CUDA를 이용한 포인트 렌더링 알고리즘

송성도[*], 김선정[*]

요약

본 논문에서는 CUDA를 이용한 포인트 렌더링 알고리즘을 제안한다. 포인트 렌더링 기술은 3차원 스캐닝 시스템을 통해 얻어진 점 데이터를 입력받아, 삼각 메쉬로 복원하는 과정을 거치지 않고 직접 화면에 보여줄 수 있는 기술이다. 점 데이터는 삼각 메쉬와 달리 인접 정보를 가지고 있지 않기 때문에, 이웃하는 근접점을 찾아 점과 점 사이의 빈 공간을 메울 수 있는 2차원 사각형 또는 타원형의 splat을 사용하여 렌더링 해야 한다. 이러한 포인트 렌더링의 속도를 향상시키기 위해, 본 논문에서는 NVIDIA사의 CUDA라는 프로그래밍 언어의 병렬처리 기능을 사용하여 GUP 상에서 빠르게 포인트 렌더링을 계산할 수 있도록 알고리즘을 제안한다.

Point Rendering Algorithm Using CUDA

Seong-Do Song, Sun-Jeong Kim[*]

ABSTRACT

This paper proposes a new point rendering algorithm using CUDA. We can directly render 3D point data obtained from scanning system without the reconstruction step of a triangular mesh from them. Because a point data set has no topological information like an adjacent information of a triangular mesh, we should search it for close neighbor points of each point and then render it using 2D rectangular or elliptical splats whose size can cover the hole between points. Especially we use CUDA for point rendering algorithm in order to improve computation speed through parallel processing in GPU.

Key Words : Computer Graphics, Point Rendering, CUDA, Parallel Processing, GPGPU(General-Purpose computing on GPU)

---

* 한림대학교 컴퓨터공학과
· 제1저자(First Author) : 송성도 · 교신저자(Correspondent Author) : 김선정
· 접수일(2010년 1월 5일), 수정일(1차 : 2010년 1월 25일), 게재확정일(2010년 1월 29일)

www.kci.go.kr

Ⅰ. 서 론

현재 많은 3차원 게임이나 컴퓨터 그래픽스 응용분야에서는 삼각형을 기본 도형으로 다루는 삼각 메쉬가 많이 사용된다. 하지만, 3차원 스캐닝된 대용량의 점 데이터로부터 삼각 메쉬로 모델링하기 위해서는 각 정점(Vertex)들의 위치 정보와 정점들 간의 인접 정보를 모두 저장해야 하기 때문에 메모리의 사용이 더욱 증가하게 된다.

2000년대 초반부터 연구되고 있는 포인트 렌더링(Point Rendering) 방법은 삼각형 기반의 방법과 달리, 점들 간의 인접 정보가 없이 해당 점의 위치에 직접 2차원 사각형 또는 타원 모양의 splat을 그리기 때문에 인접 정보가 필요하지 않다. 그래서 삼각형 기반의 방법에 비해 저장 공간이 적게 들고 빠르다는 장점을 갖는다. 하지만 인접 정보가 존재하지 않기 때문에 점들 사이에 빈 공간이 생기지 않도록 점의 위치에 그려지는 splat의 크기를 조정할 필요가 있다.

또한 최근 그래픽 하드웨어 분야에서는 그래픽적인 계산을 CPU가 아닌 GPU에 분담시키는 방향으로 연구가 활발히 진행되고 있다. GPU를 사용하면 CPU만 사용한 렌더링 방법에 비해 계산 속도뿐만 아니라, 렌더링 결과 품질에 대한 향상을 기대할 수 있기 때문이다.

본 논문에서는 NVIDIA에서 최근 출시한 CUDA(Compute Unified Device Architecture) 프로그램을 이용한 포인트 렌더링 알고리즘 계산을 수행한다. 특히, 포인트 렌더링을 수행하기 위해서는 첫 번째 단계로 각 점의 splat 크기를 결정해야 하는데, 이 splat의 크기는 근접하는 이웃점들과의 거리만큼 늘려야 splat 사이에 빈 공간을 제거할 수 있다. 각 점들의 근접점들을 구하기 위해서 병렬 처리가 가능한 CUDA를 사용하여 계산 속도의 향상을 목표로 한다.

본 논문의 2장에서는 기존의 포인트 렌더링 기법들을 소개하고, 3장에서는 본 논문에서 사용한 기존 기법들에 대해서 설명하며, 각 단계에서 발생한 문제점과 해결 방안에 대해서 기술한다. 마지막으로 4장에서는 결과와 향후 연구 과제를 제시한다.

Ⅱ. 관련 연구

QSplat[1]에서는 스크린 상에서 점을 타원이나 사각형, 원 모양의 splat을 이용해 포인트 렌더링이 시도되었고, 이웃 근접점들을 포함하는 구를 이용한 계층구조가 제안되었다. 구의 반지름과 구의 법선벡터 원추(normal cone)을 미리 계산하여 저장하였다. 이런 계층구조는 LOD(level-of-detail)와 후면 제거에 적합하다. 또한 중요 정보들을 양자화(quantization)된 데이터 공간을 사용하여 메모리 사용을 최소로 하였다.

다양한 모양과 법선벡터 원추의 사용으로 렌더링 시간이나 품질의 조절이 가능한 장점이 있다. 하지만 양자화를 위한 전처리 과정이 필요해서 실시간 렌더링에 적합하지 않고, 양자화 과정에서 발생되는 오류로 인해서 점들 사이에 빈 공간이 생길 수도 있다. 또한 적은 색상 값의 범위로 인해서 다양한 색상의 표현이 쉽지 않은 단점이 있다.

Surfel[2]은 QSplat과 마찬가지로 포인트 렌더링 방법들의 중심이 되는 연구이다. Surfel은 모양과 색상속성을 가지는 0차원 집합으로 구성되어 있고, 탄젠트 디스크라 부르는 각 surfel의 중심에 원을 그림을 그려서 포인트 렌더링을 수행한다. 또한 고해상도를 위해서 세 개의 LDI(Layered Depth Image)들로 구성된 LDC(Layered Depth Cube) 트리를 통해 다양한 레벨의 자료구조를 구성하고, LDC를 하나의 LDI로 줄이는 3대1 간략화(3-to-1 reduction)를 통해서 자료구조의 크기를 줄인다.

QSplat이 오브젝트 공간에 고정된 모양의 splat을 사용하는 것에 반해서 Surfel은 텍스쳐 공간에서 타원을 그리는 것과 각 픽셀의 값이 미리 필터링된 텍스쳐 생상을 가지는 것을 큰 차이점으로 볼 수 있다. 또한

www.kci.go.kr

QSplat이 계층 구조의 구를 사용해서 후면제거를 하는 것에 비해서 LDC 트리를 사용하는 것이 또 다른 차이점이다. Surfel은 타원을 사용함으로써 결과 이미지에서 QSplat에 비해서 더 부드러운 결과를 보여준다.

Surface Splatting[3]은 기존 연구들과 달리 새로운 EWA(Elliptical Weighted Average) 필터를 사용한다. 새로운 스크린 공간 EWA 필터를 사용해서 실시간 시스템을 구축한다. 이 새로운 EWA 필터는 불규칙하게 분포된 점들 간의 전역 텍스쳐 파라미터화 없이 splat의 모양과 크기를 결정할 수 있게 해준다. 또한 오브젝트 공간 기술인 가우시안 리샘플링 커널을 이용한 EWA splatting은 렌더링 품질이 뛰어나고 안티앨리어싱도 가능하다.

Deferred Splatting[4]은 멀티패스 하드웨어 가속화 EWA splatting을 사용한다. 각 단계는 Z-버퍼(깊이 버퍼), 색상 버퍼를 사용하여 GPU에서 계산을 수행한다. 또한 고수준 알고리즘과 저수준 알고리즘을 선택할 수 있도록 설계되어 있고, 동영상 같은 움직이는 물체에서의 시간적 오류 현상을 해결하기 위해서 반복적으로 단계를 수행하여 점들 사이에 빈 공간이 생기지 않도록 설계되었다.

III. 본 문

3.1 시스템 개요

첫 번째 단계에서는 Splat의 크기에 영향을 주는 각 정점의 근접점들을 찾는다. 이 단계에서 CUDA를 사용하여 정점의 개수만큼의 쓰레드를 생성한 후 병렬처리가 가능하다. 두 번째 단계에서는 찾아진 근접점들을 이용해서 빈 공간이 생기지 않도록 Splat의 크기를 결정한다.

세 번째 단계에서는 Splat의 모양을 결정하는데, 모양은 사각형부터 원, 타원 등 여러 모양이 가능하다. 또한 텍스쳐를 사용하여 Gaussian 커널을 알파 값으로 사용하는 반투명의 Splat 표현도 가능하다. 네 번째 단계에서는 Splat의 방향을 결정한다. 이 단계에서는 카메라 벡터와 정점의 위치에 따라서 방향이 결정되는데 이 부분도 CUDA를 사용해서 병렬처리가 가능하다.

마지막 단계에서는 각 정점에 그려진 Splat을 렌더링 한다. 이 단계에서 블렌딩을 위해서 Splat을 깊이 정렬을 할 필요가 있는데 이 부분도 CUDA를 이용한 병렬처리가 가능하다. 또한 Phong 조명계산을 통해서 고화질을 렌더링도 가능하다.

CPU를 이용해서 렌더링을 하면 정점 버퍼가 변경될 때마다 새로 그리기 때문에 속도가 저하되지만, CUDA를 이용해서 정점 버퍼를 관리하면 더 빠른 속도를 기대할 수 있다.

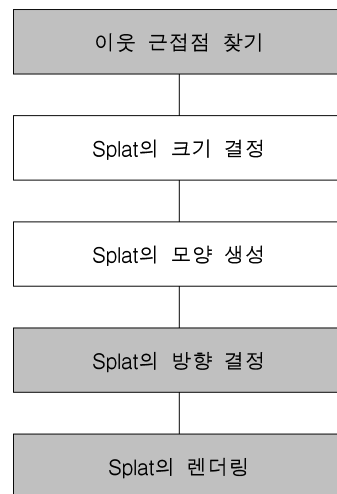


그림 1. 알고리즘 흐름도 (음영부분: CUDA 이용)
Fig. 1. Flowchart (shadow parts: using CUDA)

3.2 이웃 근접점 찾기

포인트 렌더링의 최소단위가 splat이기 때문에 렌더링 시에 splat과 splat 사이에 홀(hole)이 생길 수 있다. 그러한 홀의 생성을 피하기 위해서 splat의 크기를

www.kci.go.kr

알맞게 만들어야 한다.

점 데이터에는 삼각형 메쉬의 인접 정보가 존재하지 않기 때문에 각 정점의 근접한 이웃점들을 찾아 이웃점들 사이에 홀이 생기지 않도록 splat의 크기를 결정하는 과정은 중요하다.

한 정점의 근접점을 찾기 위해, 우선 모든 점들을 포함하는 경계 상자(bounding box)를 계산 한 후, 격자로 공간을 분할한다. 그리고 각 정점은 자신이 포함된 격자 셀과 이웃하는 26개 격자 셀에 포함된 점들과의 거리를 계산하여 k 개의 근접점을 찾아낸다. 본 논문에서는 실험적으로 k=12개의 근접점을 추출한다.

이런 반복적인 탐색을 병렬처리가 가능한 CUDA를 사용해서 처리한다. 정점의 개수만큼의 쓰레드를 생성해서 각 쓰레드에서 격자 셀에 포함된 정점들과의 비교를 한다. CUDA는 반복되는 동일한 작업의 프로그램에서 속도의 향상을 보이기 때문에 CUDA의 사용으로 인해서 속도가 향상되었다.

3.3 Splat의 크기

Splat의 크기는 정점 주변에 있는 k 개의 근접점의 위치를 고려하여 정점에서부터 근접점까지 늘려야 한다. 그림 2.(a)에서 보는 것처럼 splat의 크기로 정점과 근접점사이의 거리의 1/2을 사용한다면, 만약 정점의 위치가 규칙적이라면 홀이 발생하지 않겠지만, 정점의 위치가 불규칙하다면 splat의 사이사이에 홀이 생길 수 있다. 그러므로 splat의 크기는 정점과 근접점 사이의 거리로 지정한다. 물론 splat의 크기를 근접점까지의 거리로 설정할 경우 겹침 현상이 많아져 불필요한 렌더링을 하게 되지만, 포인트 렌더링 알고리즘에서의 가장 중요한 문제는 홀이 생기지 않도록 하는 것이기 때문에 겹침 현상은 알파 블렌딩과 같은 고급 렌더링 기술로 보완한다.

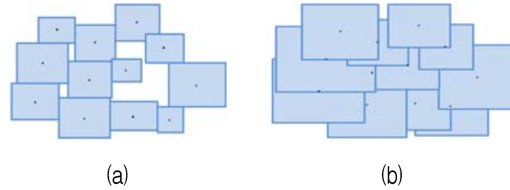
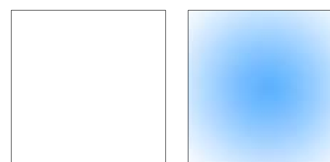


그림 2. Splat의 크기 비교
(a) 이웃 근접점들까지의 거리 1/2을 splat의 크기로 이용,
(b) 이웃 근점점들까지의 거리를 splat의 크기로 이용
Fig. 2. Comparison with the size of splats:
Using the (a) half distance and the (b) distance between the point and its neighbor points

3.4 Splat의 모양

많은 포인트 렌더링 알고리즘에서 정점의 위치에 일정한 모양의 splat을 사용한다. 본 논문에서는 splat을 위해 DirectX의 sprite를 이용하여 렌더링 한다. Sprite는 게임에서 많이 사용되는 기술로, 정점 위치에 텍스처를 입힌 2차원 사각형을 렌더링 시키는 기술로, 사각형에 대한 픽셀 단위의 크기 지정과 기하변환 행렬을 이용한 변형이 가능하다.

본 논문에서는 splat으로 사용될 사각형의 sprite에는 그림 3과 같은 텍스쳐를 사용한다. 흰색의 색상의 RGB 채널(그림 3.(a))과 Gaussian 모양의 가중치를 갖는 알파 채널(그림 3.(b))을 가짐으로써, 타원 모양의 splat이 렌더링 되고 이웃하는 splat과 자연스럽게 블렌딩 되어 부드러운 결과 이미지를 생성할 수 있다.



(a) RGB 채널 (b) 알파 채널
그림 3. Splat에 사용될 텍스쳐
Fig. 3. Texture for splat:
(a) RGB channel, (b) alpha channel

www.kci.go.kr

3.5 Splat의 방향

정점의 자리에 사각형 모양의 sprite를 이용하여 splat을 배치하면 그림 4.(a)와 같이 사각형 모양의 splat들이 놓여 실루엣 부분에 계단 현상이 발생된다.

그림 4.(a)는 50%로 축소된 크기의 splat들이 아직 알파 블렌딩 단계를 거치지 않았기 때문에 검은색으로 렌더링 된 모습이다.

Splat을 좌우로 크기 변환을 위해서는, 수식 (1)과 같이 우선 각 정점에서 카메라로 향하는 시점 벡터 $\vec{v}$를 계산하고, 카메라의 상향 벡터(up vector)와 $\vec{v}$를 외적 하여 $\vec{s}$를 구한다. 그리고 벡터 $\vec{t}$는 시점 $\vec{v}$와 $\vec{s}$를 외적 하여 구한다. 그 식은 아래와 같다.

$$\begin{aligned} \vec{v} &= Camera(c_x, c_y, c_z) - V(v_x, v_y, v_z) \\ \vec{s} &= Up(u_x, u_y, u_z) \times \vec{v} \\ \vec{t} &= \vec{v} \times \vec{s} \end{aligned} \quad (1)$$

물론, 벡터 $\vec{v}$, $\vec{s}$, $\vec{t}$는 모두 정규화 시켜 사용하고, 만약 카메라의 위치가 변경되면 벡터의 방향도 모두 변경되는 각 정점마다 가지는 카메라 종속적 벡터들이다. 이를 이용하여 splat의 크기 변환 행렬을 다음 수식 (2)와 같이 구할 수 있다.

$$\begin{bmatrix} 1 - |\vec{n} \bullet \vec{s}| & 0 \\ 0 & |\vec{n} \bullet \vec{v}| \end{bmatrix} \quad (2)$$

여기서 $\vec{n}$은 각 정점의 법선벡터이다. 이 크기변환 행렬은 sprite에 넘겨져 곱해져, 그림 4.(b)와 같이 시점 벡터에 따른 다양한 모양의 splat이 그려지게 된다. 그림 4.(b)에서도 역시 splat의 방향을 알아보기 위해 크기를 50% 축소시킨 모습이다.

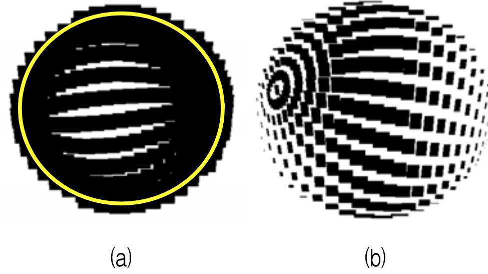


그림 4. Splat의 방향 변환 전(a)과 후(b)
(Splat의 방향을 보기 위해 크기를 50% 축소시킴)
Fig. 4. Before (a) and after (b) transformations of splats
(Decreasing the size of splats for visualization)

3.6 Splat의 렌더링

3.6.1 깊이 검사 단계

첫 번째 단계에서는 색상 버퍼를 비활성화 시켜서 깊이 버퍼에만 splat을 렌더링을 한다. 첫 번째 단계 결과 splat들이 겹치는 곳에서는 맨 앞에 놓인 splat을 알 수 있게 된다.

즉 깊이 버퍼의 깊이 값이 블렌딩 단계에서 비슷한 깊이 값을 갖는 splat들만 모아서 렌더링을 하도록 만든다. 즉 블렌딩 단계에서 다른 splat에 가려서 보이지 않는 splat들은 제거하고, 일정 값 이하의 깊이 차이 값을 가지는 splat들을 모아서 가중치 합이 1이 되도록 색을 섞어 결과 이미지의 품질을 높인다. 하지만 이런 방법은 정점의 기하학 정보를 두 번 그려야 한다는 약점이 존재하기 때문에 많은 논문에서 블렌딩을 나중에 처리하는 단일 패스 알고리즘을 제안한다[5].

3.6.2 블렌딩 단계

기존의 연구에서는 모든 splats은 뒤에서 앞으로 깊이 버퍼를 참조해서 블렌딩을 수행한다. 그래서 결과는 단일 패스 렌더링 알고리즘의 이점을 가지게 된다. 대조적으로 EWA splatting[6]은 정규화된 값을 사용

www.kci.go.kr

한다. 픽셀당 정규화하는 대신에 surfel당 정규화를 하고 각 spalt을 알파 값을 나눈다.

하지만 CPU에서 정규화 계산을 수행해서 그래픽 하드웨어로 넘기는 것은 데이터 전송 비용이 많이 들기 때문에 Botsch와 Kobbelt [7]는 우선 offscreen 버퍼에 가중치 splat을 그릴 것을 제안했다. 이 논문에서는 윈도우 크기의 단일 사각형 텍스쳐를 사용한다. 이 사각형은 프레그먼트 파이프라인을 통해서 각 픽셀에 실행되고, 마지막에 결과 값을 가중치 알파 값으로 나누어준다. 이 테크닉은 AGP 버스를 통한 픽셀 데이터 전송이 없기 때문에 픽셀 쉐이더 프로그램을 통해서 픽셀당 정규화를 수행한다.

3.7 CUDA 사용

CUDA는 NVIDIA가 내놓은 병렬처리 프로그래밍 언어로 NVIDIA의 그래픽 카드에서만 작동이 가능하고, 특히 8시리즈 이상의 그래픽카드에서만 실행이 가능하다. CUDA는 그림 5와 같이 쓰레드를 기본으로 하고 있고, 쓰레드가 모인 블록이 그 상위에, 블록이 모인 그리드가 상위에 존재한다.

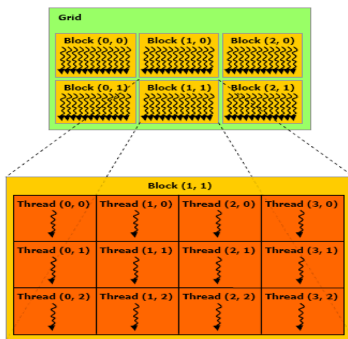


그림 5. 쓰레드 블록의 그리드
Fig. 5. The Grid of thread blocks

CUDA로 병렬프로그램을 작성하면 각 쓰레드에서 각각의 작업을 수행하게 된다.

쓰레드는 각각 자신만의 로컬 메모리를 가지고 있고, 같은 블록 안에 있는 쓰레드끼리 공유하는 공유 메모리, 그리고 모든 영역에서 사용하는 글로벌 메모리가 존재한다. CUDA는 일반적인 CPU코드 작성하는 분을 호스트(Host)라고 지칭하고, 쓰레드를 사용하는 부분은 디바이스(Device)라고 지칭한다. 그리고 호스트와 디바이스의 메모리는 서로 참조할 수 없게 되어 있다. 그래서 호스트와 디바이스 간에는 서로 메모리 복사를 통해서 메모리를 참조하고 작업을 수행하도록 구성되어 있다.

3.7.1 메모리 사용

CUDA를 사용해서 디바이스 계산을 위해서는 cudaMemcpy()를 사용해서 호스트에서 디바이스로 메모리 복사가 필요하다. 호스트에서 디바이스로 메모리 복사를 위해서는 cudaMemcpy()의 네 번째 입력값에 cudaMemcpyHostToDevice()을 사용해서 복사를 하고 호스트에서 디바이스로 복사를 위해서는 cudaMemcpyDeviceToHost()를 사용해서 복사를 한다. 디바이스에서 호스트로 복사할 때와 호스트에서 디바이스로 복사할 때 항상 같은 크기의 저장 공간이 필요하다. 입력 값과 출력 값의 저장 공간이 다를 경우에는 메모리 오버플로가 생기거나 잘못된 값이 입력되게 된다. 본 논문에서는 아래와 같은 구조체의 출력 배열을 사용하는데, 각 GridSet은 임의의 VertSet List를 가진다.

```
class VertSet {
public:
    float   x, y, z;
    int     index;                          // 버텍스 인덱스

    VertSet() { x = y = z = 0.0f; index = 0;}
};

class GridSet {
public:
    VertSet *List;
    int     ListCnt;                        // 그리드당 총 버텍스의 갯수 저장
    int     Counter;                        // 저장시에 사용될 카운터

    GridSet() { ListCnt = 0; }
};
```

그림 6. 근접점 출력 구조체
Fig. 6. The structure of output for neighbor points

www.kci.go.kr

CUDA의 Device에서는 Host의 동적 메모리 생성이 되지 않아서 초기에는 고정 값을 사용하였지만, GridSet의 List의 개수를 고정 값으로 사용할 경우 사용하지 않는 메모리의 공간이 생기게 되어서 메모리의 낭비가 생긴다. 그래서 그리드의 개수가 증가할 때 생기는 메모리 오버플로를 줄이기 위해서 List를 동적으로 관리할 필요가 생겼고, 호스트에서의 동적으로 List의 개수를 결정해서 사용하게 되었다.

3.7.2 연산의 개수

앞에서 설명한 것과 같이 CUDA 프로그램은 쓰레드를 기본으로 구성되어 있고, 그래픽카드의 성능에 따라 생성할 수 있는 쓰레드의 개수도 제한되어 있다. 그래서 병렬프로그램을 하기 위해서는 자신의 그래픽카드와 필요로 하는 쓰레드의 개수에 따라서 생성을 제한할 필요가 있다. 일반적으로 한 블록의 쓰레드의 개수는 256개 정도로 구성한다.

본 논문에서는 정점의 개수만큼의 쓰레드를 생성해서 병렬처리를 한다. 그리고 그리드가 2차원으로 구성되어 있기 때문에 정점의 제곱근을 구한 후, 1을 더한 값을 2차원 그리드의 한 변의 길이로 구성하였다. 그 다음에 그리드의 길이에 따라서 그리드 안의 블록의 개수를 결정한다. 본 논문의 경우 쓰레드의 총 개수가 정점의 개수보다 작을 경우 정점과 쓰레드가 각각 대응하지 못해서 원하는 결과가 나오지 못했다.

3.7.3 쓰레드의 제한

쓰레드는 쓰레드 인덱스의 순서대로 실행되지 않기 때문에 인덱스의 반복을 막기 위해서 인덱스의 설정을 할 필요가 있다.

```
unsigned int xIndex = blockIdx.x * BLOCKX_DIM + threadIdx.x;
unsigned int yIndex = blockIdx.y * BLOCKY_DIM + threadIdx.y;

if( (xIndex<width) && (yIndex<width) ) {
```

그림 7. 인덱스 반복 제거
Fig. 7. Removal of repetitive indices

그림 7의 코드에서처럼 그리드의 x, y축을 검색할 때 그리드안의 각 쓰레드의 개수로 제한을 걸어야 한다. 그렇지 않으면 같은 번호의 쓰레드가 반복된다. 그래서 값의 변경되게 되고, 원하는 결과 값을 얻을 수가 없기 때문에 반드시 제한을 걸어야 할 필요가 있다.

또한 디바이스에서 계산할 때 정점의 개수를 넘어서는 쓰레드 인덱스가 나올 경우를 대비해서 쓰레드 인덱스가 정점의 개수보다 클 경우 제한을 할 필요가 있다. CUDA에서는 쓰레드를 강제로 종료할 수 없다. 왜냐하면 쓰레드를 강제로 종료할 경우 다른 쓰레드도 종료될 수 있기 때문에 종료하는 기능이 존재하지 않는다. 그래서 종료를 위해서 그림 8과 같이 조건문을 사용해서 제한을 걸어야 한다.

```
if( index <= gridSize ) {
if( vertIndex < numVertices ) {
```

그림 8. 쓰레드의 제한
Fig. 8. Restriction of threads

만약 제한을 걸지 않으면 필요 없는 쓰레드가 사용되게 되고, 결국 시간의 낭비로 이어지게 된다.

3.7.4 atomicAdd() 함수

본 논문에서는 각 그리드마다 저장되는 정점의 개수를 계산하기 위해서 디바이스의 커널에서 GridSet 구조체의 ListCnt를 증가시킨다.

하지만 CUDA에서는 커널이 병렬적으로 작동하기 때문에 각 쓰레드가 ListCnt의 값을 동시에 증가시키기 때문에 증가 값의 정확성을 보장할 수 없다. 그래서 디바이스의 메모리 값을 증가시키기 위해서 atomicAdd()함수가 존재한다.

www.kci.go.kr

atomicAdd()함수는 Device상의 메모리의 int값을 증가시키는 함수로 사용을 위해서 [프로젝트] -> [속성]의 GPU Architecture의 sm_10을 sm_11로 바꾸어 주어야 한다. 이 atomic 함수는 capability 1.1이상에서만 작동한다.

Ⅳ. 결론 및 향후 연구

본 논문의 실험은 AMD Athlon64 Dual Core 2.71Ghz 프로세서와 RAM 4GB, Nvidia GeForce GTS 250에서 실험되었다. 단일 패스 알고리즘을 이용하여 splat을 블렌딩 하기 위해서는 카메라의 위치로부터 거리에 따라 splat을 정렬을 해야 한다.

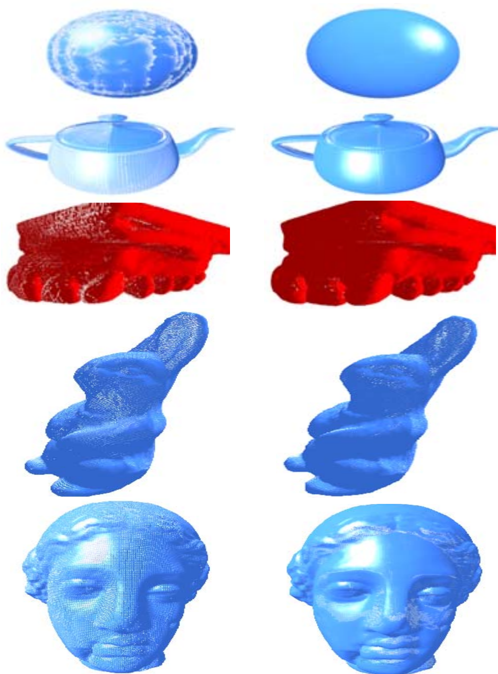


그림 9. 포인트 렌더링의 최종 결과 (우)
(좌) Splat의 정렬 생략, (우) 정렬 후 블렌딩
Fig. 9. Result of point rendering (right)
(left) without sorting, (right) with sorting

그림 9의 좌측은 카메라의 위치로부터 splat의 깊이 값을 정렬하지 않은 후 블렌딩 결과이고 우측은 정렬 후 블렌딩 결과로, 올바르게 렌더링 된 결과를 볼 수 있다.

본 논문에서는 각 정점마다 이웃하는 근접점을 찾기 위해 NVIDIA에서 제공하는 CUDA를 이용해서 GPU상에서 병렬로 처리되는 함수를 구현하였다.

표 1. CPU와 GPU를 이용한 근접점 계산 시간 비교
Table 1. Time Comparison with CPU and GPU

프로세서 \ 정점 개수	CPU			GPU		
	메모리 할당	근접점 계산	총시간 (ms)	메모리 할당	근접점 계산	총시간 (ms)
530	0	47	47	0	0	0
3,242	16	516	532	16	0	16
7,262	16	1,312	1,328	16	0	16
12,882	16	3031	3,047	16	107	123
20,102	16	5,047	5,063	31	447	478
39,342	47	13,047	13,094	31	751	782

CUDA에 의한 GPU상의 병렬처리를 이용하는 경우, 그래픽 하드웨어가 제공할 수 있는 최대 개수의 쓰레드마다 각 정점에서 수행되어야할 알고리즘을 한 번에 실행시킬 수 있기 때문에, 순차적으로 CPU에서 모든 정점에 대해 같은 알고리즘을 반복적으로 수행하는 것보다 계산속도가 빠른 것은 당연하다. 그러나 (표 1)에서 보는 것처럼 CPU와 GPU 두 방법 모두 격자를 사용하여 근접점을 찾기 때문에 메모리를 할당하는 부분에서 격자의 개수와 하나의 격자에 저장되어야 하는 정점의 개수를 결정해야 하기 때문에 시간이 많이 소요된다.

반면 GPU를 사용하는 부분에서는 근접점을 계산하는 부분보다 메모리를 할당하는 부분에서의 시간 소요가 많다. 그 이유는 메모리 할당부분에서 CPU에서와 마찬가지로 격자 관련정보들을 계산해야하고, CPU인 호스트 부분에서 GPU인 디바이스부분으로의

www.kci.go.kr

메모리 복사를 하는 부분에서의 시간이 많이 소요된다. 하지만 메모리가 할당된 후의 계산부분에서는 정점의 개수가 증가해도 계산양이 증가하지 않는 것을 볼 수 있다. 그 이유는 정점의 개수가 증가해도 계산을 병렬도 처리하기 때문에 계산시간이 증가하지 않는 것을 알 수 있다. 그러나 CUDA로 계산하는 것은 디바이스로 넘어가는 데이터의 양이 일정 수준을 넘어가게 되면 메모리 오버플로가 생겨서 계산할 수 없는 문제가 생긴다. 따라서 계산을 위해서 적당한 크기의 데이터양을 조절해야 하는 번거로움이 생긴다. 또한 디바이스(GPU)에서 구조체나 클래스의 동적 할당을 할 수 없기 때문에 호스트(CPU)에서 동적 할당을 해야 할 필요가 있다.

향후 연구로는, 현재 블렌딩 단계를 위해서 Splat을 깊이 값에 따라 정렬하는데, 카메라의 위치가 변경될 때마다 재계산을 해야 한다. 그러나 CUDA에서는 정점 버퍼를 바인딩 하여 동적으로 변경이 가능하므로, 향후 CUDA를 이용한 빠른 깊이 정렬을 위해 Splat의 정점 버퍼를 생성하여 수행한다면 렌더링 속도가 향상될 것이다. 또한 현재는 부드러운 조명 계산을 수행하지 못하고 있으나, 향후 정점과 이웃하는 근접점들 사이의 곡률을 고려한 법선 벡터를 생성한다면 고품질의 렌더링 결과를 보여 줄 수 있을 것이다.

Acknowledgement

본 연구는 2009년 정부(교육과학기술부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구입니다. (2009-00765445)

참고 문헌

[1] Rusinkiewicz S and Levoy M. *"Qsplat: a multiresolution point rendering system for large meshes"*, SIGGRAPH 2000, pp. 343-352, 2000.

[2] Pfister H, Zwicker M, van Baar J and Gross M. *"Surfels: surface elements as rendering primitives"*, SIGGRAPH 2000, pp. 335-342, 2000.

[3] Zwicker M, Pfister H, van Baar J and Gross M. *"Surface splatting"*, SIGGRAPH 2001, pp. 371-378, 2001.

[4] GUENNEBAUD, G., BARTHE, L., and PAULIN, M. *"Deferred Splatting"*, Comp. Graph. Forum Vol 23, Issue 3, pp. 653-660, 2004.

[5] Zhang, Yanci and Pajarola, Renato, *"Single-Pass Point Rendering and Transparent Shading"*, Symposium on Point-Based Graphics, 2006, pp. 37-48, 2006.

[6] Ren L, Pfister H and Zwicker M. *"Object space EWA surface splatting: a hardware accelerated approach to high quality point rendering"*, EUROGRAPHICS 2002. Comp. Graph. Forum Vol 21, Issue 3, pp. 461-470, 2002.

[7] Botsch M and Kobbelt L. *"High-quality point-based rendering on modern GPUs"*, Pacific graphics 2003. pp. 335-343, 2003.

[8] 이세열, "디지털 영상에서의 자막추출을 이용한 자막 특성 분석에 관한 연구", *한국지식정보기술학회 논문지*, 제4권 제2호, pp.37-44, 2009.

송성도(Seong-Do Song)

2008년 한림대학교 컴퓨터공학 학사
2008년~현재: 한림대학교 컴퓨터공학과 석사과정
※ 관심분야 : 컴퓨터 그래픽스, 3차원 게임

김선정(Sun-Jeong Kim)

1996년 고려대학교 컴퓨터학과 학사
1998년 고려대학교 컴퓨터학과 석사
2003년 고려대학교 컴퓨터학과 박사
2005년~현재: 한림대학교 컴퓨터공학과 부교수
※ 관심분야 : 컴퓨터 그래픽스, 3차원 게임, 가상현실

www.kci.go.kr