

종합 설계 프로젝트 수행 보고서

|  |  |  |
| --- | --- | --- |
| **프로젝트명** | Unity에서의 GPGPU 기반 광선 추적 렌더러 | |
| **팀번호** | S2-2 | |
| **문서제목** | 수행계획서 |  |
| 2차발표 중간보고서 |  |
| 3차발표 중간보고서 |  |
| 최종 결과 보고서 | O |
| **팀장** | 김 한 상 | |
| **팀원** | 김 수 혁 | |
| 정 지 윤 | |
| **지도교수** | 공 기 석 | |
| 노 영 주 | |

# **목차**

종합 설계 프로젝트 수행 보고서 0

목차 1

종합설계 프로젝트 수행 계획서 2

1. 서론 2

**1.1.** **작품선정 배경 및 필요성** 2

**1.2.** **기존 연구/기술동향 분석** 6

**1.3.** **개발 목표** 8

**1.4.** **팀 역할 분담** 8

**1.5.** **개발 일정** 8

**1.6.** **개발 환경** 9

2. 본론 10

**2.1.** **개발 내용** 10

**2.2.** **문제 및 해결방안** 18

**2.3.** **시험 시나리오** 19

**2.4.** **상세 설계** 20

**2.5.** **Prototype 구현** 29

**2.6.** **시험 / 테스트 결과** 41

**2.7.** **Coding & Demo** 43

3. 참고문헌 48

종합설계 프로젝트 수행 계획서

1. 서론

## **작품선정 배경 및 필요성**

* + 1. 배경 : General Purpose of GPU, GPU 의 범용 계산

원래 GPU는 실시간 컴퓨터 그래픽을 위하여 개발된 하드웨어이나, GPU 가 가진 다른 계산 방식의 특성을 살려 GPU 자체를 여러 프로그램을 동시에 실행시키는, GPGPU로 사용하는 기술이 2000년대에 출시되었다. CPU는 많은 캐시메모리/레지스터를 이용한 메모리 I/O 속도를 빠르게 하여 적은 수의 강력한 코어를 사용하는 strong/multi-core 전략이였다면, GPU는 많은 코어 수를 가지나 이에 대한 한계로 적은 레지스터/캐시메모리를 가져, 보다 많이 느린 메모리 I/O 속도를 가지는 week/many-core 전략을 가진 형태로 발전했다.

이는 기존 GPU 시장의 가능성을 열어주는 것으로 굉장히 획기적 이였으나, 기존의 폰 노이만 구조에 익숙해진 인력풀이 GPU를 사용하는 이기종 컴퓨팅 기술에 진입할 시간이 부족했으며, 이를 리드할 수 있는 산업의 인력 풀 자체가 크지 않았기 때문에 지금처럼 크게 알려지지는 않았었다. (많은 경우의 B2B 상품에 사용, 예:MRI) GPU 기술의 리드는 NVidia 에서 담당했는데, 기존의 수익구조가 뚜렷하고(조립 PC 시장), 연구진들에 대해 꽤 많은 투자를 꾸준하게 했기 때문에 GPU H/W, S/W 기술은 점진적으로 계속 발전하고 있었다.

2012년, AlexNet 이 인공 신경망에 대한 GPU 의 활용으로 인공 신경망 자체의 실용성, 실현 가능성을 보여주어 딥 러닝 연구들의 활성과 함께 GPU 산업 자체가 빠르게 성장하기 시작하여 현재 많이 쓰이는 딥러닝 프레임워크(tensorflow, café, ..)들은 대부분 NVidia의 cuda 기반 API 인 cuDNN 을 사용하여 구현한 것으로 시작되었다. 그림 1에서 이를 볼 수 있다.

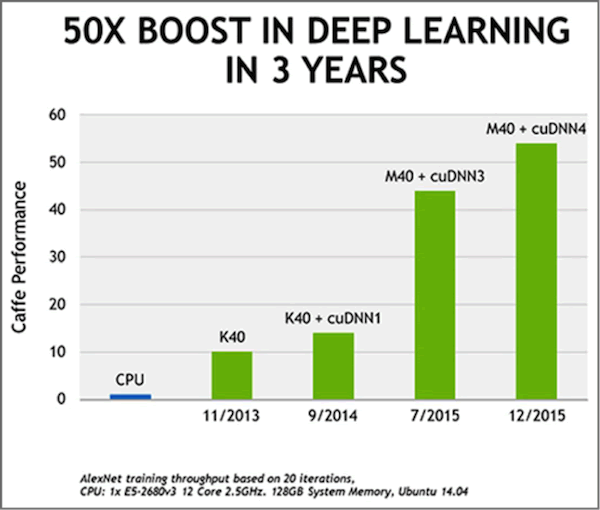


그림 1 : AlexNet의 CPU 및 각종 GPU & 라이브러리와의 성능 비교

* + 1. 배경 : Ray-Tracing, 광선 추적

컴퓨터 그래픽 기술은 크게 두가지, Ray-Tracing 그리고 Rasterization으로 나뉜다. 이들은 전부 3D 로 된 물체의 위치와 각 물체의 표면의 특징, 광원과의 상호작용을 계산하여, 하나의 2차원 색의 집합으로 만드는 것, 즉 모든 3D 물체의 위치에 따라 2D 이미지에 투영시키는 것은 같다. 하지만 그림 2에서 두가지의 차이점을 볼 수 있는데, ray-tracing 은 이미지의 픽셀의 위치와 카메라 방향을 따라서 물체의 존재여부를 추적하여, 이 정보들을 사용해 색을 계산한다. Rasterization 은 존재하는 각각의 3D 물체들을 homogenous transform을 사용하여 2차원 이미지 상에 투영하는 방법이다. 즉 ray-tracing 은 픽셀별로 모든 물체들을 탐색을 하고, rasterization은 각각의 물체를 이미지의 픽셀에 투영시키는 방법이다.

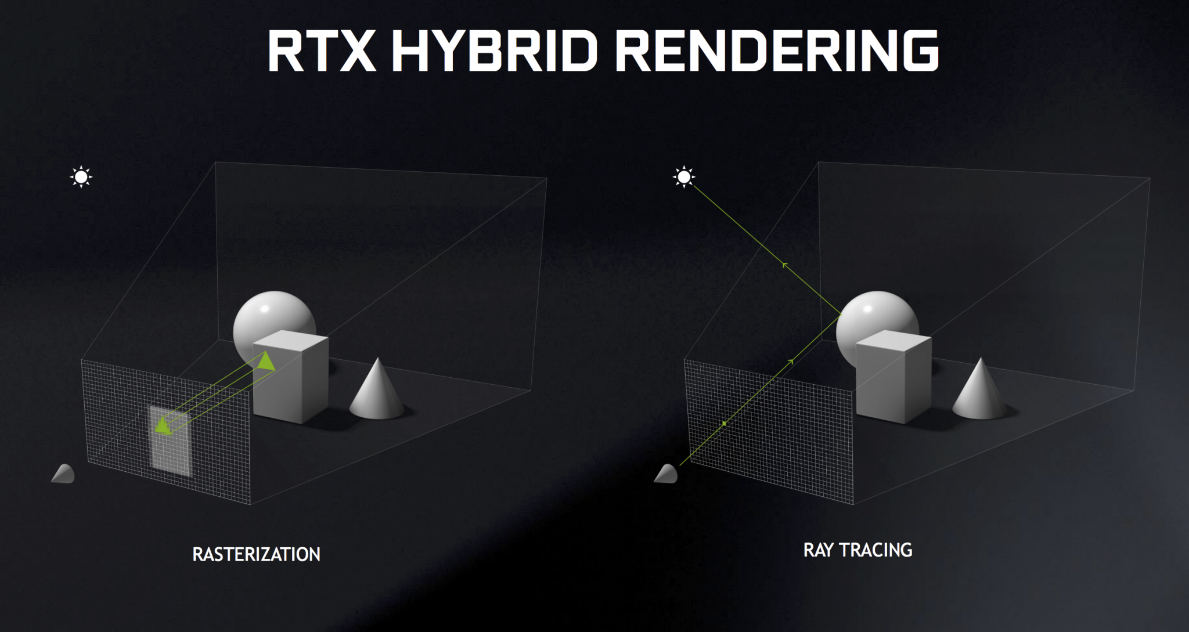


그림 2 : rasterization vs ray-tracing

RTX 기종이 출시되기 전, 상용 GPU들은 전부 다 rasterization 기반의 계산만을 지원했다. 기본적으로 3차원 그래픽을 게임 같은 반응성이 좋아야 하는 실시간 렌더링에 사용되기도 했고, ray-tracing은 rasterization보다 HW 가속을 지원하여 얻는 성능의 이득이 너무 적었고, 이전의 경우에는 HW 제조의 입장에서도 크게 차이가 나기도 했었기 때문에 rasterization으로만 HW 가속을 지원했었다. 그리하여 graphics API 인 OpenGL, DirectX 모두 rasterization을 위한 API 로 구성되어 있다.

앞서 언급한 딥 러닝의 도구로 GPU가 사용되면서, NVidia 에서는 기존의 프로그램만을 돌리는 CUDA Core 를 변형하여, Tensor Core 라는 이름의 딥 러닝에서 필요한 Matrix 계산을 빠르게 할 수 있는 코어를 개발하여 CUDA Core + Tensor Core 를 가진 워크스테이션용 가속기(GPU의 역할을 수행하지 못하는 병렬 연산만을 제공하는 칩)을 만들었다. 이에 영감을 얻어 다음 신제품을 발표할 때, CUDA Core + Tensor Core 와 동시에, Ray-Tracing을 계산하는 RT Core를 가진 RTX 제품을 출시했다. 즉 하나의 칩에 적어도 3가지의 큰 종류를 가진 코어들로 칩이 구성된 것이다. 동시에 DirectX API 가 이를 제어하기 위한 DirectX Ray-tracing API를(이하 DXR) 지원하기 시작했다.

하지만 HW 가속을 받는 경우에도 모든 것을 Ray-tracing으로 계산하지는 않는다. 모든 것을 계산하기 위해서는 아직 HW 자체의 지원이 부족하다. 그래서 이전부터 Rasterization으로 한계가 있는 부분들을 Ray-tracing을 모방한 기법들로 해결했던 부분을 이제는 Ray-Tracing 자체를 활용하여 그 한계를 서로 메꿔주는 방법을 사용하는 Hybrid Rendering 방식이 지금은 Ray-Tracing을 사용하는 가장 나은 방법이라고 할 수 있다.

Hybrid-Rendering 방식은 Rasterization 이 기반이기 때문에 Photorealistic한 이미지를 만들어 내기 위해서는 많은 사전처리가 필요하다. 특히 이 작품에서 ***중점***적으로 고려하는 **빛의 움직임의 결과를** 미리 계산하여 저장하는 방법을 고민하고, 이를 달라지는 3D 환경에 따라서 직접 전부다 계산을 돌려주어야 한다. 또한 성능, 결과와 관련하여 이 알고리즘들과 맞물려 조합되는 것들이 존재하기 때문에 이들은 꽤나 골치 아픈 부분이다.

만약 시간의 제한이 꽤 적은 편이고, 모든 것을 Ray-Tracing에서 심화된 Path-Tracing을 사용하여 계산할 경우, 사용자는 쉽게 사실적으로 빛이 표현된 이미지를 얻을 수 있다. 이는 게임 같은 반응성이 굉장히 중요한 실시간 렌더링에서는 사용될 수 없지만, 이외의 반응성이 필요 없는 경우에는 유용하다고 할 수 있다.

* + 1. 배경 : Unity

Unity는 UE4와 함께 일반적으로 많이 쓰이는 상용 게임 엔진 중 하나다. 아래 두개의 통계 자료에서 이를 알 수 있다. 크로스 플랫폼을 지원하며, 다른 상용 엔진에 비해 바닐라 버전이 light-weight 이기 때문에 모바일에서 많이 쓰이는 것을 알 수 있다. 그림 3에서 이를 볼 수 있다.

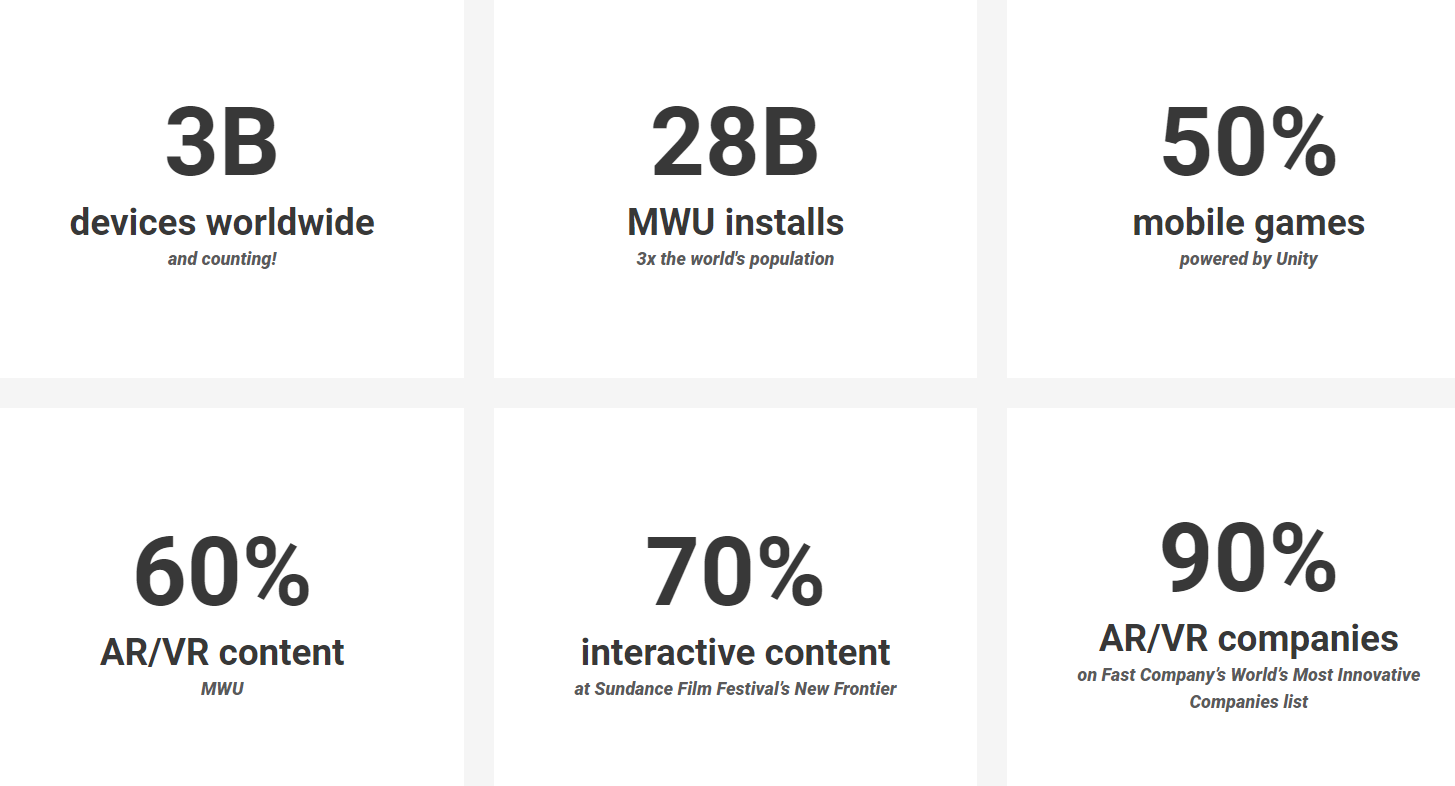


그림 3 : Unity 사용률 ([Powered by Unity](https://unity3d.com/public-relations))

아래 그림 4는 2014년에 진행된 설문에서 Unity 사용률이 가장 높은 비율을 차지한 것을 볼 수 있다. 100명의 응답으로 정확하진 않지만 결과적으로 많은 사용자들이 Unity를 사용한다는 것을 알 수 있는 자료로는 충분하다.

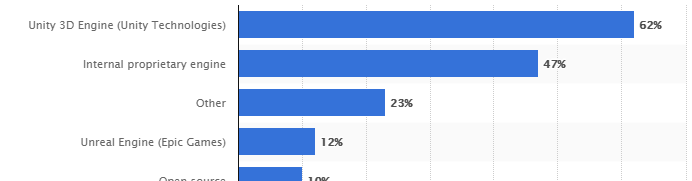


그림 4 : 엔진 사용 통계(영국, 2014, 100명의 응답, 비디오 게임 개발자 대상)([stastista.com](https://www.statista.com/statistics/321059/game-engines-used-by-video-game-developers-uk/))

또한 VR 디바이스의 등장으로, VR 컨텐츠 들이 주목받기 시작하며 많은 컨텐츠들이 쏟아져 나왔는데, Unity가 이에 크게 기여한 것을 위의 통계자료에서 알 수 있다. 주목해야할 사실은 이전의 주로 사용되던 출력 디바이스인 모니터에 비해 체험의 용도로 활용성이 높은 VR 디바이스에 많은 사람들이 집중되었고, 상용 엔진으로 제작하는 VR 컨텐츠는 단순히 게임만이 아니라 인터랙티브 컨텐츠로 일반화되는 계기가 되었다. 이에 더불어 상용 엔진들이 시네마틱 에디팅을 지원함으로써, 디지털 영상 컨텐츠 제작 툴로써의 길을 열게 되었다. 그림 5는 Unity 타임라인을 통해 에디팅을 하는 모습을 직접 담았다.



그림 5 : Unity 타임라인, 시네마틱 에디팅 기능을 지원

* + 1. 필요성

실시간 렌더링 기반의 상용 게임 엔진을 사용하는 사용자 중, 영상 제작의 용도로 사용하는 사용자들이 늘어나게 되었다. 대부분은 선형 분기의 동영상을 만드므로, 이를 미리 계산해 동영상으로 만드는 기능을 원한다. 또한 컨텐츠 제작자의 필요에 따라, 사실적인 빛의 표현을 원하는 경우가 있을 수 있다. 이러한 경우, 앞서 언급한 GPGPU 기반으로 Ray-Tracing을 계산하여 나은 결과를 보여줄 수 있다.

## **기존 연구/기술동향 분석**

1.2.1 OctaneRender

Unity에서 가장 널리 알려진 Ray-Tracing을 활용한 이미지 생성기는 OctaneRender가 존재한다. 이는 Unity에서 약간의 씬 에디팅을 통해 환경을 구성하면, 이를 미리 구현된 플러그인에 넘기어 이미지를 생성한다.

Unity에서 OctaneRender의 파이프라인은 Unity에서 C#, managed memory 기반으로 3D 데이터(메쉬, 물체 별 변환(위치,회전,크기), 재질, 광원)를 자체적으로 구현한 unmanaged memory 기반의 C/C++와 CUDA를 사용한 구현체에서 실질적인 Ray-Tracing 계산을 수행한다. 이는 기본적인 파이프라인에 대한 실질적인 제시라고 할 수 있다.

Unity에서 GPGPU를 사용하기 위해서는 DirectX에서 지원하는 compute shader나, OctaneRender처럼 C#과 C/C++의 상호운용성을 이용하여 CUDA 혹은 OpenCL로 실행해야 한다. compute shader는 일반화된 API이라서 더 세밀하지 못하기 때문에, 퍼포먼스를 생각한다면 CUDA/OpenCL로 갈 수밖에 없다.

OctaneRender는 각 DCC[[1]](#footnote-1)/ 상용엔진 별로 많은 라이선스가 존재한다. Unity의 경우, 하나의 GPU에서 계산하는 경우는 무료, 두개의 GPU에서 계산하는 것부터 유료 구독형 라이센스로 전환된다. 만약, 두개의 GPU의 계산을 지원하면서 무료 라이선스를 지원한다면, 더 나은 서비스라고 할 수 있다.

또한, OctaneRender의 기능들은 실시간 렌더링에(lightmapping) 포커스가 맞춰져 있는 만큼, 영상 제작의 편의성에 있어 더 나은 기능을 보인다면, Unity 플러그인 분야의 다른 시장을 공략한다고 할 수 있다. 그림 6에서는 OctaneRender를 통해 만들어진 컨텐츠를 보여준다.

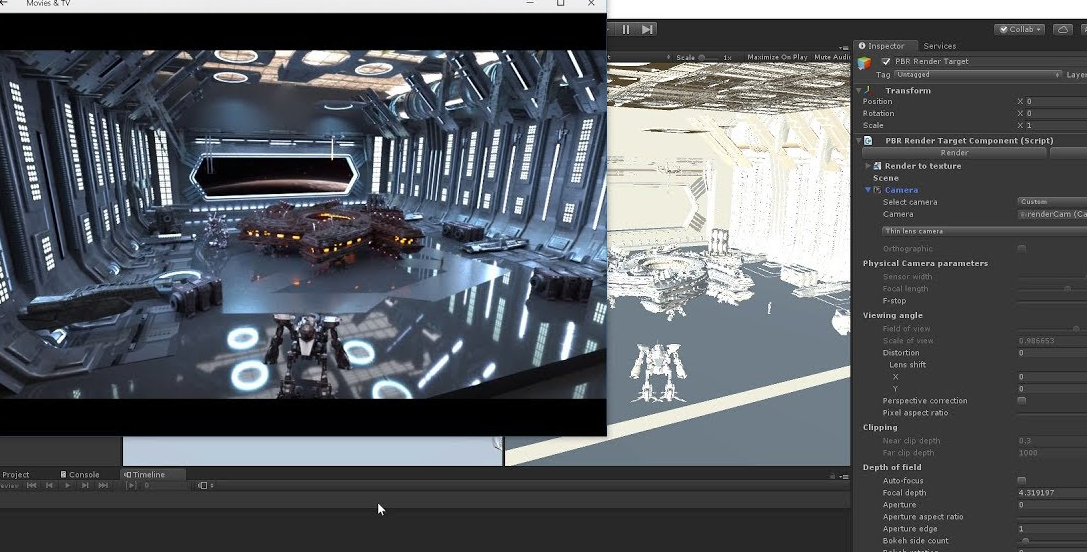


그림 6 : OctaneRender in Unity

1.2.2 Ray-Tracing in HDRP/DXR

RTX 시리즈에서 Ray-Tracing을 HW 자체적으로 지원하기 시작하고, DXR API가 등장함과 동시에, 이를 많은 엔진들이 차용하여 하이브리드 렌더링을 구현하기 시작했다. 이는 Unity에서 고품질의 실시간 렌더링을 지향하는 HDRP 또한 마찬가지다. 하지만 이는 렌더링의 패러다임을 바꾸는게 아닌, 단순히 기존의 Rasterization 기반 구조에서의 품질/성능 향상을 위해 부가적으로 DXR을 사용하는 것 뿐이다. 이는 현 시점의 GPU의 한계를 명확하게 보여주는 예시이다.

또한 이는 현재 Exprimental 버전이라서 안정적이지 않으며, HDRP 전용 메터리얼(재질)만 지원하므로 접근성 + 확장성에서도 한계가 명확하게 존재한다.

1.2.3 비교

OctaneRender는 메터리얼이 Standard를 기준으로 지원하고, HDRP는 HDRP메터리얼만 지원한다. Unity에서 지원하는 것들은 Standard, HDRP, URP 메터리얼들이 존재하는데, HDRP의 경우에는 굉장히 복잡하므로, 추후에 기능확장시 구현하는 것이 합당하다. 표 1은 1.2.1,1.2.3에서 언급한 내용을 종합적으로 비교하기 위해 정리한 내용이다. ***RadianceGrabber***는 해당 작품을 의미한다.

표 1 : OctaneRender, HDRP/DXR, RadianceGrabber(해당 작품) 비교

|  |  |  |  |
| --- | --- | --- | --- |
|  | 목적성 | 프레임워크/메터리얼 | H/W |
| OctaneRender | 실시간 렌더링 보조  레이 트레이싱 렌더 | Standard 메터리얼 | NVidia GPU |
| HDRP/DXR | 레이 트레이싱 렌더 | HDRP 런타임/메터리얼 | DXR 지원 H/W |
| ***RadianceGrabber*** | 레이 트레이싱 렌더  영상 제작 | Standard, URP 메터리얼 | NVidia GPU |

## **개발 목표**

기존의 존재하는 상용엔진/DCC에서 돌아가는 사실적인 명암을 가진 이미지를 생성하는 것, 이를 연속적으로 생성한 것을 바탕으로 동영상을 생성하는 것을 기능적인 목표로 가진다.

여기에 더불어, OctaneRender와의 차별성을 두기 위해서, Multi-GPU를 사용하여 성능 상의 이점을 가지며 계산하는 기능, 보다 현실적인 방법으로는 2개의 GPU를 사용시 150%~170%의 성능을(시간 단축) 낼 수 있는 기능을 성능상의 목표로 가진다.

## **팀 역할 분담**

역할 분담은 최대한 쉽게 나눌 수 있는, 각자 작업의 의존성이 적은 것들로 나누었다. 그렇지만 Unity 렌더링 환경의 세팅과, 광선 추적 모델이 먼저 어느 정도의 결과가 나온 후 구현에 들어가야 하는 부분에 있어서는 의존성이 존재한다. 표 2는 의존성을 최대한 줄인 팀원 별 역할 분담을 정리한 표이다.

표 2 : 팀원 별 역할 분담

|  |  |  |  |
| --- | --- | --- | --- |
|  | 김수혁 | 김한상 | 정지윤 |
| 자료수집 | 광선추적, CUDA | CUDA | Unity 렌더링 환경 |
| 설계 | 광선 추적 계산 모델 설계  (알고리즘 선택) | CUDA 기반의  Ray Tracing 계산 설계 | Unity 환경 데이터 수집 |
| 구현 | host/device 각각의 Foundation 구현 및  CUDA 기반 광선추적 구현 | | Unity Plugin의 형태로 환경 데이터 수집 및 UI |
| 테스트 | CPU/GPU 성능 별, 씬 복잡도에 따른 성능 테스트 | | |

## **개발 일정**

기존의 알고리즘들에 대한 학습이 필요하므로, 학습 단계와 설계 단계를 같이 진행하고, 동시에 구현까지 진행한다. 본격적인 구현은 1~2월 간에 진행하며, 뒤의 테스트 및 데모의 경우 본격적인 구현이 끝나는 3월로 위치시켰다. 표 3은 개발 일정을 간트 차트의 형태로 나타낸다.

표 3 : 개발 일정, 2019.12 ~ 2020.09

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 12월 | 1월 | 2월 | 3월 | 4월 | 5월 | 6월 | 7~9월 |
| 조사 및 학습 |  | |  |  |  |  |  |  |
|  | |
|  | |
| 계산 모델 설계  SW 설계 |  | |  |  |  |  |  |  |
|  | |
|  | |
| 구현 |  |  | | | |  |  |  |
|  | | | |
|  | | | |
| 테스트 및 데모 |  |  |  |  | | |  |  |
|  | | |
|  | | |
| 문서화 및 발표 |  |  |  |  | | | |  |
|  | | | |
|  | | | |
| 최종보고서  작성 및 발표 |  |  |  |  |  |  |  |  |
|  |
|  |

## **개발 환경**

Visual Studio는 안정성을 위해 2017 버전을 사용하며, Unity 역시 안정성을 위해 2019 버전을 사용한다. PC의 경우에는 PC1에서 일반적인 구현을 하고, PC2는 테스트 및 성능 측정 용도로 사용된다. 표 4는 개발 언어, IDE, HW를 나타낸다.

표 4 : 개발 언어 및 개발 HW

|  |  |  |
| --- | --- | --- |
| 개발 언어/환경 | C++, CUDA | Visual Studio 2017, NVidia CUDA Toolkit 10.1 Update 2 |
| C# | Unity 2019 |
| 개발 기준 HW | PC1 | I7-4790 / DDR3 16GB / GTX 970 |
| PC2 | I7-6700 / DDR4 32GB / GTX-1070 |

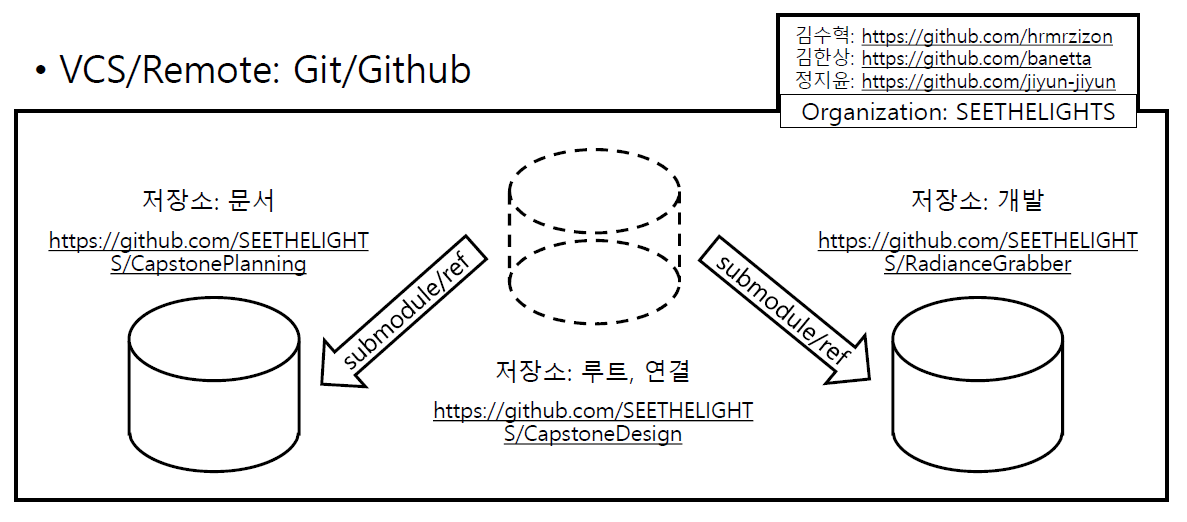


그림 7 : Git 저장소 세팅

Git 저장소는 문서화 레포지토리와 실제 소스가 들어가는 레포지토리를 따로 구성한 후, submodule 기능을 사용하여 서로 연결시켰다. 그래서 저장소의 갯수는 총 3개로 구성되며, 실질적인 저장소는 2가지로 구성되어 하나의 저장소는 프록시의 역할을 수행한다. 또한 버전 관리의 용도로도 사용될 수 있다. 그림 7은 3가지의 저장소의 구성을 한눈에 볼 수 있다.

# **본론**

## **개발 내용**

“Unity에서의 GPGPU 기반 광선 추적 렌더러”가 수행하는 역할은, 상용 엔진에서 에디팅을 끝마친 후, 해당 세팅에 맞추어 사용자가 한 프레임 혹은 여러 프레임의 이미지 생성을 요청하게 되면, 이를 수행하여 결과를 돌려주는 것이다. 이를 Unity에서 수행하기 위해서는, Unity의 렌더링 데이터를 수집하여 이를 정리하여, C++/CUDA로 구현된 광선 추적기 바이너리가 들어있는 DLL을 참조해 정리한 데이터를 파라미터로 넘겨주면서 실행을 시켜주어야 한다. 실행 시, 동시에 2차원 텍스쳐를 참조하는 포인터를 넘겨, 해당 텍스쳐에 결과를 저장하게 한다. 그렇다면 연속적으로 텍스쳐를 UI에서 보여주면서 사용자는 점점 업데이트 되는 모습을 볼 수 있다. 이 과정을 그림 8에서 시퀀스 다이어그램으로 나타내었다,

그림 9에서는 동작을 더 자세하게 묘사하였다. 그림 9의 렌더링 데이터 수집 모듈에서는 프레임의 개수에 대한 묘사가 나오는데, 이는 중요한 기능인 동영상을 처리하기 위해 중요하게 다뤄지는 부분이다. 그리고 Ray-Tracing에 대한 부분들이 더 자세히 묘사되었다. 가운데 unmanaged memory 구조를 가진 C++로 구현된 광선 추적 모듈 안에 일반적인 Ray-Tracing의 과정이 들어있는데, Ray-Tracing을 본격적으로 처리하기 전에 BVH 같은 탐색의 비용을 줄일 수 있는 가속 데이터 구조가 반드시 필요하다. 이는 많은 수의 Ray를 샘플링 하기 때문에 필수적인 부분과도 같다. BVH를 만든 이후에는 광선들을 카메라부터 시작하여 정해진 횟수만큼 광원에 도착할 때까지 계속 추적한다. 이 탐색을 GPU에서 하나의 쓰레드에 시켜서, 한꺼번에 픽셀의 처리를 할 수 있도록 한다.

이어지는 하위 항목에서는 Unity 상에서의 스크립트 구현과 스크립트를 통해 실행되는 광선 추적기 모듈의 구현과 설계 그리고 광선 추적기로 생성된 이미지의 품질 평가 방법에 대하여 설명한다.

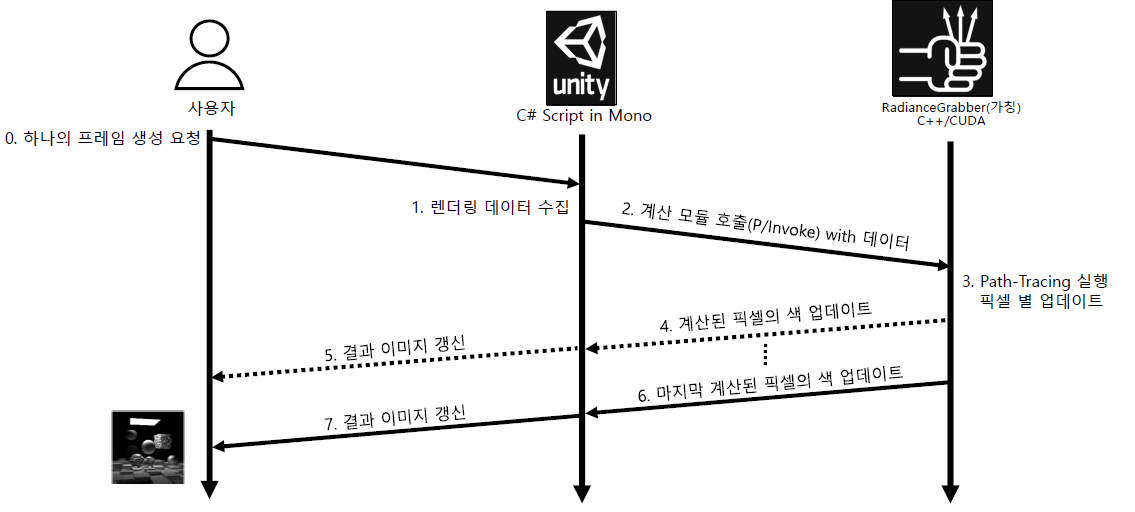


그림 8 : 시스템 수행 시나리오

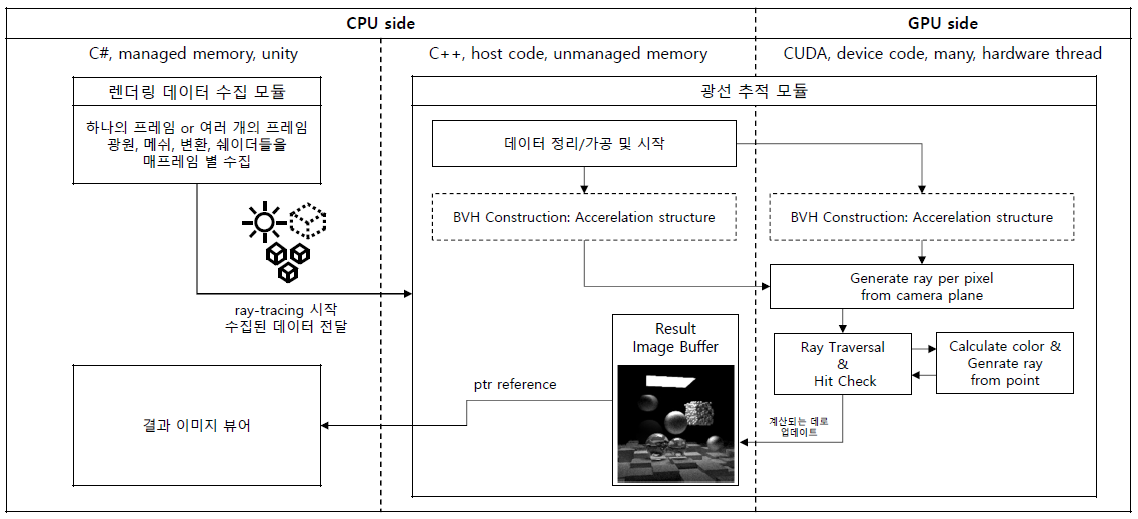


그림 9 : 시스템 구성도

2.1.1 Unity 런타임 상의 래퍼

이는 Unity에서 기능을 제공하기 위해 직접 C#을 이용하여 작성한 일종의 Front-End 모듈이다. 가장 중요한 기능은 C++/CUDA로 작성된 광선 추적 모듈에 넘겨야 할 데이터들을 제공해주는 부분인데, 이는 Unity에서 노출된 C# 기반의 데이터를 광선 추적 모듈의 정의된 형식, ABI에 따라서 넘겨준다. 광선 추적 모듈에 넘기는 정의된 데이터의 종류는 총 8가지이다.[[2]](#footnote-2) 표 5에서 이를 볼 수 있다.

표 5의 정보들을 Unity 에디터 상에서 C#을 통해 자체적으로 Serialization을 한 다음, 광선추적기 모듈에 넘겨주면서 한 프레임에 대한 계산을 실행시킨다. 그리고 Unity에서 환경을 구성하면서 Ray-Tracing 결과를 미리 보고 싶은 경우 한 프레임을 점진적으로 보여주어야 한다. 이때는 결과를 저장할 2D 텍스쳐를 생성한 후, 텍스쳐 포인터와, 함수 포인터를 넘겨주어 광선 추적 모듈에서 이를 호출해 다시 텍스쳐를 그리거나, 에디터상에서 주기적으로 다시 그린다. 그렇기 때문에 실시간 업데이트를 볼 수 있는, 샘플링을 픽셀별로 똑같이 계산하는 모드와, 이와 상관없이 빠르게 계산하는 모드 두가지의 구현이 필요하다.

Unity상의 쉐이더의 경우, 해당 쉐이더의 픽셀 쉐이더 함수의 구현에 따라서 Ray-Tracing에서 구현할 것 또한 달라진다. 그래서 모든 쉐이더에 맞추어 구현을 할 수 없고, 현실적인 제약 조건으로는 Unity에서 제공하는 Standard, Standard(Specular Setup), URP(LWRP)의 Lit 쉐이더를 Ray-Tracing과 연관시켜 구현할 계획이다. Unity에서의 환경 조성 시 Standard가 기본이기 때문에, 컨텐츠 제작의 용도로 사용되기 적합할 것으로 보인다.

영상 녹화 기능의 경우 OpenCL에서 C#형태의 라이브러리를 지원하고, OpenCL 자체에서 영상 녹화 기능을 프레임 단위로 제공하기 때문에, 이를 Unity 에디터 상에서 OpenCL 라이브러리를 사용하여 영상을 녹화한다. 에디터 상의 영상 녹화 기능의 UI는 간단하고 직관적으로 쉽게 구현할 수 있다. 대부분의 경우와 같이 IMGUI를 사용하여 제공한다.

표 5 : 입력 데이터 명세

|  |  |  |  |
| --- | --- | --- | --- |
| 데이터 | 세부 데이터 | | 설명 |
| 카메라 | 위치, 회전, 정의된 투영 행렬 | | 3D 공간에서 물체를 그리기 위한 기준의 주체 |
| 스카이박스 인덱스 | |
| 스카이박스 | Cubemap | 육면체 | 아무런 물체가 그려지지 않을 때, 기본으로 그려지는 배경 |
| Procedural | 임의의 Shading |
| Paranomic | 구의 표면을 펴놓은 사각형 |
| 광원 | Directional | 하나의 방향의 광원/배경 | 빛의 원인,  모양에 따라서 광선의 추적 시, 판정이 달라짐 |
| Point | 구 형태의 광원 |
| Spot | 원뿔 형태의 광원 |
| Area | 표면(주로 사각형)형태의 광원 |
| 메쉬 | 정점 버퍼 데이터 | 3차원상의 위치, 법선, 텍스쳐 접근을 위한 재매개화된 좌표 등의 데이터를 저장.  Graphics API에서 제공 | 하나의 물체를 나타내는 기하학적 정보들의 집합. |
| 인덱스 버퍼 데이터 | 정점 버퍼의 인덱스로 구성되어, 하나의 면(삼각형, 사각형, ..)을 이루게 하는 정보  Graphics API에서 제공 |
| Bindpose 행렬 데이터 | 초기 세팅된 위치에서 “뼈”의 위치, 회전 정보를 없에주는 행렬, 각 “뼈” 별로 저장함. |
| 메쉬 렌더러 | 메쉬 참조 인덱스 | 참조할 메쉬 인덱스, 하나만 존재 | 메쉬와 메터리얼을 참조하고, 위치,회전,크기 정보들을 가짐. |
| 메터리얼 참조 인덱스 | 참조할 메터리얼 인덱스,  여러 개가 존재 할 수 있음. |
| 동차 변환 행렬 | 위치, 회전, 크기 정보를 가지는  변환 행렬 |
| AABB | 메쉬를 감싸는 박스 형태의 모양. |
| 스킨메쉬 렌더러 | 메쉬 참조 인덱스 | 참조할 메쉬 인덱스, 하나만 존재 | 메쉬 렌더러의 데이터와 더불어, “뼈”들의 위치, 회전 정보를 가지고 있음. |
| 메터리얼 참조 인덱스 | 참조할 메터리얼 인덱스,  여러 개가 존재 할 수 있음. |
| 동차 변환 행렬 | 위치, 회전, 크기 정보를 가지는  변환 행렬 |
| AABB | 메쉬를 감싸는 박스 형태의 모양. |
| “뼈”의  위치, 회전 | 각 “뼈”의 위치와 회전 정보  스키닝 계산시 사용됨. |
| 텍스쳐 | 1D | 1차원 좌표로 접근 | 데이터를 1D,2D,3D로 저장하여 샘플링 할 수 있는 형태의 GPU 특유의 데이터 구조 |
| 2D | 2차원 좌표로 접근(주로 쓰임) |
| 3D | 3차원 좌표로 접근 |
| 메터리얼 | 쉐이더 | Unity에서 접근하는 프로그래밍 가능한 코드,  여기서는 Unity의 기본 Shader만 지원함. | 쉐이더와 그 쉐이더에 넘길 파라미터들을 가지고 있는 인스턴스 |
| 파라미터 | 해당 Shader에서 정의한 매개변수  고정된 쉐이더만 지원하기 때문에 이를 하드 코딩으로 지원. |

2.1.2 광선 추적기

광선 추적기는 구현에 앞서, 여러 방법에 대한 연구가 필요하다. 복잡한 것들은 여태까지도 연구가 진행되고 있기 때문에 경우에 맞는 방법에 대한 검색과 연구는 구현에 앞서 반드시 선행되어야 한다. 다만 현실적인 제약 조건, 특히 일정의 한계가 있을 수 있으므로 처음에는 전부 가장 기본적인 방법들만 검색과 연구가 이루어진 후, 이후에 성능과 품질을 올리기 위하여 여러 연구와 구현의 반복이 이루어질 수 있다.

2.1.2.1 알고리즘 선택 및 설계

알고리즘 선택 및 설계에서 고려해야할 것은 3가지가 있다. 첫번째로는 공간상의 물체들을 빠르게 추적하기 위한 자료구조를 선택하는 것이다. 일반적으로 Ray-Tracing에 사용되는 자료구조는 Bounding Volume Hierarchy(이하 BVH)라는 방법으로, 공간상의 물체를 특정 축을 기준으로 정렬한 후, 이를 트리의 anary에 따라서 묶어서 최종적으로 모두 이어진 트리 한 개를 가지게 되는 자료구조이다.

Ray-Tracing에 사용되는 BVH 자료 구조에서 성능을 얻기 위해 다양한 시도들이 존재했다. 최근의 연구에서는 내부의 연결된 노드들을 두가지 형태로 나누는 방법이 있었다 [1]. 해당 연구에서는 GPGPU 상에서의 구현을 목적으로 하기 때문에 GPGPU 기반의 BVH 구현에 대한 연구도 존재하였다 [2] [3].

다음으로 연구해야 할 알고리즘은 Path-Guiding 알고리즘이다. Ray-Tracing의 고 비용, 고 품질 방법인 Path-Tracing 방법 중 가장 고전적이고 naïve한 방법이나 쉽게 구현할 수 있는, 무작위로 방향을 정하여 광선을 쏘아 해당 표면으로 이동하여 이를 반복하고, 빛을 발견하면 해당 색을 recursive하게 계산하는 방법은 치명적인 단점이 있다.

그림 10에서의 표에서는 광원을 발견하지 못해 luminance가 0인 전혀 필요 없는 샘플링이 Path Tracing, BDPT는 대부분을 이룬다. 전 문단에서 언급한 방법이 Path Tracing에 해당한다. 옆의 MMLT라는 것과 비교하면 현저히 차이가 나는 것을 알 수 있다. 즉, 유효한 빛의 경로로 샘플링을 하게 하는 것, 즉 Path-Guiding이 Path-Tracing에 있어 중요한 방법으로 떠오르게 된다. 그래서 Bidirectional Path Tracing(이하 BDPT) [4] 같은 광원에서와 표면에서의 경로를 찾는 방법이 고안되었고, 지금은 이를 활용하여 다양하게 사용한다. 또한 BDPT와 더불어 효율적인 경로 샘플링을 통계적 샘플링 모델을 활용하여 고안한 Metropolis Light Transport(이하 MLT)이 존재한다 [5]. MLT는 1997년에 발표된 기법으로, 이 이후에도 많은 변종들: PSSMLT, MMLT, RJMLT 등 이 연구되었다 [6] [7] [8].

이러한 Path-Guiding방법은 샘플링 비용을 현저하게 줄일 수 있어, 적은 개수의 샘플링 비용 제약이 존재할 시 noise 제거와 이미지에서의 빛의 전반적인 질이 올라간다. 그림 9에서 이를 볼 수 있다.

마지막으로는 noise 제거 알고리즘의 연구이다. MLT는 이를 최대한 막기 위한 방법으로 알려졌지만 근본적인 해결책은 아니다. 앞서 말한 것들 보다는 덜한 노력이 필요하지만 방법의 연구는 반드시 필요하다. 다만, 그림 10에서 보이는 것처럼 이들은 흔히 알려진 post-processing(cross bilateral filter, …) 방법을 통하여 어느 정도 제거될 수 있을 것으로 보인다.

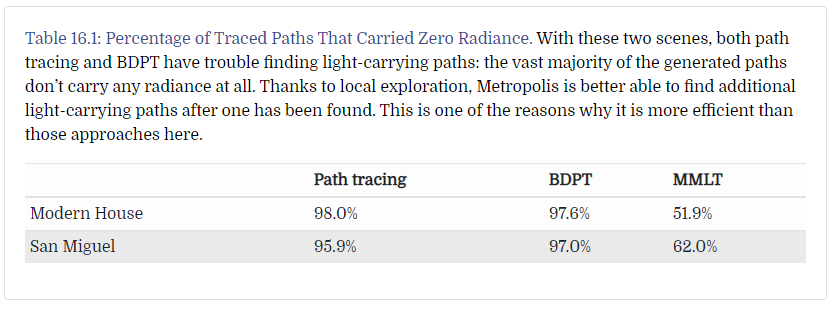


그림 10 : 0의 에너지를 발견한 경우, PT vs BDPT vs MMLT ([pbr-book.org](http://www.pbr-book.org/3ed-2018/Light_Transport_III_Bidirectional_Methods/Metropolis_Light_Transport.html))

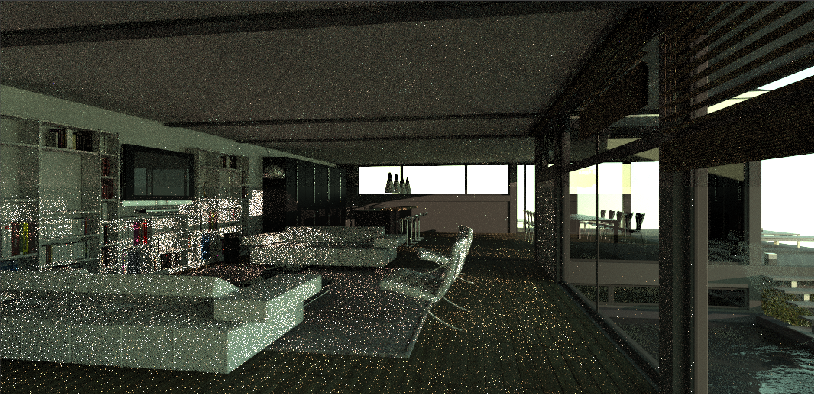


그림 11 : MMLT(위 절반), Path-Tracing(아래 절반), 같은 시간의 계산결과 ([pbr-book.org](http://www.pbr-book.org/3ed-2018/Light_Transport_III_Bidirectional_Methods/Metropolis_Light_Transport.html))

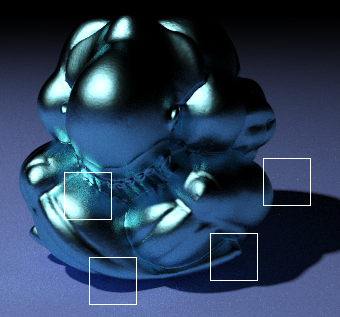
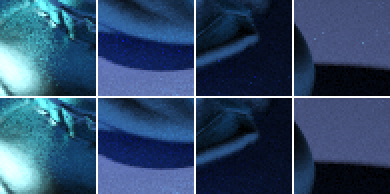
 

그림 12 : PT의 결과, noise(좌측), 확대한 이미지(우측/위), variance reduction한 결과(우측/아래)

2.1.2.2 구현

구현은 unmanaged memory상에서 CPU와 GPU를 활용하기 위하여, C++과 CUDA를 함께 사용하여 구현해야 한다. 다만 구현에 있어 퍼포먼스가 가장 중요한 요소이기 때문에, OOP를 크게 활용하지 않을 가능성이 높다. 미리 정의된 Graphics API의 데이터를 받아오기 때문에 이들을 처리하기 위해서는 DirectX, OpenGL과 같은 라이브러리로 처리를 직접 해주어야 한다. CUDA에서는 이를 처리하는 방법들을 지원하기 때문에 CPU레벨에서 계산하지 않을 시 직접 메모리 공간을 처리해주지 않아도 된다. 전체적인 동작은 그림 12에서 확인할 수 있다. HW기준으로 표기하였다.

순서는 기존에 Unity/C# 레벨에서 넘겨준 데이터들을 버퍼를 할당하여 복사한 후, 먼저 메시를 처리해 주어야 한다. 앞서 언급한 스킨 메시/스킨 메시 렌더러의 경우, 각 정점의 위치를 “뼈”의 가중치와 인덱스를 활용하여 존재하는 bindpose matrix를 가중합한 행렬을 통해 기존 정점의 위치에 저장된 “뼈”와 관련된 위치와 회전을 제거해주고, 그 다음 각 “뼈”의 현재 위치와 회전을 적용하여, 정점의 위치를 재계산한다. 이 과정에서 CUDA를 활용하여 계산해야 한다. 원래의 경우에도 GPGPU로 계산하기 때문에 기존의 실시간 렌더링에서 하는 것과 동치이다. 테셀레이션은 Unity에서 제공하는 Standard, URP(LWRP)에서 지원하지 않기 때문에 초기 버전에서는 제외한다. 만약 구현한다면 지금 언급한 스키닝과 같은 타이밍에 처리된다.

다음은 Path-Tracing에 앞서 BVH를 생성해야 한다. BVH는 물체들을 트리의 형태로 저장하여, 일반적으로 O(n)의 시간 복잡도를 가지는 것을, O(logn)의 시간을 가지도록 하는 오브젝트에 따라 트리를 구성하는 방법이다. 이 부분은 많은 자유도가 존재하기에, 하나의 정확한 타협점을 찾기 위해서는 여러 방법, 여러 환경에 따른 많은 프로파일링이 필요하다.

하지만 데이터가 정해져 있으므로 그 부분에 맞춰야 할 필요가 있다. 일반적으로 여러 논문에서는 BVH 구성 시 리프 노드에는 폴리곤 (인덱스 버퍼의 정보로 구성된 면(Plane))의 참조 정보를 넣어 놓는다. 하지만 여기에서는 조금 다른 방식으로 구성된다. 각 물체들의 위치, 회전 정보와, 메쉬는 같이 구성되어 있지 않고, 참조 형식으로 분리되어 있다. 그렇다면, 오브젝트들의 위치, 회전을 저장한 것과, 각 메쉬들을 따로 BVH로 구성하여 2 Layer 구조를 가지게 하는 방법이 가장 처리를 덜하는 방법이다. 하지만 위와 같은 방법은 충분히 달라질 수 있다. BVH 생성시에는 꽤나 많은 코스트가 들지만, 그 뒤의 광선을 가지고 BVH를 차례차례 타고 들어가는 Ray-Traversal의 성능이 가장 중요하기 때문이다. 어디까지나 이는 구현 반복의 1단계라고 말할 수 있다.

메쉬의 개수가 굉장히 많다면, VRAM ~ RAM 간의 메모리 통신을 하지 않고 GPGPU를 통해 처리하는 것도 하나의 방법이다. 하지만 대부분 적은 경우가 많기에 CPU에서 처리하는 것이 더 빠를 것으로 예상된다.

Path-Tracing 계산 방법은 크게 두가지로 나뉜다. 하나는 무작위로 경로를 찾는 방법, 하나는 MLT를 사용하여 더 high-contribution인 경우에 더 높은 확률로 경로를 결정하는 방법이다. MLT를 사용하는 방법은 연구가 필요하기에 맨 처음에는 단순한 무작위 경로 찾기부터 구현한다. 또한 레퍼런스 이미지 생성에는 완전한 균일한 샘플링 방식으로 수많은 샘플링 횟수를 들여야 하기 때문에 MLT의 평가를 위해 필요하다.

위의 언급한 반복의 의미는, 기본적인 틀을 위해 한번 위의 언급한 순서대로 쭉 구현한 다음, 변경점이 생길 시에 부분부분 바꾸어 가면서 구현하는 것을 의미한다. BVH의 구성의 경우도 맨 처음에는 데이터에 맞추어 구성하나, MLT의 구현 이후 시간이 허락한다면, 여러 논문의 기법들을 참고하여 다양한 방법을 찾을 수 있다. 실질적인 구현의 순서는 그림 11과 같다. 최소한의 틀을 구현하는 것이 MLT 구현 까지고, 그 다음 부터는 추가적인 구현으로 점선으로 표기하였다.[[3]](#footnote-3)

Unity Wrapper 구현

BVH 구축/탐색 구현

Path-Tracing 구현

MLT 구현

Mesh Skinning 구현

MLT 변종 구현

BVH 변종 구현

그림 : 구현 순서도

2.1.3 광선 추적기로 생성된 이미지의 검증 방법

마지막으로 이 SW에서 중요하게 판단되어야 할 것은, 비교적 짧은 시간 내에, 사람의 눈으로 인식하여 만족할 만한 빛의 시뮬레이션을 나타내는 것이다. 이를 엄밀하게 판단하기 위해서는 위 요소들을 메트릭으로 나타내야 할 필요가 있다.

첫번째 요소는 생성 시간을 측정하여 나타낸다. 이는 다음에 언급할 두번째 요소보다 굉장히 간단하다. 두번째 요소인 사람의 눈으로 인식하여 만족할 만한 것은 균일하게 랜덤으로 경로를 찾는 Path-tracing 방법으로 수많은 샘플링을 들여 만든 이미지(Reference Image, Ground Truth)와 현실적으로 가능한 샘플링 수를 적용하여 만든 이미지(Synthesized Image)를 IQA[[4]](#footnote-4) 기법으로 메트릭을 계산한다 [9]. 에 따르면, 수많은 IQA 기법 중 MS-SSIM 혹은 SC-QI 기법의 사용을 추천한다. 이는 Reference Image의 노이즈를 고려해보았을 때, 가장 나은 기법이라고 말한다.[[5]](#footnote-5)

## **문제 및 해결방안**

Ray-Tracing을 구현 시 가장 까다로운 점은, 알고리즘을 이해하고, 이에 맞게 구현하는 것이다. 특히 supplemental code를 제공하지 않는 경우에는 검증하는 과정이 비교적 까다로운 편이다. 특히 최신의 기법을 사용할 경우에는 자료 자체가 존재하지 않기 때문에 노력이 배로 들어가게 된다.

BVH의 변종들의 경우에는 전부 Traversal 코드들을 지원하지만 Construction 코드는 지원하지 않는 경우가 특히 많다. 더군다나, 대부분 퍼포먼스를 일정한 기준으로 측정하기 위해 BVH를 구성한 후 이를 기준으로 다시 Reconstruction 과정을 거치기 때문에 코드를 지원한다고 해도 쓸모 없는 경우가 많다. MLT의 변종들의 경우에는 대부분 방법론에 관한 것들이기 때문에 사실상 코드를 지원하지 않는 경우가 많다.

이에 대한 해결 방안은, 쉽게 Ray-Tracing을 구현하기 위한 튜토리얼인 Peter Shirley의 “Ray Tracing in ~” 시리즈가 있다 [10] [11] [12]. 이를 통해 CPU 상의 구현을 쉽게 따라하고 이해할 수 있다. 또한 첫번째 시리즈를 CUDA로 구현하기 위한 튜토리얼인 “Accelerated Ray Tracing in a one Weekend”이 존재한다 [13]. CUDA는 하드웨어 의존적인 코드가 필요하기 때문에 이에 대한 것들을 덧붙인 튜토리얼이다. 그 다음 복잡한 Ray-Tracing의 원리부터 코드까지 자세히 설명해 놓은 저서인 Physically Based Rendering에서 [14] 참고자료로 쓰이고, 꽤나 많은 유지보수를 거치고 있고 오픈소스인 PBRTv3[[6]](#footnote-6)이라는 Ray-Tracing Renderer가 있다. 최신의 방법들은 없지만 기본적으로 널리 알려진 방법들은 전부 구현되어 있다. BVH의 구축의 경우에는 Surface Arera Heuristic외에도 여러 방법들이 구현되어 있으며 MLT는 MMLT의 구현체를 지원한다. 이를 CUDA의 코드에 적합하도록 구현하는 것이 가장 중요한 포인트다. 또한 그 이것 외에도 CUDA로 구현된 Ray-Tracer는 얼마든지 존재하니, 이를 참고하여 구현에 박차를 가할 수 있겠다.

CUDA를 활용하는 프로그래밍은 일반적인 프로그래밍만을 접한 프로그래머들에게는 전혀 직관적이지 않다. 기본적으로 CPU, RAM 간의 관계만을 신경 써야 했던 것과는 달리, CPU, RAM, GPU, VRAM 등 다른 방법을 가진 프로세서와 저장 매체들 서로 간의 동기화를 신경 써야 하기 때문에 더욱더 복잡하다. 또한 GPU에서 돌아가는 프로그램의 성능을 최대화하기 위해서는 HW적인 특성까지 고려해야 하기 때문에 어려운 작업이라고 할 수 있다. 이것들을 포함하여 CUDA를 활용한 프로그램의 구현 전략을 담은 책들을 통해 기본적인 GPGPU 구현의 줄기를 잡을 수 있었다 [15] [16].

## **시험 시나리오**

해당 SW의 테스트 시나리오는 높은 레벨에서의 기능들을 테스트하는 방법을 정의한다. 구체적인 방법은 명시적으로 제시되지 않을 수 있으며, 경우에 따라서 바뀔 수 있음을 암시한다.

아래 시나리오는 이미지의 품질을 테스트하는 시나리오다.

1. Unity에서 테스트에 필요한 씬을 세팅한다.
2. 무작위로 탐색하는 Path-Tracing으로 몇 만개의 샘플링 횟수를 사용하여 레퍼런스 이미지를 생성한다.
3. 테스트할 기법을 선택하여 세팅한 후, 현실적으로 계산할 수 있는 샘플링 수만큼 설정 후 Path-Tracing을 실행한다. (사용자들에게는, 샘플링 수는 디폴트 값이 정해져 있다.)
4. 이미지 생성 시간과 레퍼런스 이미지, 생성된 이미지를 IQA 기법으로 계산하여 결과를 도출한다.

2.1.3에서 설명한 것들을 테스트 시나리오로 구체화한 것이다. 0번, 1번은 대부분 고정된 수의 레퍼런스 이미지와 테스트 씬을 만들고, 2,3번의 경우를 반복하여 통계를 낸 후 실질적으로 사용자에게 노출할 방법들을 결정한다.

## **상세 설계**

설계의 과정은 클래스 다이어그램부터 시작하여 메소드 형식을 나타냄으로써 마무리한다. 그림 14의 위쪽 클래스들은 입력 데이터를 처리, Unity 상에서의 GUI, 계산 과정을 Unity와 동기화를 처리하고, 마샬링을 한 여러 데이터들 아래의 클래스들은 C++ 모듈들로 실질적인 계산, Path-Tracing, BVH의 역할을 하는 클래스들이 존재한다.

텍스트, 지도이(가) 표시된 사진

자동 생성된 설명

그림 : 클래스 다이어그램, 위쪽은 데이터 처리 모듈과 아래쪽은 계산 모듈

표6에서 데이터 처리 모듈의 메소드들에 대한 설명을 볼 수 있는데, 기능에 대한 관리를 하는 것과 마샬링을 하는 부분들이 대부분이다. *TaskController* 클래스는 C++/CUDA 모듈에서 구현된 함수를 동적으로 링킹하여 해당 함수를 호출하고 각 작업들을 관리한다. 접두사로 *Chunk* 가 붙은 클래스들은 전부 데이터 마샬링을 위한 것들로, 하나의 렌더링 객체와 일대일 관계를 가진다. 또한 고정 길이의 형태를 가지며, 데이터가 여러 개인 경우에는 C/C++에서 처럼 포인터를 이용하여 접근한다.

표 : 데이터 처리 모듈 클래스들의 메소드

|  |  |  |
| --- | --- | --- |
| 클래스 | *TaskController* | |
| 설명 | Unity plugin과 C++/CUDA로 구현된 함수 사이에서 계산 태스크들을 제어해주는 클래스 | |
| 메소드 형식 | public int StartSingleFrameGeneration(FrameRequestOption opt, UnityRuntimeData runtimeData) | |
| 파라미터1 | FrameRequestOption opt | 계산에 대한 옵션 값, 사용자에게서 입력받음 |
| 파라미터2 | UnityRuntimeData runtimeData | Unity 런타임에서의 렌더링 데이터 |
| 반환 값 | int | 시작한 TaskID, 멈추는데 사용 |
| 설명 | 하나의 프레임을 만들기 위한 계산 시작. C++/CUDA 로 구현된 함수를 호출.  비동기적 작업으로, FrameRequestOption 에 위임자를 통해 계산 상태 정보를 전달. | |
| 메소드 형식 | public bool StopSingleFrameGeneration(int taskID) | |
| 파라미터1 | int taskID | 멈출 작업의 taskID |
| 반환 값 | bool | 해당 taskID를 실질적으로 멈추었는가? |
| 설명 | 하나의 프레임을 만들기 위한 계산을 멈추게함. C++/CUDA 로 구현된 함수를 호출함. | |
| 메소드 형식 | public int StartMultiFrameRecord(MultiFrameRequestOption opt, UnityRuntimeData runtimeData) | |
| 파라미터1 | MultiFrameRequestOption opt | 계산에 대한 옵션 값, 사용자에게서 입력받음 |
| 파라미터2 | UnityRuntimeData runtimeData | Unity 런타임에서의 렌더링 데이터 |
| 반환 값 | int | 시작한 TaskID, 멈추는데 사용 |
| 설명 | 여러 프레임을 만들기 위해 데이터를 누적시키기 위한 기본 정보들을 입력받아 준비함. 실질적인 누적은 RecordOneFrame 메소드에서 실행 | |
| 메소드 형식 | public bool StopMultiFrameRecord(int taskID) | |
| 파라미터1 | int taskID | 멈출 작업의 taskID |
| 반환 값 | bool | 해당 taskID를 실질적으로 멈추었는가? |
| 설명 | 단순하게 프레임 별 누적을 하던 보조적인 메모리 버퍼들을 해제함. | |
| 메소드 형식 | public void RecordOneFrame() | |
| 설명 | 해당 시간의 렌더링 데이터들을 수집하여 저장, 이전에 저장된 것들에 누적됨. | |
| 메소드 형식 | public int StartMultiFrameGeneration() | |
| 반환 값 | int | 시작한 TaskID, 멈추는데 사용 |
| 설명 | 누적시킨 렌더링 데이터들을 통하여 비동기적 계산을 수행함. C++/CUDA 로 구현된 함수를 호출함. | |
| 메소드 형식 | public bool StopMultiFrameGeneration(int taskID) | |
| 파라미터1 | int taskID | 멈출 작업의 taskID |
| 반환 값 | bool | 해당 taskID를 실질적으로 멈추었는가? |
| 설명 | 여러 프레임을 만들기 위해 계산하던 것을 멈춤. 비동기적으로 멈추는 것을 수행. | |
| 클래스 | *UnityRenderingDataBuilder* | |
| 설명 | Unity상의 렌더링 데이터를 C++/CUDA에서 사용할 수 있도록 하는 변환을 수행하는 클래스 | |
| 메소드 형식 | public bool ConvertRenderingData (UnityRuntimeData runtimeData, out UnityFrameInput inputData) | |
| 파라미터1 | UnityRuntimeData runtimeData | Unity 런타임에서의 렌더링 데이터 |
| 파라미터2 | out UnityFrameInput inputData | 하나의 프레임을 생성하기 위한 변환된 정보 |
| 반환 값 | bool | 해당 taskID를 실질적으로 멈추었는가? |
| 설명 | Unity 런타임에서 제공하는 렌더링 데이터들을 마샬링하여 넘길 수 있는 메모리 형태로 만들어줌. 기존의 렌더링 데이터의 주소 공간이 MBE 안에서의 주소공간인데 비해, out 파라미터로 나가는 inputData는 Unmanaged 주소 공간에 존재함. | |
| 클래스 | *MarshalUtil* | |
| 설명 | 마샬링을 위한 유틸리티 클래스 | |
| 메소드 형식 | public static void GetIndexAndAppendIfNotExist<ListItem>(List<ListItem> itemList, ListItem item); | |
| 파라미터1 | List<ListItem> itemList | 존재하는지 검사할 제너릭 리스트 |
| 파라미터2 | ListItem item | 존재하는지 검사할 때 쓰이는 객체 |
| 설명 | List에 해당 객체가 존재하는지 검사하고, 없을 시 리스트에 추가시켜주는 메소드 | |
| 메소드 형식 | public static void ConvertArrayToPtr<NativeChunk, ManagedObj>(ref IntPtr nativeArrayPtr, IList<ManagedObj> managedList, int itemCount, Func<IntPtr, ManagedObj, bool> setAction); | |
| 파라미터1 | ref IntPtr nativeArrayPtr | Unmanaged 주소 공간의 모든 객체가 마샬링된 (NativeChunk) 배열의 주소 |
| 파라미터2 | IList<ManagedObj> managedList | Managed 주소 공간의 제너릭 리스트 |
| 파라미터3 | int itemCount | 최대 객체의 개수 |
| 파라미터4 | Func<IntPtr, ManagedObj, bool> setAction | 객체를 변환하는 델리게이트, 각 하나의 제너릭 리스트 아이템마다 실행 |
| 설명 | Managed memory 주소 공간에 있는 IList<T> 인터페이스를 구현한 객체를 | |
| 메소드 형식 | public static int GetIndexAndAppendIfNotExist<ListItem>(this List<ListItem> itemList, ListItem item) where ListItem : UnityEngine.Object | |
| 파라미터1 | this List<ListItem> itemList | 검사할 제너릭 리스트, 자체적으로 사용 가능하도록 this 키워드를 붙여줌. |
| 파라미터2 | ListItem item | 검사할 객체, UnityEngine.Object로 한정됨. |
| 반환 값 | int | 리스트에서의 인덱스 |
| 설명 | 제너릭 리스트에 해당되는 객체가 있는지 검사하고, 없으면 추가하고 해당 객체가 있는 리스트의 인덱스를 반환한다. | |
| 클래스 | *RadianceGrabberEditorWindow* | |
| 설명 | Unity 내에서의 GUI를 지원하기 위한 클래스. | |
| 메소드 형식 | private void OnGUI() | |
| 설명 | 유틸리티 메소드 호출로 만들어지는 GUI 메소드로 Immediate GUI 를 사용 | |
| 메소드 형식 | private void Update() | |
| 설명 | 시간 혹은 샘플링 임계치에 따라서 GUI의 내용을 갱신하는 메소드 | |
| 메소드 형식 | private void DrawResultImage() | |
| 설명 | 이미지를 다시 그려주는 메소드, 색 변경 시 사용 | |
| 클래스 | *MeshChunk* | |
| 설명 | Mesh객체를 마샬링하기 위한 클래스 | |
| 메소드 형식 | public bool MarshalFrom(Mesh ms) | |
| 파라미터 | Mesh ms | Unity 렌더링 Mesh 객체 |
| 반환 값 | bool | 마샬링 성공 여부, True 성공, False 실패 |
| 설명 | Mesh 정보를 마샬링한다. vertex buffer, index buffer의 정보를 빼온다. | |
| 클래스 | *CameraChunk* | |
| 설명 | Camera객체를 마샬링하기 위한 클래스 | |
| 메소드 형식 | public bool MarshalFrom(Camera camera, List<Material> skyboxList) | |
| 파라미터1 | Camera camera | Unity 렌더링 Camera 객체 |
| 파라미터2 | List<Material> skyboxList | 카메라에서 참조하는 스카이박스를 저장하기 위한 리스트, 미리 할당된 리스트가 들어와야함. |
| 반환 값 | bool | 마샬링 성공 여부, True 성공, False 실패 |
| 설명 | Camera 정보를 마샬링한다. | |
| 클래스 | *LightChunk* | |
| 설명 | Light객체를 마샬링하기 위한 클래스 | |
| 메소드 형식 | public bool MarshalFrom(Light light) | |
| 파라미터1 | Light light | Unity 렌더링 Light 객체 |
| 반환 값 | bool | 마샬링 성공 여부, True 성공, False 실패 |
| 설명 | Light 정보를 마샬링한다. | |
| 클래스 | *Texture2DChunk* | |
| 설명 | Texture2D 객체를 마샬링하기 위한 클래스 | |
| 메소드 형식 | public bool MarshalFrom(Texture2D texture) | |
| 파라미터1 | Texture2D texture | Unity 렌더링 Texture2D 객체 |
| 반환 값 | bool | 마샬링 성공 여부, True 성공, False 실패 |
| 설명 | Texture2D 정보를 마샬링한다. 텍스쳐 압축으로 인한 인코딩 문제로, 초기 버전에서는 Rendering Pipeline(SM5.0이상)의 기능을 사용하여 압축없는 순수한 데이터로 변환한다. | |
| 클래스 | *MaterialChunk* | |
| 설명 | Material 객체를 마샬링하기 위한 클래스 | |
| 메소드 형식 | public bool MarshalFrom(Material material, List<Texture2D> textureList) | |
| 파라미터1 | Material material | Unity 렌더링 Material 객체 |
| 파라미터2 | List<Texture2D> textureList | Material에서 참조하는 Texture2D 를 저장하기 위한 리스트, 미리 할당된 리스트가 들어와야함. |
| 반환 값 | bool | 마샬링 성공 여부, True 성공, False 실패 |
| 설명 | Material 정보를 마샬링한다. Unity의 Material은 Dictionary 데이터 구조를 가지고 있다. 하지만 고정 길이의 데이터만 취급해야 하므로, Shader별로 파라미터를 뽑는 코드를 따로 작성하여 사용한다. | |
| 클래스 | *SkyboxChunk* | |
| 설명 | Skybox 객체를 마샬링하기 위한 클래스 | |
| 메소드 형식 | public bool MarshalFrom(Skybox skybox, List<Material> skyboxList) | |
| 파라미터1 | Skybox skybox | Unity 렌더링 Skybox 객체 |
| 파라미터2 | List<Texture2D> textureList | Skybox 에서 참조하는 Texture2D 를 저장하기 위한 리스트, 미리 할당된 리스트가 들어와야함. |
| 반환 값 | bool | 마샬링 성공 여부, True 성공, False 실패 |
| 설명 | Skybox 정보를 마샬링한다. | |
| 클래스 | *MeshRendererChunk* | |
| 설명 | MeshRenderer 객체를 마샬링하기 위한 클래스 | |
| 메소드 형식 | public bool MarshalFrom(MeshRenderer mr, List<Mesh> meshList, List<Material> maerialList) | |
| 파라미터1 | MeshRenderer mr | Unity 렌더링 MeshRenderer 객체 |
| 파라미터2 | List<Mesh> meshList | MeshRenderer 에서 참조하는 Mesh 를 저장하기 위한 리스트, 미리 할당된 리스트가 들어와야함. |
| 파라미터3 | List<Material> maerialList | MeshRenderer 에서 참조하는 Material 를 저장하기 위한 리스트, 미리 할당된 리스트가 들어와야함. |
| 반환 값 | bool | 마샬링 성공 여부, True 성공, False 실패 |
| 설명 | MeshRenderer 정보를 마샬링한다. | |
| 클래스 | *SkinnedMeshRendererChunk* | |
| 설명 | SkinnedMeshRenderer 객체를 마샬링하기 위한 클래스 | |
| 메소드 형식 | public bool MarshalFrom(SkinnedMeshRenderer smr, List<Mesh> meshList, List<Material> maerialList) | |
| 파라미터1 | SkinnedMeshRenderer mr | Unity 렌더링 MeshRenderer 객체 |
| 파라미터2 | List<Mesh> meshList | SkinnedMeshRenderer 에서 참조하는 Mesh 를 저장하기 위한 리스트, 미리 할당된 리스트가 들어와야함. |
| 파라미터3 | List<Material> maerialList | SkinnedMeshRenderer 에서 참조하는 Material 를 저장하기 위한 리스트, 미리 할당된 리스트가 들어와야함. |
| 반환 값 | bool | 마샬링 성공 여부, True 성공, False 실패 |
| 설명 | SkinnedMeshRenderer 정보를 마샬링한다. | |

마샬링은 그림 15에서도 나와있듯이, 서로 다른 메모리 구조를 가진 시스템 간의 변환을 하는 과정이라고 할 수 있다, 해당 연구에서는 C#의 관리되는 메모리 구조와 C++에서의 관리되지 않는 메모리 구조 둘 사이의 데이터를 넘기기 위한 용도로 사용된다. 정확히 말하자면, C#의 메모리 구조를 C++에서 사용할 수 있게 변환하는 것이 해당 연구에서 마샬링을 사용하는 목적이다. C++/CUDA 계산 모듈에서 이를 참조하여 계산하기 때문이다.

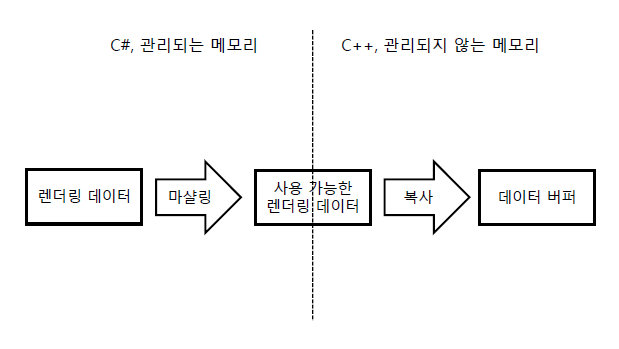


그림 : RadianceGrabber의 마샬링

그림 16에서는 Path-Tracing의 전체 알고리즘 순서도를 보여준다. 기하의 충돌 판정 그리고 부딫친 기하가 광원인지에 따라서 계산하는 것들이 달라지는 것이 키 포인트다. 또한 구현 단계에서는 이를 계산할 방법 또한 중요한 키 포인트 중 하나다. 이는 성능에 투명하게 영향을 미치므로 개발 전반에 걸쳐 고민되어야 한다.

CUDA의 성능 포인트는 SIMT의 특성을 가지기 때문에 각자 실행되는 프로그램들이 같은 시간에 걸쳐 끝나야 여러 코어들이 놀지 않을 수 있고, 최대한 통일성 있게 메모리를 참조해야 캐시 효율이 올라가서 프로그램의 실행 시간 자체가 줄어들 수 있다. 하지만 후자인 일관성(coherency) 있는 메모리 참조는 Path-Tracing의 큰 맥락에서는 거의 불가능하다. 가장 많은 시간을 소요하는 BVH 광선 탐색의 경우, 일관성 있는 메모리 참조를 위해서는 BVH를 매번 다시 생성하거나 매번 자식들을 정렬해야 한다. 하지만 이것은 더 많은 시간을 소요하므로 결과적으로는 일관성이 없는 메모리 참조는 BVH 탐색에 있어서는 가져 가야할 수 밖에 없다. 결국 최대한 노력과 성능이 비례할 수 있는 가능성이 높은 것은 전자인 최대한 같은 실행시간을 가지도록 하는 것이다.

이는 임계시간을 두거나, 씬 기하 vs 광선 판정의 횟수를 세어 종료하거나, 아예 이를 신경 쓸 필요 없이 작업들을 잘게 나누는 방법이 존재한다. 초기에는 아예 작업들을 잘게 나누는 것으로 협상한다.

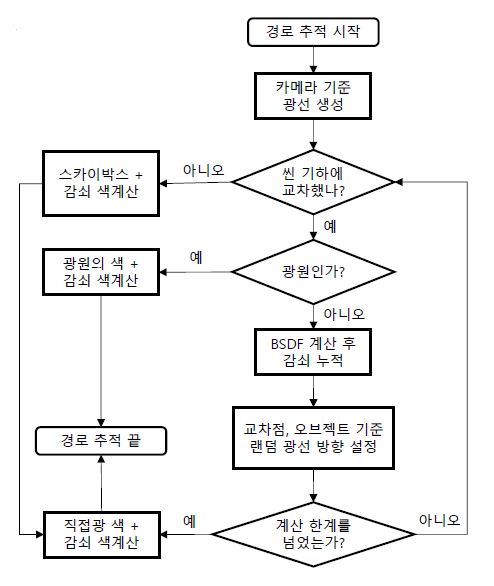


그림 : Path-Tracing 알고리즘 순서도

그림 17에서는 BVH의 생성과 광선에 따른 탐색 알고리즘 순서도를 보여준다. 생성의 경우에는 일반적으로는 Task를 병렬적으로 나누지 않고 순차적으로 CPU에서 실행한다. 그렇기 때문에 구현하는 코드는 탐색에 비하면 비교적 간단한 편이다. 그림 18에서 나오는 transform과 mesh, 두 레이어로 나뉘는 BVH는 멀티 스레딩을 하여 계산할 수도 있다. 탐색의 경우 GPU에서 실행해야 하기 때문에 최대한 적은 메모리 접근이 필요하다. 하지만 탐색 알고리즘 자체에서 선형 자료구조를 필요로 하기 때문에 메모리 접근을 한다는 가정하에 가장 효율적인 방법을 고려해야 한다. 고려할 수 있는 요소는 shared memory 가 있는데 , 이는 NVidia GPU 기준으로 각 워프마다 사용할 수 있는 메모리 공간으로, 레지스터에 준하는 속도로 접근이 가능하다. 그리고 이는 L1 Cache와 트레이드 오프를 할 수 있는 자원이기 때문에 캐시 효율을 고려하느냐 사용자가 그 공간을 직접 사용하느냐의 선택이라고 할 수 있다. shared memory의 특성 상 모든 쓰레드가 전부 다른 블록의 shared memory를 접근해야 최대한의 성능이 올라가기 때문에 커스텀 캐시로는 사용하기에는 굉장히 까다롭다. 그렇기에 초기 버전은 캐시 성능을 높이는 방향으로 가는 것이 바람직하다.

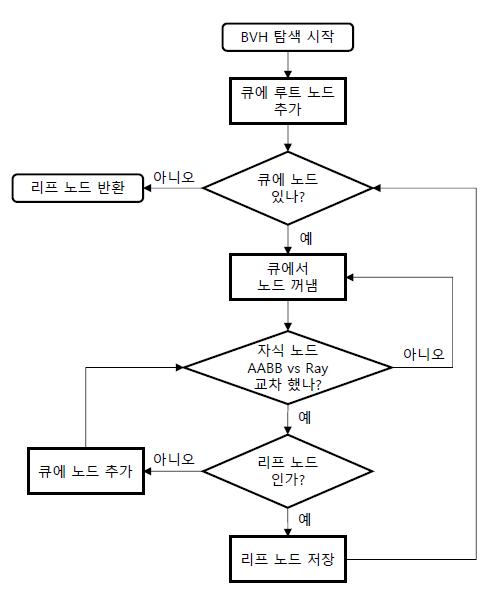
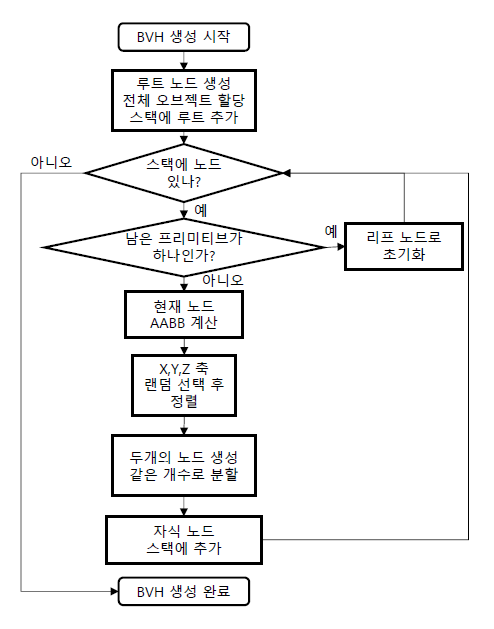


그림 : BVH 생성 순서도와 탐색 순서도

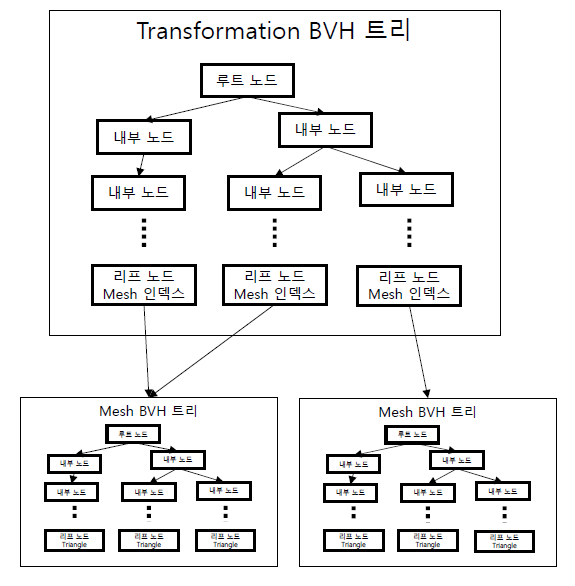


그림 : BVH 2단계 생성 전략

표 7에서 Path-Tracing 계산을 하는 클래스의 메소드의 앞에 <<device>> 표시는 GPU 메모리에 있거나, GPU에서 실행하는 함수임을 나타낸다. <<global>> 표시는 CPU코드에서 GPU 커널을 실행시키기 위한 함수로 실질적인 동작은 GPU에서만 실행한다. 모듈 설계 상, <<global>> 표시가 붙은 메소드는 GPU에서의 최소한의 실행 단위를 의미한다. 메소드의 실행 시간은 설계에 있어 굉장히 중요한 요소이다. 너무 많은 시간이 걸리면 GPU가 디스플레이 역할을 할 경우 실행 시간 동안 디스플레이 기능이 멈춰버리고, 너무 적은 역할을 하게 되면 CPU와 GPU의 동기화와 데이터 준비를 위한 driver overhead 들이 퍼포먼스를 잡아먹기 때문에 적절하게 실행 단위를 나누어야 한다.

표 : 계산 모듈 클래스들의 메소드

|  |  |  |
| --- | --- | --- |
| 클래스 | *PathIntegrator* | |
| 설명 | Path-Tracing 계산을 위한 클래스 | |
| 메소드 형식 | void Render(const IAggregate& scene, ColorTarget& target) | |
| 파라미터1 | const IAggregate& scene | 각 기하들과의 광선 교차 판정을 위한 씬 정보 |
| 파라미터2 | ColorTarget& target | 각 픽셀의 색을 저장할 버퍼 |
| 설명 | IIntergrator 순수 가상 함수의 구현, Path-Tracing 계산이 이 함수에서 이루어짐. | |
| 메소드 형식 | <<global>> void IntersectTest(const IAggregate& scene) | |
| 파라미터1 | const IAggregate& scene | 각 기하들과의 광선 교차 판정을 위한 씬 정보 |
| 설명 | CPU에서 GPU에 드라이버 레벨에서 작업을 요청하는 커널 함수로, 가장 시간이 오래 걸리는 씬 교차 판정을 실행한다. 정보는 멤버 변수인 mSegments에 기록된다. | |
| 메소드 형식 | <<global>> void ComputeScatteringAndAccumAttenutation() | |
| 설명 | CPU에서 GPU에 드라이버 레벨에서 작업을 요청하는 커널 함수로, 교차 판정 이후 실행 되어야 할 광선 방향과 감쇠를 설정하고, 교차 여부에 따라서 다음 할 것들을 결정한다. 정보는 멤버 변수인 mSegments에 기록된다. | |
| 클래스 | MLTIntegrator | |
| 설명 | MLT 계산을 위한 클래스 | |
| 메소드 형식 | void Render(const IAggregate& scene, ColorTarget& target) | |
| 파라미터1 | const IAggregate& scene | 각 기하들과의 광선 교차 판정을 위한 씬 정보 |
| 파라미터2 | ColorTarget& target | 각 픽셀의 색을 저장할 버퍼 |
| 설명 | IIntergrator 순수 가상 함수의 구현, MLT 계산이 이 함수에서 이루어짐. | |
| 메소드 형식 | <<global>> void CalcBDPTAsPathFind(const& IAggregate, ColorTarget& taget) | |
| 파라미터 | const IAggregate& scene | 각 기하들과의 광선 교차 판정을 위한 씬 정보 |
| 설명 | CPU에서 GPU에 드라이버 레벨에서 작업을 요청하는 커널 함수로, 맨 처음 BDPT 통한 각 픽셀의 가중치 계산 | |
| 메소드 형식 | <<global>> void CalcMLT(const IAggregate& scene) | |
| 파라미터 | const IAggregate& scene | 각 기하들과의 광선 교차 판정을 위한 씬 정보 |
| 설명 | CPU에서 GPU에 드라이버 레벨에서 작업을 요청하는 커널 함수로, BDPT Mutation과 함께 metropolis-hastings 알고리즘을 사용하여 경로 샘플링을 한다. | |
| 클래스 | *BVH* | |
| 설명 | 광선 탐색을 빠르게 하기 위한 가속된 탐색 구조. Transform부터 Triangle까지 전부 가짐 | |
| 메소드 형식 | <<device>> bool Intersect(const Ray& ray, Intersection& isect) | |
| 파라미터1 | const Ray& ray | 검사의 기준이 되는 Ray |
| 파라미터2 | Intersection& isect | 교차 시, 해당 Triangle과의 충돌 정보 |
| 반환 값 | bool | 교차한 Triangle이 있는가? |
| 설명 | GPU 내에서 실행되는 함수, Ray-BVH 탐색을 하여 가장 가까운 거리의 Triangle의 Intersection 정보를 세팅하여 종료한다. | |
| 메소드 형식 | void RecursiveBuild(BVHNode \*\*node) | |
| 파라미터 | BVHNode \*\*node | 노드 포인터 변수의 주소 |
| 설명 | 재귀적으로 랜덤으로 축을 정하여, 오브젝트의 개수가 반이 되도록 가르고, 오브젝트 개수가 한 개가 될 때까지 나누면서 bounding box를 계산한다. | |
| 메소드형식 | void Initialize(const MeshRendererChunk\* mrc, int mrcCount, const SkinnedMeshRendererChunk\* smrc, int srmcCount, const LightChunk\* lc, int lightCount) | |
| 파라미터1 | const MeshRendererChunk\* mrc | MeshRendererChunk 배열 시작 주소 |
| 파라미터2 | int mrcCount | MeshRendererChunk 배열의 개수 |
| 파라미터3 | const SkinnedMeshRendererChunk\* smrc | SkinnedMeshRendererChunk 배열 시작 주소 |
| 파라미터4 | int srmcCount | SkinnedMeshRendererChunk 배열의 개수 |
| 파라미터5 | const LightChunk\* lc | LightChunk 배열 시작 주소 |
| 파라미터6 | int lightCount | LightChunk 배열의 개수 |
| 설명 | BVH 초기화 함수, RecursiveBHV를 호출 | |
| 클래스 | TwoLayerBVH | |
| 설명 | Transform과 Mesh를 나누어서 트리를 구성. | |
| 메소드 형식 | <<device>> bool Intersect(const Ray& ray, Intersection& isect) | |
| 파라미터1 | const Ray& ray | 검사의 기준이 되는 Ray |
| 파라미터2 | Intersection& isect | 교차 시, 해당 Triangle과의 충돌 정보 |
| 반환 값 | bool | 교차한 Triangle이 있는가? |
| 설명 | GPU 내에서 실행되는 함수, Ray-TransformBVH-MeshBVH 탐색을 하여 가장 가까운 거리의 Triangle의 Intersection 정보를 세팅하여 종료한다. | |
| 메소드 형식 | void RecursiveBuild(TransformBVHNode \*\*node) | |
| 파라미터 | TransformBVHNode \*\*node | 노드 포인터 변수의 주소 |
| 설명 | 재귀적으로 랜덤으로 축을 정하여, 오브젝트의 개수가 반이 되도록 가르고, 오브젝트 개수가 한 개가 될 때까지 나누면서 bounding box를 계산한다. | |
| 메소드 형식 | void RecursiveBuild(MeshBVHNode \*\*node) | |
| 파라미터 | MeshBVHNode \*\*node | 노드 포인터 변수의 주소 |
| 설명 | 재귀적으로 랜덤으로 축을 정하여, 오브젝트의 개수가 반이 되도록 가르고, 오브젝트 개수가 한 개가 될 때까지 나누면서 bounding box를 계산한다. | |
| 메소드형식 | void Initialize(const MeshRendererChunk\* mrc, int mrcCount, const SkinnedMeshRendererChunk\* smrc, int srmcCount, const LightChunk\* lc, int lightCount) | |
| 파라미터1 | const MeshRendererChunk\* mrc | const MeshRendererChunk\* mrc |
| 파라미터2 | int mrcCount | int mrcCount |
| 파라미터3 | const SkinnedMeshRendererChunk\* smrc | const SkinnedMeshRendererChunk\* smrc |
| 파라미터4 | int srmcCount | int srmcCount |
| 파라미터5 | const LightChunk\* lc | const LightChunk\* lc |
| 파라미터6 | int lightCount | int lightCount |
| 설명 | BVH 초기화 함수, RecursiveBHV를 호출 | |
| 클래스 | *LinearAggregate* | |
| 설명 | 초기 Path-Tracing 구현 및 성능 비교를 위한 씬 기하를 가지고 있는 클래스 | |
| 메소드 형식 | <<device>> bool Intersect(const Ray& ray, Intersection& isect) | |
| 파라미터1 | const Ray& ray | 검사의 기준이 되는 Ray |
| 파라미터2 | Intersection& isect | 교차 시, 해당 Triangle과의 충돌 정보 |
| 반환 값 | bool | 교차한 Triangle이 있는가? |
| 설명 | GPU 내에서 실행되는 함수, Ray-BVH 탐색을 하여 가장 가까운 거리의 Triangle의 Intersection 정보를 세팅하여 종료한다. | |
| 메소드형식 | void Initialize(const MeshRendererChunk\* mrc, int mrcCount, const SkinnedMeshRendererChunk\* smrc, int srmcCount, const LightChunk\* lc, int lightCount) | |
| 파라미터1 | const MeshRendererChunk\* mrc | const MeshRendererChunk\* mrc |
| 파라미터2 | int mrcCount | int mrcCount |
| 파라티터3 | const SkinnedMeshRendererChunk\* smrc | const SkinnedMeshRendererChunk\* smrc |
| 파라티터4 | int srmcCount | int srmcCount |
| 파라티터5 | const LightChunk\* lc | const LightChunk\* lc |
| 파라티터6 | int lightCount | int lightCount |
| 설명 | 초기화 함수, 단순히 배열을 저장 | |

## **Prototype 구현**

2.5.1 구현 현황

프로토타입 기준 구현 현황은, 기본적인 기능을 갖춘 코어 기능만이 보여질 수 있다. 코어 기능은 다음과 같다. 붉은 색으로 표시된 부분은 3차 첫번째 발표 ~ 재심 간의 변화사항이다.

* 메쉬, 메터리얼, 광원 간의 상호 작용 및 동작 : 메터리얼의 경우 광선 추적 시에 각 쉐이더 별로 어디로 튕겨나갈 지, 색에 대한 감쇠가 어떻게 적용될 것인지를 결정하는데 이는 부분적으로만 구현되었다, 메쉬는 기하학적 정보를 가지고 충돌 처리를 하는 부분에 대한 부분인데, 이는 현재 선형적으로만 탐색 가능하도록 되어 있다. 광원의 경우에는 광선이 처음 카메라에서 나간 이후에 메쉬와 함께 충돌 처리에 포함되며, 광선이 넓이를 가진 광원에 부딫치면 하나의 광원의 색과 감쇠, pdf를 처리하여 픽셀의 색을 구한다.
* Unity에서의 데이터 연결 및 에디터 지원 : 하나의 프레임을 계산하기 위해 광선 추적 모듈에 Unity에서의 데이터를 전달하는 기능, Unity C#에서의 주소공간, 호스트 주소공간, 디바이스 주소 공간 메모리의 할당 및 데이터 전달과 에디터 상에서의 계산을 할지 말지에 대한 제어 및 결과를 보여주는 뷰어 기능을 제공한다. 데이터 전달 기능 중, 싱글 프레임을 위한 데이터 전달만 구현되어 있다. Unity의 에디터 기능 또한 싱글 프레임을 계산하는 프리뷰 창과 이에 관한 설정, 제어를 위한 UI를 제공한다.
* stochastic path-tracing : 광선이 물체에 부딫칠 때, 물체의 재질에 따라서 정해진 특정 범위의 방향을 랜덤으로 설정하여 계산하는 기본적인 알고리즘으로 위의 첫번째로 언급한 상호 작용을 통하여 색을 계산하고 광선의 방향을 반구 혹은 정해진 Cone 안에서 랜덤하게 설정한다.
* 기하 가속 자료 구조 : 구현할 자료구조는, 물체 별 경계 상자를 트리 형태로 가지는 BVH, 정적 메쉬의 폴리곤을 트리 형태로 저장하는 kd-tree, 매 프레임별 바뀌는 메쉬의 폴리곤을 저장하는 BVH가 있다. 그 중에서 가장 먼저 필요한 정적 메쉬의 kd-tree를 먼저 구현했다. 일반적으로 쓰이는 메쉬의 폴리곤의 개수는 몇 백개에서 몇 만개 사이로, 3가지의 자료 구조에서 가장 많은 것들을 저장하고, 가장 많이 쓰이기 때문이다.

이외의 동영상 생성을 위한 기능, MLT는 구현되지 않았다.

표 : 프로토타입 버전 구현 현황

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 모듈 | | 진행 상태 | | | | | | | | |
| Core : 싱글 프레임 계산 | | 모든 기능 미지원, 시연 가능 버전 | | | | | 70% | | | |
|  | 메터리얼, 메쉬, 광원 interact | 메터리얼 부분적 지원 | | | | | | 80% | | |
|  | Unity 데이터 연결 및 에디터 지원 | 멀티 프레임 계산 용 데이터 연결 미지원 | | 60% | | | | | | |
|  | Path-Tracing |  | | | | | | | 100% | |
| 멀티 프레임 계산 | | 0% | | | | | | | | |
| 지오메트리 가속 구조 | |  | 30% | | | | | | |
| 정적 메쉬 Kd Tree | | Median 분할 구현, sah 분할으로 성능 개선 | | | | 70% | | | | |
| 동적 메쉬 BVH | | 0% | | | | | | | | |
| 물체 단위 BVH | | 0% | | | | | | | | |
| MLT | | 0% | | | | | | | | |
| 전체 진행 현황 | |  | | | 65% | | | | | |

2.5.2 구현 디테일

2.5.2.1 Unity에서의 환경

해당 연구의 기본적인 실행 환경은 Untiy 프로젝트를 에디팅 하는 환경이다. Unity 내부에서는 프로젝트의 바이너리 자체를 기본적으로 두가지로 나누는데 Runtime과 Editor이다. Runtime은 실제 리얼타임으로 돌아가는 것들을 의미하고, Editor는 개발자가 Unity를 통해 작업할 때의 바이너리를 의미한다. Unity는 Runtime에 집중하여 만들지만, 해당 연구에서는 Editor 바이너리가 중점적인 환경이다. 이런 Unity Editor 환경에서 기본적인 뷰어 기능부터 광선 추적을 하는 기능까지 모든 것들이 실행된다.

전체적인 구조는, Unity 에디터 상의 메인 쓰레드에서 UI와 관련된 모든 것들을 관리한다. 이는 에디터에서는 당연히 내장 UI를 사용하기 때문에 실행되는 환경이라고 할 수 있다. 중요한 점은 Unity 내부의 모든 객체는 에디터 상에서의 메인 쓰레드나, 런타임 상에서의 메인 쓰레드에서 밖에 접근을 할 수 없다. 이는 Unity의 메인 쓰레드가 아닌 다른 쓰레드에서 내부 오브젝트를 사용하기 위해선 해당 메모리를 복사하여 사용할 수 밖에 없다는 것을 암시한다.

그렇게 Unity 에디터 상에서 UI를 통해 광선 추적을 실행하려면 미리 만들어진 dynamic link library에 있는 바이너리 코드를 통하여 실행되어야 한다. (Unity 내부에서도 가능하지만 CUDA와 C++의 성능을 최대화하기 위해서는 이와 같은 선택이 가장 합리적이다.) DLL안에는 CPU에서 실행되는 호스트 코드와(C++) GPU에서 실행되는 커널 코드와 순수 디바이스 코드(CUDA)가 들어있다. 그렇게 DLL을 통해 Unity 에디터 상에서 실행이 되면, 그대로 광선 추적을 실행하는 루틴을 실행시킬 수 없다. 왜냐하면 Unity 에디터는 싱글 쓰레드로 실행이 되기 때문에 계산이 완료될 때까지 멈춰 있기 때문이다. 이는 결국 Unity 에디터의 C#코드에서나 C++ 코드에서 쓰레드를 생성하여 관리해야 되는 것을 의미한다. 그렇게 비동기적으로 실행이 되면 그때부터는 직접적으로 CUDA를 통해 GPU를 사용하여 계산한다.

Unity에서 받은 데이터를 직접 접근하기 위해서는 Graphic Driver API를 사용하여 데이터를 가져와야 한다. 메쉬나 텍스쳐 같은 데이터들은 전부 Graphic Driver API 전용으로 만들어져 있기 때문에 이를 복사하기 위해서는 알맞은 사용 방법을 통해 데이터를 복사해야 한다. 이를 지원하기 위해서 Unity에서는 low-level native plugin이라는 형태의 예제 플러그인을 지원한다. 이는 Unity에서 사용하는 Graphic Driver API 객체들을 연결해주어 개발자가 이러한 자원을 사용할 수 있도록 해준다. 이렇게 low-level native plugin을 프로젝트에 포함하여 이를 활용하면 데이터를 가져올 수 있게 된다. DirectX11 기준으로 ID3D11Device 객체를 가져와야 ID3D11DeviceContext를 획득할 수 있다. 그런데 ID3D11Device 객체는 일반적으로 얻을 수 없고, Unity에서 제공하는 API를 통해 가져와야 한다. 이 때문에 네이티브 플러그인을 사용해야한다. OpenGL, DirectX 11,12, Vulkan, Metal 등 여러 API를 지원하게 해 놓았는데 현재는 DriectX11만 사용하여 구동된다.

2.5.2.2 CUDA

Visual Studio에서 CUDA를 사용하기 위하여 프로젝트를 만들면 SDK와 GPU 디바이스의 하위 호환성 같은 이유로 기본적으로 상위 버전 SDK에서 사용하는 옵션들을 활성화하지 않는다. 하지만 CUDA는 대부분 병렬적으로 실행 가능한 계산의 높은 퍼포먼스를 위하여, OpenCL과 다르게 공학 분야에서 개발의 편의성도 갖추어 사용되기 때문에 최대한 상위 버전에서 사용될 수 있는 것은 사용해야 한다. 컴파일 옵션도 해당하는데, 대부분의 옵션은 지금의 PC 게이밍 시장에서 우위를 갖추고 있는 GPU들도 지원하고, 대부분 개발용 PC의 경우에는 게이밍 컴퓨터 이상으로 옵션을 맞추기 때문에 이를 수동으로 전부 세팅해주어야 한다.

CUDA는 GPU를 일반적인 목적으로 사용하기 위해, GPU와 CPU에서 실행되는 바이너리 코드를 만들기 위하여 함수들을 각각 세가지 분류로 나누어 다르게 컴파일한다. 설계에서 언급된 것 처럼 \_\_host\_\_ 마크가 붙은 CPU에서 실행되는 코드, \_\_device\_\_ 마크가 붙은 GPU에서 실행되는 코드, \_\_global\_\_ 마크가 붙은 CPU에서 GPU에서 실행하기 위해 넘어가는 커널 코드로 나뉘게 된다. 이러한 분류 아래 호스트 코드와 커널 코드가 연관이 되고, 커널 코드와 디바이스 코드가 연관이 되어 컴파일이 진행된다. 이러한 분류 아래, 개발시에 보통 클래스 별로, 혹은 그 관련된 집합 별로 소스 파일을 나누게 된다. 그리고 소스 간의 의존성이 생기고 이때 서로 간의 링크를 해주어야 한다. 아무 옵션을 설정하지 않으면 컴파일 에러로 인해 문제가 생긴다. 구버젼 SDK 에서는(5.0 이전) 애초에 이런 분할된 컴파일을 지원하지 않았기에 그렇다. 이를 세팅하는 옵션은 파일별 컴파일 세팅에서는 -rdc=true, 링킹 세팅에서는 -dlink 를 설정하면 된다. 이는 디바이스 코드, 커널 코드, 호스트 코드 간의 링킹을 따로 할 수 있게 해주는 기능으로, rdc=true는 relocatable device code를 생성할 수 있게 해주는 것을 뜻하는 것으로, 각 \_\_device\_\_, \_\_global\_\_ 코드를 전부 따로 목적 파일로 만들어 놓는 것을 뜻하고, -dlink는 이러한 만들어진 목적 파일과 기존의 C/C++ 호스트 코드 전부 간의 링킹을 해주게 하는 옵션이다.

CUDA에서 커널은 GPU 관점에서의 프로그램을 실행하기 위한 시작점이다. 그렇기에 커널 실행의 코스트는 굉장히 크고, 이의 구조 또한 기존의 CPU 코드의 실행과는 사뭇 다르다. 커널 실행 시 특정 Configuration을 통해 이를 엿볼 수 있다. 커널의 실행은 다음과 같이 4개의 Configuration을 통해 이루어진다.

*kernel*<<<dim3 **block\_cnt\_in\_grid**, dim3 **thread\_cnt\_in\_block**, int **stack\_limit**, int **stream\_num**>>>(~~~); (1)

**thread\_cnt\_in\_block**와 **block\_cnt\_in\_grid**는 한꺼번에 실행할 쓰레드의 개수와, 그 쓰레드의 집합의 개수를 의미한다. 조금 더 정확하게 말하자면, 하나의 블록은 여러 개의 쓰레드로 이루어진다. 이것이 **thread\_cnt\_in\_block** 의미하는 것이고, 하나의 그리드는 여러 개의 블록으로 이루어진다. **block\_cnt\_in\_grid** 이 의미하는 것은 이와 같다. 이것이 다수의 쓰레드를 실행하기 위한 커널 실행을 위한 메모리 레이아웃이다.



그림 : CUDA의 grid, block, thread

NVidia GPU 칩에서 가장 핵심적인 부분은 여러 개의 SM(Streaming Multiprocessor)이다. 실질적인 병렬적인 계산을 하는 유닛이다. 이 SM안에는 여러가지 유닛들이 존재하는데, 가장 쉽게 알 수 있는 부분은 CPU에서 Core 라고 부르는 개념의 SP(Streaming Processor)가 여러 개 존재한다. 거의 대부분의 아키텍쳐에서는 워프의 개수만큼 한 SM안에 SP가 존재한다. (인텔의 하이퍼쓰레딩을 무시하면)CPU의 Core가 하나의 쓰레드를 실행하듯이 SP마다 하나의 쓰레드가 존재한다.



그림 : Streaming Multiprocessor의 구조

커널 실행을 통한 프로그램 실행을 SM에 할당하기 위하여 하나의 그리드에 있는 블록들을 각각의 SM에게 여러 개 할당한다. 모든 SM의 블록 개수의 한계가 오면 이는 처리 대기를 위해 기다린다. 그리고 SM은 이들을 실행하기 위하여 32개의 쓰레드 문맥을 워프라는 단위로 묶어 각 블록들을 워프 단위로 잘라서 실행한다. 여러 개의 워프로 관리되어 실행하는 이유는 하나의 워프의 실행 시 지연시간이 긴 경우에 컨텍스트 스위칭을 통해 다른 워프를 실행하는 것을 선택하여 코어를 쉬지 않고 실행시키기 위함이다. CPU의 컨텍스트 스위칭과 확연하게 다른 점은 “컨텍스트”를 그대로 보존할 수 있다면 보존하기 때문에 “이론 상”으로는 부하가 굉장히 적은 편이다. 다만 광선 추적의 구현 특성 상 레지스터의 크기 또한 많이 차지하며, 전역 메모리 fetch가 빈번하게 일어나기 때문에 이러한 zero-overhead thread scheduling의 이점을 완벽하게 살리기는 힘들다.

문제는 최적의 성능을 가져갈 수 있는 블록, 쓰레드 개수를 찾는 것이다. 이 부분은 설계시에 크게 고려되지 않았기 때문에, 분할하여야 할 태스크의 개수를 단순하게 현재 GPU의 상태만 체크하여 최대 쓰레드 개수에 맞추어서 블록을 나누어 할당했다. 이는 결과적으로 하나의 블록안에 최대한 많은 워프를 구성하여 실행하게 만드는 환경을 갖추게 만든다. 쓰레드 수는 공유 메모리의 사용량에 따라서 최대의 개수가 정해지는 편인데, SM의 구조가 공유 메모리를 사용하게 되면 SM의 캐시가 작아 지기 때문에 메모리 접근이 엄청나게 많은 광선 추적 알고리즘을 돌리기 위해서는 공유 메모리의 사용상의 이점보다 캐시의 효율을 차라리 택하는 편이 낫기 때문에 공유 메모리는 사용하지 않는다. 이는 블록안의 최대한의 쓰레드 개수를 할당해도 크게 문제가 없다는 점을 의미한다.

하지만 이는 절대 최적의 방법은 아니다. 광선 추적 자체가 분기가 많고 그에 따라 지역변수를 사용하는 것이 달라진다. 이는 각 쓰레드의 레지스터 사용량을 다르게 하고, 각 쓰레드의 레지스터의 사용량이 늘어나면 워프 별로 실행하는 쓰레드 개수 자체가 적어진다. 이는 단순하게 하나의 계산을 통해 최적의 블록, 쓰레드 개수 값을 얻을 수 없으므로 커널 별로 몇가지의 실험을 통해 최적의 블록, 쓰레드 값을 찾아야 한다.

최대한의 GPU의 활용을 위해서 NVidia에서 제공하는 툴을 사용하여 체크를 할 수 있다. 하나는 엑셀의 형태로 지원하는 Occupancy Calculator를 지원한다. 현재 꾸준하게 지원하는 것 중 하나로 하드웨어 스펙과 블록 안의 쓰레드 개수, 쓰레드에서 사용하는 레지스터의 총량, 블록 별로 사용하는 공유 메모리의 크기에 따라서 SM 안에서 실행되는 쓰레드의 개수, 즉 점유율이 결정된다. 하지만 여기서 조금 아쉬운 점은 복잡한 프로그램의 경우에는 레지스터 개수를 직접 세기에는 어렵기 때문에 이를 적극적으로 활용하기에는 조금 어려움이 있다고 할 수 있다.

그래서 NVidia NSight Compute 라는 프로그램을 사용할 수 있는 것을 알게 되었는데 이는 실시간으로 디버깅이 가능한 모니터링 툴로, 디버그용으로 GPU의 내부 상태 같은 것들을 확인할 수 있는 것이다. 점유율 같은 정보뿐만 아니라 상태에 대한 차트를 시간별로 지원하며 원하면 해당 시간에 대한 자세한 정보, 현재 상태에서 성능 하락 이슈가 있을 시 이를 경고해주는 것들도 볼 수 있게 되어있어 최적화 실행시에는 계속 붙잡고 있을 디버깅/모니터링 용 툴이다.

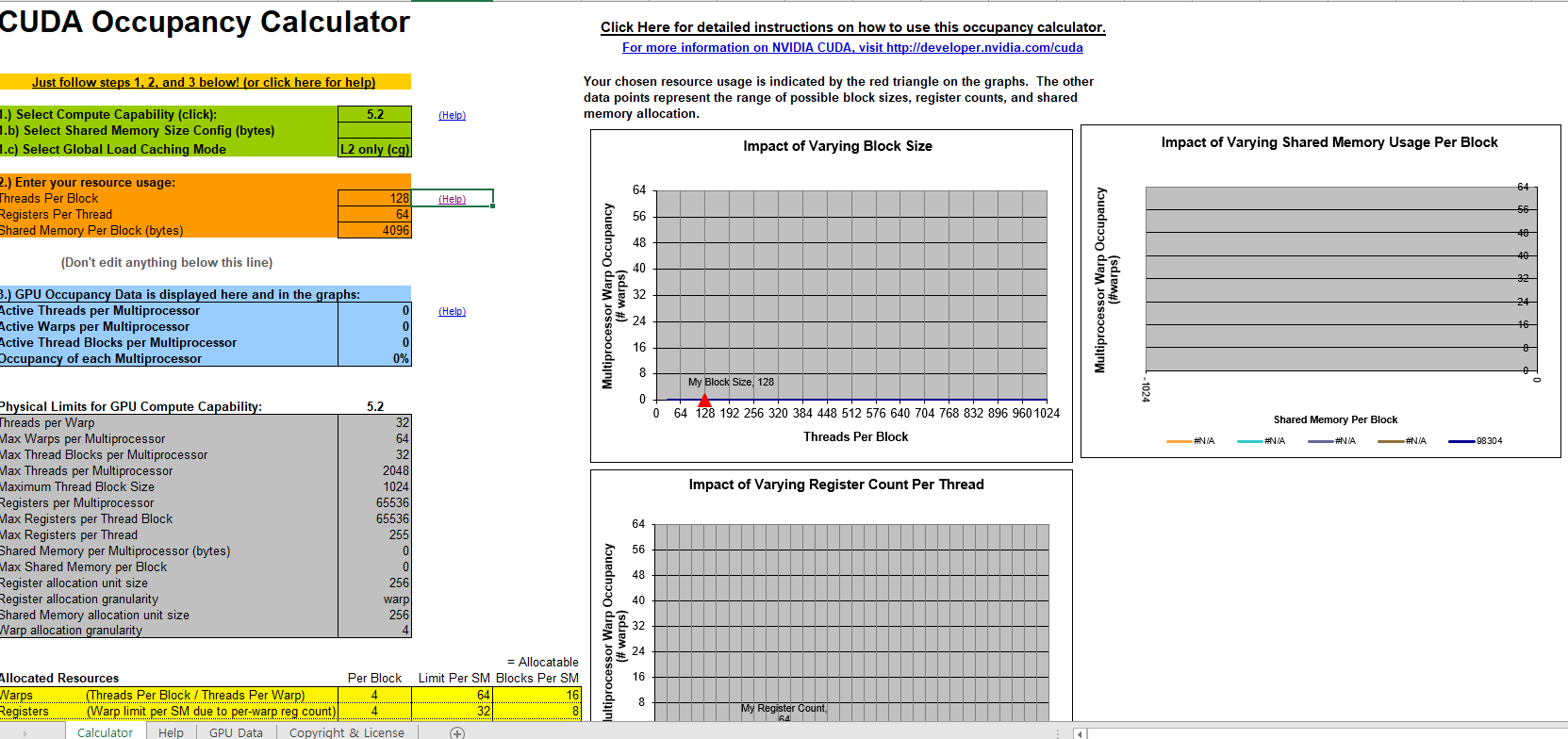


그림 : 엑셀로 지원하는 Occupancy Calculator

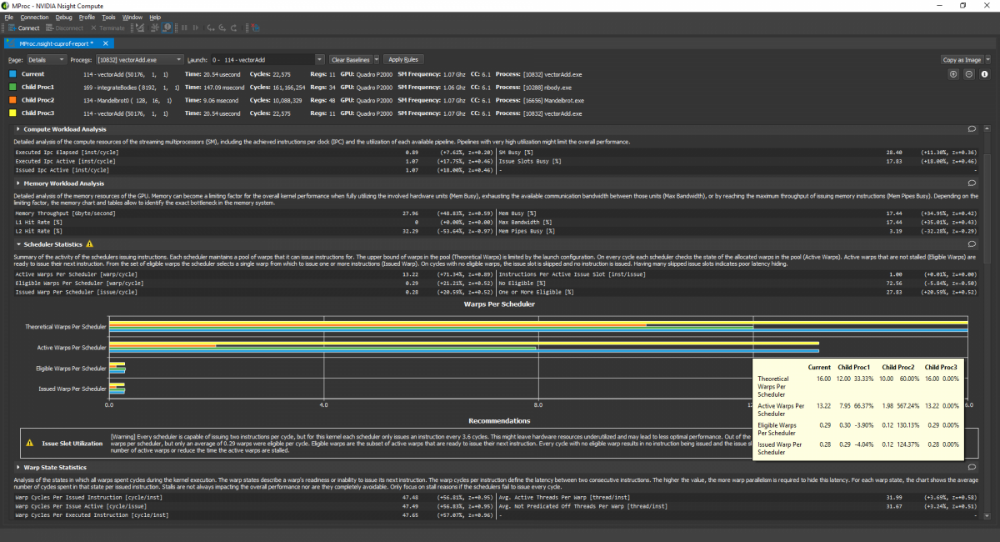


그림 : NVidia NSight Compute

2.5.2.3 PathTracing + CUDA

또한 광선 추적에서 커널을 사용하기 위해서는 여러 고안이 필요했다. 단순하게 하나의 픽셀 당 최대 개수의 샘플링을 진행하는 것은 기본적으로 시간이 오래 걸리기 때문에 컴퓨터에서 해당 GPU를 GUI 출력 장치로 이용하는 경우에 실행 시간 동안 그래픽 디스플레이 기능이 마비되기 때문에 사실 상 불가능한 방법으로 간주되었고, 이를 전체 픽셀 개수에 최대 쓰레드의 개수를 나눈 값을 블록의 개수로 설정하여 한번의 샘플링 과정을 한번의 커널 실행 단위로 놓았다. 다만 이 과정에서도 약간의 디스플레이 업데이트 딜레이가 존재하여 이에 대한 고안을 다시 해야 했다. 프로토타입의 구현 사항은 여기서 멈춘다.

위 커널 실행 configuration (1)에서 블록, 쓰레드의 개수 외에 **stream\_num** 을 언급하지 않았는데 이는 일종의 커널 태스크 큐로 활용될 수 있는 CUDA의 Runtime API에서 지원되는 기능이다. Runtime API에서 접미사가 Async 인 것들에 필요한 인자로 노출되는데 이는 함수 호출 시 해당 스트림에 함수의 행동을 예약하는 커맨드 큐에 커맨드를 넣는 것과 동일한 것이라고 할 수 있다. 광선 추적의 구현에는 전체 픽셀의 개수와 관련 된게 아니라 (SM별 최대 블록의 개수) X (SM의 개수)로 최소한의 커널 실행 단위를 작게 하고, 픽셀의 영역별로 스트림을 만들어 비동기 커널 실행을 통해 한꺼번에 태스크를 넣어 디스플레이 업데이트 딜레이를 줄이는 방법이 존재한다.

기존 설계 시 커널 함수를 클래스 멤버 함수로 설정해 놓았다. 실제 구현에 들어가면서 이 부분은 변경되었다. 커널 함수는 클래스의 멤버 함수가 될 수 없다는 것을 설계시에 반영하지 못했다. 이는 IIntergrator를 구현하는 클래스들의 멤버 함수를 대거 변경해야 하는 이유가 되었다.

2.5.2.4 Path-Tracing + PBR

Unity의 기본적인 메터리얼은 PBR[[7]](#footnote-7)(Physically Based Rendering)을 지원하기 위한 우버(uber) 쉐이더[[8]](#footnote-8)를 기반으로 한다. PBR의 일부인 microfacet theory 가 존재하는데, 이는 하나의 작은, 일반적으로 하나의 픽셀로 나타내어지는 표면이 무한한 개수의 미세면(microfacet)으로 이루어져 있어 우리는 그 표면의 성질을 통계적으로 알아서 그 통계 수치를 가지고 값을 계산할 수 있다는 것을 의미한다. 이는 메터리얼에서 일반적으로 나타내어지는 방식인 [0, 1]의 범위를 가지는 것이라고도 할 수 있다. 대표적으로 roughness 라는 값이 있다. 이는 수많은 미세면들이 모여 어느정도 표면이 울퉁불퉁해지는 지를 나타내는 값이다. 이 값이 높으면 높을수록 미세면들의 집합인 표면 울퉁불퉁하다는 것을 의미한다. 이 값이 낮으면 낮을수록 매끈한 표면 때문에 빛을 받으면 더욱 상이 깔끔하게 대상 물체에게 투영되어 보여진다.

두가지의 작업 방식과 두가지의 재질의 렌더링을 지원하여 사실상 총 4가지의 렌더링을 지원한다. 서로 다른 재질은 금속과(metallic) 절연체(dielectric)으로 나뉜다. 이러한 재질은 광선 추적에서 두가지 경우에 영향을 미친다. 첫번째로는 튕겨나가는 광선의 방향을 결정한다. 금속은 대부분의 빛을 정반사의 형태로 반사시킨다. 이러한 성질은 표면이 매끈한(roughness == 0)의 경우에는 거의 완전한 정반사가 되며 표면이 울퉁불퉁한 경우(roughness > 0) 그림 20이 표시하는 것 처럼 정확한 방향이 아닌 해당 방향의 근처로 랜덤하게 반사가 되어 표현이 된다. 두번째는 색에 영향을 미치는 것, 반사된 혹은 굴절된 빛의 감쇠이다. 금속 표면은 거의 정반사된 색과 Albedo 색으로 표면의 색이 결정이 된다. 그래서 광선 추적 알고리즘에서 금속을 다룰 때는 단순하게 반사만 다루면 되고 색의 감쇠의 경우에는 물체의 Albedo 색을 사용하면 된다. 굴절의 경우에는 거의 일어나지 않기 때문에 여기서는 무시된다. 절연체는 반사와 굴절이 동시에 일어나기 때문에 정반사와 난반사 모두 다루어져야 하고 기본 Albedo 색은 난반사의 양에 따라서 달라진다. 이러한 구분은 metallic이라고 불리는 [0,1]의 범위를 가지는 것으로 표현된다. 0에 가까울수록 절연체이고, 1에 가까울수록 금속의 성질을 띈다. 일반적으로 물체는 금속과 절연체로 딱딱 나뉘지만 표면의 녹슴과 낡음을 표현하기 위하여 약간의 활용을 할 수 있을 정도로만 사용되고 일반적으로는 거의 0 아니면 1로 구분되어서 사용된다. (0,1)의 범위면 uniform sampling을 통해 결과를 계산할 수 있다.

금속과 절연체에 따른 분류 외에도 지원되는 workflow에 따라서 메터리얼이 구분된다. PBR의 기본적인 패러다임을 따르는 “metallic workflow” 와 PBR 이전의 방식의 물체의 정반사광을 표현하는 방식을 하위 호환하는 “specular workflow” 가 존재한다. 윗 문단에서 언급한 방식은 “metallic workflow”라고 불리며, 금속이냐 절연체냐에 따라서 물체의 속성이 바뀌는, 존재하는 물체들의 특징을 정리한 방식이다. “specular workflow”는 PBR 이전의 diffuse, specular 색을 따로 물체별로 정하여 색을 계산하는 방법이다. “specular workflow” 에서의 roughness이 커지면 specular 항이 색에 영향을 주지 않게 한다.

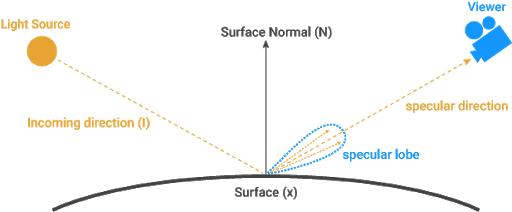
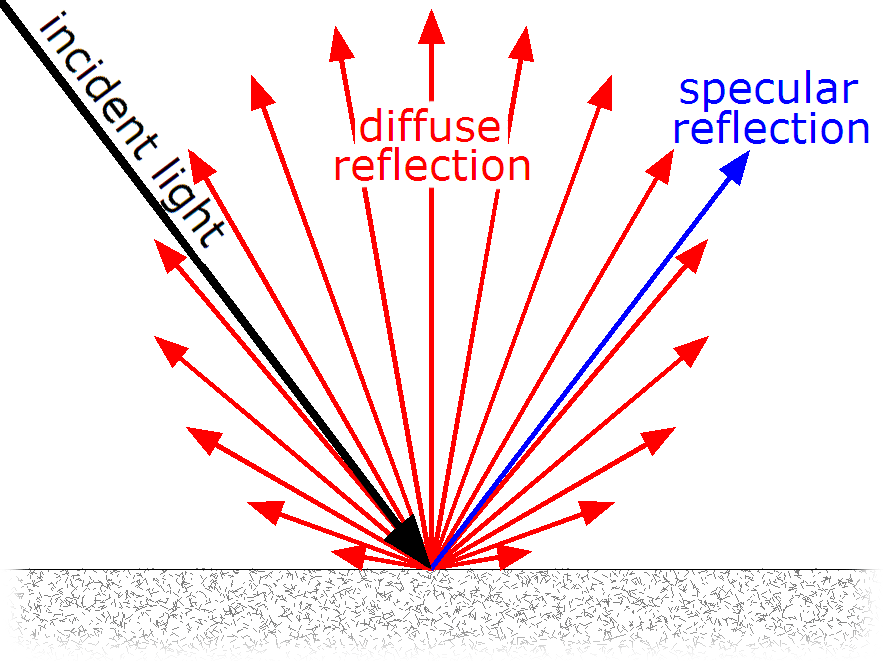


그림 23 : diffuse(왼쪽), specular reflection lobe(convolution filter)(오른쪽)



그림 24 : 금속 재질의 표면 반사, 매끈한 경우(왼쪽), 약간 거친 경우(중간), 거친 경우(오른쪽)

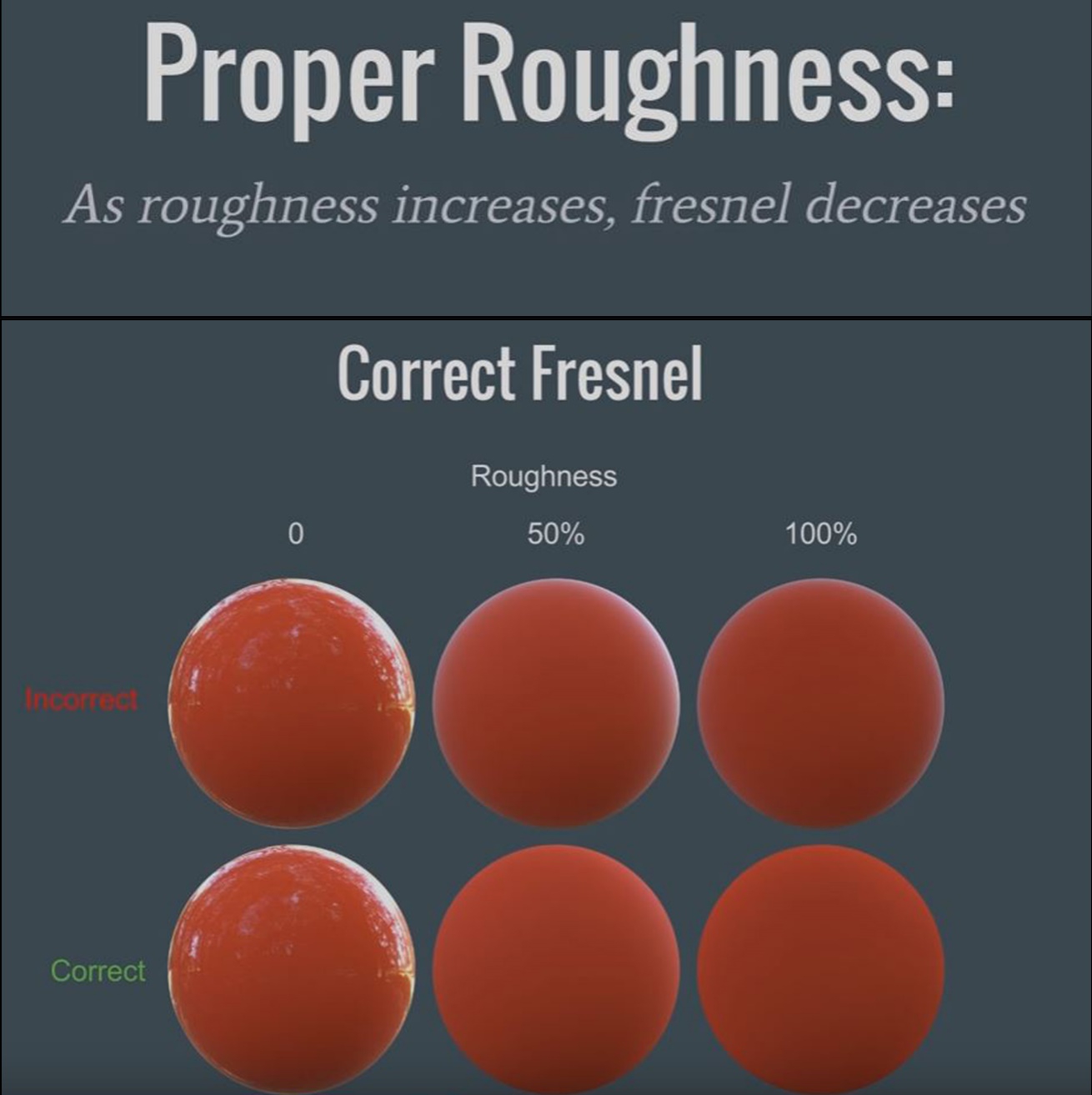


그림 25 : 절연체 재질의 표면 반사, 매끈한 경우(왼족), 약간 거친 경우(중간), 거친 경우(오른쪽)

광원은 두가지 경우로 나뉜다. 형체가 있는 광원과 없는 광원으로 크게 나눌 수 있다. 형체가 있는 광원은 일반적으로 Area Light 라는 사각의 모양으로만 지원하는 광원이 존재하고 메터리얼에서 emittance 옵션을 부여하면 광원으로 처리하여 해당 광선과 부딫친 부분의 색을 계산하여 이를 광원의 색으로 처리한다. 형체가 있는 광원은 광선 추적을 통해 단순하게 여러 메쉬처럼 처리하면 된다.

형체가 없는 광원 중, 광원을 해라고 가정하고 방향만을 정한 directional light는 쉽게 계산할 수 있다. 광선 추적 시 완전히 충돌한 물체가 없다면 빛의 방향 혹은 광선의 방향을 반대로 하여 내적을 한 값과 빛의 색을 곱하여 광원의 색으로 사용할 수 있으나, 하나의 점에서 빛이 나오는 Point Light, 이의 응용된 버전인 Spot Light는 기존의 광선 추적 알고리즘에 적용시킬 방법이 사실 상 없다. 이에 대한 방안이 마련되어야 하고 적용시켜야 한다. 현재 프로토타입에서는 이 부분에 대한 방안이 마련되어 있지 않다.

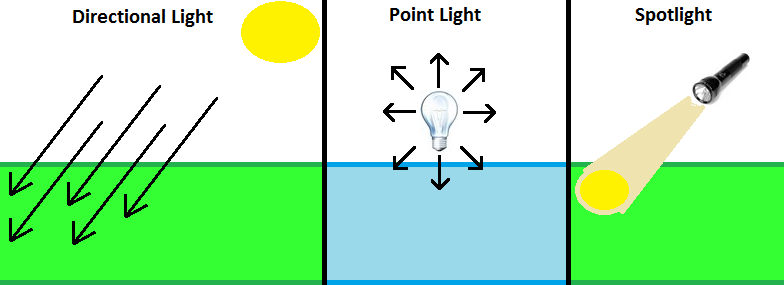


그림 : Directional Light(왼), Point Light(가운데), Spot Light(오른쪽)

2.5.2.5 Path-Tracing

무작위 광선 추적은 다른 쓰레드에서 실행된다는 것을 전제하에 다음과 같은 순서로 이루어진다.

1. 세그먼트, 광선 그리고 기하적 충돌을 저장하는 용도의 메모리를 초기화 해준다.
2. 광선과 기하 충돌을 위한 탐색을 하는 커널을 실행한다.
3. 분기에 따라 다르게 처리하는 커널을 실행한다. 경우에 따라서 4가지 분기로 나뉜다
   1. 광선과 기하 충돌이 일어나지 않을 시, 스카이박스를 샘플링하여 색을 가져와 이전까지 계산된 감쇠와 함께 색을 결정한다.
   2. 마지막 Depth에 도달했을 때, 픽셀에 검은색을 저장한다.
   3. 광원에 부딫칠 때, 광원의 색을 샘플링하여 감쇠와 홤께 색을 결정한다.
   4. 물체에 부딫칠 때, 충돌 정보와 메터리얼 정보를 통해 광선의 방향과 감쇠를 누적시킨다.
4. 기존의 처리하던 메모리를(광선 추적 세그먼트) 호스트 주소 공간의 메모리에 복사한다.
5. CPU에서 끝난 픽셀이 있는지 검사하여 색을 갱신하고 이를 다음 픽셀에 배정해준다.
6. 디바이스 주소 공간의 세그먼트에 변경한 세그먼트를 복사해준다.
7. 해당 프레임의 모든 픽셀을 처리하지 않았으면 다시 루프를 돈다.

여기서 중요한 것은 주소 공간의 변경이다. 4번 이전의 절차에서는 디바이스 주소 공간의 세그먼트에서 계속 작업을 하다가 갑자기 호스트 주소 공간의 세그먼트로 복사한다. 이는 GPU에서 복수개의 쓰레드가 동시에 하나의 변수를 증가시키려 하는 race condition이 일어난다. 이를 방지하기 위한 방법은 lock이 있는데, GPU에서의 lock은 전부 한꺼번에 동기화 시키는 배리어의 개념이 아니면 굉장히 성능에 크리티컬 하다. 증가시키는 연산의 시간이 n이라고 한다면 한 SM에서 이를 실행 시 (SM이 실행할 수 있는 쓰레드의 수) \* n 만큼 연산의 시간이 걸려버리니 그다지 좋지 않다. 그러므로 세그먼트 전체의 크기가 절대적으로 크기 않은 이상 단순하게 CPU에서 이를 처리하는 것이 간단하게 문제를 해결하는 방법이다.

2.5.2.5 Unity상의 구현

Unity에서 제공하는 IMGUI를 사용하여 에디터에서 기능을 제공한다. 현재는 하나의 프레임만을 제공하는 기능을 제공하므로, 이를 보여주는 프리뷰 윈도우를 제공한다. 그림 27의 중간에 있는 화면으로, 실시간으로 계산하여 이미지를 보여주는 기능과 이에 관한 옵션과 제어를 위한 UI를 제공한다. 프리뷰 윈도우 외에도, 여러 configuration 값들을 변경하기 위한 UI를 제공하는 윈도우와 큰 기능들을 제어하기 위한 툴바 기능을 제공한다.

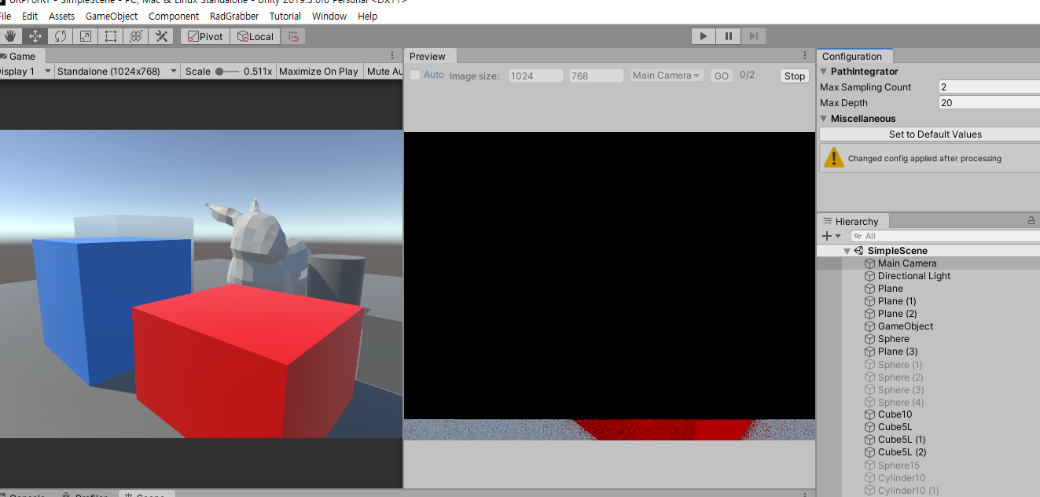


그림 : Unity 에디터상의 UI

프리뷰 윈도우는 현재 씬 상태에서 하나의 프레임을 계산하여 보기위한 용도의 UI를 제공한다. 이에 대해서 정보 변경시 자동으로 다시 시작하는 옵션 체크박스, 생성하는 이미지의 해상도를 설정하기 위한 입력필드, 생성의 기준이 되는 카메라를 Hierarchy창에서 선택할 수 있는 버튼, 생성을 시작하는 버튼과 그 생성 중에 멈추기 위한 버튼을 제공한다. 생성되는 프레임의 픽셀은 순차적으로 업데이트 된다.

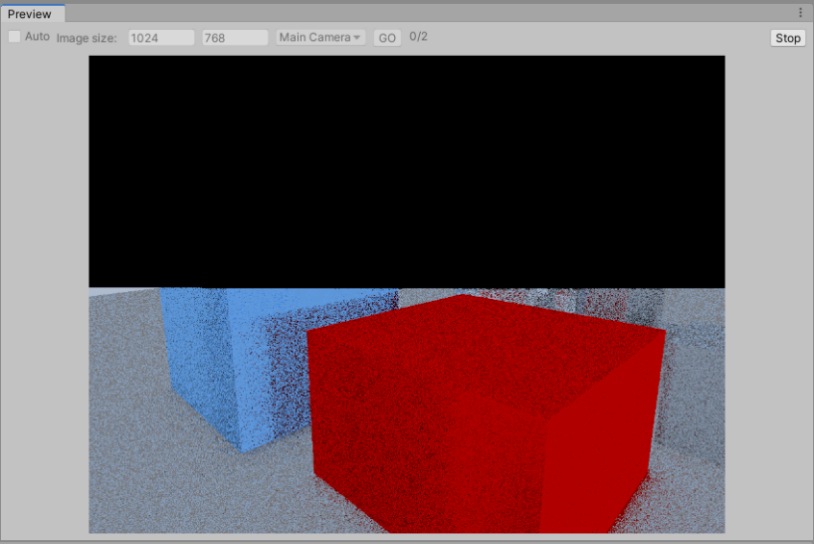


그림 : 프리뷰 윈도우

이 외에도 툴바에서 DLL을 로드/내리기 위한 기능과, 윈도우들을 켜고 끌 수 있는 기능, 생성 시에 디테일한 정보를 수정하기 위한 Configuration 창들을 지원한다.

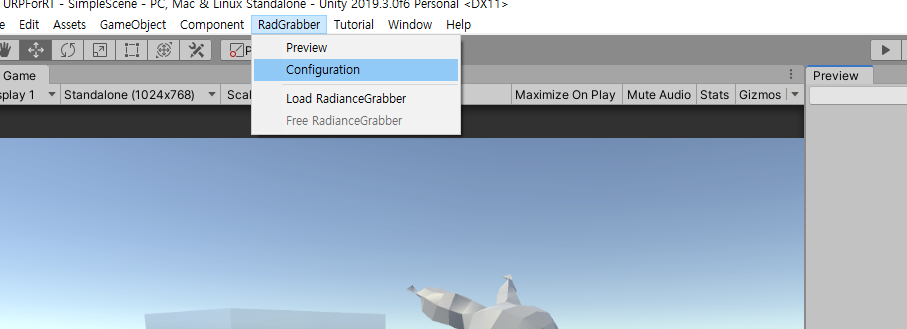
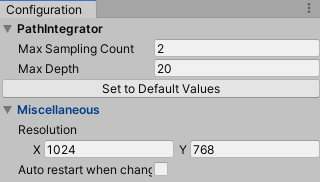
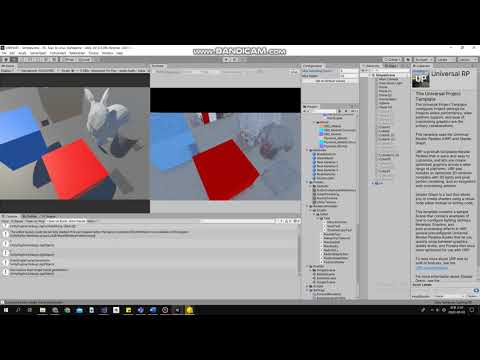
 

그림 : 툴바 UI(왼쪽), Configuration 창(오른쪽)

2.5.3 데모

[](https://www.youtube.com/embed/Ou6HaOBd3nU?feature=oembed)

영상 : 프로토타입 데모([youtu.be/Ou6HaOBd3nU](https://youtu.be/Ou6HaOBd3nU))

데모 계획은 현재 구현된 단일 이미지 생성 기능을 실시간으로 보여준다. 하나의 프레임에 최소 500번의 샘플링이 필요하다. 하지만 이 영상에서는 한번의 샘플링만 보여주기 때문에 노이즈가 잔뜩 낀 결과를 보여준다. 이는 성능의 개선과 동시에 품질 향상이 필요하다는 것을 보여주는데, 성능의 경우에는 기하 가속 자료 구조와 튜닝, 품질의 경우에는 MLT의 구현과 후처리로 개선할 수 있다.

2.5.4 향후 개발 방향

현재 구현된 부분에서 개선해야 할 것은 다음과 같다.

* 최적의 kernel launch configuration 찾기 : 각 커널 별로 다양한 예제를 통한 최적점을 시간을 들여 찾아야 한다. Occupancy Calculator 와 NVidia NSight Compute를 활용하여 최적의 메모리 사용량을 정하여 성능을 더 올릴 수 있다.
* 기하 가속 자료 구조 튜닝 : 정적 메쉬 kd-tree는 생성과 탐색이 구현되어 있으며 전체 기능 완성 이후 kd-tree 생성 시 각 폴리곤의 SAH값에 따라 분할을 시도해볼 여지가 있다.
* Path-Tracing 성능 튜닝 : 현재 픽셀의 색을 넘겨주는 부분을 CPU로 처리하고 있는데, 이를 GPU로 처리하여 RAM과 VRAM간의 메모리 동기화를 막아 성능을 튜닝할 수 있다.

현재 구현해야 할 것은 다음과 같다.

* Point Light, Spot Light를 광선 추적 알고리즘에 적용, PBRT-V3에서는 샘플링시에 해당 씬의 기하들의 전체에 한번의 광선-기하 충돌 탐색을 통해 가려졌는지 아닌지를 확인한다. 즉 한번의 광선-씬 충돌 시 PointLight / SpotLight 마다 한번씩 검사를 해야하는 것으로 볼 수 잇다.
* 멀티 프레임 데이터 연결 구현
* MLT 구현 : 일반적인 Path-Tracing은 수렴속도가 굉장히 느리기 때문에 구현되어야 어느 정도의 완성도를 가질 수 있다.
* 기하 가속 자료 구조 완성 : 물체의 경계상자 BVH, 동적 메쉬 BVH 두가지의 생성, 탐색, 업데이트가 구현 되어야 한다.

## **시험 / 테스트 결과**

프로토타입 초기에 사용된 테스트 씬을 사용하여 성능을 측정하였다. 아래 그림과 같은 도형들이 이루어져 있다. 총 8개의 MeshRenderer, 하나의 directional light, 3개의 메쉬(cylinder, rabbit, box)로 구성되어 있는 씬이다. 모든 테스트는 i7-4790, gtx970 하드웨어에서 진행되었다.

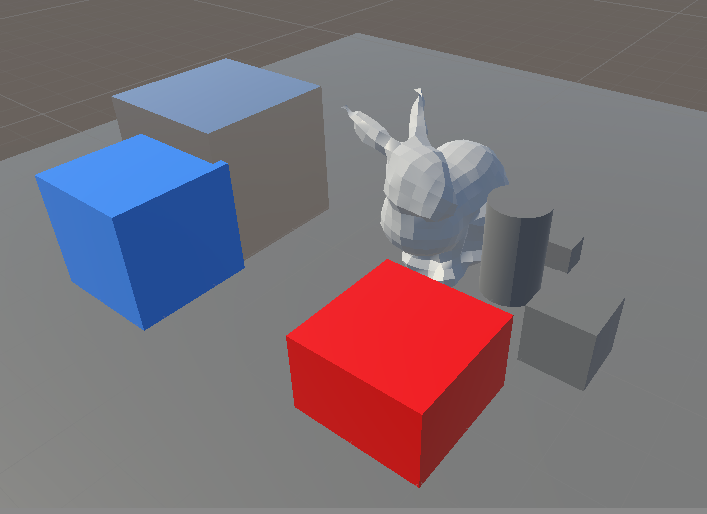


그림 : SampleScene 의 구성

커널에서의 파라미터는 최대로 올려야 하는 것을 표9에서 알 수 있다. gtx970에서는 총 쓰레드의 갯수가 66536가 한계인데, 이는 각 GPU에서의 각 멀티 프로세서 당, 최대의 쓰레드 개수 \* 워프의 크기를 곱하면 나오는 숫자이다. 테스트는 전체 파라미터의 곱이 작은 경우와 작은 수로 비교하기 위하여 경로의 깊이를 1로 설정하였다. 일반적으로 사용하는 깊이가 20정도 되므로 깊을수록 차이가 많이 날 것이다.

표 9 : 커널 파라미터 크기에 따른 성능 비교(1 sample, 1depth, 1024x768 SampleScene, Prototyping)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| thread/block | 32/64 | 32/128 | 32/256 | 32/512 | 32/1024 | 32/2048 |
| 계산 시간(s) | 24.5 | 16.7 | 7.9 | 5.0 | 3.9 | 3.3 |

커널 한번의 실행 당 정해진 쓰레드 개수가 있다면, 블록 안의 쓰레드 개수와 그리드 안의 블록 개수를 어떻게 조정해야 할지에 대한 테스트를 진행해보았다. 여러 커뮤니티에서의 경험적으로 일반적인 최적의 블록의 쓰레드 개수 값은 128, 256 이라고 하는데, 약간은 다른 결과를 나왔다. 128,256 일 때 최적인 것은 맞으나, 32/64는 보통 추천하지 않는 쓰레드의 개수라고 한다. 하지만 해당 연구에서는 64/128/256 쓰레드 개수가 최적의 결과를 낳았다.

표 10 : 블록/쓰레드 개수 밸런스에 따른 성능 비교(1 sample, 20depth, 1024x768, SampleScene, Prototyping)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| thread/block | 32/2048 | 64/1024 | 128/512 | 256/256 | 512/128 | 1024/64 |
| 계산 시간(s) | 18.4 | 16.0 | 16.1 | 16.1 | 17.6 | 21.4 |

기본적으로 커널들이 레지스터의 사용량이 많아, 쓰레드 점유율이 낮은 편이다. 어느 정도의 점유율이 적절한 성능을 낼 수 있는지 찾기 위하여 테스트를 진행해보았다. 점유율과 성능은 언제나 정비례하지 않는다고 커뮤니티에 알려져 있지만, 해당 연구에서는 점유율과 성능이 비례 하는 것으로 보인다. 이는 기본적으로 각 쓰레드에서 전역 메모리 참조를 많이 하기 때문에 절대적으로 점유율을 높이는 편이 더 나은 것으로 추측된다.

표 11 : 강제 레지스터 제한과 커널의 성능 비교표(1 sample, 20depth, 1024x768, 128/512, SampleScene, Prototyping)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 레지스터(byte) | 254(제한x) | 128 | 64 | 48 | 40 | 32 |
| 계산 시간(s) | 48.2 | 27.8 | 19.7 | 18.1 | 16.8 | 16.0 |
| 쓰레드점유율 | 0.125 | 0.25 | 0.5 | 0.66 | 0.8 | 1.0 |

PathIntergerator 모듈에서는 미리 보기 이미지를 지원하기 위한 Incremental 모드와 영상을 만들기 위한 StraightForward 모드 두가지가 존재하는데, 이 두가지 모드의 차이는 동영상 생성의 경우에는 미리보기를 필요로 하지 않기 때문에 그 차이를 이용하여 각 픽셀별 샘플링을 돌아가면서 처리하느냐, 배치처리 하느냐의 차이가 존재한다. 중점적으로 구현한 것은 샘플링 횟수에는 덜 민감하도록 구현한 것이다. 이를 검증하기 위해 실험을 하였다. 표 12에서 이를 볼 수 있다.

표 12 : 샘플링 횟수에 따른 계산 시간 비교(1 frame, 20depth, 1024x768, 128/512, SampleScene, Prototyping)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| sample count | 1 | 2 | 3 | 4 |
| Incremental 모드  계산 시간(s) | 16.1 | 33.0 | 49.6 | 66.0 |
| StraightForward 모드  계산시간(s) | 16.2 | 24.4 | 33.5 | 41.2 |

## **Coding & Demo**

2.7.1 구현 현황

전체 구현할 것들 중, 구현되지 않은 것들을 항목별로 아래에 기술한다.

* Core 기능에서 구현되지 않은 기능은. Unity에서 지원하는 메터리얼을 중, 가능한 메터리얼을 전부 지원하지 않는다. 이는 해당 연구에서의 중점적인 부분이 아니므로, 전체 구현의 우선 순위에서 뒤로 밀려나 최종 버전에서도 구현되지 않았다.
* 지오메트리 가속 구조에서 구현되지 않은 기능은, 동적 메쉬 BVH이다. 이는 프레임별로 다른 정보를 가지게 되어 첫번째로 생성된 BVH 트리에 업데이트 하는 방법으로 고안되었다. 성능의 측면에서 보자면 이는 구현되어야 하지만 일정 관리 측면에서, 정적 메쉬 kd tree 를 각 프레임별로 계산하는 것으로 대체되었다.
* MLT 또한 품질을 높이려면 반드시 구현되어야 하지만 이 또한 우선 순위에서 밀려나 최종 버전에서 구현되지 않았다.

표 : 최종 발표 버전 구현 현황

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 모듈 | | 진행 상태 | | | | |
| Core : 싱글 프레임 계산 | | 모든 기능 미지원, 시연 가능 버전 | | | 90% | |
|  | 메터리얼, 메쉬, 광원 interact | 메터리얼 부분적 지원 | | | 90% | |
|  | Unity 데이터 연결 및 에디터 지원 | 100% | | | | |
|  | Path-Tracing |  | | | 100% | |
| 멀티 프레임 계산 | | 100% | | | | |
| 지오메트리 가속 구조 | |  | 70% | | |
| 정적 메쉬 Kd Tree | | 100% | | | | |
| 동적 메쉬 BVH | | 0% | | | | |
| 물체 단위 BVH | | 100% | | | | |
| MLT | | 0% | | | | |
| 전체 진행 현황 | |  | | 80% | | |

2.7.2 구현 디테일

2.7.2.1 성능

프로토타이핑에 구현된 광선 추적 알고리즘은 성능 상으로 굉장히 좋지 않았다. 이후 이는 반드시 개선시킬 이슈가 되었다.

이를 분석하여 두 가지 이유를 찾을 수 있었다, 첫번째 이유는 커널들을 구성하는 방법이 전략적이지 않았다. SIMT 모델을 구현한 GPU의 경우에 가장 이상적인 실행 컨텍스트는 전역 메모리를 최대한 사용하지 않고, 레지스터 내에서 거의 대부분의 것을 해결할 수 있도록 하고, 모든 쓰레드가 **유효한 작업을 하면서 최대한 같은 시간**에 끝나야 한다. 그렇지만 광선 추적의 경우에는 Naïve 하게 구현한다면 알고리즘 자체가 다르게 끝날 수 있는 가능성을 가지고 있다. 더 구체적으로 말하자면, 기하 집합에서 폴리곤까지 광선 충돌을 검사할 때, 대부분의 경우 반복문을 도는 횟수가 같지 않다. 이러한 특징을 고려하지 않고 단순하게 하나의 쓰레드가 정해진 픽셀 개수만큼 광선 충돌을 검사하게 된다면, 각 쓰레드 별로 시간의 편차가 생기게 되고, 이는 더욱더 성능을 갉아먹는 요인이 된다. 앞서 언급한 “하나의 쓰레드가 정해진 픽셀만큼 충돌 검사” 가 성능하락의 중요한 이유였다. 정해진 시간 만큼의 실행, 정해진 명령어 개수 만큼 실행이 되어야 이를 효과적으로 실행시킬 수 있는 것이다. 문제는 정확한 정해진 개수 만큼의 명령어 실행은 불가능하기 때문에 적절한 함수 수준에서 기능을 잘게 나누어 빠르게 연속적으로 태스크를 분배하는 것이 최대한의 효율을 낼 수 있는 방법이다.

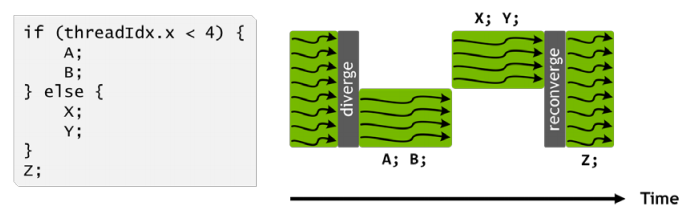


그림 : thread divergence by branching

프로토타이핑에서는 다음과 같이 커널을 구성했다.

1. N = 최대 쓰레드의 개수
2. N개의 픽셀에 대하여
   1. 커널 실행 : Path-Tracing 충돌 검사, 임의의 횟수만큼 충돌 검사
   2. 커널 실행 : Infinite Light 가려짐 검사, 1번의 충돌 검사
   3. 커널 실행 : 색 계산
3. 다음 N개의 픽셀 설정
4. 처리할 픽셀이 없을 시 종료, 아니면 2번으로

이는 앞서 언급한 문제를 가지고 있으므로 아래와 같이 바꾸었다.

1. M = 프레임의 픽셀의 개수
2. 커널 실행 : M개의 픽셀 / Path-Tracing 충돌 검사, 임의의 횟수만큼 충돌 검사
3. 커널 실행 : M개의 픽셀 / Infinite Light 가려짐 검사, 1번의 충돌 검사
4. 커널 실행 : M개의 픽셀 / 색 계산

두번째 이유는 CUDA 커널 실행 시 필요한 파라미터: 각 블록 당 쓰레드의 수, 그리드 당 블록의 개수를 해당 커널의 특징에 맞추어 설정해야 한다. 프로토타이핑에서는 두가지 파라미터를 잘못 세팅하여 퍼포먼스가 많이 떨어졌는데, 이를 최대 값으로 늘릴 시 더 나은 성능향상을 볼 수 있었다.

이를 가장 쉽게 접근하기 위해 일반적으로 알려진 최적화를 사용하였다. 이는 쓰레드/워프 별로 사용하는 자원의 수를 줄여 각 블록의 쓰레드들의 점유율(Occupancy)을 높이는 방법으로, 이에 영향을 미치는 자원은 쓰레드 별 사용하는 레지스터의 개수, 공유 메모리 사용량이 있고, 블록의 개수 또한 점유율에 영향을 미친다. 모든 커널에서의 공유 메모리 사용량은 적고, 블록의 개수는 픽셀의 개수에 비례하기 때문에 대부분의 경우 픽셀의 개수가 충분히 많아서 상관이 없다. 결국 점유율에 가장 큰 영향을 미치는 것은 쓰레드 별로 사용하는 레지스터의 개수를 줄이는 것이다. 이는 코드 자체의 최적화가 필요하므로 많은 시간이 걸린다. 우선은 점유율를 높이기 위하여 컴파일러 옵션을 통하여 레지스터의 개수를 (SM안의 레지스터 크기 / 블록 별 쓰레드의 개수) 로 제한하는 옵션을 주어 점유율을 최대로 늘렸다. 각 커널들 중 레지스터를 가장 많이 사용하는 경우는 광선의 물체 충돌 검사를 하는 커널에서 254바이트의 레지스터를 사용했으며, gtx970에서는 점유율을 최대로 늘리기 위해서는 32바이트의 레지스터가 한계이다. 단순히 점유율을 높이는 것이 무조건 적인 답은 아니지만, 표 11에서 보이는 테스트 결과를 보았을 때, 점유율이 높은 것이 더 나은 성능을 보이는 것으로 고려된다.

프로토타이핑과 비교하면, 5~10배의 성능 향상이 존재했으나, 이는 알고리즘의 개선보다 CUDA 파라미터 세팅 및 컴파일 옵션 같은 부수적인 최적화를 통하여 얻은 결과였다. 더 강력한 최적화 개선을 위해서는 완전히 다른 아이디어가 필요할 것으로 생각된다.

2.7.2.2 동영상 생성

하나의 이미지를 계산하는 것 이후, 영상 계산을 위해서는 각 프레임별 달라지는 데이터와 달라지지 않는 데이터를 구분하여 각 프레임별 필요한 데이터를 선별해야 했다. 분류한 결과는 다음과 같다. (표 5 참고)

* 유지되는 데이터 : 정적 메쉬, 스킨 메쉬, 텍스쳐
* 프레임 별 데이터 : 카메라, 광원, 메쉬 렌더러, 스킨 메쉬 렌더러, 메터리얼, 스카이박스

전체 프레임에서 유지되는 데이터는 두가지 메쉬와, 텍스쳐가 있다. 특히 메쉬는 그 자체의 크기가 크고, 거의 모든 메쉬는 런타임에서 변하지 않는다. 거기에 메쉬 별로 kd-tree를 생성하는 계산을 해야 하는데, 이는 초기 단계에서 가장 시간을 많이 소요하므로 분류해야 할 필요가 있었다. 텍스쳐의 경우, 일반적으로 사용되는 텍스쳐의 크기는 512\*512, 1024\*1024, 2048\*2048 정도의 크기가 있는데, sRGB를 사용하는 텍스쳐의 경우 각 픽셀별 크기가 3바이트 이므로, 압축하지 않은 텍스쳐를 사용한다고 하면, 2048\*2048 크기의 텍스쳐는 12MB의 크기를 가진다. 이는 보통 렌더링에서 LUT로 사용되므로, 렌더링 데이터 중 가장 큰 크기의 데이터이므로, 반드시 분류되어야 했다.

각 프레임별 다른 데이터는 성능, 데이터 크기의 측면에서 크게 문제되지 않는 데이터들을 전부 취급한다. 또한 이들은 대부분 변경되는 데이터이므로 각 프레임별로 전부 저장한다. 특히 카메라, 광원, 메쉬 렌더러, 스킨 메쉬 렌더러는 하나의 오브젝트 별로 위치, 회전, 크기 값을 전부 가지고 있는데, 이는 거의 매 프레임별로 바뀌는 데이터이고, 메터리얼이나 스카이박스는 간헐적으로 변경된다.

이렇게 분류된 데이터는 하나의 이미지를 계산하는 경우에는 변경되지 않는 데이터 하나, 변경되는 데이터 하나를 포함한 파라미터를 통해 요청되고, 동영상을 계산하는 경우에는 변경되지 않는 데이터 하나, 변경되는 데이터는 프레임의 개수만큼 포함한 파라미터를 통해 요청된다.

Unity 상에서 동영상 생성에 요청할 여러 개의 변경되는 데이터를 처리하기 위해선 여러가지 방법이 고려되었지만 Unity에서 지원하는 기능을 최대한 사용하여 간단하게 구현하였다. Unity 에서는 각 프레임별로 움직이는 것을 각 프레임을 기준으로 데이터로 저장하고, 저장한 데이터를 기준으로 각 오브젝트의 데이터를 시간을 기준으로 보간하여 런타임에서 즉각적으로 데이터를 변경하는 애니메이션 기능을 가지고 있다. 저장된 데이터는 AnimationClip이라는 클래스로 지원되는데, 해당 연구에서는 이 데이터 형식을 통해 변경되는 데이터 전부를 AnimationClip안에 전부 넣은 후 AnimationClip의 데이터를 각 프레임별로 샘플링 하는 방법을 선택했다.

또한 동영상을 생성하기 위해서는 여러 개의 이미지를 생성하여 계산한다 명시해 놓았는데, 이를 전체 계산 과정에서도 동일하게 구동된다. 만약 저장하지 않고 동작하게 된다면 프레임의 길이가 길어질수록 계산 중에 사용하는 메모리가 많아진다. 1920\*1080 해상도의 경우에 한 프레임 당, 약 6MB를 차지하게 되는데, 프레임 갯수가 천, 만 단위로 넘어가게 된다면 GB 단위의 메모리를 점유하는 상황을 만든다. 그래서 하나의 프레임이 계산이 끝나면 임시의 위치에 각 이미지를 저장한 후, 모든 프레임의 계산이 완료되면 모든 프레임을 영상으로 만들어 저장한다.

이를 빠르게 구현하기 위하여 이 기능들을 전부 지원하는 OpenCV를 사용하였다. 안정성을 위해 C/C++을 통해 사용할 수 있는 OpenCV 라이브러리를 사용하여 DLL 안에서 이를 래핑하여, Unity 상에서 이를 호출하여 사용한다. 조금 더 명시적으로 말하자면, 광선 추적 계산 요청 시 각 프레임별로 처리가 끝나면 픽셀을 담고있는 배열과 프레임의 숫자를 넣어 호출해주는 콜백 함수의 포인터를 넣을 수 있는데, 이를 사용하여 C# 스크립트에서 각 콜백 호출 시 이미지를 저장하는 루틴을 호출한다. 이는 복잡한 구현으로 보인다. 이는 구현시에 OpenCV를 C#으로 래핑한 라이브러리를 사용할 예정으로 콜백 기능을 DLL 밖에 접근할 수 있도록 한 것이나, Unity 상에서의 C# 런타임과 OpenCV를 C# 래핑한 라이브러리와의 호환을 맞추지 못하여 다음과 같은 구현이 되었다.

2.7.2.3 Unity 프로젝트에서의 구현

Unity C#을 통한 대부분의 구현은 UI 와 값을 저장하여 이를 전달하기만 하는 간단한 브릿지 역할을 수행하는 구현체들이 대부분을 이룬다.

첫번째는 2.5.2.5절에서 언급하고, 그림 27에서 보여준 프리뷰 윈도우로, 이는 영상을 만들기 전에 미리 보는 기능을 제공하기 위한 창이다. 일종의 미리보기 창이기 때문에 최대한 해당 윈도우에서 필요한 기능을 제공한다. 미리보기 이기 때문에 계산된 픽셀을 실시간으로 보여준다. 이 때문에 기본적인 코드 구조가 두가지 흐름으로 나뉜다. 하나는 실시간으로 바로바로 업데이트 하는 부분이고, 하나는 동영상 생성을 위해 즉각적인 업데이트가 아닌 이미지 별로 업데이트 하는 코드 흐름이다.

두번째는 실질적으로 영상을 생성하는 에디터 기능으로, 이를 위해 아래와 같은 세팅을 요구한다.

1. 애니메이션을 실행하려는 모든 GameObject의 루트가 존재해야 한다. 즉 하나로 Hierarchy 를 묶어야 한다.
2. GameObejct를 기준으로 세팅된 AnimationClip이 필요하다. 만약 에디팅된 GameObject의 구조와 이름이 다르다면 샘플링시에 유효한 값을 가져올 수 없기 때문에 영상에서는 반영되지 않는다.

이러한 AnimationClip 데이터에 애니메이션 자체를 넣어두기 위하여 런타임 상에서 모든 데이터를 프레임별로 저장하여 AnimationClip으로 저장해주는 GameObjectRecorder 라는 Unity 에디터 상에서 지원하는 기능을 통하여 쉽게 저장하게 하였다. 그림 32의 HierarchyRecorder 클래스를 Root GameObject에 붙여서 사용할 수 있다. 이를 사용한 워크플로우는 전체적인 정보를 애니메이션으로 한 번에 녹화한 후, 디테일한 수정은 그림 33에서 볼 수 있는 Untiy 에디터상에서 지원하는 프레임별 애니메이션을 수정하는 에디터 윈도우를 사용하여 동영상을 계산하는 것으로 고안되었다.

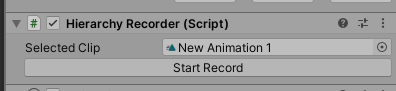


그림 32 : 런타임의 녹화 지원 클래스 : Hierarchy Recorder, 선택된 AnimationClip에 저장



그림 : Hierarchy와 Animation Clip 에디터

이러한 세팅이 된 이후에 영상을 생성하기 위한 에디터 화면을 지원한다. 이는 앞서 세팅된 Root gameobject와, AnimationClip을 가져와 넣어주고, 맨 아래의 생성 버튼을 누르게 되면 영상 생성이 시작된다. 영상 생성 에디터 화면에서는 이미지 임시 생성 디렉토리, 영상의 경로 같은 기본적인 정보부터, 광선 추적 알고리즘의 설정 값을 세팅할 수 있다.

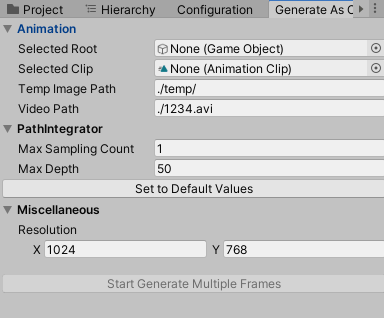


그림 : AnimationClip을 통한 영상 생성 에디터 윈도우

이 외에도 광선 추적 알고리즘에 필요한 설정 값들과 해당 연구의 구현체와 OpenCV 등 라이브러리 DLL이 존재하는 경로를 세팅하는 기능을 제공하는 에디터 윈도우를 사용할 수 있다.



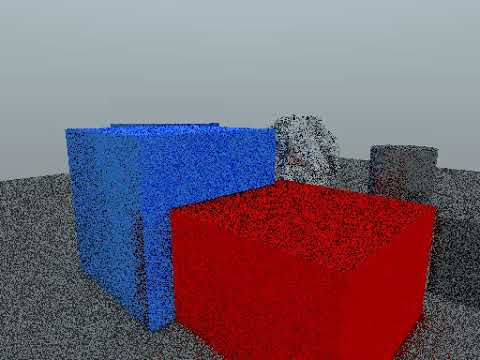
그림 : Configuration 에디터 윈도우

2.7.3 데모 시나리오

데모를 진행하여 보여줄 만한 중점적인 기능은 광선 추적을 계산하여 동영상을 생성하는 것이다. 하지만 이를 시연하여 보여주기 위해선 현재 버젼에서는 많은 시간이 걸리기 때문에 실질적인 결과를 보여주기 위해선 에디터 UI 상에서의 간단한 동작을 시연한 다음, 만들어진 동영상을 보여준다.

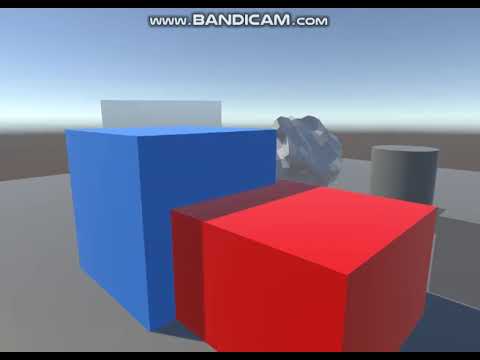
구체적인 시나리오는 다음과 같다.

1. 광선 추적을 통하여 만들어진 동영상을 보인다.

[](https://www.youtube.com/embed/wMBqn0tpi_o?feature=oembed)

영상 : 만들어진 영상

1. Unity 에디터에서 해당 원본 애니메이션을 보인다.

[](https://www.youtube.com/embed/BetucVbCl0Q?feature=oembed)

영상 : 원본 애니메이션

1. Unity 에디터에서 이미지를 미리 보여주는 것을 시연한다.
   1. 이 기능을 시연할 때, 오브젝트의 위치를 바꿔가며 실시간으로 반영이 되는 것을 보인다.

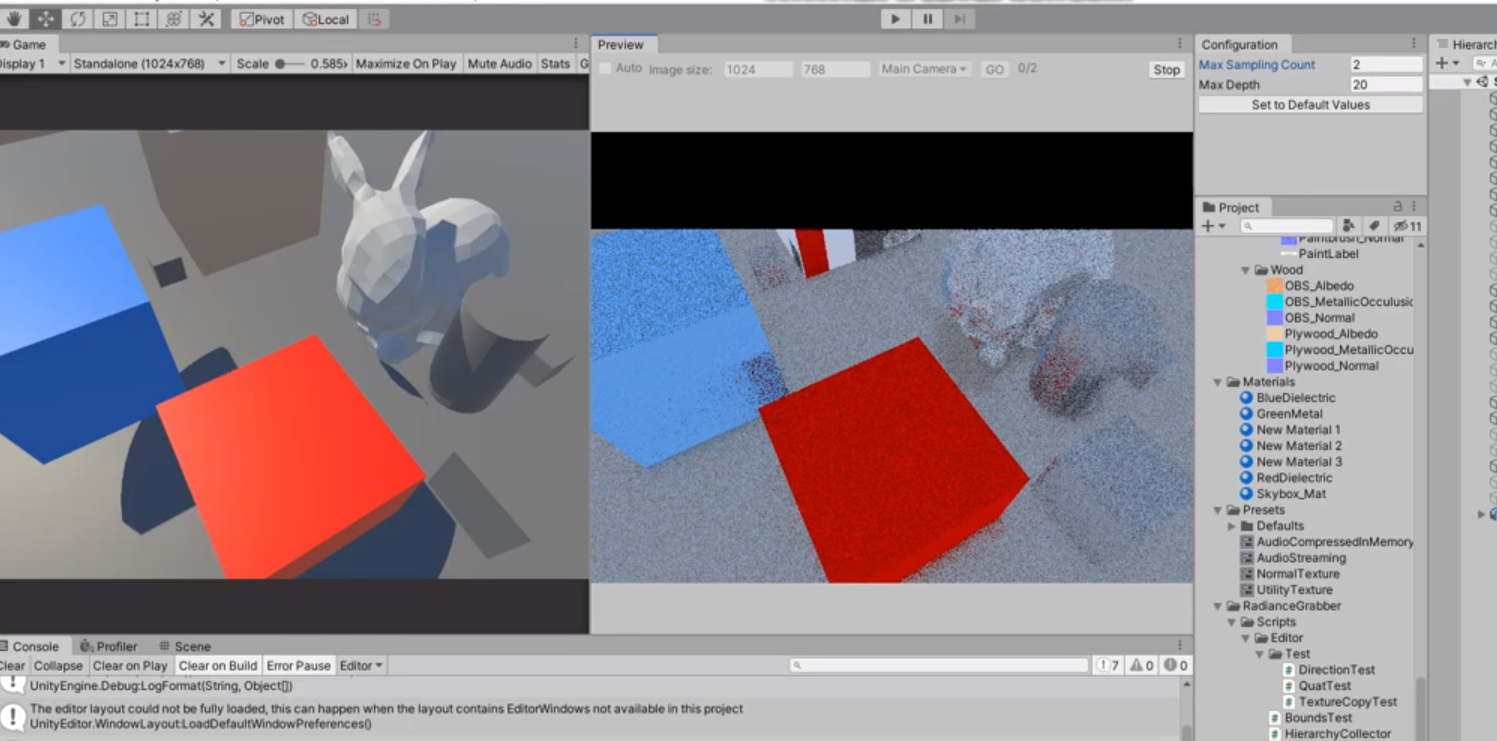


그림 : 프리뷰 이미지 생성 시연

# **참고문헌**

|  |  |
| --- | --- |
| [1] | D. Lin, K. Shkurko, I. Mallett and C. Yuksel, "Dual-Split Trees," in *Symposium on Interactive 3D Graphics*  *and Games*, 2019. |
| [2] | C. Lauterbach, M. Garland, S. Sengupta and D. P. Leubke, "Fast BVH Construction on GPUs," in *EuroGraphics*, 2009. |
| [3] | H. Ylitie, T. Karras and S. Laine, "Efficient Incoherent Ray Traversal on GPUs Through Compressed Wide BVHs," in *High-Performance Graphics*, 2017. |
| [4] | E. P. Lafortune and Y. D. Willems, "Bi-directional Path Tracing," 1998. |
| [5] | E. Veach and L. J. Guibas, "Metropolis Light Transport," 1997. |
| [6] | C. Keleman and L. Szirmay-Kalos, "Simple and Robust Mutation Strategy for Metropolis Light Transport Algorithm," DBLP, 2002. |
| [7] | T. Hachisuka, A. S. Kaplanyan and C. Dachsbacher, "Multiplexed Metropolis Light Transport," 2014. |
| [8] | B. BITTERLI, W. JAKOB, J. NOVÁK and W. JAROSZ, "Reversible Jump Metropolis Light Transport using Inverse Mappings," arxiv.org, 2017. |
| [9] | W. Joss, J. W. Mark and M. Rafal, "Analysis of reported error in Monte Carlo rendered images," *The Visual Computer,* pp. 705-713, 2017. |
| [10] | P. Shirley, “Ray Tracing in One Weekend,” 2018. |
| [11] | P. Shirley, “Ray Tracing in The Next Weekend,” 2018. |
| [12] | P. Shirley, “Ray Tracing in The Rest of Your Life,” 2018. |
| [13] | R. Allen, “Accelerated Ray Tracing in One Weekend in CUDA,” Nvidia, 05 11 2018. [온라인]. Available: https://devblogs.nvidia.com/accelerated-ray-tracing-cuda/. [액세스: 28 01 2020]. |
| [14] | M. Pharr, W. Jakob and G. Humphreys, Physically Based Rendering, Morgan Kaufmann, 2016. |
| [15] | D. B. Kirk and W.-m. W. Hwu, 대규모 병렬 프로세서 프로그래밍, BJ 퍼블릭, 2010. |
| [16] | J. Sanders and E. Kandrot, CUDA By Example, NVidia, Addison-Wesley, 2010. |
| [17] | E. Haines and T. Akenine-Möller, Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs, Apress, 2019. |

1. “Digital Content Creation”의 약자, 여기서는 Maya, 3DS MAX 같은 3D 에디팅툴을 일컫는다. [↑](#footnote-ref-1)
2. 제안서 발표 코멘트 : “입력데이터의 scope를 잘 정의 할 것 구체화 필요” [↑](#footnote-ref-2)
3. 제안서 발표 코멘트 : “작더라도 확실한 성과가 나올 수 있도록 할 것” [↑](#footnote-ref-3)
4. Image Quality Assessment, 이미지의 품질 자체를 평가하는 방법 [↑](#footnote-ref-4)
5. 제안서 발표 코멘트 : “목표치에 대한 검증 방법 완성도가 나와야함” [↑](#footnote-ref-5)
6. <https://github.com/mmp/pbrt-v3> [↑](#footnote-ref-6)
7. 2010년대 초반부터 유행한 일종의 패러다임으로, 기존의 정립이 되어있지 않은 렌더링 방식을 실제의 단위계부터 시작하여 재질의 빛의 반사의 형태 또한 실제의 경우와 비슷하게 근사하여 렌더링하는 패러다임을 말한다. [↑](#footnote-ref-7)
8. 하나의 기능만 지원하는게 아닌 파라미터와 여러 렌더링 방식을 지원하는 쉐이더, 보통 다중 컴파일을 통하여 서로 다른 프로그램의 형태로 지원된다. [↑](#footnote-ref-8)