한국CAD/CAM학회 논문집
Transactions of the Society of CAD/CAM Engineers

Vol. 19, No. 3, pp. 294-304. September 2014
DOI http://dx.doi.org/10.7315/CADCAM.2014.294

〈응용논문〉

pISSN 1226-0606
eISSN 2288-6036

대규모 점군 및 폴리곤 모델의 GLSL 기반 실시간 렌더링 알고리즘

박상근†

한국교통대학교 기계공학과

A Real-Time Rendering Algorithm of Large-Scale Point Clouds or Polygon Meshes Using GLSL

Sangkun Park†

Department of Mechanical Engineering, Korean National University of Transportation

Received 2 July 2014; received in revised form 27 July 2014; accepted 29 July 2014

ABSTRACT

This paper presents a real-time rendering algorithm of large-scale geometric data using GLSL (OpenGL shading language). It details the VAO (vertex array object) and VBO(vertex buffer object) to be used for up-loading the large-scale point clouds and polygon meshes to a graphic video memory, and describes the shader program composed by a vertex shader and a fragment shader, which manipulates those large-scale data to be rendered by GPU. In addition, we explain the global rendering procedure that creates and runs the shader program with the VAO and VBO. Finally, a rendering performance will be measured with application examples, from which it will be demonstrated that the proposed algorithm enables a real-time rendering of large amount of geometric data, almost impossible to carry out by previous techniques.

Key Words: Fragment shader, GLSL, Large-scale point cloud, Real-time rendering, Shader program, Vertex shader, Video memory

1. 서 론

최근 컴퓨터 그래픽스 분야에서 일어나는 커다란 추세는 렌더링 시 실제 물체의 샘플링 표현 방식(sampled representations)을 사용하여 특별한 조작 없이 그대로 그래픽 처리하는 방향으로 그 관심이 이동하고 있다. 이러한 추세를 보여주는 대표적인 사례로, 실제 물체로부터 3차원 샘플 모델을 추출해내는 3D 스캐닝 시스템의 사용이 날로 증가하고 있다. 그러나 현재 이 3D 스캐닝 시스템의 큰 문제점 중의 하나가 3D 스캐닝 측정을 통해 획득한 데이터의 개수가 상당히 대규모이고, 이것을 신속하게 처리해야 한다는 것이다. 지난 수년 동안, 3D 스캐닝 시스템 관련 하드웨어 및 소프트웨어의 눈부신 발전에 힘입어, 측정된 폴리곤 모델(polygonal model)의 수용 가능한 크기가 약 1억 개의 샘플 점까지 증가하였다. 그러나 현재의 워크스테이션 수준은 실시간으로 이러한 크기의 폴리곤 모델을 화면상에 렌더링 하지 못하며, 메시 간략화(mesh simplification) 등의 기능 수행에 필요한 계산량과 메모리 요구량이 수백만 개 이상의 점들로 이루어진 점군(point cloud)[1] 모델의 경우엔 거의 비현실적인 상황이 되고 만다. 이러한 문

†Corresponding Author, skpark@ut.ac.kr

제를 해결하기 위해 대규모 형상 데이터를 하나하나 정확히 다루지 않고, 그 대신 옥트리(octree) 등의 계층 구조를 사용하여 원래 점군의 밀도를 감소시키는 방식으로, 측정 데이터를 간결하고 효율적으로 표현하는 자료구조를 제안하는 몇몇 렌더링 시스템이 발표되었다. 대표적인 예로 Krishnamurthy와 Levoy의 spline-fitting system[2], Curless와 Levoy의 range image merging system[3], Yemez와 Schmitt의 옥트리 입자기반 렌더링 시스템[4] 등이 있다.

이러한 시스템은 비록 거대 데이터의 렌더링 효율성 증대에 큰 기여를 다하였다고 말할 수 있으나 대규모 데이터에서 비롯된 근본적인 한계점을 해결한 것은 결코 아니었다. 본 연구에서 제시하는 렌더링 방식은 이전의 다수 연구에서 보여준 데이터 표현 구조의 효율성 차원이 아닌, 전혀 새로운 GPU-가속화 렌더링 방법에 관한 것이다. GLSL 기반 최신 그래픽 기술을 사용하고, 병렬 처리 능력을 갖은 GPU와 빠른 메모리 접근 능력을 갖은 비디오 메모리를 활용하는 하드웨어 기반 가속화 렌더링 기법을 소개하고자 한다.

2. GLSL 렌더링

2.1 GLSL 소개

GLSL(OpenGL shading language)[5]은 OpenGL[6] API(application programming interface)의 필수적인 핵심 요소로서 OpenGL을 사용하여 작성되는 모든 그래픽 프로그램은 향후 하나 이상의 GLSL program을 사용하게 될 것이다. GLSL로 작성된 프로그램을 쉐이더 프로그램(shader program) 혹은 간략히 쉐이더(shader)라 부른다. 쉐이더 프로그램은 그 이름이 의미하는 바와 같이 3차원 그래픽 이미지의 조명(lighting) 및 쉐이딩(shading) 효과와 관련된 알고리즘을 구현하기 위해 작성되는 그래픽 프로그램이다. 그러나 최근에 쉐이딩 알고리즘의 구현을 포함하여 그 이상의 많은 작업을 위해 사용된다. 예를 들어 애니메이션, 테써레이션(tessellation), 심지어 일반적인 계산용으로도 사용되고 있다.

쉐이더 프로그램은 GPU 상에서 직접 수행되며 대개 병렬 방식에 의해 동시 작업이 일어난다. 그래픽 카드의 프로세서 개수(예: NVIDIA GeForce GTX 480은 쉐이더 프로세서가 480개임)는 이러한 동시 작업의 처리 능력을 나타내는 지표로 사용될 수 있다. 이러한 병렬 처리 특징으로 최근에 쉐이더 프로그램이 대용량 데이터의 병렬 계산용으로도 점차 주목을 받고 있다.

쉐이더 프로그램은 고정 파이프라인(fixed-function pipeline)로 언급되는 OpenGL 아키텍처의 일부를 교체하려는 의도를 가지고 설계되었다. 조명 및 쉐이딩에 관한 디폴트 알고리즘이 바로 고정 파이프라인의 핵심 부분인데, 과거 프로그래머는 보다 실감하는 고급의 특수 효과를 구현하고자 할 때 고정 파이프라인에 강제적으로 유연성을 주기 위해 여러 형태의 기법 등을 고안하여 사용하였다. 그러나 GLSL의 출현으로 이 고정 파이프라인은 GLSL로 작성된 쉐이더 프로그램으로 교체되면서, 상당한 수준의 프로그램 유연성(일반 사용자도 쉽게 프로그램화할 수 있는 programmable graphics pipeline임을 의미)과, 보다 신속한 그래픽 처리 능력(여러 vertices 혹은 fragments을 한꺼번에 동시에 처리하는 병렬처리 장치를 제공)을 가지게 되었다.

사실 최근 OpenGL 버전은 이러한 능력을 지원할 뿐만 아니라 모든 OpenGL 프로그램이 핵심 부분으로서 쉐이더 프로그램의 사용을 요구하고 있다. 즉 새로운 파이프라인(이것의 핵심 부분은 GLSL로 작성된 쉐이더 프로그램임)에 의해 과거의 고정 파이프라인이 점차 중요도가 떨어지면서 사라지게 될 대상이 된 것이다. 대표적인 예로 glBegin( ) 및 glEnd( ) 함수가 OpenGL 3.0에서 삭제 예정으로 소개되었고 마침내 3.1에서 삭제되었다. 그러나 과거 버전에서 사용되었던 함수들과의 호환성 유지를 위해 compatibility profile[7] 개념이 OpenGL 3.2에서 소개되었고, 더불어 삭제 예정인 함수 및 특징들을 모두 없앤 core profile[8]이 사용자의 선택에 의해 사용되고 있다. GLSL는 OpenGL의 탄생 이후 처음으로 진행되는 중대한 수정 사항으로서 다음과 같은 특징들을 가지고 있다[5].

- □ 다중 플랫폼을 지원한다. 즉 여러 운영 체제 상에서 호환성을 갖는다.
- □ GLSL를 지원하는 그래픽 카드라면 그 카드의 종류와 하드웨어 제작 사에 무관하게 하나의 쉐이더 프로그램을 작성하여 어디든지 사용할 수 있다.
- □ 하드웨어 제작 회사들은 그들의 드라이버에

GLSL 컴파일러를 포함하고 있어, 사용자가 작성한 쉐이더 프로그램을 입력 받아 그들의 특별한 그래픽 카드 아키텍처에 가장 최적화된 코드를 생성해준다.

2.2 GLSL Shader

GLSL shader[5]는 대개 text 파일 형태로 작성된 후 string의 형태로 그래픽 드라이버에 전달되어 컴파일 되며, 다른 특징의 쉐이더와 함께 하나의 쉐이더 프로그램 내에서 링크된다. 현재 총 4개의 쉐이더가 사용되고 있는데, 필수 쉐이더로서 vertex shader와 fragment shader가 있고, 이밖에 선택 가능한 쉐이더로서 geometry shader와 tessellation shader가 있다. 본 연구에서 사용된 vertex shader와 fragment shader의 특징을 간략히 살펴보면 다음과 같다.

○ VERTEX SHADER

Vertex shader는 전체 렌더링 파이프라인 안에서 가장 처음으로 수행되는 필수 쉐이더로서 vertex array buffer에서 1개씩의 vertex를 입력받아 계획된 프로세싱을 적용한 후 그 결과 변화된 vertex를 출력한다. 여기서 계획된 프로세싱이란 일반적으로 입력된 vertex에 모델뷰(modelview) 행렬을 곱하고 다시 투영(projection) 행렬을 곱하는 것을 말한다.

Fig. 1의 GLSL shader는 점군 렌더링을 위해 본 연구에서 작성한 vertex shader이다. location 0을 통해 vertex의 위치 데이터(VertexPosition)가 입력되고, location 1을 통해 vertex의 색상 정보(VertexColor)가 입력된다. 입력된 정보를 vertex attributes라 부른다. 또한 uniform variables로서 ModelViewMatrix 행렬과 ProjectionMatrix 행렬이 입력된다. 입력된 VertexPosition은 ModelView Matrix와 ProjectionMatrix에 의해 변환되어 출력되고, 입력된 VertexColor는 vertex_color란 이름으로 그대로 출력된다. 여기서 vertex attributes와 uniform variables의 의미를 살펴보면 다음과 같다.

```
#version 400

layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexColor;

uniform mat4 ModelViewMatrix;
uniform mat4 ProjectionMatrix;

out vec3 vertex_color;

void main(void)
{
   gl_Position = ProjectionMatrix *
        ModelViewMatrix * vec4(VertexPosition, 1.0);
   vertex_color = VertexColor;
}
```

Fig. 1 Vertex shader used in this work

• Uniform Variables

쉐이더 프로그램의 전역 변수로서 마치 상수처럼 사용되며 프로그램 내부에서 수정되거나 어떤 함수의 입출력 파라미터로서 사용되지 않는다. 그 값은 외부에서 설정하며 이를 위해 uniform location을 갖는다. 이 때 두 개의 uniform을 같은 location으로 설정할 수 없다. 주로 조명 파라미터(예를 들어, 조명 위치 및 방향), 변환 행렬(예를 들어, 모델뷰 행렬, 투영행렬), 텍스처 등의 데이터를 보관하기 위해 사용된다.

• Vertex Attributes

Vertex shader 내에서 사용되는 비전역 변수로서 uniform variables와는 다르게 각 vertex마다 속성에 해당하는 값들이 설정되는데, vertex의 위치, 색깔 및 법선 벡터 등은 내부에서 미리 정의된 속성들(built-in vertex attributes)이고 이밖에 사용자 자신의 속성들(custom vertex attributes)을 설정할 수 있다. 한편 이러한 속성들은 fragment shader에서는 정의될 수 없다.

○ FRAGMENT SHADER

Fragment shader는 가장 마지막으로 수행되는 필수 쉐이더로서 geometric primitive를 래스터화한 후 primitive가 차지하는 각 픽셀에 대해 픽셀 크기의 fragment들을 생성한다. 각 fragment는 관련된 픽셀의 위치와 깊이(depth) 값을 가지고 있으며 또한 색깔, 텍스처 좌표값 등의 보간(interpolated) 파라미터들을 가지고 있다. primitive를 구성하는 vertex를 변환한 이후 이들로부터 보간 파라미터들은 유도된다. 개념적으로 fragment를 potential pixel로 간주하여 볼 수 있다. 만약 fragment가 여러 래스터화(rasterization) 테스트를 통과한다면 그 fragment는 frame buffer 안의 픽셀을 업데이트하게 된다.

```
#version 400

in vec3 vertex_color;

layout (location = 0) out vec4 fragColor;

void main(void)
{
    fragColor = vec4( vertex_color, 1.0 );
}
```

Fig. 2 Fragment shader used in this work

Fig. 2의 GLSL shader는 점군 렌더링을 위해 본 연구에서 작성한 fragment shader이다. vertex shader에서 출력된 vertex_color가 fragment shader로 입력된다. 그리고 특별한 변경 없이 그대로 fragColor로 출력된다.

ㅇ SHADER PROGRAM

쉐이더 프로그램의 생성 과정은 소스파일로부터 실행파일을 만드는 과정과 매우 유사하다. 즉 소스파일에 해당하는 vertex shader와 fragment shader를 컴파일(compile)하고, 이들을 쉐이더 프로그램 안에 붙인 후(attach) 연결(link)하면 생성이 종료된다. 이후에 필요시 쉐이더 프로그램 안에 drawing 함수를 삽입하여 사용하면 된다. Fig. 3은 이러한 일련의 과정을 서술한 것이다. gl***() 함수는 각 단계에서 호출되는 OpenGL API 함수를 가리킨다.

▶ Generation of Vertex & Fragment Shaders
1) Create a vertex and fragment shader.
   (glCreateShader(...))
2) Set the source codes in each shader.
   (glShaderSource(...))
3) Compile the source code string stored in each shader.
   (glCompileShader(...))

▶ Generation of Shader Program
1) Create a shader program.
   (glCreateProgram())
2) Attach the complied shaders to the shader program.
   (glAttachShader(...))
3) Link the shader program.
   (glLinkProgram(...))

▶ Execution of Shader Program
1) Install the shader program.
   (glUseProgram(...))
2) Specify the values of uniform variables or attribute variables if applicable.
3) Draw geometric objects.
   (glDrawArrays(...))
4) Uninstall the shader program.
   (glUseProgram(0))

Fig. 3 Shader program used in this work

2.3 VAO와 VBO

고정 파이프라인이 점차 사라짐에 따라 glEnable(...), glVertex(...), glColor(...) 등을 더 이상 사용할 수 없게 되었다. 이것은 형상을 렌더링하기 위해서는 그래픽 카드에 데이터를 전송할 새로운 방식이 필요함을 의미한다. 이를 위해 VBO(vertex buffer object)[9]를 사용할 수도 있고 또는 OpenGL 3+의 새로운 특징인 VAO(vertex array object)[9]를 사용할 수도 있다.

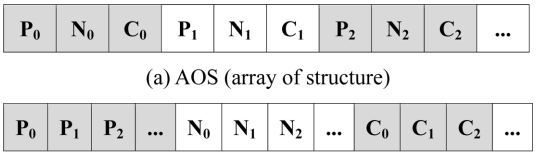
VBO는 vertex attributes(예: 위치, 법선벡터, 색깔 등)를 저장하기 위한 buffer 객체를 말하고, VAO는 여러 개의 VBO를 관리하는 객체를 의미한다. 예를 들어 vertex의 위치 데이터를 하나의 VBO에 저장하고, vertex의 색깔 데이터를 다른 VBO에 저장할 수 있으며, 이 2개의 VBO를 1개의 VAO 안에 존재하도록 설정할 수 있다. 이밖에 vertex의 법선 벡터 또는 vertex당 필요로 하는 임의의 데이터를 VBO에 저장하여 같은 VAO 안에 존재하도록 설정할 수 있다. 즉 VAO는 VBO 객체의 상태에 관한 정보를 저장 및 관리하는 방식을 기술하고, VBO는 정점 데이터(vertex data)를 저장하거나 필요시 쉐이더 프로그램에 보내는 객체로 비교하여 설명할 수 있다. VAO와 VBO의 자세한 생성 과정을 살펴보면 2.4절의 MakeVertexAttributes 알고리즘과 같다.

한편, VAO와 VBO를 생성하기 전에 형상을 정의하는 정점 데이터의 준비가 필요하고 이를 비디오 메모리에 전달하기 위해서는 사전에 설계된 형식의 정점 스트림(vertex stream)이 필요하다. 예를 들어 Fig. 4은 3차원 (x, y, z) 좌표계에서 vertex 6개의 스트림 형식을 보여주고 있다.

본 연구에서는 다음과 같은 두 가지 형식(format)의 정점 스트림을 제안한다. vertex가 위치(position), 법선(normal), 색상(color)으로 이루어져 있을 때,

x_0	y_0	z_0	x_1	y_1	z_1	x_2	y_2	z_2	...	x_5	y_5	z_5

Fig. 4 An example of vertex stream representing 3D position data of six vertices



(a) AOS (array of structure)

(b) SOA (structure of arrays)

Fig. 5 Examples of a vertex stream

하나는 Fig. 5(a)의 AOS(array of structure) 형식이고, 다른 하나는 Fig. 5(b)의 SOA(structure of arrays) 형식이다. 여기서 $\mathbf{P}_i$는 *i*-번째 정점의 위치(vertex position), $\mathbf{N}_i$는 *i*-번째 정점의 법선(vertex normal), $\mathbf{C}_i$는 *i*-번째 정점의 색상(vertex color)을 의미한다.

2.3 전체 알고리즘

본 연구에서 제안하는 렌더링 방법은 다음과 같이 2개의 렌더링 준비 과정과 1개의 렌더링 수행 과정으로 구성된다. 각 과정에서 수행되는 주요 내용을 간략히 요약하면 다음과 같다.

○ **렌더링 준비 과정**

- vertex shader와 fragment shader로 구성된 쉐이더 프로그램을 생성한다.
- 렌더링 하려는 물체의 형상을 정의하는 정점 스트림을 VAO와 VBO를 사용하여 GPU 비디오 메모리에 저장한다.

○ **렌더링 수행 과정**

- 렌더링 필요 시, 준비된 쉐이더 프로그램이 비디오 메모리에 저장된 정점 스트림을 사용하여 화면 상에 이미지를 출력한다.

위와 3개 렌더링 과정의 상세한 알고리즘은 각각 (1) SetupScene, (2) MakeVertexAttributes, (3) RenderScene에 기술되어 있다. 각 알고리즘을 살펴보면 다음과 같다.

(1) SetupScene 알고리즘

Algorithm SetupScene

Step 1: Initialize the rendering state
1) Set background color. (glClearColor(...))
2) Set point size, line width, etc. (glPointSize(...))

Step 2: Initialize shader program.
1) Create a vertex and fragment shader. (glCreateShader(...))
2) Set the source codes in each shader. (glShaderSource(...))
3) Compile the source code string stored in each shader. (glCompileShader(...))
4) Create a shader program. (glCreateProgram(...))
5) Attach the complied shaders to the shader program. (glAttachShader(...))
6) Link the shader program. (glLinkProgram(...))

Step 3: Initialize uniform variables
1) Get the location of specific uniform variables (in this work, modelview and projection matrices) within the shader program for future use. (glGetUniformLocation(...))

Step 4: Initialize a camera
1) Compute the bounding box of objects to be rendered.
2) Determine eye position, center of interest, view-up vector from the bounding box.

□ Step 1: 렌더링 준비를 위한 OpenGL 상태를 초기화한다. 예를 들어 배경색, 점크기, 선두께 등을 초기화한다.

□ Step 2: 쉐이더 프로그램을 초기화 한다.

vertex shader와 fragment shader를 생성하고 각 쉐이더에 해당하는 소스코드를 설정한다. 소스코드는 Fig. 1, 2와 같다. 그리고 이어서 각 쉐이더에 저장된 소스코드를 컴파일한다. 다음은 쉐이더 프로그램을 생성하고 그 안에 앞에서 컴파일한 vertex shader와 fragment shader를 첨부한다. 마지막으로 이들 쉐이더들을 연결하여 쉐이더 프로그램 초기화 작업을 완성한다.

□ Step 3: Uniform variables을 초기화 한다.

쉐이더 프로그램 내에 정의된 모델뷰 행렬과 투영 행렬에 추후 렌더링 수행을 위한 데이터 전송하기 위해 인터페이스 역할을 할 location을 저장 관리한다.

□ Step 4: 카메라 객체를 초기화 한다.

렌더링 대상 물체를 감싸는 박스를 먼저 구한 후, 이 박스로부터 카메라 눈의 위치, 대상 물체의 원점, 업-벡터(up-vector) 등을 설정한다. 이 카메라 객체는 OpenGL 객체가 아니다. 뷰잉(viewing)

과 투영(projection)을 수행하고 동적 뷰(dynamic view)를 구현하기 위해 본 연구에서 개발한 객체이다.

(2) MakeVertexAttributes 알고리즘

Algorithm MakeVertexAttributes
Step 1: Enable a vertex array object. 1) Create a new VAO and get the VAO id. (glGenVertexArrays(...)) 2) Make the new VAO active. (glBindVertexArray(...))
Step 2: Define a buffer object for memory storage. 1) Create a new VBO and get the VBO id. (glGenBuffers(...)) 2) Make the new VBO active. (glBindBuffer(...)) 3) Upload vertex data to the video device. (glBufferData(...)) 4) Specify the attribute index which will get its vertex data from the active VBO and the stream format of vertex data. (glVertexAttribPointer(...)) 5) Enable attribute index as being used. (glEnableVertexAttribArray(...))
Step 3: Disable the vertex array object. Make the VAO inactive. (glBindVertexArray(0))

□ Step 1: VAO(vertex array object)를 활성화한다.

VAO를 생성한다. 이때 VAO를 가리키는 id도 함께 생성된다. 그리고 id에 해당하는 VAO를 활성화한다. OpenGL에서 어떤 VAO 혹은 VBO를 사용하려면 먼저 그것을 활성화 상태로 만들어줘야 한다. 만약 최초의 활성화라면 필요 메모리가 할당되고 디폴트 상태로 활성화 된다.

□ Step 2: VBO(vertex buffer object)를 구성한다.

VBO를 생성한다. 이때 VBO를 가리키는 id도 함께 생성된다. 그리고 id에 해당하는 VBO를 활성화한다. 다음은 정점 데이터를 VBO에 업로드한다. 그리고 쉐이더 프로그램에서 업로드된 정점 데이터를 읽어 들일 속성 색인(attribute index)를 설정하고, 전송될 정점 데이터의 형식(format)을 설정한다. 마지막으로 그 속성 색인이 작동할 수 있도록 활성화 상태를 만들어준다.

□ Step 3: VAO를 비활성화 한다.

(3) RenderScene 알고리즘

Algorithm RenderScene
Step 1: Set viewport (glViewport(...))
Step 2: Update the camera. 1) Update the projection matrix. (glm::ortho(...)) 2) Update the modelview matrix. (glm::lookAt(...))
Step 3: Run the shader program. 1) Install the shader program. (glUseProgram(...)) 2) Specify the values of uniform variables (in this work, modelview and projection matrices) for the shader program. (glUniformMatrix4fv(...)) 3) Draw geometric objects. (glDrawArrays(...)) 4) Uninstall the shader program. (glUseProgram(0))

□ Step 1: viewport를 설정한다.

□ Step 2: 카메라를 업데이트 한다.

렌더링 뷰의 변경을 나타내는 모델뷰 행렬과 투영 행렬을 업데이트한다. 이 행렬들은 step 3에서 쉐이더 프로그램에게 전달된다. 참고로 glm::ortho(...)와 glm::lookAt(...)는 GLM(OpenGL Mathematics)[10]에서 제공하는 뷰 관련 함수로서 각각 투영 행렬과 모델뷰 행렬을 리턴한다. GLM은 그래픽 소프트웨어 개발을 위해 만든 GLSL 기반의 C++ 수학 라이브러리이다.

□ Step 3: 쉐이더 프로그램을 실행한다.

특정 쉐이더 프로그램을 선택하고, 선택된 쉐이더 프로그램에게 step 2에서 업데이트한 모델뷰 행렬과 투영 행렬을 전달한다. 쉐이더 프로그램 내부에서 이 두 개의 행렬을 전달받는 변수가 uniform variables이다. 다음은 OpenGL의 drawing 함수를 호출하고, 마지막으로 쉐이더 프로그램을 종료시킨다.

본 외부에서 쉐이더 프로그램으로 입력되는 데이터의 전달 과정을 살펴보면 다음과 같다. 시스템 메모리의 정점 데이터, 즉 정점 위치와 정점 색상은 비디오 메모리인 VBO에 저장되고, 각각은 쉐이더 프로그램에서 location 0의 VertexPosition 변수와 location 1의 VertexColor 변수로 전달된다. 또한 시스템 메모리에 저장되었던 모델뷰 행렬과 투영 행렬은 쉐이더 프로그램에서 각각 uniform variables인 ModelViewMatrix 변수와 Projection Matrix 변수로 전달된다.

3. 적용예제 및 토의

본 연구에서 제시한 GLSL 기반 렌더링 방법의 성능을 확인하기 위해 LAS[11] 파일과 STL[12] 파일을 읽어오는 기능을 구현하였다. LAS[11] 파일은 LiDAR sensors로부터 획득한 점군 데이터를 교환하거나 저장하기 위해 ASPRS에서 개발한 공개형 파일 형식이다. LiDAR(Light Detection and Ranging)는 고해상도 지도를 만들기 위해 사용되는 일반적인 측정 기술로서 지리정보학, 지리학, 지형학, 지진학, 임학(삼림관리), 대기물리학 등의 분야에서 주로 사용되며, 인공위성에 의한 원격탐사 및 등고선 지도 제작 등에 많이 사용된다. STL[12] 파일은 3D Systems사에서 SLA(Stereolithography) 제품을 만들기 위해 개발된 파일 형식인데, 여러 종류의 소프트웨어에서 지원함에 따라 3차원 물체의 곡면 정보를 삼각 메시 형태로 저장하는 가장 일반적인 파일 형식이 되어, CAD/CAM 분야를 비롯하여 3D 스캐닝 및 첨삭(additive manufacturing) 분야 등에서 곡면을 표현하는 구조로 많이 사용하고 있다.

본 연구는 대규모 형상 데이터(점군 및 면군)를 신속하게 렌더링 하고, 렌더링 결과를 실시간으로 회전시켜 볼 수 있는 렌더링 엔진을 개발하는데 있다. 따라서 렌더링 성능을 측정하기 위한 평가 기준으로서 1) 대규모 형상 데이터를 렌더링 하는데 소요된 시간과 2) 렌더링 이후 뷰 변경에 따른 동적 렌더링 속도를 초당 프레임 수(frame per second)로 측정하였다.

본 연구에서 사용한 컴퓨터 운영 체제는 window 7 (64bit), 모델명은 HP Z820 워크스테이션, CPU는 Intel Xeon (3.30GHz), RAM은 16.0GB이다. 그리고 사용한 그래픽 카드는 NVIDIA Quadro 4000, GPU는 GF100, 비디오 메모리 크기는 2,048MB, 대역폭(Bandwidth)은 89.9 GB/s이다.

○ 대규모 점군(point cloud) 예제

대용량 LAS 파일을 사용하여 대규모 점군 데이터의 렌더링 성능을 측정하였다. Table 1은 각 LAS 파일 별로 측정된 파일 크기(KB), 렌더링 된 점의 개수, 파일 로딩부터 렌더링까지 소요된 총 시간(second), 동적 뷰 렌더링 속도(fps), 그리고 시스템 메모리 사용량(KB)을 보여주고 있다. 그리고 컴퓨터 화면 상에 나타난 각 LAS 파일의 렌더링 결과는 Fig. 6에서 Fig. 9와 같다.

LAS 파일의 점군을 렌더링 하는데 사용된 정점 데이터는 정점 위치와 정점 색상이다. 위치는 3개의 float형을, 색상은 3개의 unsigned short형을 사용하였고, 정점 스트림 구성은 SOA(structure of arrays) 형식을 사용하였다. book 모델과 terrain 모델의 경우 실시간으로 렌더링 되는 것으로 측정되었고, village 모델과 road-town 모델의 경우는 점

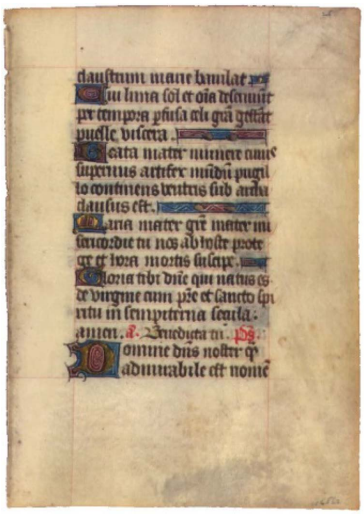


Fig. 6 Book model (point cloud)

Table 1 Rendering performances of point cloud models

Model Name	Book	Terrain	Village	Road-town
File Size (KB)	54,733	96,833	1,504,764	3,224,306
Number of Points	2,155,617	3,837,973	77,043,879	97,108,482
Rendering Time* (sec)	0.516	0.748	7.910	26.770
Rendering Speed (fps)	64.0+	64.0+	4.92	1.95
Memory Consumption** (KB)	24,964	29,436	51,440	331,932

*It means the total time taken from file loading to rendering.
**It was measured by windows task manager.

Fig. 7 Terrain model (point cloud)

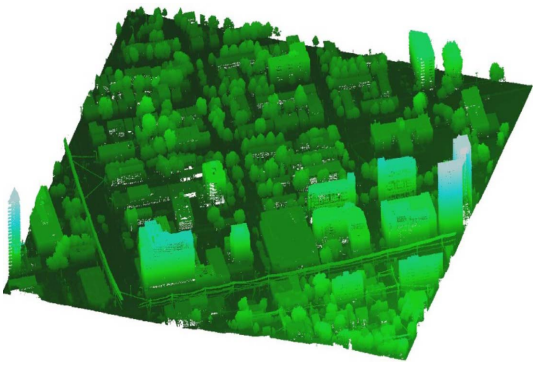
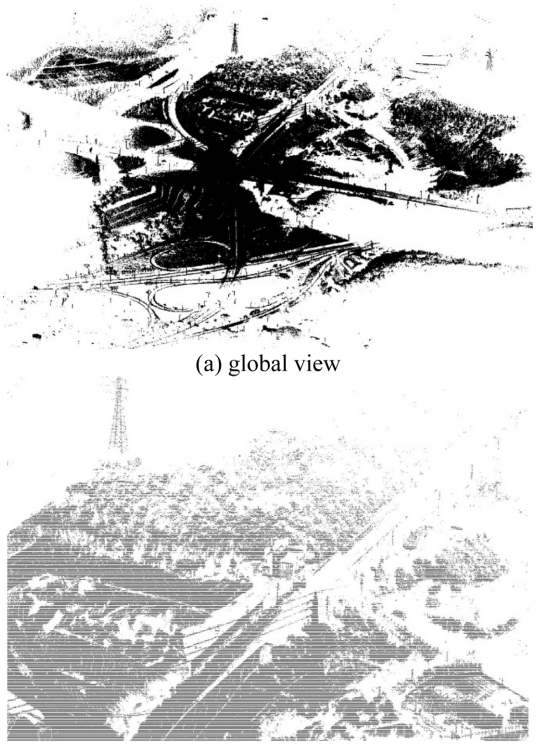


Fig. 8 Village model (point cloud)

의 개수가 각각 0.77억 개, 0.97억 개 수준이라 시스템 메모리를 사용하는 파일 로딩에 많은 시간이 소요되었고, 더불어 렌더링 속도도 4.9230 fps와 1.9468 fps로 측정되어 실시간 처리에는 실패하였다.

Table 2는 실시간 렌더링이 가능한 점군의 크기를 측정하기 위해, 특정 모델의 점군 크기에 따른 렌더링 속도(fps)의 측정 값을 보여주고 있다. 일반적으로 실시간 렌더링이란 렌더링 속도가 15 fps 이상인 경우를 말한다. 결국, Table 2의 결과로부터 점의 개수가 91,000,000개 이하일 때 실시간 렌더링이 가능함을 확인할 수 있다.



(a) global view

(b) detail view

Fig. 9 Road-town model (point cloud)

○ 대규모 폴리곤 모델(polygonal model) 예제

대용량 STL 파일을 사용하여 대규모 면군 데이터의 렌더링 성능을 측정하였다. Table 3은 각 STL 파일 별로 측정된 파일 크기(KB), 렌더링 된 점의 개수, 파일 로딩부터 렌더링까지 소요된 총 시간(second), 동적 뷰 렌더링 속도(fps), 그리고 시스템 메모리 사용량(KB)을 보여주고 있다. 그리고 컴퓨터 화면 상에 나타난 각 STL 파일의 렌더링 결과는 Fig. 10에서 Fig. 12와 같다. 참고로 각 모델의 면(facet)의 개수는 점의 개수의 1/3 수준이다.

STL 파일의 면군을 렌더링 하는데 사용된 정점 데이터는 정점 위치와 정점 법선이다. 위치와 법선 모두 각각 3개의 float형을 사용하였고, 정점 스트림 구성은 SOA(structure of arrays) 형식을 사용

Table 2 Rendering speed of the road-town models with different sizes

File Size (KB)	3,224,306	2,767,698	2,418,268	2,071,721
Number of Points	97,108,482	83,356,523	72,832,504	62,395,337
Rendering Speed (fps)	1.95	31.5	42.9	59.7
Real-time* rendering or not?	No	Yes	Yes	Yes

*Officially, real-time rendering speed must be 15 fps or faster.

Table 3 Rendering performances of polygonal models

Model Name	Mobius	Spikeball	Pnl-side
File Size (KB)	15,019	19,177	1,776,851
Number of Points	922,734	1,178,178	109,169,685
Rendering Time* (sec)	0.094	0.125	8.674
Rendering Speed (fps)	64.0+	64.0+	4.28
Memory Consumption** (KB)	24,748	24,844	611,405

*It means the total time taken from file loading to rendering.
**It was measured by windows task manager.

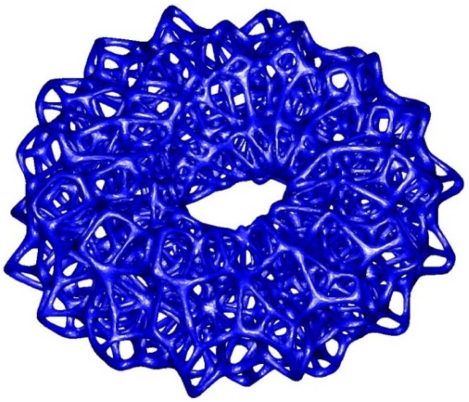


Fig. 10 Mobius model (polygon mesh)

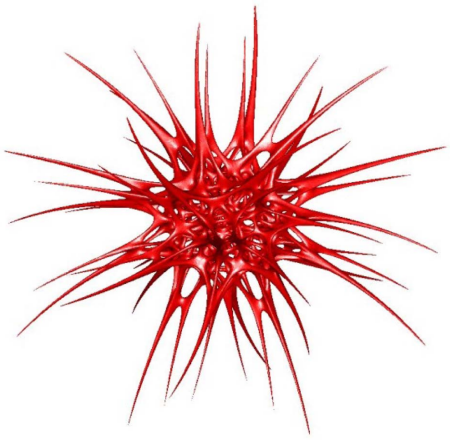


Fig. 11 Spikeball model (polygon mesh)

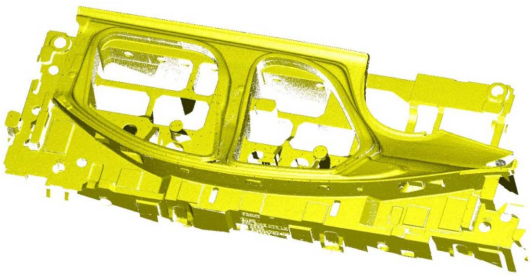


Fig. 12 Pnl-side model (polygon mesh)

하였다. mobius 모델과 spikeball 모델의 경우 실시간으로 렌더링 되는 것으로 측정되었고, pnl-side 모델의 경우는 점의 개수가 1.09억 개 수준이라 파일 로딩에 많은 시간이 소요되었고, 더불어 렌더링 속도도 4.2845 fps로 측정되어 실시간 처리에는 실패하였다.

Table 4는 실시간 렌더링이 가능한 점군의 크기를 확인하기 위해, 특정 모델의 점군 크기 별 렌더링 속도(fps)를 보여주고 있다. 일반적으로 실시간 렌더링이란 렌더링 속도가 15 fps 이상인 경우를 말하는데, Table 4의 결과를 살펴보면, 점의 개수가 98,000,000개 이하일 때 실시간 렌더링이 가능함을 확인할 수 있다.

본 연구에서 제시한 GLSL기반 렌더링 방법의 주요 특징(장점)을 기존 방식(GLSL 이전 방식)과 비교하여 살펴보면 아래와 같다.

Table 4 Rendering speed of the Pnl-side models with different sizes

File Size (KB)	1,776,851	1,348,604	532,353	265,878
Number of Points	109,169,685	82,858,215	32,707,746	16,335,498
Rendering Speed (fps)	4.28	31.3	62.8	64.0+
Real-time* rendering or not?	No	Yes	Yes	Yes

*Officially, real-time rendering speed must be 15 fps or faster.

□ 대규모 형상 데이터(점의 개수가 1억 개 수준)를 렌더링 할 수 있다. 만약 3차원 점(x, y, z) 1개가 12B를 차지하고, 사용하고 있는 컴퓨터의 비디오 메모리 크기가 1 GB라면 약 0.83억 개의 점을 렌더링 할 수 있다. 본 연구에서 사용한 비디오 메모리의 크기는 약 2 GB이므로 대략 1.66억 개의 점을 렌더링 할 수 있다.

□ 본 연구 방식을 사용할 경우에 실시간 렌더링이 가능한 점의 개수는 Table 2와 4의 결과로부터 대략 94,500,000개 이하임을 확인할 수 있다.

□ 본 연구에서 제시하는 GLSL 렌더링 방식은 과거 GLSL 등장 이전 방식에 비해 보다 많은 점군 데이터를 렌더링 할 수 있다. 워크스테이션 급 고성능 컴퓨터에서 GLSL 등장 이전에 사용하던 OpenGL API 함수, glBegin(), glVertex3d(), glEnd()를 사용할 경우에 3,000,000개의 점군을 렌더링 할 때 그 속도가 대략 10 fps이다[13]. 본 연구 방식의 경우엔 Table 2로부터 대략 93,000,000개의 점군을 렌더링 할 때 10 fps가 나온다. 결국 이러한 사실로부터 본 연구 방식이 GLSL 이전 방식에 비해 같은 렌더링 속도로 31배의 많은 점군을 처리할 수 있음을 알 수 있다.

□ 본 연구 렌더링 방식은 렌더링 할 데이터를 비디오 메모리에 바로 올려놓고 GPU에 의해 차례대로 데이터에 접근하여 렌더링하는 방식이다. 이에 비해 GLSL이전 방식은 매 프레임마다 CPU가 시스템 메모리로부터 데이터를 읽은 후 비디오 메모리로 전송하고 이것을 GPU가 차례대로 접근하여 처리한다[13]. 즉 이전 방식은 시스템 메모리에서 비디오 메모리로의 상당히 느린 전송 과정이 더 있는 것이다. 참고로 glVertex3d() 등의 함수를 실행할 때 해당 메모리를 할당하고 해지하는 작업이 일어나는데 이것 또한 렌더링 속도를 저하시키는 원인으로 작용한다.

□ 일반적으로 CPU와 시스템 메모리 사이의 대역폭보다 GPU와 비디오 메모리 사이의 대역폭이 4배 높다. 또한 GPU의 비디오 메모리 접근 속도가 CPU의 비디오 메모리 접근 속도보다 20배 빠르다[13]. 이는 대규모 형상 데이터의 실시간 렌더링을 위해서는 시스템 메모리보다 비디오 메모리에 저장하는 것이 고성능 구현에 훨씬 유리임을 보여준다.

4. 결 론

본 연구에서는 대규모 형상 데이터를 실시간으로 렌더링 하는 방법을 제시하였다. 최근 빠른 속도로 업그레이드되고 있는 OpenGL의 GLSL를 사용하여 대규모 점군 혹은 대용량의 폴리곤 모델을 그래픽 비디오 메모리에 올려놓고 GPU에 의해 렌더링을 수행하는 쉐이더 프로그램을 개발하였다. 그래픽 비디오 메모리에 형상 데이터를 업로딩하기 위한 정점 스트림에 관하여 소개하였고, 이를 VAO와 VBO를 사용하여 비디오 메모리에 업로드하는 방법을 소개하였다. 그리고 vertex shader와 fragment shader로 이루어진 쉐이더 프로그램에 관하여 본 연구에서 구현한 각 쉐이더의 소스 코드를 소개하였고 이를 사용한 쉐이더 프로그램의 생성 및 수행 절차에 관해 기술하였다. 또한 전체 렌더링 과정을 설명하기 위해 렌더링 준비 과정으로서 두 개의 알고리즘을 설명하였고, 실제 렌더링 수행에 필요한 한 개의 알고리즘을 자세히 설명하였다.

본 연구에서 제시한 렌더링 기법은 LAS 파일 및 STL 파일 예제를 통하여 기존 방식에서 수행하기 어려웠던 대규모 형상 데이터의 렌더링 가능성을 보여주었고, 더불어 본 연구 방식의 한계점도 함께 기술하였다.

향후 연구 과제로서 marching cubes을 이용하여 음함수 곡면(implicit surface)[14] 혹은 레벨 셋 곡면(level set surface)을 정교하게 실시간으로 렌더링하는 알고리즘에 관한 연구가 추진될 예정이고, 더불어 옥트리 구조에 기반을 둔 복셀(voxel) 렌더링 기법에 관한 연구 또한 필요하다. 상기 연구 주제는 형상 모델링 및 렌더링 분야에서 여러 방식으로 해결책을 모색해 왔으나 근본적으로 CPU 기반의 방식이라는 한계점을 극복하지 못하였다. 그러나 GLSL의 등장으로 GPU에 의한 실시간 렌더링이 어느 때 보다 기대되고 있는 상황이다.

감사의 글

이 논문은 2012년도 한국교통대학교의 해외파견연구교수지원금을 받아 수행한 연구임

References

1. Elseberg, J., Borrmann, D. and Nüchter, A., 2013, One Billion Points in the Cloud – an Octree for Efficient Processing of 3D Laser Scans, *Journal of Photogrammetry and Remote Sensing*, 76, pp. 76-88.
2. Krishnamurthy, V. and Levoy, M., 1986, Fitting Smooth Surfaces to Dense Polygon Meshes, *Proc. SIGGRAPH*.
3. Curless, B. and Levoy, M., 1996, A Volumetric Method for Building Complex Models from Range Images, *Proc. SIGGRAPH.*
4. Yemez, Y. and Schmitt, F. 1999, Progressive Multilevel Meshes from Octree Particles, *Proc. 3D Digital Imaging and Modeling.*
5. http://en.wikipedia.org/wiki/OpenGL_Shading_Language.
6. http://www.opengl.org/wiki/Main_Page.
7. Segal, M. and Akeley, K., 2014, *The OpenGL® Graphics System: A Specification (Version 4.4 (Compatibility Profile))*, The Khronos Group, March 19.
8. Segal, M. and Akeley, K., 2014, *The OpenGL® Graphics System: A Specification (Version 4.4 (Core Profile))*, The Khronos Group, March 19.
9. http://www.opengl.org/wiki/Vertex_Specification.
10. http://glm.g-truc.net/0.9.5/index.html.
11. ASPRS Board of Directors, 2013, LAS Specification Version 1.4 - R13, July 15, The American Society for Photogrammetry & Remote Sensing.
12. http://en.wikipedia.org/wiki/STL_(file_format).
13. Crassin, C., 2011, *GigaVoxels: A Voxel-Based Rendering Pipeline For Efficient Exploration Of Large and Detailed Scenes*, Ph.D. Thesis, Université de Grenoble.
14. Park, S. and Chung S.Y., 2008, The Design and Implementation of Implicit Object Classes for Geometric Modeling System, *Transactions of the Society of CAD/CAM Engineers*, 13(3), pp.187-199.

박 상 근

1991년 포항공과대학교 학사
1993년 서울대학교 기계설계학과 석사
1997년 서울대학교 기계설계학과 박사
1997년~1999년 삼성SDS 정보기술 연구소 책임연구원
2000년 서울대 BK21 기계분야사업단 계약교수
2000년~2002년 (주)K&I 테크놀로지 책임연구원
2003년~현재 국립한국교통대학교 기계공학과 교수
관심분야: Computational Geometry, CAD/CAM/CAE, Design Optimization, Scientific Visualization