Multi-threaded Software Graphics Pipeline

(The Access Control List based SecureOS)

지도교수 : 장병탁

이 보고서를 공학학사 학위 논문
대체 보고서로 제출함.

2014년 6월 13일

서울대학교 공과대학
컴 퓨 터 공 학 부
심 우 철

2014년 8월

Abstract

software graphics render는 매우 많은 양의 계산을 CPU에 요구하기 때문에 multi-threading 기법을 이용한 성능 향상을 고려해볼 수 있다. 그리고 직접 multi-threaded software graphics renderer를 구현해봄으로써 프로젝트 진행자는 해당 분야에 대한 이해도를 넓힐 수 있다. 이를 위해 이미 알려진 3d graphics rendering pipeline중에서 rendering에 필수적인 pipeline만을 구현하고 이를 multi-threading으로 parallel하게 처리한다. 성능 측적은 크기가 큰 모델과 작은 모델을 사용해봄으로써 입력 크기가 커질수록 multi-threading을 통한 성능 향상 효과가 명확해짐을 보인다. 또한 load balancing 방식, 동기화 방식을 2가지 고안하여 각각 비교해보았다. 여기 CPU를 profiling 하여 실제 동작을 확인해보았다. 그 결과 크기가 큰 모델에 대해 스레드 숫자에 대한 선형에 가까운 성능 향상을 확인할 수 있었다.

1. Introduction

graphics rendering은 그래픽스 객체의 좌표계 변환, vector data를 raster image로 변환, 특수 효과 적용 및 객체 가공 등의 과정을 통해 이루어진다. 이 과정을 통해 많은 양의 계산을 처리해야 하기 때문에 GPU라 부르는 graphics rendering 전용 하드웨어를 사용한다. graphics rendering 과정은 3D 벡터 모델을 2D 래스터 이미지로 바꾸는 과정으로 파악할 수 있다. 이 과정에서 특히 중요한 것이 바로 rasterizer이다. 따라서 본 프로젝트에서는 대표적인 상용 rendering pipeline인 DX11 Graphics Pipeline 모델을 기초로 하고 rasterizer에 중점을 두어 rendering pipeline을 디자인하도록 하겠다.

rasterizer는 vector data를 raster image로 직접 변환하는 pipeline으로, 이를 위한 여러 가지 알고리즘이 소개되어 있지만 그 중에서도 scanline algorithm이 가장 대표적이라고 할 수 있다. scanline algorithm은 임의의 다각형 polygon을 입력으로 받아 polygon을 구성하는 edge의 정보를 edge table 구조체에 저장한다. edge table을 기반으로 scanline list를 y좌표를 하나씩 바꿔가며 동적으로 관리해주게 되는데, 이 과정은 동적인 list 연산을 사용하기 때문에 많은 시간 비용을 지불해야 한다. 하지만 본 프로젝트에서는 삼각형 polygon만을 입력으로 받는다고 가정하기 때문에 scanline list를 array를 활용하여 정적으로 구현 및 관리할 수 있게 된다. list 연산을 array 연산으로 대체하는 것이기 때문에 비교적 성능 향상을 기대해볼 수 있다. 뿐만 아니라 본래의 scanline list를 관리하는 과정에서 발생하는 예외 처리를 삼각형 특화 algorithm에서는 배제할 수 있기 때문에 솔루션 크기가 작아지게 된다. 실제 간단한 general scaline algorithm으로 구현한 rasterizer와 삼각형 특화 sacline algorithm으로 구현한 rasterizer의 성능을 측정해본 결과 후자가 더 좋은 성능을 보이는 것을 확인했다.

멀티코어 환경이 대중화 되면서 muti-threading 기법이 더욱 주목을 받기 시작했다. 특히 graphics rendering을 위한 연산은 대체로 각자에 대해 독립적이며, 계산 양이 매우 많기 때문에 muti-threading의 수혜를 받기 쉬운 입장에 있다. 실제로 GPU에는 복수의 계산 유닛이 있어 복잡한 렌더링을 빠르게 처리할 수 있다. 다만 multi-threading 기술을 적용하는 것은, 즉 병렬화 된 프로그램을 작성하는 것은 순차적 프로그램을 작성하는 것 보다 난이도가 있는 일이다. 뿐만 아니라 스레드를 만들고 운용하는 작업은 비용이 큰 편에 속하며 이외에도 공유 자원의 원활한 관리를 위한 동기화, 각 스레드에 최대한 균등하게 작업을 분배하여 스레드의 활용률을 높이기 위한 load balancing 등등의 이슈가 산적해있다. 그럼에도 불구하고 올바른 muti-threading 활용의 효과는 강력하다고 알려져 있다.

올바른 muti-threading을 구현하기 위해서는 작업을 각 스레드에 균등하게 분배해주기 위

한 load balancing 이슈와 스레드들이 공유하는 데이터의 정합성이 유지하기 위한 synchronization 이슈를 해결할 방안을 고안해야 한다.

load balancing의 경우 작업의 분배 방식과 작업의 분할이라는 두 가지 측면에서 접근해볼 수 있다. 전자의 경우 단순하게는 각 스레드에 동일한 숫자의 작업을 미리 분배하는 정적인 방법을 생각해볼 수 있다. 하지만 이러한 방법은 쪼개진 작업의 실제 작업량이 균일한 상황에서는 유효할 수 있으나 그렇지 않은 경우에는 적절하지 않은 방법이다. 따라서 스레드에서 개별 작업을 끝마칠 때 마다 공유 작업 큐로부터 작업을 가져오는 동적인 방법이 적절할 것이다. 다만 이 경우에는 synchronization 이슈가 발생한다. 후자의 경우 polygon으로 작업을 쪼개는 방법과 screen을 grid 형태로 쪼개서 좌표별로 작업을 쪼개는 방법을 생각해볼 수 있다. 두 방법 모두 장단점이 있으나 본 프로젝트에서는 구현과 일정상 한계 때문에 polygon을 기준으로 작업을 분할했다.

synchronization은 작업의 정합성을 보장하기 위한 overhead로 최소한의 비용으로 최대한의 정합성을 보장하는 것이 관건이다. 일반적으로는 mutex를 사용하나 mutex의 경우 kernel 자원을 사용하기 때문에 심각한 속도 저하를 유발할 수 있으며 실제로 mutex로 구현한 결과 속도가 저하되는 것을 관찰할 수 있었다. 따라서 대안으로 atomic operation을 이용하는 방법을 채택하였다. 본 방법은 약간의 정합성을 희생하여 속도 저하를 막는 방법이다. 본 프로젝트에서는 두 방법에 대해서도 비교해보도록 한다.

성능 측정은 기본적으로 스레드 숫자에 따른 단위 시간당 처리한 polygon의 숫자를 측정하는 실험을 수행하였다. 추가로 모델의 숫자와 synchronization 방법을 바꿔가며 실험 데이터를 측정함으로써 입력 모델 크기와 synchronization 방법의 차이에 따라 성능 향상이 어떻게 달라지는가에 대해서 알아보도록 했다. 결과 크기가 큰 모델과 atomic operation을 썼을 때 선형에 가까운 이상적인 성능 향상을 보일 수 있었다.

2. Rendering Pipeline

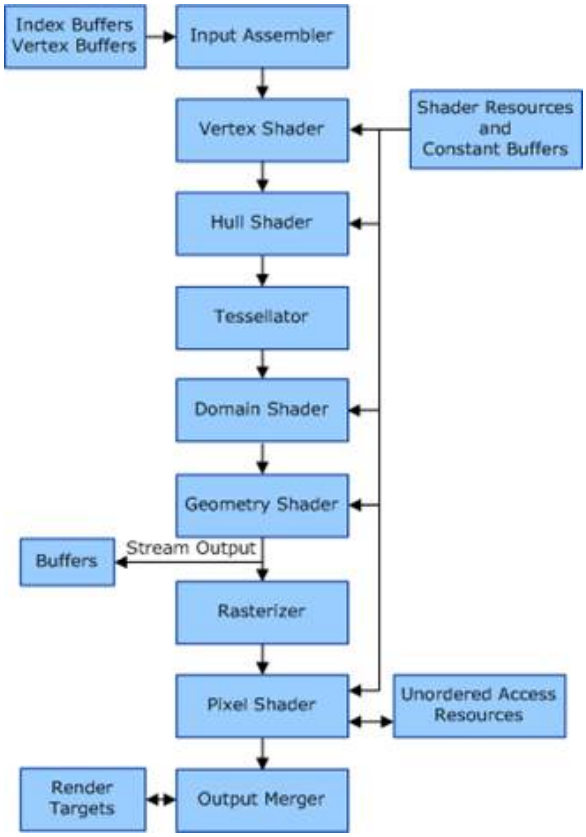


그림 1 DX11 Rendering Piepline

위 그림은 DX11의 Rendering Pipeline의 도식으로, 일반적인 Rendering Pipeline은 위에서 크게 벗어나지 않는다. 위의 파이프라인은 크게 Input Assembler, Vertex Shader, Tessellate(Hull Shader, Tessellator, Domain Shader), Geometry Shadre, Rasterzier, Pixel Shader로 분류해볼 수 있다. 이 중에서 렌더링에 필수적인 pipeline은 Input Assembler, Vertex Shader, Rasterizer 이다. 각 pipeline이 하는 일은 다음과 같다.

2..1 Input Assembler

외부로부터 3D 모델을 입력으로 받아 pipeline 에서 처리할 수 있는 형태의 자료구조로 변환하여주는 일종의 파서의 역할을 한다. 모델은 파이프라인 내부에서는 vertex와 vertex로 구성된 face(polygon)으로 나타내어진다. 모델들은 임의의 다각형 폴리곤 다수로 구성되어 있는 것이 일반적이나, 본 프로젝트는 삼각형 폴리곤으로만 구성된 모델만을 입력으로 받도록 한다.

2.2. Vertex Shader

그래픽스 렌더링 과정에서 모델은 복수의 좌표계 변환 과정을 거친다. 그 중에서도 모델 좌표계에서 월드 좌표계로의 변환을 Vertex Shader 모듈에서 처리한다. 좌표계의 변환은 평행 이동, 회전, 확대 및 축소 등을 나타내는 행렬 곱으로 나타내는데 이러한 행렬 연산을 Affine Tramformation이라고 한다. Affine Transformation은 Linear Transformation과 Translation의 합성으로 이뤄지며 점은 점으로, 선은 선으로, 면은 면으로 변환되는 성질과

선들의 평행성이 유지되는 성질을 갖고 있다.

X, X′는 서로 다른 Affine Space의 한 좌표, E는 Linear translation, L은 Translation이라 할 때 X′ = XE + L을 Q->P 인 Affine Transformation이라 고하며 다음과 같은 형태를 갖는다.

$\begin{bmatrix}\hat{x}\\ \hat{y}\\ \hat{z}\\ 1\end{bmatrix} = \begin{bmatrix}a_x & b_x & c_x & t_x\\ a_y & b_y & c_y & t_y\\ a_z & b_z & c_z & t_z\\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}x\\ y\\ z\\ 1\end{bmatrix}$ $\begin{bmatrix}\hat{x}\\ \hat{y}\\ \hat{z}\end{bmatrix} = \begin{bmatrix}a_x & b_x & c_x\\ a_y & b_y & c_y\\ a_z & b_z & c_z\end{bmatrix}\begin{bmatrix}x\\ y\\ z\end{bmatrix} + \begin{bmatrix}t_x\\ t_y\\ t_z\end{bmatrix}$	$\begin{bmatrix}\hat{x}\\ \hat{y}\\ \hat{z}\\ 1\end{bmatrix} = \begin{bmatrix}1 & 0 & 0 & 0\\ 0 & \cos\theta & -\sin\theta & 0\\ 0 & \sin\theta & \cos\theta & 0\\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}x\\ y\\ z\\ 1\end{bmatrix}$ $= \begin{bmatrix}x\\ y\cos\theta - z\sin\theta\\ y\sin\theta + z\cos\theta\\ 1\end{bmatrix}$
<기본형>	<x축 회전>
$\begin{bmatrix}\hat{x}\\ \hat{y}\\ \hat{z}\\ 1\end{bmatrix} = \begin{bmatrix}1 & 0 & 0 & t_x\\ 0 & 1 & 0 & t_y\\ 0 & 0 & 1 & t_z\\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}x\\ y\\ z\\ 1\end{bmatrix} = \begin{bmatrix}x+t_x\\ y+t_y\\ z+t_z\\ 1\end{bmatrix}$	$\begin{bmatrix}\hat{x}\\ \hat{y}\\ \hat{z}\\ 1\end{bmatrix} = \begin{bmatrix}\alpha & 0 & 0 & 0\\ 0 & \beta & 0 & 0\\ 0 & 0 & \gamma & 0\\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}x\\ y\\ z\\ 1\end{bmatrix} = \begin{bmatrix}\alpha x\\ \beta y\\ \gamma z\\ 1\end{bmatrix}$
<평행 이동>	<확대 및 축소>

그림 2 Affine Transformation의 종류

2.3. Tessellation

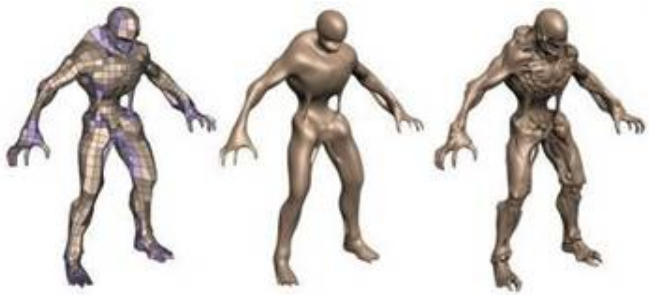


그림 3 모델에 Tessellation 기법을 적용

기존 모델의 폴리곤을 쪼개서 좀 더 세세한 부분까지 표현할 수 있게 해주는 모듈이다. 위 그림에서 왼쪽이 기존의 모델이고 가운데가 tessellate 과정을 거친 모델이며 오른쪽은 displacement mapping까지 적용한 모델이다. 오른쪽으로 갈수록 모델이 더욱 섬세해지는 것을 관찰할 수 있다. 보다 정교한 렌더링을 위한 pipeline이다.

2.4. Geometry Shader

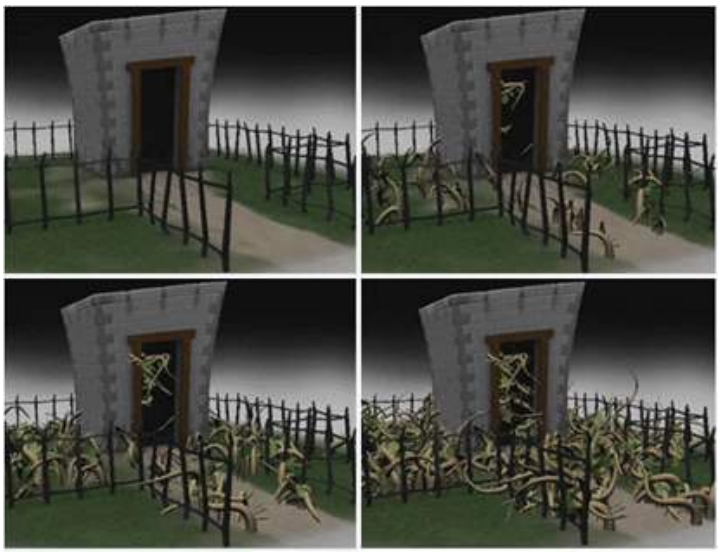


그림 4 모델에 Geomtry Shader를 적용

기존의 폴리곤을 기반으로 하여 새로운 폴리곤을 생성하는 모듈이다. 위 그림은 geometry shader를 적용하여 덩굴이 점점 무성해지는 과정을 보여주고 있다. 보다 풍성한 표현을 위한 pipeline이라고 할 수 있다.

2.5. Rasterizer

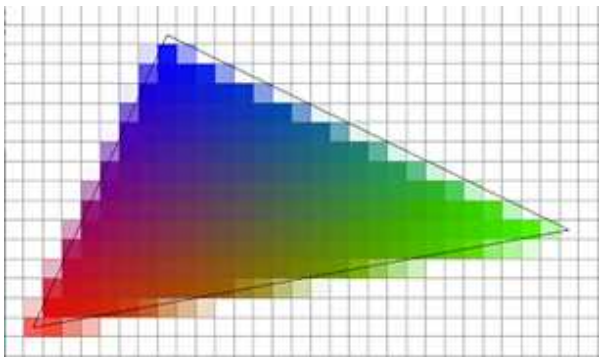


그림 5 Raster Image

실제 화면을 채워 넣을 픽셀의 상태를 결정한다. 즉 vertor image를 raster image로 변환하는 pipeline이다. 전체 pipeline에서 핵심이 되는 pipeline이라고 할 수 있다.

스크린 버퍼에 픽셀의 값을 채워 넣기 전에 렌더링할 대상의 좌표계를 월드 좌표계에서 실제 평면 화면 좌표계인 2D viewport 좌표계로 전환한다. 이 때 카메라의 위치와 표시 범위가 중요하게 작용한다. 폴리곤을 픽셀로 변환하는 과정에서 다양한 렌더링 최적화 알고리즘을 적용할 여지가 있다.

작업은 주어진 폴리곤을 순차적으로 화면에 픽셀로 나타내는 과정으로 이루어져있다. 이를 위한 방법에는 여러 가지가 있는데 가장 대표적인 것이 위에서도 언급한 scanline algorithm이다. 폴리곤 단위로 작업이 독립적으로 수행되기 때문에 병렬화 하기 매우 좋은 특성을 갖고 있다.

2.6. Pixel Shader

스크린 버퍼에 채워진 픽셀들에 조명 효과 등을 적용하여 픽셀을 변형, 보다 사실적인 그래픽스 렌더링을 가능하게 하는 단계이다. 광원의 종류, 광원의 위치와 빛의 입사각 등등을 고려하여 픽셀을 조작한다. 또한 Shading 방식에 따라 Flat Shading과 Smooth Shading으로 나뉘며 Smooth Shading은 다시 Gouraud Shading과 Phong Shading으로 나뉜다.

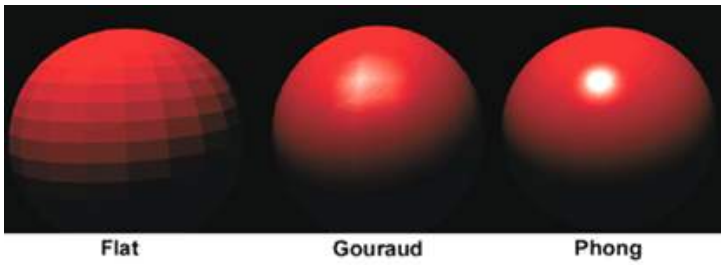


그림 6 각 Shading의 비교

Flat Shading은 face 전체를 동일한 색상으로 표현한다. 따라서 렌더링 품질이 상당히 떨어진다. 다만 계산이 단순해지기 때문에 한정된 자원밖에 쓸 수 없는 상황에서 강점을 보인다.

반면 Smooth Shading은 픽셀 별로 Shading이 적용되기 때문에 계산 량이 많아지는 대신 렌더링 품질이 좋아진다. 각 픽셀의 intensity는 face의 vertex의 intensity를 bilinear interpolation하여 구하는데 이를 Gouraud Shading이라 하며, 여기에 추가로 각 픽셀의 normal을 vertex를 기준으로 bilinear interpolation하여 Shading하는 것을 Phong Shading이라고 한다. 추가 작업이 있는 만큼 Phong Shading이 더욱 정교한 렌더링을 한다.

각 픽셀과 face에 대한 shading 작업은 모두 독립적이기 때문에 병렬화 하기에 용이하다는 특징이 있다.

3. Scanline Rasterizer

3.1. General Scanline Rasterizer

Rasterizer 단계에서 주어진 vector face 정보를 raster 정보로 변환하는데 사용하는 알고리즘이다. 각 scanline과 face간에 교차점을 찾아 이를 중심으로 픽셀이 그려지는 영역과 그려지지 않는 영역으로 나눈다. 영역을 나누는 방법은 다음과 같다.

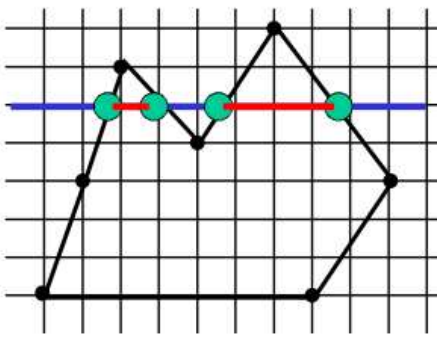


그림 7 scanline rasterizing

위 그림에서 파란 선은 scanline이고 녹색 점은 face와 scanline의 교차점이다. 단힌 face의 경우 극값과 같은 특수한 경우를 제외하면 반드시 짝수개의 교차점이 발생한다. 따라서 교차점의 x값을 기준으로 순서대로 2개의 교차점을 하나의 쌍(pair)로 묶어줄 수 있다. 이렇게 생성된 교차점의 쌍 사이에 존재하는 픽셀들을 채워주면 vector face를 raster화 할 수 있

다.

다만 위의 방법대로라면 교차점이 vertex이며 극값인 경우, 교차점인 vertex이나 극값은 아닌 경우, x축에 수평인 edge의 경우 문제가 발생할 수 있으므로 해당하는 경우에 대해서는 예외처리를 해줘야 한다. 첫번째 경우는 vertex 위치에 교차점이 2개 있는 것으로 생각하면 된다. 두번째 경우는 위쪽 edge에서 현재 scanline과 다음 scanline 사이에 속하는 부분을 잘라주고 처리하면 된다. 세번째 경우는 해당 edge를 없는 것처럼 생각한다.

3.2. Triangle Specific Scanline Algorithm

위는 임의의 다각형 폴리곤을 입력으로 받는 경우의 scanline algorithm이다. 하지만 본 프로젝트에서는 삼각형 폴리곤만을 입력으로 받게 된다. 따라서 위와 같은 일반적인 algorithm이 아닌 삼각형에 특화된 algorithm을 구현하도록 한다. 이 경우 본래의 algorithm이라면 scanline list를 관리할 때 발생하는 list 연산을 array 연산으로 대체하여 성능을 향상시킬 수 있다는 장점이 있다.

3.2.1. 반올림으로 인한 문제

실제 입력으로 받는 폴리곤의 꼭지점에 해당하는 vertex의 좌표는 float 자료형이다. 따라서 raster image를 만들기 위해서는 integer 자료형으로 변환해줘야 하는 일이 생긴다. 본 rasterizer에서는 반올림을 한다. 이로 인해 y_max의 값이 올려지는 경우 다음과 같은 문제가 발생할 수 있다.

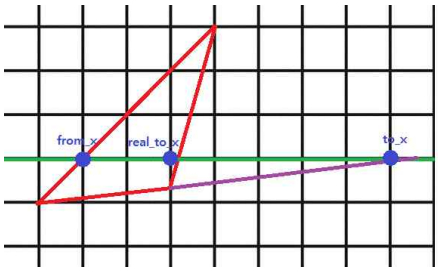


그림 8 반올림으로 인한 오차

실제 입력으로 받는 폴리곤의 꼭지점에 해당하는 vertex의 좌표는 float 자료형이다. 따라서 raster image를 만들기 위해서는 integer 자료형으로 변환해줘야 하는 일이 생긴다. 본 rasterizer에서는 반올림을 한다. 이로 인해 y_max의 값이 올려지는 경우 다음과 같은 문제가 발생할 수 있다.

3.2.2. Tiny Polygon 문제

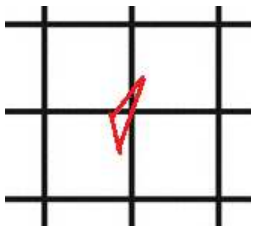


그림 9 Tiny Polygon

위와 같이 크기가 매우 작은 폴리곤의 경우 그려지지 않고 생략될 수 있는데 이런 경우 원래 그려져야 하는 픽셀이 생략되어서 모델에 구멍이 생길 수 있다. 이런 경우 픽셀의 크기

가 일정 값보다 작은 것을 감지하면 적당한 픽셀을 찾아 해당 픽셀이 채워지도록 하는 처리를 해야 한다. 단, 폴리곤 크기 검사는 넓이로 하면 float 연산의 증가로 인해 renderer의 전반적인 성능을 악화시키므로 edge의 길이를 검사하도록 한다.

4. Multi-thread Model

본 프로젝트에서 구현하기로 한 vertex shader, rasterizer, pixel shader는 모두 독립적인 복수의 작업으로 쪼갤 수 있기 때문에 병렬화에 용이하다. Vertex shader는 모든 작업의 수행 시간이 거의 일정할 것으로 예상되므로 각 스레드에 동일한 개수로 배분하면 된다. Pixel shader는 rasterizer와 함께 구현하기로 했으므로 여기서는 rasterizer의 multi-threading에 대해서만 다루기로 하겠다.

4.1. Load balancing

Rasterizer의 기법은 스크린을 격자 형태로 나누는 binning 방식과 폴리곤 별로 나누는 방식으로 나눠서 생각해 볼 수 있다. Binning 방식은 bin이라 불리는 복수의 격자로 스크린을 나누고 각 스크린을 개별 스레드 작업 단위로 하여 스레드에 할당하는 방식이다. 이 방식은 실제로 화면에 그려야 하는 픽셀의 수에 비례하여 작업이 나누어지기 때문에 균형 있는 로드 밸런싱을 기대해볼 수 있다는 장점이 있다. 반면 binning을 관리하기 위한 오버헤드가 발생하며 구현 난이도가 상승한다는 단점이 있다.

폴리곤 별로 나누는 방식은 간단하게 구현 할 수 있다는 장점이 있다. 다만 폴리곤 별로 스크린에 표시되는 영역의 크기가 다르기 때문에 극단적인 경우에 매우 큰 폴리곤 하나의 존재로 인하여 전체적으로 성능이 하락할 수 있다는 로드 밸런싱 측면에 있어 단점이 있다. 하지만 웬만한 편차는 작업을 끝낸 스레드가 남아있는 폴리곤을 순차적으로 바로 가져가는 greedy한 방식의 작업 분배를 통해 극복 가능하다. 특히 이러한 방식은 전체 폴리곤 숫자가 커지면 커질수록 균등하게 작업을 분배할 수 있다는 장점을 갖는다.

위 두 방식 중 어떤 방식이 속도 측면에서 우위에 있는지 알아보기 위해 각자의 방식으로 구현한 간단한 프로토타입을 구현하여 속도를 비교해본 결과 폴리곤 기준으로 나누는 방식이 더 우위를 보였으므로 binning 방식을 배제하도록 한다.

4.2. Synchronization

atomic operation을 이용한 방식과 mutex를 이용한 방식으로 나눠볼 수 있다. 이 두 가지 방식에 대해서는 향후 실험을 통해 성능 비교를 해보도록 한다.

4.3. Thread Recycling

매 파이프라인 처리 한번마다 작업 스레드를 생성하는 방식은 스레드 생성 Overhead가 매우 크기 때문에 부적절하다. 따라서 작업 스레드를 미리 생성해놓고, 처리할 Shader단계에 도달할때마다 작업 스레드를 Invoke시켜 해당 Shader를 처리하는 방식으로 구현한다.

5. Experiment

5.1. Experiment Design and Input Data

구현한 프로그램은 키보드나 마우스 입력에 의한 카메라 이동 등 스크린을 다시 그려야 할 필요가 있는 경우에만 Display()함수를 호출한다. 따라서 연속적인 카메라 이동을 통해 다수의 Display()호출이 발생하는 경우의 실행 시간을 측정한다. 실험에서는 원점을 기준으로 카

메라를 0.02Radian씩 일주시키는 방식을 사용한다. 또한, 동기화로 의한 Overhead를 파악하기 위해 Mutex와 Atomic Operation 두 가지 경우에 대해 파악한다.
사용한 모델의 정보는 다음과 같다.

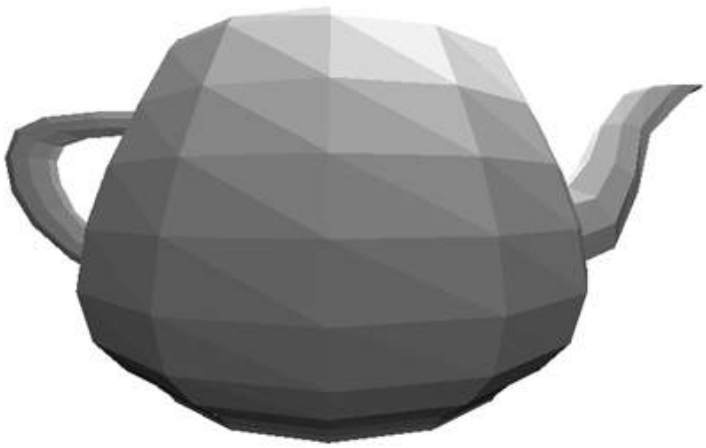


그림 10 Model 1 <Teapot>

그림 11 Model 2 <Brandenburg Gate>

Model 1<Teapot> : 1024 Polygons, 530 Vertices
Model 2<Brandenburg Gate> : 390218 Polygons, 622673 Vertices

Model 1은 약 1000여개의 polygon으로 구성되어 있고 Model 2는 약 40만개의 polygon으로 구성되어 있다. 일반적으로 올바르게 구현된 mult-threading은 문제의 크기가 커질수록 multi-threading으로 인한 ovehead에 투입되는 비용의 비중이 전체 작업 비중에 비해 점차 줄어든다고 알려져 있다. 따라서 두 모델의 rendering 성능을 측정하여 서로 비교해봄으로써 모델이 커질수록 multi-threading으로 인한 성능 향상 효과가 커지는지 혹은 그렇지 않은지를 확인함으로써 구현물에 대한 전반적인 평을 내려볼 수 있을 것이다.

5.2. Result Graph

그래프 X축 : 스레드 수
그래프 좌측 Y축 : Rasterizer Speed (단위 microsecond) -> # of Polygon / Rasterizing time
그래프 우측 Y축 : Vertex Shader Speed (단위 microsecond) -> # of Face /

VertexShader time
Back Face/Hidden Face Culling, Clipping, z-Buffer, Flat Shading, Light 적용

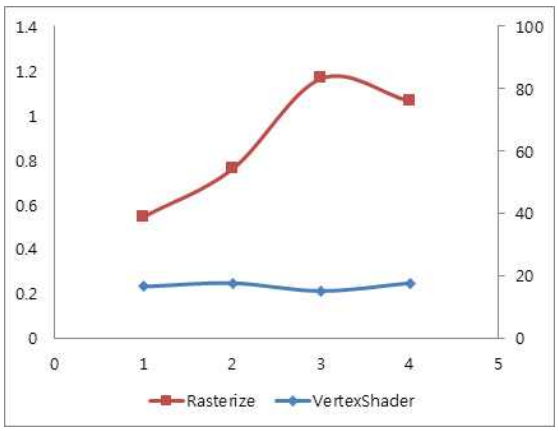


그림 12 Using Mutex (Model 1)

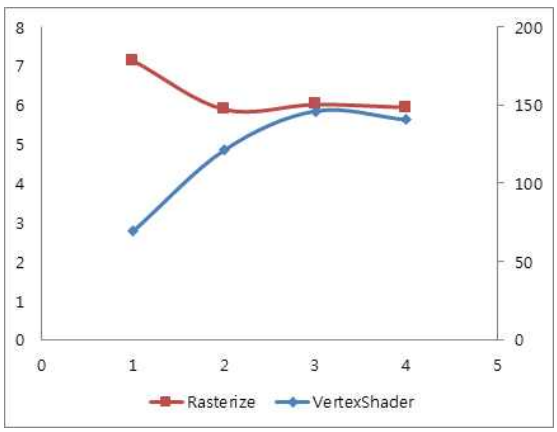


그림 13 Using Mutex (Model 2)

mutex를 사용한 결과 mutex 호출 overhead로 인해 스레드 숫자가 커질수록 multi-threading으로 인한 이득보다 overhead가 더 커지는 것을 볼 수 있다.

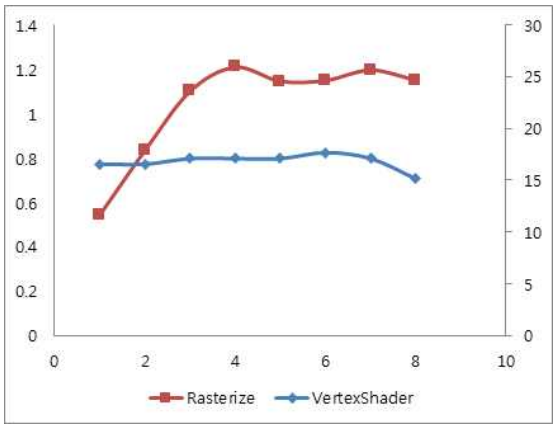


그림 14 Using Atomic Operation (Model 1)

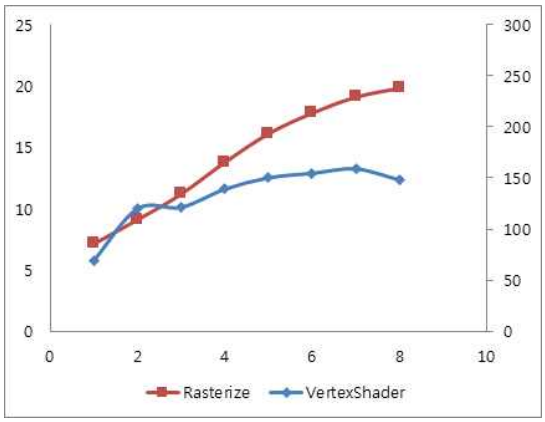


그림 15 Using Atomic Operation (Model 2)

반면 atomic operation을 사용한 경우 model 1은 어느 순간부터 성능의 변화가 거의 없으나 model 2의 경우 성능 향상이 분명하게 측정되는 것을 볼 수 있다.

처리량이 많은 BrandenBurg Gate 모델의 Rasterize는 거의 선형에 가까운 성능 향상을 나타낸다.

5.3. Result Analysis and Discussion

5.3.1. 폴리곤 수가 적을 경우

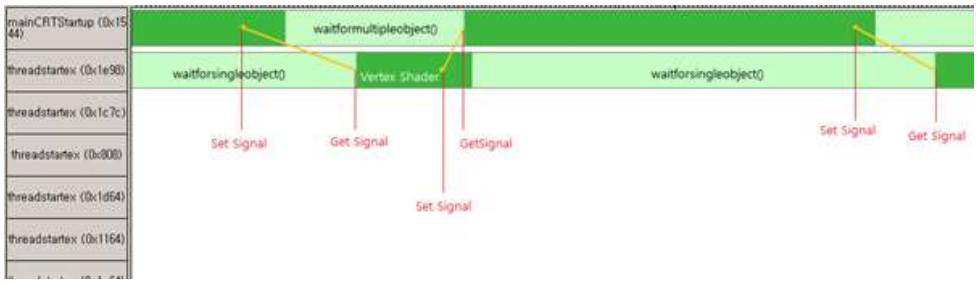


그림 16 Single Thread Invoke/Wait Overhead in Teapot

위 그림은 Teapot 모델을 1개 스레드로 처리했을 때의 모습이다. 기본적으로는 전체 Workload의 양이 작은 경우(Teapot)에는 multi-threading으로 인한 성능 향상이 한계를 보인다. 이 스레드들을 Invoke/Wait하는데 걸리는 오버헤드가 실제 계산량에 비해 크기 때문이다.

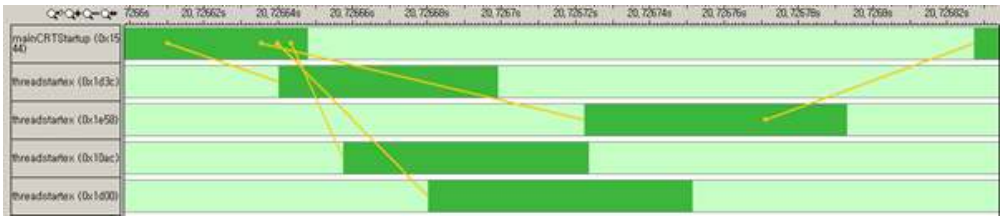


그림 17 4 Thread Invoke/Wait Overhead in Teapot

4개의 스레드를 사용할 때에는 Signal로 스레드를 Invoke하여 실제 스레드가 Idle에서 벗어나는 데 까지 걸리는 시간과 스레드가 실제 작업을 처리하는 시간이 거의 비슷하다. 반면 스레드 처리량이 매우 많아 위와 같은 오버헤드를 무시할 수 있는 경우에는 Invoke/Wait로 인한 속도 감소는 무시할 수 있으리라 예상된다. 이는 다음 그림을 통해 확인해 보도록 한다.

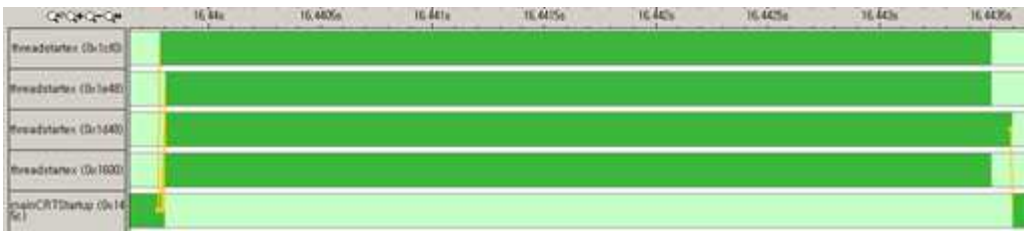


그림 18 Thread Invoke/Wiat Overhead in Brandenbrug Gate

위는 BrandenburgGate모델의 Vertex Shader부분이다. Teapot의 경우와는 달리, Invoke/Wait에 소모되는 시간이 전체 처리량에 비해 무시할 수 있을 정도로 작은 것을 알 수 있다. 이러한 경우에는 Brandenburg Gate의 그래프와 같이 Shader 처리 속도 증가의 한

계가 Teapot모델처럼 명확하게 드러나지 않는다.

5.3.2. Mutex VS Atomic Operation

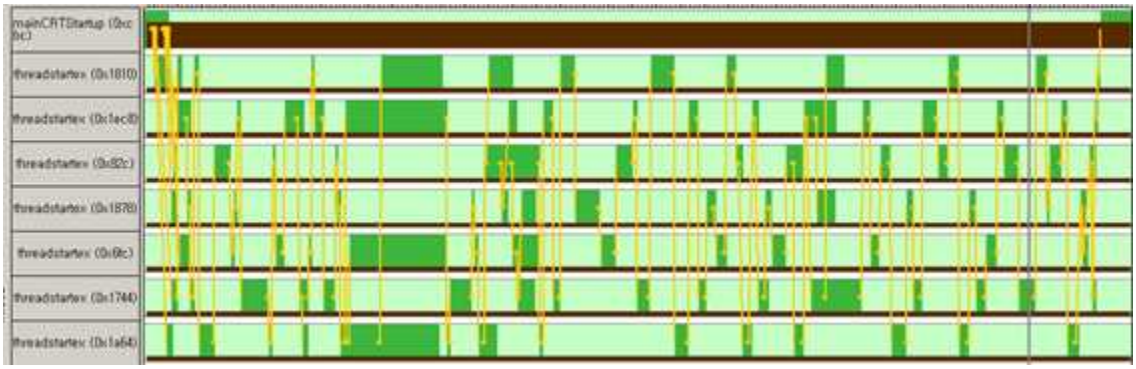


그림 19 Mutex Signaling Overhead

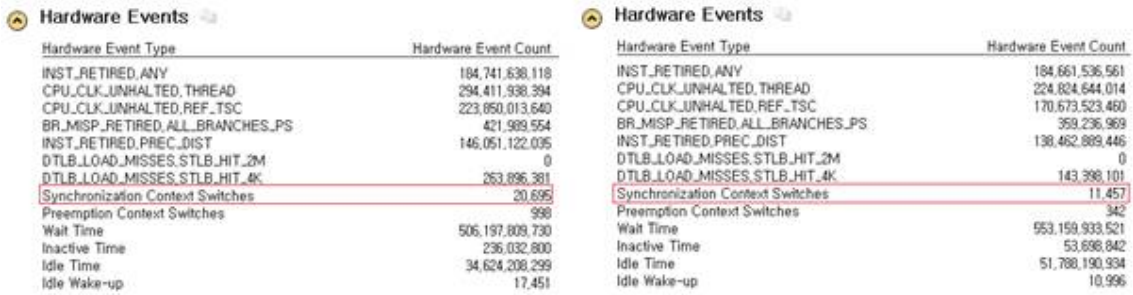
Mutex 사용 시에는 Signaling이 매우 많이 발생하여 전체적인 실행 속도가 떨어지는 것을 볼 수 있다. 따라서 프로젝트에서 구현한대로 Atomic Operation을 통한 작업 할당을 사용하는 것이 적합하다.

5.3.3. Context Switching

이상적으로는 하이퍼 스레딩을 통한 논리 코어 8개에 각각 스레드가 스케쥴되어야 하겠지만, 실제로는 다른 프로그램의 스레드도 CPU상에서 실행되는 경우가 있을 수 있다. 아래의 측정 결과에서 한 스레드가 다른 스레드에 비해 실행이 늦어지는 경우를 파악할 수 있다. 작업 스레드는 총 8개이지만, 메인 스레드도 CPU에서 작업을 하고 있을 것이기에 총 스레드의 수는 9개가 된다. 이러한 경우, 작업 스레드가 모두 8개의 코어에서 나뉘어 처리되기 위해서는 적어도 한 개의 스레드는 Context switching이 일어나야 하기 때문에 이와 같은 overhead가 생기는 것으로 보인다.

그림 20 Context Switching Overhead in 8 Thread

스레드 8개/4개를 사용하는 경우에 대해 확인한 결과, Context Switch도 2배 차이가 난다.



Hardware Events

Hardware Event Type	Hardware Event Count
INST_RETIRED.ANY	184,741,638,118
CPU_CLK_UNHALTED.THREAD	294,411,938,394
CPU_CLK_UNHALTED.REF_TSC	223,850,013,640
BR_MISP_RETIRED.ALL_BRANCHES_PS	421,989,554
INST_RETIRED.PREC_DIST	146,051,122,035
DTLB_LOAD_MISSES.STLB_HIT_2M	0
DTLB_LOAD_MISSES.STLB_HIT_4K	263,896,381
Synchronization Context Switches	20,695
Preemption Context Switches	998
Wait Time	506,197,809,730
Inactive Time	236,032,800
Idle Time	34,624,208,299
Idle Wake-up	17,451

Hardware Events

Hardware Event Type	Hardware Event Count
INST_RETIRED.ANY	184,661,536,561
CPU_CLK_UNHALTED.THREAD	224,824,644,014
CPU_CLK_UNHALTED.REF_TSC	170,673,523,460
BR_MISP_RETIRED.ALL_BRANCHES_PS	359,236,969
INST_RETIRED.PREC_DIST	138,462,889,446
DTLB_LOAD_MISSES.STLB_HIT_2M	0
DTLB_LOAD_MISSES.STLB_HIT_4K	143,398,101
Synchronization Context Switches	11,457
Preemption Context Switches	342
Wait Time	953,159,933,521
Inactive Time	53,698,842
Idle Time	51,788,190,934
Idle Wake-up	10,996

그림 21 Comparing with 4 Thread and 8 Thread

5.3.4. Load Balancing

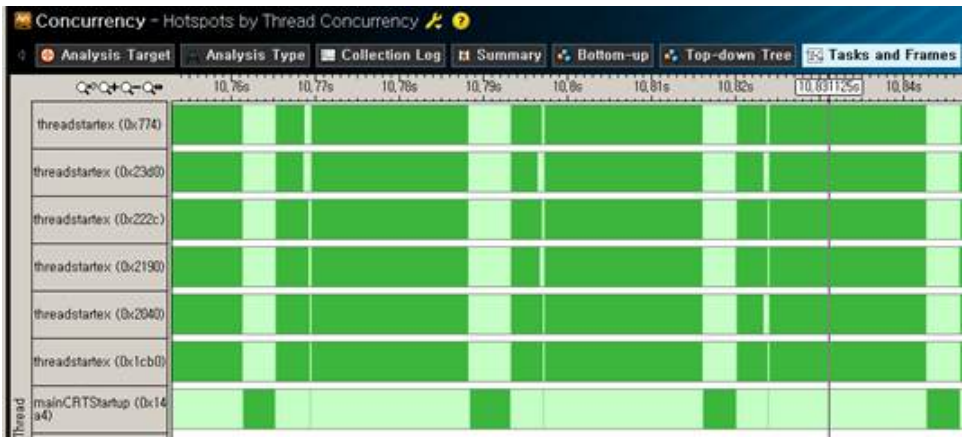


그림 22 thread 8개 이하에서의 load balancing

위 분석 결과를 통해 8개 이하의 스레드에서는 load balancing이 잘 되고 있음을 볼 수 있다.

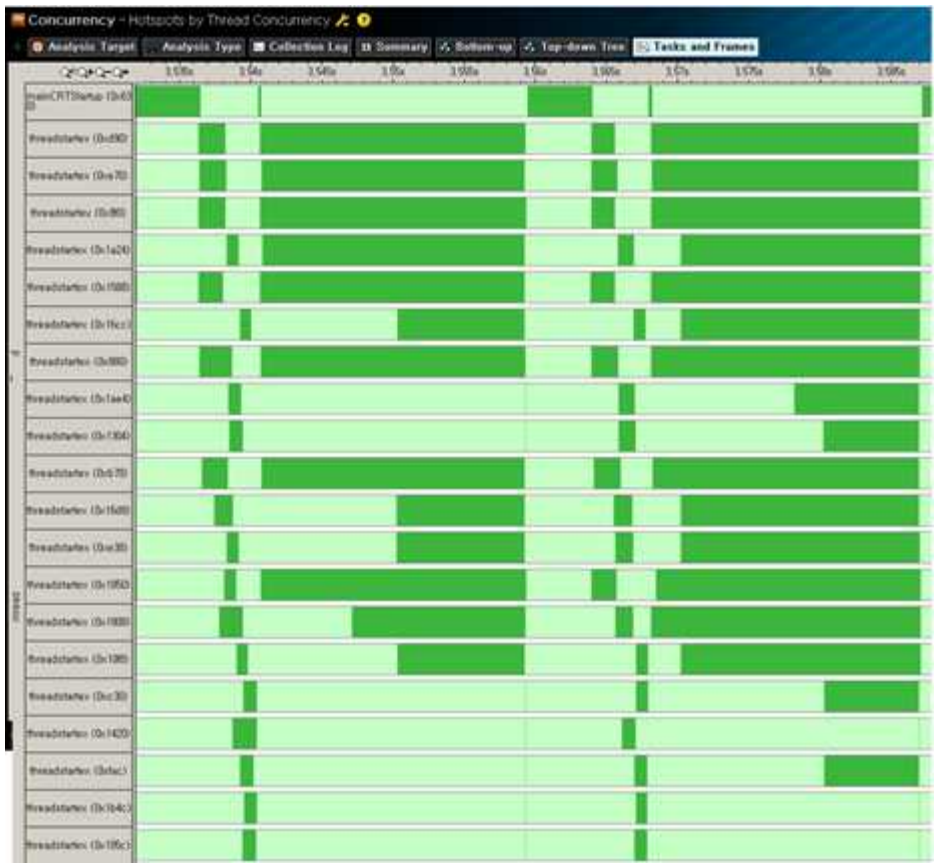


그림 23 thread 8개 초과에서의 load balancing

다만 스레드 수가 너무 많아 Context switching가 필요 이상으로 발생한다면 정상적인 Load balancing이 이루어지지 않는 것을 볼 수 있다.

6. Conclusion

software renderer를 구현하고 이에 multi-threading을 적용해보기 위해 우선 DX11 rednering pipeline을 기반으로 하여 Vertex Shader, Rasterizer, Pixel Shader를 구현하였다. 이 중에서 특히 rasterizer에 중점을 두었다.

rasterizer의 경우 삼각형 polygon에 특화된 scanline algorithm을 사용하였다. 실제로 개조한 algorithm은 프로토타입으로 구현한 기존의 algorithm에 비해 약 20fps 정도의 성능 향상을 보였다. 삼각형 polygon만 입력으로 들어오는 상황은 매우 특수한 상황이지만 입력으로 들어오는 모델의 다각형 polygon을 삼각형 polygon으로 변환해주는 converter를 개발한다면 자원이 한정된 상황에서 실용적으로 사용할 수 있는 가능성이 생긴다.

polygon 단위로 작업을 분할하고 정적으로 작업을 스레드에 분배하는 방식의 multi-threading 기법을 적용하였고 synchronization 방법과 모델의 크기에 대해서는 선택할 수 있게 하여 결과를 비교해보았다. mutex나 크기가 작은 모델을 쓸 경우에는 muti-threading overhead가 커지기 때문에 성능 향상이 나쁘게 나타날 것이라 예측하였다. 실제로 둘 중에 하나의 조건이라도 해당하는 경우에는 일정 스레드 숫자를 넘어가면 성능 향상이 없거나 오히려 성능이 악화되기도 하였다. 반면 큰 크기의 모델을 atomic operation을 이용한 synchronization을 사용하였을 때 스레드 숫자에 대해 선형과 가까운 성능 향상을 이끌어 내어 이상적인 결과를 얻어내는 데 성공하였다. 세부적으로는 vtune을 이용한 CUP profiling을 통해 각 스레드들의 running time이 거의 동일하다는 것을 발견하였으므로 load balancing도 적절하게 이뤄지고 있다고 할 수 있다. 이는 만족할만한 성과를 얻어내었다고 결론지을 수 있을 것이다. 그래도 성능 향상과 성능 측정 및 분석에 있어 아쉬운 점이 존재한다.

본 프로젝트는 multi-threading을 이용하여 renderer의 전반적인 성능 향상을 이뤄냈다. 이 때 구현한 세 가지 파이프라인에 대해 multi-threading을 적용하였다. 하지만 그래픽스 라이브러리의 함수에 프로그램이 픽셀을 채워 넣은 스크린 버퍼를 파라미터로 넘겨서 화면에 실제 픽셀을 출력하도록 하는 consumer code 부분에 대해서는 프로젝트 목표에 의한 제약 때문에 multi-threading을 적용할 수 없었다. 이 루틴은 시간 비용이 많이 소모되며 실제로 model1에 대해서는 스레드 1개와 8개일 각각 전체 수행 시간의 66%, 80%를 차지했고 model2에서는 8%, 20%를 차지했다. 따라서 해당하는 부분을 최적화할 수 있다면 또 한 번의 진전을 이뤄볼 수 있을 것이다. 이외에는 어셈블리의 인라인 코드를 이용한 극한의 최적화 방법을 생각해 볼 수 있다.

synchronization 방법의 경우 atomic operation은 속도는 보장되나 정합성 측면에 있어 약점을 보인다. 약점을 보안할 수 있는 방법으로 z-buffer와 screen buffer를 스레드별로 따로 만들어서 작업하고 최종적으로 합치는 방법을 생각해볼 수 있으나 일정상의 문제로 구현하지 못하였다. 차후 추가적으로 구현할 수 있는 부분이다.

측정 방법에 있어서는 측정 머신의 코어 숫자 한계가 아쉬웠다. 측정 머신의 코어가 8개였기 때문에 사실상 8개 이상의 스레드를 통한 성능 향상을 기대할 수 없는 여건이었다. 만약 여건이 된다면 더 많은 수의 멀티코어 환경 위에서 더 많은 수의 스레드를 동원해봄으로써 더욱 풍부한 측정 데이터를 얻을 수 있었을 것이라 생각한다.

[1] Yafei Wang, Zhenjie Chen, Liang Cheng, Manchun Li, Jiechen Wang, *Parallel scanline algorithm for rapid rasterization of vector geographic data*, Computers & Geosciences 59, 31-40, 2013

[2] Thomas Yu-Kiu Kwok, Chandrasekhar Narayanaswami, Bengt-Olaf Schneider, *Method for parallelizing software graphics geometry pipeline rendering*, International Business Machines Corporation, 1998

[3] James Tulip, James Bekkema, Keith Nesbitt, *Multi-threaded game engine design*, Proceedings of the 3rd Australasian conference on Interactive entertainment
Pages 9 - 14, 2006

[4] Allen Sherrod, Wendy Jones, *Beginning DirectX 11 Game Programming*, Course Technology, 2012

[5] 주우석, *OpenGL로 배우는 컴퓨터 그래픽스*, 한빛미디어, 2006

[6] Woo-Chan Park, Kil-Whan Lee, Kim, I.-S., Tack-Don Han, Sung-Bong Yang, *An effective pixel rasterization pipeline architecture for 3D rendering processors*, Computers, IEEE Transactions on Volume 52 Issue 11, 2003

[7] DEMETRESCU, Stefan. *High speed image rasterization using scan line access memories*, Proceedings of the 1985 Chapel Hill Conference on VLSI. 1985. p. 221-243.

[8] KIM, Donghyun; KIM, Lee-Sup. Area-efficient pixel rasterization and texture coordinate interpolation. Computers & Graphics, 2008, 32.6: 669-681.