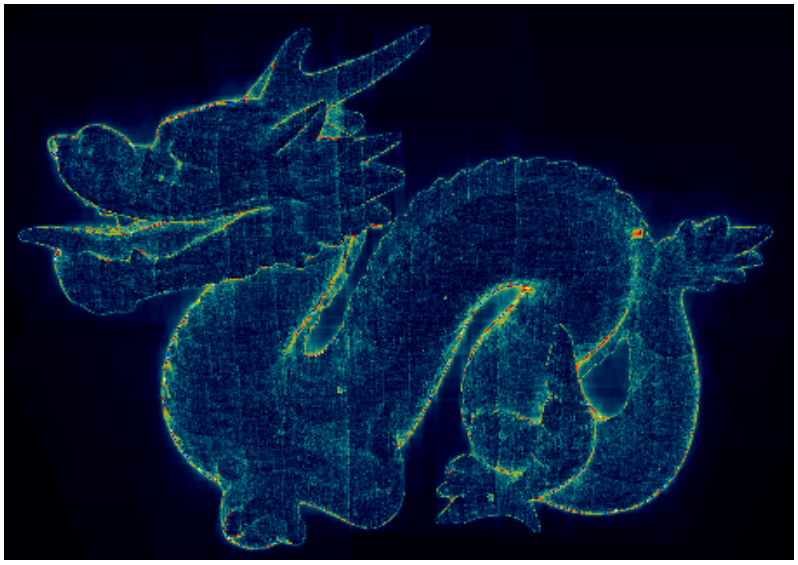
KU LEUVEN

FACULTY OF ENGINEERING

Computer Graphics Project Report



Moritz Wolter
April 11, 2016

Supervised by
Prof. Philip Dutré

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 in 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 do 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 $\mathbf{r} = \mathbf{o} + t * \mathbf{d}$ 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 $\mathbf{Ax} = \mathbf{b}$ 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 $\ltimes$ which determines, where the plane is and a point $\mathbf{a}$ 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 zero 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 move, rotate or scale the objects defined earlier four dimensional transformation matrices are defined. Four dimensions are used, as three dimensional translation can be expressed as a matrix operation.
Scaling can be done by using[6]:

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

Translation is working with:

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

[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 transformations 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 an 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 a 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.

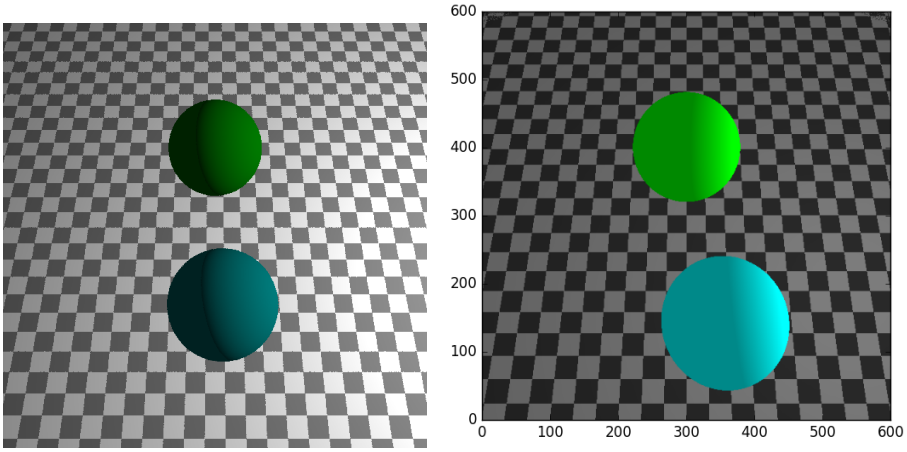


Figure 1.1: Two similar scenes rendered using comparable rendering code in Java and Python .

rendered 600x600 pixel images are shown in figure1.1. 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. If 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 a 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 further. This can be done by subdividing the bounding box. Three different ways if 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 subbox 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 is 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:

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

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. 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. This would certainly change if the implementation of the SAH tree generation function would be optimized further.

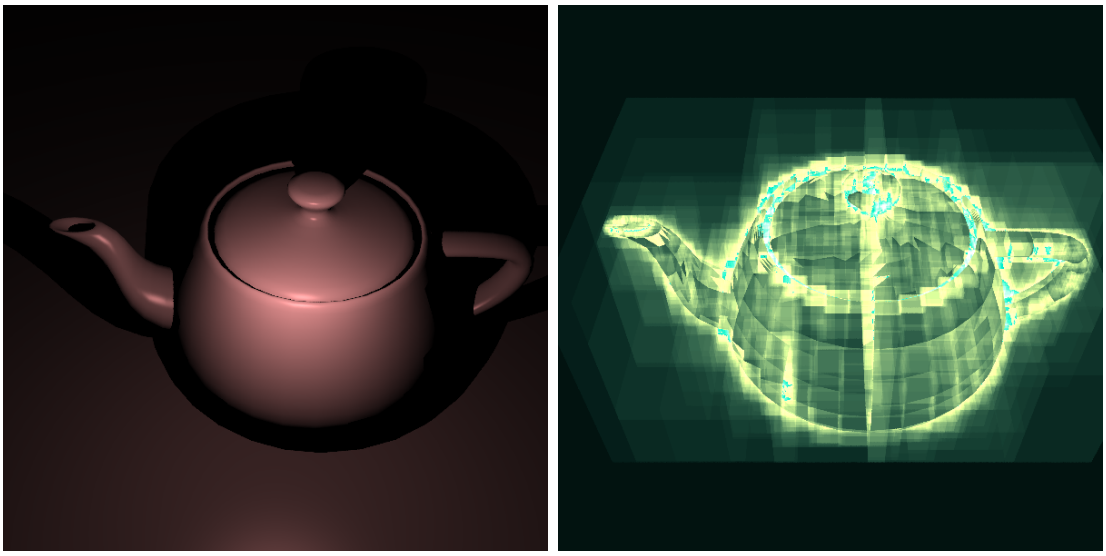


Figure 2.1: Utah-Teapot rendered image and false color visualizations of the intersections with a color map maximum at 180 intersections, with 10 SAH cuts per split.

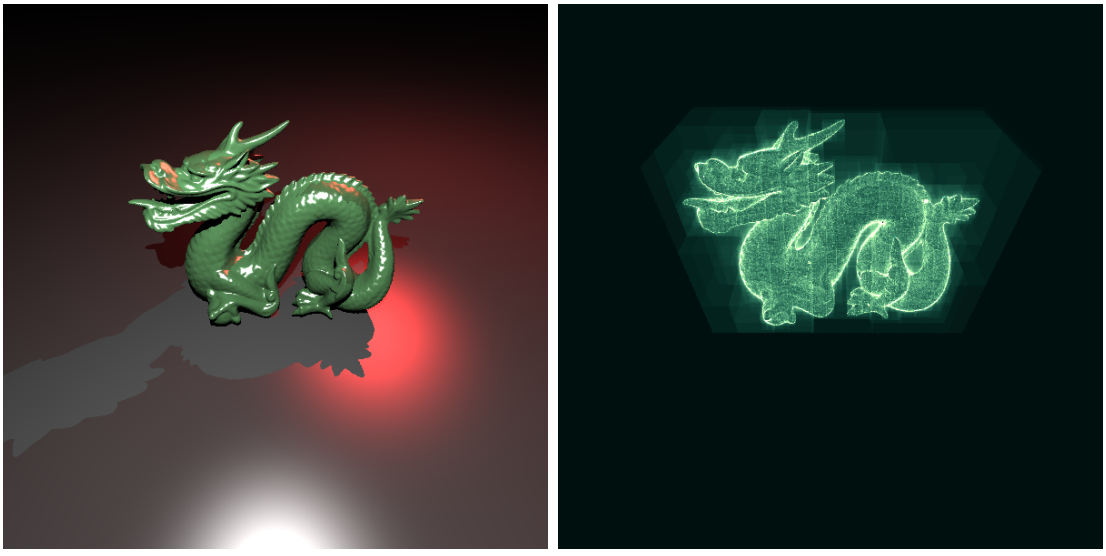


Figure 2.2: High resolution Standford-Dragon rendered image and false color visualizations of the intersections with a color map maximum at 180 intersections and 4 SAH cuts per split

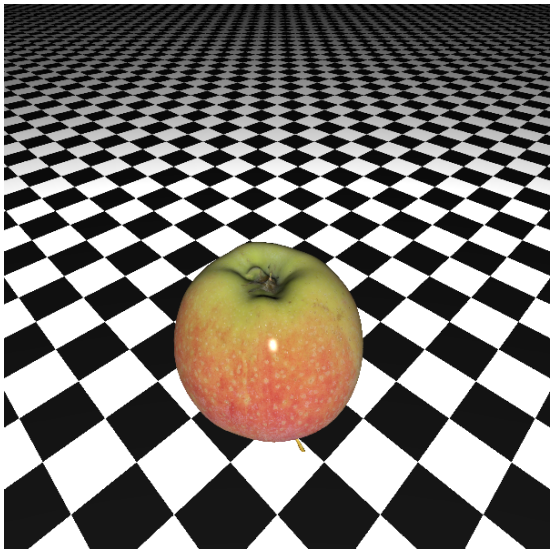


Figure 2.3: Apple on chessboard

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 of functions of the form:

$$f(z) = z^2 + c \tag{2.2}$$

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 iterations are hashed linearly from 0 to the limit value. Plots with $c = -0.07, 0.652, c = -0.02, 0.652, c = -0.02, 0.8$ with the initial $z \in [-1, 1]$ and $k = 1$ are given in figure 2.4.

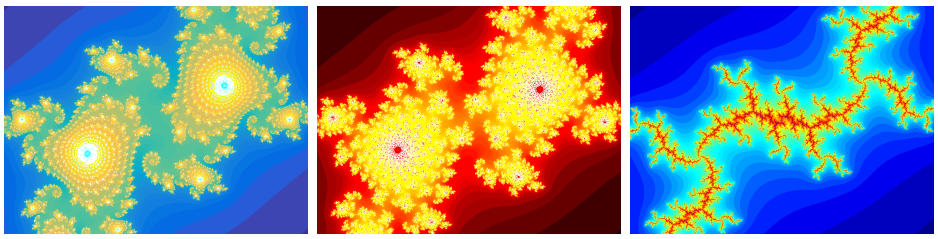


Figure 2.4: Julia fractals using different seeds and colormaps.

Milestone 3

Special Effects