KIPS Tr. Software and Data Eng.
Vol.6, No.10 pp.479~486 pISSN: 2287-5905

https://doi.org/10.3745/KTSDE.2017.6.10.479

Performance Comparison of Particle Simulation Using GPU Between OpenGL and Unity

Kim Min Sang† · Nak-Jun Sung† · Yoo-Joo Choi†† · Min Hong†††

ABSTRACT

Recently, GPGPU has been able to increase the degradation of computer performance, and it is now possible to run physically based real-time simulations on PCs that require high computational complexity. Physical calculations applied in physics simulation can be performed by parallel processing, and can be efficiently performed using parallel computation using Compute shader recently supported by OpenGL 4.3 and Unity 4.0. In this paper, we measure and compare the number of performance in real - time physics simulation in OpenGL running on various platforms and Unity, a content creation tool supporting various platforms. Particle simulation experiments show that particle simulation using Unity performs faster than 136.04%. It is expected that it will be able to select better development tools for future multi - platform support.

Keywords : Particle Simulation, Unity Shader, GPU Programming, OpenGL, Real-Time Simulation, Parallel Programming, Compute Shader

OpenGL과 Unity간의 GPU를 이용한 Particle Simulation의 성능 비교

김 민 상† · 성 낙 준† · 최 유 주†† · 홍 민†††

요 약

최근 GPGPU를 이용하여 저하된 컴퓨터 성능 향상폭을 높일 수 있게 되었고, 이로 인하여 높은 연산을 요구로 하는 물리 기반의 실시간 시뮬레이션을 PC에서 구동할 수 있게 되었다. 물리 시뮬레이션에서 적용되는 물리 계산은 병렬 처리로 수행되어질 수 있으며, 최근 OpenGL 4.3 및 Unity4.0에서 지원되는 Compute shader를 통한 병렬 연산을 이용하면 효율적으로 구동할 수 있다. 본 논문에서는 다양한 플랫폼을 지원하는 디지털 콘텐츠 제작 툴인 Unity와 다양한 플랫폼에서 구동되어지는 OpenGL에서의 실시간 물리 시뮬레이션에서의 성능을 측정 및 비교한다. 본 논문에서 particle 시뮬레이션의 실험 결과 Unity를 이용한 particle 시뮬레이션이 OpenGL을 이용한 particle 시뮬레이션에 비해 최대 136.04% 빠른 성능을 보인다. 이를 통하여 추후 멀티 플랫폼을 지원하는 디지털 콘텐츠를 제작함에 있어 더 나은 개발 도구를 선정할 수 있을 것으로 기대된다.

키워드 : Particle 시뮬레이션, 유니티 셰이더, GPU 프로그래밍, OpenGL, 실시간 시뮬레이션, 병렬 프로그래밍, 컴퓨트 셰이더

1. 서 론

1960년대부터 Moore's Law에 따른 컴퓨터 하드웨어 성능의 발전으로 인해 CPU(Central Processing Unit)의 성능이 꾸준히 증가하였으며, 2000년대 중반을 이후로는 단일 프로세서에서의 Moore's Law가 깨지게 되면서 컴퓨터 연산 속도의 발전에 한계가 발생하였다. 그러나 GPU(Graphics Processing Unit)의 성능은 꾸준히 Moore's Law를 따르게 되면서 GPU를 기반으로 하는 Nvidia의 CUDA(Compute Unified Device Architecture)를 필두로 한 GPGPU(General Purpose computing on Graphics Processing Unit)을 통해 개인용 컴퓨터에서도 물리 법칙이 적용된 보다 현실적인 시뮬레이션 결과를 실시간으로 렌더링 할 수 있게 되었다[1].

현실에서의 물리 법칙을 컴퓨터 시뮬레이션으로 구현하기 위해서는 하나의 물체를 여러 개의 질점(Material Point)을

※ 본 논문은 2017년도 한국정보처리학회 춘계학술발표대회에서 '데스크탑에서의 OpenGL과 Unity 3D간의 성능 비교'의 제목으로 발표된 논문을 확장한 것임.
※ 본 연구는 한국연구재단 이공학개인기초연구지원사업 기본연구지원사업 (NRF-2015R1D1A1A01059304)에 의하여 수행되었음.
※ 본 연구는 순천향대학교 학술연구비 지원으로 수행되었음.
† 준 회 원 : 순천향대학교 컴퓨터학과 석사과정
†† 종신회원 : 서울미디어대학원대학교 뉴미디어학부 부교수
††† 종신회원 : 순천향대학교 컴퓨터소프트웨어공학과 교수
Manuscript Received : July 18, 2017
Accepted : August 27, 2017
* Corresponding Author : Min Hong(mhong@sch.ac.kr)

가진 구조로 이해해야 한다[3]. 이 때, 각각의 질점에 계산되어지는 물리량은 각각의 질점에 대한 병렬 문제로 구성될 수 있다. 그리고 GPU는 하나의 다이 내에 CPU보다 많은 코어의 개수가 존재하기 때문에, 이러한 병렬 처리에 있어 CPU에 비해 빠르게 실시간 시뮬레이션을 구현하는 것이 적합하다.

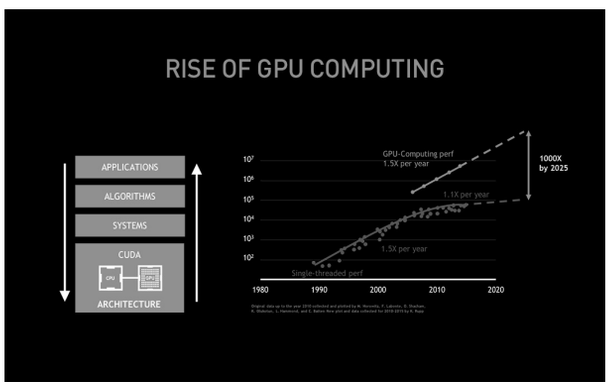


Fig. 1. Performance Improvement of Moore's Law and CPU and GPU[2]

최근 다양한 플랫폼으로의 디지털 콘텐츠 개발이 이루어지고 있는 가운데, 대표적인 멀티 플랫폼을 제공하는 디지털 콘텐츠 개발 환경으로는 OpenGL과 Unity 등이 있다. OpenGL과 Unity는 공통적으로 GLSL(OpenGL Shader Language)을 지원하는 특징을 가지고 있다[4-6]. 또한 GLSL 4.3 버전부터 추가된 compute shader를 이용하면 GPU를 이용하여 렌더링 파이프라인 이외에도 병렬 프로그래밍 연산을 수행 할 수 있다. 본 논문에서는 같은 shader 언어인 GLSL을 사용하는 그래픽 라이브러리 OpenGL과 게임 엔진인 Unity간의 렌더링 및 연산에 대한 성능 비교를 수행한다.

본 논문의 구성은 다음과 같다. 2장은 본 연구와 관련된 연구 진행 상황을 소개하며, 3장은 OpenGL과 Unity의 비교를 위한 particle 시뮬레이션 시스템의 설계 및 구현에 대해 기술한다. 4장은 구현된 시스템에 대한 다양한 particle 시뮬레이션 실험을 진행하고 각 방법에 대한 성능 비교를 수행한다. 마지막으로 5장은 본 논문의 결론을 기술한다.

2. 관련 연구

2.1 수치적분

본 논문에서는 다양하게 가상의 물체와 반응하여 움직이는 particle의 움직임을 계산하기 위해 수치적분을 이용하였다[7-8]. 수치적분은 각각의 particle에 가해지는 힘으로부터 현재 시간으로부터 일정한 시간($\triangle t$)이 지난 뒤의 particle들의 가속도를 구할 수 있다. 또한 구한 가속도를 통해 $\triangle t$ 후의 particle들의 속도를 구할 수 있고, 이러한 속도를 통해 $\triangle t$ 후의 particle들의 위치를 구하는 방법이다. 본 논문에서 사용된 수치적분법은 particle 시뮬레이션의 정확도를 유지하기 위해 4th order Runge-Kutta 방법을 이용하였다.[9] 4차 Runge-Kutta 방법은 미지의 곡선이 주어질 때, 시작점 t_0의 속도를 바탕으로 미분 방정식에서 t_0 지점에서의 기울기가 의미하는 값인 t_0에서의 가속도 k_{a1}를 Equation (1)과 같이 구해 $t_0+\frac{\triangle t}{2}$의 속도를 예상하고, 이때의 $t_0+\frac{\triangle t}{2}$에서의 가속도 k_{a2}를 Equation (2)와 같이 구한다. 또한 k_{a2}를 이용하여 $t_0+\frac{\triangle t}{2}$에서의 속도를 구하며, 이때의 가속도인 k_{a3}를 Equation (3)과 같이 구한다. 마지막으로 k_{a3}를 이용하여 t_1에서의 속도를 구하고, t_1에서의 가속도인 k_{a4}를 Equation (4)와 같이 구한다. 위에서 구해진 가속도들 $k_{a1},k_{a2},k_{a3},k_{a4}$에 가중치를 준 후, 가중치의 합으로 나눈 값을 t_0에서의 가속도로 이용하여 t_1에서의 속도를 구하는 방식이다. 위와 같은 과정을 시간-위치에 좌표계에 적용하면 Equation (5)와 같이 t_1에서의 속도를 구할 수 있다.

본 시뮬레이션 시스템에서는 이러한 방식을 응용하여 일정한 시간의 particle들의 위치에 따라 다음 시간단위 뒤의 particle들의 위치를 구하는 방식을 사용하였다. particle들에 가해지는 힘은 중력가속도 G와 particle의 질량 m의 곱으로 표현할 수 있으며 특정한 외부의 힘이 없다면 particle의 가속도는 $a=G$로 표현될 수 있다.

다음의 식에서의 $f(t,v)$ 는 t 순간에서의 속도의 값이다.

$$k_{a1}=\triangle t f(t_{t_0},v_{t_0}) \tag{1}$$

$$k_{a2}=\triangle t f(t_{t_0}+\frac{1}{2}\triangle t,v_{t_0}+\frac{1}{2}\triangle t^* k_{a1}) \tag{2}$$

$$k_{a3}=\triangle t f(t_{t_0}+\frac{1}{2}\triangle t,v_{t_0}+\frac{1}{2}\triangle t^* k_{a2}) \tag{3}$$

$$k_{a4}=\triangle t f(t_{t_0}+\frac{1}{2}\triangle t,v_{t_0}+\frac{1}{2}\triangle t^* k_{a3}) \tag{4}$$

$$v_{t_1}=v_{t_0}+\frac{(k_{a1}+2k_{a2}+2k_{a3}+k_{a4})}{6}\triangle t \tag{5}$$

Equation (5)에서 구한 속도 v_{t_1}을 이용하여 t_1에서의 다음 particle들의 위치의 값 p_{t_1}는 위의 과정과 동일한 과정을 거쳐 Equation (10)과 같이 구할 수 있다.

$$k_{v1}=\triangle t f(t_{t_0},p_{t_0}) \tag{6}$$

$$k_{v2}=\triangle t f(t_{t_0}+\frac{1}{2}\triangle t,p_{t_0}+\frac{1}{2}\triangle t^* k_{v1}) \tag{7}$$

$$k_{v3}=\triangle t f(t_{t_0}+\frac{1}{2}\triangle t,p_{t_0}+\frac{1}{2}\triangle t^* k_{v2}) \tag{8}$$

$$k_{v4} = \triangle t f(t_{t_0} + \frac{1}{2}\triangle t, p_{t_0} + \frac{1}{2}\triangle t^* k_{v3}) \quad (9)$$

$$p_{t_1} = p_{t_0} + \frac{(k_{v1} + 2k_{v2} + 2k_{v3} + k_{v4})}{6}\triangle t \quad (10)$$

2.2 Shader 언어

1) OpenGL과 GLSL

OpenGL은 Khronos 그룹에서 제정한 2D, 3D 그래픽 오픈 소스 라이브러리이다. 2012년 8월에 배포된 OpenGL 4.3의 shader 언어인 GLSL은 기존에 없던 compute shader를 추가하였다. Compute shader는 기존 파이프라인과 달리 독립된 shader로 주로 임의의 정보를 계산하는 용도로 사용된다[10]. 이러한 compute shader는 GLSL 4.3 이상 버전의 프로그램으로 접근 가능하며, 연산 속도를 향상시키기 위해 GPU에서 병렬처리 연산을 수행할 수 있다. 연산 작업량을 사용자가 지정한 워크 그룹의 크기만큼 (X, Y, Z) 3차원 작업공간으로 분할 할당이 가능하다. 워크 그룹이란 compute shader가 동작하는 단위 작업공간이며 하나의 워크 그룹에 할당된 연산의 양을 GPU 병렬처리를 통해 수행한다[7]. Fig. 2는 OpenGL 4.3과 GLSL 4.3의 파이프라인을 나타낸다[11].

2) Unity와 HLSL

Unity는 Unity Technologies에서 개발한 게임 및 디지털 콘텐츠 개발 툴이다. Unity는 Rendering Shader 언어로 ShaderLab과 HLSL(High Level Shader Language)을 지원하고 있으며, compute shader 언어로 HLSL과 GLSL을 지원하고 있다[12]. Unity에서 HLSL로 작성된 compute shader는 Unity 프로젝트에서 목표로 하는 플랫폼에 따라서 자동을 GLSL으로 번역되어 컴파일 된다. GLSL을 이용하여서도 compute shader를 작성할 수 있지만, 크로스 플랫폼을 지원하는 데에 어려움이 발생할 수 있으므로, Unity에서는 HLSL을 이용한 compute shader 작성을 권고하고 있다.

2.3 Compute shader

본 논문에서 위의 4차 Runge-Kutta 방법을 적용하여 particle들의 위치를 계산하는 compute shader의 입출력 데이터는 Table 1과 같다. 본 논문에서 사용된 compute shader의 local group size는 8, 1, 1이며, 이는 Fig. 3과 같다[13].

Table 1. Data Structure of Data used in Simulation

	Description	Data Type
Input	Position of current particle	float3[]
	Velocity of current particle	float3[]
	Mass of particle	float[]
	Center coordinates of colliding sphere	float3
	Time elapsed from previous frame	float
Output	Position of updated particle	float3[]
	Velocity of updated particle	float3[]

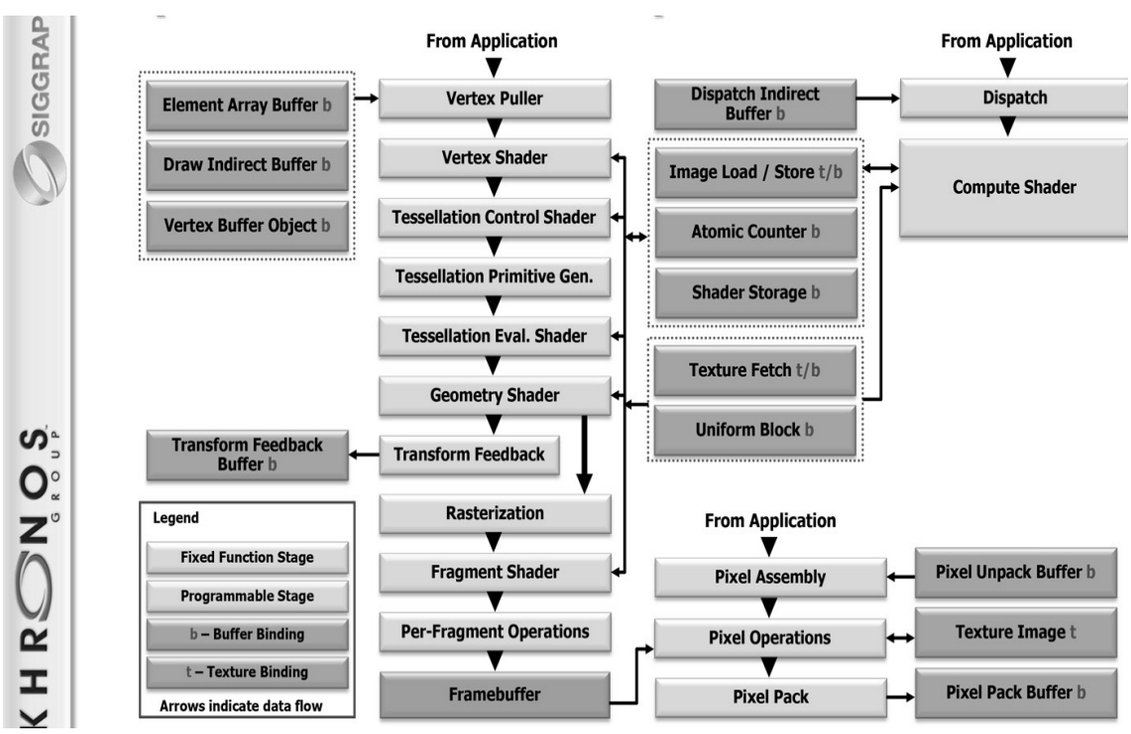


Fig. 2. Pipeline of OpenGL 4.3 and GLSL 4.3

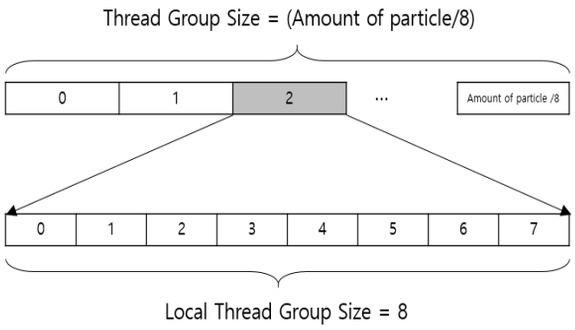


Fig. 3. Designed Compute Shader Workgroup

1) Compute shader pseudo code

본 논문에서 사용된 compute shader의 pseudo code는 아래의 Fig. 4와 같다. Compute shader 내부에서는 앞서 설명한 4차 Runge-Kutta 방법을 이용하여 particle들의 다음 위치를 계산한다. 이후에는 직육면체와의 충돌 여부를 검사한 후, 구와의 충돌 검사를 수행한다. Particle들과 다른 물체들 간의 간단한 충돌 및 반응을 반영하기 위해 직육면체와의 충돌 시에는 x, y, z의 값만을 비교하며, 구와의 충돌 시에는 particle들의 위치와 구의 중심과의 거리를 이용하여 충돌 여부를 판단하였다.

Compute Shader to calculate particle position

Input : Position of particles,
velocity of particles,
particle mass,
sphere's position,
delta time from last frame

Output: updated particle positions,
updated particle velocities

Begin
Calculate the acceleration of the particle
Calculate the velocity of the particle with RK4
Calculate the new position of the particle with RK4

If particle's x position is outer cube's x position
Calculate particle's actual position when collided to cube's edge
Reflect particle's velocity with cube face's vector

If distance between particle sphere's core is lower than sphere's radius
Calculate particle's actual position when collided to sphere
Reflect particle's velocity with sphere face's vector
End

Fig. 4. Compute Shader's Pseudo Code to Compute the Location of Particles

3. 시뮬레이션 시스템

본 논문에서 OpenGL과 Unity 간 성능 비교를 위한 실험 시뮬레이션 환경은 Fig. 5와 같다. 월드 좌표계의 중심에는 반지름이 1.0 cm 인구가 존재하며, 구의 중심을 가로지르는 $5 \times 0.1 \times 5\ cm^3$의 직육면체가 존재한다.

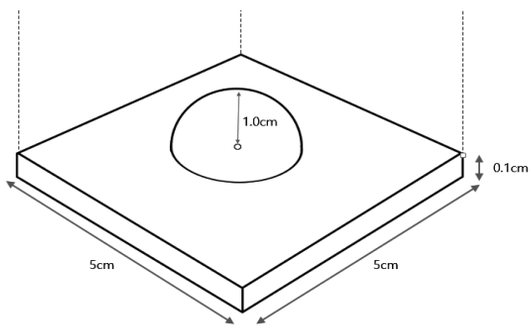


Fig. 5. Simulation Environment

Particle들은 Table 2의 모델의 형태로 구의 0.5 cm 위에서 낙하한다. 낙하하는 particle들은 구 표면 및 직육면체의 표면에 충돌할 경우 반대 방향으로 튕겨지게 된다. 또한 particle들이 화면 밖으로 튕겨지는 것을 방지하기 위해 직육면체의 x-z좌표의 경계에 가상의 벽을 만들어 모든 particle들이 Fig. 6과 같이 x-z 좌표계에서 직육면체 밖으로 튕겨져 나가지 않도록 하였다.

Fig. 7의 시뮬레이션 결과에서 보이는 각각의 particle들의 색은 fragment shader 내부에서 현재 프레임에서 particle들의 x, y, z축 방향으로의 속도에 대한 절대값을 구한 후, 정규화한 벡터의 x축으로의 속도, y축으로의 속도, z축으로의 속도를 각각 R, G, B에 대입하여 색으로 표현하였다. 이를 통해 particle들의 진행 방향을 확인할 수 있게 하였다. 따라서 x방향으로의 움직임은 빨강색으로, y방향으로의 움직임은 녹색으로 z방향으로의 움직임은 파란색으로 렌더링 된다. 아래의 Fig. 8은 Happy Buddha 모델을 이용하여 시뮬레이션을 실행한 후 15프레임이 경과한 후의 모습이다. OpenGL에 의한 결과는 Unity의 결과에 비해 충돌면에서의 particle들의 입자가 크게 보이나, 이는 Unity에서 particle을 더 크게 렌더링하기 때문이다.

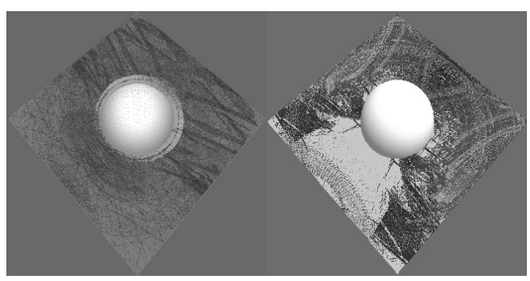


Fig. 6. Particles of Alphabet O Model Bounced off the Wall
(Left: OpenGL, Right: Unity)

Table 2. The Type of Model used in the Simulation (Left: OpenGL, Right: Unity)

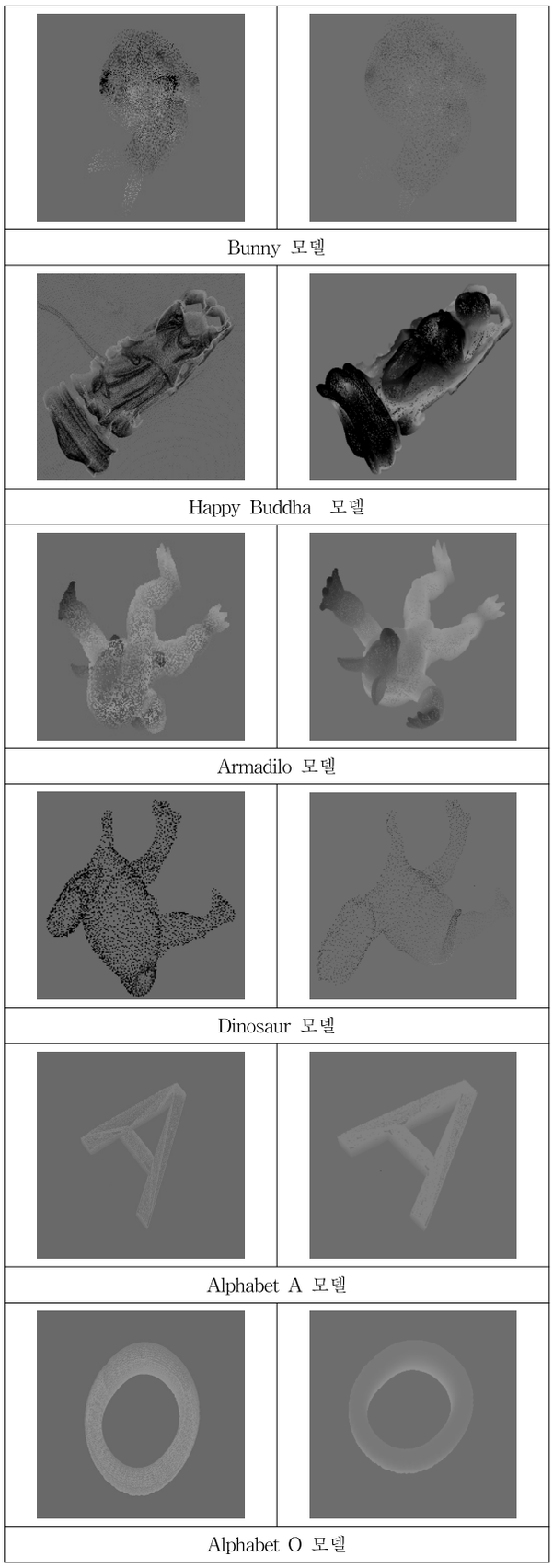


Table 3. The Type of Model used in the Simulation, the Number of Particles and the Size of the Workgroup

Model name	Size of model (Number of particles)	Number of workgroups
Bunny	8,574	1072, 1, 1
Happy Buddha	543,514	67940, 1, 1
Armadillo	227,079	28385, 1, 1
Dinosaur	3,895	487, 1, 1
Alphabet A	301,806	37726,1 ,1
Alphabet O	983,040	122880, 1, 1

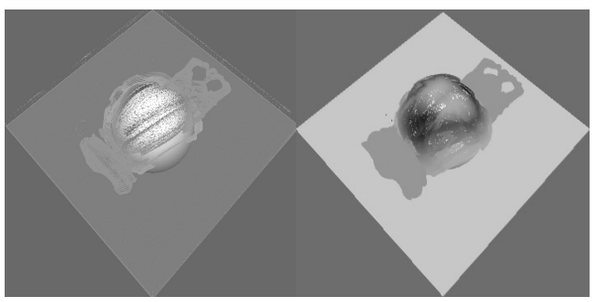


Fig. 7. Scene of Happy Buddha Model Dropped (15 frame, Left: OpenGL, Right: Unity)

4. 실 험

4.1 실험 환경

본 논문에서 OpenGL 및 Unity와의 성능 비교를 위해 수행한 하드웨어 및 소프트웨어의 환경은 다음과 같다.

Table 4. Hardware and Software used in the Simulation

Components	Names and versions
CPU	Intel Core i5 4460
Mainboard	ASUSTeK H97M-E
BIOS	American Megatrends 2302
RAM	Samsung DDR3 PC3-12800 (m378b5173eb0-ck0) 8GB
VGA	NVIDIA GeForce GTX 750 VRAM 1024MB
Windows	Windows 10.0.15063.483
Unity	5.6.2f1 personal
OpenGL	GL Core 4.4
Monitor	LG FLATRON 24EN43V

4.2 측정 방식

본 논문에서 OpenGL과 Unity에서 구현된 particle 시뮬레이션의 성능 비교를 위해 매 프레임 간의 속도 비교를 위해 수행한 시뮬레이션 환경은 다음과 같다.

1) 수직 동기화 해제

테스트에서 사용된 모니터의 주파수가 60 Hz이기 때문에 수직 동기화(v-sync)를 이용하게 되면 60 fps(frames per second) 이상의 속도로 렌더링을 할 수 없게 된다[13]. 따라서 각각의 환경에서 수직 동기화를 해제하였다. Unity의 경우 상단 메뉴의 Edit의 하위 메뉴 중 Project Settings의 하위 메뉴 중 Quality 메뉴에서 V Sync Count 항목에서 Don't Sync로 설정하였고, OpenGL의 경우 GLFW 라이브러리에서 제공하는 glfwSwapInterval 메소드의 parameter 값에 0을 넘겨주어 수직 동기화를 해제하여 실험하였다.

2) 초당 프레임 수 측정

동일한 시뮬레이션 환경에서 초당 프레임 수를 측정하게 되면 두 환경에서의 성능 차이를 비교할 수 있다. 본 논문에서는 두 환경에서 하나의 프레임을 렌더링하는 데에 걸린 시간을 측정하기 위해 다음과 같이 수행하였다. Unity의 경우 Unity에서 제공하는 UnityEngine.Time.deltaTime 값을 통해 이전 프레임을 렌더링하고 난 후의 시간을 구할 수 있다. OpenGL의 경우 ISO/IEC 14882:2011 (C++11)에서 제공하는 chrono 헤더 파일 내의 system_clock::now() 메소드를 이용하여 이전 프레임으로부터의 시간을 millisecond 단위로 계산하였다. 본 논문에서는 보다 정확한 결과값을 비교하기 위해 200 프레임 동안의 시간을 누계한 후 이의 역수를 구하여 초당 렌더링한 프레임 수(frames per second: fps)를 구하였다. 각 시뮬레이션은 10회 반복 실행한 후 평균값을 사용하여 Fig. 9와 같이 비교하였다.

4.3 성능 비교

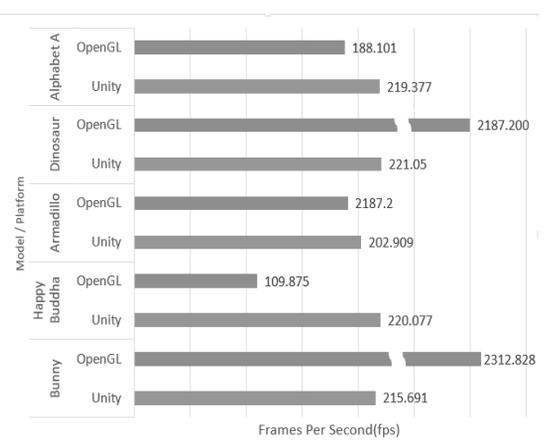


Fig. 8. Graph of Frames per Second Measured by Model / Platform

Fig. 8의 결과에서는 상대적으로 적은 particle(10,000개미만의 particle)들을 가지는 모델을 시뮬레이션하여 렌더링 할 경우 OpenGL에서의 초당 프레임 수가 Unity에 비해 압도적으로 빠른 것을 확인할 수 있다. 이를 Fig. 9와 같이 Unity 내부의 Profiler를 이용하여 분석해 보면 해당 프레임에서의 Unity 내부의 Worker Thread가 idle 상태에 의해 Unity의 main thread가 정지되어 프레임이 제한되어 있는 상태이다. Unity의 Profiler를 기준으로 실제 렌더링되는 시간을 계산하면 약 0.4ms로, 실제적인 초당 프레임 수는 약 2,500으로 계산되어진다. 이는 같은 모델을 렌더링 했을 때의 OpenGL 플랫폼의 결과값과 근사한 값이다.

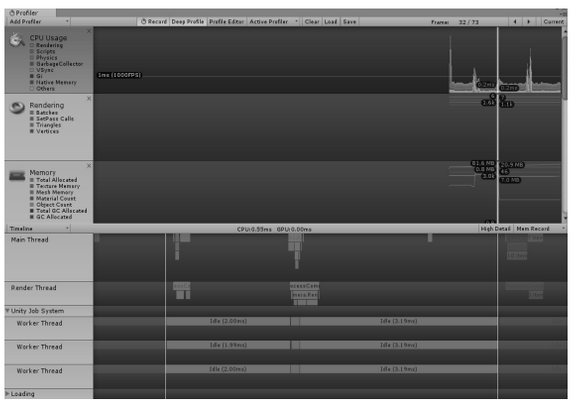


Fig. 9. Screen of Unity Profiler

Fig. 10은 각 모델에서의 노드 수와 Unity/OpenGL간의 성능 차의 상관관계를 보여주고 있다. 이때 Unity 환경에서는 적은 연산량에 대해 Unity가 자동으로 worker thread를 Idle 상태로 유지하는 것을 방지하는 것을 설정할 수 있는 옵션이 존재하지 않기 때문에 적은 particle 수로 인해 실행 시간이 오래 걸리는 Bunny와 Dinosaur 모델은 배제하였다.

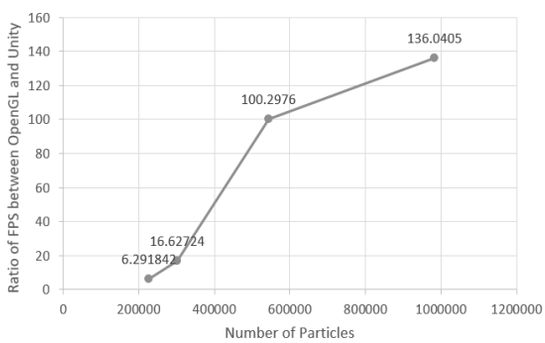


Fig. 10. Correlation between FPS Ratio and Number of Particles between Unity / OpenGL

Fig. 10에서 particle의 수와 Unity/OpenGL의 fps의 성능 비율은 거의 선형적인 관계를 보인다. 더 많은 particle들을 적용하여 시뮬레이션을 수행할수록 Unity에서의 성능 향상률이 증가한다. 전반적인 결과로 compute shader를 이용한 물리 기반의 실시간 시뮬레이션을 구동하는 환경으로는 Unity가 OpenGL에 비해 최소 6%에서 최대 136%정도 더 나은 성능을 보이는 것으로 나타났다.

5. 결 론

최근 GPGPU를 이용하여 컴퓨터의 연산 성능을 비약적으로 증가시킬 수 있게 되었고 이에 따라 계산이 비교적 많이 요구되는 물리 기반의 시뮬레이션을 PC에서 실시간으로 실행할 수 있게 되었다. 본 논문에서는 GLSL에서 일반적인 용도로 GPU를 이용하여 계산을 할 수 있는 compute shader를 이용하여 OpenGL과 Unity에서의 particle 시뮬레이션의 성능을 비교하였으며 실험 결과 Unity를 이용할 경우 OpenGL을 이용하여 구현한 particle 시뮬레이션보다 최대 136%의 성능을 얻을 수 있었다.

추후 물리 기반의 시뮬레이션을 적용하는 콘텐츠를 제작하고 하는 개발자들이 구현 방법에 대한 가이드라인으로 활용할 수 있으리라 기대한다. 또한 본 논문의 결과를 기반으로 향후 다양한 물리 기반의 디지털 콘텐츠 제작을 위해 cloth, 변형물체 등 다양한 시뮬레이션 및 개발 플랫폼에 대한 연구와 다양한 개발 플랫폼에 대한 연구를 진행할 계획이다.

References

[1] Alan Patterson, Nvidia CEO says Moore's Law is dead [Internet], http://www.eetimes.com/document.asp?doc_id= 1331836.

[2] Fedscoop, The Era of AI Computing [Internet], https://www.fedscoop.com/era-ai-computing/.

[3] D. Sulsky, S. J. Zhou, and H. L. Schreyer, "Application of a particle-in-cell method to solid mechanics," *Computer Physics Communications*, Vol.87, Issues 1-2, pp.236-252, 1995.

[4] Unity, Compute Shaders [Internet], https://docs.unity3d.com/Manual/ComputeShaders.html.

[5] Khronos Group, OpenGL Shading Language [Internet], https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language.

[6] Rost, Randi J. Bill Licea-Kane, Dan Ginsburg, John Kessenich, Barthold Lichtenbelt, Hugh Malan, Mike Weiben, "OpenGL shading language," Pearson Education, 2009.

[7] Davis, Philip J., and Philip Rabinowitz, "Methods of numerical integration," Courier Corporation, 2007.

[8] Ryckaert, Jean-Paul, Giovanni Ciccotti, and Herman JC Berendsen, "Numerical integration of the cartesian equations of motion of a system with constraints: molecular dynamics of n-alkanes," *Journal of Computational Physics*, Vol.23, No. 3, pp.327-341, 1977.

[9] Fourth Order Runge-Kutta [Internet], http://lpsa.swarthmore.edu/NumInt/NumIntFourth.html.

[10] Knronos Group, Compute Shader [Internet], https://www.khronos.org/opengl/wiki/Compute_Shader.

[11] AnandTech, Khronos Announces OpenGL ES 3.0, OpenGL 4.3, ASTC Texture Compression, & CLU [Internet], http://www.anandtech.com/show/6134/khronos-announces-opengl-es-30-opengl-43-astc-texture-compression-clu/3.

[12] Unity, Shading Language used in Unity [Internet], https://docs.unity3d.com/Manual/SL-ShadingLanguage.htm.

[13] Young-Hwan Choi, Min Hong, Seung-Hyun Lee, Yoo-Joo Choi, "The Performance Analysis of GPU-based Cloth simulation according to the Change of Work Group Configuration," *Journal of Internet Computing and Services*, pp.29-36, 2017.

[14] Wikipedia, Vsync [Internet], https://en.wikipedia.org/wiki/Vsync_(computing).

김 민 상

e-mail : ben399399@sch.ac.kr

2017년 순천향대학교 컴퓨터학과 석사과정

관심분야 : Computer Graphics, Real Time Simulation

성 낙 준

e-mail : njsung@sch.ac.kr

2016년～현 재 순천향대학교 컴퓨터학과 석사과정

관심분야 : Dynamic Simulation, Deformable Object Simulation, AR, VR, Bio Informatics

최 유 주

e-mail : yjchoi@smit.ac.kr

1989년 이화여자대학교 전자계산학과 (이학사)

1991년 이화여자대학교 전자계산학과 (이학석사)

2005년 이화여자대학교 컴퓨터공학과 (공학박사)

1991년 (주)한국컴퓨터 기술연구소 주임연구원

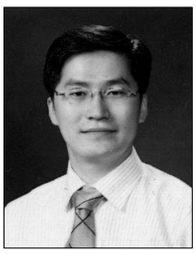
1994년 (주)포스데이타 기술연구소 주임연구원

2005년 서울벤처정보대학원 컴퓨터응용기술학과 조교수

2010년～현 재 서울미디어대학원대학교 뉴미디어학부 부교수

2015년～현 재 서울미디어대학원대학교 실감미디어연구소 교수

관심분야 : Computer Graphics, Computer Vision, HCI, Augmented Reality



홍 민

e-mail : mhong@sch.ac.kr

1995년 순천향대학교 전산학과(학사)

2001년 University of Colorado at Boulder., U.S.A., Computer Science(공학석사)

2005년 University of Colorado at Denver., U.S.A., Ph.D in Bio Informatics(공학박사)

2006년～현 재 순천향대학교 컴퓨터소프트웨어공학과 교수

관심분야 : Computer Graphics, Dynamic Simulation, Bio Informatics, Computer Vision