IMPLEMENTATION OF A MONTE-CARLO RAY TRACER

CRISPHER GU

CONTENTS

1 Overview 2
2 Implementation Details 2
2.1 Source Files . . . 2
2.2 Principles . . . 3
2.3 Phong Model . . . 4
2.4 Anisotropic BRDF . . . 4
2.5 Direct Lighting . . . 4
2.6 Russian Roulette . . . 5
2.7 Texture Mapping & Bump Mapping . . . 5
2.8 Depth of Field . . . 5
2.9 KD Tree Accelerator . . . 6
2.10 Physics Engine . . . 6
2.11 Scattering . . . 6
2.12 Reconstruction . . . 7
3 Mesh Simplification 7
3.1 Edge Contraction Algorithm . . . 7
3.2 Quadric Error Metrics . . . 8
4 Demonstrative Pictures 8

LIST OF FIGURES

Figure 1 A glass dragon on a sheet of paper . . . 9
Figure 2 Glossy dragon under pink light . . . 10
Figure 3 Cornell box with anisotropic floor . . . 11
Figure 4 Cornell box with two lambertian spheres inside . . . 12
Figure 5 *Meisje met de parel* with jigsaw texture . . . 13
Figure 6 Cornell box filled with scattering gas . . . 14
Figure 7 Dragon on anisotropic surface . . . 15
Figure 8 Glass model of our school gate . . . 16
Figure 9 Frames of animation rendered by the physics engine 17
Figure 10 Simplified block . . . 18
Figure 11 Simplified dragon . . . 18

1 OVERVIEW

The Monte Carlo Ray Tracer is implemented from scratch and with full functionality. Here is an incomplete list of implemented features:

- Ability to read Wavefront OBJ models and material specifications
- Ability to handle two type of geometry primitives: triangle (mesh) and sphere
- Rendering of Phong BRDF
- Texture mapping and bump mapping
- Depth of field
- Importance sampling and direct lighting
- KD tree accelerator
- A tiny physics engine framework with partial implementation
- Two types of sampler
- Rendering of scattering media and anisotropic BRDF
- Mesh simplification
- etc.

The entire project is written in C++. Some libraries are used to avoid triviality: 1) Open CV is used to read, write and show images; 2) Eigen Library is used to perform vector algebra such as inner product or cross product; 3) boost library is used in OBJ file parsing. The project was written, debugged and tested in Visual Studio Community 2013, under windows platform. Source code is available on my GitHub page.

2 IMPLEMENTATION DETAILS

2.1 Source Files

Here is a table describing the source files and their functions.

Source file (.h, .cpp)	Description
IntersectionTester	The virtual class IntersectionTester is the interface for a ray tracer to perform intersection test. The class SimpleIntersectionTester extends the base tester and perform intersection test by going through all triangles.
Kdtree	Contains classes BBox, which stands for Axis aligned bounding box, Kdtree, which extends the base intersection tester and perform tests by testing necessary nodes in the tree.

Object	Loads object from OBJ files and compute geometry information if necessary.
Raytracer	Combines the functions together to trace a single ray. It calls takes a ray as argument, then calls the intersection tester (simple or KD tree) to get intersection point, local material information and normal etc., then calls the sampler to generate a ray sample accordingly and trace the sample recursively.
Renderer	Reads rendering configurations from scripts, set the camera, generates ray samples for pixels and reconstructs pixel intensity, takes care of multi-threading affairs
Sampler	Contains 2D and 3D random sample generator. The 2D sampler generates uniform samples in the unit square, which is used in pixel rendering and light-source sampling. The 3D sampler generates cosine distributed ray samples. Pre-sample utility is implemented to improve efficiency.
Scene	Combines multiple objects to form a scene. Defines Camera class.
Screen	Utilities to print the image to screen. Contains definition of Color class.
Physics	A tiny physic engine framework, including simple functions such as collision testing and gravity.
MeshSimplify.cpp	Implementation of mesh simplifier, using quadric error metrics method.
stdafx.h	External includes, macro definitions.
volatile.cpp	Feel free to ignore.

2.2 Principles

The renderer is completely physically based, i.e., all calculations follows real world physical laws (or at least have a convincing physical interpretation). The general principle for designing algorithms is to evaluate the integral

$$L(\omega_o) = \int_\Omega f(\omega_i, \omega_o) L(\omega_i) \cos\theta \, d\omega,$$

a.k.a. the rendering equation using Monte Carlo methods.

2.3 Phong Model

Albeit the usefulness of ambient light in real-time rendering, it contradicts physical laws and therefore is ignored in this project. Other two types of luminance are treated differently.

SPECULAR REFLECTION The sample is generated deterministically. In the extended Phong model which allows transmittance, specular refraction is treated similarly.

LAMBERTIAN The radiance of a lambertian material at a given point is calculated via Monte-Carlo integration of the rendering equation, where $f(\omega_i, \omega_o) = \text{const}$. Reflecting ray samples are generated according to cosine distribution (importance sampling). This technique produces physically correct reflections and can capture the interreflections as illustrated in the sample picture of Cornell box. However due to the large gap between the radiance of a light source and non-emissive materials, importance sampling alone can lead to slow convergence. This is where the technique of direct lighting naturally arises.

GLOSSY Glossy material is in the midway from lambertian to specular. Energy is distributed around the specular reflecting direction ω_r and the density $f(\omega_o)$is proportional to $\langle \omega_r, \omega_o \rangle^k$, where k is the glossy index.

2.4 Anisotropic BRDF

Although I came up with this BRDF on my own I am quite sure it has been proposed before. Given incident ray ω_i, the reflected energy is distributed on a 2D manifold (instead of a volume). This kind of BRDF takes a reference direction $\vec{r}_{ref}$, s.t.$\langle \vec{r}_{ref}, \vec{n} \rangle = 0$. The probability density function is given by

$$f(\omega_o) = \frac{Z}{\|\omega_o - \omega_i\|} \langle \omega_o - \omega_i, \vec{n} \rangle^k, \forall \omega_o - \omega_i \in \text{Span}(\vec{n}, \vec{r}_{ref})$$

where Z is some normalizing factor, and $k > 0$ is some parameter (similar to the glossy index in Phong model). As $k \to \infty$, the distribution becomes specular reflection.

Intuitively, the above equation just describes a material that behaves like lambertian along the reference direction but like specular perpendicular to the reference direction.

Figure 3, 8 best demonstrate this feature.

2.5 Direct Lighting

Direct lighting is no more than a split of integration. I.e. the integral is split into two parts

$$\begin{aligned} L(\omega_o) = &\int_{\Omega_{\text{lightsource}}} f(\omega_i, \omega_o) L(\omega_i) \cos\theta \, d\omega \\ &+ \int_{\Omega_{\text{non-emissive}}} f(\omega_i, \omega_o) L(\omega_i) \cos\theta \, d\omega. \end{aligned}$$

By applying Monte-Carlo method to two parts respectively, the variance will be significantly reduced. For the first term, it can be evaluated by sampling the lightsource:

$$\begin{aligned}&\int_{\Omega_{LS}} f(\omega_i, \omega_o) L(\omega_i) \cos\theta d\omega \\ =&\int_{\Omega_{LS}} f(\omega_i, \omega_o) I_{LS} \delta \cos\theta \frac{\cos\gamma dS}{r^2},\end{aligned}$$

where I_{LS} is the radiance of the light source and δ is the visibility function, taking value 1 if the sample point on the light source is visible to the point and 0 otherwise.

A tricky question to this evaluation is that the ray sampled to estimate the second term may point to the light source again, resulting in the light sources being doubly counted. This can be avoided via various tricks. A simple method is to re-sample if the sample points to the lightsource, however this is computationally inefficient. In my implementation, 0 is returned if the sample points to a light source. This underestimates the value of second term, so we need to multiply the second term by a compensating factor $1/\Pr[non\_emissive]$. The probability is evaluated at the same time while the first term is evaluated:

$$\Pr[non\_emissive] = \int_{LS} \frac{\delta \cos\theta}{\pi} \frac{\cos\gamma dS}{r^2}.$$

2.6 Russian Roulette

When the recursive depth has meet the pre-set value, the recursive call should return an unbiased estimate of local luminance. If we simply return black the image is biased although it may be visually acceptable. So we need the Russian Roulette trick to give unbiased estimates: return 0 with probability p, return true-value$/(1-p)$ with probability $(1-p)$.

2.7 Texture Mapping & Bump Mapping

For some point P with texture value T, its induced material information is computed as $P' = (P_R T_R, P_G T_G, P_B T_B)$, P denotes an arbitrary property (Kd, Ks or Ts). Bump map uses the intensity gradient to compute a normal shift, which is added to the original normal vector. Figure 1 and 7 demonstrates the feature.

2.8 Depth of Field

The property of DOF is decided by two parameters, aperture α and the distance to focal plane d. The ray sample from eye has two intersections with the screen plane and the focal plane. When DOF is enabled, we shift the intersection on the screen plane by some random noise up to α but keep the intersection on the focal plane fixed. In this way we can render images with physically correct DOF effect.

Figure 1 is rendered with DOF effect.

2.9 KD Tree Accelerator

DATA STRUCTURE Each internal node contains 2 children and information of its bounding box. Each node contains the indices of faces located in the bounding box. Each node will split if conditions (depth of the tree, number of primitives inside) are not met. The tree is constructed recursively, starting with a bounding box that contains the entire scene. Bounding boxes are axis-aligned and split along the direction which the box has the maximal length.

INTERSECTION TESTS Tests are performed on each node, starting from the root. Here's the pseudo-code of the algorithm.

Algorithm 1: IntersectionTest

```
Input  : ray
Output : boolean indicating if the ray intersects with any triangle inside
         the region of the node
begin
    if isLeaf then
        Do intersection tests with all triangles inside;
        if intersects AND the intersection is within the associated bounding
        box then
            return true;
        else
            return false;
    else
        if NearChild→ IntersectionTest(ray) then
            return true;
        FarChild→ IntersectionTest(ray);
        return intersects AND BoundingBox.contain(intersection point);
```

2.10 Physics Engine

The structure of physics engine framework is similar to that of the project of CMU course. Geometrical and physical properties are separated. Geometrical informations are updated only after physical informations are updated.

Implemented features of the engine includes collision testing, gravity. A short sample video is available here. Also see figures 9.

2.11 Scattering

The program is able to render uniformly distributed scattering media. Given the distribution uniform, the distance between successive scatters follows negative exponential distribution, i.e.

$$f(d) = e^{-kd},$$

where $1/k$ is the expected length, or Mean-free-path in physical terms.

The direction after scatter are computed as if the "dusts" in the air are tiny lambertian spheres, basing on this model the distribution of scattered direction can be computed as

$$f(\omega_o) \propto \cos\frac{\theta}{2},$$

where θ is the angle between incident ray and scattered ray.

One more subtlety in rendering scattering involves direct lighting. In addition to quadratic decay, the radiance should be multiplied by a factor of exponential decay. Moreover, ray samples should be reject as long as it points to the light source, even if it is scattered before reaching the light source.

See figure 6.

2.12 Reconstruction

Usually hundreds of ray samples are computed to determine the color of a single pixel. The procedure to compute pixel color from hundreds of ray samples is called *reconstruction*. In this case reconstruction is done using Gaussian filter.

3 MESH SIMPLIFICATION

3.1 Edge Contraction Algorithm

Algorithm used for mesh simplification is rather straightforward. Pseudo code is given as follows.

Algorithm 2: Mesh simplify

```
Input  : Mesh, expected number of triangles
Output : A simplified mesh
begin
    initialization: compute the error of each edge and add them to data
    structure;
    while expectation is NOT met do
        find the edge e of minimum cost in the set;
        collapse the edge to a point;
        update the mesh and cost of related edge
```

Following the guideline we choose binary tree as the data structure to store the set of edges. For each update, it costs $O(\log m)$ to find an edge of minimum cost, to update length of involved edges and to erase unnecessary edges. Therefore the complexity of the algorithm is $O(k \log m)$, where k is the number of updating rounds.

A simple criteria of choosing collapsed edge is by its length, i.e. remove the shortest during each iteration, and collapse the edge to its bisection point. However it is far from perfect, a better treatment it to measure cost by quadric error.

3.2 Quadric Error Metrics

Quadric error measures if a vertex is important and give instructions on how to choose a new location for collapsed edge. In this setting each face is associated with a quadric matrix P, which can be used to compute the squared distance of an arbitrary point to the plane. Upon contraction the new location is chosen to minimize the overall error introduced by nearby faces. Typical implementation associate each vertex v with a matrix $Q_v = \sum P_f, \forall f$ incident to v. Edge $e = (u, v)$ has cost

$$err(e) = \min_r (r^T (Q_u + Q_v) r),$$

and collapse to r^* that minimize the error. A slight different implementation which can be more accurate in some cases takes the union of faces, i.e.

$$err(e) = \min_r r^T (\sum_{f \in F(u) \cup F(v)} P_f) r,$$

where $F(\cdot)$ denotes faces incident to that point. Both were implemented in my project and unlike claimed by some notes, neither significantly outperforms the other.

Usually the above algorithm works well but when there are many faces lying on the same plane, the target location of a collapsed edge is thus undefined, incurring arbitrarily large error. To resolve this problem we can perturb each vertex by a small random vector. The distortion induced is negligible but it gives much better target for collapsing edge.

See figures 10 and 11 for examples.

4 DEMONSTRATIVE PICTURES

Pictures in the following pages are rendered with my laptop, which carries a 2-core Intel i5 3230 CPU. Most of them take hours to render.

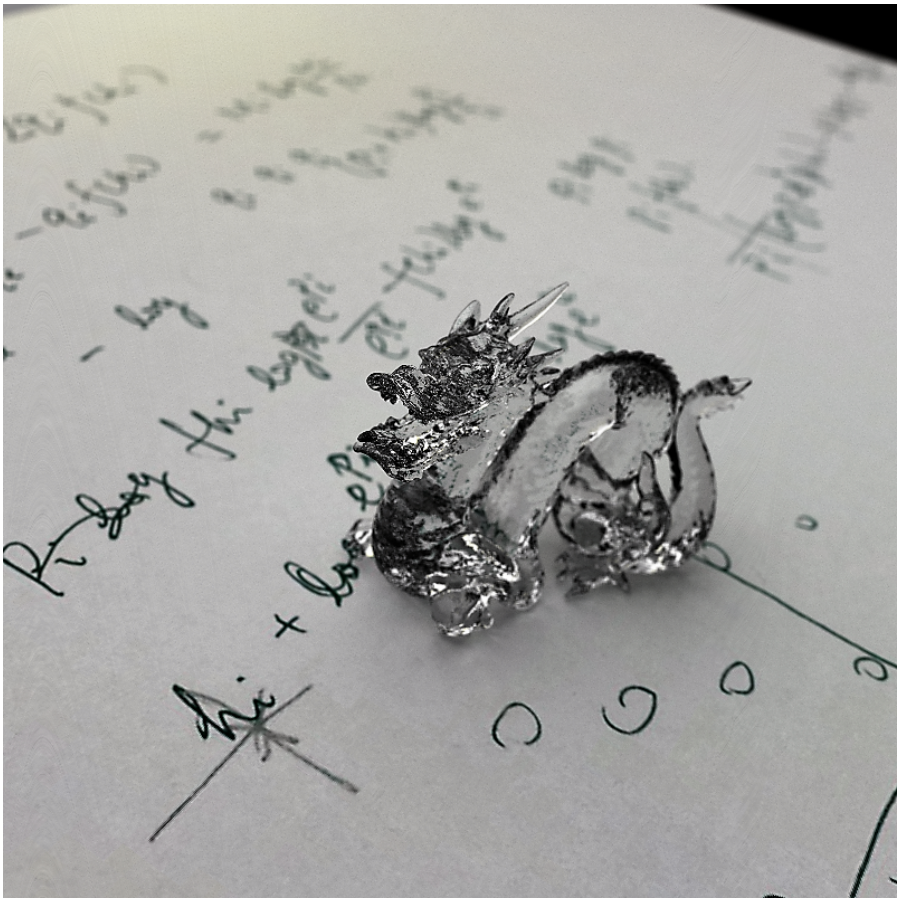


Figure 1: A glass dragon on a sheet of paper. The texture comes from a photo of my own scratch paper. Everything outside the focal plane blurs due to *depth of field*. The entire scene is illuminated by "ambient light" at infinity.

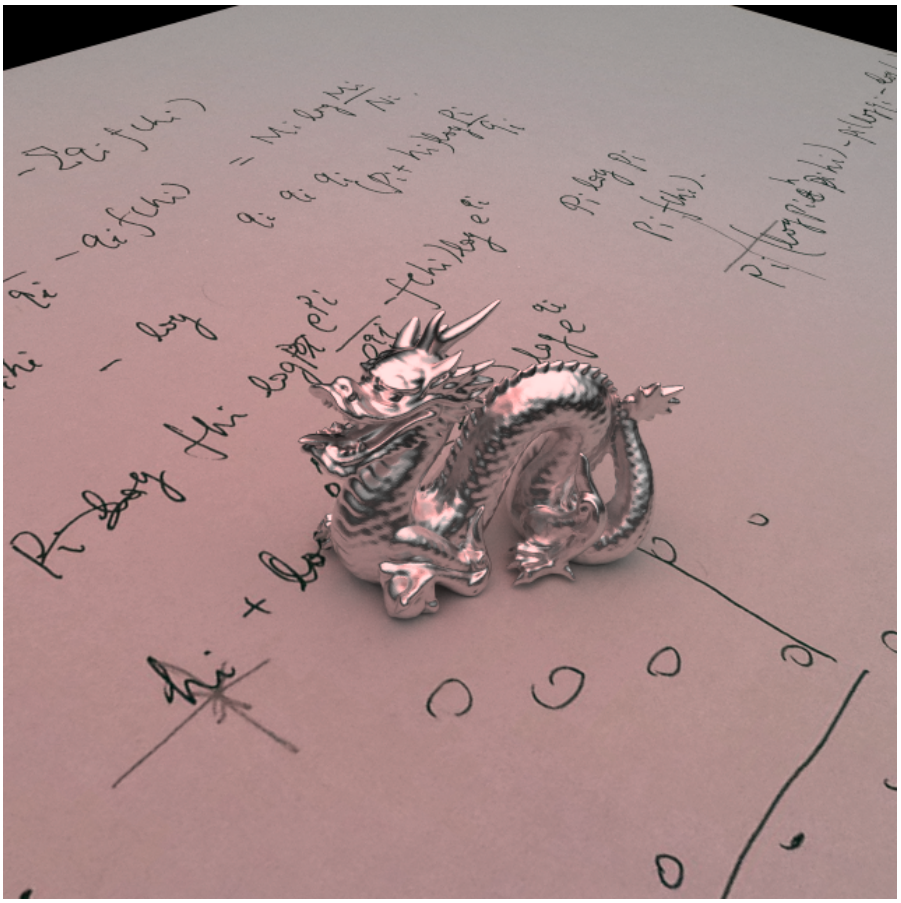


Figure 2: Glossy dragon under pink light. The glossy index is 20.

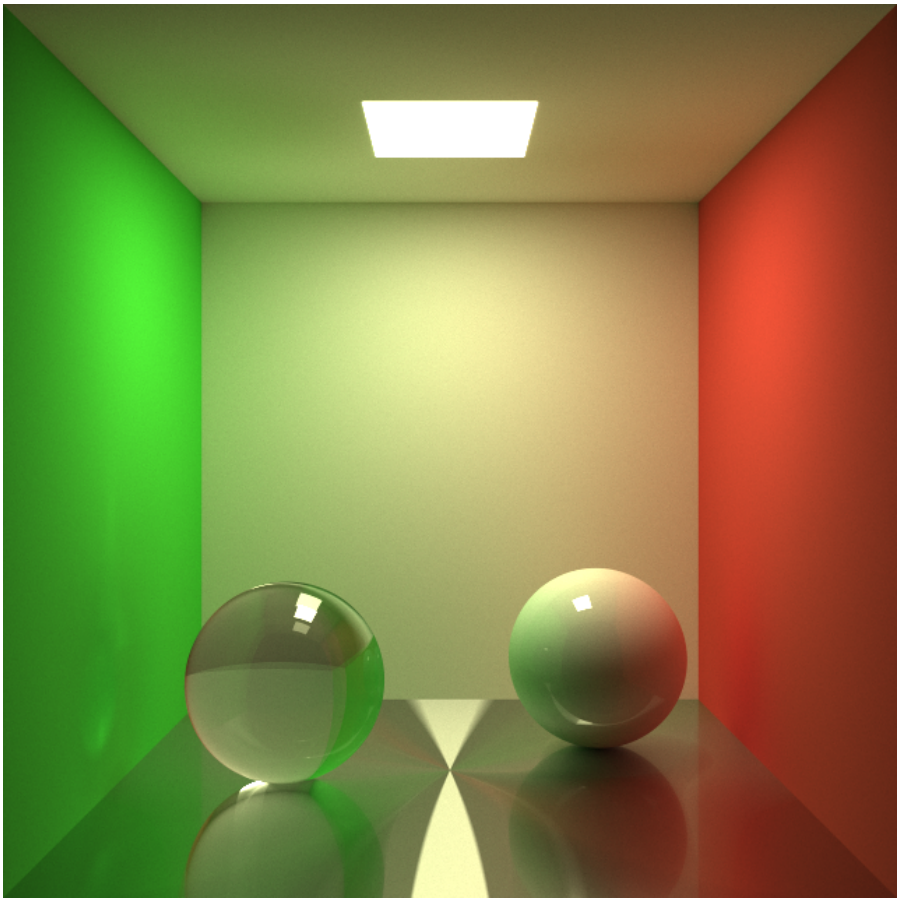


Figure 3: Cornell box with two spheres inside. The floor has anisotropic BRDF described in 2.4 and reference direction is taken tangentially. This kind of BRDF mimics some special metallic surfaces.

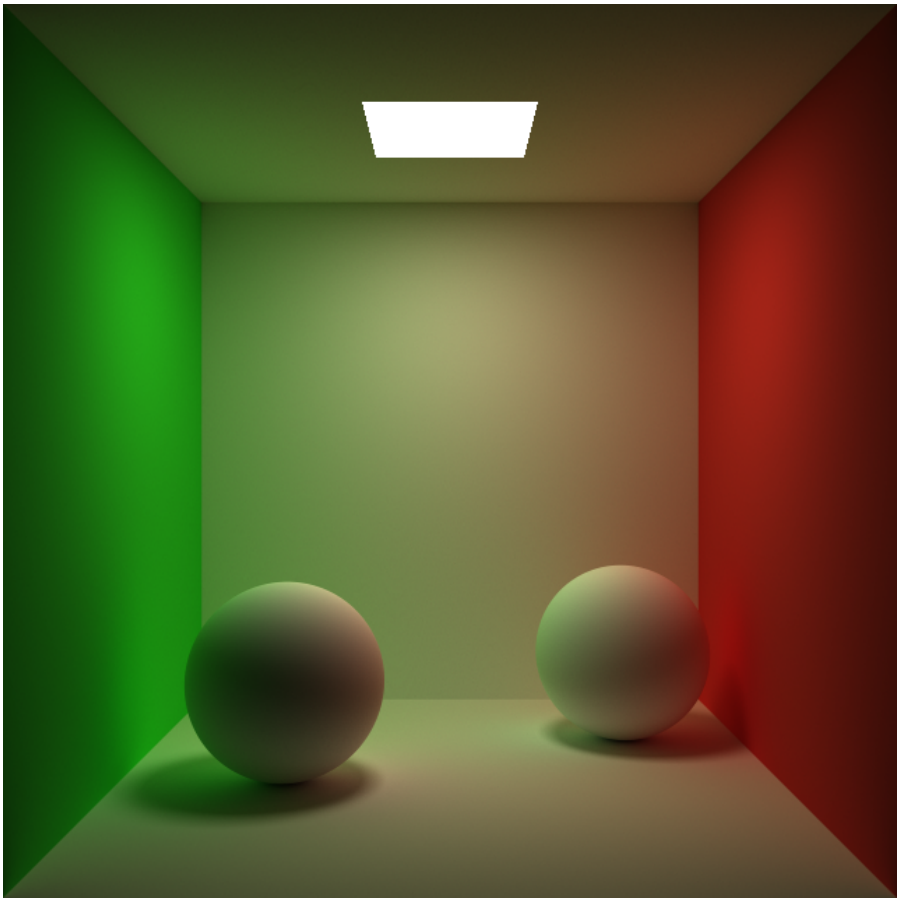
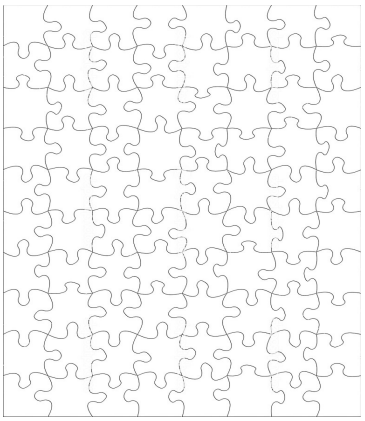


Figure 4: Cornell box rendered with 10,000 samples per pixel. Direct lighting technique is used to reduce noise.

(a) *Meisje met de parel* with jigsaw texture

(b) Texture: *Meisje met de parel*



(c) Bump texture

Figure 5: This demonstrates bump mapping technique. Texture and bump mapping are combined in this setting.

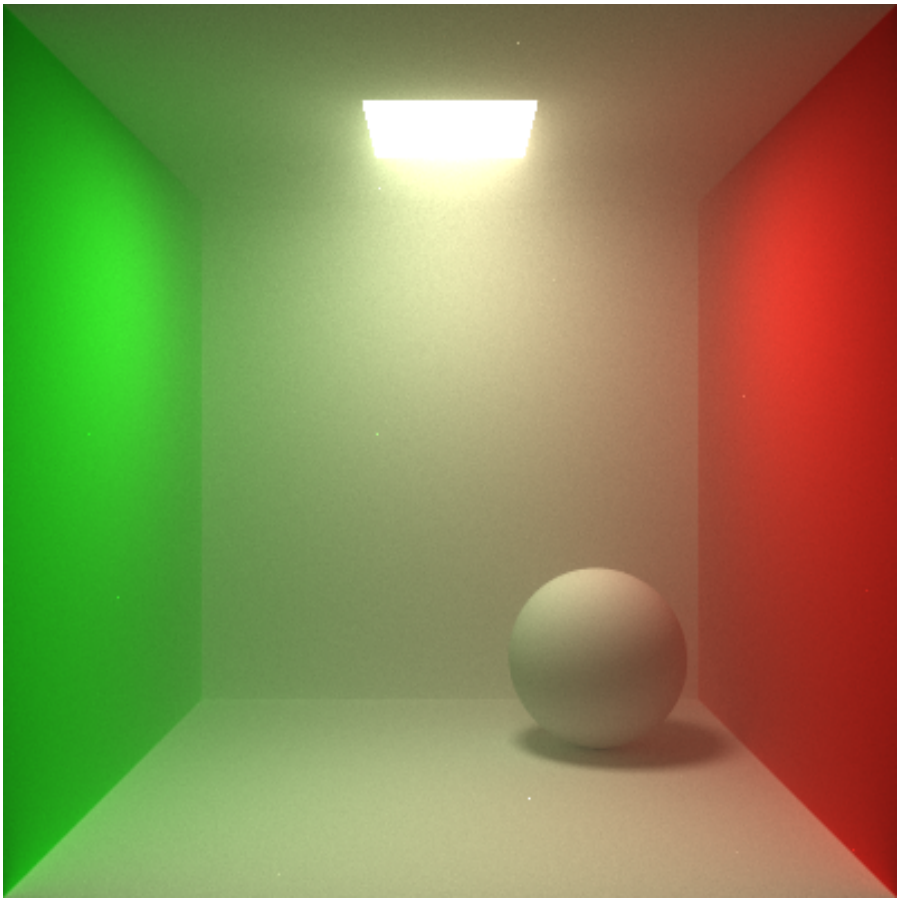


Figure 6: This picture renders a Cornell box with scattering gas filling space. The mean-free-path is equal to the size of the box.

Figure 7: Same setting as in 1, but different textures

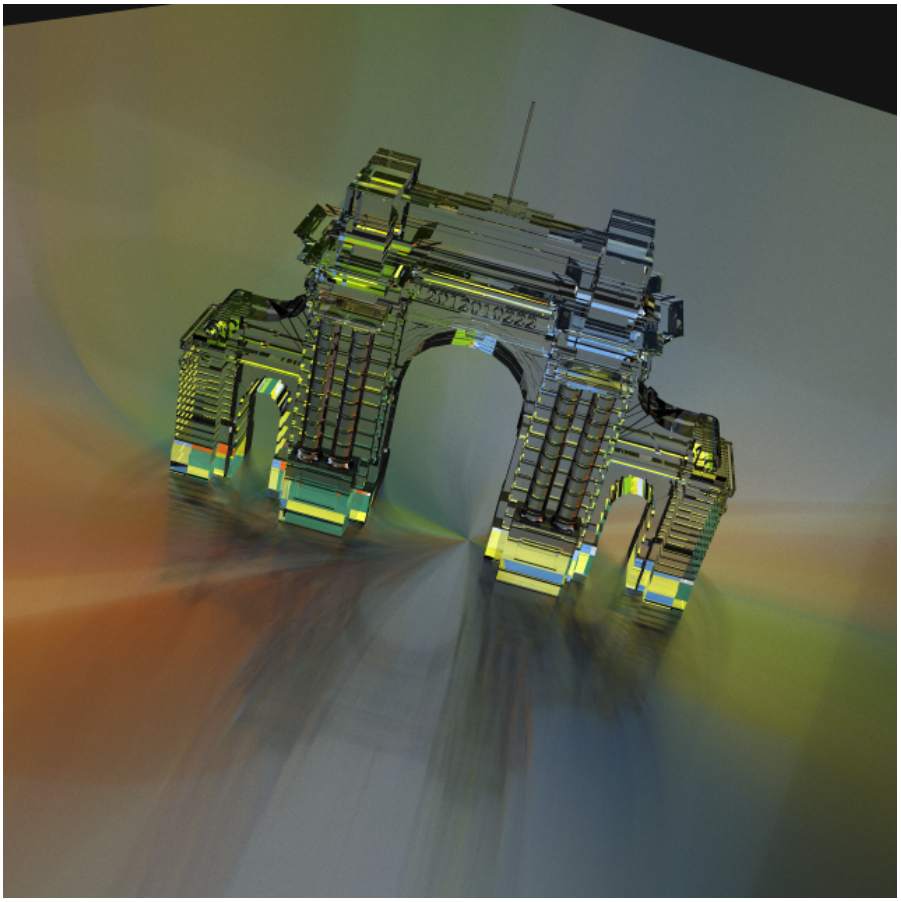
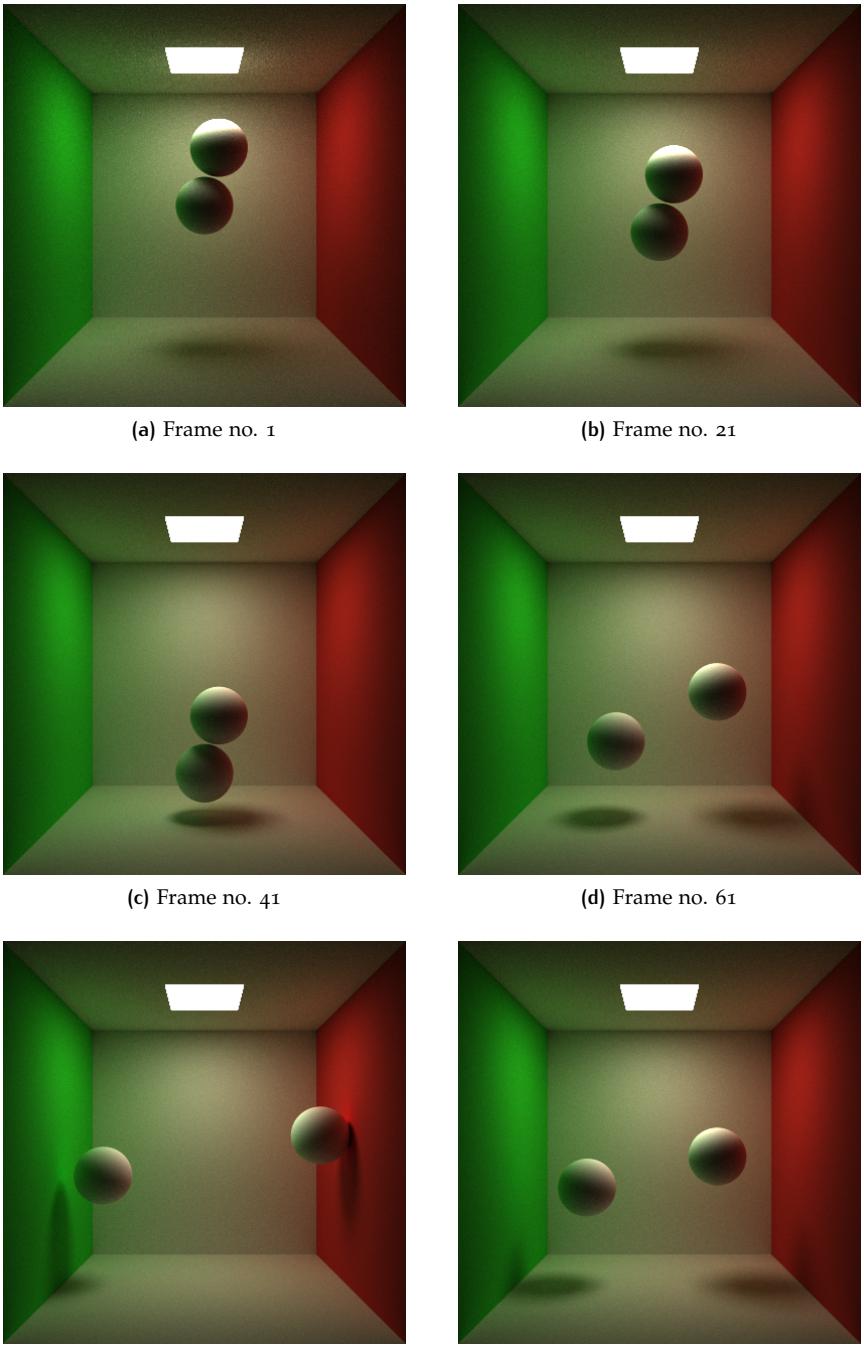


Figure 8: This renders transmittance and anisotropic BRDF surfaces. A big light source with texture is put above the scene. Total internal reflection is captured also, giving a realistic representation of glass material.



(a) Frame no. 1

(b) Frame no. 21

(c) Frame no. 41

(d) Frame no. 61

(e) Frame no. 81

(f) Frame no. 101

Figure 9: Frames of animation rendered by the physics engine

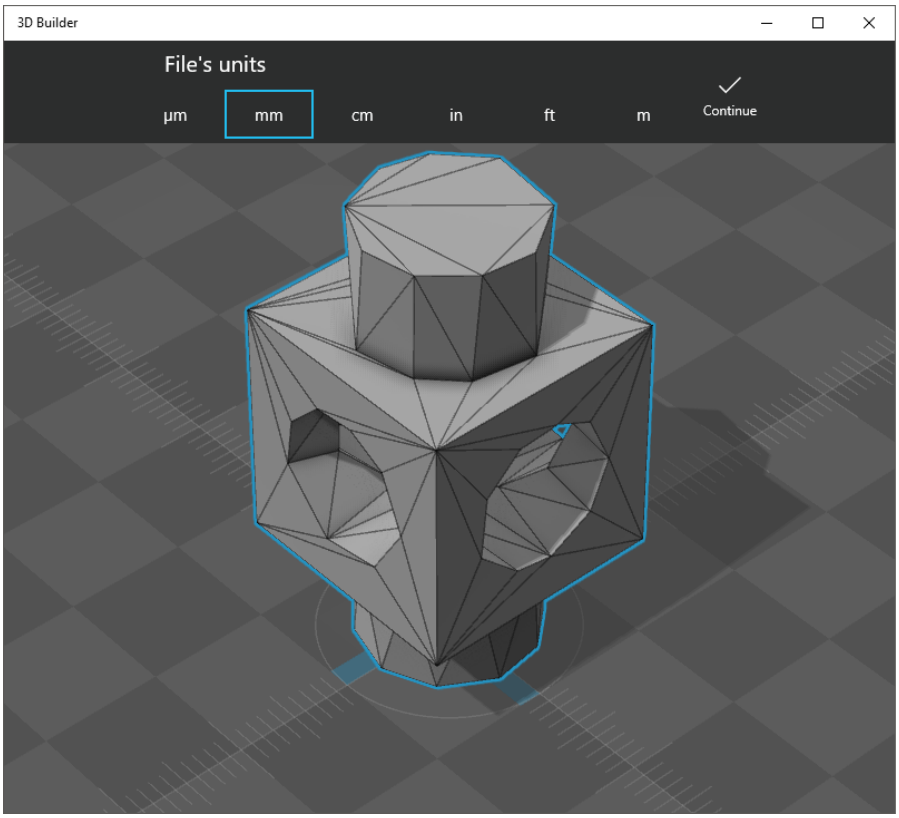


Figure 10: Block simplified to 200 faces from 4272.

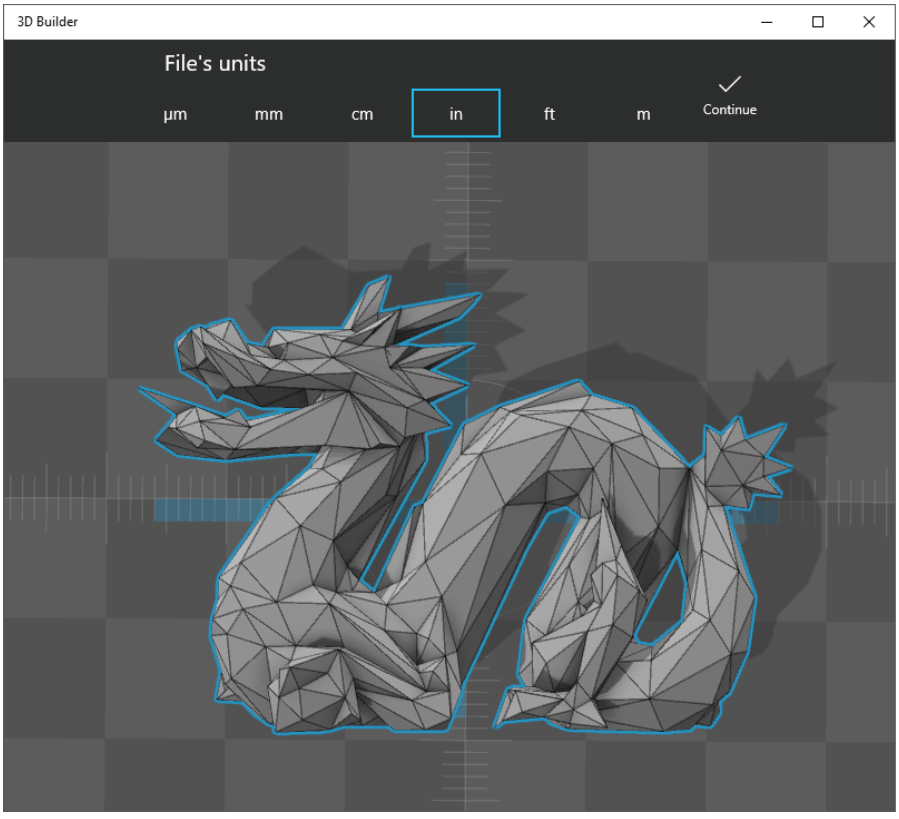


Figure 11: Dragon simplified to 1000 faces from 200k.

ACKNOWLEDGMENT Thank Wang Cunguang for he provided me with the OBJ model of our school gate (see 8).

REFERENCES

[1] Monte Carlo Ray Tracing, Siggraph Course, Pat Hanrahan et. al. http://www.cs.rutgers.edu/ decarlo/readings/mcrt-sg03c.pdf

[2] Physically Based Rendering From theory to implementation. Matt Pharr, Greg Humphreys.

[3] Computer Graphics Course Homepage, Carnegie Mellon University. http://www.cs.cmu.edu/afs/cs/academic/class/15462-f12/www/