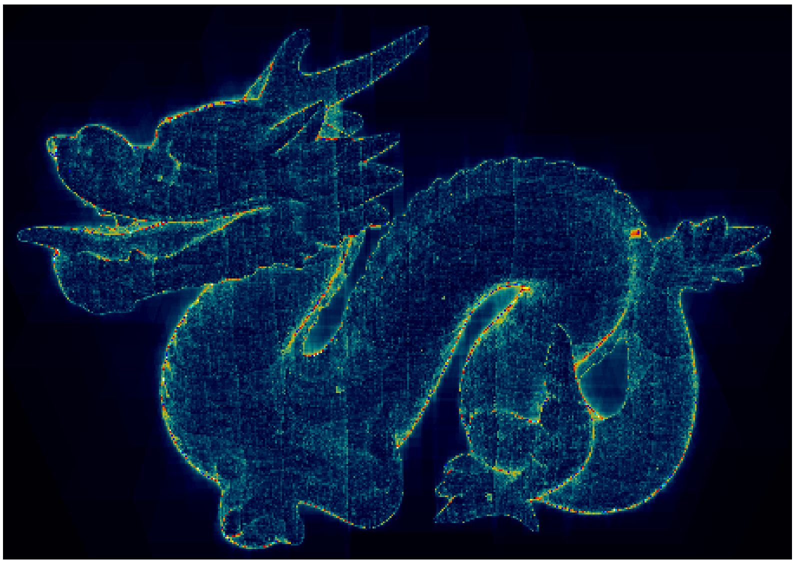
FACULTY OF ENGINEERING

Computer Graphics Project Report



Moritz Wolter
May 24, 2016

Contents

Milestone 1

Essentials

1.1 Geometry

1.1.1 The triangle primitive

This section focuses on the basic math of ray tracing. In a ray tracer so called primary rays are shot from a camera onto a scene. Irreducible objects or geometric primitives are defined. These make up a scene. In order to compute an image from this given scene, the ray-object intersection point closest to the camera has to be found. Various primitives have been implemented the most important one is the triangle. As triangle meshes can be used, to approximate more complex shapes. A triangle is defined by the three points $\mathbf{a}, \mathbf{b}, \mathbf{c}$, that make up its edges. If the edges are ordered counterclockwise the triagle normal can be computed using the formula: [1]

$$\mathbf{n} = (\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a}) / \|(\mathbf{b} - \mathbf{a}) \times ((\mathbf{c} - \mathbf{a})\|. \tag{1.1}$$

To find triangle intersection points barycentric coordinates (α, β, γ) are used. These coordinates allow to describe any point in the plane of the triangle as:

$$\mathbf{p}(\alpha, \beta, \gamma) = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}, \tag{1.2}$$

$$\text{with } \alpha + \beta + \gamma = 1. \tag{1.3}$$

If $\mathbf{p}$ is inside of the plane the inequalities,

$$0 < \alpha < 1, 0 < \beta < 1, 0 < \gamma < 1, \tag{1.4}$$

are satisfied. By substituting $\alpha = 1 - \beta - \gamma$ into the equation and setting $\mathbf{p} = \mathbf{r}(t)$ with $\mathbf{r} = \mathbf{o} + t \cdot \mathbf{d}$ for a ray the following equation is obtained:

$$\mathbf{o} + t\mathbf{d} = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a}). \tag{1.5}$$

Which can be brought into the form $\mathbf{A}\mathbf{x} = \mathbf{b}$ with $\mathbf{x} = \begin{pmatrix} \beta & \gamma & t \end{pmatrix}^T$:

$$\begin{pmatrix} a_x - b_x & a_x - c_x & d_x \\ a_y - b_y & a_y - c_y & d_y \\ a_z - b_z & a_z - c_x & d_z \end{pmatrix} \begin{pmatrix} \beta \\ \gamma \\ t \end{pmatrix} = \begin{pmatrix} a_x - o_x \\ a_y - o_y \\ a_z - o_z \end{pmatrix}. \tag{1.6}$$

[1]This section in based on Ray Tracing from the gound up page 362 and onward.

Equation 1.6 could in principle be given to a linear algebra subroutine for solution. If the inequalities 1.4 are satisfied and $t > \epsilon$. Where epsilon is a small positive number used to prevent self intersection. A valid intersection has been found. However this is not the most efficient way to proceed. Always solving the whole equation system is not necessary all the time. Using Cramers rule the a set of equations can be derived and the intersection method can return as soon as one condition is violated. This way unnecessary computations can be avoided. [2]

1.1.2 The plane primitive

Another important primitive is the plane. These objects can be defined by a normal $\mathbf{n}$ which determines the orientation of the plan and $\mathbf{a}$ which determines where it is.[3] For a point $\mathbf{p}$ on the plane, the condition

$$(\mathbf{p} - \mathbf{a})^T\mathbf{n} = 0 \tag{1.7}$$

must hold. This is the same as asking for the cosine of the angle between the vector on the plane and normal to be zero. Or equivalently saying that the vectors $(\mathbf{p} - \mathbf{a})$ and $\mathbf{n}$ must form a 90° angle. To check if a given ray intersects a plane the ray must be plugged into equation 1.7 which yields:[4]

$$(\mathbf{o} + t\mathbf{d} - a)^T\mathbf{n} = 0. \tag{1.8}$$

Solving for t then leads to:

$$t = (\mathbf{a}^T\mathbf{n} - \mathbf{o}^T\mathbf{n}))/(\mathbf{d}^T\mathbf{n}). \tag{1.9}$$

If then $t > \epsilon$ a valid intersection was computed. The exact location of the hit point can be found from the ray equation $\mathbf{p}(t) = \mathbf{o} + t\mathbf{d}$.

The rectangle primitive

A rectangle is simply a subset of all the points contained in a plane therefore the same equations can be used. Additionally the conditions

$$\|p_x\| < 1.0, \|p_y\| < 1.0, \tag{1.10}$$

have to be enforced to turn a the x,y-plane into a rectangle around the origin. This unit plane can later be rotated and scaled using transformations to fit into a desired scene.

The circle primitive

A circle in the x,y plane is obtained by a procedure analogue to the one which lead to rectangles if instead the condition

$$\sqrt{p_x^2 + p_y^2} < 1.0 \tag{1.11}$$

is used. This circle can similarly be transformed to fit into any scene.

[2] See Ray Tracing from the Ground up page 366 for further details.
[3] Ray Tracing from the ground up page 31.
[4] Ray Tracing from the Ground up page 54.

1.1.3 The sphere primitive

A sphere is a set of points located within a given distance r around a specified point **c**. For any point **p** within this sphere the condition:

$$\|\mathbf{p} - \mathbf{c}\| \leq r \tag{1.12}$$

$$\text{or } (\mathbf{p} - \mathbf{c})^T(\mathbf{p} - \mathbf{c}) \leq r^2 \tag{1.13}$$

must hold. For a sphere around the origin with radius one $\mathbf{c} = 0$, $r^2 = 1$ and substituting $\mathbf{p} = \mathbf{o} + t\mathbf{d}$ the expression below is obtained:

$$(\mathbf{o} + t\mathbf{d})^T(\mathbf{o} + t\mathbf{d}) - 1 = 0 \tag{1.14}$$

Now using an equality instead of an inequality to compute the intersection with the outer bound of the sphere. The distributive property of the dot product leads to:

$$t^2\mathbf{d}^T\mathbf{d} + 2t\mathbf{o}^T\mathbf{d} + \mathbf{o}^T\mathbf{o} - 1 = 0. \tag{1.15}$$

The obtained quadratic equation can be solved using standard methods with:

$$a = \mathbf{d}^T\mathbf{d} \tag{1.16}$$

$$b = 2\mathbf{d}^T\mathbf{o} \tag{1.17}$$

$$c = \mathbf{o}^T\mathbf{o} - 1 \tag{1.18}$$

$$\text{then } t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{1.19}$$

It is possible for learn the number of intersections from the discriminant $d = b^2 - 4ac$. If $d < 0$ there will be no intersections, one if $d = 0$ and two if $d > 0$. If two intersections are found the one with the smaller t should be used, as it will be closer to the camera.[5]

1.2 Object Transformations

In order to move, rotate or scale the objects defined earlier four dimensional transformation matrices are defined. Four dimensions are used, as three dimensional translations can be expressed as a matrix operation.
Scaling can be done by using[6]:

$$T_{scale} = \begin{pmatrix} a & & & \\ & b & & \\ & & c & \\ & & & 1 \end{pmatrix} \quad T_{scale}^{-1} = \begin{pmatrix} 1/a & & & \\ & 1/b & & \\ & & 1/c & \\ & & & 1 \end{pmatrix} \tag{1.20}$$

Translation is working with:

$$T_{translate} = \begin{pmatrix} 1 & & & d_x \\ & 1 & & d_y \\ & & 1 & d_z \\ & & & 1 \end{pmatrix} \quad T_{translate}^{-1} = \begin{pmatrix} 1 & & & -d_x \\ & 1 & & -d_y \\ & & 1 & -d_z \\ & & & 1 \end{pmatrix} \tag{1.21}$$

[5]Ray Tracing from the Ground up page 57
[6]Ray Tracing from the ground up page 404

Rotation around the tree axes:

$$T_x = \begin{pmatrix} 1 & & & \\ & \cos(\theta) & -\sin(\theta) & \\ & \sin(\theta) & \cos(\theta) & \\ & & & 1 \end{pmatrix} \tag{1.22}$$

$$T_y = \begin{pmatrix} \cos(\theta) & & \sin(\theta) & \\ & 1 & & \\ -\sin(\theta) & & \cos(\theta) & \\ & & & 1 \end{pmatrix} \tag{1.23}$$

$$T_z = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & & \\ \sin(\theta) & \cos(\theta) & & \\ & & 1 & \\ & & & 1 \end{pmatrix} \tag{1.24}$$

$$\tag{1.25}$$

The inverse rotations can be obtained by using $-\theta$ in place of θ or by transposing the matrices. Several transformations can be combined by matrix multiplication.
Instead of transforming objects it is common practice in ray tracing to transform rays. Prior to intersection testing a ray ($\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$) is multiplied with the inverse transformation matrix. The obtained transformed ray is then intersected with the unchanged object. The hit point is found and transformed back into the original space using the transformation matrix. [7] For normals the process is slightly more complicated. [8] In the unchanged space the normal must be orthogonal to infinitely small vectors along the surface $\mathbf{n}^T\mathbf{m} = 0$. The same must be true for the transformed space $\mathbf{n}'^T\mathbf{m}' = 0$. Like points differences between points will be transformed by the original transformation matrix $\mathbf{Tm} = \mathbf{m}'$. Furthermore a matrix multiplied with its inverse is the identity $\mathbf{T}^{-1}\mathbf{T} = \mathbf{I}$. This leads to the derivation:

$$\mathbf{n}^T\mathbf{m} = 0 \tag{1.26}$$

$$\mathbf{n}^T\mathbf{I}\mathbf{m} = 0 \tag{1.27}$$

$$\mathbf{n}^T\mathbf{T}^{-1}\mathbf{T}\mathbf{m} = 0 \tag{1.28}$$

$$(\mathbf{n}^T\mathbf{T}^{-1})(\mathbf{T}\mathbf{m}) = \mathbf{n}'^T\mathbf{m}' = 0 \tag{1.29}$$

$$(\mathbf{n}^T\mathbf{T}^{-1})(\mathbf{T}\mathbf{m}) = \mathbf{n}'^T(\mathbf{T}\mathbf{m}) \tag{1.30}$$

$$\mathbf{n}^T\mathbf{T}^{-1} = \mathbf{n}'^T \tag{1.31}$$

$$\Rightarrow \mathbf{n}' = \mathbf{T}^{-T}\mathbf{n} \tag{1.32}$$

Thus normals will be transformed by the transposed inverse transformation matrix.

1.3 Point Lights

Point lights emit light from a single point only. To create a point light its location $\mathbf{p}_l$, color $\mathbf{c}_l$ and intensity l_s have to be stored.

[7] Ray Tracing from the Ground up page 418.
[8] Fundamentals of Computer Graphics, Lecture 3 Transformations, Philip Dutré, slide 60

1.4 Shading

Shading with respect to point lights can be done with a simplified form of the rendering equation. Taking only into account ambient and distance attenuated direct illumination one arrives at:[9] [10]

$$L(\mathbf{p},\omega) = \rho_a \mathbf{c}_d \cdot (l_s \mathbf{c}_l) + \sum_{j=1}^{n} (\rho_d \mathbf{c}_d/\pi) \cdot (l_{s,j} \mathbf{c}_{l,j})(\mathbf{n}^T \mathbf{l}_j)/(d_j^2) \cdot V(\mathbf{p}, \mathbf{p}_{l,j}) \tag{1.33}$$

With ρ describing the material reflectivity with respect to ambient or diffuse illumination. Light intensity is denoted by l. The $\mathbf{c}$ vectors represent r,g,b color values, $\mathbf{n}$ denotes the normal vector. The vector ω points from $\mathbf{p}$ to the camera and $\mathbf{l}$ to the light source. Finally the visibility function V is one if the location of the point light $\mathbf{p}_l$ is visible from the object intersection point $\mathbf{p}$ and zero if not.

1.5 Camera

In this project a perspective camera is used. To set it up the resolution (r_x, r_y), origin $\mathbf{e}$, view direction $\mathbf{l}$, spacial orientation $\mathbf{up}$ and the field of view angle δ have to be provided. Using δ and the resolution data the width w and hight h of the view plane can be computed:

$$w = 2 \cdot d \tan(\delta/2) \tag{1.34}$$

$$h = r_y \cdot w \cdot 1/r_x \tag{1.35}$$

The view plane distance can be kept at $d = 1$ for all practical purposes. Changing d would change the zoom of the camera just like changing the view angle does. Making d a variable user parameter would therefore have no additional value. In order to describe the view plane and generate primary rays more easily the camera coordinate system is used. This system is set up using:

$$\mathbf{w} = (-1) \cdot \mathbf{l} \tag{1.36}$$

$$\mathbf{u} = (\mathbf{up} \times \mathbf{w})/\|\mathbf{up} \times \mathbf{w}\| \tag{1.37}$$

$$\mathbf{v} = \mathbf{w} \times \mathbf{u} \tag{1.38}$$

In this coordinate system points on the view plane can be expressed in terms of $\mathbf{u}$ and $\mathbf{v}$, which in turn become primary ray directions if $\mathbf{w}$ is subtracted. With the known camera position as the ray origin all necessary information to generate primary rays is available.

1.6 Python or Java

In this section two basic ray tracing implementations using only diffuse shading without shadows in Java and Python will be compared. A scene consisting of a plane with a chess like procedural texture as well a two spheres will be set up in both languages. The

[9] Ray tracing from the ground up page 265.
[10] Computergrafiek Project sessie 1 page 41.

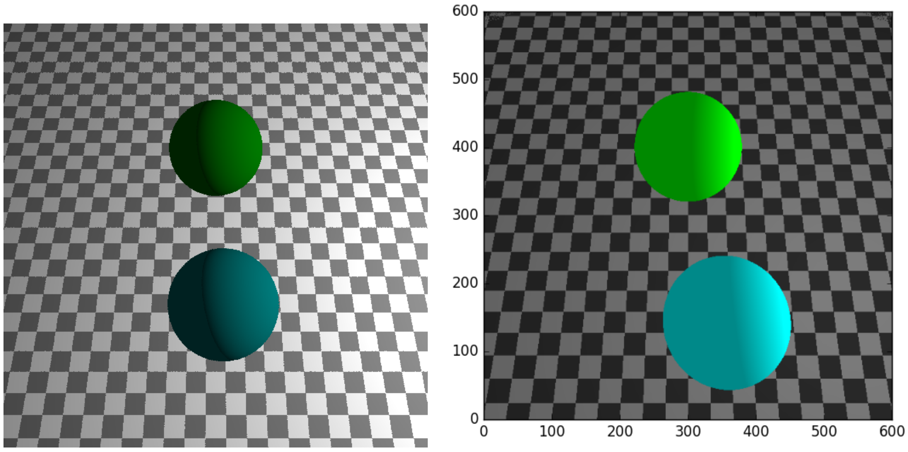


Figure 1.1: Two similar scenes rendered using comparable rendering code in Java and Python .

rendered 600x600 pixel images are shown in figure 1.1. With the result of the java code on the left and the python version on the right. The first image took 1.18s to compute while the second was ready after 16.61s. The python code is executed faster if it is run a second time as compiled python files will be generated, but for this scene it never falls under 10s. A popular method to speed up python code is to use the typed Cython dialect.[11] However cythons multi-threaded capabilities are limited due to pythons global interpreter lock, additionally cython compiled classes can only be accessed from compiled code, which leads to a significant extra amount of code conversion work. Therefore the remainder of the project was done exclusively in Java. Javas strong typing avoids the python overhead which slowed down the ray tracing process considerably, as the python interpreter looses a lot of time determining the type of every object.

[11]Cython: The Best of Both Worlds,Computing in Science & Engineering, S. Behnel; R. Bradshaw et al., Computing in Science & Engineering `http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5582062`

Milestone 2

Textures and acceleration

2.1 Acceleration

Rendering the triangle meshes quickly becomes a time intensive task if the triangles forming the model are stored in a simple list. For example the scene shown in figure 2.1 consists of the Utah teapot standing on a plane, illuminated by a point light. Rendering it without an acceleration structure with enabled shadows takes approximately 1151 seconds or 19 minutes and 11 seconds. A first measure to improve matters is to surround the mesh with a box. Only if an incoming ray hits the box is it intersected with the triangles of the mesh. Using a bounding box reduces the rendering time to 613 seconds or about 10 min for scene 2.1. To reduce computations required to render the scene further the number of intersection tests per pixel has to be reduced. This can be done by subdividing the bounding box. Three different ways of subdivision will be considered. All of them start by finding the longest dimension of the box, but differ in the way this longest dimension is split.

2.1.1 Middle-Split

The first method splits the parent box into two children in the geometric middle of the longest axis. After the split the triangles are assigned to the left or right sub-box according to the position of their centroid. The two subboxes are then resized to ensure that no triangle vertices's lay outside the two boxes. After the resizing operation it is possible, that the two boxes overlap.

2.1.2 Split a sorted list

The second method sorts the triangles contained in the box according to their centroid coordinate of the axis under consideration. The dimensions of the child boxes are then obtained by splitting the sorted list in the middle and assigning each child half of the triangles. The new boxes are then resized such that they contain all of their triangles entirely.

	read[s]	split[s]	render[s]	total[s]
Middle-Split	0.16	0.95	23.61	24.72
Sort-Split	0.17	0.06	30.47	30.7
SAH	0.16	0.18	17.27	17.61

Table 2.1: Time measurments taken using various splitting methods when rendering the teapot scene.

	read[s]	split[s]	render[s]	total[s]
Middle-Split	11.4	3.7	10.53	25.63
Sort-Split	10.6	4.6	20.5	35.7
SAH	11.0	14.3	6.71	32.01

Table 2.2: Time measurments taken using various splitting methods when rendering the stanford dragon scene.

2.1.3 Surface area heuristic

Finally the surface area heuristic is considered, which splits along several points and chooses from a set of available options by evaluating the cost function:

$$\text{cost} = N_{\text{left}} \cdot \frac{S_{\text{left}}}{S_{\text{total}}} + N_{\text{right}} \cdot \frac{S_{\text{right}}}{S_{\text{total}}} \tag{2.1}$$

Where N denotes the number of primitives in a given box and S its surface along the dimension under consideration. Before the cost function can be evaluated the parent box is split along its longest edge c_{SAH} times, and $c_{SAH} + 1$ child boxes are created. The child boxes of each side of the cut are then combined and resized and the cost function is evaluated for each combination. Finally the combination with the lowest cost is chosen.

2.1.4 Intersection testing

In order to avoid unnecessary computation work it is important to avoid intersecting boxes, where a visible hit cannot be found. To do this a ray is always intersected with the box having the centroid closer to the camera. If the difference between the two centroid distances is larger then a predefined constant, the second box is only considered if an intersection found in the first box lies in a space where both boxes overlap.

2.1.5 Results

Table 2.1 shows the timing results of the three implemented splitting methods. Considering the decrease from the initial 10 minutes the reduction of rendering time is impressive for each method. The surface area heuristic is the best choice here as its tree is the most efficient and the additional time needed to construct it does not make a difference in this example. Table 2.2 shows the situation for the dragon scene. Here the surface Area heuristic is still the most efficient in terms of rendering time, but it looses during the tree generation phase. These results indicate, that there is still room for further optimization of the SAH-tree generation function.

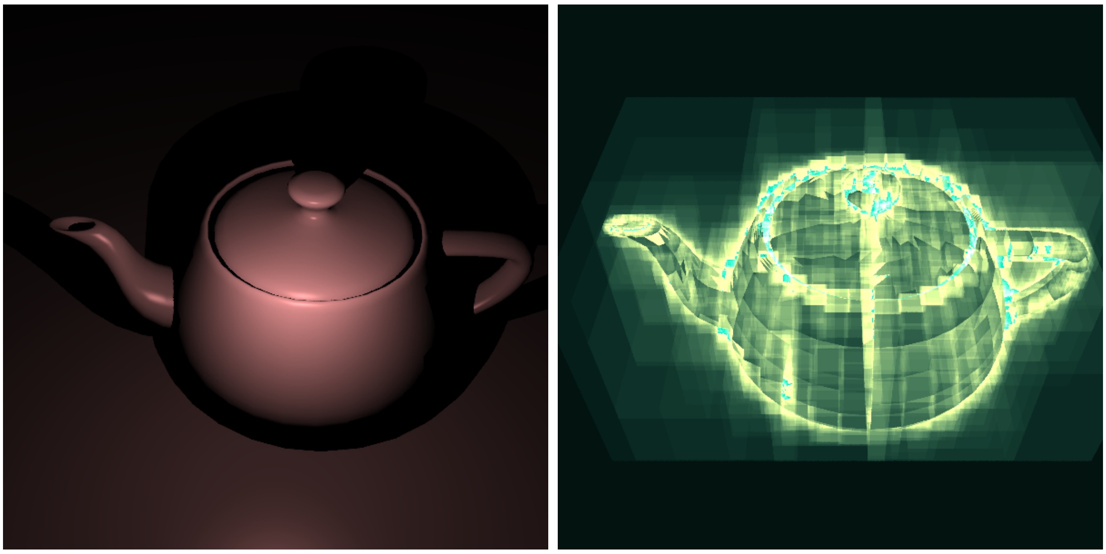


Figure 2.1: Utah-Teapot rendered image and false color visualizations of the intersections with a color map maximum at 180 intersections, with 10 SAH cuts per split.

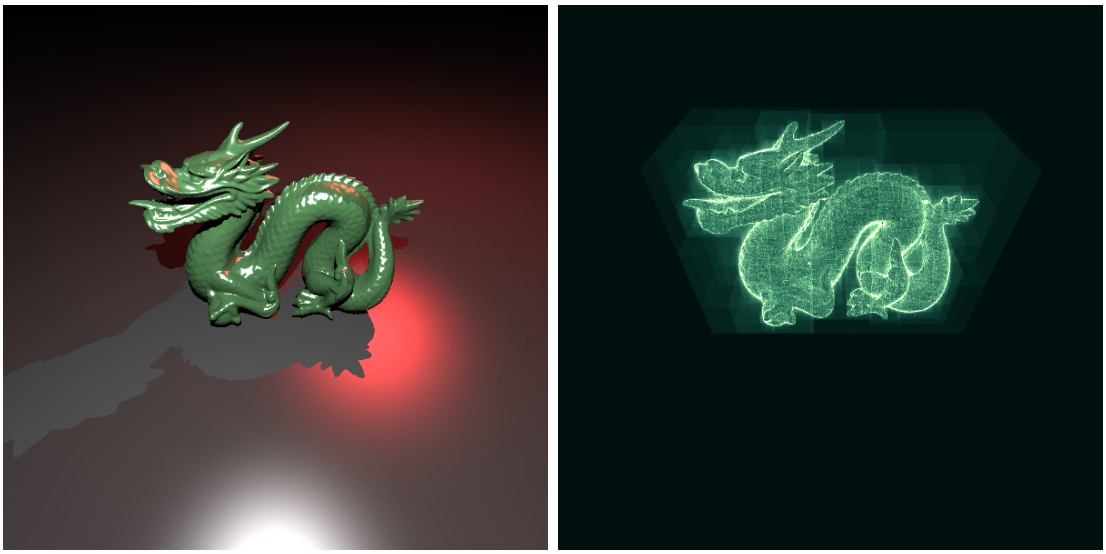


Figure 2.2: High resolution Standford-Dragon model consisting of 871306 triangles. Rendered image and false color visualizations of the intersections with a color map maximum at 180 intersections and 4 SAH cuts per split

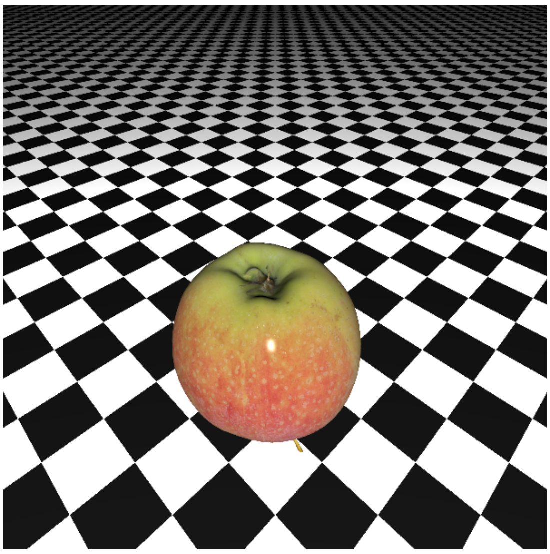


Figure 2.3: Apple on chessboard

2.2 Textures

In order to use textures each object hit point has the be associated with coordinates in the u, v texture space. For wavefront (`.obj`) these coordinates come from a data file along with the triangle-mesh data. It is important to note that the coordinate system for wavefront objects is located at the top left of the image file. With that in mind textures, which sometimes come shipped along with `.obj`-files can be used. An apple texture is shown in figure 2.3. In the same image a procedural texture is used for the plane.

2.2.1 Procedural texturess

The chess-board like pattern is generated by dividing the u and v coordinate by the desired size of the rectangles. If the rounded results of the division are added and the modulo with two is computed every point can be associated with one or zero. Which serves as the criterion to choose the texture color.

Julia-set textures

More interesting patterns can be computed by using fractals. Often used examples are Julia fractals found from functions of the form:

$$f(z) = z^2 + c \quad z \in \mathbb{C} \tag{2.2}$$

Where c is considered the seed. Changing c changes the appearance of the fractal drastically. Julia sets are generated by applying the function $f(z)$ to itself repeatedly, compute $f(z), f(f(z)), f(f(f(z))), \ldots$. This process is repeated until the iteration exceeds a predefined limit value $\|z_n\| > k$, or a maximum number of iterations is reached. Each point is then colored according to how many iterations n it took to reach the limit k. The logarithm of the iterations counter is hashed linearly from 0 to the largest occurring value to colors. Plots with $c = -0.07, 0.652, c = -0.02, 0.652, c = -0.02, 0.8$ with the initial $z_{\text{init}} \in [-1, 1]$ and $k = 1$ are given in figure 2.4.

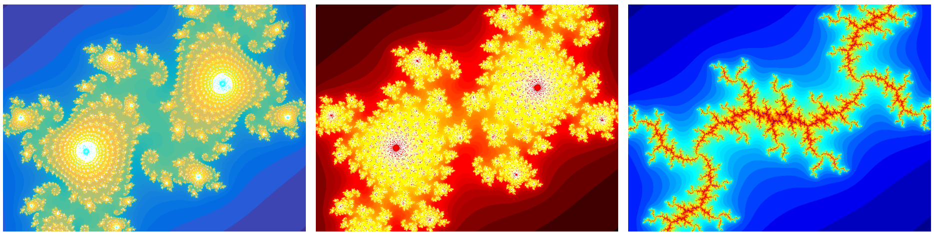


Figure 2.4: Julia fractals using different seeds and colormaps.

Function meshes

To obtain a 3d visualization of a function, a triangle mesh can be computed from grid data. The mesh can be set up by assembling triangles following the rule:

$$p_1 = \{x[i], y[j], f_n(x[i] \cdot y[j]\} \tag{2.3}$$
$$p_2 = \{x[i+1], y[j], f_n(x[i+1] + i \cdot y[j])\} \tag{2.4}$$
$$p_3 = \{x[i], y(j+1), f_n(x[i] + i \cdot y[j+1])\} \tag{2.5}$$
$$p_4 = \{x[i+1], y(j+1), f_n(x[i+1] + i \cdot y[j+1]\} \tag{2.6}$$
$$\mathrm{tri}_1 = \{p_1, p_2, p_4\} \tag{2.7}$$
$$\mathrm{tri}_2 = \{p_1, p_4, p_3\} \tag{2.8}$$
$$\tag{2.9}$$

Which is assuming that the x and y values are stored in vectors and that f_n has been stored in a two dimensional matrix-array. Then the meshes' triangles can be constructed by looping over the expressions outlined above. The triangle vertices are stored in a counterclockwise order. Thus the normals follow from equation 1.1. The result of setting up such a mesh consisting of $(2*(N-1))^2$ with $N = 800$ triangles using the inverse value of the logarithmic iteration counter as z-coordinate is shown in figure 2.5. A top down view is shown on the left and a 3d perspective view computed from intersecting the mesh on the right.

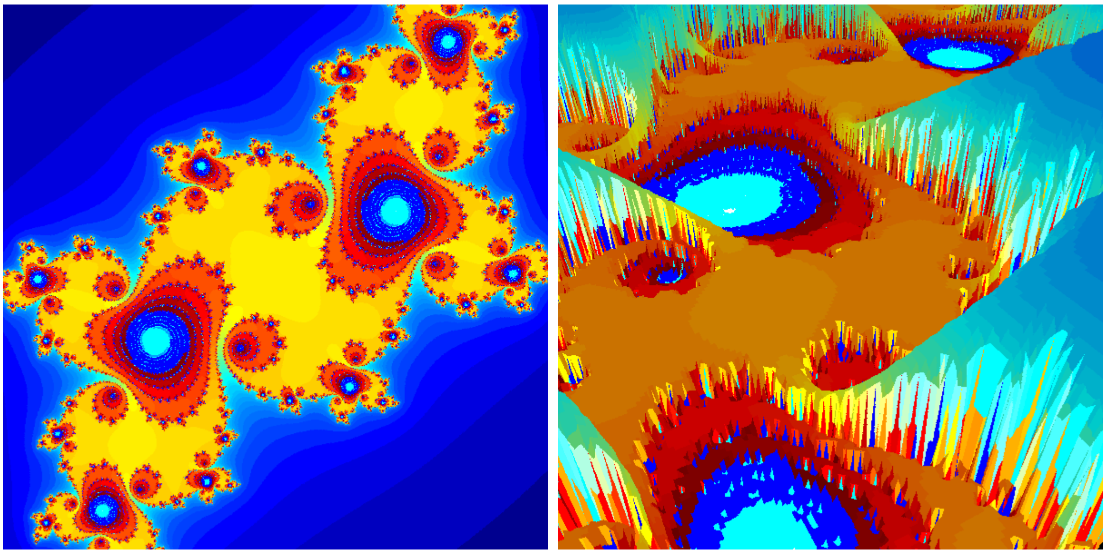


Figure 2.5: Top and 3d view of the Julia fractal computed from $f(z) = z^2 + c$ with $c = -0.1, 0.651$.

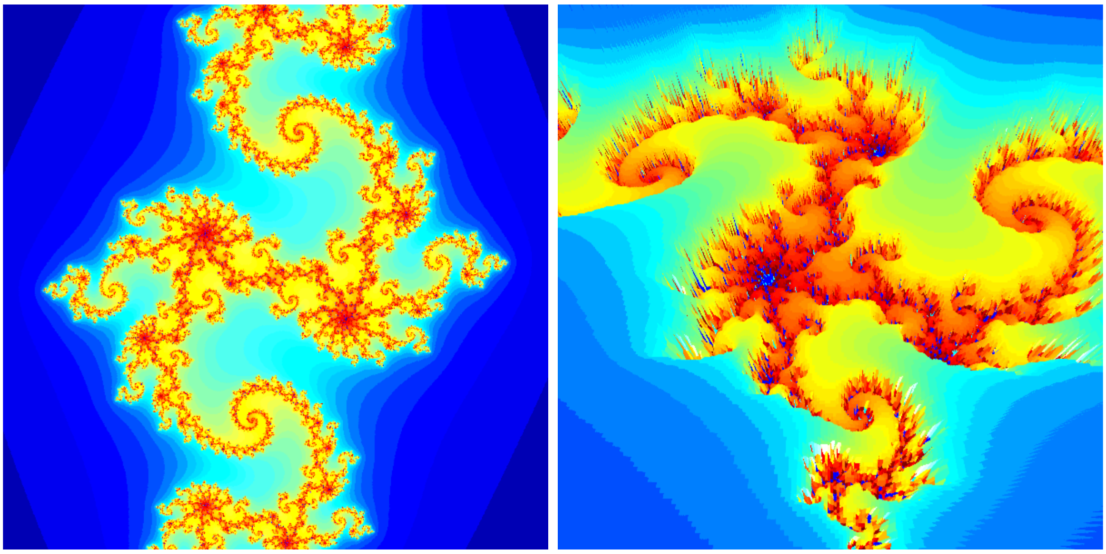


Figure 2.6: Top and 3d view of the Julia fractal computed from $f(z) = z^2 + c$ with $c = -0.8, 0.175$.

Milestone 3

Special Effects

3.1 Common part

3.1.1 Anti-aliasing

In computer graphics using raster images can lead to ”stairstepping” or aliasing. Aliasing can generally be replaced by noise, if the ray origins are randomly placed inside a pixel. Fully random placement of the ray-starting point is not optimal, as the randomly distributed pixels can clump together and leave some areas poorly sampled. Therefore each pixel is subdivided into sub-areas and each sub-area is given one sample. This is called jittered sampling.

3.1.2 Area Lights

In contrast to point lights, area lights have a surface. In order to solve the rendering equation for scenes lit by area lights, the area form is considered [1]:

$$\mathbf{L}_r(\mathbf{p}, \omega_0) = \int_{A_{\text{light}}} f_r(\mathbf{p}, \omega_i, \omega_0)\mathbf{L}_e(\mathbf{p}', -\omega_i)V(\mathbf{p}, \mathbf{p}')G(\mathbf{p}, \mathbf{p}')dA_{\mathbf{p}'} \tag{3.1}$$

The above equation allows it to find the outgoing light at a primary ray intersection point $\mathbf{p}$. Given the reflection orientation vector ω_0 and the incoming light direction $\omega_i = (\mathbf{p} - \mathbf{p}')/\|\mathbf{p} - \mathbf{p}'\|$. The visibility function $V(\mathbf{p}', \mathbf{p})$ is one if point $\mathbf{p}$ is visible from point $\mathbf{p}'$ and zero if not. The geometry term G is given by:

$$G = \frac{cos(\theta_i)cos(\theta')}{\|\mathbf{p}' - \mathbf{p}\|^2} \tag{3.2}$$

With $cos(\theta_i), cos(\theta')$ given by $\mathbf{n}_{\mathbf{p}}^T\omega_i$ and $\mathbf{n}_{\mathbf{p}'}^T(-\omega_i)$ respectively. Finally $\mathbf{p}'$ must be a random point within the area light source. The solution of the rendering equation can be estimated by using Monte-Carlo integration:

$$\mathbf{L}_r(\mathbf{p}, \omega_0) \approx \frac{1}{n_s}\sum_{j=1}^{n_s} \frac{f_r(\mathbf{p}, \omega_{i,j}, \omega_0)\mathbf{L}_e(\mathbf{p}', -\omega_{i,j})V(\mathbf{p}, \mathbf{p}'_j)G(\mathbf{p}, \mathbf{p_j}')}{p(\mathbf{p}')} \tag{3.3}$$

[1] Ray Tracing from the ground up page 329

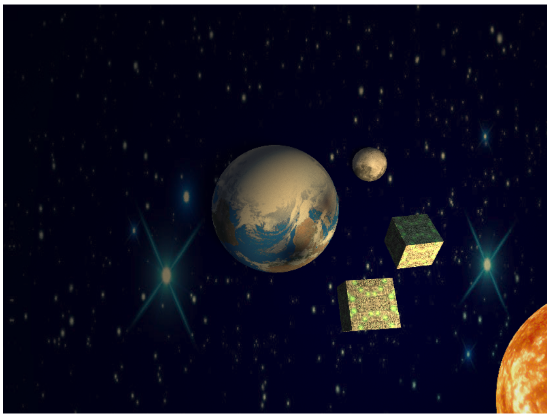


Figure 3.1: A spherical area light causing smooth shadows on a sphere

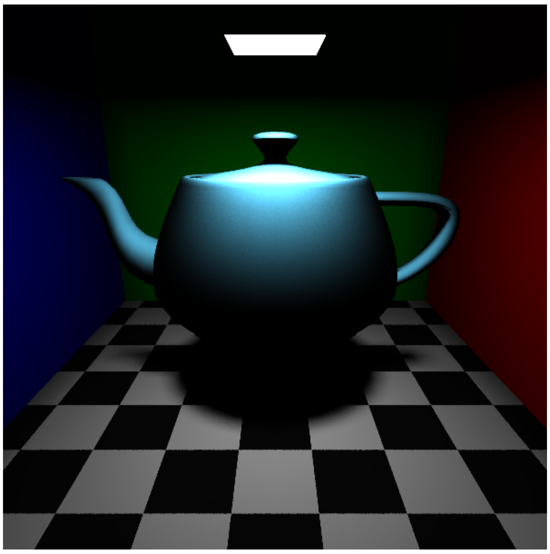


Figure 3.2: A rectangular area light illuminating the Utah teapot

Here n_s denotes the number of samples, and $p(\mathbf{p}')$ the probability density function. The simplest choice is to distribute the samples uniformly which leads to $p(\mathbf{p}') = \frac{1}{A_l}$. A scene with a spherical and planar light source is shown in figure 3.1. Using area light sources leads to smooth shadows, like those caused by the cubes on the earth's surface.

3.2 Special effects

3.2.1 Normal Mapping

In order to add additional normal information or speed up the rendering of wavefront objects, normal data can be loaded from a file. The normal-information is coded as:

$$c_r = 0.5 + 0.5 \cdot n_x$$
$$c_g = 0.5 + 0.5 \cdot n_y$$
$$c_b = 0.5 + 0.5 \cdot n_z$$

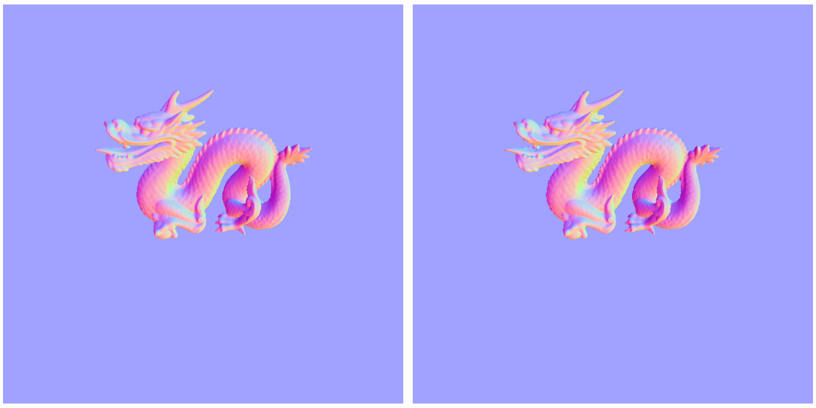


Figure 3.3: Stanford-Dragon on plane rendered using a high resolution mesh (left) and low resolution mesh with normal map(right).

which can be transformed back using:

$$n_x = -1.0 + 2 * (c_r/255) \tag{3.4}$$
$$n_y = -1.0 + 2 * (c_g/255) \tag{3.5}$$
$$n_z = -1.0 + 2 * (c_b/255) \tag{3.6}$$

The normal information can then be used for rendering purposes. In case of the Stanford dragon shown in figure 3.3 it took 32s for the high resolution mesh and 6.02s for the low resolution mesh with normal map to produce the same result.

3.2.2 Physical materials

An important factor in improving the quality of the rendered images is the bidirectional reflectance distribution function (brdf) $f_r(\mathbf{p}, \omega_i, \omega_0)$. The brdf-model may be split into a diffuse and specular component

$$f_r(\mathbf{p}, \omega_i, \omega_0) = f_d(\mathbf{p}, \omega_i) + \rho_s f_s(\mathbf{p}, \omega_i, \omega_0). \tag{3.7}$$

In this project a lambertian

$$f_d(\mathbf{p}, \omega_i) = \frac{\rho_d}{\pi} cos(\theta_i) \tag{3.8}$$

model is used for diffuse part. θ_i denotes the angle between incoming light and surface normal. The cosine term can be computed using $\mathbf{n}_\mathbf{p}^T \omega_i = cos(\theta_i)$. For the specular part a Phong reflection model

$$f_s(\mathbf{p}, \omega_i, \omega_0) = \rho_s(\mathbf{r}^T \omega_0)^e \tag{3.9}$$

is considered.[2] The vector r denotes the mirror reflection direction. Furthermore a Cook-Torrance model for the specular part has been implemented. This model is taken from

[2]Ray Tracing from the ground up page 281

their original 1982 publication: [3]

$$\mathbf{h} = \frac{\mathbf{c} + \omega_i}{\|\mathbf{c} + \omega_i\|} \tag{3.10}$$
$$c = \mathbf{c}^T\mathbf{h} \tag{3.11}$$
$$g = \sqrt{n * n + c * c - 1} \tag{3.12}$$
$$n = \frac{1 + \sqrt{f_0}}{1 - \sqrt{f_0}} \tag{3.13}$$
$$F = \frac{1}{2}\frac{(g-c)^2}{(g+c)^2}(1 + \frac{[c(g+c)-1]^2}{[c(g-c)+1]^2}) \tag{3.14}$$
$$\alpha = \cos^{-1}(\mathbf{h}^T\mathbf{n_p}) \tag{3.15}$$
$$D = \frac{1}{m^2 cos^4(\alpha)} e^{[-(\tan(\alpha)/m)]^2} \tag{3.16}$$
$$G = \min(1, \frac{2(\mathbf{n_p}^T\mathbf{h})(\mathbf{n_p}^T\mathbf{c})}{(\mathbf{c}^T\mathbf{h})}), \frac{2(\mathbf{n_p}^T\mathbf{h})(\mathbf{n_p}^T\omega_i)}{(\mathbf{c}^T\mathbf{h})}) \tag{3.17}$$
$$f_s(\mathbf{p}, \omega_i, \mathbf{c}) = \frac{\rho_s}{\pi}\frac{DG}{(\mathbf{n_p}^T\omega_i)(\mathbf{n_p}^T\mathbf{c})}F(c,g) \tag{3.18}$$
$$\tag{3.19}$$

The model depends on the two input parameters f_0 and m as well as the scaling factor ρ_s. The first input is the value of the Fresnel-equation at normal incidence. The second is the root mean square slope m of assumed material facets. The smaller both parameters become the less mirror like will the resulting material will be. The vectors $\mathbf{c}, \omega_\mathbf{i}$ point to the camera and the light source respectively. The vector $\mathbf{h}$ represents the normalized vector in the direction of the angular bisector of the view and light orientation vectors. The c variable stands for the cosine of the angle between $\mathbf{c}$ and $\mathbf{h}$ or ω_i and $\mathbf{h}$. The Fresnel term F determines how light is reflected from each of the assumed smooth micro-facets. A facet slope distribution function D is used. This term depends on the angle between $\mathbf{n_p}$ and $\mathbf{h}$ called α. Finally the geometric attenuation factor G is the last term required to compute the Cook-Torrance BRDF. Gerhard Richter's design of the cologne cathedral's south window and the light effects it produces is shown in figure 3.4. Inspired by the design a rectangular light source with pseudo-randomly distributed squares has been created. This area light source is placed at a ninety degree angle on top of a floor plane. The Specular brdf-component of the floor plane will be varied. First Phong's model is used.

Results are shown in figure 3.5. It can be observed that the larger the Phong-exponent e is chosen, the more pronounced the reflection becomes. Which makes sense as with larger values for e the set of possible incoming light vectors at $\mathbf{p}$ with large specular contributions becomes smaller and smaller. Therefore the largest specular contributions come from only a small set of sample points $\mathbf{p}'$ on the light source, which are close to each other. Therefore one expects to see more detailed reflections for larger values of e.

In the images shown in figure 3.6, the Phong model, has been replaced with a Cook-Torrance model for specular reflection. It can be observed that the larger m is chosen the

[3] A Reflectance Model for Computer Graphics, Robert L. Cook, Kenneth E. Torrance, ACM Transactions on Graphics, Vol. 1, No. 1, January 1982, Pages 7-24

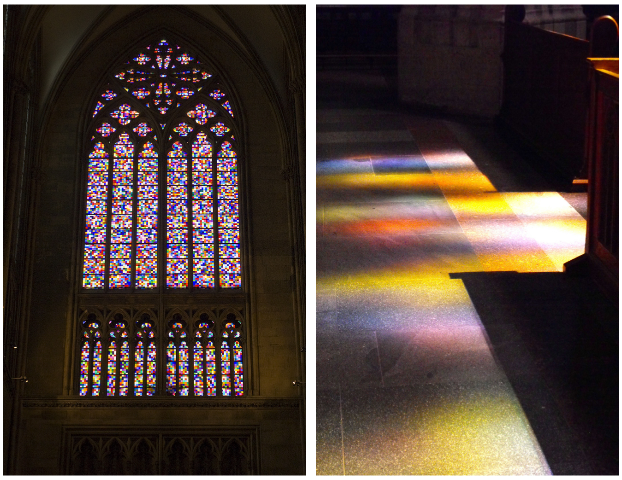


Figure 3.4: Gerhard Richter's southern wing main window of the cologne cathedral consisting of 11.500 pixels colored using 72 different colors and pseudo-randomly arranged. On the right the lighting effect of the window on the cathedral floor can be seen.

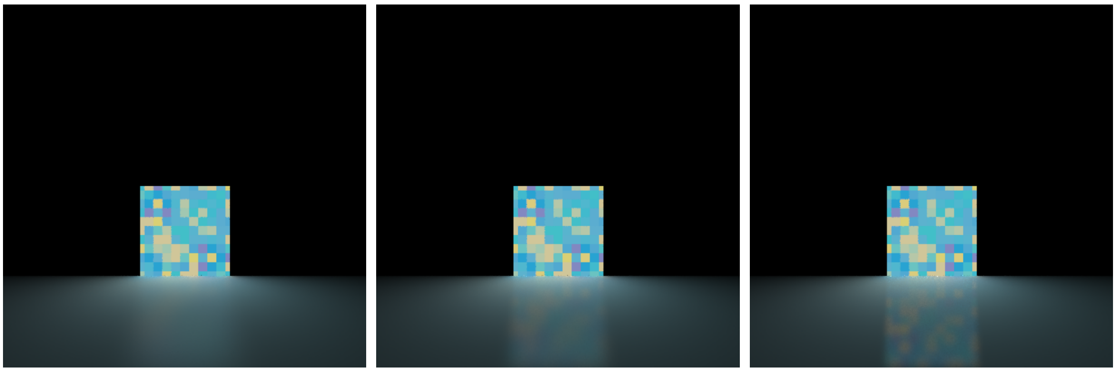


Figure 3.5: Importance sampling of Phong's specular reflection model using $e = 100$(left), $e = 500$(middle), $e = 2000$(right). In all cases $\rho_d = 0.8$ and $\rho_s = 0.2$

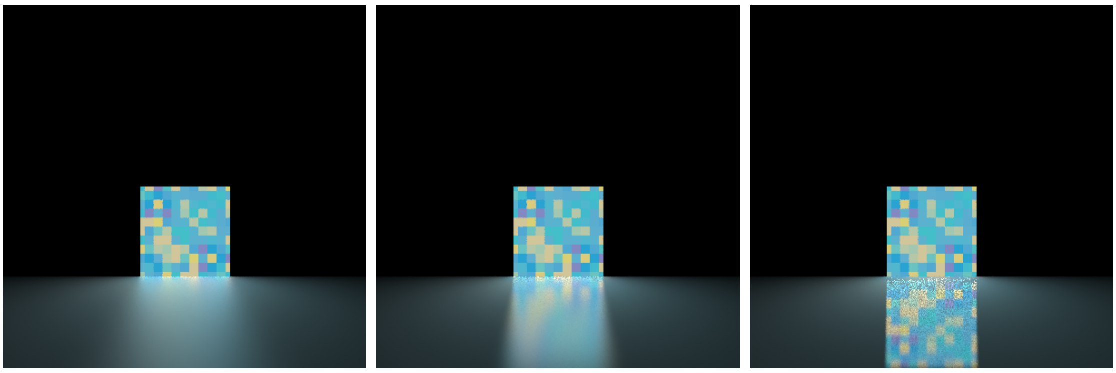


Figure 3.6: Scenes computed using Cook-Torrance specular reflection models with parameters $f_0 = 0.0366, m = 0.276$ (left), $f_0 = 0.025, m = 0.076$(middle), $f_0 = 0.03, m = 0.01$(right). In all cases $\rho_s = 0.2$ and $\rho_d = 0.8$

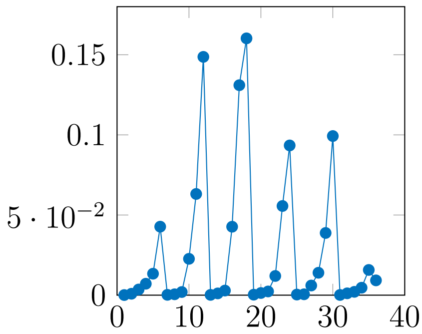


Figure 3.7: Discrete probabilities for 36 light source partitions (left). Right histogram of 500 samples drawn from this distribution using the discrete version of the inverse transform method.

less visible reflection details become. Which is to be expected as m models the root mean square slope of model material facets, and the larger these slopes are the more bumpy and therefore less mirror like the material will be.

3.2.3 Importance Sampling

In oder to reduce the amount of samples required to come close to a decent approximation of the solution importance sampling is used. It's key concept is to distribute the sample points $\mathbf{p}'$ on the light source according to their contribution to the solution, important areas of the light should be sampled more often then insignificant ones. To achieve this goal the light source is subdivided into parts. A random point of each part is sampled and an importance fraction

$$q_j = i_j / I_{tot} \tag{3.20}$$

consisting of the sampled importance function values over the total sum $I_{\text{tot}} \sum_{i=0}^{N} i_j$ is computed. These fractions generally add up to one, except for cases where the total sum is very small. If the fractions add up to one they can be interpreted as a discrete probabilities of a random variable X, while $P(X = x_j) = q_j$. A realization of X can then be generated by using: [4]

$$\sum_{j=0}^{i-1} q_j \leq U < \sum_{j=0}^{i} q_j. \tag{3.21}$$

Here a uniformly distributed random variable $U \in (0, 1)$ is generated first. X is then set to $X = x_i$ if the condition above holds. This can be seen as a way of mapping the uniformly distributed space $(0, 1)$ into the discrete distribution space given by the set of $\{q_j\}$, by computing the discrete cumulative probability density function and mapping U into it. Figure 3.7 shows the effect of transforming a uniformly distributed variables the way described above. It can be observed that the histogram resembles the shape of the discrete probability density function. During the Monte-Carlo estimation of the integral the probability term in the denominator changes to $p(\mathbf{p}') = q_j \cdot p_j(\mathbf{p}')$ with $p_j(\mathbf{p}') = \frac{1}{A_j}$ and A_j the area of the sub-light.

[4]Marko Boon, Technische Universiteit Eindhoven, 2WB05 Simulation Lecture 8: Generating random variables, page 17. `http://www.win.tue.nl/~marko/2WB05/lecture8.pdf`

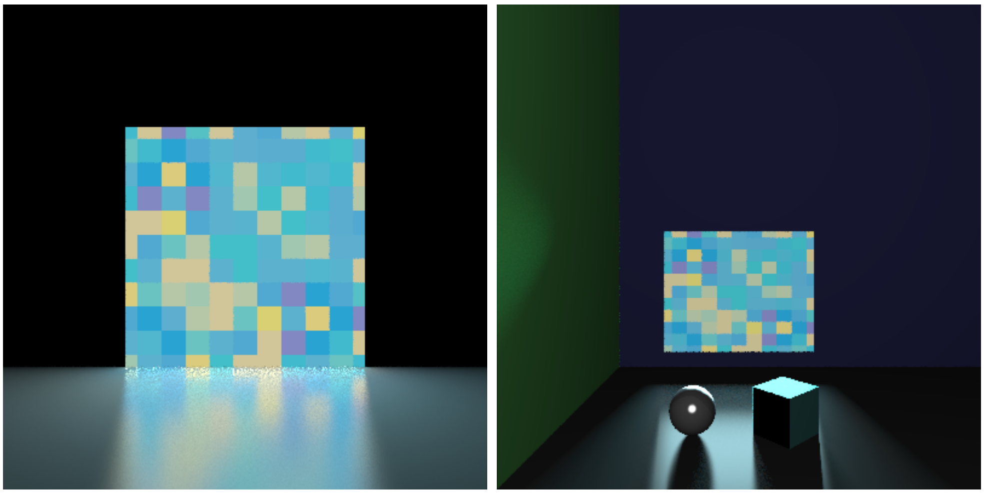


Figure 3.8: Rendering of the pixelated window scene using 2500 samples per intersection point.

Error analysis

This section is devoted to an analysis of the convergence behavior of the rendering equation given the scene consisting of the pixeled light source illuminating a plane. Mean square error computations are used to judge the degree of convergence in various images. In this report the mean square error is defined as:

$$\mathrm{e}_{\mathrm{mse}} = \frac{1}{N}\sum_{i=0}^{N}\frac{1}{3}\sum_{j=0}^{2}(p_{1,i,j} - p_{2,i,j})^2 \tag{3.22}$$

Figure 3.8 shows two images which where rendered using 2500 uniformly distributed samples on the light source. In the right image an additional point light has been added to illuminate the wall behind the area light source. Using this large amount of samples it can be assumed, that the solution of the rendering equation has converged to a point where the solution does not vary anymore. Using a similar low resolution images with 200 by 200 pixels a series of experiments is conducted to verify the quality of the implemented methods. Using the importance function

$$f_{\mathrm{imp}} = G \cdot (f_{\mathrm{diffuse}} + f_{\mathrm{specular}}). \tag{3.23}$$

The scene shown in figure 3.8 has been computed hundred and fifty times with the number of area light samples linearly increasing. The results are shown in figure 3.9. Initially the importance sampled solution converges faster but as the total number of samples is increased the difference decreases and figure 3.10 plain Monte-Carlo integration actually does better than the importance sampling version using function 3.23. This is probably due to the fact that this importance function used so far does only take geometry into account, but neglects the texture of the light source additionally it assumes that the $V(\mathbf{p}, \mathbf{p}') = 1$, so it does not allocate fewer samples in areas which lie in the shadow of the two new objects and do not contribute to the solution. Increasing the number of partial light sources leads to faster convergence, but longer computation times. In general the importance sampling method implemented here could be improved further, for example by

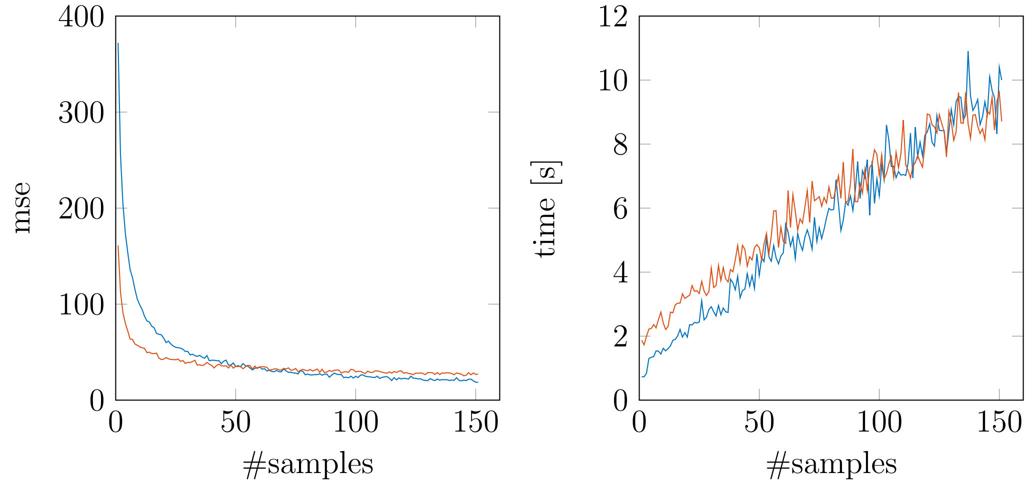


Figure 3.9: Convergence and computation time of Monte-Carlo integration with (red) and without (blue) priority sampling for the scene shown in figure 3.8 on the left. Using linearly increasing numbers of light source samples.

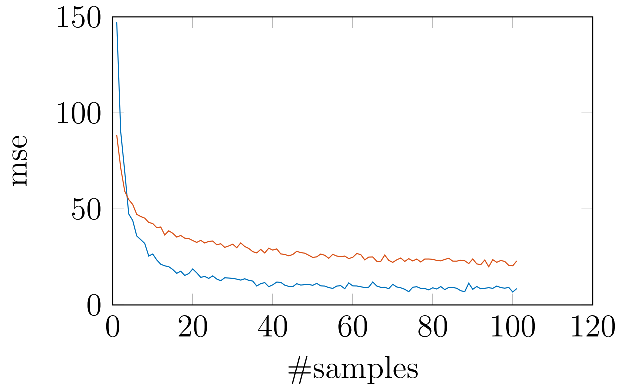


Figure 3.10: Convergence of Monte-Carlo integration with (red) and without (blue) priority sampling for the scene shown in figure 3.8 on the right. Using linearly increasing numbers of light source samples.

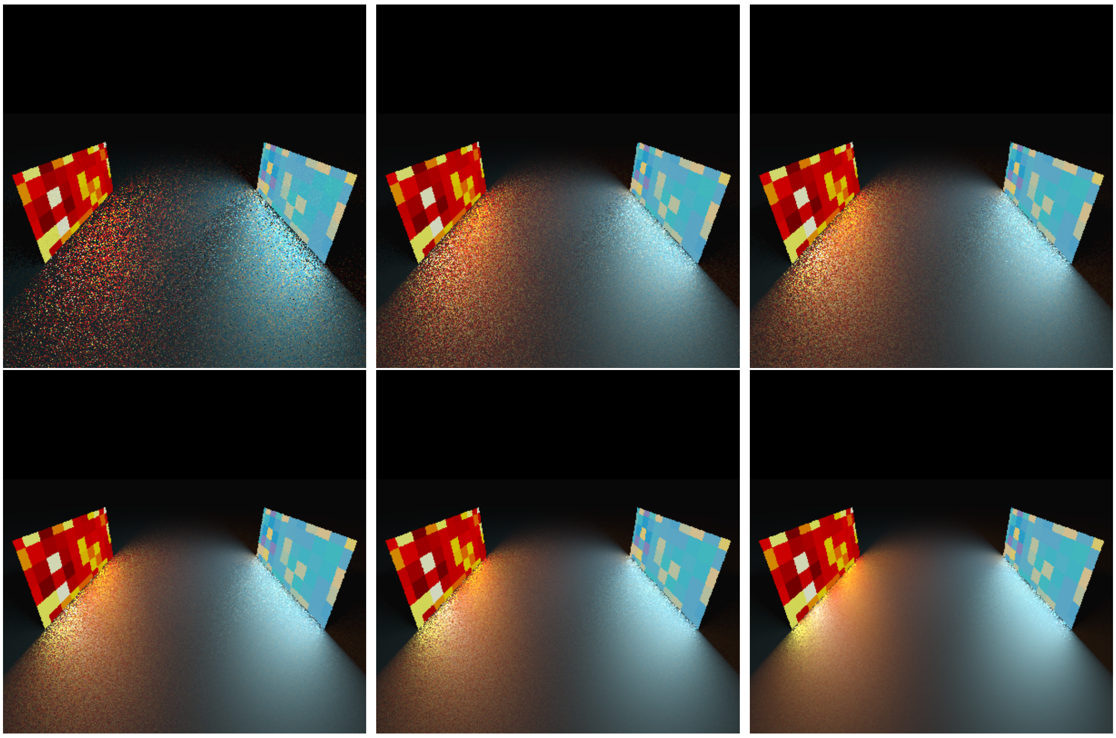


Figure 3.11: Plain Monte Carlo (red) and importance sampled Monte-Carlo with 121 subdivisions (blue). Results for 1,11,21,51,101 and 501 samples are shown from left to right. The plane material is given by $\rho_{\text{diff}} = 0.25, \rho_{\text{spec}} = 0.75, f_{\text{ct}}(0.015, 0.01)$.

finding better importance functions, which in turn could yield better sample distributions leading to faster convergence. It has been shown that the importance sampling method implemented here works better than plain Monte-Carlo integration for scenes where not too much area is covered by shadows and small sample numbers. For larger numbers of samples the two methods deliver similar results, depending on the arrangement of objects plain Monte-Carlo can be the better choice. Figure 3.11 shows a uniformly sampled Monte-Carlo light source in red on the left and a importance sampled Monte-Carlo light source on the right. The top row shows how well the importance sampling does in terms of variance reduction for small light source sample numbers.

3.3 Conclusion

In this project a ray tracer consisting of more then six-thousand lines of code has been written. Another thousand lines have been written from scratch in python, but that version had to be abandoned out of speed concerns. The Java version of the ray tracer covers key areas of the course, generally the implemented features work, but much room for further improvement still remains. The tree generation process especially for SAH-Trees could be optimized further, currently SAH beats all the other methods in terms of rendering time, but takes longer overall for complex wavefront objects consisting of large numbers of triangles, because the tree generation is slow. Importance sampling works as well but currently takes only the scene geometry into account. An improved version could use the texture information to converge even faster.