Journal of Digital Contents Society Vol. 13 No. 2 Jun. 2012(pp. 169-176)

3차원 콘텐츠의 효율적인 병렬 시각화를 위한 CUDA 환경 기반 객체 지향 프로그래밍 기법

박태정*

요 약

본 논문에서는 3차원 콘텐츠의 효율적인 병렬 시각화 프로그래밍을 위한 CUDA(Compute Unified Device Architecture) 환경에서의 객체 지향 플랫폼을 제안한다. 이러한 목적을 위해 GPU 프로그래밍을 위한 CUDA 환경에서의 C++ 객체 지향 프로그래밍의 특성과 제약을 논의하고 그 해결 방안을 제시하며 MVC (Model/View/Controller) 디자인 패턴에 기초한 3차원 병렬 시각화 플랫폼의 구현을 제안한다. 또한 이 MVC 디자인 패턴에 따라 적분형 MLS(iMLS)와 부호 거리장(SDF)을 이용한 3차원 모델링 기법을 Marching Cubes 및 Raytracing으로 시각화하는 예제의 구현을 논의한다. 제안하는 방법은 간단한 인터페이스의 구현만으로 GPU 병렬 처리가 자동화된다는 특징이 있으며 개발자 입장에서 객체 지향 프로그래밍의 일반적인 장점들, 즉, 코드 관리 용이성, 코드 재활용 등의 이점을 추상화와 상속을 통해 병렬 환경에서도 실현한다. 본 논문에서는 제안하는 플랫폼에 대해 두 가지 사례만 구현했으나 다양한 모델링 기법과 시각화 기법을 조합할 수 있기 때문에 컴퓨터 그래픽스 전반에서 널리 활용 가능할 것으로 기대한다.

CUDA-based Object Oriented Programming Techniques for Efficient Parallel Visualization of 3D Content

Taejung Park*

Abstract

This paper presents a parallel object-oriented programming (OOP) platform for efficient visualization of three-dimensional content in CUDA environments. For this purpose, this paper discusses the features and limitations in implementing C++ object-oriented codes using CUDA and proposes the solutions. Also, it presents how to implement a 3D parallel visualization platform based on the MVC (Model/View/Controller) design pattern. Also, it provides sample implementations for integral MLS (iMLS) and signed distance fields (SDFs) based on the Marching Cubes and Raytracing. The proposed approach enables GPU parallel processing only by implementing simple interfaces. Based on this, developers can expect general benefits that are common in general OOP techniques including abstractization and inheritance. Though I implemented only two specific samples in this paper, I expect my approach can be widely applied to general computer graphics problems.

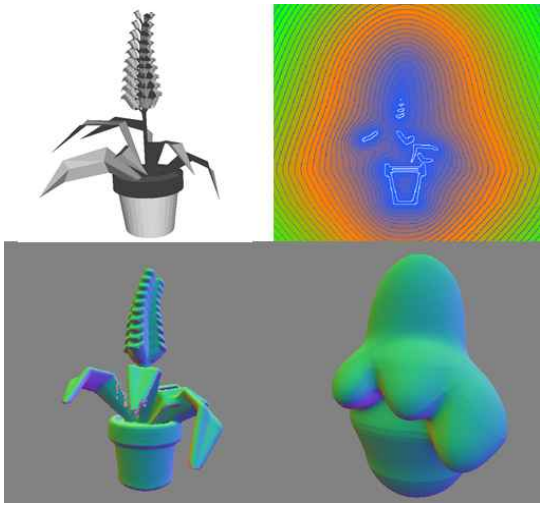
Keywords : CUDA, GPU, parallel processing, visualization platform, object-oriented programming

1. 서론

과거 컴퓨터 시스템에서 주로 그래픽 처리만을 담당하던 GPU(Graphics Processing Unit)는 하드웨어 제작 공정 및 컴퓨터 그래픽스 분야의 기술 발전을 기반으로 범용 병렬 프로그래밍

※ 제일저자(First Author) : 박태정
접수일:2012년 04월 30일, 수정일:2012년 05월 23일
완료일:2012년 06월 04일
* 연세대학교 BK 연구교수
taejung.park@gmail.com

(GPGPU, General-Purpose computing on Graphics Processing Units)이 가능한 장치로 발전해 가고 있다. 특히 NVIDIA에서 최근 발표한 Fermi 아키텍처(GTX400/500 시리즈)는 그 이전까지 불가능했던 각 멀티프로세서별 L1/L2 캐시를 하드웨어적으로 구현함으로써 병렬 프로그래밍에서 가장 해결하기 힘든 문제들 중 하나인 메모리 대역폭 문제를 크게 개선했다. 또한 이



(그림 1). 3차원 콘텐츠 시각화 및 설계 시 필요한 다양한 시각화 기법들.
(왼쪽 위) 3차원 실시간 매쉬 랜더링,
(오른쪽 위) iMLS[6]로 계산한 음함수 표면 주변의 필드 분포,
(아래쪽) iMLS에서 서로 다른 오프셋 값을 적용한 후 raycasting을 적용해서 랜더링한 모습

아키텍처에서는 그동안 불가능했던 객체 지향 프로그래밍 기법과 재귀호출, 함수 포인터 등을 지원한다. NVIDIA에서는 자사의 GPU에서 C++ 언어를 기반으로 효율적인 병렬 프로그래밍을 수행할 수 있도록 CUDA(Compute Unified Device Architecture) 프로그래밍 환경을 제공하고 있다[1].

객체 지향 프로그래밍(Objected Oriented Programming) 기법은 이미 컴퓨터 분야에서 보편적 기술로 정착된 지 오래이며 2000년대 들어서 열어 객체 지향 프로그래밍 기법들 중에서 효율성과 이식성이 뛰어난 패턴들을 정리한 디자인 패턴(Design Pattern) 기법으로 발전해 오고 있다. 특히 코드 재활용, 코딩 프로세스 관리 등 여러 측면에서 객체 지향 프로그래밍 기법을 활용하지 않고서는 설계와 코딩에서 뿐만 아니라 코드의 사용자 측면에서도 원하는 효율성과 기능을 구현하기가 쉽지 않다.

특히 (그림 1)에서 정리한 것처럼, 음함수 기법 모델링(implicit surface modeling)[2], 매쉬 기반 모델링(mesh-based modeling) 등 다양한 모델링 기법이 존재하고 동시에 광선투사(ray-casting), 광선추적(ray-tracing), 폴리곤화(polygonization), 단면 렌더링(cross-sectional rendering) 등 다양한 시각화 및 랜더링 기법이 적용될 수 있는 3차원 콘텐츠 설계 및 시각화 환경에서는 이런 다양한 객체(object)의 존재 때문에 객체 지향 프로그래밍 기법에 기초한 프레임워크가 필수적이며 다양한 연구가 수행되어 왔다[3].

이러한 3차원 콘텐츠가 2차원 콘텐츠와는 다른 특징 중 하나는 2차원 콘텐츠에 비해 3차원 콘텐츠 구성을 위해 필요한 연산 밀도가 상당히 높다는 사실이다. 따라서 실용적인 3차원 콘텐츠 설계 및 시각화 플랫폼을 구축하기 위해서는 병렬 처리 역시 필수적이다.

즉, 이상적인 3차원 콘텐츠의 설계/시각화 플랫폼은 병렬 처리 및 객체 지향 프로그래밍이 동시에 가능한 플랫폼이라고 할 수 있다. 여러 가지 유형의 조합을 생각해 볼 수 있으나 가장 저렴하면서도 성능이 탁월한 방식은 앞서 논의한 CUDA 기반 Fermi 병렬 아키텍처 상에서 객체 지향 프로그래밍 기법으로 유연성이 뛰어난 플랫폼을 구축하는 것이라고 볼 수 있다.

그러나 전 세계 여러 개발자들이 지적하고 있는 것처럼[4], CUDA에서의 객체 지향 프로그래밍은 기존 CPU 기반 객체 지향 프로그래밍 환경과는 다른 여러 특성과 문제점을 내포하고 있다. 이러한 사안들은 크게 '형식적인 측면'과 '병렬 구조상의 측면'으로 나누어 볼 수 있다.

형식적인 측면에서의 문제점은 CUDA의 C++ 객체 지향 프로그래밍은 코드 구성에서 제약이 있으며 이러한 사실은 NVIDIA의 공식 문서[5]에서도 언급되지 않았다는 사실이다. 또한 병렬 구조 측면에서는 단일 프로세서 및 단일 메모리 공간에서 실행되는 C++ 코드와는 달리 다중 스레드, 다중 메모리 레벨 상에서 실행되는 CUDA

병렬 코드의 특성으로 인해 객체가 상주하는 메모리 공간에서 제약이 있다. 따라서 CUDA 환경에서 성공적인 객체 지향 프로그래밍을 위해서는 이러한 제약을 잘 이해하고 극복할 수 있는 설계가 필요하다.

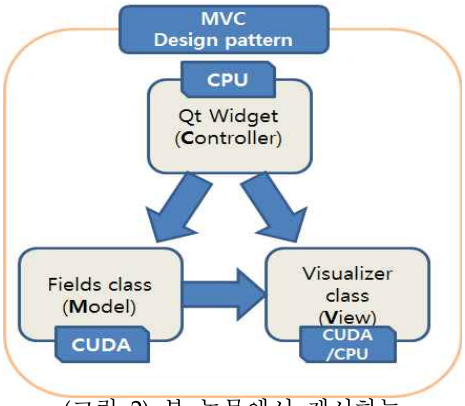
본 논문에서는 3차원 콘텐츠의 효율적인 병렬 시각화를 위해 CUDA 환경에서 객체 지향 프로그래밍을 구현하는 방법을 제안하고, 이 과정 중에서 앞서 논의한 '형식적인 측면'과 '병렬 구조상의 측면'에서의 사안들을 해결하는 효과적인 해결책을 제시한다.

1. 본론

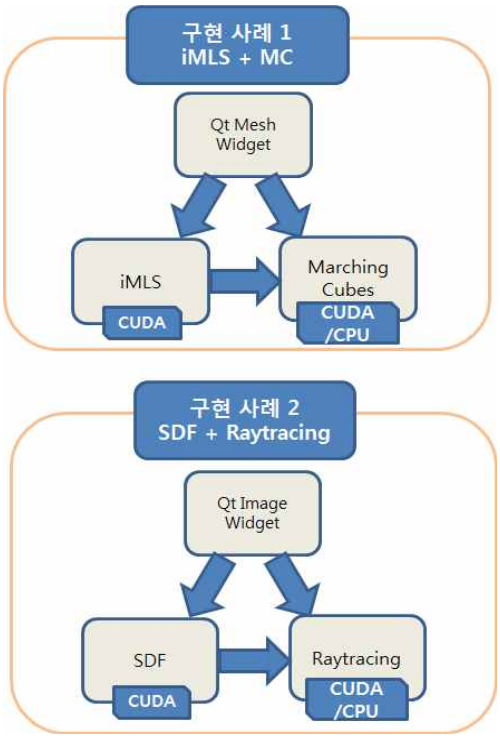
2.1 객체 지향 디자인 패턴

본 논문에서 제시하는 객체 지향 디자인 패턴은 기본적으로 MVC(Model/View/Controller)를 따른다. 이 패턴은 시각화를 직접적으로 담당하는 View와 시각화의 대상이 되는 Model, 그리고 이 두 가지 객체를 제어하는 Controller로 구성된다. 이 구성은 데이터와 시각화, 컨트롤을 각각 독립적으로 구성할 수 있으며 개념적으로도 이해가 쉽다는 장점이 있다.

그러나 본 논문에서 제안하는 MVC 디자인 패턴의 특징은 CPU 기반의 일반적인 프로그래밍의 경우와는 달리 각 구성 객체들이 서로 다른 메모리 공간에 속한 상태로 구현된다는 점이다. (그림 2)에서는 본 논문에서 제안하는 MVC 패턴을 제시한다. 본 논문에서 제안하는 MVC 패턴에서는 3차원(또는 2차원도 가능) 모델 정보를 캡슐화(encapsulate)한 Model은 순수하게 CUDA 코드로 작성되어 각각의 객체에서 상속을 받는 자식 객체들은 GPU 메모리 공간(device memory)에서 생성되는 반면, 시각화 알고리듬을 캡슐화한 View의 경우 코드는 그 특성 상 CUDA 및 CPU 코드로 작성되어 해당 부분별로 각각의 메모리 공간에 위치한다. 이에 비해 전체적인 컨트롤을 수행하는 Controller 객체의 경우, CPU 메모리 공간(host memory)에서 생성되며 NOKIA의 GUI 툴킷 라이브러리인 Qt[7]를 기반으로 위젯(Widget) 형태로 제작되었다.



(그림 2) 본 논문에서 제시하는 Model/View/Controller 기반 패턴. 각 객체는 순수 가상 클래스로 구성



(그림 3) 본 논문에서 제안하는 실제 구현 사례. (iMLS = integral Moving Least Squares 기반 음함수 곡면 모델링 기법[6], SDF = Signed Distance Fields 기반 음함수 곡면 모델링 기법[8])

(그림 2)에서 제시한 패턴은 C++에서 가상 클래스(virtual class)로 구현되며 실제 인스턴스

생성 가능한 클래스는 이 가상 클래스를 기반으로 구현된다. (그림 3)에서는 이 가상 클래스 구조에서 파생 가능한 두 가지 예제를 제시한다.

구현 사례 1의 경우 iMLS(integral Moving Least Squares) 기법[6]을 적용한 한 3차원 모델링 객체를 Marching Cubes[9]로 매쉬화를 한 후 Qt Mesh Widget으로 최종적으로 화면에 표시할 수 있는 플랫폼을 구현한다. 구현 사례 2에서는

<표 1> 가상 클래스 코드 (일부)

```
class Visualizer
{
public:
      Visualizer() {;};
      virtual void initialize() = 0;
      virtual void draw() = 0;
      virtual void compute() = 0;
};
```

```
template<class T>
class Field
{
public:
      __device__ Field() {;};
      __device__ virtual
    REAL getValue(T position) = 0;
};
```

삼각형 매쉬를 입력 받은 후 (Controller에서 입력 처리) 그 매쉬를 기반으로 부호 적용 거리장(SDF, Signed Distance Fields)을 계산하고 그 결과를 Raytracing으로 시각화하는 플랫폼에 대한 사례이다. 이 두 사례에서 SDF와 iMLS는 Model에 해당하는 Field 클래스를 상속 받은 자식 클래스이기 때문에 서로 교환해서 구성 가능하며 Marching Cube와 Raytracing의 경우도 Visualizer 클래스에서 상속 받은 자식 클래스이기 때문에 서로 교환 가능하다. 따라서 큰 코드 수정 없이 용도에 따라 다양한 방식으로 플랫폼 구성이 가능하다. <표 1>에서는 Model(Field)과 View(Visualizer)에 대한 가상 클래스 코드를 정리했다. Controller에 대한 가상 클래스의 경우 Qt 라이브러리에서 제공하는 Widget 클래스를 기본 클래스로 사용하기 때문에 이 표에서는 제외했다.

Visualizer 클래스의 경우 초기화를 담당하는 `initialize()`함수, 화면 그리기를 처리하는 `draw()` 함수, Field 클래스에 계산을 요청하는 `compute()` 함수가 모두 순수 가상 함수로 정의된다. 또한 Field 클래스에서는 실제 모델링 함수의 값을 계산하는 위치(질의점, query position)를 3차원 (x,y,z) 좌표뿐만 아니라 2차원 (x,y) 좌표로 계산할 수 있도록 template로 인자를 넘길 수 있도록 설계했다. 주목할 점은 Field 클래스는 생성자를 제외하고는 질의점을 템플릿 타입 T로 받으면 그 값을 실수형(REAL)로 반환하는 `getValue(T position)`이 유일한 순수 가상 함수로 선언되었다는 사실이다. 이렇게 함수 하나만을 인터페이스로 설정하고 있기 때문에 설계상의 간편함뿐만 아니라 향후 코드 유지보수 시의 효율성을 꾀할 수 있도록 설계했다. 이 `getValue(T position)` 함수는 유일한 순수 가상 함수이지만, 상속을 받은 자식 클래스에서 구체적인 코드가 작성되며 런타임에는 GPU 하드웨어에 따라 수 백 또는 수천 개의 스레드로 병렬로 실행된다.

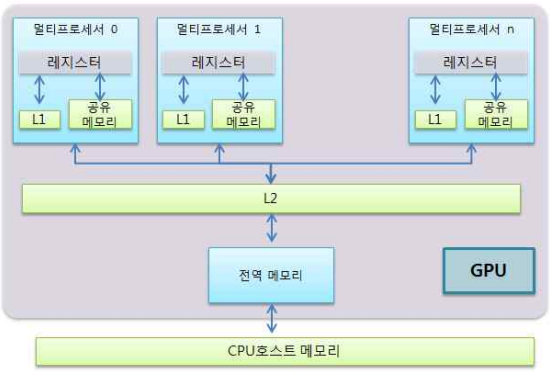
2.2 CUDA C++ 소스 레벨 차원에서의 문제 및 제안하는 해결

NVIDIA에서 개발한 CUDA는 초기에 C 기반 문법으로 GPU 프로그래밍을 수행할 수 있도록 한 개발 플랫폼이었으나, 이후 GPU 하드웨어 아키텍처의 발전을 반영하며 CUDA 4.0부터 C++ 객체 지향 프로그래밍을 지원한다. 그러나 앞서 논의한 대로 CUDA의 C++ 지원은 소스 코드 차원에서 표준적인 C++의 소스 코드 작성과는 상당 부분 차이점을 보이며 예상대로 작동을 하지 않는 경우가 흔히 발생한다(CUDA 4.1 기준).

가장 흔하게 경험하는 문제들 중 가장 치명적인 하나는, 일반적인 C++ 코드 작성 시처럼, 클래스의 선언과 구현을 각각 헤더 파일(일반적인 C++의 경우 *.h 또는 *.hpp, CUDA의 경우 *.cuh 확장자 사용 가능)과 구현 파일(*.cpp, CUDA의 경우 *.cu)로 구분할 경우 CUDA 코드 컴파일 시에는 문제가 발견되지 않으나, 작동 시 가상 클래스를 상속 받은 자식 클래스의 작동이

무시되거나 올바르지 않다는 점이다. 더구나 CUDA는 이론적으로 수 백, 수천 개의 스레드가 동시에 실행되는 구조이기 때문에 일반적인 방식으로는 원하는 대로 코드가 수행되고 있는지 파악하기조차 어렵고 따라서 이 문제의 원인조차 분석하기 쉽지 않았다.

여러 차례의 시행착오를 거친 결과, CUDA(4.1)에서는 헤더 파일과 구현 파일을 구분해서 작성하지 않고 구현 부분을 헤더에 선언된 클래스의 내부에 모두 포함시켜야지만 올바



(그림 4) CUDA 메모리 구조
(L1/L2는 각각 L1 캐시와 L2 캐시를 의미)

로 코드가 작동한다는 사실을 밝혀냈다. 이 문제는 CUDA가 이용하고 있는 LLVM(low-level virtual machine) 컴파일러의 문제 또는 CUDA로의 포팅 시 발생한 과도기적인 문제로 보이며 향후 버전 업데이트 시 해결될 것으로 기대한다.

2.3 메모리 액세스

2.1.1 CUDA 메모리 모델

(그림 4)에서는 CUDA 아키텍처(Fermi GPU 적용 시)에서의 메모리 모델을 제시하고 있다. CUDA 아키텍처는 기본적으로 GPU (device) 실행 코드 뿐만 아니라 CPU (host) 실행 코드도 동시에 생성하는 하이브리드 환경이기 때문에 프로그래밍 시에 GPU 메모리뿐만 아니라 CPU 메모리까지도 고려하는 코드 설계가 필요하다.

즉, CUDA와 같은 하이브리드 환경 속에서 new 명령을 통해 특정 클래스의 인스턴스를 힙(heap) 공간에 생성하게 되는데, 이 힙 공간이 GPU 메모리인지, CPU 메모리인지를 결정해야 하며, delete 명령에 대한 주의도 필요하다.

또한, (그림 4)에서 볼 수 있듯이, CPU 측의 메모리(host 메모리)와 GPU 메모리(device 메모리) 사이를 연결하는 버스의 대역폭이 상대적으로 좁기 때문에 최적 성능을 보장하기 위해서는 이 두 메모리 사이에서의 데이터 이동을 최소화해야 한다.

2.1.2 객체 생성 메모리 공간

객체 생성이 실제로 이루어지는 new 명령은 힙 공간에 객체를 저장할 장소를 확보한 후 그

<표 2> CUDA 클래스 인스턴스 생성

```
cudaMalloc(&d_pField,
   sizeof(Field<float3> **)));  // 1)

d_pField = new iMLS(...);       // 2)
```

1) 생성할 클래스 인스턴스의 주소를 저장할 포인터 변수를 GPU 메모리 공간에 생성
2) new를 이용해서 순수 가상 클래스 Field<float3>를 상속한 iMLS 클래스를 GPU 메모리 공간에 생성

주소를 포인터 변수로 반환한다. 일반적인 경우라면 반환 받은 주소가 지역 포인터 변수에 저장되며 이 변수를 통해서 생성된 객체를 액세스하지만, CUDA의 경우 물리적으로 메모리가 격리되어 있기 때문에 이 과정이 좀 더 복잡하며 개발자의 주의가 필요하다. 순수 가상 클래스 Field를 상속 받은 자식 클래스 iMLS를 생성하는 과정을 <표 2>에서 정리한다.

2.1.3 OpenGL 버퍼 객체와의 연동

앞서 2.3.1에서 논의한 대로 CUDA 메모리 모델은 CPU와 GPU에 각각 메모리가 할당되고 서로 간의 통신을 위해서는 비교적 대역폭이 좁은 통신 버스를 통해 데이터가 교환되기 때문에 높은 성능을 얻기 위해서는 CPU와 GPU 사이의 데이터 교환을 최소화해야 한다. 많은 병렬 처리 코드의 경우 병렬로 계산한 결과를 다시 화면에 시각적인 정보로 표현해야 하는 경우가 많은데 이 때 GPU에서 병렬로 계산한 결과를 CPU 측

의 메모리로 이동시킨 후 시각화 정보로 변환(예를 들어 수치 정보를 컬러 코드로 변환)한 후에 다시 GPU로 전송한 후 화면에 표시하는 과정을 수행한다면 불필요하게 메모리 버스를 2번 거치게 되어 전체적인 성능이 저하된다.

따라서 CUDA에서는 OpenGL과의 연동 기능을 제공해서 OpenGL에서 제공하는 버퍼 객체(Buffer Object)와 GPU 상의 메모리 공간을 일대일로 대응시킬 수 있는 기능을 제공해서 이 문제를 해결한다. 이 구조에서는 GPU가 연산 결과를 CPU로 전송하지 않고 시각화 가능한 정보로 직접 바꾼 후에 OpenGL 버퍼 객체 공간에 대응된 GPU 메모리 주소로 연산 결과를 전송한다. 이 과정이 완료되면 OpenGL은 버퍼 객체 내의 데이터를 그대로 프레임버퍼로 출력함으로써 메모리 버스를 거치지 않고 GPU 내부에서 모든 연산이 완료된다.

OpenGL 버퍼 객체와 GPU 메모리 연동에 대한 보다 자세한 정보는 [10]을 참고한다.

2.1.4 getValue 함수 구현 사례

<표 3>는 Field<float3> 클래스에서 상속받은 iMLS 클래스가 실제로 CUDA C++ 문법에 따라 구현된 코드를 정리하고 있다. 앞서 논의한 대로 getValue 함수는 Field<T> 클래스에서 요청되는 유일한 virtual 함수이며, 각 스레드마다 이 함수를 호출함으로써 실질적인 병렬 연산이 수행되기 때문에 가장 중요한 함수들 중 하나라고 할 수 있다.

<표 3>에서 특이할 만 한 점으로 먼저 2.2에서 설명한 것처럼, CUDA C++의 올바른 실행을 위해 iMLS 클래스 선언 내에 getValue 함수 전체가 정의된다는 점을 들 수 있다(CUDA 4.1 기준). 또한 getValue 함수의 반환 형식은 REAL로 정의되어 있는데 헤더 파일에 #define 문을 이용해서 필요에 따라 32비트 float 또는 64비트 double로 정밀도를 선택할 수 있도록 설계했다.getValue 함수 앞부분에는 __device__라는 CUDA 확장 명령문을 볼 수 있다. 기본적으로 __device__의 의미는 이 함수가 실행될 공간이 CPU 메모리 공간(host 영역)이 아니라 GPU 메모리 공간(device 영역)임을 의미한다. 사실 CUDA에서 생성하는 객체들은 GPU 메모리 공간에서 생성되기 때문에 CUDA 클래스 내의 모든 멤버 함수의 반환 형식은 __device__로 시

<표 3> getValue 함수 구현 샘플

```
class iMLS : public Field<float3>
{
__device__ virtual REAL getValue(float3 qp)
{
  REAL result = 0.0;
  REAL numerator = 0.0, denominator = 0.0;
  REAL distance = 0.0,abs_distance =0.0 ;
  REAL Ak = 0.0, ak = 0.0;
  REAL area = 0.0, den = 0.0, solidangle=0.0;
 bool bOnTri = false
 unsigned int i;
 float3 v0, v1, v2, normal;

for (i = 0; i < m_numFaces; i++)
{
  v0 = m_d_pV0[i];
  v1 = m_d_pV1[i];
  v2 = m_d_pV2[i];
  normal = m_d_pNormal[i];
 if(IsWithinTriangle(qp, v0, v1, v2))
 {
  result = 0.0;
  bOnTri = true;
  break
 }

 distance = Distance(qp, normal,  v0);
 abs_distance = fabs(distance);
 if(abs_distance < 1.0e-6)
 {
  area = tri_area(v0,v1,v2);
  den = calculateDen(qp,v0,v1,v2);
  Ak = (4.0 * area) / (3.0 * den);
}else
{
  solidangle =
   calculateSoildAngle(qp,v0,v1,v2);
  Ak = solidangle/distance;
}
ak = Ak * (abs_distance );
numerator += ak;
denominator += Ak;

}

if(!bOnTri)
{
  result = numerator / denominator;
}

 return result;
}

// (이하 생략)
```

작해야 한다.

이 코드에서 제시하는 알고리듬은 연결성이 없는 삼각형 폴리곤수프를 입력으로 받아서 구멍이나 연결 정보가 잘못된 표면(예. T-junction)을 매끄럽고 구멍이 없는 완벽한 음함수 표면으로 변경하는 기법들 중 하나로 일반적으로 합(sum)의 형태로 표현되는 MLS(Moving Least Squares) 기법을 연속 적분형으로 발전시킨 iMLS (integral Moving Least Squares)를 계산하는 루틴이다. 이 알고리즘과 관련한 보다 자세한 내용은 [6]을 참고한다.

2. 구현 환경 및 결과

3.1 구현 환경

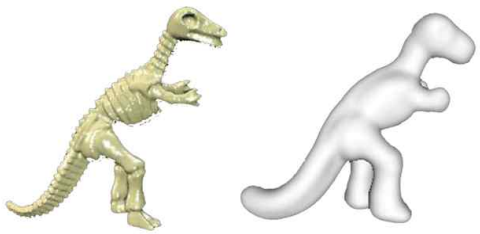
앞서 논의한 디자인 패턴 및 CUDA 객체 지향 기법의 여러 사안들을 고려해서 (그림 3)에서 제시한 두 가지 사례를 구현했다.

본 논문에서 제시하는 내용은 Visualizer와 Field의 다양한 조합을 가능하게 하고 동시에 구현과 코드 관리를 단순화하는 프레임워크의 제시를 목표로 하기 때문에 이 두 가지 사례 이외에도 다양한 조합이 가능하다. 이 구현 사례에서도 기본적으로는 <표 1>에서 제시한 가상 함수의 구현만 필요하기 때문에 기존 코드의 연결 및 구현이 매우 용이했다.

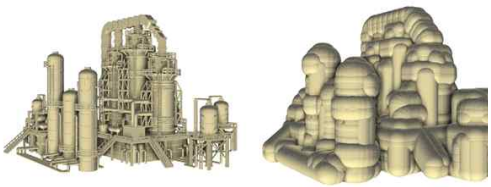
본 논문에서 제시한 내용은 모두 Intel i7 2.67GHz, RAM 12 GB, Windows 7 64비트 환경에 NVIDIA의 GTX580 GPU 상에서 CUDA 4.1 버전으로 구현되었다.

3.2 결과

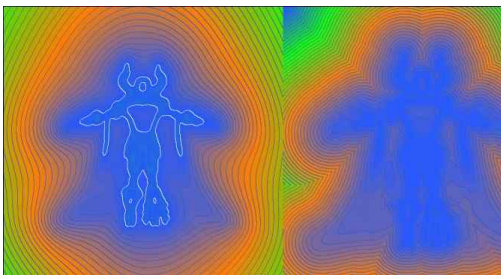
(그림 5)와 (그림 6)은 각각 (그림 2)에서 제시한 사례를 구현한 결과이다. 이 두 가지 경우 모두 getValue 함수가 수 백 개 단위의 스레드에서 병렬로 동작했다. 본 논문에서 제안하는 객체 지향 프로그래밍 기반 프레임워크의 장점들 중 하나는 추상화된 간단한 인터페이스(getValue 함수)만을 구현하면 다른 병렬 프로그래밍 관련 사항들은 거의 고려할 필요 없이 자동으로 병렬화 결과를 얻을 수 있다는 사실이다. (그림 7)에서는 이 실험에 사용된 두 가지 모델링 기법 즉, iMLS와 SDF의 차이점을 볼 수 있다. iMLS는 일반적인 MLS의 특징대로 원본(왼쪽 그림에서 흰색 실선으로 표시)의 주변을 부드럽게 연결하는 곡선으로 표시되나 이에 비해 단순히 원본에서부터의 거리를 나타내는 SDF에서는 움푹 들어간 골이 보인다.



(그림 5) 적분형 MLS(iMLS) + Marching Cube
음함수 모델링 기법 중 하나인 적분형 MLS[6]의 시각화를 위해 Marching Cube 적용



(그림 6) 거리장(SDF) 오프셋 + Raytracin
SDF의 등거리면 오프셋 적용 음함수 표면 raytracing으로 시각화한 사례



(그림 7) 적분형 MLS(iMLS, 왼쪽)과 부호
거리장(SDF, 오른쪽)의 단면 비교

3. 결론

본 논문에서는 일반적인 단일 스레드 프로그래밍 기법에서는 많은 시간이 소요되는 3차원 모델링 및 시각화를 병렬화하는 객체 지향 프로그래밍 기반 플랫폼을 제안한다. 제안하는 기법은 3차원 콘텐츠의 병렬 시각화 및 모델링과 관련된 여러 요소들을 캡슐화 함으로써 다양한 모델링 및 시각화 방식을 병렬화하는데 드는 노력을 줄이고 코드 관리와 재활용을 원활하게 수행할 수 있는 방안을 제시했다. 즉, 제안하는 CUDA 병렬 객체 지향 프레임워크에서는 개발자가 추상화된 간단한 인터페이스(getValue 함수)의 구현에만 집중하면 나머지 부분은 일반적인 객체 지향 프로그래밍 기법의 장점에서처럼 상속 및 가상 함수 오버로딩을 통해 자동으로 처리되도록 했다.

또한 객체 지향 프로그래밍 구현 시 CUDA만의 문제를 코드 수준과 메모리 공간 측면에서 살펴보았다.

결론적으로 본 논문에서 제안하는 기법은 (그림 2)에서 제시하는 두 가지 사례에 그치지 않고 컴퓨터 그래픽스 분야 전반에 적용할 수 있는 가능성이 높다고 판단한다.

참 고 문 헌

[1] Farber, R., “CUDA Application Design and Development", 1st Ed., pp.109-132, Morgan Kaufmann, 2011

[2] Geoff Wyvill, “Introduction to Implicit Surfaces”, Morgan Kaufmann, 1997

[3] 이성호, 박태정, 감형렬, 김창헌, "실수 정의역 데이터 시각화와 그 응용 사례," 한국컴퓨터그래픽스학회 논문지 Vol.16, No.4, 2010.12

[4] http://stackoverflow.com/questions/5722942/using-virtual-functions-in-cuda-kernels

[5] NVIDA CUDA Zone (http://developer.nvidia.com/category/zone/cuda-zone)

[6] Taejung Park, Sung-Ho Lee, and Chang-Hun Kim, "Analytic Solutions of Integral Moving Least Squares for Polygon Soups", IEEE Transactions on Visualization and Computer Graphics, PrePrint, 2011

[7] Nokia Qt 공식 사이트: http://qt.nokia.com

[8] Taejung Park, Sung-Ho Lee, Jong-Hyun Kim, and Chang-Hun Kim. “CUDA-based Signed Distance Field Calculation for Adaptive Grids” In Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology (CIT '10). IEEE Computer Society, 2010

[9] William E. Lorensen, Harvey E. Cline, “Marching Cubes: A high resolution 3D surface construction algorithm”, Computer Graphics, Vol. 21, No. 4, July 1987

[10] Joe Stam, “What Every CUDA Programmer Should Know About OpenGL“, NVIDIA GPU Technology Conference, 2009

박 태 정

1997년 : 서울대 전기공학부(학사)
1999년 : 서울대 전기공학부 대학원 (공학 석사)
2006년 : 서울대 전기컴퓨터공학부 대학원 (공학박사)
2006년~2012년: 고려대학교 연구교수
2012년~현 재: 연세대학교 BK 연구교수
관심분야 : 컴퓨터그래픽스, 병렬처리, 게임 물리, 수치해석, 3차원 모델링