|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **GPU-Accelerated Raytracing Renderer for Unity(Plugin)** | | | | |
| 지도교수 - 팀원 | | 공기석 - 김수혁 | **유사사례** | [Octane Renderer at Unity(Youtube)](https://youtu.be/k0U_-9Mjuxw?t=120)  [UE4: Reflection Real-Time Ray Tracing Demo](https://www.youtube.com/watch?v=J3ue35ago3Y)  [Disney: Hyperion Render: San fRansokyo](https://youtu.be/QTA2Qt5Wk-A) |
| 노영주 - 김한상, 정지윤 |
| **작품 설명** | real-time rendering 을 하는 상용 엔진에서 씬 렌더링 데이터를(Camera, Mesh, Material, Light) 가져와GPGPU(CUDA/OpenCL) 를 사용하여 [raytracing](https://youtu.be/frLwRLS_ZR0) 기법으로 photorealistic image 를 생성해내는 plugin 형태의 S/W 이다.  기대되는 활용 방안은 1. 응답성이 중요한(16ms~32ms의 시간) real-time graphic application 을 상용 게임 엔진을 통해 개발하고, 개발 중인 컨텐츠가 사실적인 그래픽을 모방하고자 할 때, raytracing 기법을 사용하여 사실적인 이미지를 만들어 내어 컨텐츠에서 사실적인 빛의 밝기를 모방할 때 [reference image 로 사용](https://youtu.be/k0U_-9Mjuxw?t=120)할 수 있고, 2. 상용 엔진을 통해 애니메이션 같은 영상 컨텐츠를 제작할 때, 사실적인 빛의 묘사가 필요하다면, 연속적인 이미지를 생성하여 [비디오](https://youtu.be/QTA2Qt5Wk-A)로 만들 수 있다.  필요한 선행 사항들은 효율적인 raytracing 기법의 research 와 속도 향상을 위해 CPU에서 돌아가는 프로그램이 아닌 GPGPU를 활용하기 위한 CUDA 혹은 OpenCL 을 사용한 GPU 에서 돌아가는 프로그램을 짜기 위한 학습, 상용 게임 엔진을 어떤 것을 사용할지 결정할 것이 필요하다. 현재 정해진 사항은 raytracing 기법은 위 링크에서 언급된 Octance Render 에서 사용하는 방법인 [MLT](https://www.youtube.com/watch?v=f0Uzit_-h3M) 를 사용하고, GPGPU 기술은 CUDA를 사용할 것이다. MLT 기법과 여러 optimization 에 대한 부분이 과제로 남아있다. 상용 엔진은 Unity를 사용할 예정이다.  아래에는 naïve한 raytracing 기법을 구현해 만들어진 이미지가 있으며, 현재 CPU 기반, CUDA 기반으로 구현되어 있습니다. noise가 꽤 많이 존재하는데, 이는 MLT 기법으로 개선할 수 있다. | | | |
| **작품 구성도** | | | | **기술 프로토타입** |
| raytracing_one_page_proposal (1) | | | |  |

|  |  |
| --- | --- |
| **기술 개발 목표** | |
| 1. Unity 플러그인의 형태로 해당 씬의 지오메트리/메터리얼/라이트 정보를 가지고 Ray-Tracing 기법을 통한 FHD 이미지를 수십초 이내에 reference image / single frame 을 생성하는 기능을 목표로 한다.  A. 픽셀 별 샘플링 횟수가 100~300(spp)를 가지면서 노이즈가 없는 이미지를 생성할 수 있어야 한다.  2. 다른 엔진/DCC와의 확장성을 위해 ray-tracing 모듈은 Unity 와는 독립적인 구조를 가져야 한다.  3. 하나의 컴퓨터에서 여러 개의 GPU를 가진 경우 이를 활용하여 더 빠른 계산을 할 수 있도록 한다. | |
| **기술 개발 전략** | |
| 구현  및  모델 | 1. **Unity 와 path-tracer 간의 데이터 연결 모듈** 2. Unity 에서의 Rendering 데이터 수집 기능: 해당 씬의 Mesh / Instance(Transformation) / Material / Light 정보들을 매 프레임 별로 수집한다. (Unity C# 으로 구현) 3. Ray-tracing 모듈의 연결은 Unity 프로젝트 내에서 Ray-tracing 모듈이 담겨있는 DLL을 링크하여 실행한다. 4. **Implementation: CUDA** 5. 결과의 이미지의 픽셀별로 GPU 의 코어를 이용해 병렬적으로 실행하는 naïve ray-tracing 은 구현이 되어있다. 6. CUDA 는 두 가지의 API 를 지원하는데, 하나는 많은 기능들이 구현되어 코딩을 쉽게 할 수 있는 Runtime API 와 노력이 더 필요하지만 GPU 자원들을 더 세밀하게 조정을 할 수 있는 Driver API 가 있다. 우선 Runtime API 를 사용하여 알고리즘의 구현과 optimization 을 하고, 설계 시 Driver API 에 대한 구현 변경을 염두 해 두어 필요 시 Driver API 로 바꿀 예정 이다. 7. Runtime API 에서는 쓰레드 별로 GPU에 작업을 넣을 수 있는 방법을 지원하다. 또한 Task 큐에 계속 Task 을 방식인 비동기 방법을 지원하기 때문에 쉽게 구현이 가능하다. Driver API의 경우에는 조금 복잡하지만 대부분의 API들이 thread-safety 하기 때문에 직접 구현하여 사용할 수 있다. 모두 여러 개의 GPU에 작업을 넣을 수 있다. 8. **Model: path-tracing** 9. ray-tracing 계산 모델은 카메라를 기준으로 광선을 쏘아서 그 광선에 맞는 표면의 재질을 사용하여 무작위로 빛이 얼마나 받는지를 계산하는 path-tracing 방법을 쓸 것이다. 그 중에서도 가장 많이 알려진 metropolis light transport 를 사용할 계획이다. 이 방법은 기존의 무작위 방식과는 조금 다르게 빛의 강도가 세면 셀수록 확률이 올라가는 방법이다. 그래서 단순한 랜덤 선택보다 노이즈가 덜한 편이다 10. 씬 내의 오브젝트가 많이 존재하면 존재할수록 ray 하나를 계산할 때, 선형 자료구조를 가질 경우 O(n) 의 시간이 걸린다. 그래서 3차원 공간의 데이터들을 효율적으로 탐색할 수 있는 방법이 필요한데 BVH 방법이 가장 일반적으로 알려져 있다. 이는 이진 트리 형태의 구조를 가지는데, 문제는 성능의 coherency 가 필요하기 때문에 이를 balancing 하는 방법도 고안 해야 한다. 하지만 아직은 명확한 정해지지 않았다. 11. MLT(metropolis light transport)의 계산 방식이 노이즈가 존재하지 않는다고는 할 수 없다. 그렇다면 이를 제거할 방법, 흔히 말하는 denosing 방법을 찾아야 한다. 동영상의 경우 주로 게임에서 사용되는 temporal 한 방법이 있지만, 이미지 하나를 처리하는 경우는 아직 명확한 방법은 정해지지 않았다. |
| 오픈 소스  및  사례  연구 | 1. [**pbrt-v3**](https://github.com/mmp/pbrt-v3): physically based rendering 3판의 예시 자료, CPU 기반으로 짜여져 있고, 학습 및 연구 목적으로 사용될 수 있다. 단순히 책의 추가 자료가 아니라 활발하게 이루어지는 오픈소스 프로젝트이고, 대부분의 기법들이 적용되어 있기 때문에 구현 시 대부분의 알고리즘을 참고할 수 있다 2. [**SOL-R**](https://github.com/favreau/Sol-R): GPGPU(CUDA, OpenCL) 기반의 ray tracer 로, 여기서 참고할 부분은 ray-tracing 을 어떻게 GPGPU 기반으로 짰는지에 대한 테크닉들이 있다. 3. **Octane Render** : GPGPU + Path-Tracing 를 구현한 상용 소프트웨어로, 해당 작품의 목표 중 하나이다. 소스가 공개가 아니기 때문에 구현 레벨에서는 정보를 얻을 수 없었고, 계산 모델에(MLT) 대한 정보만 얻을 수 있었다.. |

|  |
| --- |
| **동작 시나리오** |
| 1. 상용엔진/DCC 에서 렌더링 할 데이터(Objects, Mesh, Material, Shader)들을 플러그인에서 수집한다. 2. 수집한 데이터를 사용해 GPU의 각 코어에 픽셀별로 빛을 거꾸로 쫓아가는 ray-tracing 작업을 수행한다. 이를 수행하기 위해 CPU에서 여러 개의 픽셀을 계산하는 태스크 수행 명령을 비동기적으로 넣는다. 3. 모든 픽셀의 ray-tracing 을 수행하여 픽셀의 색을 얻은 후, 대부분의 경우 noise가 생길 수 있는데, 이를 해결하기 위해 denoising 프로세스를 수행한다. Denosing 을 성공적으로 수행한 후에 이를 출력한다. |
| **SW 구성도** |
|  |