〈양식 2〉 **결과 보고서**_2023학년도 (1)학기 창의 개인 연구 결과 보고서

연구 횟수	1	2	3	4	5
참여 날짜	6.1	7.9	7.10	7.11	7.12

지도교사 확인
김현철(인)

OpenGL 컴퓨트 셰이더를 활용한 3차원 볼록 껍질 계산의 최적화

3304 박찬웅
지도 교사: 김현철
경기북과학고등학교

Optimization of 3D convex hull computation using OpenGL Compute Shaders(영문)

3304 Park Chan Ung
GyeonggiBuk Science High School

초록(Abstract)

본 연구에서는 GPU를 이용한 3차원 볼록 껍질 알고리즘 (Quick Hull)을 OpenGL의 Compute Shader로 구현하고, CPU 구현과의 속도를 비교하였다. Compute Shader의 3차원 볼록 껍질 알고리즘 구현은 CUDA나 OpenCL과 달리 그래픽스 파이프라인에 연동하여 사용하기 용이하다는 장점이 있어 렌더링 사이클 내에 Convex Hull 알고리즘을 삽입할 수 있을 것으로 기대하였으나, GPU의 연산이 의미 있게 동작하는 지점에서는 처리 시간이 너무 길어 렌더링 사이클 내에 삽입할 수 없었다. 따라서 추후 추가적인 최적화를 진행하고, 추가 연구를 진행할 계획이다.

주제어: Computer Graphics, OpenGL, GPGPU, Convex Hull

Ⅰ. 서론

3차원 볼록 껍질(Convex Hull)은 그래픽스에서 흔히 쓰이는 알고리즘 중 하나로, 3차원 공간 상에 퍼져 있는 점들(point cloud)이 있을 때, 이 점들 일부를 골라 다른 모든 점들을 내부에 포함할 수 있는 다면체를 구성하는 알고리즘이다.

이러한 볼록 껍질 알고리즘의 시간 복잡도는 알고리즘에 따라 다소의 차이가 있지만 최대 O(nlogn)으로, 절대적으로 복잡도가 높은 알고리즘이라고 보기는 어렵지만, 모니터나 그래픽 카드의 최대 화면 갱신 주기 (주사율, 주로 60Hz) 내에 모든 계산을 완료하는 것을 목표로 하는 실시간 렌더링에 활용 시 점이 일천 개만 넘어가더라도 볼록 껍질 계산에 의해 렌더링 루프에 병목이 걸릴 수 있다.

따라서 GPU의 범용 연산 (General-Purpose computing on Graphics Processing Unit, GPGPU)이 본격적으로 활용되기 시작한 2010년경 부터 GPU를 이용해 3차원 볼록 껍질 연산을 가속하려는 시도가 이어졌는데, Quick Hull 알고리즘의 GPU 가속이나 GPU 연산을 목표로 한 볼록 껍질 알고리즘인 gHull 등을 들 수 있다. 하지만 이러한 연구 들은 모두 Nvidia CUDA나 OpenCL 등 고부하 GPGPU 연산을 목표로 한 플랫폼 상에서 동작하므로, 볼록 껍질 알고리즘을 렌더링 파이프라인에 접목하여 사용하기는 번거롭고, 사용하더라도 별개의 플랫폼 위에서 동작하게 되므로 큰 오버헤드가 발생하여 보조 연산으로는 사용할 수 있어도, 렌더링 루프 내에서 알고리즘을 실행할 수는 없었다.

따라서 본 연구에서는 OpenGL 4.3에서 도입된 새로운 셰이더 유형인 컴퓨터 셰이더 (Compute Shader)를 이용하여 변형된 Quick Hull 알고리즘을 가속하는 방법을 제안한다.

Ⅱ. 이론적 배경

1. 볼록 껍질

1) 볼록 껍질의 정의

볼록 껍질 (Convex Hull)은 주어진 점이나 영역을 포함하는 가장 작은 볼록 집합으로, 일반적으로 그래픽스에 사용되는 2, 3차원 유클리드 공간에서는 주어진 점의 일부를 꼭짓점으로 하는 다각형 혹은 다면체로 구성된다.

2) 볼록 껍질 알고리즘

볼록 껍질을 효율적으로 계산하는 다양한 알고리즘이 여러 계산 기하학 연구에 의해 제안되었다.
2차원 볼록 껍질 알고리즘으로는 O(nh)의 선물 포장 알고리즘 (Gift Wrapping), PS(Problem Solving) 분야에서 주로 활용되는 O(nlogn)의 그레이엄 스캔 (Graham Scan), 일반적인 경우 O(nlogn), 최악의 경우 O(n^2)의 퀵 헐 (Quick Hull) 등이 있으며,
3차원 볼록 껍질 알고리즘으로는 2차원 퀵 헐 알고리즘을 확장하여 만든 3차원 퀵 헐 알고리즘과 O(nlogn)이 보장되는 클락슨-쇼어 (Clarkson-Shore) 알고리즘 등이 있다.
본 연구에서는 병렬화가 용이한 3차원 퀵 헐 알고리즘을 변형하여 연구를 진행하였다.

2. OpenGL

1) OpenGL

OpenGL은 컴퓨터 그래픽의 하드웨어 가속 처리 및 범용성 보장을 위해 1992년에 발표된 저수준 그래픽스 API의 규격이다. 주 경쟁 상대로 비교되는 Windows의 Direct3D와 달리 그 구현이 그래픽 처리 장치를 설계하는 제조사의 드라이버와 플랫폼에 대해 별개이기 때문에 Windows, GNU/Linux, Android, macOS 등 다양한 운영체제에서 사용 가능하나, 발표된지 오랜 시간이 지나 드라이버의 복잡도가 높고, 비영리 기술 컨소시엄인 크로노스 그룹이 관리 주체인 명세 특성상 각 제조사 및 플랫폼의 OpenGL 드라이버 품질 관리를 기대할 수 없어 버그가 상당히 많이 발생하며, 전역 상태 머신(Global State Machine)이라는 다소 시대에 뒤떨어진 구조를 이용하고 있어 멀티스레딩 및 버그 추적이 어렵다는 단점이 있다.
따라서 크로노스 그룹은 2016년 Vulkan이라는 차세대 그래픽스 API를 출시하였으나, 30년간 이용된 OpenGL에 대한 경로의존성과, Vulkan 프로그래밍의 높은 난도로 인해 아직 널리 이용되고 있지는 않다.

2) GLFW

OpenGL은 그래픽 출력에 대한 명세이기 때문에, 각 윈도우 시스템 상에서의 창 출력과 사용자 입력, / 출력 등의 디바이스 컨텍스트와 관련된 사양은 명시되어 있지 않다. 따라서 실제로 OpenGL을 사용할 때는 GLFW 등의 추가 라이브러리를 이용하게 된다. GLFW는 OpenGL, Vulkan 등을 위한 멀티 플랫폼 라이브러리로, Windows, macOS, X11, Wayland의 코드를 동일하게 작성할 수 있게 도와준다.

3) 셰이더

셰이더(Shader)는 그래픽 카드에 전송되어 병렬적으로 실행되는 프로그램 조각으로, OpenGL에서는 렌더링 파이프라인을 구성하는 정점 셰이더(Vertex Shader), 픽셀 셰이더(Fragment Shader), 기하 셰이더(Geometry Shader) 등과 렌더링 파이프라인과 별개로 동작하는 컴퓨트 셰이더(Compute Shader)로 나뉘어진다. 이러한 셰이더는 HLSL(High Level Shading Language), GLSL(OpenGL Shading Language) 등으로 작성되며, 프로그램 실행 시 컴파일 되어 그래픽 처리 장치에 전송된다.

4) 컴퓨트 셰이더

컴퓨트 셰이더는 OpenGL 4.3에 도입된 새로운 셰이더 유형으로, 렌더링 파이프라인과 별개로 GPGPU를 이용할 수 있도록 한다. 작업 단위에 X, Y, Z 축을 지정하여 병렬로 작업을 진행시킬 수 있고, SSBO (Shader Storage Buffer Object)를 이용하여 CPU와 자료를 주고 받는다. 사용 가능한 작업 단위의 개수와 SSBO의 크기는 그래픽 처리 장치에 따라 다르다.

5) GLM

GLM은 OpenGL과 함께 사용하기 위해 개발된 수학 라이브러리로, 벡터, 행렬 등 GLSL의 자료형과 연산을 C++에서도 이용할 수 있도록 돕는다.

Ⅲ. 연구 내용 및 방법

먼저 C++과 OpenGL을 이용하여 CPU를 통한 Convex Hull 계산을 구현한다. 전체 소스코드는 https://github.com/zlfn/ConvexGL에서 열람할 수 있고, https://github.com/zlfn/ConvexGL.git URL의 git clone을 통해서도 다운로드할 수 있다.

소스코드 별 각 함수의 간략한 설명은 아래와 같다.

include/KHR, include/glad, include/glm: glm, glad (OpenGL loader)의 소스코드이다.

include/shader.hpp: 셰이더 소스코드를 로딩하기 편하도록 만든 Shader 클래스와 ComputeShader 클래스가 존재한다. 각 클래스의 생성자에서는 GLSL 소스코드의 경로로부터 셰이더를 컴파일하여 그래픽 처리 장치에 송신하는 코드가 존재하고, 셰이더 클래스의 ID를 이용해 셰이더를 사용하도록 지시하는 use() 메소드, 셰이더의 uniform 변수에 값을 대입하는 메소드가 존재한다.

Convex/geometry.hpp: 좌표 계산을 편하게 진행 할 수 있는 Vertex, Line, Plane 클래스가 존재하며, ConvexHull을 구성하기 위한 함수들이 존재한다.

- **Vertex**: 정점을 구성하는 클래스이다. x, y, z 좌표와 각 정점을 구별하기 위한 ID, 색깔을 가진다.
- **Line**: 선을 구성하는 클래스이다. 두 Vertex와 색깔을 가진다.
- **Plane**: 면을 구성하는 클래스이다. 세 Vertex와 세 Line, 노말벡터, 색깔을 가지며, 점과 평면 사이의 거리를 계산할 수 있는 메소드를 가진다.
- **DivideOutside**: 다면체를 나타내는 Plane 벡터와 Vertex 벡터를 받아, Vertex 벡터가 다면체 안에 있는지 밖에 있는지를 계산하는 함수이다.
- **GetFurthestPoint**: 다면체 벡터와 Vertex 벡터를 받아 Vertex 벡터 중 다면체에 가장 멀리 위치한 점을 계산하는 함수이다.
- **CheckVisible**: 특정 점에서 Plane의 앞면이 보이는지 계산하는 함수이다.
- **NextPolyhedron**: 퀵헐 다면체와 가장 먼 점, 다면체 안쪽의 한 점을 받아 다음 퀵헐을 구성한다.
- **CreateSimplex**: 퀵헐 알고리즘을 시작하기 위한 사면체를 구성하는 함수이다.

Convex/draw.hpp: Vertex Array Object를 간편하게 구성하기 위한 헤더파일 이다. 쉬운 색깔 지정을 지원하는 Color Enum, 간편한 그리기를 지원하기 위한 drawVertex, drawLine, drawPlane 함수가 존재한다.

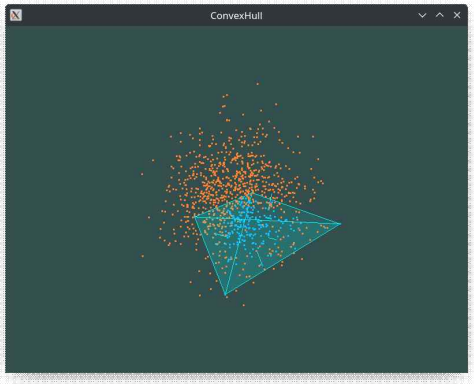
Convex/shader/vertex.vert: GLSL Vertex Shader이다. 각 정점에서 projection, view, model 행렬을 곱해 화면에 투영되는 좌표를 계산하고, 입력된 색깔을 렌더링 파이프라인을 따라 다음 셰이더(Fragment)로 넘겨준다.

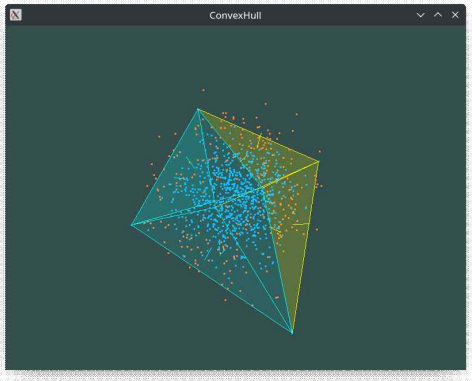
Convex/shader/fragment.frag: GLSL Fragment Shader이다. 받은 색깔을 단순히 화면에 표시하기만 한다.

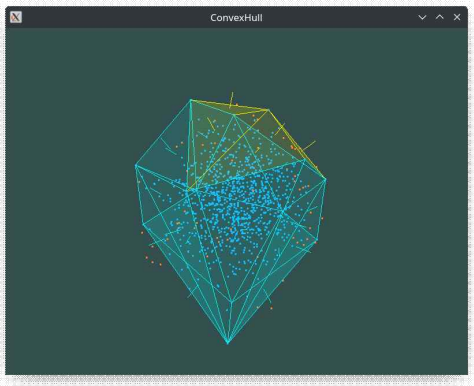
위와 같이 프로그램을 제작하면 다음과 같이 Convex Hull이 계산되게 된다.

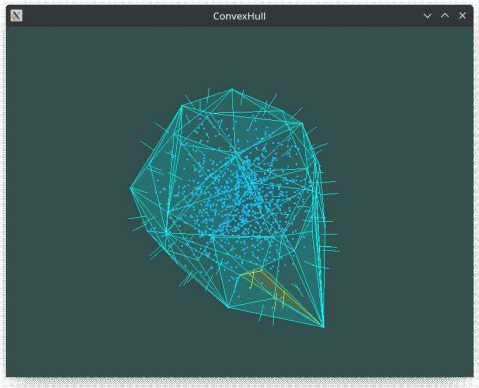
다각형 내부에 있는 점은 파란색으로, 새로 추가된 면은 노란색으로 렌더링하고, 모든 면에 대해서

법선벡터도 렌더링하였다.









성능 측정을 위해 10~100000000개의 정점에 대해 Convex Hull을 구성하는데 걸리는 시간을 측정한다. 그리고, 퀵헐을 계산하는 과정 중 DivideOutside 함수에서 모든 평면에 대해 바깥에 있는 점에 계산하는 부분과 GetFurthestPoint 함수의 모든 평면과 모든 점에 대해 거리를 계산하는 부분을 GLSL Compute Shader를 이용하여 병렬로 처리하는 코드를 작성한다.

ConvexGPU/compute.hpp: DivideOutside, GetFurthestPoint 함수를 GPU를 이용해 계산하는 함수인 DivideOugsideGPU, GetFurthestGPU가 존재한다.

PlaneE, VertexE: geometry.hpp의 Plane과 Vertex 클래스는 용량이 클뿐더러, C++ 클래스라서 그 형태로 GLSL에 대입할 수 없다. 따라서 Plane과 Vertex의 간소화 버전인 PlaneE와 VertexE 구조체를 제작하여 이 형태로 셰이더에 넘기게 된다.

getDistanceGPU: Plane 벡터와 Vertex 벡터를 받아, 모든 평면에 대해 모든 정점에 대한 거리를 병렬로 계산하는 함수이다. 이때 정점은 작업 그룹의 x 좌표에, 평면은 작업 그룹의 y 좌표에 들어가게 되는데, 그래픽 프로세서별로 작업 그룹의 x, y, z 좌표의 최댓값이 다름에 유의하여야 한다.

DivideOutsideGPU, GetFurthestPointGPU: geometry.hpp의 DivideOutside와 GetFurthestPoint 함수를 getDistanceGPU 함수를 이용하여 병렬로 처리하는 함수이다.

ConvexGPU/shader/distance.comp: GLSL Compute Shader이다. PlaneE SSBO, VertexE SSBO, 출력용 SSBO의 세 Buffer Object를 바인딩 받아 각자의 할당받은 작업 그룹의 id에 따라 계산을 진행한다.

CPU Convex Hull과 동일하게, GPU를 이용해 계산하는 Convex Hull도 정점이 10~100000000개일 때 처리에 소요되는 시간을 계산한다.

Ⅳ. 연구 결과

CPU, GPU0, GPU1의 사양과 환경은 다음과 같다.

CPU : i7-10750H, 단일 스레드 활용

GPU0 : Intel(R) UHD Graphics

작업 그룹 최대 개수 X, Y, Z 모두 2147483646

SSBO 최대 크기 128MB

GPU1 : Nvidia RTX 2070 with Max-Q Design

작업 그룹 최대 개수 X=2147483646 Y, Z=65535

SSBO 최대 크기 2047MB

Env : Arch Linux 64bit, X11, KDE

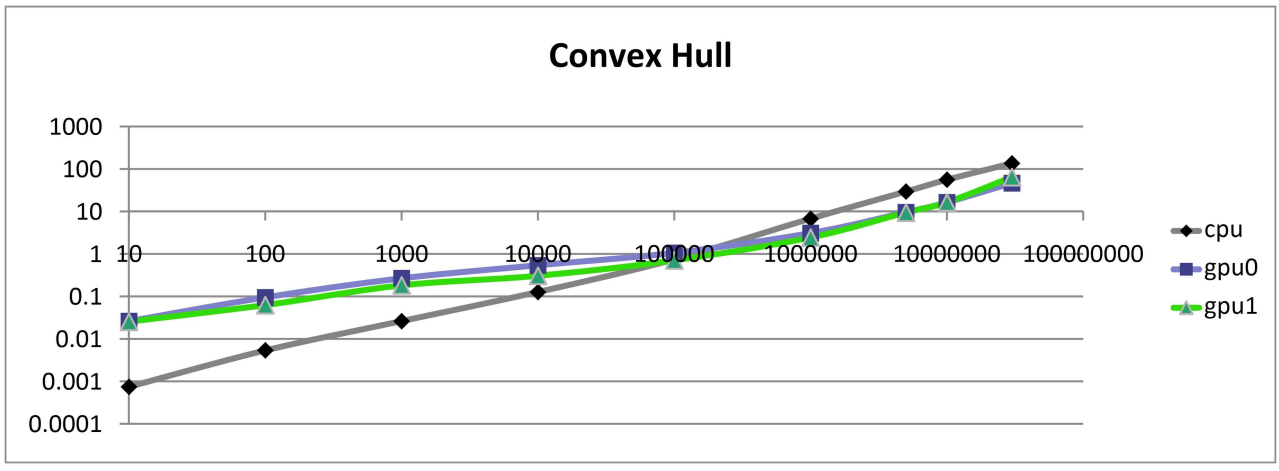
상기 장비를 사용해서 Convex Hull을 구성하는 알고리즘을 실행했을 때 걸리는 시간은 다음과 같았다.

Vertex	10	10^2	10^3	10^4	10^5	10^6	5*10^6	10^7	3*10^7	5*10^7	10^8
CPU	0.0007	0.0053	0.026	0.126	0.734	6.716	29.12	55.4	135.1	211.8	653.7
GPU0	0.026	0.094	0.264	0.534	1.048	3.062	9.448	16.17	47.10	실패	실패
GPU1	0.025	0.062	0.180	0.302	0.698	2.412	9.348	16.16	64.13	실패	실패

단위: 초

CPU는 실행 시간이 길어 점 500만 개에선 3회, 1000만 개 이상에서는 1회 측정하였고, GPU0는 3000만 개에서 3회, GPU1은 안정성 문제로 1회 측정하였다. 이외의 모든 측정값은 5회 실행 후 평균값이다.

GPU의 경우, 5000만 개 이상의 점에 대해서는 GPU0과 GPU1 모두 프로그램이 실행 도중 강제 종료되어 실패하였다. GPU 프로그래밍 특성상 에러를 포착하기 쉽지 않아 정확한 원인은 알아내지 못했으나, GPU의 그래픽 메모리 용량 초과 문제 혹은 GPU 과부하로 인한 X 서버 충돌 문제일 것으로 추정된다.



위 표의 그래프. X축과 Y축은 모두 로그 스케일이다.

Ⅴ. 결론 및 제언

GPU를 통해 Convex Hull을 계산하였을 때, 10만 개 이상의 점부터 GPU 병렬 연산의 의미가 있음을 알 수 있었다. 이는 CPU에서 GPU로 SSBO를 전달할 때의 오버헤드의 영향이 있을 것으로 추정된다. 점이 100만 개 이상일 경우 처리 시간이 1초 이상이기 때문에, 1초에 60번~300번 루프를 돌리는 것을 목표로 하는 실시간 렌더링에 활용하기는 어려울 것으로 보이며, 이 GPGPU 알고리즘을 렌더링 파이프라인과 연동하여 사용하려면 CPU의 메모리 할당 횟수를 줄이거나 한 번에 할당하는 GPU 메모리의 용량을 늘리는 방식으로 최적화를 진행하거나 멀티스레딩을 이용해 GPU 연산을 루프에 연동시키지 않는 방식으로 진행해야 할 듯하다.

추후 알고리즘을 더 최적화한 뒤, Convex Hull 알고리즘을 이용하여 분산 조명을 최적화하는 등의, 기존의 선행 GPGPU Convex Hull 연구에서는 불가능하였던 Convex Hull의 렌더링 파이프라인에의 응용과 관련한 연구를 진행할 예정이다.

■ 참고 문헌

[1] Finding Convex Hulls Using Quickhull on the GPU (Stanley Tzeng, John D. Owens)
https://arxiv.org/abs/1201.2936
[2] LearnOpenGL (Joey de Vries)
https://learnopengl.com/About
[3] OpenGL UBO and SSBO (HUAWAI, 发表于)
https://developer.huawei.com/consumer/cn/forum/topic/0204393528146900119
[4] QuickHull 3D (Jordan Smith)
http://algolist.ru/maths/geom/convhull/qhull3d.php