

HW3

Dependencies

```
$ sudo apt update  
$ sudo apt upgrade  
$ sudo apt install build-essential  
$ sudo apt-get install freeglut3-dev  
$ sudo apt install g++  
$ sudo apt install libglm-dev
```

Env	Version
OS	Ubuntu 20.04
OpenGL	3.0 Mesa 20.0.8
freeglut3-dev	2.8.1-3
g++	7.5.0
glm	0.9.9~a2-2
gsl	2.7
CPU 요구 사항	AVX2

GSL 설치

1. GSL 다운로드

```
$ wget https://ftp.gnu.org/gnu/gsl/gsl-latest.tar.gz
```

2. 압축 해제 및 자동 설정

```
$ tar -xvzf gsl-latest.tar.gz  
$ cd ./gsl-latest  
$ ./configure
```

3. Build

```
$ make
```

4. 설치

```
$ sudo make install
```

5. 자동 설정 확인

```
$ sudo ldconfig -v | grep libgsl
```

실행 및 조작법

```
$ cd src  
$ make  
$ copy src/sceneView to [folder contains scene.txt]  
$ cd ./[folder contains scene.txt]  
$ [Edit scene setting if need]  
$ ./sceneView scene.txt
```

- 종료 : q
- 회전 : 화면 내 마우스 드래그
- zoom : Ctrl키를 누르면서 화면 내 마우스 좌우 드래그
- Translate : Shift 키를 누르면서 화면 내 마우스 상하 드래그
- Dolly : Shift키와 Ctrl키를 동시에 누르면서 화면 내 마우스 드래그
- 해상도 조정 : r키를 누른 후 콘솔에서 해상도 입력
- 레이 트레이싱 실행 : P키를 누르면 렌더링 시작.
 - 렌더링 시작 후 화면의 첫 번째 줄 연산 완료 후 opengl에서의 화면은 opengl.png로 출력
 - 첫 번째 줄 계산 이전에 화면에 이상이 생길 시 해당 화면이 opengl.png로 출력되기 때문에 그 전까지는 어떠한 조작을 하지 않는 것을 권장합니다.
 - 화면의 30줄 마다 임시 이미지 파일이 raytrace_tmp.png로 출력
 - 렌더링 완료 시 raytrace.png로 출력
- 렌더링 관련 scene.txt 명령어
 - thread [# of thread] : BVH 생성 시 사용될 스레드의 수. CPU의 총 스레드 수와 동일하게 설정하는 것을 권장
 - depth [# of depth] : ray tracing 시 ray의 depth.
 - 1로 설정할 시 diffuse만 렌더링하게 된다.
 - sample [# of sample] : ray tracing 시 각 pixel에 대해 시행할 multi sampling의 횟수
 - soft_shadow [float] : soft shadow를 위한 scattering factor.
 - 현실적인 이미지를 위해서 0.1~0.2 사이의 값을 권장
 - 0.0으로 설정 시, soft_shadow가 없는 것과 동일한 결과
 - sample_shadow [# of shadow ray] : 각 지점 당 shadow ray를 casting하는 횟수.
 - sample_spec [# of reflection ray] : fuzzed reflection을 위한 한 번의 reflection 계산 당 casting하는 reflection ray의 갯수
 - sky [red : 0.0~1.0] [green: 0.0~1.0] [blue: 0.0~1.0] : 배경색 설정
 - bKDtree [0/1] : **BVH**를 활성/비활성화시키기 위한 설정 (KD tree가 아닙니다.)
 - 0으로 설정 시 비활성화 되며 이후 관련 설정을 무시하고 BVH를 생성하지 않음
 - 1로 설정 시 활성화 됨
 - KDdepth [# of maximum depth] : pruning 이후 BVH tree가 가질 최대 depth.
 - pruning 이전 depth-4로 설정하는 것을 권장.
 - KD_MAX_TRI : 현재 사용되지 않는 옵션

Implemented features

Ray tracing sphere and polygons

1. Ray와의 충돌 감지

Sphere

공간 상에 Sphere를 나타내는 방정식과 ray의 이동거리 t에 parameterized된 직선의 방정식 간 교점을 계산하여 충돌 여부와 충돌 지점을 찾았다. 충돌 지점에서 normal vector는 Sphere의 중심에서 충돌 지점을 향하는 normalized 된 벡터로 정하였다. [Sphere::hit in Drawables.hpp at line 175]

Polygon(Triangles)

이전 과제에서 모든 polygon을 triangle로 분할하였기 때문에 triangle에 대한 충돌 처리만 구현하였다. triangle의 법선 벡터와 ray가 수직을 이루는지 검사한 뒤, 그렇지 않을 경우 triangle을 포함하는 평면과 ray의 교점을 구한 뒤 해당 교점의 세 정점에 대한 barycentric coordinate를 계산한다. barycentric coordinate를 통해 교점이 삼각형 내부에 있는지 검사한다. [Triangle::hit in Drawables.hpp at line 295]

2. Ray trace

카메라의 위치에서 각 Pixel을 향하는 ray를 casting하여 tracing을 시작한다.

[Renderer::thread_RenderPixel in Render.hpp at line 52] tracing하는 ray와 가장 먼저 충돌한 지점에서 광원들을 향하는 shadow ray를 casting하여 Phong illumination을 계산하고 필요한 경우 충돌 지점에서 refraction과 reflection ray를 각각 casting하여 trace한다. [Renderer::trace in Render.hpp at line 192]

Phong illumination

충돌 지점에서 각 광원들로부터 직접적으로 빛을 받을 수 있는지 검사하는 shadow ray를 cast하여 광원과 충돌 지점 사이에 빛을 막는 primitives가 있는지 확인한다. 없을 경우 해당 shadow ray로 입사 광의 벡터를 찾고 법선 벡터를 사용하여 Phong illumination을 계산한다. [Renderer::calcShawdow in Render.hpp at line 138]

Recursive reflection

앞서 설명한 것과 같이 reflection ray를 다시 trace하여 reflection color를 계산한다. trace의 depth가 max_depth(in parameters.hpp)를 넘을 때까지 recursive하게 반복한다. [Renderer::trace in Render.hpp at line 218]

이 때 reflection ray는 반사 광선이, 입사 광선의 충돌 지점의 탄젠트 평면에 대칭인 것과 동일하다는 성질을 이용하여 정하였다. [Renderer::trace in Render.hpp at line 219]

Recursive refraction

Reflection과 동일하게 refraction ray를 trace하여 refraction color를 계산하며 이는 depth가 max_depth(in parameters.hpp)를 넘을 때까지 recursive하게 반복한다.

Refraction ray는 snell's law를 이용해 구하였다. [Renderer::trace in Render.hpp at line 256] 다만 refraction의 경우 입사각과 materials의 refraction index에 의해 전반사가 발생하는 경우가 있다. 이 경우 reflection ray를 trace한다. [Renderer::trace in Render.hpp at line 246]

Export & import image files

이미지 출력, 입력의 경우 [stb library](#)를 사용하였다.

opengl의 렌더링 결과를 얻기 위해 glReadPixels를 이용하였다. [Image::exportImage in Image.hpp at line 48]

Texture mapping

Texture image를 stb library를 이용해 메모리에 올릴 수 있었다.

Bilinear interpolation을 적용하였다. [Texture::at in material.hpp at line 40]

1. Polygon(triangle)

Texture mapping을 위해 정점의 texture coordinate들을 polygon 상의 점의 barycentric coordinate를 가중치로 하여 interpolation을 하여 u,v 좌표를 구하였다.[Triangle::getDiffuseColor in Drawables.hpp at line 340]

2. Sphere

normalized된 법선 벡터에 대해 구면좌표계를 적용해 위도와 경도를 구하고 이를 각각 texture의 v, u 좌표에 적용한다. [Sphere::getDiffuseColor in Drawables.hpp at line 214]

Distributed ray tracing

1. Multisampling

각 픽셀의 좌상단 부분에 $[0,1] \times [0,1]$ 의 random한 offset을 더해준 지점으로의 ray trace한 결과를 sample_factor in parameters.hpp만큼 반복한 뒤 이의 평균을 해당 픽셀의 색상 값으로 지정하였다. [Renderer::thread_RenderPixel in Render.hpp at line 49]

2. Soft shadow

shadow ray를 casting할 시 기존 shawdow ray의 방향에 soft_shadow in parameters.hpp 크기의 임의의 벡터를 더한 후 이를 normalize한 결과를 shawdow ray로 하였으며 [Renderer::calcShawdow in Render.hpp at line 154] 이를 sample_shadow in parameters.hpp만큼 반복하여 그 평균을 해당 지점의 색상으로 지정하였다. [Renderer::trace in Render.hpp at line 206]

3. fuzzied reflection

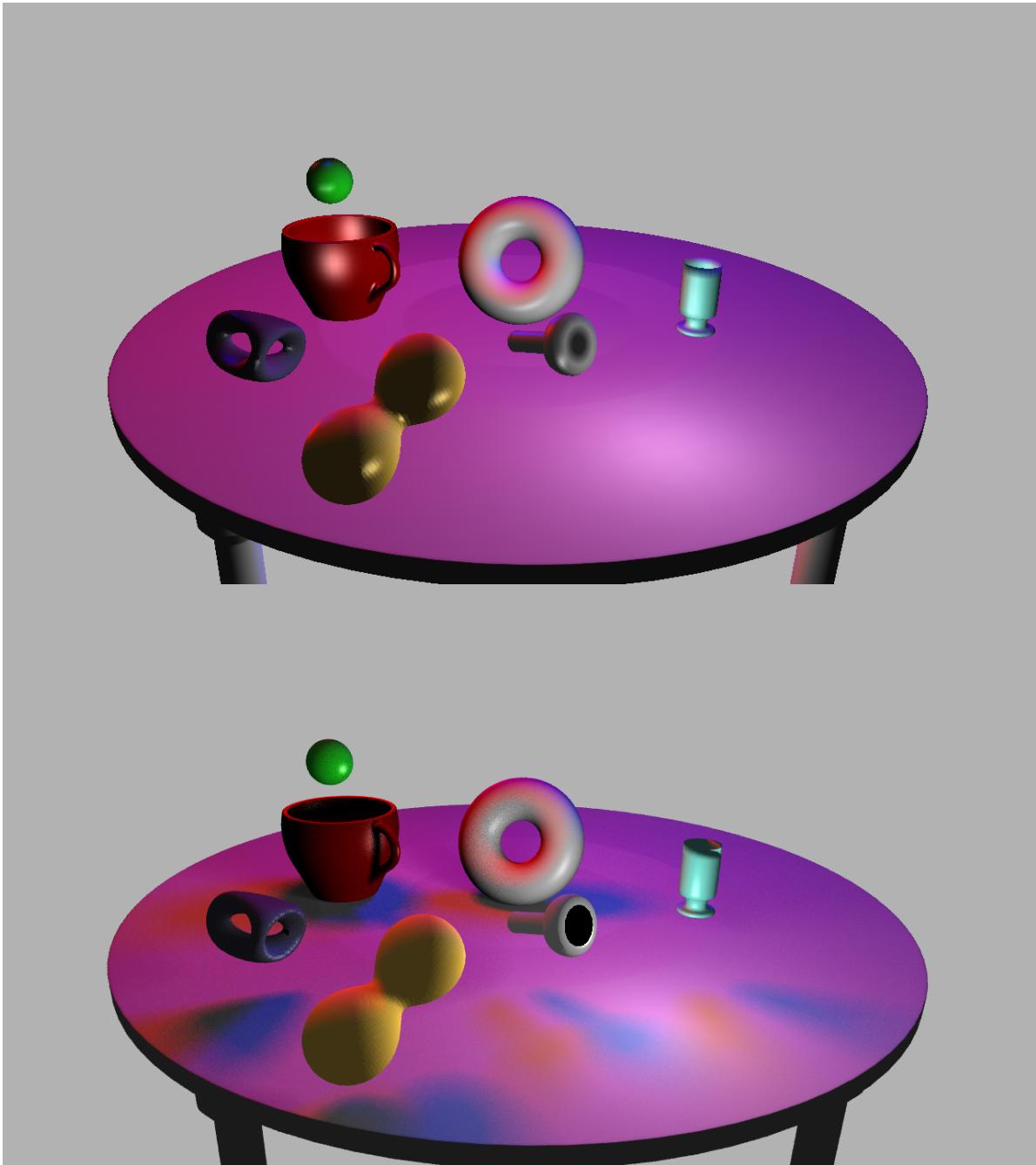
reflection ray로 기존 reflection ray의 방향에 해당 materials의 fuzzy와 같은 크기를 가지는 임의의 벡터를 더한 결과를 사용하였으며 이를 sample_spec in parameters.hpp만큼 반복하여 그 평균을 해당 지점의 reflection color로 지정하였다. [Renderer::trace in Render.hpp at line 217]

Spatial partitioning

Spatial partitioning을 위해 bottom-up 방식으로 Bounding volume hierachy를 생성하였다. 이 때 cost를 각 volume의 크기로 지정하였다. [BVHTree::buildTree in BVH.hpp] 각각의 AABB 알고리즘을 통해 [AABB.hpp] 각 node와 ray의 충돌 검사를 하여 BVH tree를 DFS하여 충돌 검사할 primitives를 추려냈다. [BVHTree::ray_traversal in BVH.hpp at line 208]

Result

Rerendering scene from HW4



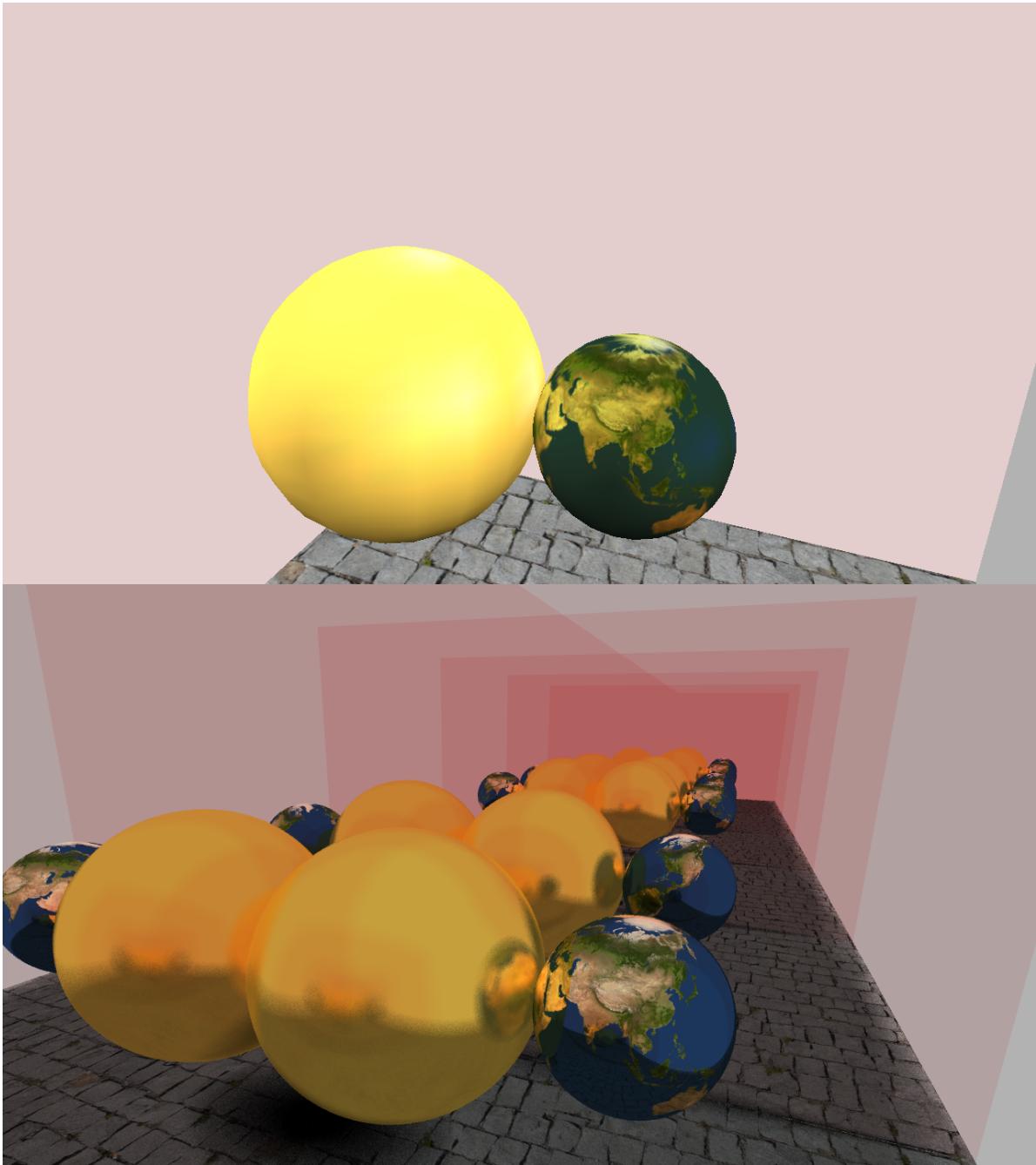
[Figure 1] 이전 과제의 결과를 raytracing으로 렌더링한 결과이다. Scene 내에 총 110515개의 object가 존재하며 HD 이미지로 렌더링하는데 3164초가 소요되었다. Scene folder : glHW4/scene

OpenGL로 렌더링한 것과 달리 ray tracing으로 렌더링한 이미지에선 그림자가 생기 빛이 닿는 부분과 닿지 않는 부분을 구분할 수 있다. 또한 여러 생상의 광원을 사용했기 때문에 색그림자 현상을 관찰할 수 있다.

0.15 정도의 크기로 4번 샘플링한 soft shadow를 적용하여 자연스러운 그림자를 관찰할 수 있다.

Ray tracing으로 렌더링한 결과에서 나사 모양의 모델과 작은 잔 모양의 모델에서 제대로 그려지지 않은 부분들을 관찰할 수 있는데 이들은 swept surface로 구현한 모델들로 매우 작은 polygon들이 있는 지점이 존재한다. 해당 폴리곤의 크기가 매우 작아 그 넓이가 0에 매우 가까운 지점이 존재하고 이 때문에 barycentric coordinate가 -nan이 되는 경우가 발생한다. 해당 오류들은 이러한 이유로 발생한다. 따라서 모델의 크기를 확대하거나 swept surface의 보간의 정도를 줄이면 해결할 수 있을 것이다.

Rendering reflecting scene



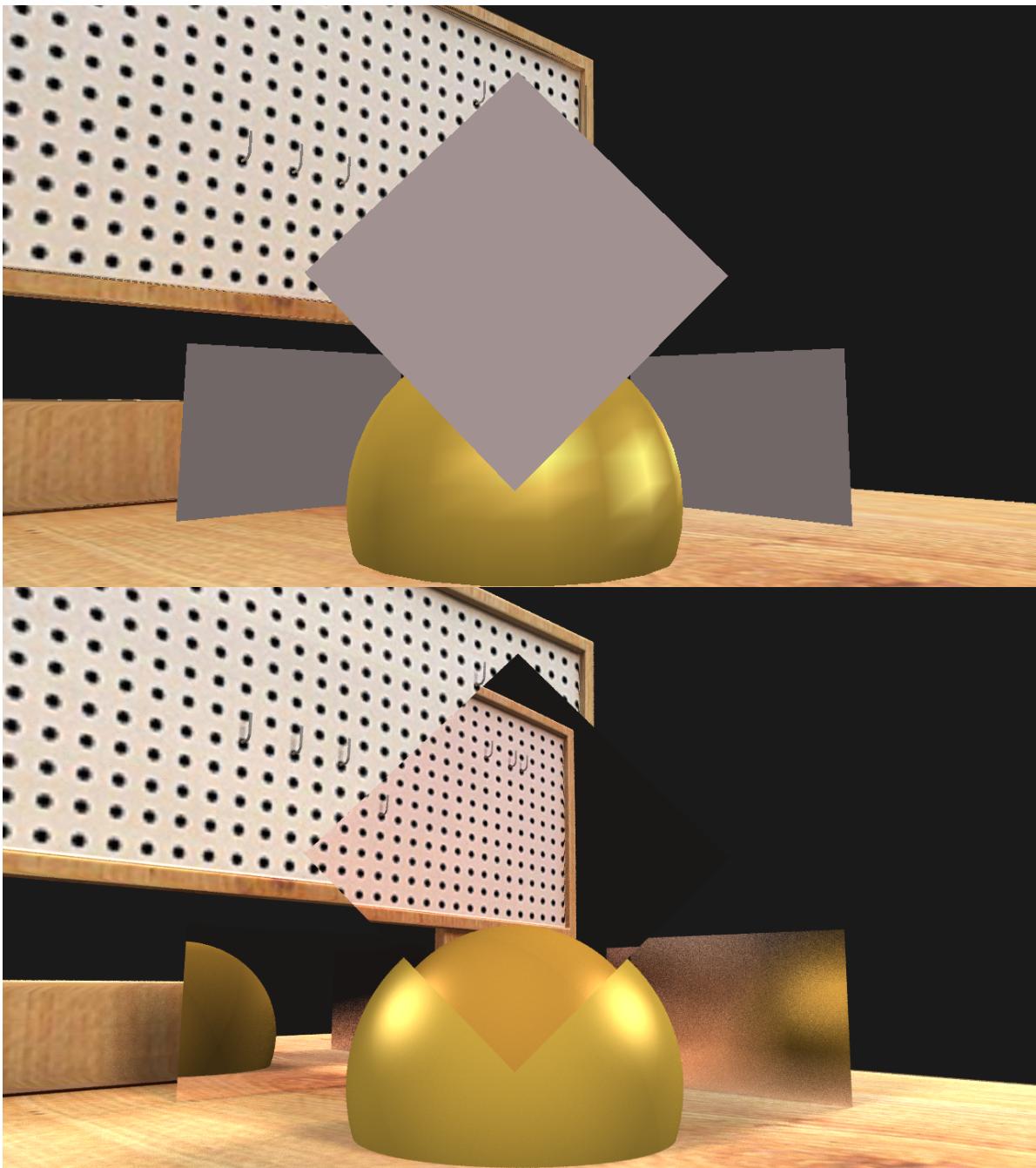
[Figure 2] 약한 붉은색을 띠는 거울이 삼면을 감싸고 반사되는 두 개의 sphere가 있는 scene이다. 하나의 sphere는 fuzzied reflection이 적용되었다. 최대 depth는 6이다. HD 이미지로 렌더링하는데 3951초의 시간이 소요되었다. Scene folder : src/scene

ray tracing으로 렌더링한 이미지에서는 반사가 되는 것을 확인할 수 있다. 서로 마주보는 거울들에 의해 여러 번 반사되는 것을 확인할 수 있다.

또한 Fuzzied reflection하는 sphere에선 거친 표면에서 반사되는 것처럼 선명하지 못한 반사를 관찰할 수 있다.

텍스쳐가 attenuation으로 작용하는 것도 확인할 수 있다.

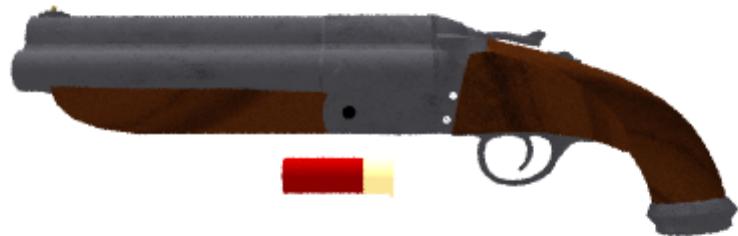
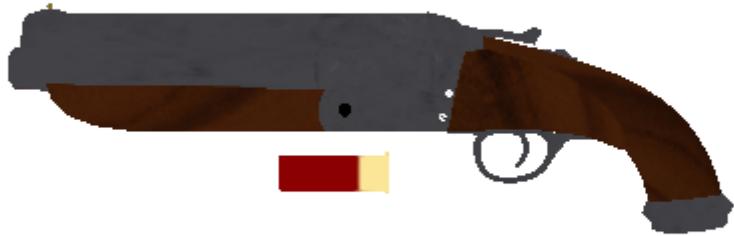
Rendering refraction scene



[Figure 3] workbench 모델, 금색의 sphere, 완전히 반사하는 거울, fuzzied reflection하는 거울과 refraction하는 약하게 붉은 색을 띠는 유리판이 있는 scene이다. HD 이미지로 출력하는데 10162초가 소요되었다. Scene folder : src/scene/workbench

중앙의 유리판에서 refraction이 발생하여 원래 크기보다 작은 모습으로 그려지는 것을 확인할 수 있다. 또한 workbench 위에 여러 부분에서 그림자가 그려지는 것을 관찰할 수 있다.

Improvement using BVH

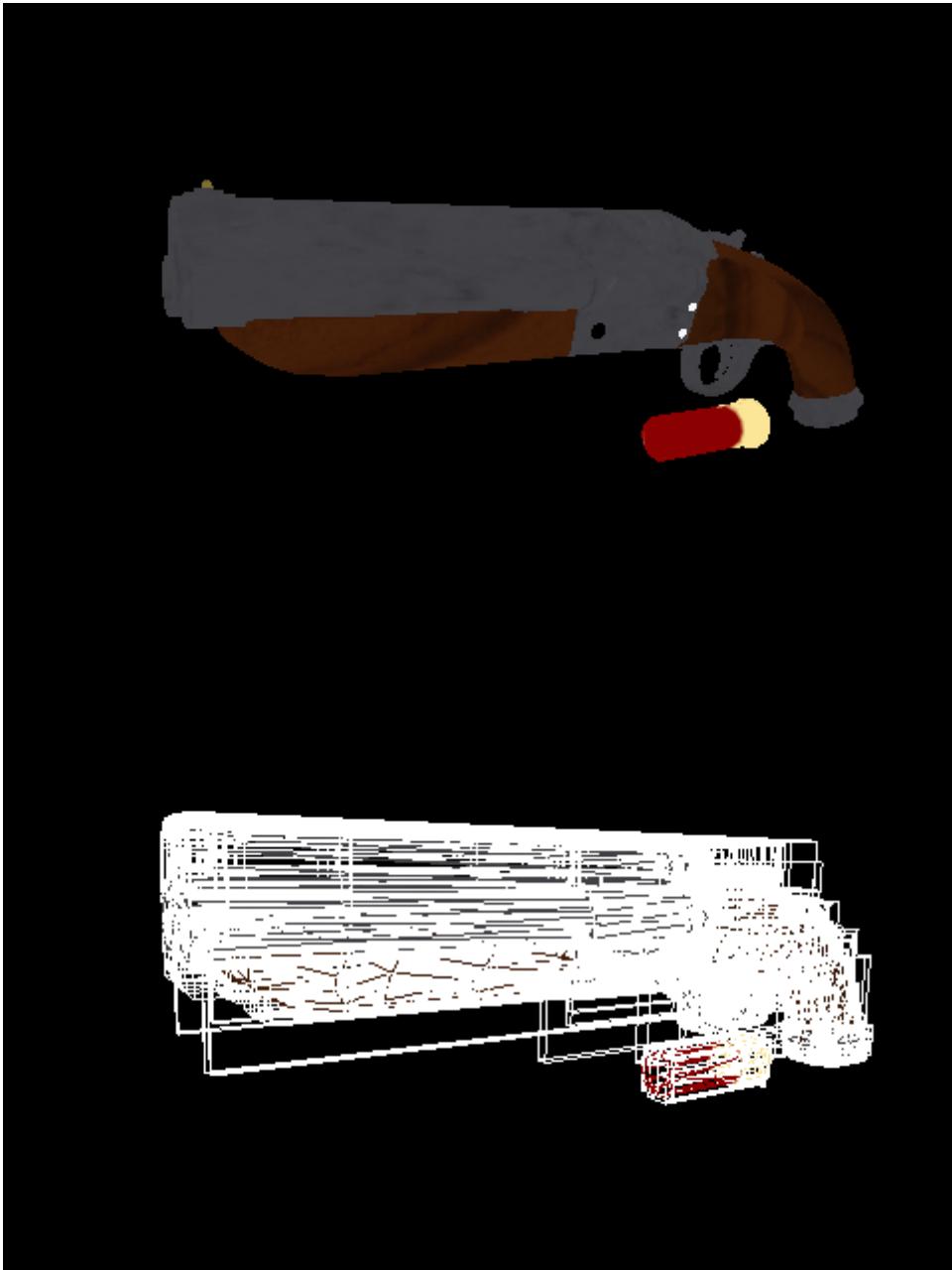


[Figure 4] 총기 모델이 있는 Scene을 BVH를 통한 성능향상을 알아보기 위해 렌더링하였다. Scene folder : src/scene/shotgun

480 x 320 사이즈의 이미지를 BVH의 사용 여부를 제외하고 동일한 조건에서 렌더링하였다.

8배의 멀티샘플링, 4배의 shadow sampling을 적용하였으며, BVH의 최대 depth를 11로 지정하였다.

BVH를 사용하지 않은 경우 326초가 소요되었고 BVH를 사용한 경우 22초가 소요되었다. 대략 14.8배의 성능 향상을 얻을 수 있었다.



[Figure 5] 원본 모델과 BVH tree의 leaf노드들의 bound volume을 표시한 이미지이다.

해당 volume들을 통과하지 않는 ray들의 경우 어떠한 충돌 연산 없이 trace를 마칠 수 있다. 또한 통과하더라도 모든 primitives와 충돌 검사를 할 필요 없이 통과한 volume에 속하는 primitives에 대해서만 검사하기 때문에 계산량이 매우 크게 감소한다.