FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Interactive Scene Manipulation Techniques for Ray Tracing

BACHELORTHESIS

zur Erlangung des akademischen Grades

Baccalaureus

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Kevin Streicher
Matrikelnummer 1025890

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Msc. Károly Zsolnai

Wien, 25.03.2013

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Interactive Scene Manipulation Techniques for Ray Tracing

BACHELORTHESIS

submitted in partial fulfillment of the requirements for the degree of

Baccalaureus

in

Media informatics and visual computing

by

Kevin Streicher
Registration Number 1025890

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Msc. Károly Zsolnai

Vienna, 25.03.2013

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Kevin Streicher
Schlachthausgasse 46/1/2, 1030 Vienna

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

I would like to express my gratitute to my advisor Károly Zsolnai, not only for supervising an interesting topic, but also for opening my mind for an amazing field of research.

I also would like to thank Christian Hafner for providing some of his previous work and the nice rendering related talks.

Contents

Abstract

‘Ray tracing is the future and will ever be’. This was the title of the ray tracing course at SIGGRAPH 2013, which shows what an important field of research ray tracing currently is. The most important reason ray tracing or path tracing has not yet replaced rasterization are the long computation times. While ray and path tracing already have replaced rasterization for offline rendering in general, we still rarely see it in real time applications. With a lot of promising results presented in the last years we can expect ray tracing to become more popular in the next years.

In interactive applications fast response times are needed to create smooth and usable tools. Many different things influence the final rendering time, like the number of refractive and reflective objects, pixels covered by those objects and objects in general. We have developed a basic scene designing tool using ray tracing and have benchmarked different code styles and the impact of various of these parameters on the final render time. We also present a simple technique to determine which regions require re-rendering when changes are introduced to the scene, allowing to save a considerable amount of computation time.

When using scene editing designing tools, for certain features, it is usually desirable to trade artifacts, higher noise levels or reduced image quality for faster render times during interaction. In this thesis we propose different options for interactive scene designing and and our benchmark results as well as the implementation of our scene designing tool.

Kurzfassung

‘Ray Tracing ist die Zukunft und wird es immer sein’. Dies war der Titel des SIGGRAPH 2013 Kurses zum Thema Ray Tracing, was zeigt wie wichtig dieses Forschungsfeld im Moment ist. Der wichtigste Grund warum Ray Tracing oder Path Tracing Rasterisierung noch nicht weitgehend ersetzt haben sind die langen Renderzeiten. Während Ray und Path Tracing für Offline-Rendering Rasterisierung in den Hintergrund drängen, gibt es nur wenige Echtzeit-Ray Tracing Anwendungen. In den letzten Jahren wurden viele vielversprechende Arbeiten präsentiert und es ist zu erwarten das Ray Tracing in den kommenden Jahren weiter an Popularität gewinnen wird.

In interaktiven Anwendungen sind schnelle Responsezeiten notwendig um flüssige und verwendbare Anwendungen zu entwickeln. Viele unterschiedliche Faktoren beeinflussen die gesamte Renderzeit, wie lichtbrechende und reflektierende Objekte, die Anzahl an Pixeln welche von solchen Objekten abgedeckt werden oder die Anzahl an Objekten im Allgemeinen. Wir haben einen grundlegenden Ray Tracing basierten Szenendesigner entwickelt und den Einfluss unterschiedlicher Codestile sowie Features auf die Renderzeit gemessen und verglichen. Wir präsentieren außerdem eine einfache Technik zur Ermittlung welche Regionen eine Neuberechnung erfordern wenn eine Szene sich geändert hat. Dies reduziert die notwendige Berechnungszeit nennenswert.

Wenn Szenendesigner verwendet werden kann es für betimmte Bildeigenschaften wünschenswert sein, Artefakte und stärkeres Bildrauschen zu erlauben, um die Renderzeit während der Interaktion zu verringern. In dieser Arbeit stellen wir unterschiedliche Optionen für interaktive Szenendesigner, sowie unsere Implementation und unsere Benchmarks vor.

CHAPTER 1

Introduction

1.1 Motivation

In 2013 we still see several completely different rendering approaches, each with its own benefits and drawbacks. Rasterization is very fast and supported by graphic cards. When physical correctness is not a criterion and you need high frame rates rasterization still might be your choice. Ray tracing achieves physically correct images in terms of reflection and refraction and easily generates soft or hard shadows. Bokeh effects are also very easy to implement in a ray tracer. Although ray tracing can generate really convincing images, it still cannot cover some physical attributes which make images look unnatural for the trained eye. One of these features is indirect diffuse illumination (color bleeding) which is created by light paths including more than one diffuse surface.

Path tracing is a global illumination technique that enhances the idea of ray tracing by stochastic sampling the surface hemisphere for each surface hit point hit by a ray. Offline rendering often relies on either path tracing, photon mapping or other global illumination techniques creating images barely distinguishable from photographs. This comes with the price: a lot of processing power is used to accomplish this.

Real-time applications like computer games currently mostly rely on rasterization. However, for offline rendering, different approaches are in use. DreamWork's rendering engine uses point-based global illumination ([Eri12]). Pixar relies on hybrid techniques with their RenderMan® engine([Pixa]). SolidAngle developed an unbiased physically based path tracer named Arnold ([Sol]) which currently is not publicly available.

The reason so many companies work on different rendering engines are the specific needs developers have. The requirements are very dependent on model size like the 337 million triangle model with a size of 31.4 GB of the Boeing 777 ([Att12]), light path difficulty ([Wen12]) and whether a real-time solution is necessary. Mike Seymor published an overview of rendering engines in use in 2012 in [Mik12].

Ray tracing in real-time applications is currently a hot topic as the even more advanced path tracing gives us amazing images and achieving it in real-time is something we can expect in the

near future. A lot of research effort is invested in real-time ray/path tracing ([Jaca, Jacb]). For scene editing real-time path tracing is amazing as the artist can play with indirect lighting, caustics, participating media, depth of field, motion blur on the fly, which simplifies the tedious task of capturing a specific image. Real-time path tracing is advancing and we see quite promising results already: 3D editors are expected to be able to edit a scene with a level of quality close to current result images soon.

One of the near future goals of real-time path tracing is to develop advanced filtering techniques to reduce the stochastic noise generated and to get appropriate ray/path tracing hardware([Jacc]). Although there are already computer games ported to ray tracing ([Gra]), they either rely on server based rendering with thin clients or have to struggle with noise or low frame rates([Tom]).

Developers have invested a lot of work to make scene editing for ray tracers easier ([Pixb, Pixc]). To achieve real-time experience we need algorithms as fast as possible and graphic cards designed for ray tracing. Intelligent acceleration techniques are the key to provide a good interaction experience.

1.2 Interactive real time ray tracing

As path tracing in real time still have to solve the problem of noise and low frame rates the faster ray tracing is still an interesting topic. For the acceleration of ray tracing in interactive software we differentiate between image preserving and distorting methods, fast and slow code, feature preserving or loosing techniques.

1.3 Contributions

With this thesis a small scene designing tool was developed which is based on the *smallpaint* path tracer ([Ká]). The used AABB-Tree implementation was contributed by Christian Hafner, which was adapted to our needs. We used the efficient and robust ray-box intersection algorithm from [Uni05]. The glsl, C++ shader and file input code was taken from Simon Wallner's tutorial ([Sim]). MinGW 4.7 g++ compiler was used to compile the project. This is important to know as some of the proposed code improvements might have different impact on different compilers.

1.4 How to read this document

To learn the basic information necessary to understand ray tracing and develop a basic ray tracer, work your way through 'Ray Tracing and Rendering Equation' (Chapter 2). To learn about acceleration structures and how to speed up interaction in ray tracers, read 'Interactive Techniques for Ray Tracing' (Chapter 3). In 'Results' (Chapter 4) you will find C++ code styles we have benchmarked to be faster, our final rendering times and the impact of different features.

CHAPTER 2

Ray Tracing and Rendering Equation

2.1 Rendering and the Rendering Equation

Rendering is the process of generating an image out of a model with the use of a computer. There are different types of renderings. Artistic renderings focus on beauty, style or inducing a specific mood. Medical and technical renderings have the goal to aid as assistance and visualize what otherwise would not be visible. Physical rendering tries to reproduce what happens in nature.

It is impossible to take the continuous and infinite detail of the real world into account. We have to reduce the information to render an image in finite time. Discretizing the continuous information reduces complexity but can introduce artifacts. Instead of rendering arbitrary light paths only direct diffuse illumination and light reflected or transmitted is taken into account with with ray tracing.

We model the light falling into our eye by creating an discretized image plane made of $width \cdot height$ pixels. Instead of infinite different wavelengths we only use one per red, green and blue color channel. Every point on the image plane is associated with exactly one point on an object's surface. What we are interested in is the color of exactly that point, which we determine through the process of shading.

The total amount of energy is conserved, no energy can be lost. Even if we completely ignore light absorption, we know that the amount of light reflected from a point x cannot be higher than the light shining on x. Light sources are modeled as constant light emitters. With this assumption we can calculate the energy which is emitted into one outgoing direction depending on all other incoming directions (Figure[1] 2.1, Equation 2.2).

[1] http://en.wikipedia.org/wiki/File:Rendering_eq.png

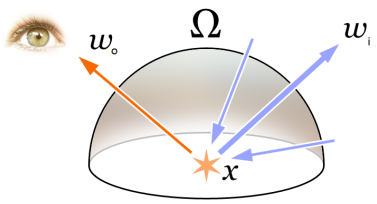

Figure 2.1: Surface hemisphere

$$L_{out} = L_{emitted} + L_{in} = L_{emitted} + \int_{\Omega} L_{incoming} d\omega'$$

Figure 2.2: Intuition behind the Rendering Equation

When a point x is hit by light from any direction Ω the direction of reflection and transmittance is defined by its material, surface and surface normal. Physical attributes like opaqueness define how much light will be transmitted. We model these attributes with the bidirectional reflectance distribution function (BRDF) and the amount which is scattered after transmission with the bidirectional transmittance distribution function (BTDF,Figure[2] 2.4). Together they are named bidirectional scattering distribution function (BSDF, Figure[3] 2.5).

Physically the BTDF is defined by the molecular structure of the material and the object's surface. A rough surface like sand will lead to a diffuse/matte shading as the rough surface scatters light in all directions whereby a mirror-like surface will have for each outgoing direction exactly one incoming direction. Between those two material types lie the glossy materials. Polished wood would reflect light primarily into one direction but scattering it sufficiently to blur the reflection. The rendering equation (Equation 2.3) describes the outgoing light for each point x.

$$\underbrace{L_o(x, \overrightarrow{\omega})}_{outgoing} = \underbrace{L_e(x, \overrightarrow{\omega})}_{emitted} + \underbrace{\int_{\Omega} L_i(x, \omega') f_r(\overrightarrow{\omega}, x, \overrightarrow{\omega}') \cos(\theta) d\overrightarrow{\omega}'}_{reflected}$$

Figure 2.3: Rendering Equation

The importance of the rendering equation (Equation 2.3) lies in its simplicity. We add the amount of energy emitted and light reflected in direction $\overrightarrow{\omega}$. If it would be analytically solveable, we could render physically correct images, but there are several limitations. Every hit point has an infinite amount of incoming and outgoing directions. Each point reflecting light on our hit point x depends on the light from every other point y, which depend on x again, and so forth.

The BDTF of a real object can be complex as we might have different densities in different regions of objects. There are materials which are frequency dependent and the physical behaviour of metal or phosphorescent materials add another layer of complexity. Other visual important features are subsurface scattering and the participation of particles like dust in the air.

[2] `http://collagefactory.blogspot.co.at/2010/04/brdf-for-diffuseglossyspecular.html`

[3] `http://en.wikipedia.org/wiki/File:BSDF05_800.png`

Each feature we calculate increases the needed amount of processing time, which is our main limiting resource. We have to simplify our analytically unsolvable infinite dimensional integral.

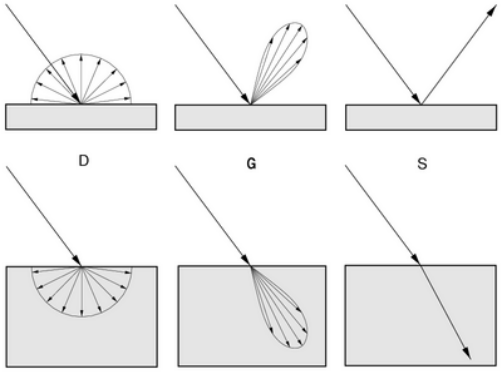

Figure 2.4: Different examples of BRDF and BDTF models

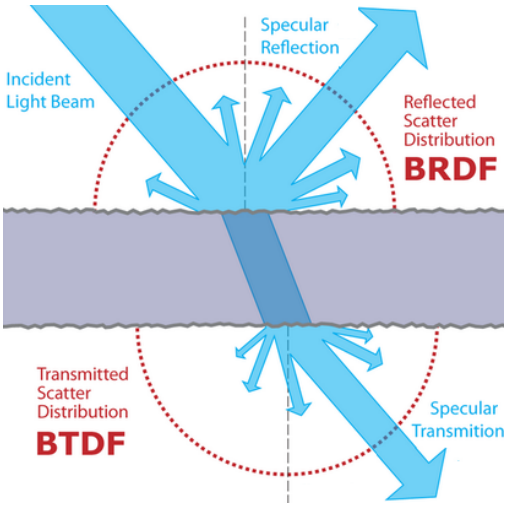

Figure 2.5: Bidirectional Scattering Distribution Function

2.2 Raytracing and Illumination equation

Each pixel on screen is associated with either empty space or the surface point of one object. We want to determine these hit points x by shooting a ray from the camera origin through the pixel position on our camera plane and determine the intersection point with each object (Figure[4] 2.6).

From all intersection points we choose the point with a minimum positive parameter t (Parameterized ray representation, Equation 2.2). The origin of the ray is our camera position and the direction from origin to the pixel on the image plane. We are interested in the value of t which gives us the hit point x on our object's surface. If t is small the hit point was close to our image plane. If t is negative it was behind it and we can ignore it. Self intersections with $t = 0$ are ignored and we will return 0 as if no object was hit.

A simplified version of the rendering equation is the illumination equation. The I term represents the illumination values of each individual light source. We use the phong illumination model ([W. 75]). k_d is the amount of light scattered as defined by the lambertian law and k_s is the amount of light reflected as defined by the law of reflectance. $k_a I_a$ is the ambient term, which is added to take account for the loss of energy due to our simplifications.

[4] `http://commons.wikimedia.org/wiki/File:Raytracing.svg`

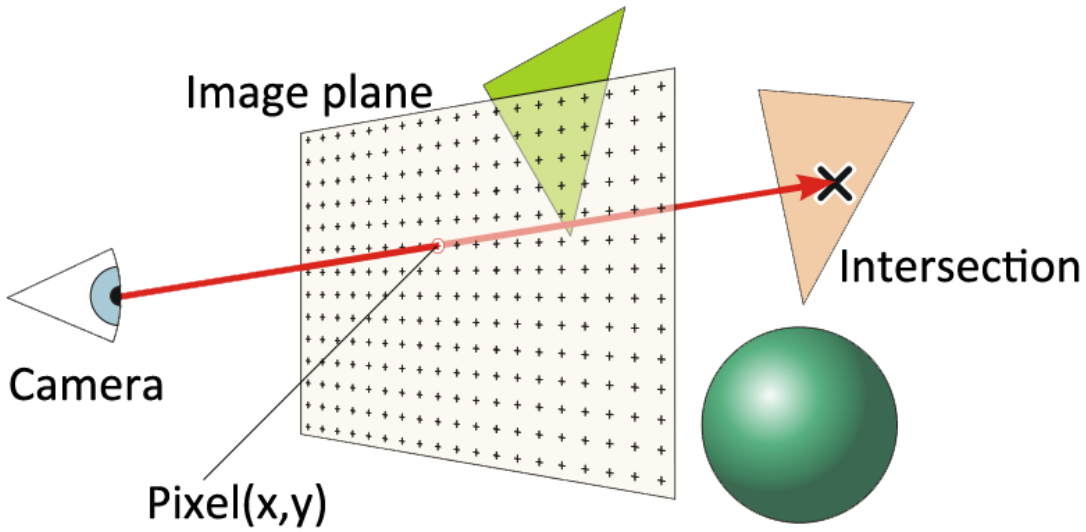

Figure 2.6: Raytracing principle

$$I = k_a I_a + I_i(\underbrace{k_d(\overrightarrow{L} \cdot \overrightarrow{N})}_{diffuse} + \underbrace{k_s(\overrightarrow{V} \cdot \overrightarrow{R})}_{specular}) + \underbrace{k_r I_r + k_t I_t}_{recursion}$$

Figure 2.7: Illumination Equation

$$\vec{r} = \vec{o} + \vec{d} \cdot t$$

Figure 2.8: Parameterized ray representation

When we add the light from the reflected direction $k_r I_r$ we can render reflective surfaces. With the term $k_t I_t$ we are able to add the light transmitted through an translucent object made of something like glass material (Equation 2.7). Each recursive term corresponds to another hit point x' on an object surface after a secondary light bounce.

2.3 Radiance, HDR and Tonemapping

Radiance describes the energy throughput as Watt per square meter per steradian. This is the energy passing through a surface weighted by the cosine of the incident angle. The human is able to see a high range of intensities but the range of a computer monitor is low. During ray tracing we measure radiance in a high dynamic range exceeding the range of display devices. The high range of radiance needs to be mapped down to the low range of the display device. This is called tone mapping.

The simplest way to handle HDR images is to clamp the values to our LDR, but the result looks usually bad. Linear mapping is better, but leads to areas which are too dark and contrast will be lost. The choice of operator depends on what features are considered important (Fig-

Figure 2.9: Different Tone Mapping examples

ure[5] 2.9, [Jü05, Lab00]). Tone mapping is usually not the biggest concern in ray tracing, but the right operator has to be chosen to achieve the sufficient performance while retaining a desirable degree of realism.

Tonemapping operators are categorized in local and global operators. Global operators are usually faster and use the same function for each pixel. Local Operators use functions depending on the local neighbourhood. Different operators try to preserve different features and not all take human perception into account. Possibilities to consider are contrast ([Gre92]), brightness ([Gen84]) or loss of focus ([Pro96]).

2.4 Ray Tracing Algorithm

The main idea of ray tracing is to follow the way light would travel through a scene. We shoot rays from our pixel through the scene and follow its way until we hit an object. On each hit point we gather the color of the hit object. When we shoot the rays from the camera origin through

[5] `http://commons.wikimedia.org/wiki/File:Tone_Mapping_Methods.jpg`

the pixel on the image plane the resulting image will be the same as a perspective camera would see. When the rays are sent out orthogonal to the image plane the resulting image would be one of an orthogonal camera model.

This concept leads us to minimalistic ray tracing. We are able to analytically solve the ray-object intersection. The color of the closest intersecting object, this is at least ϵ units far away, will be the plotted pixel color (Algorithm 1).

```
1  ray.origin = Camera.origin;
2  ray.direction = (Camera.currentPixel-ray.origin).normalize();
3  min = INFINITY;
4  foreach Object object do
5  |   t = ray.intersect(object);
6  |   if t < min and t > ε then
7  |   |   min = t;
8  |   |   pixel.color = object.color;
9  |   end
10 end
```

Algorithm 1: Minimalistic Ray Tracing

Ray tracing creates a 2D projection of the object on our image plane. Our minimalistic ray tracer does not give us any information about the third axis. A sphere will be rendered as a circle (Figure 2.10). Lambertian shading takes the surface normal and the incident angle of the light on the surface into account. The projection looks like a sphere and not like a circle (Figure 2.11). Multiple objects could be visible on one pixel, but we sample the pixel only once. This can lead to artifacts on object edges (Figure 2.12). One possibility to reduce this artifacts is to divide the pixel into smaller subpixels and average the subpixel-samples. This is called super sampling (Figure 2.13). Naive supersampling does sample every pixel of the plane multiple times. This increases the render factor a lot and is unecessary as homogeneous regions will not improve in quality as the same. Only subpixels with different objects, surface normals or other shading variables are needed to be supersampled, which is named adaptive supersampling.

There are more accurate models than the lambertian shading. One of them is the Oryen-Nayar model, which does not darken the object edges as strong as the lambertian, but is also more complex ([Dep94]). Adding lambertian shading and object materials to our minimalistic ray tracer (Figure 2.11,Algorithm 1) improves the rendered image (Algorithm 2). When using lambertian shading we are only interested in light hitting the front of the surface, we therefore forfeit light from an incident angle of zero or less.

1 *setup ray*;
2 **foreach** *Object object* **do**
3 | *find closest intersection point and get object normal at that point*;
4 **end**
5 $\overrightarrow{N}$ = object.normal;
6 $\overrightarrow{L}$ = (intersectionPoint - light.position).normalize();
7 cosTheta = max(0,$\overrightarrow{N} \cdot \overrightarrow{L}$);
8 pixel.color = object.material.color · cosTheta;

Algorithm 2: Ray tracing and lambertian shading

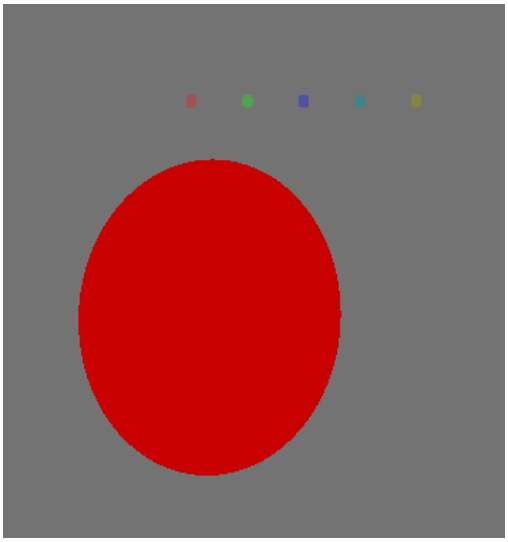

Figure 2.10: Without lambertian shading

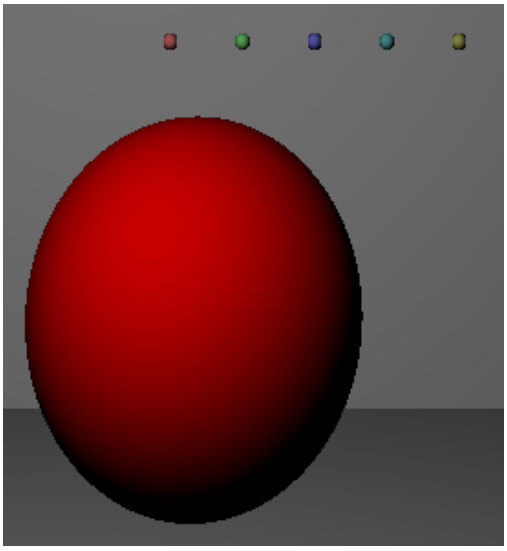

Figure 2.11: With lambertian shading

Figure 2.12: Discretization Artifacts

Figure 2.13: 8X Supersampling

The next step advancing our ray tracer is to add reflection and refraction. This is an important change as now we add one of the ray tracing features which are not that easily achievable with rasterization. We need to determine the direction our ray will be reflected in after hitting the surface point and the direction in which our light is transmitted. The reflection equation (Equation 8) calculates the direction in which the ray will be reflected. The view direction is the direction of the light path our ray is currently following. The fresnel-equations ([Alp]) would be the exact way to calculate the amount of energy reflected and transmitted, but as they are computation heavy we use schlick's approximation (Equation 2.15).

$$\text{R} = 2(\overrightarrow{V} \cdot \overrightarrow{N}) \cdot \overrightarrow{N} - \overrightarrow{V}$$

Figure 2.14: Reflection Vector

$$\text{R}(\theta) = R_0 + (1 - R_0)(1 - cos(\theta))^5$$
$$\text{R}_0 = \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2$$

$\text{R}(\theta)$ – probability for reflection for incident ray angle θ
R_0 – probability for reflection for normal incident ray

Figure 2.15: Schlick's approximation

$$\frac{sin\theta_1}{sin\theta_2} = \frac{n_2}{n_1}$$

Figure 2.16: Snell's law

Different materials change the speed of light, which causes light refraction. Snell's law (Equation 2.16) defines that the ratio of the refractive indices is indirectly proportional to the ratio of the sine of the incident and the refracted angles. On each reflective surface point we redirect the ray in the direction expressed by the law of reflection. When the object is translucent we redirect a second ray through the object in the direction based on Snell's law (Algorithm 3). The resulting color is a linear combination $(1 - \alpha) \cdot refraction + \alpha \cdot reflection)$ where α is calculated with schlick's approximation (Equation 2.15). As we split the ray on translucent objects the runtime for this ray increases exponentially by $O(2^{depth})$. It would be possible that the $depth$ is infinite for infinite bouncing light. We limit the recursion depth by the number of reflections of reflections we want to render. It is possible to add other factors to stop ray tracing earlier, for example when the change of color is not noticeable anymore.

```
raytrace(ray,depth,pixelColor)
Find closest ray-object intersection parameter t;
if t <= ε then
    return;
end
if depth > maxDepth then
    return;
end
ray.origin = ray.origin + ray.direction · t;
if object.material.type = diffuse then
    See algorithm 4;
end
if object.material.type = reflective then
    See algorithm 5;
end
if object.material.type = refractive then
    See algorithm 6;
end
```

Algorithm 3: Recursive ray tracing

```
raytrace(ray,depth,pixelColor)
if object.material.type = diffuse then
    pixelColor += object.getDiffuseShading();
    return;
end
```

Algorithm 4: Handling diffuse materials

```
raytrace(ray,depth,pixelColor)
if object.material.type = reflective then
    N⃗ = object.normal;
    V⃗ = ray.direction;
    ray.direction = 2(V⃗ · N⃗) · N⃗ − V⃗;
    Color tmp;
    tmp = raytrace(ray,depth + 1,pixelColor);
    pixelColor += tmp;
    return;
end
```

Algorithm 5: Reflection

```
raytrace(ray,depth,pixelColor)
if object.material.type = refractive then
    N = object.normal;                                                    end
    V = ray.direction;
    n = object.indexOfRefraction;
    R0 =1 - n 1/(1+n);
    R0 = (R0)^2 if N · V > 0then
        Inside of the object;
        N = (−1) · N;
        n = 1/n;
    end
    n = 1/n;
    cos(θ1) = −(N · V) cos(θ2) = 1 − n^2(1 − cos(θ1)^2);
    if cos(θ2) < 0 then
        Total internal reflection
        return;
    end
    Schlick's approximation
    Rprob = R0 + (1 − R0) · (1 − cos(θ1))^5;
    ray.direction = (V ·n + (N(n · cos(θ1) − sqrt(cos(θ2))))).normalize()
    Color tmp;
    tmp = raytrace(ray, depth + 1, pixelColor);
    pixelColor += tmp ·Rprob
    return pixelColor;

```

Algorithm 6: Refraction

CHAPTER 3

Interactive Techniques for Ray Tracing

When rendering with ray tracing, most time will be spent on intersection tests. This is also the case for small scenes with a dozen objects. The intersection routines are one of our frame rate limiting factors when the polygon counts increase. To solve sphere-ray intersections we need to solve a square root. This can become quite computation heavy and even for a low amount of polygons/objects a lot of time will be spent on intersection tests. Naive intersection tests mean one test for every object on every pixel but only one object will be visible per pixel. To speed up ray tracing we want to reduce number of objects checked on each pixel.

3.1 Acceleration Structures

The best case would be to associate each pixel with the correct object and hit point. Before our intersection tests we cannot know which object will be visible. After each intersection test we do not need to re-render the scene until we change it, which could invalidate the pixel-object relation.

With space partitioning we group objects to ensure that when a partition was missed all objects inside are missed too. There are different partitioning structures. Kd-trees ([Mic08, SCI06]) are widely used, but to rebuild them tends to be slow which makes them better for static than for dynamic scenes. Another possibility is to enclose objects in a bounding volume. All volumes are then structured hierarchically. The simplest bounding volume is a sphere centered on half the longest axis with a radius equal to half of that axis. The bounding volume needs to enclose the object as tight as possible to decrease the number of false-positive intersections. It is easy to understand that a sphere with a volume equal to $\frac{4}{3}r^3\pi$ cannot enclose a triangle tightly as the triangle volume is zero and defines a lower bound for r with $r > \frac{c}{2}$, where c is the longest edge. For ray tracing we are not interested in the volume of objects but in the area of the projection onto the image plane. The projected area defines the chance of a ray to hit the volume. The projection of a sphere is independent of the camera view always a circle or an ellipsoid, while the projection of a triangle can be either a triangle or it collapses to a line when

looked at exactly from the side. In the latter case our sphere would create almost entirely false hits.

The goal is to minimize the difference between the area of the projection of the bounding volume and the object itself. For the edge case where the triangle lies on any of the planes xy,xz,yz we can always create a box with the length of the third axis of 0. This means we have the best possible fit for at least one dimension. We can enclose any finite object with a box representated by 2 points as long as the box is axis aligned. We used AABB as bounding volume as they are easy to built and update which is important for dynamic scenes. Oriented bounding boxes (OOB) allow a tighter fit ([Jef03]) but they take more time to built and adjust so AABB might be the better choice when you have to deal with deformable objects ([Dep98]).

The implementation of a bounding volume acceleration structure is mainly defined by two algorithms. The initialization algorithm (Algorithm 7) creating the bounding volume hierarchy and the intersection algorithm (Algorithm 8). We have not optimized the AABB-tree rebuilding in any way and simply rebuild the whole tree on object movement.

Our Bounding Volume Hierarchy (BVH) is a binary tree which contains our objects only in leaf-nodes (Figure 3.1). We build the tree by splitting the longest axis for each node in half and inserting objects which lie on this axis on the left or right into the corresponding child nodes. We traverse the tree by intersecting each node with the ray and return a miss as soon as we miss our Bounding Volume or the innermost child objects (Figure 3.2).

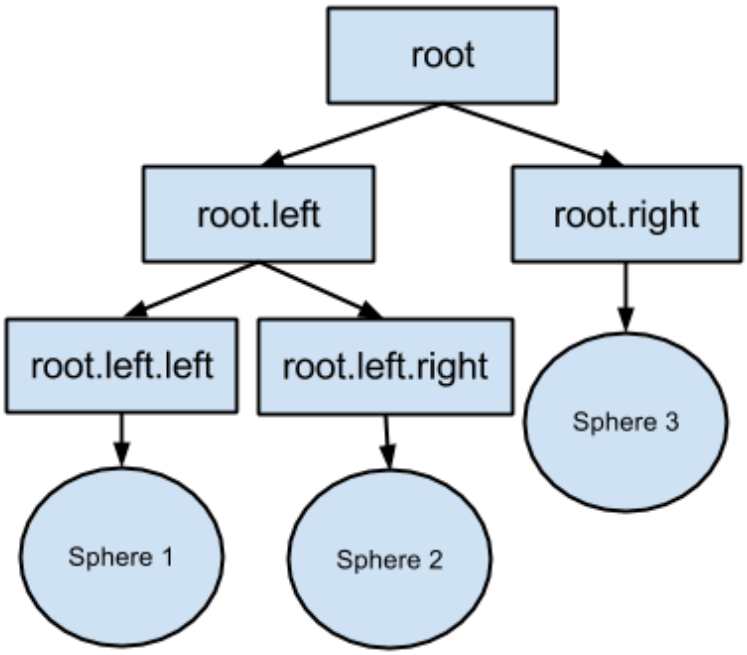

Figure 3.1: Example AABB-tree. Rectangles are AABB Nodes. Circles represent scene objects

We still have to deal with the teapot in a stadium problem where a small high polygon model is placed inside of a large low polygon model. The intersection test of the large object will most

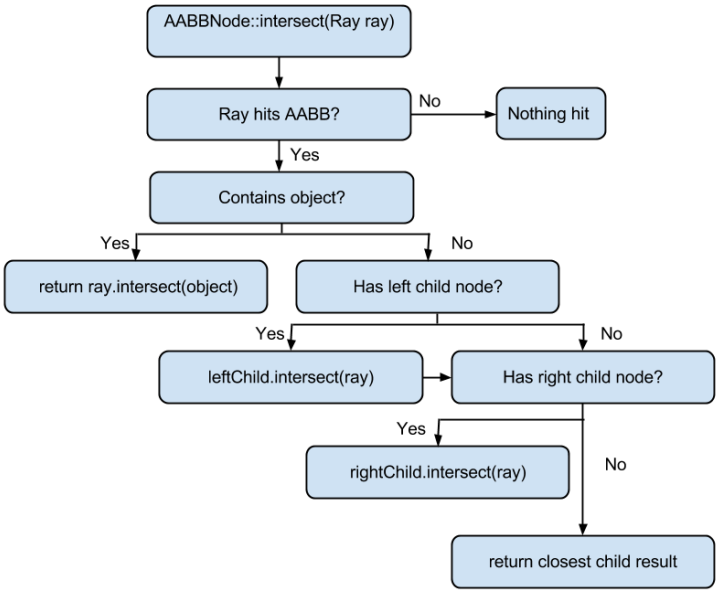

Figure 3.2: Structure of the AABB intersection algorithm

likely always hit but the small teapot will mostly be missed. Another problem is the handling of infinite objects like planes. Each AABB is defined by a minimum and a maximum point but for infinite objects we cannot enclose them. We suggest to split large objects into smaller ones which will likely improve the balance of the BVH. A room made of 6 planes can be replaced by 6 rectangles or 12 triangles, which can be inserted into an AABB-tree.

As we chose axis aligned bounding boxes we can define each box by 6 slabs and check for each pair of slabs if for our intersection parameter $t_{near} < t_{far}$ holds true ([G. 98]). The value t_{near}, t_{far} are the t values where our ray hit's the near,far slab. This test can be improved by pre-calculating the sign of the inverse of the direction of the ray as proposed by Amy Williams et al. [Uni05]. We use their proposed pre-calculation (Algorithm 8) and follow this idea through our whole ray tracing algorithm. By pre-calculating as much as possible and thus reducing the allocation of memory and calculation the process of ray tracing can be sped up. Everytime our ray changes its direction we pre-calculate the inverse of the ray and also the sign of the inverse direction for each dimension.

When our AABB-tree is balanced we can reduce the number intersection tests per pixel on the first depth from $O(N)$ down to $log(N)$. The balance of the tree depends on how well our splitting heuristic for left/right child nodes represents the scene. It is important to enclose each group of objects as tight as possible and algorithms to do this efficiently and stable for different and dynamic scenes are still in development.

```
Result: AAAB tree initialized
if objects->size() == 1 then
    this->object = objects[0];
    return;
else
    Vec3 dist = max - min;
    int axis = dist.longestAxis;
    float limit = (min[axis] + max[axis]) / 2.0;
    Vec3 leftMin = {Float.MAX,Float.MAX,Float.MAX};
    Vec3 leftMax = {Float.MIN,Float.MIN,Float.MIN};
    Vec3 rightMin = {Float.MAX,Float.MAX,Float.MAX};
    Vec3 rightMax = {Float.MIN,Float.MIN,Float.MIN};
    foreach Object object do
        float center = obj->center[axis];
        if center <= limit then
            leftChilds.push_back(obj);
            leftMin = leftMin.min(obj->min);
            leftMax = leftMax.max(obj->max);
        else
            rightChilds.push_back(obj);
            rightMin = rightMin.min(obj->min);
            rightMax = rightMax.max(obj->max);
        end
    end
    float ratio = (float) left.size() / (float) right.size();
    if leftChilds.empty() || rightChilds.empty() then
        Split objects half/half based on any criterion
    end
    leftAAABNode = new AABBNode(leftChilds, leftMin, leftMax);
    rightAAABNode = new AABBNode(rightChilds, rightMin, rightMax);
end
```

Algorithm 7: Initialization of AAAB tree

Result: Return hit object and ray parameter t

```
1  float tmin,tmax,tymin,tymax,tzmin,tzmax;
2  tmin = ((ray.sx ? min.x:max.x) - ray.o.x) · ray.invd.x;
3  tmax = ((ray.sx ? max.x:min.x) - ray.o.x) · ray.invd.x;
4  tymin = ((ray.sy? min.y:max.y) - ray.o.y) · ray.invd.y;
5  tymax = ((ray.sy? max.y:min.y) - ray.o.y) · ray.invd.y;
6  if (tmin > tymax) || (tymin > tmax) then
7  |  No object hit return 0;
8  end
9  if tymin > tmin then
10 |  tmin = tymin;
11 end
12 if tymax < tmax then
13 |  tmax = tymax;
14 end
15 tzmin = ((param.ray.sz?min.z:max.z) - param.ray.o.z) · param.ray.invd.z;
16 tzmax = ((param.ray.sz?max.z:min.z) - param.ray.o.z) · param.ray.invd.z;
17 if  (tmin > tzmax) || (tzmin > tmax) then
18 |  No object hit return 0;
19 end
20 if tzmin > tmin then
21 |  tmin = tzmin;
22 end
23 if tzmax < tmax then
24 |  tmax = tzmax;
25 end
26 if tmin > tmax then
27 |  No object hit return 0;
28 end
29 tmax = FMAX;
30 return Algorithm 9.
```

Algorithm 8: Improved AABB-Ray intersection test

Result: Recursive result of child nodes

```
1  if isLeafNode() then
2  |   tmax = object->intersect(param.ray);
3  |   if tmax > eps then
4  |   |   param.object = object;
5  |   |   return tmax;
6  |   end
7  |   return 0;
8  end
9  if hasLeftChild() then
10 |   Object * leftChildObject = param.object;
11 |   tmin = leftChildNode->intersect(param);
12 |   if tmin < tmax  tmin > eps then
13 |   |   tmax = tmin;
14 |   else
15 |   |   param.object = leftChildObject;
16 |   end
17 end
18 if hasRightChild() then
19 |   Object * rightChildObject = param.object;
20 |   tmin = rightChildNode->intersect(param);
21 |   if tmin < tmax  tmin > eps then
22 |   |   tmax = tmin;
23 |   else
24 |   |   param.object = rightChildObject;
25 |   end
26 end
27 return tmax == FMAX ? 0 : tmax;
```

Algorithm 9: AABBNode check childs

3.2 Adaptive Rendering

When the artist is done with setting up the appropriate materials for the objects, most of the remaining interactions are move actions. As we know we have no indirect diffuse illumination in ray tracing we can skip diffuse pixel not affected by the movement. We have to re-render any pixel which showed a translucent or reflective object. We cannot know whether we would lose indirect light paths when we skip those pixels.

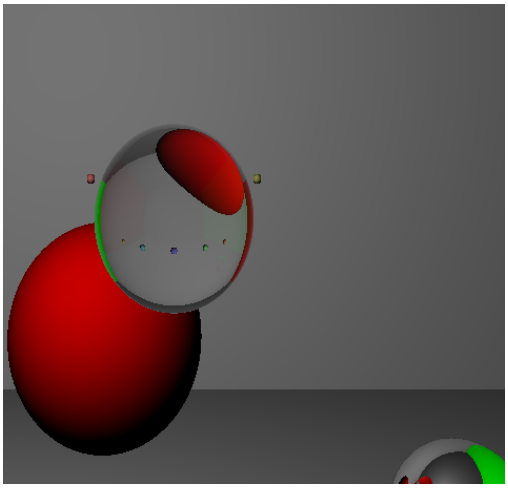

Figure 3.3: Before movement

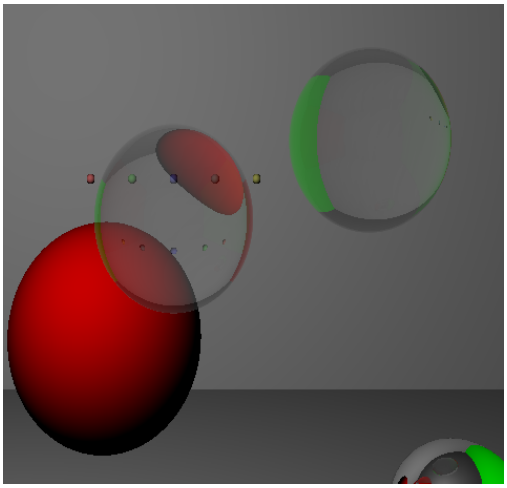

Figure 3.4: Before and after movement

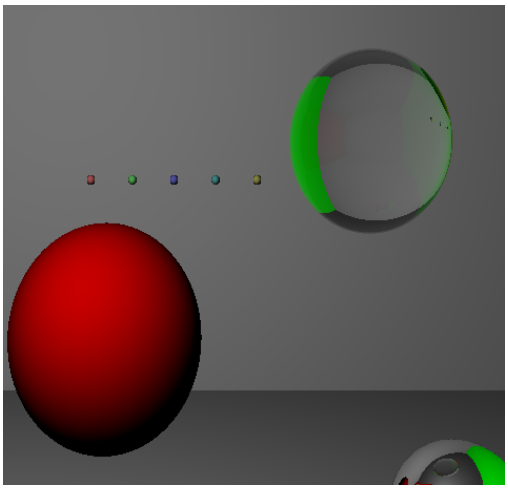

Figure 3.5: After movement

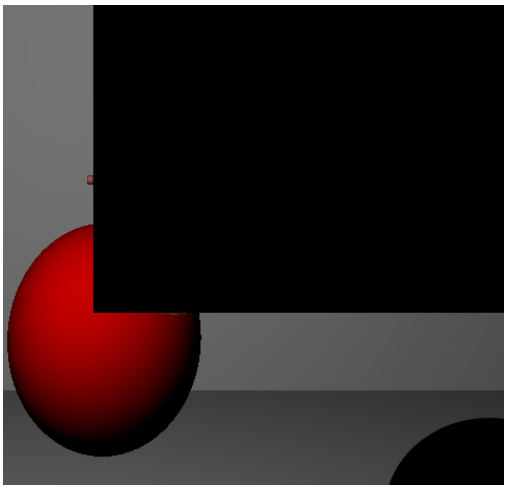

Figure 3.6: Masked area

We already use a deep texture to store the object IDs for selection and object highlighting on selection. Only masked areas as in Figure 3.6 need to be rerendered. The most expensive pixels are still reflective and translucent pixels, which we have to render in any case. When the

object moves we can skip any diffuse pixels on the opposite side of the image. This holds true as long as we do not render shadows, where other objects could occlude the light source shining on diffuse areas.

When using path tracing, we can observe noise on the unconverged image due to the stochastic sampling and integration process. When using ray tracing, we do not have any noise on the rendered images, and we also get a convenient, interactive output during interaction where we want more frames for a smooth movement of objects. If we instead use 2-pass rendering where we only render every second pixel our rendering needs roughly the same amount besides rendering the texture twice onto our fullscreen quad, which is by far faster than our ray tracing. The time for both halves is the same as for the original image but for interaction lasting over several frames we double the frames with the drawback of introducing noise. In each renderpass we alternate the pixels we render. Two render passes after our last interaction we again have a fully rendered and noise less image.

Another similar way would be to undersample. This means we render only half the resolution and stretch it back onto our fullscreen quad. Obviously this speeds up interaction a lot, but also can generate quite strong artifacts.

No matter how fast our ray-AABB intersection code in the end is, when we only are interested in the position of our object we could also skip the object intersection and render the object we are moving as a box. This always saves at least one intersection test for a sphere and $log(N)$ for an object made of N triangles which are stored in a balanced bounding volume hierarchy themselves.

For the bounding volume hierarchy we rebuild our AABB-tree bottom down. With objects made of million of triangles this is a waste of time. The initialization of the AABB-tree can be either top-bottom or bottom up. The rebuild usually can be bottom-up as no AABB can leave any node higher in the hierarchy without leaving the AABB nodes in between.

As we know memory allocation is costly and the number of Ray and Vector3 objects we need from frame to frame is roughly stable. We have not benchmarked this but the preallocation of objects is a widely used technique which should for millions of total allocated rays and vector objects during one render pass give quite some amount of speed. All you do of course is to trade in memory for speed. When you allocate more memory than the RAM of your machine has this will lead to page swapping slowing down your program a lot.

3.3 Deep Textures

We speak of deep textures as textures which hold more information than the pixel color. We used deep textures for the object ids to post process the image for object selection. With sufficient memory and low recursive depth it is possible to store all hit points per pixel to add shadows without re-rendering the whole image. This combines the best of both worlds. You can edit your scenes as fast as possible without shadows and still turn them on without re-rendering anything but the shadow rays themselves.

CHAPTER 4

Results

4.1 Fast code and algorithms

This section focuses on c++ as widely used language for ray tracers, although ray tracer do exist in different languages (Java [Aly97], WebGL [Eva10]). If you are not using c++ you might want to skip to section 4.2.

Fast ray tracing is mostly a matter of efficient structures and algorithms. The scene editing tool we developed as reference was developed in c++ and compiled with 'MinGW 4.7 g++' in mode '-O3 -s'. It is not easy to compare absolute render times as the difference between a popular 3,4GHz quad core and a 2,2GHz dual core had been between 1,5x up to 2x the time per frame. Although clock rate, cache size are important, the number of concurrent threads has the biggest impact. It is easy to see why GPU based ray tracing is the future as graphic cards are capable of higher degrees of paralellism.

The timings and percentages need to be read in order of appearance to be meaningful. The reference image for all improvements in this section is Figure 4.1. The measured timings represent average render times.

Note that just because we suggest specific code styles for a part of the program, this does not mean we suggest dropping best practices or object orientated programming paradigms anywhere else. Code styles can be very compiler dependant and it is never suggested to rework any code without profiling the bottlenecks. Proposed code styles should be compiler independently at least as fast as their counterparts.

Get by reference vs direct access

One of the most accessed variables in a ray tracing application are the rays member variables like origin, destination and all precalculations stored there. It is basic programming knowledge why not to access member variables by value, but we have benchmarked it anyhow (Table 4.1). The difference between direct access and get by reference are still noticable 217ms. For classes where variables are accessed billions of times per frame this is recommended.

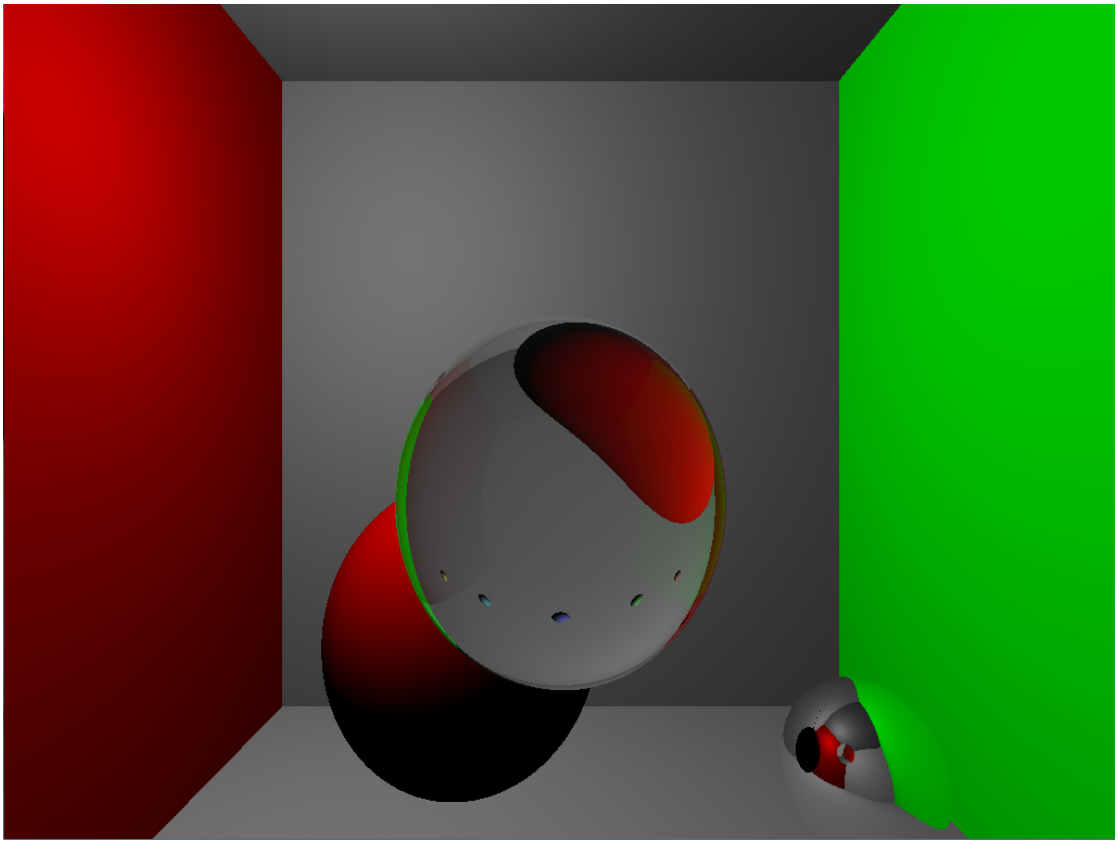

Figure 4.1: 1024x768, Reference image, 8 Spheres, 6 planes

Style	Calls	% of processing time	Total time
Vec getOrigin Vec getDirection	958017203 603370408	11.62% 5.98%	4.3s
Vec& getOrigin Vec& getDirection	958017203 603370408		2.35s
ray.origin ray.direction			2.133s

Table 4.1: Get by reference vs direct access

Array vs three variables for Vec3

In every 3D related programm a representation of points is needed. We evaluated the difference between storing the doubles or floats for x,y and z coordinate an array as double[3] or as double x,y,z. The difference of 450ms per frame is huge.

Style	Total time
double[3]	2.11s
double x,y,z	1.66s

Table 4.2: Array vs single variables

Pass by value vs pass by reference

Although this is a bsaic coding principle, we benchmarked the difference for passing our ray as value or as reference. The difference are noticable 350ms per frame. Our AABBNode::intersect(Ray,Object *) method is with 5782942 calls reliable for 34% of our total render time. As it is unlikely that the used algorithms will be improved code wise by a high degree, it is necessary to look for different ways to do intersection tests. One possibility would be to use Ray-Slobe tests for the AABB intersection tests ([Com08]).

Style	Total time
intersect(Ray,Object *&)	1.6s
intersect(Ray&,Object *&)	1.25s

Table 4.3: Pass by value vs pass by reference

Multiplication vs Division

As our AABBNode::intersect accounts for about one third of the rendering time we measured the difference of two divisions (Algorithm 10) vs one division and two multiplications (Algorithm 11) for the parameter of our ray-box algorithm.

```
1 double ty1 = (min.y - ray.origin.y)/ray.direction.y;
2 double ty2 = (max.y - ray.origin.y)/ray.direction.y;
```

Algorithm 10: Two divisions, Total time:1.25s

```
1 double inv_ray_d_y = 1/ray.direction.y;
2 double ty1 = (min.y - ray.origin.y)· inv_ray_d_y;
3 double ty2 = (min.y - ray.origin.y)· inv_ray_d_y;
```

Algorithm 11: 1 Division, 2 Multiplications, Total Time:1.19s

Double vs float precision

It takes more time to process double precision variables and the difference between float and double for our Vec3 class which is the most used container class in our ray tracer was 80ms. It is important to state that you also loose half of the precision which can increase the arithmetic problems already occuring with floating point arithmetic ([Fut13]). The solution in the future

might be the use of integer based ray tracing as proposed with the tangere ray tracer ([Int13]) when it can handle difficult surfaces better.

Style	Total time
double x,y,z	1.14s
float x,y,z	1.06s

Table 4.4: Double vs float

AABB Intersection routine

Every calculation which can be moved out of recursion should be. The improved AABB-Ray intersection code from Amy Williams et al ([Uni05]) improved the speed per frame by 750ms in our ray tracer.

Style	Total time
G. Scott Owen ([G. 98])	1.06s
Amy Williams et al. ([Uni05])	250ms

Table 4.5: Impact of efficient intersection routines

Efficient Ray and Vec3

During recursive ray tracing millions of Ray and Vec3 objects are created. This needs to be as fast as possible. In C++ for a class to be handled as Plain Old Data (POD) it may not have constructors and destructors or only trivial ones. Additionally the class may itself only contain POD member variables. To intialize the ray by passing references instead of values and converting the Vec3 class to a POD improved the speed by a total of 100ms. As we have removed the constructor the code can become unhandy and the Vec3 class will be initialized in old c style like 'Vec3 vec = {0.0f,0.0f,0.0f}'. The Vec3 destructor has no side effects, why we declare it trivial '~Vec3() = default;'. This also ensures that there will be no pointer for the destructor in the virtual table of the class.

Style	Performance gain
Ray(Vec3& origin,Vec3& direction)	80ms
~Vec3() = default;	10ms
No constructor for Vec3	10ms

Table 4.6: Impact of efficient ray initialization and Vec3 as POD

Final render time

As a last and final step we also allow the g++ compiler to drop strict standard compliance with the '-Ofast' flag which turns on '-ffast-math'. Our reference image finally renders in 110ms. This is a level where we can really speak of interactive response. We have not reached the desired 25 frames per second mark any real time application aims for, but there are still a lot of improvements to implement and we have not even replaced the 6 planes by rectangles. As we check the 6 planes naively, they account for 22% of our whole render time.

4.2 Disable Features during interaction

The speed of an interactive ray tracer is extremely important. Compared with offline rendering your ray tracing result might be as good as possible but no one will use it when it is annoying to use. All of the following images have, if not otherwise declared, been rendered with a ray tracing depth of 3, a resolution of 1280x720 and no super sampling. Most features described here cannot be disabled for games as the incohorent image, artifacts and popping effects would not be accepted by gamers. These ways of speed up are interesting for editing or visualization based tools where it is not a problem to change the image during interaction.

Shadows

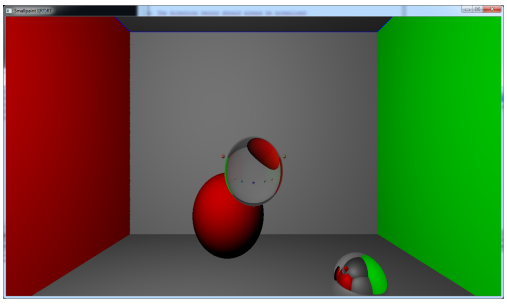

Figure 4.2: No shadow rays: 90ms

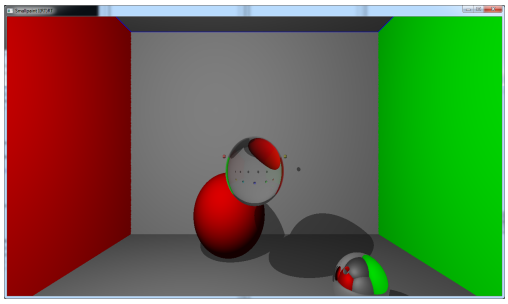

Figure 4.3: 1 shadow ray: 234ms

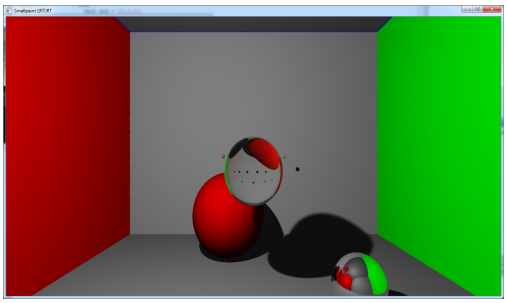

Figure 4.4: 27 shadow rays: 1.4s

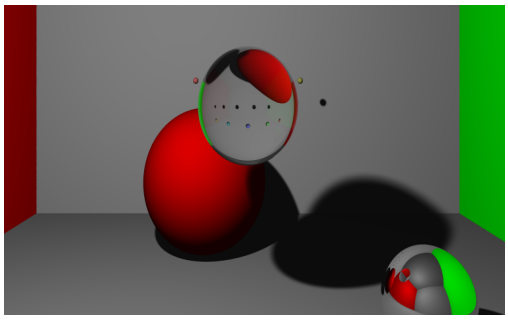

Figure 4.5: 64 shadow rays: 5.2s

The number of shadow rays has an important impact on the renderrtime. The visual difference between no and at least one shadow ray is important to help to understand the three dimensional scene. Most of the time shadows are unimportant during movement. When the user drags an object for 750ms we most of the time do not need shadows. This is one of the drawbacks of ray tracing compared to path tracing. In unbiased path tracing your image automatically improves by rendering and merging the multiple images. In ray tracing we have to decide most of the time prior to our rendering how the final result should look like and we will not get a better result than that even when the user does not change anything in the scene.

Refractive and reflective objects

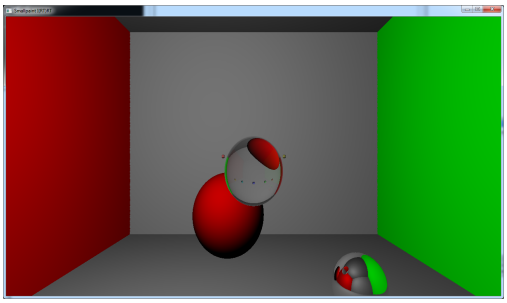

Figure 4.6: Refractive object covers only small part: 90ms

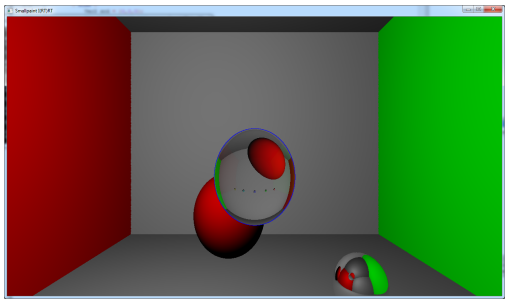

Figure 4.7: Refractive object moved a bit closer: 120ms

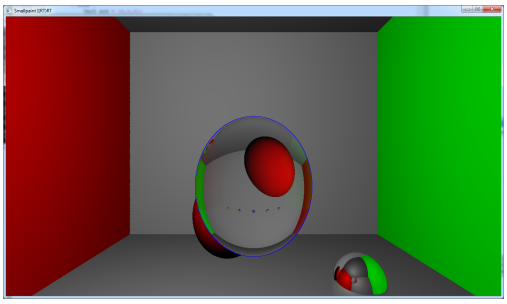

Figure 4.8: Refractive object moved closer: 140ms

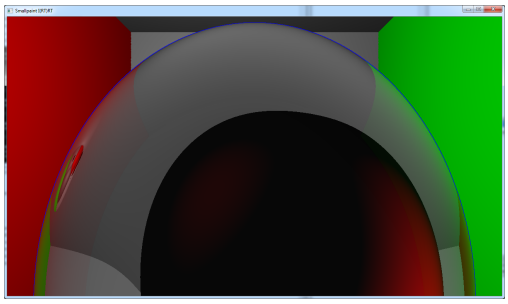

Figure 4.9: Refractive object moved very close: 552ms

Refractive objects increase the amount of rays exponentially. The rendertime correlates to the number of pixels covered by them. As we know that these pixels increase the amount of time needed per frame more than the other types of objects we have covered until now we need to disable their rerendering during interaction to keep up with full speed. When the refractive object is moved we still need the feedback of its position but often not the refraction. We therefore can treat refractive objects during interaction as diffuse objects and render the refraction when the movement has stopped. This is not possible when you want to place for example a lense over a book to make use of the refraction. In this case you will need render the refraction at least for the

currently selected object. It still might be possible to treat all other refractive objects as diffuse. Another option is to reduce the recursion depth down to 2. This will allow us to see diffuse surfaces through our the refractive object but the object itself will not be rendered in reflections. Reflective objects can be treated similar although they are less of a problem.

Rendering depth

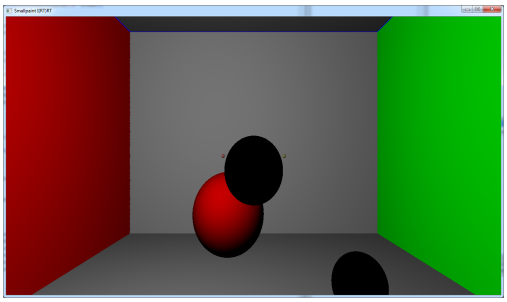

Figure 4.10: Recursion depth 0: 72ms

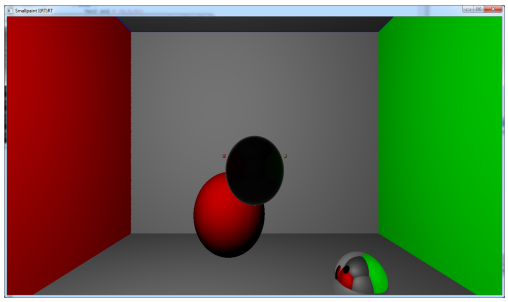

Figure 4.11: Recursion depth 1: 81ms

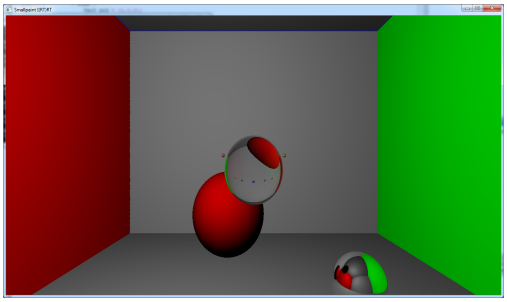

Figure 4.12: Recursion depth 2: 86ms

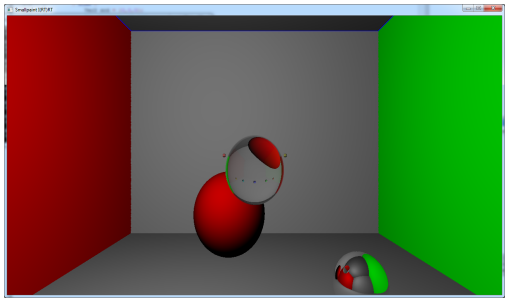

Figure 4.13: Recursion depth 44: 91ms

The recursion depth is important for refraction and reflection heavy scenes. Combined with disabling the refraction and reflection rendering you can get a lot of speed during object movement.

Supersampling

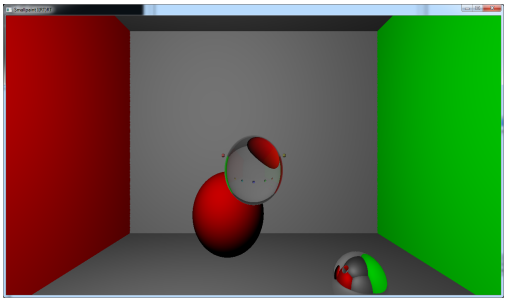

Figure 4.14: No supersampling: 90ms

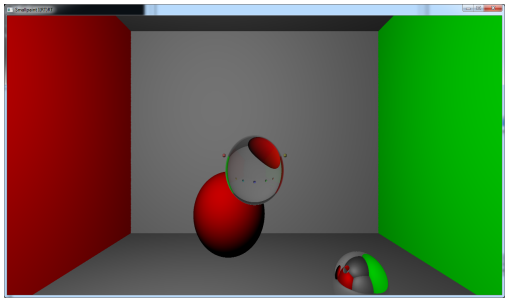

Figure 4.15: 2x supersampling: 183ms

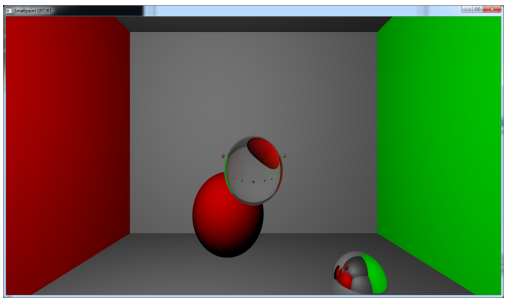

Figure 4.16: 4x supersampling: 383ms

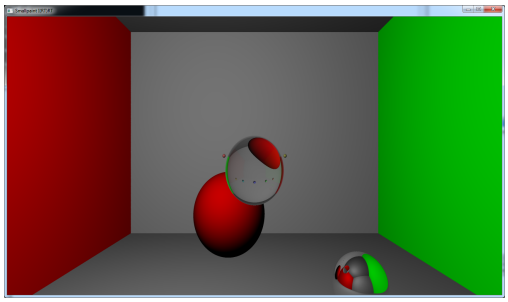

Figure 4.17: 8x supersampling: 729ms

Supersampling is very computation heavy, as it means sampling pixels multiple times. Naive supersampling can therefore be understand as a multiplicator of the rendering time. As supersampling does not improve the quality of coherent image areas it is very important to do adaptive supersampling for areas where it matters. The main benefit of supersampling for interactive ray tracing is that this feature can be rendered after the initial image was rendered. 729ms for our 8x supersampled image would therefore mean we still have 90ms during interaction and 639ms after interaction has stopped we get a 8x supersampled image even with naive supersampling.

CHAPTER 5

Conclusion

Interactive ray and path tracing is possible as this and previous work shows. As graphic cards are currently developing faster and are able to support higher levels of parallelism we can expect to achieve best results with GPU-based results and in the future with ray tracing hardware.

Code style is important, but it should not be considered as the solution for low frame rates - it should be seen as essential ingredient to writing fast code. The main goal is to have faster ray and path tracing algorithms with the same image features as in offline-rendering, but in real time. We have presented the different impact of image features and the scene structure and space partition including intersection algorithms, which are the main part to improve. The reason is, that we spent most of our time doing nothing but intersection tests for a lot of rays. Unbiased path tracing compared to ray tracing has the additional problem of stochastic noise but with the benefits of progressively improving images and additional renderable image features like indirect diffuse illumination. For every feature like shadows we have to find out how little is sufficient to create a realistic result.

Although there are currently astonishing results achieved, when the goal is to get offline-rendering quality and difficult image paths combined with complex lighting, models and materials there is still a long way to go. The most promising way to improve ray and path tracing is to find ways to approximate image features in low cost with a low error and to reduce the number of rays and intersections necessary.

Bibliography

[Alp] Wolfram Alpha. Fresnel equations. http://scienceworld.wolfram.com/physics/FresnelEquations.html. [Online; accessed 24-August-2013].

[Aly97] Alyosha Efros. Ray tracing with java. http://www.cs.cmu.edu/~efros/java/tracer/tracer.html, 1997. [Online; accessed 27-August-2013].

[Att12] Attila T. Áfra, editor. *Interactive Ray Tracing of Large Models Using Voxel Hierarchies*, volume 0. Budapest University of Technology and Economics, Hungary;Babe¸s-Bolyai University, Cluj-Napoca, Romania, 2012. [Online; accessed 21-August-2013].

[Com08] Computer Graphics Lab, TU Braunschweig; Max-Planck-Institut für Informatik, Saarbrücken. *Fast Ray/Axis-Aligned Bounding Box Overlap Tests using Ray Slopes*, 2008. [Online; accessed 27-August-2013].

[Dep94] Department of Computer Science, Columbia University. *Generalization of the Lambertian Model and Implications for Machine Vision*, 1994. [Online; accessed 23-August-2013].

[Dep98] Department of Mathematics and Computing Science, Eindhoven University of Technology. *Efcient Collision Detection of Complex Deformable Models using AABB Trees*, 1998. [Online; accessed 25-August-2013].

[Eri12] Eric Tabellion, editor. *Point-Based Global Illumination Directional Importance Mapping*. DreamWorks Animation LLC., 2012. [Online; accessed 21-August-2013].

[Eva10] Evan Wallace. Path tracing with we. http://madebyevan.com/webgl-path-tracing/, 2010. [Online; accessed 27-August-2013].

[Fut13] *Ray Tracing is the Future and Ever Will Be*, 2013. [Online; accessed 27-August-2013].

[G. 98] G. Scott Owen. Ray - box intersection. http://www.siggraph.org/education/materials/HyperGraph/raytrace/rtinter3.htm, 1998. [Online; accessed 27-August-2013].

[Gen84] Gene S. Miller, C. Robert Hoffman. Real pixels. http://web.archive.org/web/20000830075557/http://www.cs.berkeley.edu/~debevec/ReflectionMapping/illumap.pdf, 1984. [Online; accessed 24-August-2013].

[Gra] Graphics research group intel. Wolfenstein ray traced. http://wolfrt.de/. [Online; accessed 21-August-2013].

[Gre92] Greg Ward. Real pixels. http://www.contrib.andrew.cmu.edu/~yihuang/radiance_pic/Real%20Pixels%20-%20Greg%20Ward.pdf, 1992. [Online; accessed 24-August-2013].

[Int13] *Integer Ray Tracing*, 2013. [Online; accessed 27-August-2013].

[Jü05] Bert Jüttler, editor. *Perceptual Effects in Real-Time Tone Mapping*. MPI Informatik, Saarbrücken, Germany, ACM, 2005. [Online; accessed 23-August-2013].

[Jaca] Jacco Bikker. Arauna, real-time ray tracing. http://igad.nhtv.nl/~bikker/. [Online; accessed 21-August-2013].

[Jacb] Jacco Bikker. Brigade, real-time path tracing. http://igad.nhtv.nl/~bikker/. [Online; accessed 21-August-2013].

[Jacc] Jacco Bikker. Ray tracing in real-time games. https://sites.google.com/site/raytracingcourse/. [Online; accessed 21-August-2013].

[Jef03] Jeff Lander. When two hearts collide: Axis-aligned bounding boxes. http://www.gamasutra.com/view/feature/131833/when_two_hearts_collide_.php, 2003. [Online; accessed 25-August-2013].

[Ká] Károly Zsolnai. smallpaint. http://cg.iit.bme.hu/~zsolnai/gfx/smallpaint. [Online; accessed 21-August-2013].

[Lab00] Laboratory for Computer Science, Massachusetts Institute of Technology. *Interactive Tone Mapping*, 2000. [Online; accessed 23-August-2013].

[Mic08] Microsoft Research Asia;Tsinghua University;Zhejiang University. *Real-Time KD-Tree Construction on Graphics Hardware*, 2008. [Online; accessed 25-August-2013].

[Mik12] Mike Seymor. The art of rendering (updated). http://www.fxguide.com/featured/the-art-of-rendering/, April 2012. [Online; accessed 21-August-2013].

[Pixa] Pixar. http://renderman.pixar.com/view/dp23827. [Online; accessed 21-August-2013].

[Pixb] Pixar. Renderman relighting. http://renderman.pixar.com/view/relighting. [Online; accessed 21-August-2013].

[Pixc] Pixar. Renderman rerendering. http://renderman.pixar.com/view/rerendering. [Online; accessed 21-August-2013].

[Pro96] Program of Computer Graphics, Cornell University. *A Model of Visual Adaptation for Realistic Image Synthesis*, 1996. [Online; accessed 24-August-2013].

[SCI06] SCI Institute, University Utah; Czech Technical University Prague. *On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N)*, 2006. [Online; accessed 25-August-2013].

[Sim] Simon Wallner. Kocmoc - a small handwritten opengl 3.2 core demo. `https://github.com/SimonWallner/kocmoc-demo/`. [Online; accessed 21-August-2013].

[Sol] SolidAngle. http://www.solidangle.com/info.html. [Online; accessed 21-August-2013].

[Tom] Tom Verhoeve et. al. It's about time. `http://igad.nhtv.nl/~bikker/`. [Online; accessed 21-August-2013].

[Uni05] University of Utah. *An Efficient and Robust Ray–Box Intersection Algorithm*, 2005. [Online; accessed 21-August-2013].

[W. 75] W. Newmann, editor. *Illumination for Computer Generated Pictures*. Graphics and Image Processing, University of Utah, 1975. [Online; accessed 24-August-2013].

[Wen12] Wenzel Jakob, Steve Marschner, editor. *Manifold Exploration: A Markov Chain Monte Carlo Technique for Rendering Scenes with Difcult Specular Transport*. Cornell University, 2012. [Online; accessed 21-August-2013].