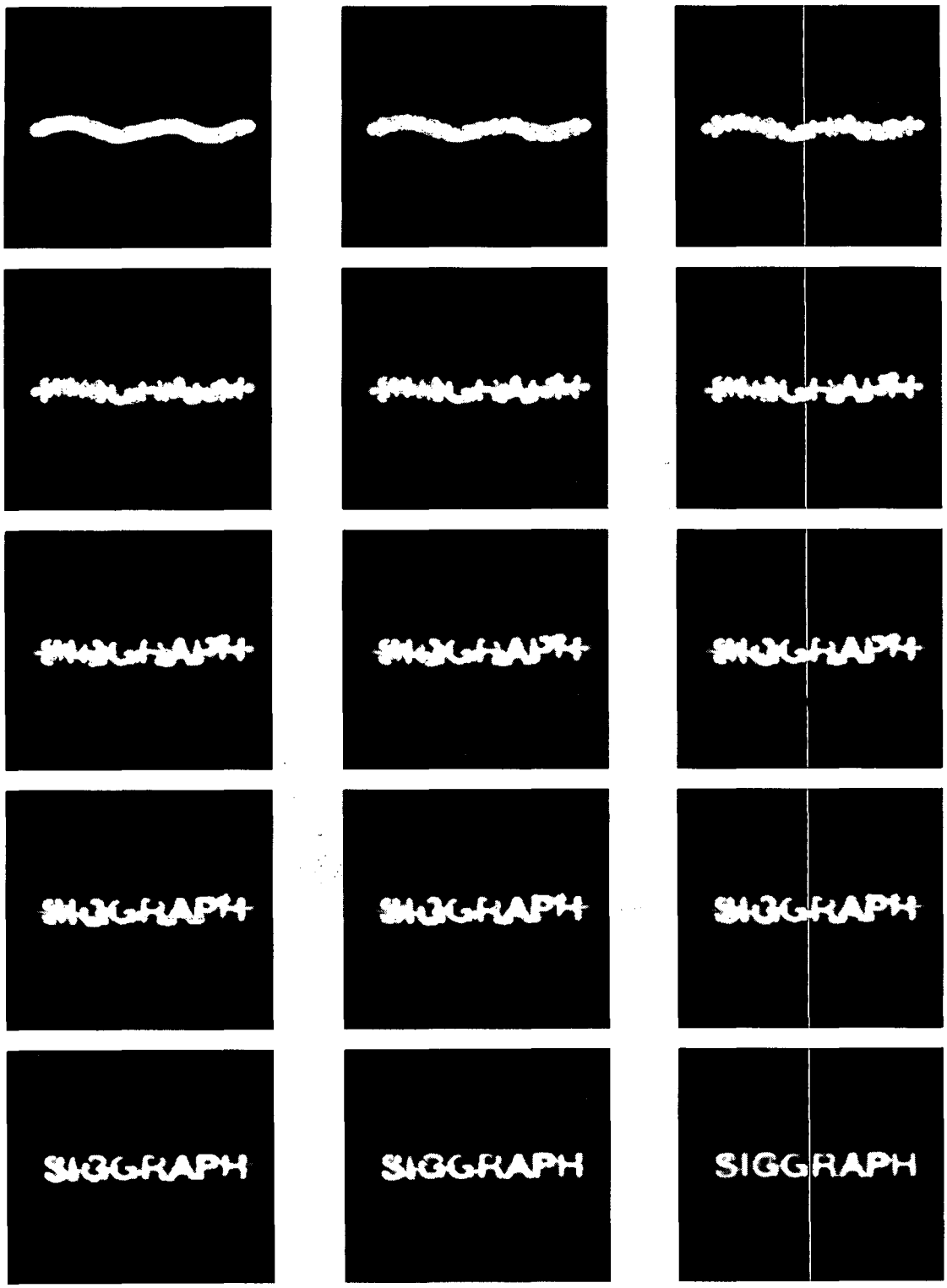


그림 7. 애니메이션 프레임 (2): *Wave to Siggraph*

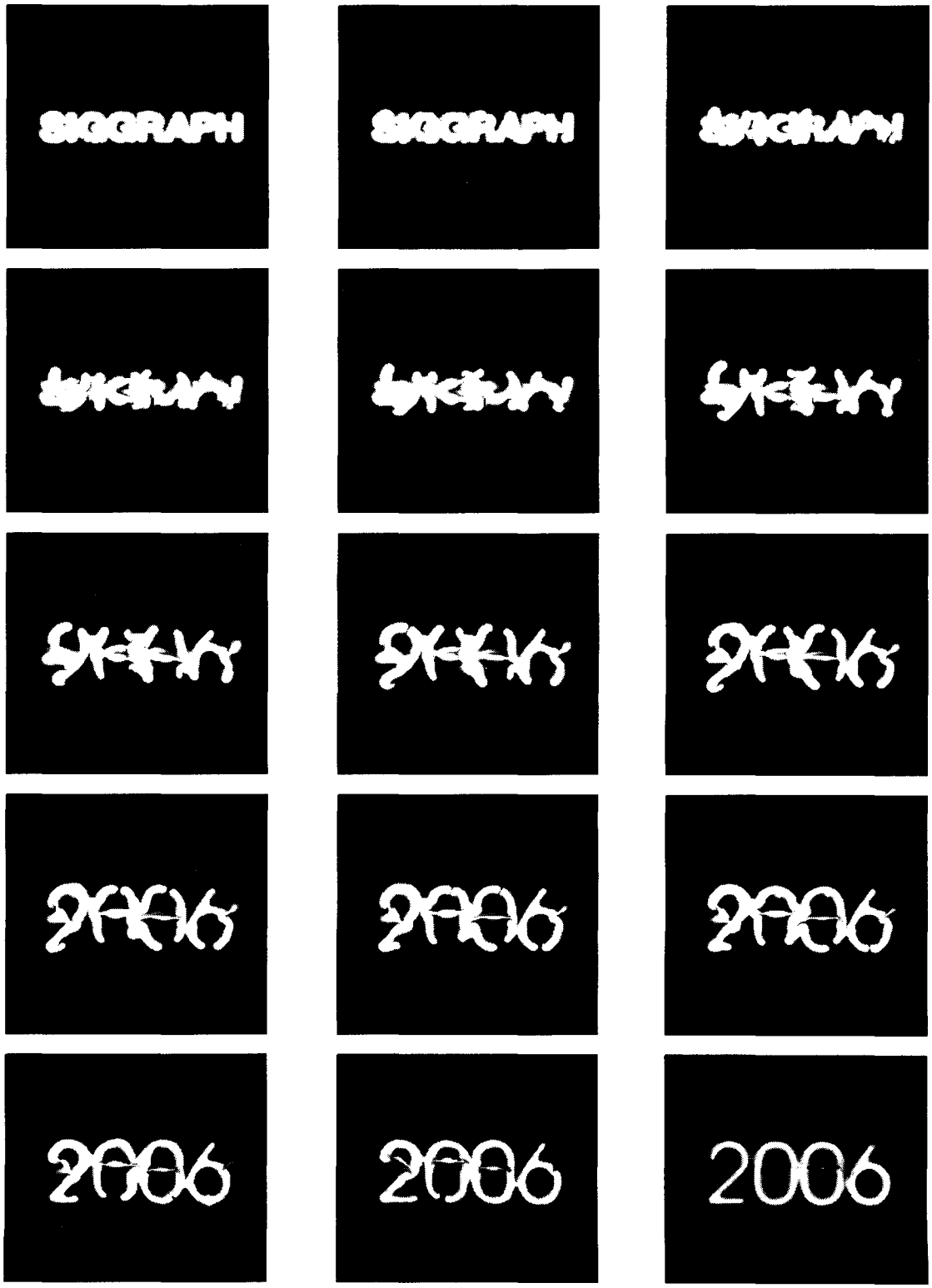


그림 8. 애니메이션 프레임 (3): *Siggraph to 2006*

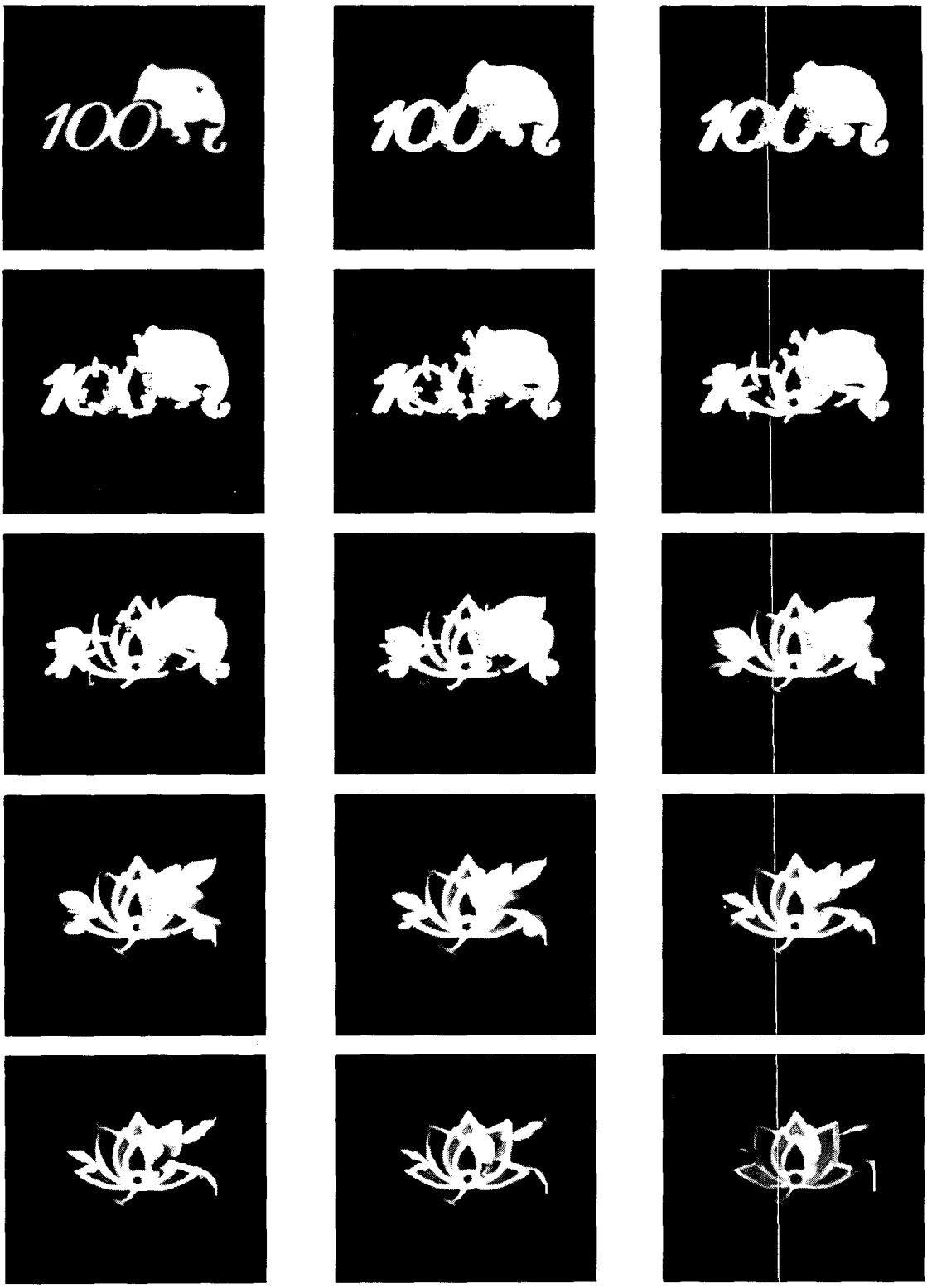


그림 9. 애니메이션 프레임: *Elephant to Flower*

5. 결론 및 향후연구

본 논문에서는 컴퓨터 게임과 같이 실시간 렌더링이 필수적인 응용분야에서 활용될 수 있는 키-프레임 기반의 실시간 유체 시뮬레이션 기법에 대해 설명하였다. 기존의 CPU 기반 오프라인 유체 애니메이션 제작의 경우, 원하는 고화질의 결과 영상을 얻기 위해 복수 개의 파라미터들의 값을 변경시켜가며 수많은 시행착오와 반복을 통해 최적의 애니메이션 프레임들을 제작하기 때문에 너무 많은 시간을 필요로 했다. 반면 본 논문에서 제안한 프로그래머블 그래픽스 파이프라인을 이용한 실시간 유체 애니메이션 기법은 파라미터 변경에 따른 애니메이션 영상의 변화를 실시간으로 확인할 수 있기 때문에, 비록 오프라인 기법들에 비해 화질은 떨어지지만, 애니메이션을 실시간에 제어할 수 있다는 장점을 갖는다.

개발된 방법이 실시간 응용에 효과적으로 이용될 수 있음에도 불구하고 아직까지 다음과 같은 한계를 갖고 있다. 우선, 실험 결과를 통해 확인한 바와 같이, 해상도에 따라 시뮬레이션 속도의 차이가 크고 실시간 성능을 얻을 수 있는 해상도의 크기가 제한된다. 또한, 비디오 메모리 크기와 GPU 성능 제약으로 인해 유체 시뮬레이션을 3차원으로 확장하는데 한계를 갖는다. 그러나 그래픽스 하드웨어 기술의 발전 속도를 고려할 때, 이와 같은 문제는 향후 몇 년 내에 해결될 수 있을 것으로 예상된다.

추후 연구로 본 논문에서 제시한 GPU 기반의 실시간 유체 애니메이션 제어 프로그램과 CPU 기반의 고화질 오프라인 애니메이션 프로그램을 연결한 통합 시스템을 구축할 예정이다. 이 시스템이 성공적으로 구현될 경우, 사용자는 구현될 그래픽 사용자 인터페이스의 미리보기(pre-view) 영상을 통해 실시간으로 원하는 유체 애니메이션의 렌더링 파라미터들을 설정하고, 이를 이용해 고화질 오프라인 프로그램을 구동함으로써 최종 애니메이션을 최소의 비용으로 제작할 수 있을 것이다.

참 고 문 헌

[1] N. Foster and D. Metaxas, "Controlling Fluid Animation," *Proceedings of Computer Graphics International*, pp. 178-188, 1997.

[2] J. Stam, "Stable Fluids," *Computer Graphics (SIGGRAPH 1999)*, pp. 121-128, 1999.

[3] N. Foster and R. Fedkiw, "Practical Animation of Liquids," *Computer Graphics (SIGGRAPH 2001)*, pp. 23-30, 2001.

[4] D. Enright, S. Marschner, and R. Fedkiw, "Animation and Rendering of Complex Water Surfaces," *ACM Transactions on Graphics (SIGGRAPH 2002)*, Vol. 21, No. 3, pp. 736-744, 2002.

[5] A. Treuille, A. McNamara, Z. Popovic, and J. Stam, "Keyframe Control of Smoke Simulation," *ACM Transactions on Graphics (SIGGRAPH 2003)*, Vol. 22, No. 3, pp. 716-723, 2003.

[6] R. Fattal and D. Lischinski, "Target-Driven Smoke Animation," *ACM Transactions on Graphics (SIGGRAPH 2004)*, Vol. 23, No. 3, pp. 441-448, 2004.

[7] M. Harris, W. Baxter, T. Scheuermann, and A. Lastra, "Simulation of Cloud Dynamics on Graphics Hardware," *Proceedings of Graphics Hardware*, pp. 92-101, 2003.

[8] M. Harris, "Fast Fluid Dynamics Simulation on the GPU," *GPU Gems*, pp. 637-665, Addison-Wesley, 2004.

[9] W. Li, Z. Fan, X. Wei, and A. Kaufman, "Flow Simulation with Complex Boundaries," *GPU Gems2*, pp. 747-764, Addison-Wesley, 2005.

[10] M. Harris, "Mapping Computational Concepts to GPUs," *GPU Gems2*, pp. 493-508, Addison-Wesley, 2005.

[11] R. Rost, *OpenGL Shading Language (2nd Ed.)*, Addison-Wesley, 2006.

[12] W. Mark, R. Glanville, K. Akeley, and M. Kilgard, "Cg: A System for Programming Graphics Hardware in a C-like Language," *ACM Transactions on Graphics (SIGGRAPH 2003)*, Vol. 22, No. 3, pp. 896-907, 2003.

[13] W. Engel (Editor), *ShaderX2: Introductions and Tutorials with DirectX 9.0*, Wordware Publishing, Inc., 2003.

[14] M. McCool, S. DuToit, and S. Toit, *Metapro-*

gramming GPUs with Sh, AK Peters, Ltd., 2004.

[15] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, "Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid," *ACM Transactions on Graphics*, Vol. 22, No. 3, pp. 917-924, 2003.

[16] K. Hillesland, S. Molinov, and R. Grzeszczuk, "Nonlinear Optimization Framework for Image-Based Modeling on Programmable Graphics Hardware," *ACM Transactions on Graphics* (*SIGGRAPH 2003*), Vol. 22, No. 3, pp. 925-934, 2003.

[17] J. Krüger and R. Westermann, "Linear Algebra Operators for GPU Implementation of Numerical Algorithms," *ACM Transactions on Graphics* (*SIGGRAPH 2003*), Vol. 22, No. 3, pp. 908-916, 2003.

[18] J. Krüger and R. Westermann, "A GPU Framework for Solving Systems of Linear Equations," *GPU Gems2*, pp. 703-718, Addison-Wesley, 2005.

[19] A. Lefohn, J. Kniss, and J. Owens, "Implementing Efficient Parallel Data Structures on GPUs," *GPU Gem2*, pp. 521-545, Addison-Wesley, 2005.

유 지 현

1992년 2월 서강대학교 수학과 졸업(이학사)
1994년 2월 서강대학교 수학과 졸업(이학석사)
1999년 2월 서강대학교 수학과 졸업(이학박사)
2000년 7월~2003년 5월 University of Texas at Austin 박사후연구원
2004년 1월~2004년 12월 학술진흥재단 박사후연구원
2005년 3월~2006년 2월 서강대학교 시간강사
2006년 3월~현재 세종대학교 응용수학과 초빙교수
관심분야 : 최적화 이론, 계산 기하, 물리기반 시뮬레이션, 컴퓨터 그래픽스 등

박 상 훈

1993년 8월 서강대학교 수학과 졸업(이학사)
1995년 8월 서강대학교 컴퓨터학과 졸업(공학석사)
2000년 2월 서강대학교 컴퓨터학과 졸업(공학박사)
2000년 3월~2000년 6월 서강대학교 컴퓨터학과 박사후연구원
2000년 7월~2002년 8월 University of Texas at Austin 박사후연구원
2002년 9월~2005년 2월 대구가톨릭대학교 컴퓨터정보통신공학부 조교수
2005년 3월~현재 동국대학교 영상대학원 멀티미디어학과 조교수
관심분야 : 컴퓨터 그래픽스, 과학적 가시화, 가상현실, 모바일 컴퓨팅, 컴퓨터 게임 등.