지도교수	지도교수
공기석	노영주

종합 설계 프로젝트 수행 보고서

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

목차

종합설계 프로젝트 수행 계획서

1. 서론

1.1. 작품선정 배경 및 필요성

1.1.1 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.

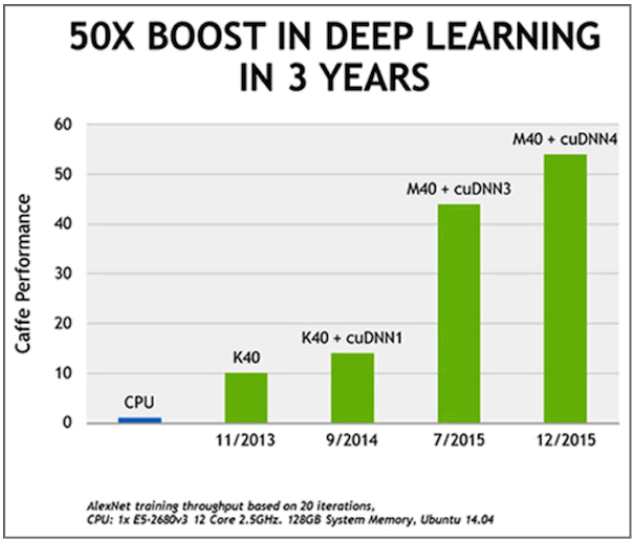

그림 1 : AlexNet의 CPU 및 각종 GPU & 라이브러리와의 성능 비교

1.1.2 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.

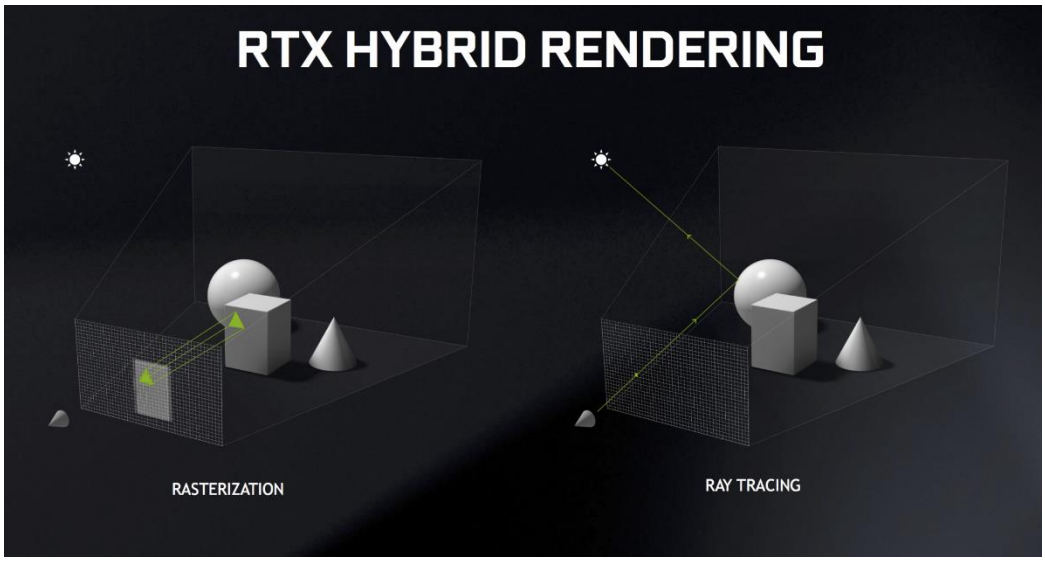

그림 2 : rasterization vs ray-tracing

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 가이를 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.

1.1.3 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.

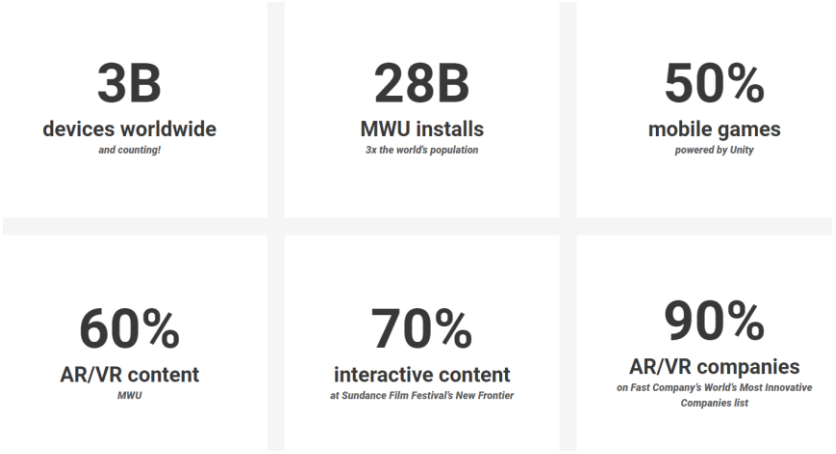

그림 3 : Unity 사용률 (Powered by Unity)

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.

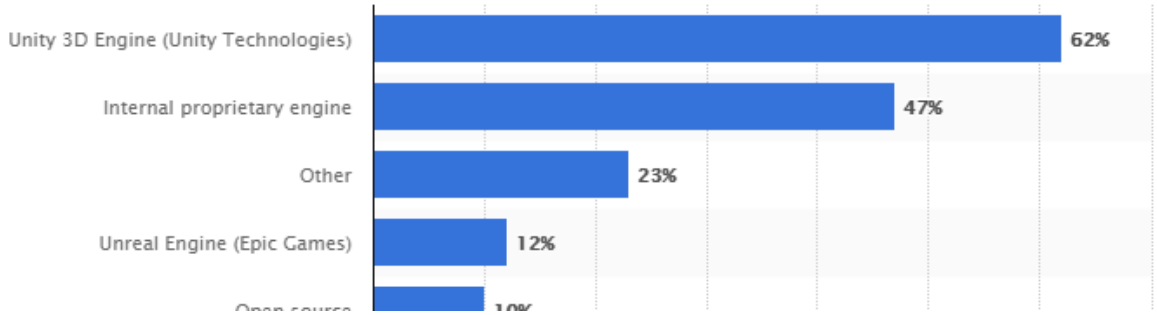

그림 4 : 엔진 사용 통계(영국, 2014, 100명의 응답, 비디오 게임 개발자 대상)(stastista.com)

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

1.1.4 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.

1.2. 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, OctaneRender 처럼 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] / 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.

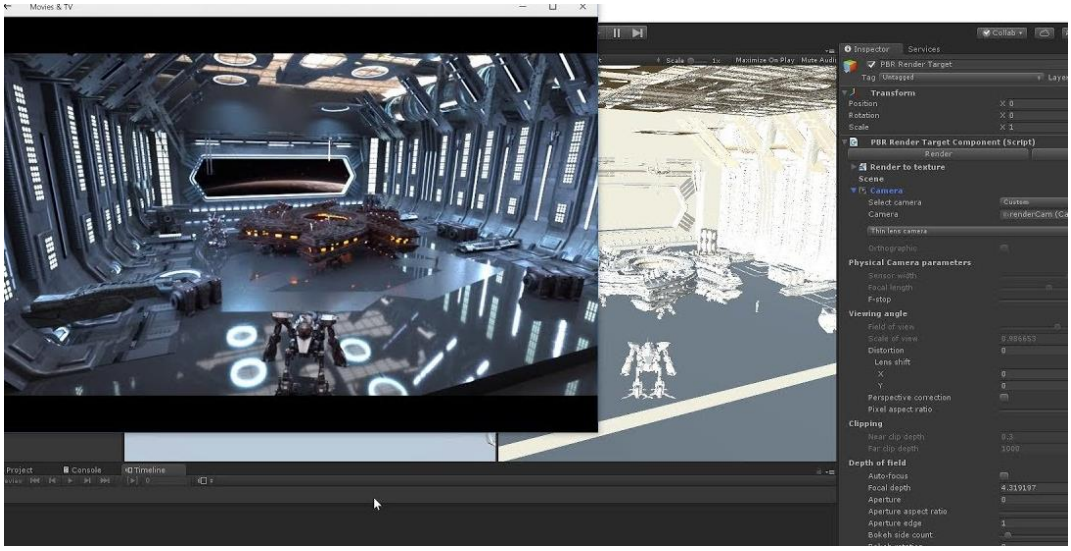

그림 6 : OctaneRender in Unity

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.

[1] "Digital Content Creation"의 약자, 여기서는 Maya, 3DS MAX 같은 3D 에디팅툴을 일컫는다.

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

1.3. 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.

1.4. 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		

1.5. 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								

1.6. 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

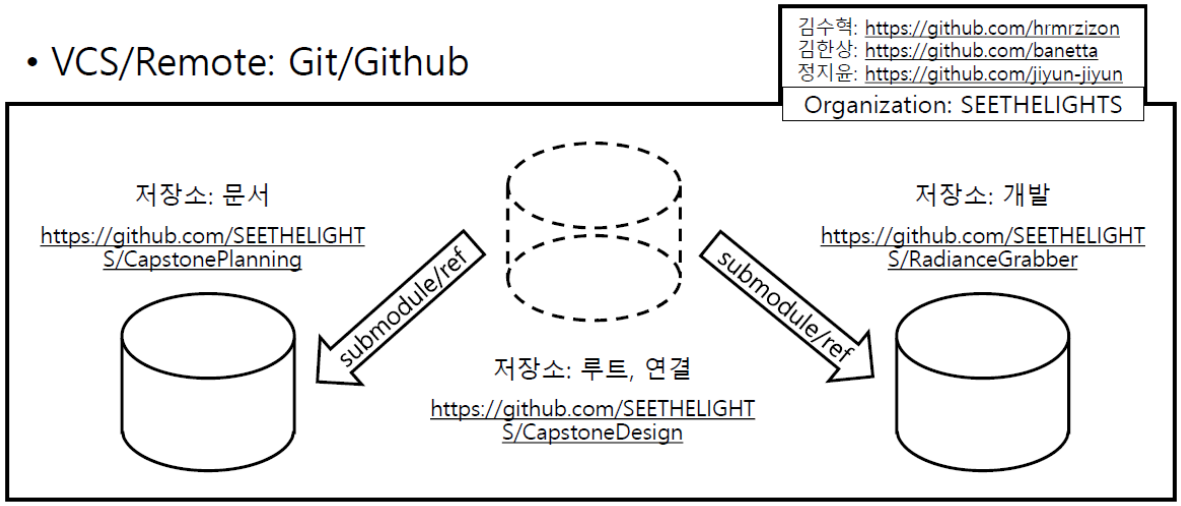

그림 7 : Git 저장소 세팅

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.

2. 본론

2.1. 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 포인

터를 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.

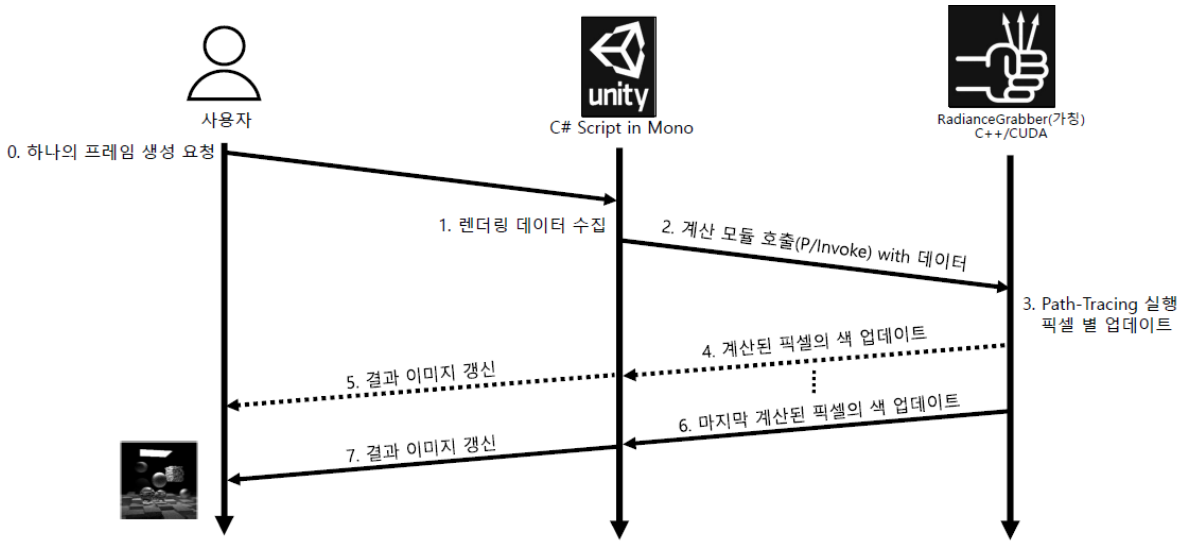

그림 8 : 시스템 수행 시나리오

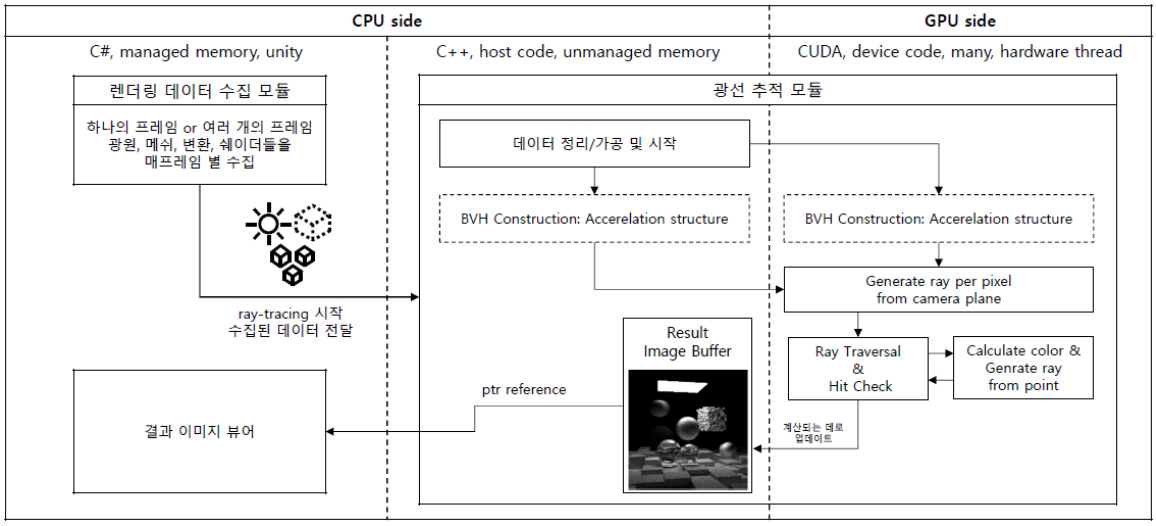

그림 9 : 시스템 구성도

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] 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.

---

[2] 제안서 발표 코멘트 : "입력데이터의 scope를 잘 정의 할 것 구체화 필요"

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.

Table 16.1: Percentage of Traced Paths That Carried Zero Radiance. With these two scenes, both path tracing and BDPT have trouble finding light-carrying paths: the vast majority of the generated paths don't carry any radiance at all. Thanks to local exploration, Metropolis is better able to find additional light-carrying paths after one has been found. This is one of the reasons why it is more efficient than those approaches here.

	Path tracing	BDPT	MMLT
Modern House	98.0%	97.6%	51.9%
San Miguel	95.9%	97.0%	62.0%

그림 10 : 0의 에너지를 발견한 경우, PT vs BDPT vs MMLT (pbr-book.org)

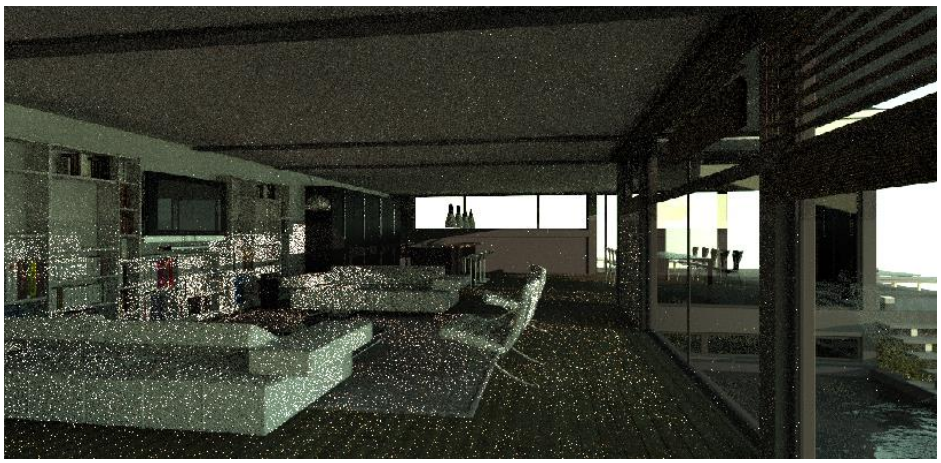

그림 11 : MMLT(위 절반), Path-Tracing(아래 절반), 같은 시간의 계산결과 (pbr-book.org)

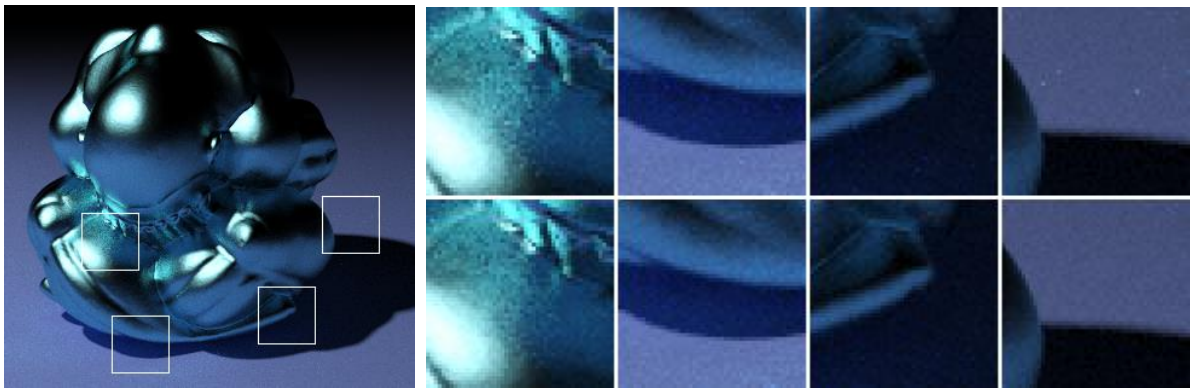

그림 12 : PT의 결과, noise(좌측), 확대한 이미지(우측/위), variance reduction한 결과(우측/아래)

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]

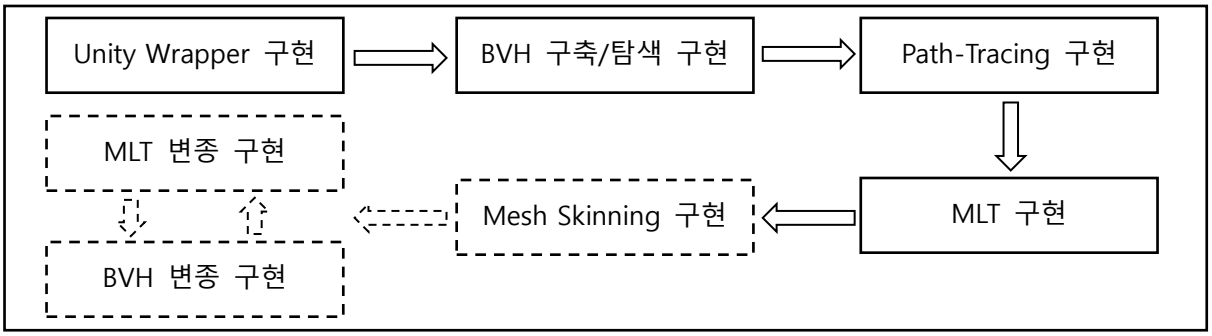

그림 13 : 구현 순서도

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] 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]

[3] 제안서 발표 코멘트 : "작더라도 확실한 성과가 나올 수 있도록 할 것"

[4] Image Quality Assessment, 이미지의 품질 자체를 평가하는 방법

[5] 제안서 발표 코멘트 : "목표치에 대한 검증 방법 완성도가 나와야함"

2.2. 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]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].

[6] https://github.com/mmp/pbrt-v3

2.3. 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.

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

2.4. 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.

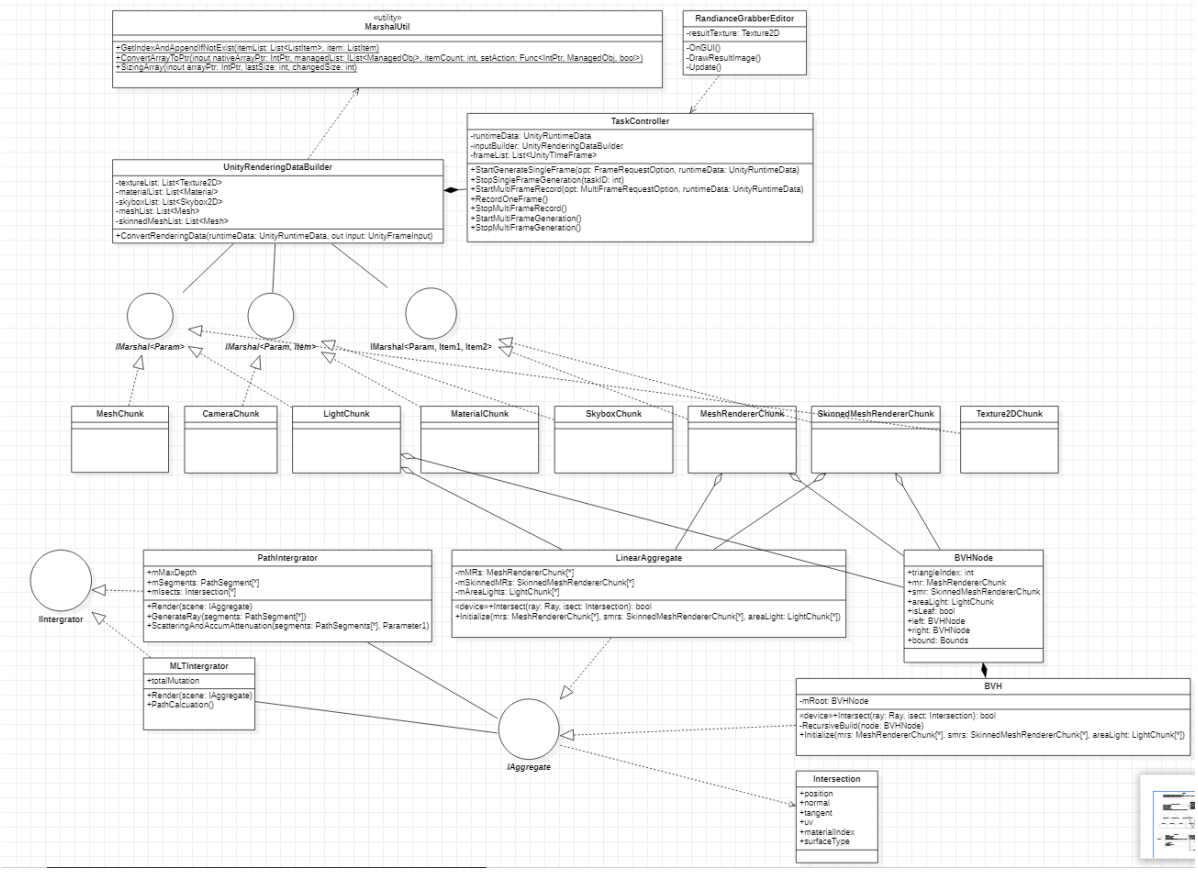

그림 14 : 클래스 다이어그램, 위쪽은 데이터 처리 모듈과 아래쪽은 계산 모듈

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 6 : 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 들어와야 함.
파라미터3	List<Material> maerialList	SkinnedMeshRenderer 에서 참조하는 Material 를 저장하기 위한 리스트, 미리 할당된 리스트가 들어와야 함.
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.

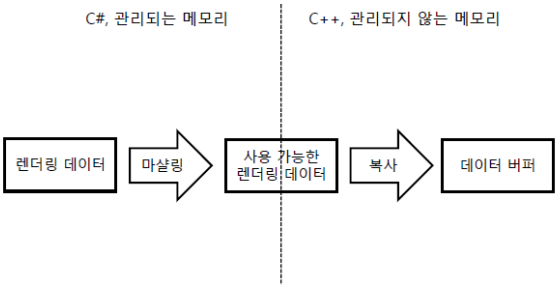

그림 15 : RadianceGrabber의 마샬링

그림 16에서는 Path-Tracing의 전체 알고리즘 순서도를 보여준다. 기하의 충돌 판정 그리고 부딪친 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.

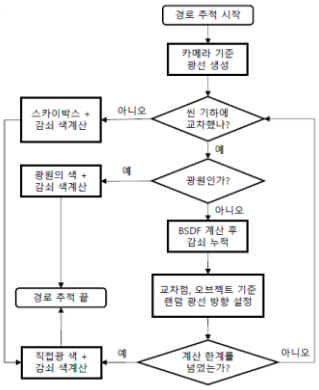

그림 16 : Path-Tracing 알고리즘 순서도

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.

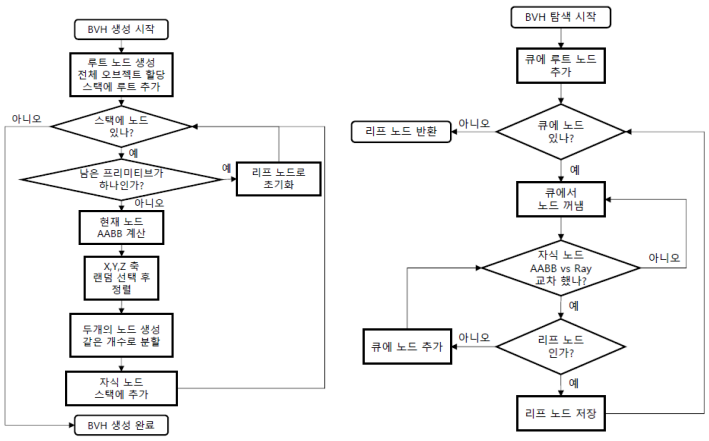

그림 17 : BVH 생성 순서도와 탐색 순서도

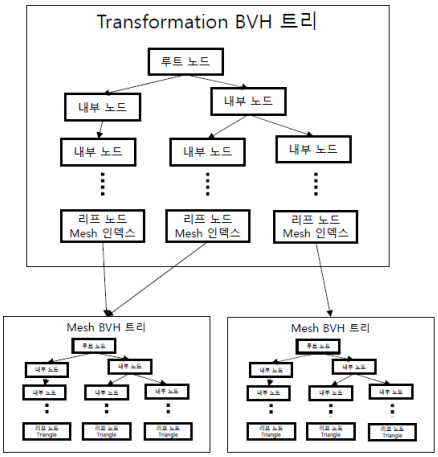

그림 18 : BVH 2단계 생성 전략

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 7 : 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 실행되어야 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	

2.5. Prototype 구현

프로토타이핑의 목적에 따라서 구현의 큰 목표는 “시연 가능한 구현체”이다. 이에 맞추어 path-tracing을 통해 이미지를 생성할 수 있는 구현체가 필요하다. 그리하여 기본적인 동작을 하는 것을 목표로 프로토타이핑을 한다. Path-Tracing 알고리즘의 구현, Unity 상에서의 동작 구현이 최소한의 요구 조건이고, 이후에 BVH, MLT를 구현하는 것을 이정표로 잡는다.

프로토타입의 시연 계획은 Unity 상에서의 에디팅을 통해 계산하는 과정을 직접 보여주는 것으로 한다. 아래 그림 19의 화면은 OctaneRender의 사용 영상 중 일부인데, 이와 같이 실제 Unity 상에서 에디팅을 Scene, Game 윈도우를 통해 조작하는 것을 보여줌과 동시에 에디팅한 물체들이 다른 에디터 창에서 광선 추적을 통해 그려진 것을 보여줄 수 있게 한다.

구현의 형태는 Unity 프로젝트 내부에 광선 추적기 코드를 포함하고 있는 dynamic link library를 넣고, 이를 만들어진 스크립트를 통해 그 안의 함수를 호출하여 Unity 에디터에서 계산하도록 한다.

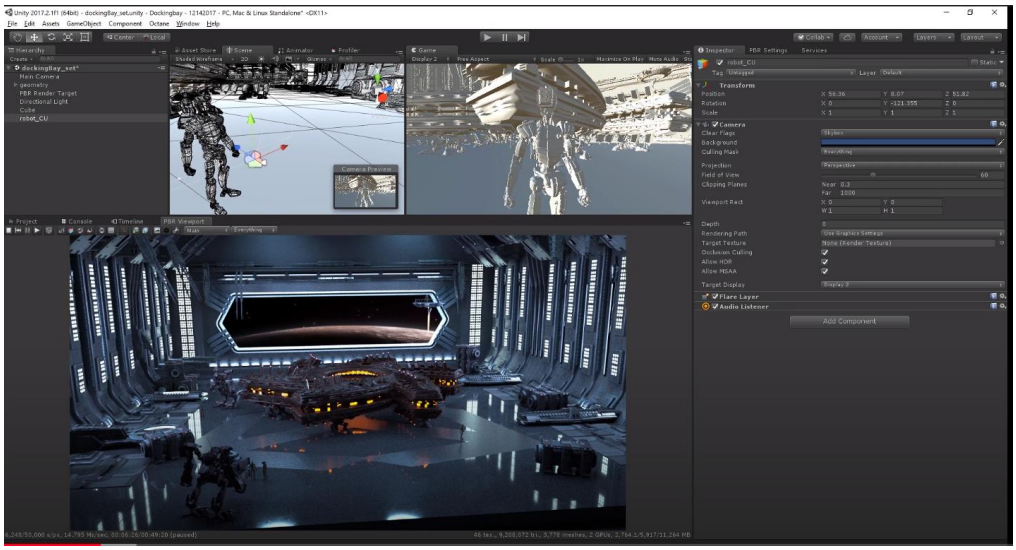

그림 19 : Octane Render in Unity

3. 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.