Kungliga Tekniska Högskolan
Valhallavägen 79
100 44 Stockholm

Computer graphics and interaction

« Raytracing »

April 6^{th} - May 19^{th}

Author
Rémi Domingues 920604-T239
<remido@kth.se>

Teachers
Christopher Peters

Scholar year 2014-2015

Contents

Introduction

This document describes the implementation and output of a raytracer supporting triangles and spheres, direct light and soft shadows, diffuse, specular, semi-specular and glass materials, anti-aliasing and multithreading.

The model used in this experiment is the Cornell Box available at `http://www.graphics.cornell.edu/online/box`.

1 Ray-triangle intersection

By representing rays as lines in 3D, defined by a starting point and a direction (3D vector), we are able to compute the intersection between a ray and a triangle as described below. This can be expressed as a line-plane intersection, where we check that the intersection lies in the triangle using the vectors $T_1\vec{T}_3$ and $T_1\vec{T}_2$.

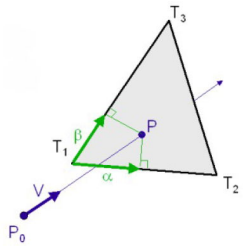

FIGURE 1 – Ray-triangle intersection

These computations are implemented in the following method, which returns the intersection (t u v) between a ray described by a starting point and a direction and a triangle, with t distance and u and v triangle delimiters.

```
vec3 GetIntersection(vec3 start, vec3 dir, Triangle triangle) {
        vec3 v0 = triangle.v0;
        vec3 v1 = triangle.v1;
        vec3 v2 = triangle.v2;
        vec3 e1 = v1 - v0;
        vec3 e2 = v2 - v0;
        vec3 b = start - v0;
        mat3 A(-dir, e1, e2);
        return glm::inverse(A) * b;
}
```

2 Tracing Rays

Using the pinhole camera model, rays are traced from the camera, lying in position (0, 0, -3) and directed toward the z axis. Their closest intersection are then computed in order to display the closest triangle.

A focal length equal to the screen width has been used. With $width = height = 500$, we obtain the following field of views :

$$\begin{aligned} FOV_{vertical} &= 2 * arctan(\frac{\frac{height}{2}}{f}) = 2 * arctan(\frac{500}{500}) = 90° \\ FOV_{horizontal} &= 2 * arctan(\frac{\frac{width}{2}}{f}) = 2 * arctan(\frac{500}{500}) = 90° \end{aligned} \tag{1}$$

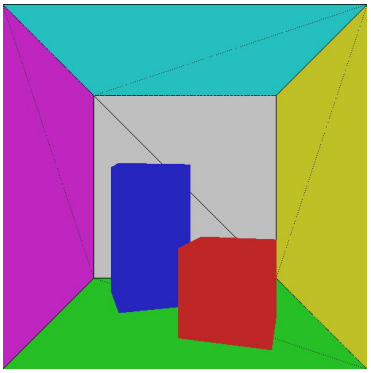

FIGURE 2 – Rendering output

3 Moving the Camera

A camera rotation on the y-axis has been introduced by multiplying the rays direction vector by the rotation matrix below, with θ the camera angle.

$$R = \begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{pmatrix} \tag{2}$$

Then, obtaining the axes direction using equation 3, we can apply a camera translation by adding the corresponding axes direction to the camera position.

$$\begin{aligned} \vec{x} &= (R[0][0], R[0][1], -R[0][2]) \\ \vec{y} &= (R[1][0], R[1][1], R[1][2]) \\ \vec{z} &= (-R[2][0], R[2][1], R[2][2]) \end{aligned} \tag{3}$$

These translations and rotations are triggered by pressing the arrow keys.

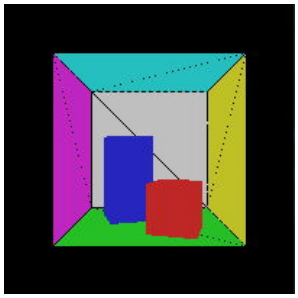

FIGURE 3 – Backward translation

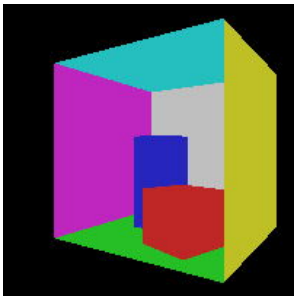

FIGURE 4 – Right rotation and translation

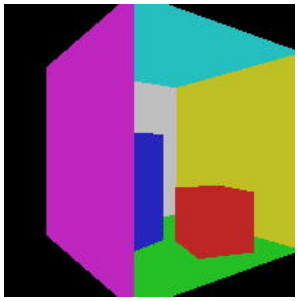

FIGURE 5 – Left rotations and translations

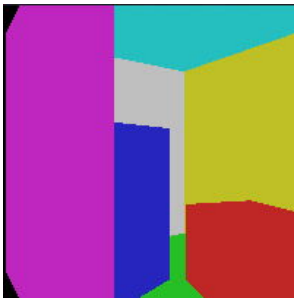

FIGURE 6 – Forward translation

4 Illumination

4.1 Direct light

Direct light is computed by estimating the light power reaching a triangle. This data is processed for every intersection computed in sections 2 and 3. This estimation only does not take into account obstacles between the light source and the intersection, which is why the renderings below do not contain any shadow.

The power D is explained in equation 4, using P the power of the light source, i.e. energy per time unit for each color component (14.f * vec3(1, 1, 1) for example), r the distance from the intersection to the light source, $\hat{n}$ the normal of the triangle and $\hat{r}$ a unit vector describing the direction from the surface point to the light source.

$$D = \frac{P * max(\hat{r}.\hat{n}, 0)}{4\pi r^2} \tag{4}$$

Using the same translation as described in section 4, the light source can be moved in the 6 possible directions using W, A, S, D, Q and E.

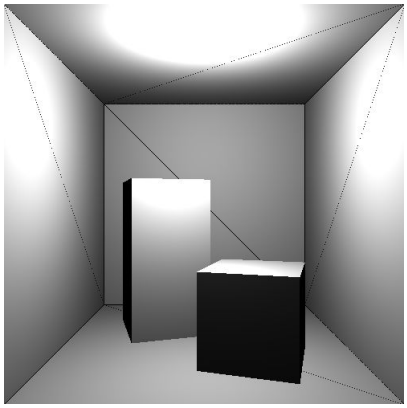

FIGURE 7 – Direct light

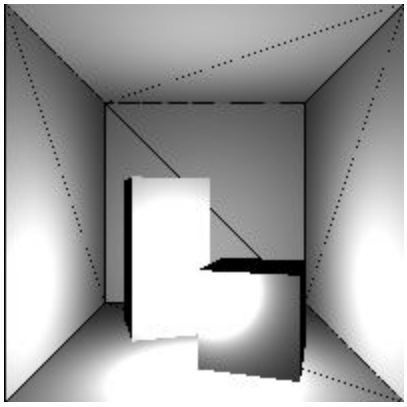

FIGURE 8 – Moving down

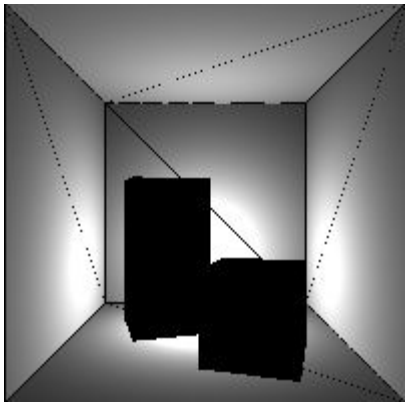

FIGURE 9 – Moving forward

Multiplying the triangle color by the power D for each color component reaching the intersection, we obtain the following renderings.

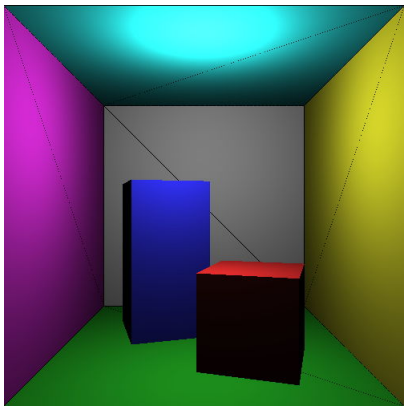

FIGURE 10 – Direct light

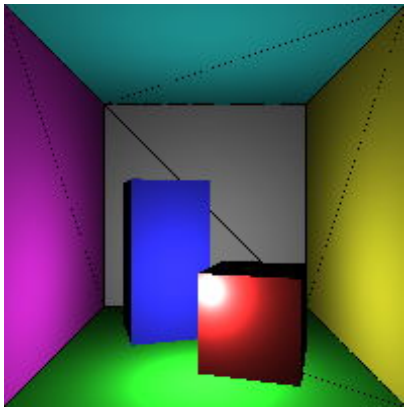

FIGURE 11 – Moving down

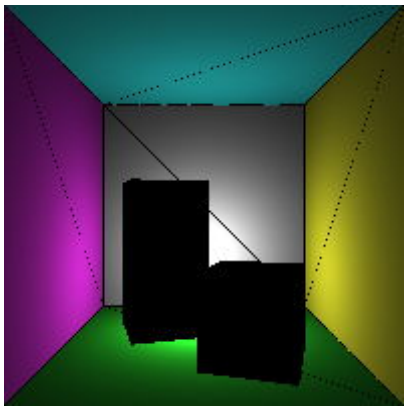

FIGURE 12 – Moving forward

4.2 Direct shadows

In order to make this scene more realistic, we render shadows by tracing rays from the light source to our intersections. If the intersection returned is closer than the distance between the intersection and the light source, this mean that light is hidden by another object. Light power is thus set to 0 for each color component.

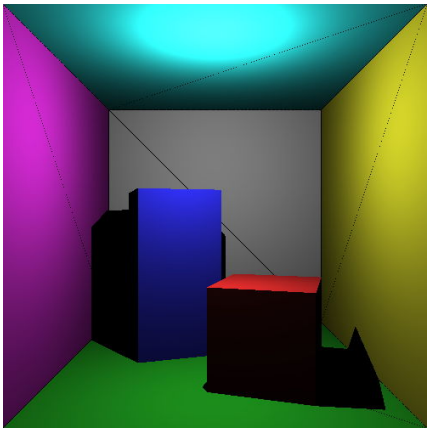

FIGURE 13 – Direct shadows

4.3 Indirect illumination

Eventually, to avoid black shadows and simulate the indirect illumination obtained by multiples bounces of rays from the light source, we add an indirect illumination constant to the power received by every intersection.

Note that computing those bounces would result in a rendering much more realistic but also in a loss of performances.

Therefore, the color of a triangle is now computed as follows, with N the indirect illumination constant and ρ the RGB vector of a triangle (reflectance).

$$R = \rho * (D + N) \tag{5}$$

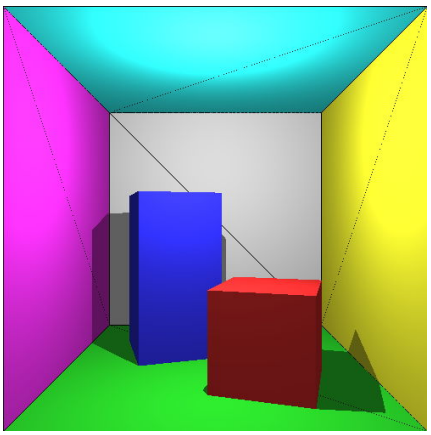

FIGURE 14 – Indirect illumination

5 Spheres

The first improvement brought to our raytracer is a sphere. This object should not be hard to include in the existing algorithm since the same kind of computations must be applied to a triangle and a sphere. Both are 3D objects, they have a reflectance ρ, are made of a specific material (only diffuse for now), have a normal and must be able to compute their intersection with a ray.

Where a triangle is described by its 3 vertices, a sphere has a center C(cx, cy, cz) and a radius R. The intersection between a sphere and a ray is explained below, with P(px, py, pz) the starting point of the ray, $\vec{v}$ the ray direction and I the intersection.

$dx = \vec{v}.x$

$dy = \vec{v}.y$

$dz = \vec{v}.z$

$a = dx^2 + dy^2 + dz^2$

$b = 2dx(px - cx) + 2dy(py - cy) + 2dz(pz - cz)$

$c = cx^2 + cy^2 + cz^2 + px^2 + py^2 + pz^2 - 2(cx * pc + cy * py + cz * pz) - R^2$

The discriminant is $d = b^2 - 4ac$

If $d < 0$ there is no intersection, if $d = 0$ the ray is tangent to the sphere and intersects it in one point, if $d > 0$ the ray intersects the sphere in two points.

To compute the closest intersection, solve our equation by computing m, $m = \frac{-b-\sqrt{d}}{2a}$

This value represents the number of times $\vec{v}$ must be added to P to reach the closest intersection I.

Therefore, I(ix, iy, iz) is

$ix = px + tdx$

$iy = py + tdy$

$iz = pz + tdz$

Now that we have our intersections, we must still compute the normal $\vec{n}$ at the intersection to obtain the amount of light reaching our intersection. The normal direction depends on the relative position between the intersection and the center of the sphere. Therefore,

$$\vec{n} = \frac{I - C}{R} \tag{6}$$

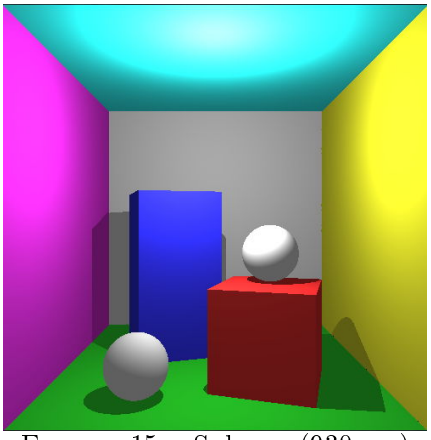

FIGURE 15 – Spheres (930 ms)

6 Ray-triangle intersection : optimization

The intersection algorithm is the core of our raytracer and the most called method in our entire program. It is also a bottleneck which could be optimized and lead to great performances.

By trying to reduce the render time of the ray tracer, I found an interesting paper describing a fast ray-triangle intersection algorithm by T. Möller and B. Trumbore [2].

While the algebra used for the calculations is slightly more complex than our previous algorithm, this implementation happens to be way faster, and reduced the render time of the overall process by 2.6 times. Our scene is now rendered in 360 ms instead of 930 ms.

7 Specular materials

The performances have been improved but our scene could be better. An interesting effect, which can also be very costly, would be to add specular materials.

The way to do so is quite simple. Previously, the color of a pixel was obtained by multiplying the reflectance of the intersected object to the power light hitting the area, plus the indirect light constant. This procedure won't change, but we now check the material of the intersected object. If it's a specular material, we trace a new ray from the intersection and the color of the pixel will be the one hit by the new ray.

The direction of the new ray is described in equation 7, where $\vec{d_i}$ is the incident ray direction, $\vec{n}$ the normal at the intersection and $\vec{d_r}$ the reflected ray direction.

$$\vec{d_r} = \vec{d_i} - 2(\vec{n} \cdot \vec{d_i})\vec{n} \tag{7}$$

This operation can be costly if we have many bounces between two specular materials, but it will also give a very nice visual effect. To prevent infinite bounces, we must however limit the maximal number of bounces, and return a default value when this threshold is reached. In the figure below, we can observe the multiple bounces with the left purple

wall rendered in the mirror of the tall blue box after a bounce on the specular sphere. The black color reflected is the default color assigned to the "void", i.e. when a ray don't hit any object.

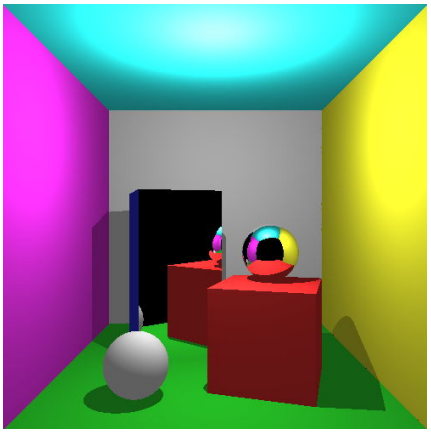

FIGURE 16 – Specular materials

8 Glass refraction

Glass is another interesting material. Yet, this one is tricky since is should refract and reflect rays, each ray having its own percentage of light transmitted according to the incident ray angle. In a first step, we will only implement refraction.

The refraction and reflection formulas to compute those directions and percentages are described in by T Whitted in [4]. However, a faster method, which can actually be proven as equivalent to the previous one using trigonometry and vector algebra, is detailed by PS Heckbert in [1].

The idea lying in those papers is to first compute the cosine angle between the incident vector and the normal at the intersection.

$cos\theta_1 = \vec{d_i} \cdot \vec{n}$

If the cosine is higher than 0, the incident and normal vectors have the same direction. If so, this means that we are inside the material, so we have already performed a refraction and want to perform a second one. We must therefore use $-\vec{n}$ instead of $\vec{n}$ to compute the next ray direction.

We respectively define i_1 and i_2 the refractive indices of the current and next material. We remind that $i_{air} \approx 1.0$ and $i_{glass} \approx 1.52$ for most glass materials (nice effects can be obtained by using crystal and caustics, but this feature has not been implemented here).

$\eta = \frac{i_1}{i_2}$

$$cos2\theta_1 = 1 - \eta^2(1 - cos\theta_1^2) \tag{8}$$

If $cos2\theta_1$ is lower than 0, the incident ray is completely reflected. This effect, happening when the angle between the incident ray and the normal is higher than a certain threshold,

is called *total internal reflection*. $cos2\theta_1$ can actually be approximated as the percentage of light refracted. This is a very useful measure.

The reflected ray direction is then calculated in the same way as in equation 7. The refracted ray direction is details in equation 9 where d_R is the refracted ray direction.

$$\vec{d_R} = \eta\vec{d_i} + (\eta cos\theta_1 - \sqrt{cos2\theta_1})\vec{n} \tag{9}$$

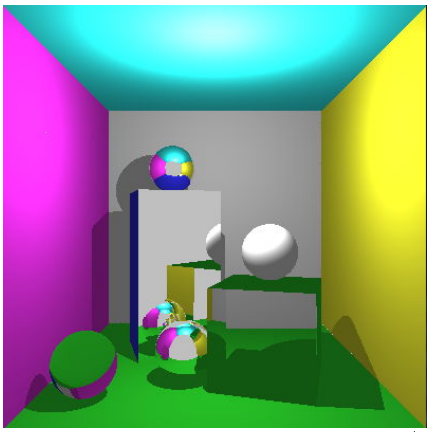

FIGURE 17 – Refraction through glass (430 ms)

As specified before, the reflection is not rendered in the previous figure. However, we can observe a total internal reflection with the yellow wall reflected in the mirror after multiple bounces in the glass. Yet, if most of the light should be reflected by the top of the box due to the incident angle, we haven't applied yet the percentage to the refracted ray, which explains the green color.

The smoked glass effect is obtained by reducing the refracted light transmitted by 5% when entering the object.

9 Anti-aliasing

Our scene looks better now. Though the geometry could be improved. Many artifacts could be observed on the edges of figure 17. This aliasing is quite annoying, let's fix it. The following describes various anti-aliasing methods supported by the raytracer. Those algorithms are applied in post-processing, i.e. after the scene has been generated.

9.1 Edges detection

Since anti-aliasing can be a costly operation, we want to target only the pixels on the edges, this will avoid an important waste of time.

Instead of displaying a pixel immediately after its computation, we store them in a 2D matrix representing our screen and containing the RGB vector of each pixel. When our scene is complete, we transform this matrix in a grayscale matrix containing values from 0

to 255 instead of RGB vectors. The transformation in equation 10 respects the CCIR 601 recommendations. I is the grayscale intensity, and ρ a RGB vector (reflectance). Figure 18 is the output of this transformation.

$$I = \rho \cdot (0.2989, 0.5870, 0.1140) \tag{10}$$

Once we have computed the intensity, we must detect the intensity variations in the scene. This is achieved using the Sobel operator [3]. We define p_1 ... p_9 the nine values constituting a 3x3 matrix, where p_1 is above and to the left, p_5 is the pixel itself, and p_9 is below and to the right. If the result of equation 11 is higher than a threshold (here 0.5), we add the pixel coordinates to a list on which anti-aliasing must be applied. Those pixels are displayed in white in figure 19

$$g = |(p_1 + 2 * p_2 + p_3) - (p_7 + 2 * p_8 + p_9)| + |(p_3 + 2 * p_6 + p_9) - (p_1 + 2 * p_4 + p_7)| \tag{11}$$

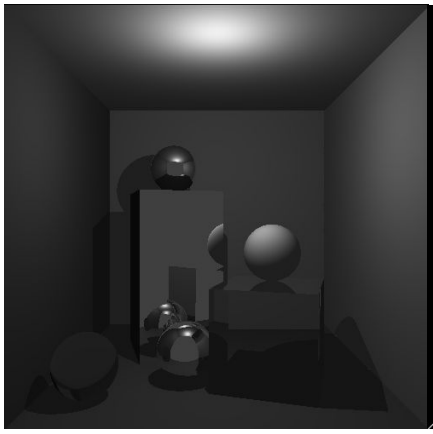

FIGURE 18 – Grayscale scene

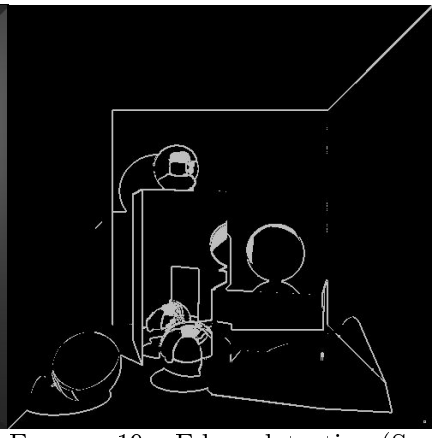

FIGURE 19 – Edges detection (Sobel operator)

9.2 Uniform

Once we know which pixels need anti-aliasing, we can improve the accuracy of our edges by shooting multiple rays in various directions inside each pixel. This process, also called supersampling and illustrated in figure 20, soften the edges by averaging the value of the current pixel and the one of 8 new rays, each ray having the weight $\frac{1}{9}$. Note that the rays must be traced inside the pixel. Averaging the values of the pixels around would only result in a blur instead of improving the accuracy.

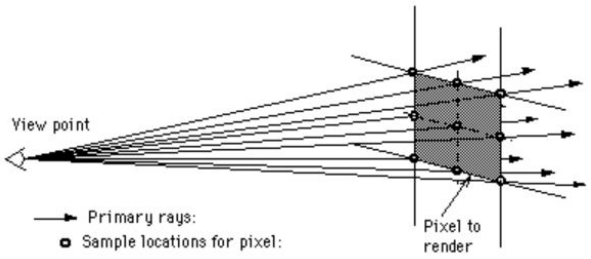

FIGURE 20 – 8 new rays are shot per pixel

Figure 23 shows the result of this algorithm. The image is a zoom on the mirror after a few forward camera translations and a rotation.

9.3 Stochastic sampling

The previous method improves our edges, but the uniform distribution of the rays on the pixel sides preserves the edge geometry and aliasing is still likely to occur.

Since the human eye is very efficient at identifying patterns and their discontinuations, stochastic sampling aims at adding noise to the edges in order to make this task harder, resulting in soften edges perceived.

To do so, we divide each pixel in a grid, e.g. 4x4 for an anti-aliasing 16x (figure 21). Instead of only shooting rays on the sides of the pixel, we now shoot one ray per grid cell. The coordinates of the ray shot per cell are randomly sampled inside the cell. We eventually average the value returned by those rays with the current pixel value, each ray having here a weight of $\frac{1}{17}$. Note that we could also ignore the value of the current pixel, but more data is always better.

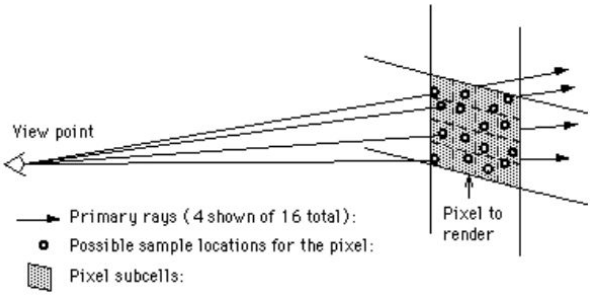

FIGURE 21 – 16 new rays are sampled on a grid inside each a pixel

Our raytracer allows a 2x, 4x, 8x, 16x and 64x anti-aliasing by stochastic sampling.

9.4 Results

The following pictures describes the results obtained by some of the previous anti-aliasing algorithms. The computation time is also mentioned.

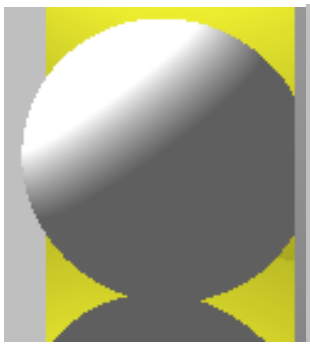

FIGURE 22 – AA disabled

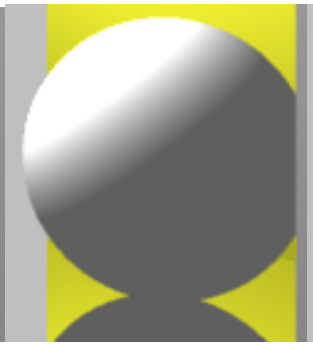

FIGURE 23 – Uniform 8x

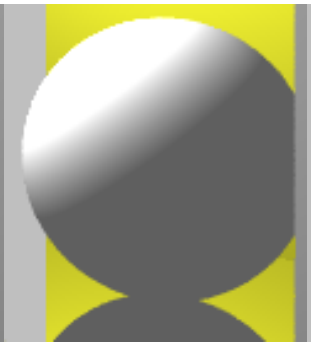

FIGURE 24 – Jittered 8x

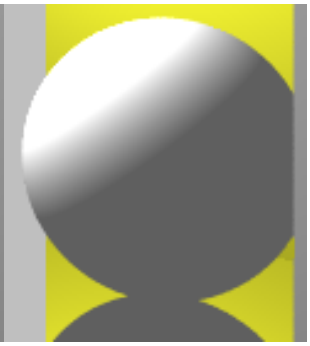

FIGURE 25 – Jittered 16x

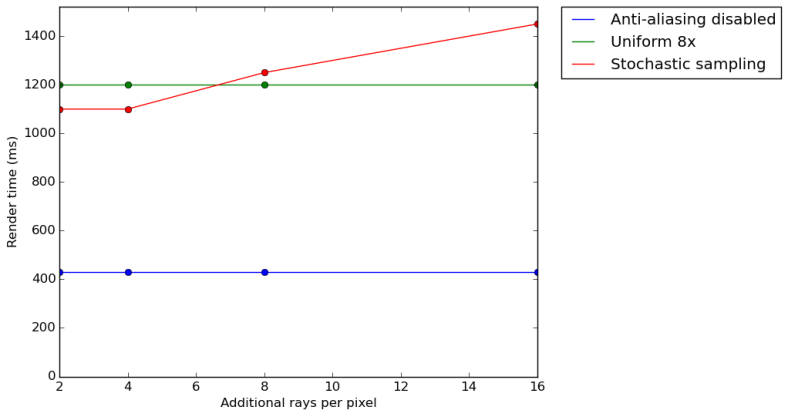

FIGURE 26 – Anti-aliasing benchmarks

Based on the previous figures and as expected, a stochastic sampling 16x seems to give the best edges.

Another interesting method, not implemented here, is called adaptive sampling. This algorithm shoots 1 ray at each corner of a pixel and then compare their grayscale intensity. If these intensities varies significantly, the pixel is divided into 4 cells and the same algorithm is applied for each cell, until a certain deepness is reached.

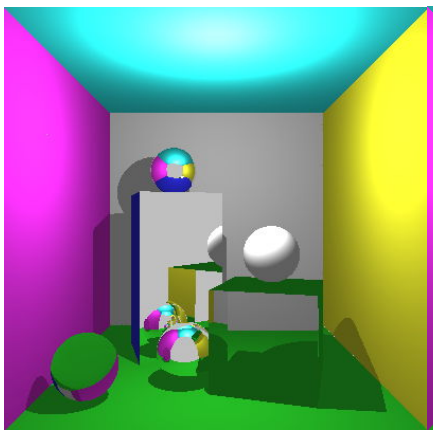

FIGURE 27 – Anti-aliasing disabled (430 ms)

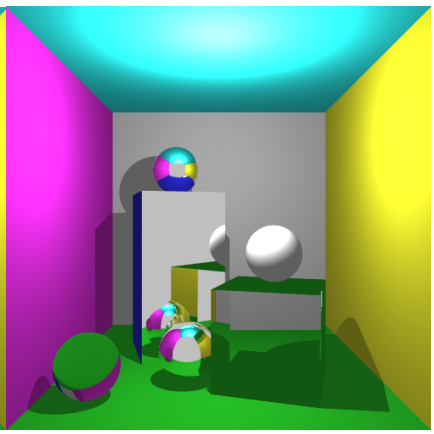

FIGURE 28 – Stochastic sampling 16x (1450 ms)

10 Glass reflection, semi-specular materials

Our scene is getting good, but we can still add a few improvements. First, we remove the light filter applied on the smoked glass, and replace it with a real percentage (equation 8. We also compute the color of a glass material with equation 12 by shooting a reflected in addition to a refracted ray for each intersection with a glass material. $\rho(I, \vec{v})$ if a function returning the reflectance (RGB color) of a point when shooting a ray from an intersection I in a direction $\vec{v}$. $cos2\theta_1$ is the percentage of refracted light, $\vec{d_R}$ is the refracted vector direction and $\vec{d_r}$ the reflected vector direction.

$$\rho_I = cos2\theta_1 * \rho(I, \vec{d_R}) + (1 - cos2\theta_1)\rho(I, \vec{d_r}) \tag{12}$$

As for a specular material, we limit the number of bounces. We also change the void color and the color of the diffuse materials.

To make it even better, we introduce a new material, which is diffuse and specular. When a ray will hit this material, a part of the light will be absorbed by the material while some light will bounce like a specular material. The color of such a material is given in equation 13, where r is the percentage of reflected light, here 0.8. We could also have taken the incident angle into account, but wanted a material more uniform.

$$\rho_I = r\rho_I + (1 - r)\rho(I, \vec{d_r}) \tag{13}$$

The result looks great, especially regarding the specular sphere reflected less and less by the glass cube with the number of bounces increasing. This rendering has been computed in 8500 ms with 20 bounces max and anti-aliasing by stochastic sampling 16x. The sphere in the bottom left if made of glass, while the right sphere in the wall is specular.

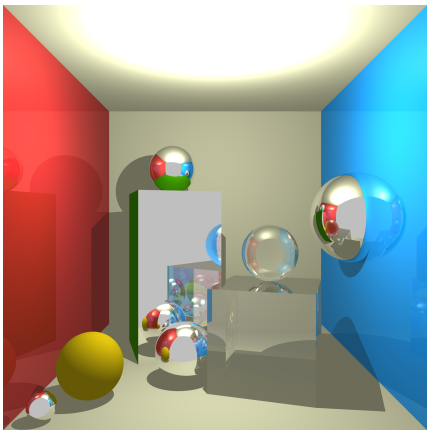

FIGURE 29 – Glass : refraction and reflection

11 Soft shadows

We have a great scene, but the light source looks very unrealistic. Actually, using a single point as light source also results in hard shadows, which could be improved. Let's implement improve it.

We start by adding new triangles to the ceiling of the Cornell Box in order to create a rectangular hole. To prevent that rays going through this whole simply return the void color, we add two other triangles on top of the box. If a ray hit that light surface, we don't compute its illumination according to its color but rather return the light power as a pixel color.

11.1 Uniform light sources

Until now, we processed the color of an intersection, multiplied it by the power hitting the intersection (which was (0, 0, 0) if another object was found between the intersection and the light source) and added an indirect illumination vector.

This method won't change, but we now introduce multiple light sources. The previous light position is now considered as the light center. We compute a NxN grid centered on this point, and put a light source in the center of each cell.

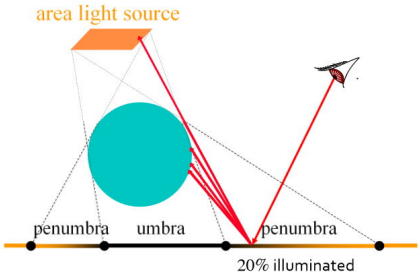

FIGURE 30 – Soft shadows

In the following equation, $P(I, L_i)$ return the light power hitting an intersection I given a light source L_i according to the light distance and the normal of the object. $(0, 0, 0)$ is returned if the light source cannot be reached. C is the final pixel color.

$$C = \frac{\sum_{i=1}^{N*N} P(I, L_i)}{N * N} \tag{14}$$

The following scenes were processed in a 500x500 resolution without anti-aliasing.

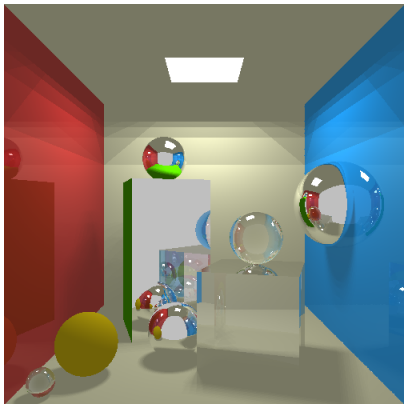
FIGURE 31 – 16 lights, uniform (8700 ms)

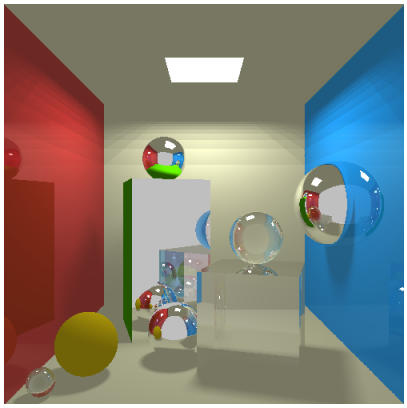
FIGURE 32 – 64 lights, uniform (30 600 ms)

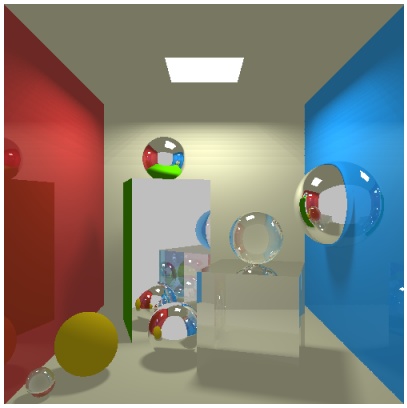
FIGURE 33 – 256 lights, uniform (118 500 ms)

The computation cost of this technique is high, but the shadows look better. Yet, even with 256 lampes (16x16 grid), we can still see that our light source is composed of multiple light sources.

11.2 Jittered light sources

To make it look even smoother we use the same idea as described for the anti-aliasing using stochastic sampling. The grid is of course wider than a pixel, the grid width and height is actually $\frac{1}{6}^{th}$ of the box size. Instead of having a light in the center of each cell, we now randomly sample the position of each light in the cells.

If done only once, this process would give the same result as before. That's why we sample the light positions for each intersection. This has a small additional cost but the result is way better. Note that if we don't have enough light, shadows are very noisy.

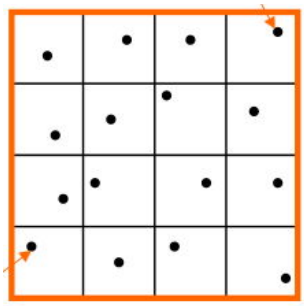

FIGURE 34 – Light sources jittered by stochastic sampling

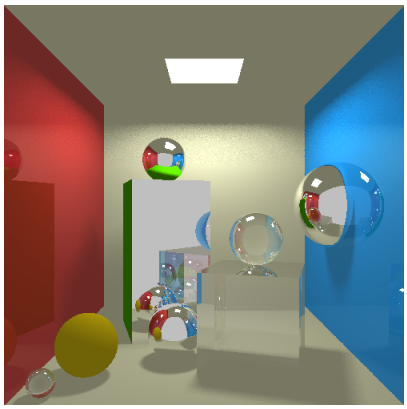

FIGURE 35 – 16 lights, jittered (9400 ms)

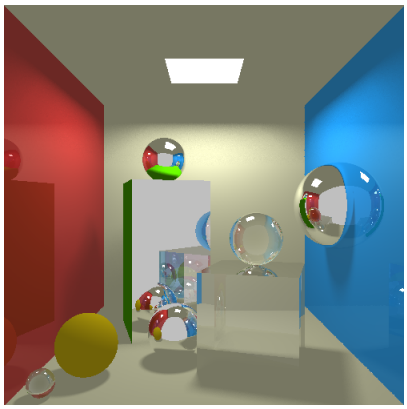

FIGURE 36 – 64 lights, jittered (33 600 ms)

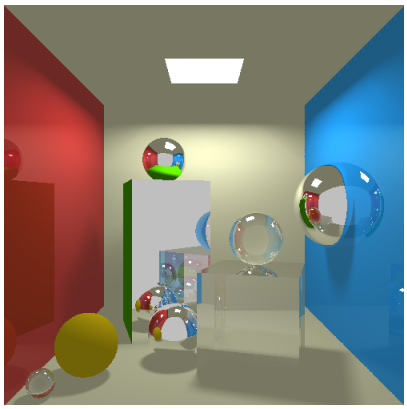

FIGURE 37 – 256 lights, jittered (128 600 ms)

Our raytracer support uniform and jittered light sources for any number of lights L distributed on a NxM grid (NxM = L).

12 Multithreading

We could stop here, but we want to make our light sources look slightly better. We add a few more rectangles under the hole in the ceiling to make it look like a real light source.

The void is also annoying since it does not support illumination. That's why we extend the walls and close the box behind the camera. We love specular surfaces, so we use the diffuse and specular material for all the walls. Even with limiting the number of bounces, we now have a very very important computation cost, which can take up to a few minutes. The solution ? Multithreading.

12.1 Scene processing

In a first time, we only make parallel the scene processing, i.e. the anti-aliasing stays sequential. Since we iterate through every pixel of our screen in order to trace a ray to get their color, we decide to divide the screens into equal parts, giving each part to a thread. If the number of pixels cannot be divided by the number of threads, we linearly distribute the remaining pixels, i.e. if our screen of 101x101 is divided in a grid of 3x3 threads, the two first threads will in each row and column will have one more pixel in width and height. Each thread has therefore its own width and height, a width and height offset but access to the same pixel matrix as the other threads.

Since each thread writes in its own part of the pixels matrix, two threads will never write in the same pixel, so the screen is not a critical resource. However, a synchronization is needed after processing the scene to sequentially compute the grayscale of the whole pixels matrix.

Multithreading benchmarks are plotted in figure 38. Scene is 250x250, 1 uniform light source, 15 bounces max. If an anti-aliasing is used, it's stochastic sampling 16x. Computations are performed on a quad-core hyperthreaded. The *No AA* curve shows a nice speedup when increasing the number of threads. However, by looking at the *Sequential AA* curve, it's quite obvious that we have too much sequential code in our implementation when using anti-aliasing.

12.2 Anti-aliasing

We improve our performances by building a list containing the coordinates of the pixels needing anti-aliasing, and dividing this list into equal parts assigned to each thread (we actually don't divide the list, but give an offset and number of pixels to compute to each thread). Each pixel eventually compute anti-aliasing for each thread it has been assigned.

A synchronization is set after computing the anti-aliasing, since displaying pixel on the screen is a sequential code due to the fact that the screen is a critical resource.

The *Multithreaded AA* curve looks better than the previous ones. Yet, a lot of time is wasted by waiting at the synchronization barrier. Indeed, some parts of the scene are longer to compute than others, and the thread responsible for the part containing the two

specular surfaces and the glass cube has to compute a lot more bounces.

12.3 Optimization

To balance the computation between every thread, we use another philosophy. Processing a pixel or applying anti-aliasing to a pixel is now considered as a job, and each thread is now a worker. Each worker starts with a job, and as soon as its job is finished, it starts another. By doing so, we guarantee that every thread will spend almost the same amound of time processing or post-processing the scene, and that the synchronization won't last longer than the computation time of one pixel.

Regarding the scene processing, we use a pair of coordinates (*pair<int, int>*) as critical resource protected by a mutex. This structure contains the (x, y) coordinates of the next pixel available for a job. When reading those coordinates, a worker updates them for the next thread (x is incremented or set to 0 if equal to width - 1, y is incremented if the new x is equal to 0).

The anti-aliasing work in the same way. We already have a list of pixel positions for which post-processing is needed, so we simply replace the pair of coordinates by the one in front of the list in the critical section protected by a mutex.

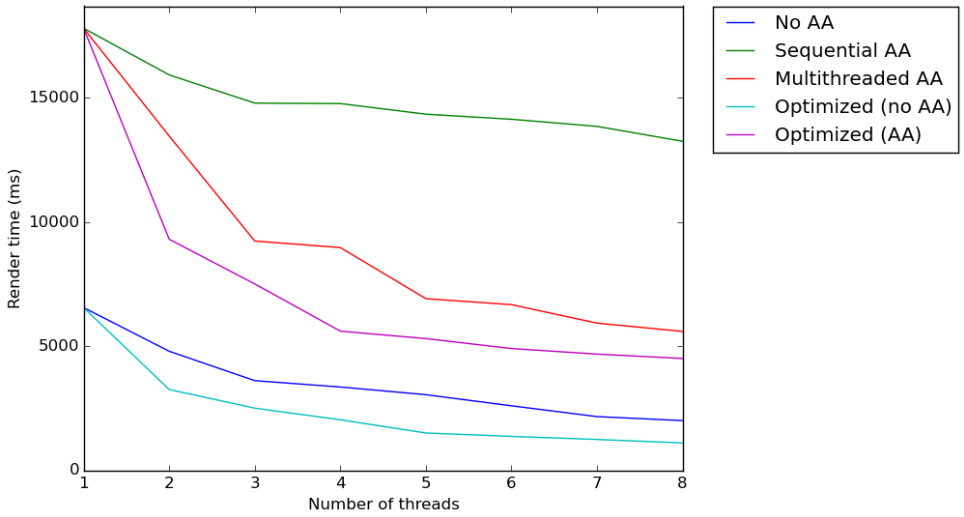

FIGURE 38 – Multithreading benchmarks

Eventually, we now stop tracing rays if the percentage of light transmitted is lower that a certain threshold, since that percentage can be very low after bounces in diffuse and specular surfaces. Our box containing a majority of specular surfaces, this last point reduced our render time by more than ten times.

This final scene has been processed by 8 threads in 55 minutes. Scene configuration is 2000x2000, 256 jittered lights, 20 bounces, minimum weight per ray is 0.0001, anti-aliasing by stochastic sampling 16x. Without our previous optimizations, this scene would have needed more than 50 hours.

FIGURE 39 – Final scene

Future work

The following features would add a lot of value to our raytracing algorithm.

- **Unix support :** The current multithreading functionality uses the Windows threading library. Allowing users to run this program on both Windows and Unix-based systems would be a great improvement.
- **Global illumination** : One way to improve our current scene would be to implement global illumination using algorithms such as photon mapping. This algorithm could even be extended to add **caustics**, which will add a very nice lighting effect using glass materials.
- **Loading general models** : This feature opens many scene possibilities and would allow us to use our raytracer on pre-defined complex models
- **Water surfaces** : Generating water surfaces could be achieved by using normal maps.
- **Lens flare** : This light effect is an artifact caused by material inhomogeneities in the lens of a camera. This effect could be add by post-processing and will make the scene more realistic.

References

[1] Paul S Heckbert. Derivation of refraction formulas. *Introduction to Ray Tracing*, pages 288–293, 1989.

[2] Tomas Möller and Ben Trumbore. Fast, minimum storage ray/triangle intersection. In *ACM SIGGRAPH 2005 Courses*, page 7. ACM, 2005.

[3] Irwin Sobel and Gary Feldman. A 3x3 isotropic gradient operator for image processing. 1968.

[4] Turner Whitted. An improved illumination model for shaded display. In *ACM SIGGRAPH Computer Graphics*, volume 13, page 14. ACM, 1979.