3D graphics rendering using a polygon-voxel-hybrid approach

Diploma Thesis
by

Daniel Gritzner
born in
Worms

submitted to
Lehrstuhl für Praktische Informatik IV
Prof. Dr.-Ing. W. Effelsberg
Fakultät für Mathematik und Informatik
University of Mannheim

March 2014

Supervisor: Philipp Schaber

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mannheim, den 17.03.2014 Daniel Gritzner

Contents

List of Figures

List of Tables

1. Introduction

1.1. Motivation

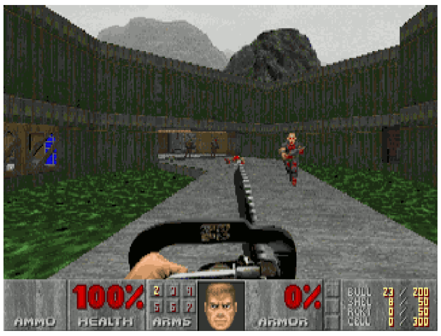
Virtual 3D environments have become a common part of our culture. From planning and previewing the interior of a house to simulations of entire worlds in video games virtual environments are used for a wide variety of applications. In particular the video games industry, which makes heavy use of those 3D environments, has grown into one of the largest entertainment industries in recent years.

Just as the applications are very varried the demands made on 3D virtual environments are very varried as well. They range from “highest quality at acceptable rendering times on expensive computer clusters”, e.g. movies, to “best possible quality at very fast rendering times on cheap hardware”, e.g. games on mobile platforms. Several different approaches for creating and rendering these 3D environments have been proposed in the past few decades. Nowadays the two most common ones are rasterization and ray tracing. Both techniques have their own advantages and disadvantages and they both scale reasonably well, e.g. by changing the number of primitives or the resolution of the resulting image.

The processing power of modern computers has steadily increased and so has the image quality of virtual 3D environments. This can be easily seen in figure 1.1. But not only has processing power and quality kept increasing, the demands put on 3D environments are steadily increasing as well. Consumers keep demanding higher resolutions and more realism. Even though the advancements made in the past 20 years are quite significant, there never has been a time in which content developers have not been limited by the processing power of the hardware. This is especially true for real-time content like video games. This in turn means that there is demand for algorithms which can produce higher quality results in the same processing time or produce the same quality results in less processing time.

Rasterizing animated polygons can be done quite fast but certain effects like lighting or volumetric effects, e.g. fire and smoke, can be hard to implement and they can quickly slow down the renderer. Ray tracing, or ray casting, allows for an easier implementation of those effects because implementing ray reflection or “picking up information like smoke density along the way” is quite straight forward. But ray tracing usually requires more processing time. Especially when considering that fast ray tracing renderers rely on pre-computed data structures which make animations difficult or at least very expensive.

A hybrid approach which combines rasterization and ray tracing in order to gain the advantages of both may satisfy the aforementioned increasing demand of consumers and content creators. The idea is to render each object of a scene with a different technique depending on which one is more suited. For example the static parts of a


(a) Doom (1993)

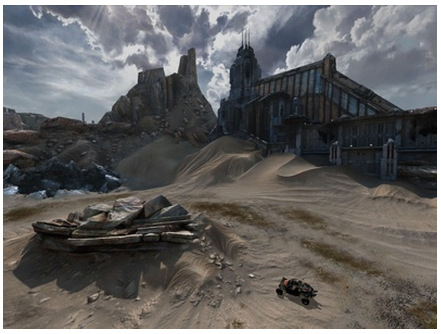

(b) Rage (2011)

Figure 1.1.: Progress of almost 20 years in 3D computer graphics. Both games are first person shooters developed by id Software. Doom is actually just pseudo-3D. Image sources: Wikipedia.[1] [2]

scene could be rendered using ray tracing with realistic lighting and volumetric effects like fog while all the dynamic, animated objects like characters are rendered into the ray traced image using rasterization. The difficulty lies within in the proper identification of which renderer to choose for which object and how to combine them. In the given example the ray tracing renderer would have to output a depth map in the proper value range for the rasterization renderer to use. Such a depth map is usually not necessary when using ray tracing.

The goal of this thesis to solve those issues and propose a renderer which combines rasterization and ray tracing in a way that improves upon renderers only relying on either technique. The final hybrid renderer should still be capable of real-time performance under modern circumstances. This means on modern hardware when rendering images of high quality, e.g. through the use of realistic lighting, the output resolution should be at least 720p and the framerate should not drop below 30 FPS.

1.2. Scope

The scope of this thesis is to explain the state-of-the-art in rasterization and ray tracing followed by a proposal of a new hybrid renderer and an evaluation of said renderer. The thesis is structured as follows. The first two chapters after the introduction are dedicated to rasterization and ray tracing. Each of those chapters explains the state-of-the-art of one these rendering techniques. The next chapters gives an overview of the related work done in the field of 3D computer graphics. This chapter is followed by three chapters discussing the newly proposed hybrid renderer. The first of those chapters discusses the theory behind said renderer while the next two chapters are about non-obvious implementation details and an evaluation of the performance and quality of the renderer. The thesis concludes with a short summary and suggestions for future work.

[1]http://en.wikipedia.org/wiki/Doom_(video_game)
[2]http://en.wikipedia.org/wiki/Rage_(video_game)

2. Rasterization

Before a new hybrid renderer combining rasterization and ray tracing can be discussed an overview over the two techniques is necessary. This chapter gives an introduction to 3D computer graphics and in particular to rasterization.

Computer graphics is a field of computer science that concerns itself with efficient algorithms and data structures to represent and create images or sequences of images. The methods used for three-dimensional objects and environments are especially important for this thesis. Rasterization is one such method. At its core it is a form of sampling of a continuous space.

It is common in computer graphics to use models which are actually wrong but which allow for efficient implementations while still delivering high image quality. The perception of light is one thing which is often modeled incorrectly for performance reasons. Humans visually perceive the world through light that travels into their eyes after being reflected off the environment. Cameras work the same way. A physically correct way to render a virtual 3D scene would be to simulate all the rays of light leaving every light sources until all rays have either been absorbed completely or left the scene. A virtual camera or eye could be placed into the scene and the rays hitting this object could be used to generate the final image. But doing so would be very computationally expensive. A large amount of rays would have to be simulated with only a small fraction ever contributing to the final image.

Computer graphics uses a model which might not be physically correct but which is a lot easier to compute. Basically a model of "vision rays" is used. Those rays do not start at light sources and eventually end in the virtual eye. Instead they start at the virtual eye and eventually end in the environment or maybe even a light source, depending on the complexity of the renderer. The starting point and direction of each ray defines which pixel of the final image it corresponds to while the end point of each ray, and sometimes its path as well, define the color of said pixel. This approach ensures that only rays which contribute to the final image are ever taking into consideration. This does not mean that all the rays which may contribute to the image are always accounted for. Some algorithms may terminate too early for performance reasons.

The remainder of this chapter discusses rasterization, which uses the aforementioned "vision ray" model implicitly, in more detail. At first the basics of how 3D objects are represented and drawn onto the screen are introduced. This is then followed by explanations of more advanced techniques which improve the image quality, e.g. how good looking animations can be implemented or how sampling artifacts (aliasing) can be reduced. The chapter concludes with a brief recap of rasterization to highlight the most important parts and how they work in combination with each other.

The topics discussed in this chapter are mostly standard text book knowledge. Basically any book with "Computer Graphics" in its title can be used for further reading. A quick look into the table of contents of a book should quickly reveal if it actually deals with rasterization and in how much detail it does. Three good examples are [23], [9] and [21]. There are many more good books on computer graphics. Which one is the best depends on the reader's interests and preferences. Also books about video game programming often contain a lot of useful information on computer graphics as graphics are an important part of modern video games.

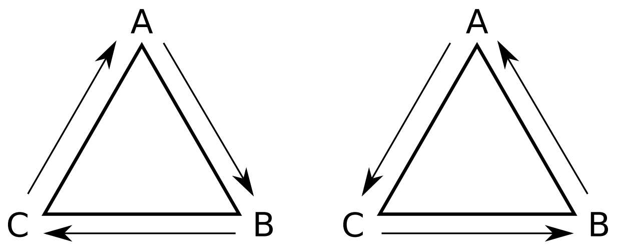
2.1. Basic approach

As previously mentioned rasterization is a form of sampling. A set of surfaces is used to define three-dimensional objects. Rasterization calculates how each surface looks from the point of view of a virtual eye or camera and then samples this transformed representation of each surface.

2.1.1. Polygons

The most common way to define a 3D surface, which shall be used in a rasterizer, is to use polygons. In particular triangles are used. They are the most simple, finite surface in a 3D space. This makes them easy to understand and so easy to work with both from the point of view of a user creating an object and from the point of view of a programmer implementing a rasterizer. Triangles are also the only type of surface supported by early 3D graphics acceleration hardware. This probably has contributed to their popularity in 3D modeling.

The most basic and common way to define a single triangle is to define the three vertices which are the corners of the triangle. A vertex is simply a point in 3D space, i.e. a three-dimensional vector. The order in which the vertices are defined is often used to distinguish between the two sides of a triangle.



(a) Order: A, B, C (b) Order: A, C, B

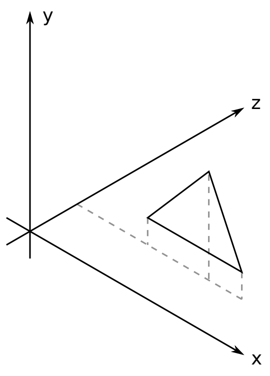
Figure 2.1.: Two triangles defined by the same vertices A, B and C but using a different order of vertices. Using the expression (second − first) × (third − first) to calculate the normals for each triangle results in normals facing in opposite directions. This would result in the normal $(B - A) \times (C - A)$ for the first triangle and $(C - A) \times (B - A) = -((B - A) \times (C - A))$ for the second triangle.

There are two possible normals which can be calculated for each triangle. These two normals define the same line but point into opposite directions. The order in which the vertices are specified can be used to decide on one of the normals as the "correct" one. The normal's direction can be used to decide which side of a triangle is the so-called front side and which one is the back side. A common convention is to say that if the normal points roughly toward the virtual eye the front side of the triangle is facing the eye. Figure 2.1 shows how normals may be calculated and how the order of the vertices matters for this.

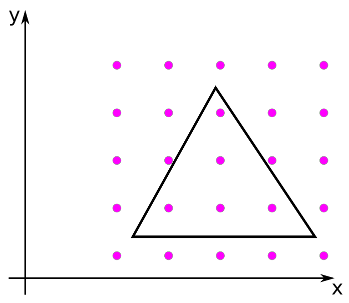
2.1.2. Rasterization

It has already been mentioned that rasterization is a form of sampling. This sampling is done on triangles which have already been transformed (e.g. translated and rotated) in such a way that their vertices are in the coordinate system of the virtual camera. In this coordinate system the Z axis points into the direction the camera is looking. The X axis corresponds to the horizontal axis of the output image and the Y axis corresponds to its vertical axis. A rectangular subsection of the XY plane, e.g. $[-1, 1] \times [-1, 1]$, is sampled.

The sampling rate depends on the resolution of the desired final image and is of course finite. Each sample point corresponds to exactly one pixel of the output image and the sample points have the same equidistant spacing as the pixels of the screen which is usually the final render target. Often the sample points are chosen such that they represent the center of a pixel. Figure 2.2 illustrates this.



(a) A triangle viewed in 3D. The Z value is the same for all vertices.



(b) Sampling of a triangle. Each purple circle is a sampling point and corresponds to exactly one pixel on the screen.

Figure 2.2.: The coordinate system used for rasterizing (sampling) triangles.

The triangles and the camera are defined using floating point numbers which are conceptually real numbers. This means that the exact subpixel position of the boundaries of each surface are arbitrary and so the "signal frequency" has no upper bound except for limitations imposed by the precision of floating point numbers. As a result the actual sampling rate is often below the minimum sampling rate for error-free reconstruction as defined by the Nyquist-Shannon sampling theorem. The artifacts this causes and how they can be dealt with is explained later in this chapter.

The actual sampling is implemented using a depth map. This is an image of the same resolution as the output image. But instead of color information depth information is stored in each pixel. Initially the depth is set to infinity for each pixel. The Z (depth) value of the image plane is also required for the algorithm. It works as follows:

1. initialize depth map
2. for each triangle:
    a) compute minimal bounding box of triangle
    b) for each sample point (X, Y) in the bounding box:
        i. test if it is inside the triangle, if it is not continue with next sample point
        ii. calculate the Z value of the triangle at (X, Y) and perform the Z test:
            - if the Z value is lower than the Z value of the image plane continue with the next sample point
            - if the Z value is larger than the Z value stored in the depth map for (X, Y) continue with the next sample point
        iii. store the Z value in the depth map in the pixel corresponding to (X, Y)
        iv. calculate the color of the triangle at (X, Y) and store it in the appropriate pixel in the output image

This algorithm ensures that for each pixel of the output image the front-most triangle, which is still behind the image plane from the point of view of the camera, is chosen. The properties of the triangle at that point determine the color of the pixel. The details of how that is done is explained in later sections of this chapter.

This algorithm calculates the orthogonal projection of all triangles onto the image plane with pixel-accurate depth checks so that overlapping triangles render correctly. It is also an implicit implementation of the “vision ray” model explained in the introduction of this chapter. A direct implementation would be to calculate a ray for each sampling whose direction is parallel to the Z axis. If the same Z tests are used to decide which triangle to choose to calculate the color, the resulting output image would be exactly the same as the one produced by the algorithm above.

2.1.3. Projective Geometry

The rasterization algorithm requires all the triangles to be in a specific coordinate system, namely one in which everything that is visible is by the virtual camera ends up inside a cube. Rasterization uses the projective geometry and its properties to transform triangles to said coordinate system.

Projective geometry is a subfield of geometry and so a subfield of mathematics. It concerns itself with the perspectively correct “drawing” of n-dimensional objects viewed from an $(n-1)$-dimensional space, e.g. the correct drawing of the 3D world as humans perceive it onto a 2D canvas.

While this is a vast field only some concepts are important for understanding rasterization. In particular homogeneous coordinates are important. When tracing a line from an eye or camera into a 3D space every point on said line will project onto the same 2D point in the image plane of the camera. Homogeneous coordinates are used to account for the phenomenon that an infinite number of points in the higher dimensional space represent the same point in the lower dimensional space. This is done by simply adding an additional dimension to each vector. For example in homogeneous coordinates the point (x', y') is represented by all vectors (x, y, z) for which $x' = \frac{x}{z}$ and $y' = \frac{y}{z}$ is true.

As a consequence all interesting transformations like translations or rotations can be expressed by matrices. For example

$$\begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

is a translation of the point (x, y) in homogeneous coordinates. This means that a rasterization-based renderer only needs to implement matrix-vector multiplication to cover all required transformations. Also an implementation of a matrix-matrix multiplication can be used the implement arbitrary combinations of transformations. When multiplying a matrix representing transformation A with a matrix representing transformation B the resulting matrix will represent a transformation that first applies B to a vector and then applies A to the result. Modern rasterizers, especially hardware accelerated implementations, use homogeneous coordinates for the 3D data representing objects and the environment. That means the vertices representing a triangle are actually represented by 4D vectors and 4×4 matrices are used to represent transformations on said vertices.

As initially mentioned the rasterization algorithms requires all triangles to be in a specific coordinate system or space. In total there are four relevant spaces:

- object space
- world space
- camera space
- screen space

Every object in a 3D scene is defined within its own coordinate system. This is a rather obvious requirement because each object is created separately and requires some coordinate system for storage of the vertices.

World space is the space in which all objects are placed. Transformations are used to calculate each object's position and orientation in world space. This implementation also allows to use the same object multiple times, e.g. to simulate a crowd of people, with low overhead. The object's vertices do not have to be stored multiples. Only the transformation parameters need to be stored per instance.

While world space already defines the scene properly it is not suited for direct rendering using rasterization. World space vertices are further transformed into the camera space. Translations and rotations are used so that Z axis becomes the direction the camera is looking in. Camera space is almost sufficient for the rasterization algorithm. The remaining issue is that every visible object is inside a frustrum, in particular a truncated pyramid as shown in figure 2.3. This means an orthogonal projection is not sufficient to account for all perspective-related phenomena, e.g. objects appear to get smaller as they move away from the camera.

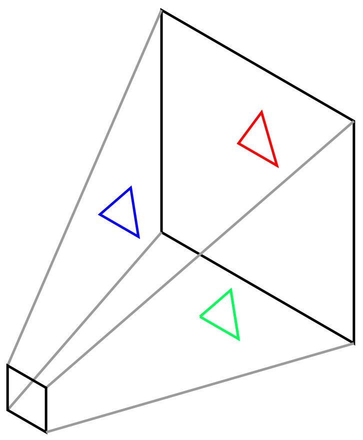


Figure 2.3.: A viewing frustrum with three different objects in it.

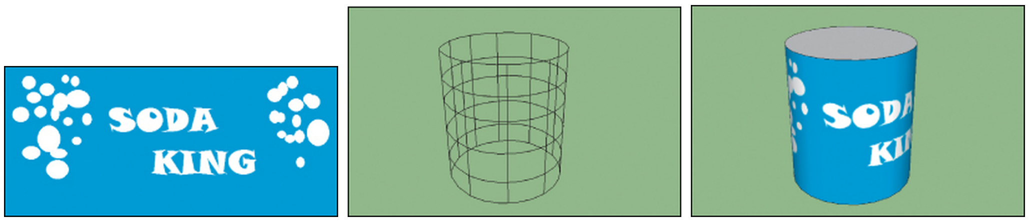
One last transformation is thus required namely a transformation that transform a frustrum into a cube. The resulting space after applying this transformation is called screen space. In this space the rasterization algorithm as described in the previous subsection can be used. The resulting image will look natural. The transformation from camera to screen space is also used to account for parameters like the field of view (FOV) (how much of the scene is visible) or to account for the aspect ratio of the screen that is used for display. When calculating the matrix representing the transformation the frustrum is set up such that it has the desired properties.

On overview of the relevant transformations and their matrix form can be found in the appendix.

2.1.4. Texture mapping

Rasterization uses the properties of triangles at the sampled points to determine the color of the pixel corresponding to each sample point. This is done through two related techniques namely texture mapping and shading. The former is used by the later as an input in modern renderers.

While triangles defined by vertices are sufficient to describe surfaces in terms of size and shape they do not define what the inside of a triangle actually looks like. This is done via texture mapping. Additional coordinates, so called texture coordinates, are assigned to each vertex. These coordinates specify a point in a 2D bitmap. When the rasterization algorithm samples a point from a triangle the texture coordinates of its vertices are used to interpolate the texture coordinates at the sample point. These coordinates are then used to retrieve a value from the texture (2D bitmap) associated with the triangle. Modern renderers allow the use of multiple textures to define different properties, e.g. diffuse color (color of an object under "good" or usual lighting conditions) or normal (to create the illusion of a more detailed model with little cost). An example of a wireframe model (polygon edges without any surface information), a diffuse color texture and a textured model is shown in figure 2.4.



(a) A diffuse color texture (b) A wireframe model (c) A textured soda can

Figure 2.4.: A soda can without and with a texture. Texture coordinates for each vertex (intersections in b)) are used to define a mapping of the 2D bitmap in a) to the model. The result is shown in c). This example has been taken from chapter 20 "Textures and Texture Mapping" in [23].

Retrieving the information from a bitmap can be done in different ways depending on the available processing power and desired image quality. These techniques are called texture filters. Each filter returns a value for each input texture coordinates (u, v). It reads a certain number of pixels from the textures depending on its complexity. u and v are floating point numbers. Neither the coordinates specified at the vertices nor the interpolated coordinates need to lie exactly on a pixel in the texture. They may actually specify points in between pixels.

The most simple filter is the nearest neighbor filter. It simply retrieves the pixel that is the closest to the texture coordinates. This usually creates a "blocky" look as the square nature of the pixels in the bitmap becomes visible as the camera gets close to objects. The more advanced bilinear filter interpolates between the four closest pixels giving higher weights to the closer pixels. It is called bilinear filter because it uses a linear interpolation horizontally and a linear interpolation vertically to calculate the final color. This results in a smoother image. An example of these two filter techniques is illustrated in figure 2.5.

Distracting visual artifacts may also occur if neighboring sample points during rasterization have texture coordinates that are too far apart. Information contained in the texture may be completely skipped because the texture filter never reads it for any of the rasterized points. To avoid this problem multiple resolutions are stored for each texture. This is called mip-mapping. The resolution of a texture almost always is a power of 2, usually they are even square. To create the lower resolution textures the resolution

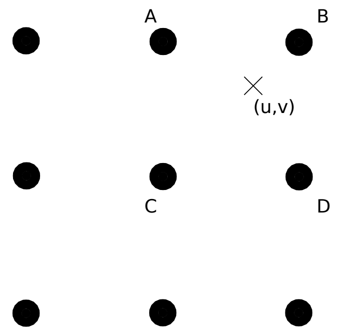


Figure 2.5.: Texture filtering. The black circles represent pixels in a 2D bitmap while the cross represent the coordinates for which a sample is requested. Nearest neighbor filtering would choose B as it is the closest pixel. Bilinear filtering would interpolate between A, B, C and D with the highest weight for B and the lowest weight for C.

is successively divided by 2 along each axis and the texture is downscaled to the new resolution in each step. This creates a texture pyramid as shown in figure 2.6. When using a nearest neighbor or a bilinear filter one image from the pyramid is chosen to use as an input for the filter. The choice depends on the depth of the sample point so that smaller textures are used for points that are further away.

The jump from one mip-mapping level (image in the pyramid) to the next creates a visible artifact. A flat surface that extends toward the horizon, e.g. the ground in an outdoor scene, shows a visible line at each jump of the mip-mapping level. The textures appear to get suddenly less detailed. To avoid this artifact a trilinear filter can be used. This filter is very similar to a bilinear filter. It chooses the two best fitting mip-map levels, performs a bilinear look up in each of those and then linearly interpolates between the results based on the depth. This creates a smooth transition between mip-map levels.

Surfaces whose normal is almost perpendicular to the viewing direction often appear blurry along one axis when using bilinear or trilinear filtering. This happens when surface has an aspect ratio in screen space that differs significantly from the aspect ratio of the texture. For example if a surface appears very wide but not very high in screen space yet the associated texture is square. This would result in a low resolution mip-map level to be chosen due to the low height. But this texture has a low resolution along each axis causing blur along the horizontal axis. One solution would be to create even more copies of the texture at different resolutions. But this quickly becomes too expensive in terms of memory cost. Instead anisotropic filters are used. They use even more than the eight samples of a texture which are used by trilinear filters. The additional samples are placed so that mip-map levels which are not direct neighbors are used so that a low resolution is used for one axis while a high resolution is used for the other axis. This can quickly cause memory bandwidth problems if too many samples are used. The issue is

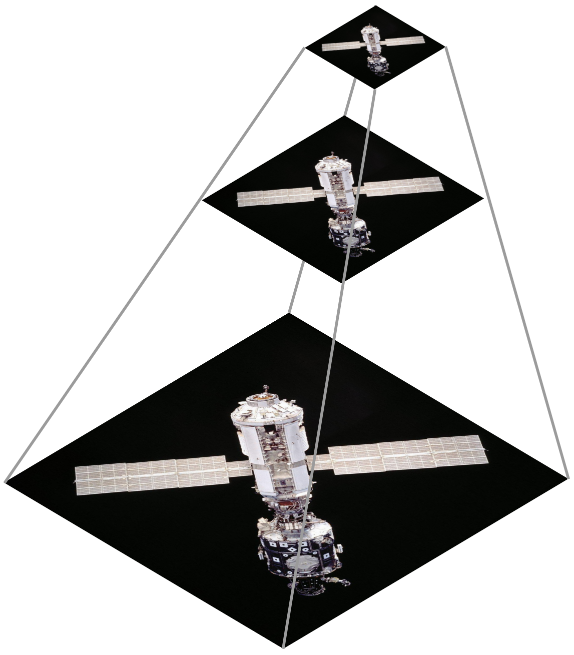


Figure 2.6.: A mip-map pyarmid. Each level decreases the width and the height of the texture by a factor of $\frac{1}{2}$. Using a texture of appropriate resolution avoids artifacts caused by sampling points being spread out too far across the texture. Image source: Wikimedia.[3]

illustrated in figure 2.7.

2.1.5. Shading

The second technique which is used to determine a pixel's color is shading. The exact meaning of the term "shading" has shifted somewhat in recent years. It used to solely refer to techniques used to calculate how much light is reflected by a surface at a given point. This amount of light was then used to modify the color information retrieved from a texture to brighten or darken each pixel individually. Nowadays shading refers to the more general process of determining a pixels color after sampling it from a triangle. Shading is done through small programs that are executed for each pixel individually. These programs have access to information like the interpolated texture coordinates and use this to calculate a final color. The filtered texture reads as described in the previous section are actually triggered by these shader programs. The filtered values read this way are then used in the programs calculations. But shader programs do not have use textures. They may also procedurally generate content on-the-fly.

The most common approaches to calculate the amount of light reflected use the normal

[3]http://commons.wikimedia.org/wiki/File:ISS_from_Atlantis_-_Sts101-714-016.jpg

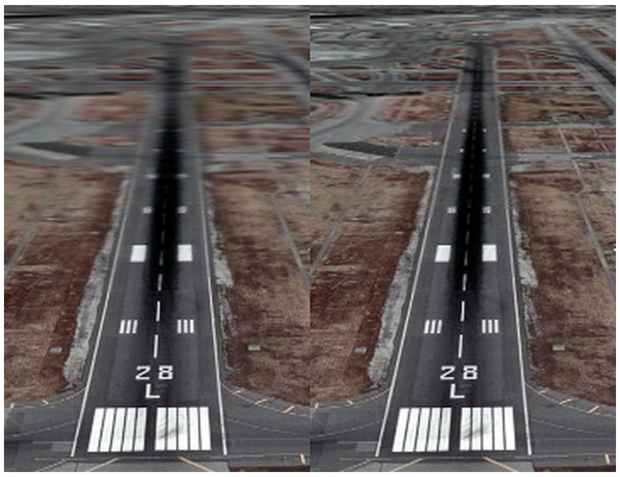


Figure 2.7.: Two pictures of the same runway being viewed in Google Earth. The picture on the left uses trilinear filtering and starts to lose details as lower mip-map levels get chosen for the parts that are further away. The picture on the right uses anisotropic filtering and so preserves more detail. Image source: Wikipedia.[4]

of a surface at a given point and the direction a point light source is in. Often a simple scalar product of these two vectors is used to approximate how much light is reflected towards the camera at that point. The most simple implementation, flat shading, uses the actual normal of each triangle and so calculates how much light is reflected for all points of that triangle. This highlights the boundaries of triangles as the lighting changes in sudden steps. More advanced techniques are Gouraud shading and Phong shading. Gouraud shading uses normals for each vertex to calculate the amount of reflected light at each vertex. Similarly to texture coordinates these amounts are then used to interpolate amounts of light at each pixel. Phong shading uses a similar setup but computes interpolated normals for each pixel and then calculates the amount of reflected light for each pixel in this way. Each of those lighting models usually adds a small constant light value to each pixel to account for the ambient light, e.g. the light provided by the sun. Phong shading usually creates the best results especially for models with a low number of polygons. But it is also the computationally most expensive technique.

Modern rasterization renderers often combine several texture reads to compute the color of a pixel. In addition to textures storing the diffuse color of a surface there are two very common types of textures:

- light maps
- normal maps

[4]`http://en.wikipedia.org/wiki/Anisotropic_filtering`

Light maps store the amount of light reflected at a given point. These are useful for static scenes in which light can be pre-calculated once. Integrating this information directly into the texture storing the color information often is not an option as diffuse color textures may be reused. For example a brick wall may be textured using a small brick texture that is repeated over and over again, yet the light hitting the may not repeat in the same fashion. Normal maps store the normal at each point of a surface. The advantage of normal maps is that they allow models to appear more detailed while being less computationally expensive to use than models with more polygons. Small details like scratches or scars can be produced by normal maps in a convincing way.

Shading programs are also used to procedurally compute content. For example when rendering quiet bodies of water with only very shallow waves. Those bodies of water are modeled as a single flat surface. Shader programs which implement some kind of wave equation are used to calculate colors and normals for each sampled pixel. Procedurally generated content requires only few parameters to be stored in memory.

2.2. Hardware acceleration

All modern computers for personal use have a so called Graphics Processing Unit (GPU). This processor is designed to perform graphics-related computations quickly. Due to the popularity of rasterization-based 3D graphics all modern GPUs have dedicated hardware for those type of graphics. Hardware-based solutions for fast rasterization became popular in 1996 when 3dfx Interactive launched their Voodoo Graphics GPU. These GPUs were available as Peripheral Component Interconnect (PCI) cards user could install in their Personal Computer (PC). They had no support for 2D graphics which means they had to be installed in addition to a regular GPU. The first 3D GPUs only supported a subset of all the algorithms and techniques necessary for rasterization. For example the Voodoo Graphics GPUs could perform texture mapping but transformations between coordinate system had to be performed by the CPU in advance. In 1999 nVidia released the GeForce 256 which featured support for the most common rasterization-related algorithms. One feature it added was hardware-based transformations which previous GPUs could not do. They still had one major drawback, namely the limitations on their feature set. These GPUs offered a fixed set of features which programmers could only enable or disable but not change. This drawback has been negated more and more by the introduction of shaders. These are basically small programs written by software developers specifically for the GPU. These shaders are able to peform tasks like executing transformation, triggering filtered texture look-ups or peforming shading. In 2005 the first almost fully programmable consumer GPU was available in the Xbox 360. Only the sampling of triangles is still performed by a piece of fixed function hardware while all prior and all subsequent steps are performed by fully programmable shaders. The following year this GPU design had been adopted by GPUs available for PCs. Continuing improvements in the area of shaders have even let to GPUs being used for non-graphics tasks as the processing units used for executing shader code have become general purpose processing units.

2.3. Advanced techniques

This section explains several advanced rasterization techniques which improve performance or image quality. They range from using the basic algorithms in clever ways over using general image processing algorithms to adding entirely new algorithms.

2.3.1. Optimizations

While the basic rasterization algorithms already ensure that only the visible polygons contribute to the final image, always processing an entire scene, even parts that are not visible, decreases performance. Transforming polygons which are not visible still takes processing time that could be used more efficiently elsewhere. This is particularly important on modern GPUs which have one large pool of processing units for all shader code instead dedicated fixed hardware units for each step.

In order to decide which polygons are actually processed by the GPU spatial partitioning can be used. This means nothing more than subdividing a scene into regions and deciding which regions might contain visible content. Only those regions are then processed further. There is a wide range of spatial partitioning techniques. A simple, manual approach is to let content creators define "rooms" and define which rooms are connected. Only the room in which the camera is located and all adjacent rooms are processed. This of course requires the rooms to be build in such a way that the camera can never look through an adjacent room into yet another room.

More advanced spatial partitioning techniques automatically subdivide scenes without the need of content creators doing any manual work in addition to creating the scene itself. They usually use a tree-based data structure which can be traversed quickly. Common techniques are Binary Space Partitioning (BSP) trees and Axis-aligned Bounding Boxes (AABB) trees. The intersection of the view frustrum of the camera and the scene can be quickly calculated using such trees. The resulting set of polygons is then processed further. BSP trees are binary trees which use planes to subdivide a scene. Every inner node of the tree contains a plane and pointers to the subtrees on either side of the plane. The leaf nodes contain the actual content of the scene. AABB trees enclose everything in a scene in minimal volume boxes whose sides are parallel the planes spanned by the axis. This allows for fast testing whether any giving point is inside such a bounding box or whether a volume intersects such a box. The leaf nodes may contain single primitives, e.g. one single polygon, or small objects which do not need to be subdivided further like the body parts (legs, feet, arms, hands, etc.) of a person. Moving up the tree towards the root node are increasingly large bounding boxes enclosing more and more objects. Each inner node has two children. One child is the subtree of everything contained in the bounding box of the node while the other child is the subtree containing everything outside of this box.

In the case of games or physics simulations spatial partitioning also helps with detecting collisions of objects with the scene as queries whether an objects overlaps with anything in the scene can be processed quickly.

Many 3D environments are designed to only contain objects which are either always facing the camera or have a thickness that is strictly greater than 0. In such environments the backside of a polygon can never be visible. This means that the normal of a polygon and the viewing direction of the camera can be used to quickly decide whether said polygon might be visible. This technique of terminating the processing of a polygon early depending on the direction it is facing is called backface culling.

2.3.2. Animation

Animations are an important aspect of many applications using 3D environment. Very few scenes do not have any moving content. And once there is at least one moving object some kind of animation system is necessary. Animating objects which do not deform during their animation is possible by simply updating the transformation parameters which transform those objects from their object space to the world space. This allows for those objects to move around and rotate freely.

Deforming objects, e.g. people moving their extremities, are harder to implement. This is usually done through a keyframe-based system. All required poses for each such object are stored together with how long it takes to get into the next pose. Each time a frame is rendered the animation system interpolates between the previous pose and the next one to determine what the object currently looks like. Different parts of an object may be animated independently from each other. For example the upper body of a soldier holding a gun may be animated independently from his legs to allow for the soldier to aim freely even while moving in any direction.

Storing where each vertex is for each keyframe quickly creates a lot of data. To reduce the amount of data needing to be stored skeletal animation is often used. Each animated object has a skeleton. A skeleton simply is a set of connected bones. This can be imagined like a graph with the edges being the bones and vertices of the graph being the joints. These bones are often in a hierarchy affecting each other just like the hand of a person moves around when said person moves her arm. The polygons making up the object are assigned to the bones. Each polygon may be assigned to multiple bones with varying weights. Instead of storing all polygons for each pose, the polygons need only to be stored once while only the skeleton is stored for each pose.

Modern animation systems do not store the actual location and angles of the skeleton for each keyframe but instead store transformation parameters which are required to reach said keyframe. This makes it easier and more efficient to calculate the necessary interpolated transformation parameters at runtime. There are two ways to determine the transformation parameters which get stored. The first and less common is forward kinematics. This means that the content creator directly specifies each transformation parameter. While this gives designers a lot of control it is actually quite tedious and hard to use for complex movements. The second and more common approach is inverse kinematics. Here the designer moves the skeleton in the desired pose and the animation system automatically calculates the required transformation parameters.

To store rotations in particular quaternions have become quite popular. Quaternions are very similar to complex numbers. But instead of just one imaginary part they have three “imaginary” parts and so span a 4D space instead of the plane spanned by complex numbers. Like other concepts borrowed from mathematics this has become quite a vast field of its own. The important characteristic, which makes them interesting in computer graphics, is that rotations defined by quaternions are very numerically stable. In particular interpolations between two rotations can easily be calculated by interpreting two quaternions, each representing a rotation axis and an angle, as 4D vectors and then linearly interpolating between those two vectors. This generally gives good results at a low computation cost.

2.3.3. Lighting

The proper behavior of light in a scene or at least the illusion of proper behavior adds significantly to the quality of a rendered image. Fully simulating how light spreads out from a given light source is computationally expensive and usually reserved to scenarios like rendering the effects in a movie. Calculating this so called global illumination takes too long for real-time applications. In those cases lighting is usually limited to light rays which are reflected only once (source to object to camera) or twice (source to object to other object to camera). Calculating only this direct lighting without all the indirect lighting, like the light from the sun which can bounce of many surfaces before actually being captured by an eye or camera, is calculating the local illumination. As a trade-off between quality and performance many real-time applications calculate local illumination to a certain degree and add a constant amount of light called ambient light to give the illusion of global illumination.

A more sophisticated illusion of global illumination can be achieved by pre-calculating how the light spreads and storing this information in textures. These so called light maps can then be used during shading to determine the ambient light term added to the light calculated via local illumination. Light maps work well as long as there are not too many moving objects in a scene. Every moving object may affect the light maps, especially moving light sources. These dynamic lights are practically always rendered using local illumination techniques.

Local illumination is calculated during the shading step. One approach is to calculate the distance and direction of the n closest light sources for each pixel being shaded. This information can then be used to calculate the sum of the contribution of each light source to said pixel. Choosing a fixed number of light sources to account for was necessary in early rasterization renderers as the instructions available in shader programs was quite limited. Even today choosing a different n for each pixel may affect performance negatively due to the architecture of modern GPUs. A more advanced implementation may even perform a query to check if the line between the pixel being shaded and the light source intersects with any geometry. This helps creating convincing shadows but is rather expensive to compute.

In recent years deferred shading and deferred lighting have become popular. The important ideas and concepts of deferred rendering algorithms actually date back to 1988 and 1990 ([7], [20]). Those papers do not use the term “deferred” but they describe the same processes that are described in this paragraph. Deferred shading and deferred lighting use multiple passes to create the final image. The rough algorithm of deferred shading is to render the scene but instead of creating an image which gets send to the display the output gets written to one or more textures. These textures do not contain the final color information but information used to calculate said color like the normal, the diffuse color, etc. After computing this information each light is rendered as a 2D or 3D shape depending on the kind of light. In this lighting pass the information in the textures created in the previous pass is used to calculate how much light a given source contributes to each pixel. This information is then accumulated in yet another buffer which eventually becomes the final image. Deferred shading allows for rendering a relatively large amount of dynamic lights at the cost of requiring a lot of memory. Deferred lighting is similar yet requires less memory at the cost of required an additional pass which processes the geometry of the scene again. Due to having become popular only recently these techniques may still be hard to find in Computer Graphics textbooks. Two good books which each contain a detailed chapter on deferred shading are [19] and [17]. Both use an actual video game and its renderer as an example to explain deferred shading in practice. Both also contain helpful references of further material on the topic.

A popular trick to implement reflective surfaces like mirrors or water surfaces is to use environment maps. These can be imagined like a cube surrounding the reflective surface. Only the inner sides of this cube are textured. The texture for each side is created by rendering the scene from the center of the cube facing said side. The cube itself is never rendered. When the reflective surface is rendered the normal of each pixel is used to determine which side of this imaginary cube would be hit by a ray from the camera being reflected by the surface. This reflected ray is also used for a look-up in the texture belonging to that side of the cube. With this information convincing reflections of static geometry can be created efficiently.

2.3.4. Volumetric effects

Rendering volumetric effects like smoke or fog efficiently and convincing is still a problem for rasterization-based renderers. A trick for rendering fog is to use the Z value during shading to implement a linear interpolation between the color of each pixel and the desired color of the fog. Objects further away appear to be covered by increasingly strong fog. Early games used this effect to limit the viewing distance to improve performance as fewer objects needed to be rendered. The only drawback of this trick is that it is not entirely physically accurate. The screen space depth does not reflect the distance in world space. Pixels near the border of sampled screen space are actually further away from the camera than pixels near the center of the sampled space with the same depth in screen space. This means that some objects may become visible by simply turning the camera (so that the object moves from the center of the screen to its edge) even though the distance between object and camera never changed.

More general volumetric effects like smoke require the use of so called “billboards”. These are simply rectangular polygons, usually rendered by two triangles, which by definition always face the camera. To render smoke the texture of such a billboard is actually a sequence of textures showing the smoke itself. Billboards work well enough in some situations like static cameras or objects which are very far away and hardly move.

But once the camera moves or is close to a billboard the illusion of a volumetric effect quickly breaks down as it becomes apparent that a sequence of 2D images that always faces the camera is being rendered.

Modern GPUs also support 3D textures. These are simply texture with several layers. Each layer has the same size. Volumetric effects may simply be stored as a sequence of 3D textures. During rendering multiple billboards intersecting the cube spanned by the 3D texture at increasing depths are used to render the content the 3D texture. The texture for each billboard is determined by sampling from the 3D texture. While this creates much better results it comes at a high memory cost and at a high performance cost. 3D textures are large due to their many layers and rendering multiple billboards and blending them together requires a lot of bandwidth and many operations.

2.3.5. Anti-aliasing

As previously mentioned the rasterization algorithm is a form of sampling. The frequency of the “signal” (polygons) being sampled has no upper bound. This means that aliasing artifacts may potentially occur. This is actually quite common. These appear at the boundaries of objects as the switch from polygon to the next when stepping from one pixel to an adjacent one is the source of the high frequencies which are not sampled correctly. Aliasing artifacts look like the pixels, which define the border of an object, form a staircase instead of a smooth line.

One approach to dealing with this problem is to use super-sampling. This means rendering the image at a higher resolution than is required for the final output to the display. The large rendered image is downscaled to the output resolution before display. This causes every pixel on screen to depend on several samples created during rasterization, hence the term super-sampling. Full-scene anti-aliasing (FSAA) is an implementation of such super-sampling. FSAA implementations usually allow the user to choose the amount of super-sampling via a factor. This factor is the number of samples being taken for each pixel. The final pixel color is the average of the color of all samples for that pixel. An example of FSAA 2x is shown in figure 2.8.

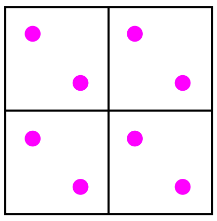


Figure 2.8.: A 2×2 pixel grid with two sample points (purple) per pixel.

FSAA has a high performance and memory cost though as each image is rendered at a higher resolution. Multisample anti-aliasing (MSAA) is an attempt to reduce the performance cost of FSAA while still providing similar quality. MSAA still uses multiple sample points per pixel with independent depth tests. Yet for each pixel and polygon, for which at least one sample point passes the depth test, shading is done only once usually for a point in the center of the pixel. The color computed this way is then copied to all the sample points which passed the depth test. While successfully reducing the performance cost by doing the shading for fewer sample points the memory cost still stays the same. For each sample the color and depth information is stored separately.

Simply averaging the color of all sample points implies that each of the n sample points per pixel contributes $\frac{1}{n}$th to the final color of the pixel. Large n produce better results as the area of a pixel covered by each polygon is approximated more precisely. This can be seen in figure 2.9.

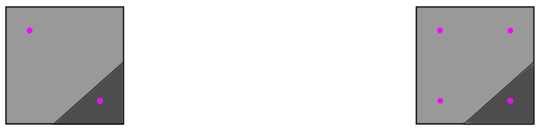

(a) Two sample points (b) Four sample points

Figure 2.9.: A pixel partially covered by a light gray and a dark gray polygon. Using more sample points estimates the area covered by each polygon more accurately.

Yet a large n also increases the memory cost substantially. In 2006 nVidia introduced Coverage Sampling anti-aliasing (CSAA) which allows for a good approximation of the area covered by each polygon without using quite as much memory ([18]). CSAA introduces a small color table for each pixel. This table has fewer entries than the number of samples per pixel. Each sample stores the index of a row of the color table. This reduces the memory cost per sample to a small integer (index) and the depth information. Even when accounting for the color table for each pixel the memory cost per pixel is still reduced compared to MSAA. An example configuration is a color table with four entries and 16 sampling points, both per pixel. As it is unlikely that more than four polygons overlap within the area of one pixel (with each polygon being the front-most one for at least one sample point) this configuration would produce almost identical results to MSAA with 16 sampling points. Yet 12 fewer color vectors would have to be stored per pixel. This reduces the memory cost and also the memory bandwidth cost. The later might translate into a performance improvement depending on the GPU and scene being rendered.

Despite these optimizations super-sampling is still quite expensive. Especially when combined with deferred shading which already requires a lot of memory. This has made image processing based anti-aliasing algorithms popular in recent years. These algorithms usually do not rely on any information created during rasterization beside the rendered image so that they can be applied to as many scenarios as possible. For example nVidia's Fast approximate anti-aliasing (FXAA) uses edge detection combined with blurring to find aliasing artifacts and hide them ([12]). This usually results in the entire image becoming slightly blurry.

2.3.6. Screen space effects

Anti-aliasing is not the only area in which information from screen space is used together with image processing algorithms to improve image quality. Two common techniques from cinematography implemented this way are depth blur and motion blur.

In movies everything but actors and objects at a certain distance to the camera are sometimes intentionally blurred to shift the audience's focus to the unblurred regions. To simulate this effect any image blur algorithm which allows for a per-pixel blur strength adjustment may be used. The blur strength of each pixel is chosen as a function depending on the distance to a pre-defined depth. This causes everything but objects at the desired depth to become blurry.

Motion in movies is blurry due to technological reasons. Real cameras do not capture exact points in time but short time intervals (the time the shutter is open). To simulate this effect, after rasterization each object may be blurred in the direction of the motion of said object from the point of view of the virtual camera.

Another screen space effect is so called Screen space ambient occlusion (SSAO). Ambient occlusion algorithms attempt to determine if an object or point is occluded from light sources due to its surroundings. SSAO uses the normals of each pixel to approximate ambient occlusion and makes occluded pixels appear darker. This enhances image quality by giving the viewer more depth cues. Like deferred shading SSAO has only achieved high popularity fairly recently. As such information about it is hard to find in textbooks. [5] provides a good starting point into the topic.

2.4. Recap

This chapter explained the first of the two techniques which shall be combined into a new hybrid renderer. Rasterization is based on polygons. Each object is defined as a set of polygons in its own coordinate system called object space. Transformations are used to determine each polygons position and facing in world space in which all objects co-exist. World space coordinates are then transformed to camera space and further into screen space. In screen space the polygons are sampled into a 2D image. Each pixel is shaded by a small program calculating the amount and color of the light reflected by the polygon covering that pixel. Bitmaps called textures are used as a source of varying information like the diffuse color or the normal of a polygon at different points.

3. Ray tracing

This is the second chapter discussing the fundamentals of 3D computer graphics which are necessary for implementing the hybrid renderer. It focuses on ray tracing. Ray tracing is a technique that is very similar to rasterization, especially when it comes to the implementation of some of the advanced techniques. For example the algorithms used for anti-aliasing or for screen space effects work pretty much the same way.

The significant difference lies in the way the "vision ray" model introduced in the previous chapter is being implemented. Rasterization uses transformations between several spaces and computes an orthogonal projection in the end. Ray tracing only needs world space, and object space if instancing a model several times is required, and actually traces the path of each vision ray directly in this space.

This chapter is structured similar as the rasterization chapter. It starts with a discussion of the basic ray tracing algorithm and primitives, e.g. polygons and voxels, suitable for said algorithm. This is followed by a short overview of hardware capable of fast ray tracing. Next is a discussion of more advanced techniques, which includes a comparison to rasterization and how many effects can be implemented similarly. The chapter concludes with a short recap of the most important points.

Just like the previous chapter the topics discussed here are mostly standard text book knowledge. Finding a good text book is a little bit harder though. Many books on computer graphics, e.g. [23] and [21], include chapters on ray tracing as well as rasterization but ray tracing is not quite as commonly discussed in text books as is rasterization. As with rasterization the best book on ray tracing depends on the reader's preferences and a quick look into the table of contents of a computer graphics book should reveal if and how detailed ray tracing is discussed. Some books use the term ray casting instead of ray tracing.

3.1. Basic approach

3.1.1. Ray tracing

Ray tracing is a much more direct implemenation of the "vision ray" model. It does not need to concern itself with coordinate systems and spaces as much. Unless instancing, using the same object over and over again while only varying some parameters such as its position and orientation, is used one space, namely world space, is sufficient. In this space a virtual camera is defined by defining its location, orientation and parameters for setting up the image plane. These parameters are basically the same as those necessary for computing the final transformation matrix which transforms from camera space to screen space. With the help of aspect ratio, field of view and distance to the camera a

```
Color castRay(Vector start, Vector direction):
  Float lambda = infinity
  Object nearest = nil

  for each Object o in scene:
    objectLambda = intersect(o, start, direction)
    if objectLambda >= imagePlaneDistance and objectLambda < lambda
      nearest = o
      lambda = objectLambda
    end if
  end for

  return computeShade(nearest, start, direction, lambda)
end
```

Figure 3.1.: Pseudo-code implementation of ray tracing. *intersect* has to return values smaller than *imagePlaneDistance* in the case the ray defined by *start* and *direction* does not intersect *o* for the correct result to be computed.

rectangular plane perpendicular to the viewing direction of the camera can be calculated. The location of the camera and equidistant points in said plane are used to define rays. The amount and layout of these points is the same as the pixels in the final output. This means there is one point, and thus ray, for each pixel. For each pixel a *castRay* method is called to determine its color. The pseudo-code for this method is shown in figure 3.1.

This is in so far similar to rasterization as it determines the closest object for each pixel uses that for shading. The difference is that the outer-most loop, which is not shown in this pseudo-code, iterates over all pixels and tests all objects for each pixel in the case of ray tracing. Rasterization iterates over all objects and tests all relevant pixels for each object.

As ray tracing follows the path of each ray starting at the camera's location traveling through the image plane to the first object being hit, it is a direct implemenation of the "vision ray" model. This is illustrated in figure 3.2.

3.1.2. Primitives

The pseudo-code in figure 3.1 does not make any assumptions about the kind of objects being tested other than it being possible to test whether that object intersects a ray. This means that any geometric object for which a ray-intersection test can be implemented can be used for ray tracing.

Planar polygons, including triangles as they are used for rasterization, can be easily used. The intersection test has to find out in which point the ray intersects the plane in which the polygon lies and then test whether this point is inside the finite region defined by the polygon. Any algorithm that requires texture coordinates can be implemented in

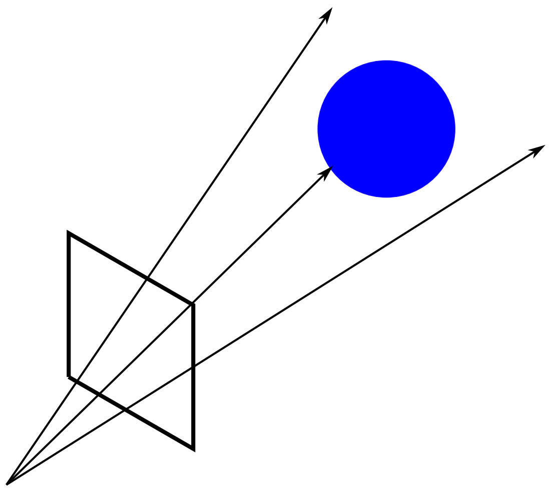


Figure 3.2.: For each pixel a ray is sent from a common starting point through the image plane (at the location corresponding to the pixel) into the scene. Each pixel's final color depends on which object it hits and where said object is hit.

a ray tracing renderer just as easy as it can be implemented in a rasterization renderer. The intersection test simply takes care of calculating the interpolated texture coordinates once a ray is found to intersect a polygon. From that point on everything works the same as it does in the case of rasterization.

One particular strength of ray tracing is that it is quite easy to implement intersection tests for curved surfaces such as spheres. Rasterization renderers require these kinds of surfaces to be approximated with triangles. Testing whether a ray intersects a sphere can be implemented easily allowing for an efficient implementation of actual spheres instead of approximated spheres. Any kind of curved surface for which an efficient ray-intersection test is known can be used easily as a primitive to build objects or entire scenes.

When modeling an object the designer sometimes subtracts one primitive from another one, e.g. subtract a sphere from a thin plate to create a plate with a hole in it. While rasterization relies on approximating the result with as many triangles as necessary, ray tracing renderers can perform such operations efficiently in real time. When a ray hits the plate the renderer can check if it is also inside the sphere. If so, the ray is treated as if it does not intersect the plate. This reduces the need for approximation further and allows ray tracing to use models which look more realistic.

One kind of primitive which has become popular are voxels. These are discussed in their own section.

3.1.3. Voxels

Voxels (volume elements) are a natural extension of pixels (picture elements) into a 3D space. While a pixel represents a finite area in a 2D space such as a picture, a voxel represents a finite volume in a 3D space. Voxels share many properties with their 2D counterpart. The most common shape used for voxels are cubes just like pixels are very often squares. Technically a location, size and color/shading parameters are necessary to fully define a voxel. To reduce the amount of information which needs to be stored, voxels are often arranged in a 3D grid just like pixels are stored in 2D grids. All grid cells have the same fixed size and the location is implicitly stored by defining an order in which the grid cells are stored. The shading information for a given location can by found in memory by calculating its address based on the predefined order, just like the address of the color information for a given pixel location can be calculated.

Many image processing filters can be naturally extended to 3D data. For example a gaussian blur filter can be easily extended to more dimension because it is separable, which means that it can be applied as a series of 1D filters. Also the output from 3D scanners can be turned into voxel information easily. 3D scanners usually output "point clouds". Sampling these into a 3D voxel grid is easier than trying to extract surface information (polygons). This makes it easier to create realistic looking objects as existing objects can easily be scanned and rendered using voxels.

3D imaging based on voxels is particularly popular for medial applications. In this field they have been studied and used for many years already. Their popularity comes from the properties described above. Voxel data is easy to acquire and process.

Unfortunately voxel data requires a lot of memory especially at high resolutions. Also rendering can be unnecessarily costly if there is a lot of empty space (grid cells which are marked as empty). To minimize these two issues voxels are often put into a spatial partitioning data structure as a pre-processing step before rendering when possible. A popular data structure is the so called Sparse Voxel Octree (SVO). SVOs partition a 3D space by defining three planes in each inner node. For each plane spanned by a pair of basis vectors there is one dividing plane parallel to it. These three planes subdivide the space represented by the inner node into eight disjoint subspaces. While the root node represents the entire 3D space each inner node only represents a fraction of it. Each leaf node then represent a distinct and unique subset of that space. Each node can be interpreted as a voxel whose location and size is implicitly defined by the tree's structure. Often not only the leaf nodes but also the inner nodes store shading information. The shading information in inner nodes is averaged from the information stored in their subtrees. By doing so the SVO stores the same information at different resolutions with each depth level representing a different resolution. SVOs also discard all leaf nodes which are unreachable from the outside and so are never visible, i.e. the inside of objects stored in a SVO is usually hollow. It is sufficient to store one single 3D point to define all three dividing planes, namely the "center point" in which all three planes intersect. This precisely defines the location of all three planes.

SVOs are not balanced. Empty space causes the tree to terminate into a leaf node early. This is actually an advantage because the shallow depth allows empty space to be traversed quickly during rendering. It also reduces the amount of data which needs to be stored. Combined with the discarding of invisible voxels SVOs are quite efficient in terms of storage and rendering costs. Their main disadvantage is that they require a costly pre-processing step before rendering which can not be performed in real-time.

3.2. Hardware acceleration

Modern rasterization implementations are backed by powerful hardware to perform each task quickly. There are several proposals for hardware which does the same for ray tracing. For example Woop et al. from the Saarland University have proposed a programmable Ray Processing Unit (RPU) which can perform ray-primitive intersection tests efficiently and execute shader programs for each ray hitting a primitive ([24]). While their results are promising especially considering the low clock rate of their implementation (as a result of the hardware they used for testing) no hardware implementation for accelerating ray tracing has become widely adopted yet. With the popularity of rasterization and the continued improvements in that area there was little need to switch to ray tracing in recent years.

Implementing ray tracing on GPUs meant for rasterization has become more and more common in recent years. Due to their high programmability modern GPUs have become general purpose processing units which are suited for a variety of tasks. Ray tracing is often implemented by sending a four-sided polygon to the GPU which covers the entire screen. The programs responsible for shading then perform all the ray-primitive intersection tests and by doing so implement ray tracing on a GPU designed for rasterization. While this is not the most efficient use of the available processing power of a modern GPU it works well enough for real-time applications.

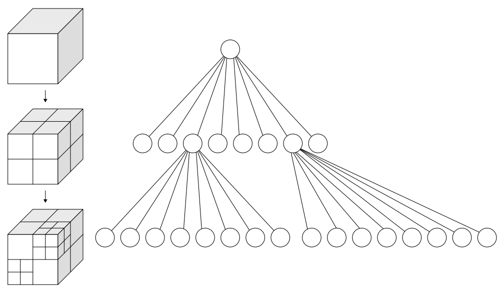
3.3. Advanced techniques

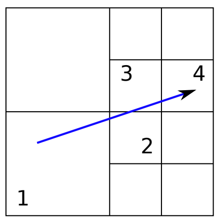
Many of the algorithms used for rasterization can also be used for ray tracing. Algorithms which are based on image processing like filter-based anti-aliasing or screen space effects can be implemented just as easily for ray tracing. Also super-sampling-based anti-aliasing works the same way, as does texture mapping when using polygons as primitives. Spatial partitioning also works similarly except that it is even more important in order to get good performance. This makes animations harder to implement. The animated primitives are no longer in an optimized data structure once their new position and orientation has been calculated. This slows down rendering.

3.3.1. Spatial partitioning

The chapter on rasterization already included an overview of spatial partitioning techniques. Due to the central importance of spatial partitioning for ray tracing, in particular when using voxels and SVOs, this section discusses tree traversal to find the first object hit by a ray in more detail. As an example, algorithms for finding the first voxel hit by a ray by traversing a SVO are described. Figure 3.3a shows an example of an octree in general, while figure 3.3b uses a quadtree, a 2D octree, to illustrate traversal through such a tree structure.

Finding the leaf node in an octree which contains a certain point is rather easy. The three separating planes at each non-leaf node are used to determine in which octant the point is. The traversal then proceeds with the child for that particular octant. This is


(a) An octree of depth 2. The left shows an increasingly subdivided spatial representation of the octree while the right-hand side shows the associated tree and its levels.



(b) A quadtree and a blue ray traversing it. At first the leaf node 1 is visited and then the traversal proceeds on to leaf node 4 in the order shown. Before nodes 2 and 3 are visited their parents, each representing an entire quadrant, are visited by the algorithm.

Figure 3.3.: An octree and a quadtree. These are data structures commonly used for spatial partitioning in particular when using voxels. The examples always use the center of the current square or cube to subdivide the space. In general this restriction is not necessary. Source of the octree picture: Wikipedia.[5]

repeated until a leaf node is reached. To find the first voxel hit by a ray the SVO has to be traversed to the leaf node containing the starting point of the ray.

Once this node is reached the algorithm traverses to leaf nodes along the path of the ray until one is encountered which is not marked as empty. If it is transparent, shading information is accumulated for use in the final shading process later and traversal continues. If it is opaque, the final shading is performed using all the information accumulated along the path and the shading information stored in the opaque voxel.

Finding the next leaf node along the path works by finding the first of the three separating planes which the ray intersects. Each plane can be viewed as a passage to a particular sibling node. If this sibling is not a leaf node, the entry point into that siblings space can be used to traverse down the tree to the next leaf node. Also a check whether the ray leaves the space represent by the current parent node is necessary. If so, the algorithm has to traverse one level up the tree and proceed to identify the proper sibling at that level. This traversal up the tree may happen multiple times in succession.

There are several ways to implement the traversal to the siblings. This is also the key difference of the different implementations of SVOs. Some implementations store parent pointers and maybe even sibling pointers in addition to child pointers to be able to traverse directly as described above. Other implementations update the starting position of the ray or use a stack to remember the last path taken from the root node to a leaf node. This information is then used after restarting the traversal from the root node.

As previously mentioned in the section on voxels SVOs often also contain shading

[5] `http://en.wikipedia.org/wiki/File:Octree2.svg`

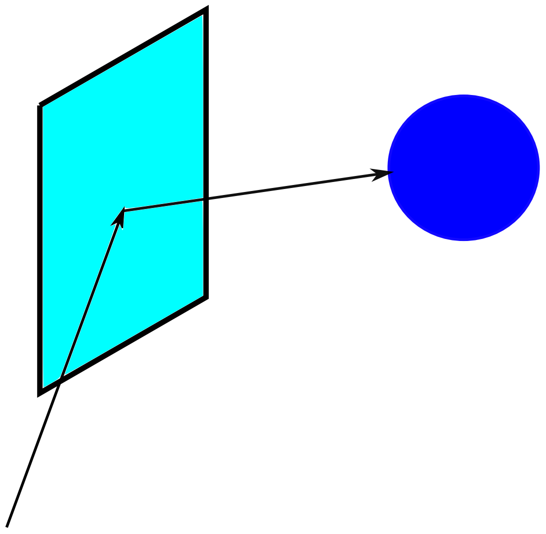


Figure 3.4.: Reflection of a ray. When the first ray hits the reflective surface (light blue) a second ray is triggered which eventually hits an opaque object.

information in the inner nodes. By doing so the traversal down the octree can be terminated early once the size associated with the current node is roughly the size of a pixel from the point of view of the camera. Doing so not only improves performance by processing viewer nodes, this also avoids aliasing artifacts as for each pixel on screen information of an appropriate resolution is used. This issue and solution is similar to texture mapping and why mip-mapping is used there.

Traversal through other spatial partitioning data structures is usually quite similar. The main difference is how the next child or sibling is identified. This depends on the criterion used to subdivide the space.

3.3.2. Reflection and refraction

Reflections, and consequently lighting, are quite easy to implement in a ray tracing renderer. Every time a ray hits a primitive the primitive's surface normal can be used to determine the direction of the reflection of that ray. The *castRay* method can then recursively call itself with the reflected ray as parameter. An example of this is shown in figure 3.4. The final color associated with the initial *castRay* call is calculated as linear combination of the shade determined by the primitive initially hit and the color returned by the recursive method call. The weights of the linear combination can be a property of the first primitive to allow for objects that reflect light in different ways.

As lighting is the amount of light reflected, dynamic lighting can easily be implemented by casting additional rays towards the light sources from each hit primitive. Each such secondary ray, which is yet another recursive method call, which is not obscured then contributes a certain amount of light based on the light source, its direction and its distance.

Refraction can also be implemented via secondary rays and recursion. Each ray which hits a transparent primitive triggers secondary rays whose directions depend on the reflective and refractive properties of the primitive. The final color again is a linear combination of the primitive's properties and the color information returned by the recursive method calls for the secondary rays.

3.3.3. Volumetric effects

Volumetric effects can also be implemented easily using ray tracing. As a ray travels through such an effect it simply "picks up" shading information which is used in a linear combination when determining the final color. The best implementation for doing so depends on the volumetric effect itself and how the renderer works exactly. One possible solution is to determine where a ray enters and exits the volumetric effect and then use the distance traveled inside the effect to determine the shading parameters. For effects which are not distributed evenly a better solution would be to take samples along the path that is inside the volumetric effect and combine those samples into shading parameters. If voxels are used as primitives each voxel along the path of the ray may serve as a sample. These solutions are still more efficient than their rasterization counterparts as they minimizes the number of samples taken from the volumetric effect and they also minimize the amount of overdraw, i.e. the number of times a pixel is shaded only to be overwritten later with new information because triangles have been processed in a non-optimal order from the point of view of said pixel.

3.4. Recap

This chapter gave an overview of ray tracing. In contrast to rasterization it does not need as many different spaces or coordinate systems. Instead it relies on intersection tests which test whether and where a ray intersects with a given primitive. Spatial partitioning is much more important for ray tracing to be able to quickly decide for which primitives intersection tests have to be performed. Once the intersection with the primitive closest to the camera has been found, shading is done similar to the shading in rasterization. In fact, many advanced techniques, especially those relying on image processing algorithms, work the same or similarly with ray tracing.

4. Related work

Very few publications have focused on basic rendering algorithms in recent years. Most research in the field of computer graphics is focused on advanced and specialized topics like physics-based simulations to increase the realism of animations. For example simulating realistic behavior of cloth, e.g. a flag in a windy environment, has become a popular topic. But in these cases only the deformation of the cloth mesh is interesting and not so much the way it is rendered. This is particularly true for rasterization as the most recent developments, which have any notable impact, are the adoption of deferred shading, a technique from the late 80's as mentioned in the lighting section of the rasterization chapter, and filter-based anti-aliasing in modern games. There are few research opportunities outside of advanced, specialized topics since a lot of research has already been done and the application of its results only waits for hardware which make them feasible as is the case in deferred shading.

In the case of ray tracing, in particular when using optimized data structures to speed up the rendering, there are more recent publications to be found. Using modern GPUs and their unified shader architectures for tasks other than rasterization has become quite popular. This has lead to research on using GPUs for ray tracing in order to achieve hardware accelerated ray tracing without the need of specialized hardware. Publications on this topic are still fairly new and recent.

This chapter aims to mention research and applications from recent years, which has not yet become standard text book knowledge as many rendering algorithms already have. The mentioned publications are less about specialized topics like physics simulations but more about the basic rendering approaches. It is subdivided into three sections. The first one focuses on rasterization related works, the second one focuses on ray tracing related work and the last section is about works which already combine the two approaches.

4.1. Rasterization

In the screen space effects section of the rasterization chapter motion blur and depth blur, also called defocus blur, were discussed. A novel approach to implement these two effects is stochastic rasterization. In stochastic rasterization a modified sampling approach is used. Traditional rasterization only considers a specific point in time. This means a sample's color is only dependent on its coordinates. In stochastic rasterization time intervals are considered instead. A sample's color also depends on a time parameter. Instead of an exhaustive sampling over the entire space and time interval for each primitive a random sampling approach is used. Sampling efficiency, meaning the number of random samples which are actually inside a primitive, is still a problem. Recent publications have successfully improved this efficiency ([10], [16]). But the performance cost still does not justify the added realism over simpler and faster approximations such as deriving a smoothing filter from motion and depth parameters to blur the image after rasterization.

A quite novel and new approach to rasterization is wavelet rasterization proposed by Manson and Schaefer ([13]). They use object boundaries to calculate wavelet coefficients which can then be used to synthesize a rasterized image. Their approach is very interesting as it automatically creates anti-aliased images. Unfortunately it currently does not allow for texture mapping and is confined to a CPU implementation. This makes wavelet rasterization unsuitable for many computer graphics applications as it lacks realism and performance. If those issues are resolved by further research though this approach may become a viable alternative to traditional rasterization.

4.2. Ray tracing

Laine and Karras have proposed an implementation of ray tracing using a SVO on a GPU ([11]). Their implementation renders a four-sided polygon covering the entire screen. It then uses the pixel shader step in the rasterization pipeline of a modern GPU to perform the ray tracing. They also add contour information to the SVO. This means that each voxel is not necessarily a cube but the intersection of a cube and the space enclosed by two parallel planes with arbitrary rotation and distance. Using contours improves the image quality when zooming so close into the scene that even the voxels at the highest level of detail become larger than a pixel. Laine and Karras' implementation also renders the image at a low resolution first and uses the information gained this way to start the octree traversal of each ray for the full resolution at a point closer to its final destination. They are able to achieve rendering in real-time.

Crassin et al. use a very similar approach in their so called GigaVoxel renderer ([6]). Their basic setup is similar yet they use rasterization of bounding volumes to determine a starting position close to the final destination for each ray. The actual ray tracing is done in the pixel shader step as well. Their underlying data structure is not an octree. Instead they use N^3 brick trees. This is a generalization of octrees. While octrees subdivide the space into two regions along each dimension each step a N^3 brick subdivides the space into N regions along each dimension each step. This allows for trade-offs between memory usage and performance. Large N generate shallow trees which are fast to traverse but are also less efficient in terms of storage as more empty space gets encoded. The GigaVoxels renderer also achieves interactive framerates.

While the previously mentioned implementations perform the tree traversal for each ray individually and do so in a depth-first way Garanzha and Loop have shown an efficient breadth-first traversal of ray bundles ([8]). Their implementation bundles similar rays and computes a containing frustum for each bundle. While traversing their spatial partitioning tree at each depth intersection tests for all frustums, which have not yet reached a leaf node, are performed before any intersections tests at higher depths are performed. By doing so the authors reduce the divergence between threads to utilize the

GPU more effectively. The downside of this implementation is an increased complexity which implies a higher cost when used in a commercial application.

Dennis Bautembach implemented animated SVOs using skeletal animation ([4]) in an attempt negate a disadvantage of spatial partitioning data structures, namely that they are hard to animate. But he ignored the spatial partitioning by simply transforming each voxel as if it was a vertex of a triangle. The transformed voxels were rendered as if they were small four-sided polygons of the size of one pixel. This even introduced new artifacts such as holes when an object came too close to the camera. While the implementation works and even can compensate for the new artifacts it introduces, it loses the performance advantage usually associated with spatial partitioning. This implementation of SVOs is closer to rasterization as it is to ray tracing even though SVOs are usually associated with the later.

A new approach for efficient ray tracing has been proposed by some researchers ([3], [15]). A naive ray tracing implementation tests all ray and primitive pairs for intersection. While other ray tracing implementations use pre-computed spatial partitioning data structures to efficiently decide which tests have to be performed this new approach uses a divide-and-conquer strategy during the ray tracing process. Rays are not traced individually but as bundles. A recursive function only performs the ray-primitive intersection tests if the number of pairs is sufficiently small. If that is not the case spatial partitioning is performed on-the-fly and recursive function calls are made with each call only processing one subset of the space and the associated rays and primitives. While the results of these implementations are promising, in particular for dynamic scenes, unfortunately no massively parallel implementation as required for GPUs has been proposed yet. The existing implementations are restricted to run only on a CPU and so can not achieve the framerates of GPU-based renderers.

Another novel approach which does not rely on pre-computed spatial partitioning was proposed by Szécsi and Illés ([22]). They also do not use polygons or voxels as their primitives but so called metaballs. Metaballs are spherical shapes which have a core and decreased density as the distance to the core increases. This kind of primitive is useful for representing particles, e.g. when simulating fluids. The proposed implementation to render these metaballs uses a rasterization step in which each metaball is represented by an enclosing billboard. This rasterization step does not determine the final color of each pixel but gathers information about which metaball potentially contributes to the color of which pixel. In a subsequent step ray-metaball intersection tests for each pixel are used to accumulate the final color information. Interactive framerates are achieved even when the number of primitives is in the order of hundreds of thousands.

4.3. Hybrid approaches

There is no research on combining rasterization and ray tracing into one hybrid renderer. The only works in this area come from the video games industry. In 1999 a game called Outcast was published which used ray traced height maps for its environments ([2]). To fill the environments with objects and characters rasterization of texture-mapped polygon meshes was used. The game was often praised for its visuals. It features effects like simulated waves and convincing reflections on water surfaces which were quite spectacular for its time. The game suffered from low image resolutions and low framerates though. The GPUs of that time were only able to assist in rasterization and bus bandwidth for uploading data to the GPU was quite limited. This resulted in Outcast using a renderer implemented entirely in software and so requiring a very strong CPU.

Cevat Yerli et al. from Crytek mentioned in their keynote for the High Performance Graphics conference in 2010 ([25]) that ray tracing instead of rasterization or maybe a hybrid approach may become the preferred way to render graphics in games in the future. They mention using data structures like SVOs but it appears that they still do not use ray tracing, even though it is a data structure for efficiently implementing ray tracing. Instead their SVOs are used during production and are later exported to a different format suitable for rendering via rasterization. It is hard to say if the actual CryENGINE, Crytek's game engine, really uses more than rasterization for the final rendering during gameplay as it is proprietary technology. Judging from the keynote it sounds like alternatives including hybrid approaches are being considered, especially for stronger hardware in the future, but are not yet used.

In a recent demo of the capabilities of the new Unreal Engine 4 Epic Games has shown some very impressive lighting effects among other eye-catching visuals. In a presentation on the technology used in this demo ([14]) they revealed to have used a sparse voxel data structure and voxel cone tracing, which is very similar to ray tracing, to achieve their lighting. In voxel cone tracing the path of a cone is traced instead of the path of a ray. This can simply be achieved by tracing the path of a ray in a voxel data structure but instead of visiting all leaf nodes, to get the highest resolution, the tracing stops at coarser detail levels to simulate the effect of the ray actually being a cone. As the ray travels the detail level of the voxel data structure being used is decreased further and further. This simulates the width of the cone increasing with respect to the distance to the starting point. Voxel cone tracing is basically a fast approximation of actual voxel ray tracing. In the case of the Unreal Engine 4 demo this was used to collect lighting information to achieve global illumination. As shown by the demo this approximation is sufficient for realistic lighting effects. It appears as if this technology will not be used in the near future as it is still too computationally expensive. Tim Sweeney, CEO of Epic Games, mentioned so in an interview ([1]).

5. Hybrid rendering

This chapter presents the core of this work. It discusses how rasterization and ray tracing may be combined into a hybrid approach in such a way that different objects in a scene are rendered by different algorithms depending on what is more suitable. The chapter's structure is as follows. It begins with a brief analysis of the goals of combining the two rendering approaches. After this the problems faced when combining the techniques are discussed. This is then followed by explanations of how the two rendering approaches are used by the hybrid renderer with a focus on ray tracing, the less commonly used approach. Since no advanced lighting or other effects are implemented by the proposed hybrid renderer, implementing rasterization required little more than calling the proper functions of the Application programming interface (API) used to access the GPU. The chapter ends with exploring an algorithm to convert a textured polygon mesh into an SVO and a recap of the chapter. SVOs are the data structure that has been chosen for the ray tracing part of the hybrid renderer. While creating polygon meshes and texturing them is well understood with many free tutorials and examples on the internet the creation of voxel data is less common. This necessitates the discussion of a way to acquire voxel data in this chapter.

Implementation details and an evaluation are left to subsequent chapters. The following sections describe the high level theory behind the implementation.

5.1. Goals

As initially mentioned rasterization and ray tracing have different advantages and disadvantages. The goal of this work is to combine the two approaches to gain the advantages of both.

Both approaches have in common that they are highly parallel problems. The tasks performed per primitive or per screen pixel such as transformations or shading can be performed in parallel for all primitives or pixels at the same time. This makes both rasterization and ray tracing well suited for being performed by a modern GPU which offers far more Floating-point operation per second (FLOPS) than a Central Processing Unit (CPU).

Rasterization has a low dependence on spatial partitioning to achieve high performance. This makes it well suited for dynamic scenes which include animated meshes that deform over time. For example animating many kinds of movements of a person requires deformations as muscles contract or the clothing stretches. Rasterization can render such meshes efficiently even though the position of the vertices of each triangle, both the absolute position in space and the relative position to the other vertices, changes in every frame that is being rendered.

Rasterization works by sampling from surfaces. This is both an advantage and a disadvantage. The amount of bytes required to define a surface is independent of its size, i.e. to make a triangle bigger the vertices have to be moved further apart but each vertex still requires the same amount of memory. This is a clear advantage when working on a system with limited capabilities such as a mobile phone. It is also a disadvantage because rasterization can not render all kinds of primitives directly. As an example fog is best represented by a volume. But rendering volume data via rasterization requires approximations or tricks such as a series of "billboards" with each billboard representing a slice of the volume data.

Another big disadvantage of rasterization is that there is no obvious way to implement many phenomena which are based on how light travels through space. Phenomena such as shadows, reflections, refraction or transparent surfaces required additional non-trivial algorithms in the shading step on top of the rasterization algorithm itself. Years of research in computer graphics resulted in many algorithms for more realistic lighting but many effects are still usually faked (ambient occlusion) or almost completely absent in many implementations (reflections and refraction).

In theory the math behind rasterization, in particular all the coordinate systems and transformations between those, is harder to understand than the math needed for ray tracing, which just requires simple vector algebra for setting up an image plane, calculating rays through said plane and intersection tests of rays with primitives. But in practice most of the mathematical complexity is hidden behind modern graphics APIs such as DirectX and OpenGL. These APIs offer utility functions to create all the necessary matrices and for multiplying matrices and vectors which means developers only need a high level understanding of rasterization without having to understand all the mathematical details.

Ray tracing is in many ways the opposite of rasterization. The math is easier to understand and has to be understood by the developers because there is little API support to help with it. Also implementing advanced lighting effects is rather easy as the same ray tracing algorithm, that is used to find an intersection with a primitive for each pixel, can be used during shading to trace the path of light rays back to the light sources in a scene. Reflections and refraction are also merely such secondary rays with their direction adjusted by the laws of physics. Also rendering volumetric effects such as fog or smoke is easy by tracing the path of a ray through a volume. Ray tracing can be used with any kind of primitive for which an efficient intersection test is known, e.g. planes/triangles or voxels.

But ray tracing is quite slow usually. The high number of potential intersection tests (each ray with each primitive) can cause rendering to slow down quickly. As a consequence ray tracing heavily relies on spatial partitioning, which itself often has to be pre-computed because its computational cost is high, to minimize the number of intersection tests actually being performed. This in turn means that any kind of animation that is more than just a translation, rotation or scaling is hard to implement. In particular deformations require the recomputation of the spatial partitioning.

So the goal of this thesis is to propose a hybrid renderer which offers efficient animation of deformable objects and also an easy and obvious way to implement advanced lighting effects and easy rendering of more than just surfaces. This will be achieved by rendering objects with different algorithms based on their kind.

5.2. Combining the two techniques

To achieve the goals set for the hybrid renderer the choice was to render each object in a scene based on what algorithm is best suited for it. Any object that is animated and requires deformations during animation, e.g. a person or an animal, is being rendered via rasterization of polygons. Any other object, in particular the static environment, is rendered via ray tracing of voxels. Ray tracing itself allows for easy implementation of advanced lighting and volumetric effects.

Voxels have been chosen for several reasons. They allow for easy filtering similar to images. Voxels are just a 3D extension to pixels and a 3D grid of voxel data is nothing more than a 3D image. Many image processing algorithms can easily be extended to 3D images which means voxel data offers new capabilities for content creators, e.g. by first creating a polygon mesh with common industry tools, then converting the mesh into voxel data and finally applying filters on the voxel data to achieve the final look. SVOs are well known as a spatial partitioning data structure to efficiently store and render voxel data so there is a well understood data structure available for use in the hybrid renderer. And finally voxels, as volume primitives, are very different from polygons which are surface primitives. As such they proof that different kinds of primitives can be used in the rendering of a scene.

There are two problems that need to be solved when using different rendering algorithms based on what object is being rendered. The first problem is occlusion. Rasterization uses a depth buffer to decide for each pixel which is the front most triangle. As the triangles may be processed in any order and as they may overlap the depth of each pixel has to be known at all times. Ray tracing on the other hand either uses an algorithm which will automatically test the primitives in the order they are seen by the ray, e.g. tracing in an SVO works this way, and terminates once it reaches a fully opaque primitive. Or it calculates the distance each time it finds an intersection and uses the shade computed for the intersection with the lowest distance. In either case ray tracing does not need a depth buffer. The other problem is that when shading an object every object rendered via the other technique has to be accounted for, e.g. when calculating the lighting, even though it may not be in the same spatial partitioning data structure.

At first it may appear that the solution to the occlusion problem is to use the same depth buffer for ray tracing as well and compute a depth based on the distance between image plane and intersection. In fact reading from the same depth buffer and updating it when necessary while ray tracing is part of the solution. However the depth written into the depth buffer by rasterization is not simply the distance between the image plane and the front most polygon at each pixels location. The perspective projection matrix and subsequent division by the original Z value used to ensure that the result of an orthogonal projection appears to be a perspective projection causes the depth value of each pixel to increase non-linearly with increasing Z value of the vertex. Multiplying a vertex

(X, Y, Z, W) with the perspective projection matrix (appendix A.4) and subsequently dividing by the original Z results in

$$depth = \frac{1}{nearZ - farZ} \cdot \left(-nearZ - farZ + 2 \cdot nearZ \cdot farZ \cdot \frac{W}{Z} \right).$$

Table 5.1 shows the resulting depth for different values of Z. As can be seen when increasing Z from $2 \cdot nearZ$ to $3 \cdot nearZ$, an increase of $nearZ$, the depth increases by $\frac{1}{3} \cdot farZ$. When increasing Z by another $nearZ$ to $4 \cdot nearZ$ the depth only increases by $\frac{1}{6} \cdot farZ$. So with increasing Z the increase in depth gets smaller. In turn this means that increasing the depth linearly implies a more than linear increase in the Z value. This means that the precision of the depth buffer is higher in close proximity to the camera which is usually desirable.

The easiest way compute the depth, as defined by the depth buffer used by rasterization, for each intersection when performing ray tracing is to transform the point at which the intersection occurs as if it was a vertex of a polygon. Adding a model to world transformation in this process is cheap as doing so only requires to multiply the matrices to perform the projection with the model to world matrix once for each object and then reusing the matrix obtained this way for each intersection. By also including a model to world space transformation each ray traced object can be rendered while it is in its own coordinate system and it will still appear at the correct position on screen. This is desirable for rendering SVOs as the voxels are axis-aligned and so the intersection tests become simpler and faster. Later in this chapter a method will be explained to use this model to world transformation to allow for almost arbitrary transformations, such as rotation or scaling, when rendering ray traced objects.

The shading problem can be solved by triggering secondary rays in the pixel shader program of each object rendered via rasterization and by using a two-step ray tracing algorithm which once traces through objects which are stored in a spatial partitioning data structure and once traces through all other objects. The two-step ray tracing then picks the front most intersection based on the depth at which each trace step encountered an opaque object. This is certainly not an efficient way of solving this problem as many

Z	**depth**
$nearZ$	-1
$2 \cdot nearZ$	$\frac{nearZ}{farZ - nearZ}$
$3 \cdot nearZ$	$\frac{nearZ + \frac{1}{3} \cdot fearZ}{farZ - nearZ}$
$4 \cdot nearZ$	$\frac{nearZ + \frac{1}{2} \cdot fearZ}{farZ - nearZ}$
$farZ$	1

Table 5.1.: Depth values for $W = 1$ after transforming a vector with the perspective projection matrix and dividing the new Z value by the old one.

intersection tests have to be performed, in particular in scenes with many deformable objects. The hybrid renderer proposed in this thesis does not account for this problem. In fact it does not implement secondary rays at all as the focus was to get a simple proof of concept first and so leaving any advanced effects that require secondary rays or complex pixel shader programs for future work.

5.3. Rasterization

The rasterization part of the hybrid renderer works just as described in chapter 2. Each object is defined in its own coordinate system or space and its vertices are transformed, in several steps, to the coordinate system of the screen. The resulting triangles are rasterized (sampled from) and for each pixel a shader program is executed to determine its color. In this shader program the color is determined by reading a filtered sample from a texture. During the rasterization of each triangle the depth buffer is also updated. Modern graphics APIs offer functions to do all those steps with little more than a high level understanding required by the user.

The only more advanced feature added for proof of concept purposes is skeletal animation. The data structure used for doing so is the one used by the MD5 format which is part of id Tech 4, the game engine used for example for the video game Doom 3. MD5 stores a skeleton for each mesh in the form of a tree of joints. Each joint has a position and an orientation. The orientation is stored as a quaternion which defines a rotation axis and angle. The actual position and orientation of each joint is determined by rotating the stored values according to the parent's orientation. This means movement of a joint affects all its children. Figure 5.1 shows an example of a skeleton defined this way.

Each vertex' position is defined as a set of weights. A weight consists of a reference to a joint, a weighting factor and a position relative to the referenced joint. This means a vertex' position is a weighted average of positions which are relative to the joints. When a joint moves so do all the vertices referencing it via their weights which finally means the polygons defined by the vertices move and may potentially deform.

The actual animation is stored as a list of frames and a frame rate at which the animation shall be played. A so called base frame stores positions and orientations for all joints. Each frame of the animation may replace those values with new ones. This means every frame fully defines the position and orientation of all joints. Since all vertices are dependent on the joints via the weights this means the position of all vertices can be different for each frame.

The frame rate and the current time can be used to determine the two closest frames of animation to the current point in time when rendering a new image. Interpolation between those frames allows for smooth animations. The best results are achieved by interpolating the joint parameters. Linear interpolation is sufficient to calculate the new position. The new orientation is best calculated via so called spherical linear interpolation, i.e. basically a linear interpolation across a circle arc. Spherical linear interpolation is offered by modern graphics APIs as a pre-defined function. The interpolated joint pa-

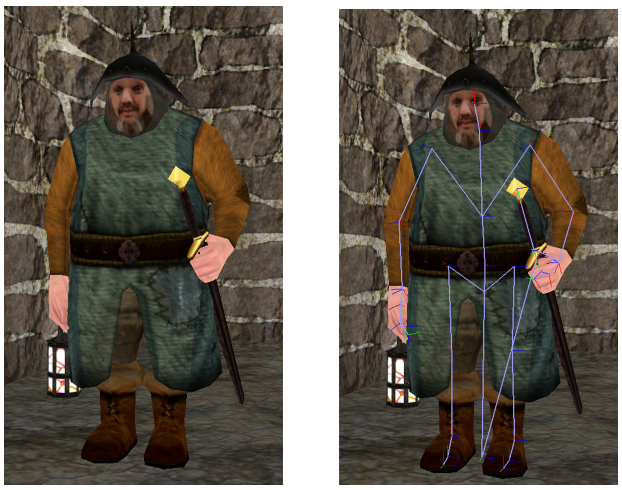

(a) without bones (b) with bones

Figure 5.1.: An MD5 mesh rendered once without its underlying skeletal structure for animation and rendered once with said structure. Source of the screenshots: cubemapdemo found at KatsBits.[6]

rameters can then be used to calculate the vertex positions for each frame.

5.4. Voxel ray tracing

As previously explained voxels were chosen as primitives for ray tracing. While easy to understand as being just small cubes or a 3D extension of a pixel, voxels do have some drawbacks. A 3D grid of voxels can quickly use a lot of memory. For example a grid of merely $256 \times 256 \times 256$ voxels, with each voxel consisting of just a 24 bit RGB color value with its location and size being implicitly defined by the location inside the grid, would already require 48 MB of memory while not being very pleasant to look at up close. Increasing the size along each dimension by just a factor 2 results in a total increase by a factor of 8 for the memory requirement. A 1024^3 grid of RGB color values, which would result in a decent image quality in many cases, needs 3 GB of memory.

But for many objects the majority of these grid cells would have to be marked as empty anyway. So an efficient way of storing voxels is required. An SVO is a data structure well suited for this task. Not only allows it to waste little space on empty grid cells but it also allows for quickly tracing a ray through the space occupied by the SVO by enabling the tracing process to quickly skip over empty space.

[6]`http://www.katsbits.com/download/models/`

5.4.1. Sparse voxel octrees

A sparse voxel octree (SVO) is the application of spatial partitioning via an octree to a voxel grid. As previously explained an octree iteratively subdivides the space it represents into smaller chunks. In the case of SVOs each node represents a cube which encloses a part of the underlying voxel grid. The root node encloses the entire voxel grid. A node can have either no children, e.g. if it is empty anyway, or exactly eight children. If it does have children all of them are cubes which partition their parent into eight chunks of equal size. Figure 3.3a illustrates this. To simplify the implementation of an SVO it makes sense to use an axis-aligned voxel grid so that the sides of each voxel or cube are parallel to one of the planes spanned by the basis vectors of the coordinate system.

SVOs are unbalanced trees with nodes representing large cubes of empty space and nodes representing small cubes of actual voxel data, e.g. RGB color values. Even though the tree structure itself requires some memory for storing pointers the total memory requirement of an SVO is usually much lower than storing the voxel grid directly as most objects occupy only a small part of the grid.

5.4.2. Tree traversal

To find the first voxel in an SVO, which intersects a given ray, the intersection with the root node has to be calculated first. If it does not intersect the root at all the ray obviously does not intersect the object at all. If there is an intersection this point must be used as the starting point of the tree traversal.

The most obvious way of traversing down an octree to a given point is by simply testing for each child whether or not the point is contained within it. A more efficient way would be to answer these three question to uniquely identify the correct child:

- is the point in the top or bottom half?
- is the point in the left or right half?
- is the point in the front or back half?

Both approaches assume that the octree represents an actual partition and that there is no overlap between the children of each node. A modified traversal can even handle the case when the sides of each child overlap with its siblings and the point being traversed to is part of a ray with a known direction. By taking the direction of the ray into account when testing all the children for containment and terminating at the first success the correct child can be identified in the case of overlap. If the direction vector points upwards the children representing the top half have to be tested before the children representing the bottom half are tested and vice versa. The same is true for the other axes, e.g. if the direction vector points to the left the children in the left half have to be tested first. Figure 5.2 illustrates this in the 2D case. The idea is that when a ray arrives at a point, whose containment is ambiguous, it only does so when leaving one child and entering the other. The direction of the ray identifies which child is being

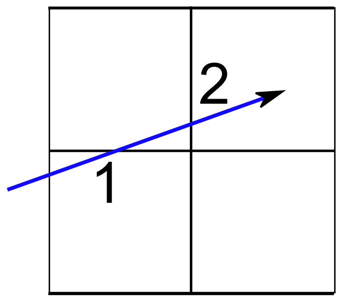


Figure 5.2.: A ray travelling through a quadtree node. During traversal it will reach the ambiguous points 1 and 2, when the ray reaches the border between two children, but the direction will determine which child is the correct one. In this case the top children have to be tested before the bottom ones and the right-hand children have to be tested before the left-hand ones.

left and which one is being entered. If a ray moving upwards encounters the overlapping side of two cubes on top of each other it will only do so when leaving the bottom cube and entering the top cube. So testing the top cube first results in correct traversal.

Once a leaf node has been found the traversal may either end and shading may be triggered if it is non-empty or the traversal must continue of the leaf is found to be just empty space. When continuing traversal the first intersection with one of the sides of the leaf's cube has to be determined. The point at which the intersection occurs can be used to restart the traversal from the root node until the ray either leaves the tree or a non-empty leaf is found.

5.4.3. Containment and intersection

Testing if a point is contained inside an axis-aligned cube or if a ray intersects one of its sides are both easy tasks.

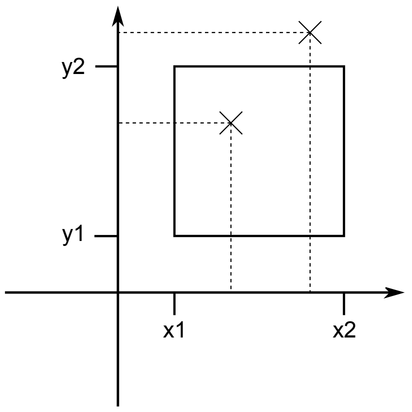
The test whether or not a point is contained within the cube can be reduced to three tests whether a value is contained within an interval. The cube's sides define one interval along each axis. A point is only contained within the cube if the point's value for each axis is within the interval for that axis. Figure 5.3a illustrates this in the 2D case.

To find the intersection of a ray with one side of an axis-aligned cube an easy equation results from the parameter form of a ray

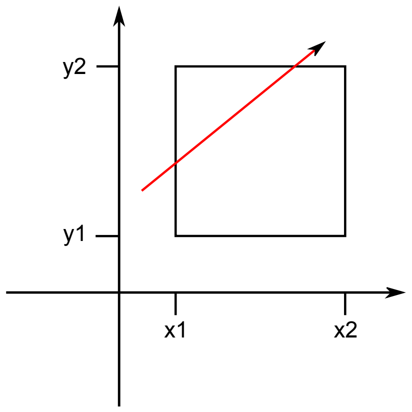
$$\mathbf{x} + \lambda \cdot \mathbf{d}$$

with vector $\mathbf{x}$ being a point on the ray and vector $\mathbf{d}$ being the direction of the ray. For each dimension i this can be written with scalars as

$$\mathrm{x}_i + \lambda \cdot \mathrm{d}_i.$$



(a) Containment. The square defines the intervals $[x1, x2]$ and $[y1, y2]$. Only if a point's values for each axis is within the corresponding interval it is contained in the square.



(b) Intersection. The parameter form of a ray gives one equation to solve for each dimension but only one such equation is necessary to find the intersection with a side.

Figure 5.3.: Examples of containment of points in squares and intersection of a ray with squares.

To find the intersection with a given side its one constant value has to be identified and set equal to the ray expression for that particular dimension. For example in figure 5.3b the intersection with the right side of the square can be found by solving the equation

$$\mathrm{x2} = \mathrm{x}_1 + \lambda \cdot \mathrm{d}_1$$

for λ (assuming $i = 1$ is the horizontal dimension) and then using λ to calculate the actual point of intersection and test it for containment within the square (or cube in 3D).

The first side intersected by a ray starting at **x** and pointing in the direction **d** can be identified by comparing the λ values for each intersection. The smallest non-negative λ value identifies the first side being intersected. To also exclude **x** itself the smallest positive λ has to be picked.

5.4.4. Pseudo-backtracking

Restarting the tree traversal from the root node every time is inefficient, in particular when the correct child to pick at each node is identified via testing all children in the correct order. Backtracking up the tree to the parent would require additional pointers though. A compromise between restarting the traversal and true backtracking is to use a stack to speed up the traversal restarts and so implementing a form of pseudo-backtracking.

While traversing down the tree an identifier can be pushed onto a stack to identify which child was chosen at which depth. A 3 bit identifier is sufficient. One bit indicates whether the child belongs to the top or bottom half, another bit indicates whether the child belongs to the left or right half while the last bit indicates whether the child belongs to the front or back half. This is a minimal way to uniquely identify each child.

When encountering an empty leaf node the cube side which is intersected first can be used to identify which identifiers have to be removed from the stack for the next iteration. For example if the right-hand side of the cube is intersected first every identifier that also indicates the right half has to be removed up to the first one that does not. This identifier belongs to the first node at which the traversal will differ. This identifier will indicate the left-hand side, yet the next iteration will choose the corresponding child on the right-hand side. So the left/right bit of this identifier must be flipped. This works analogously for every other cube side.

The next iteration can then use the stack, if it is not empty because the same side was chosen at every node in the previous iteration, to quickly traverse the tree to the first child that is different and then continue traversing down as before.

5.4.5. Depth buffer

Usually ray tracing does not require a depth buffer. But when rendering different objects using different approaches a depth buffer must both be taken into account and updated even when rendering via ray tracing. The depth of a given voxel can be determined by transforming the point used for traversing down the octree in the same way as vertices of polygons are transformed before rasterization. Assuming the depth of the vector (x, y, z) shall be calculated then $(x, y, z, 1)$ has to be multiplied with all the matrices used to transform a vertex from object/model space to world space to camera space and finally to screen space. This results in a vector (x', y', z', w'). The depth of the voxel is then simply

$$\text{detph} = \frac{z'}{w'}.$$

This last division is a step that can not be expressed in the matrices themselves and is usually performed automatically by the GPU without the application developer having to specify this.

Comparing the voxel's depth to the corresponding one in the depth buffer can be done at two times during the traversal. Every time before an iteration of traversing down is started the depth of the point used to traverse can be calculated and compared to the existing depth. By doing the depth test this early unnecessary tree traversal iterations can be avoided. But this also implies that the matrix-vector multiplication to compute the depth has to be performed several times per ray traced pixel.

Alternatively the depth test can be delayed up to the point when a non-empty leaf node is found. By doing so the depth only needs to be computed once but potentially unnecessary tree traversal iterations have to be performed.

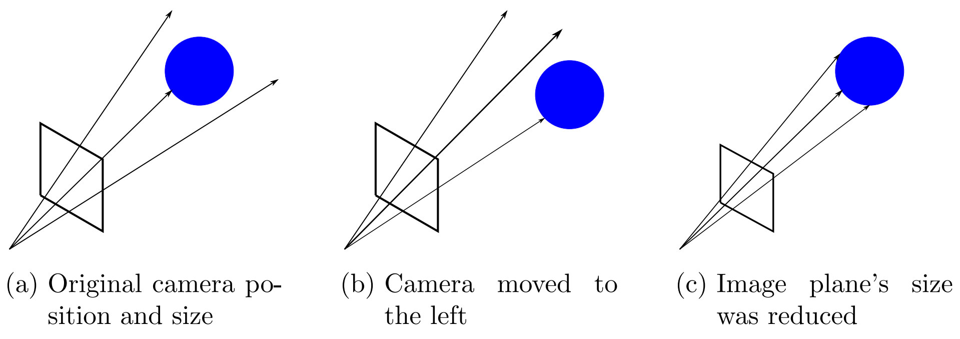

(a) Original camera position and size (b) Camera moved to the left (c) Image plane's size was reduced

Figure 5.4.: An example of how camera motion can induce motion in the scene. Also reducing the image plane's size has an effect. It makes objects appear larger.

5.4.6. Transformations

As previously mentioned some kinds of transformations such as translation, rotation and scaling can also be applied to ray traced SVOs without having to change the underlying data structure at all. Basically any transformation that is invertible can be applied.

The camera's position and the corners of the image plane form a pyramid. This pyramid can be used to determine the starting point and direction of each ray to trace such that there is exactly one ray for each pixel on the screen. As with a real camera movement of the camera itself induces an inverse effect in the scene. This is shown in figure 5.4. With a virtual camera even more effects are possible such as reducing the size of the image plane which makes everything appear larger.

By exploiting this behavior almost arbitrary transformations can be applied dynamically at runtime to SVOs. A world matrix with all the transformations applied to the SVO must be maintained. Before rendering the inverse of this matrix is applied to the points forming the "camera pyramid". This new pyramid is then used to determine each ray's starting point and direction. After rendering is finished it will appear as if the transformations had actually been applied to the SVO.

The only limitation is that every transformation itself must be invertible. This follows from

$$(A \cdot B)^{-1} = B^{-1} \cdot A^{-1}$$

for matrices A and B. The final world matrix, after all transformation have been applied to via multiplication, is only invertible if each transformation was invertible to begin with.

5.5. Generating voxel octrees

Most 3D graphics application nowadays use polygons and rasterization. While it is more or less easy to find free polygon mesh examples for testing it can be rather difficult to obtain example voxel data. For this reason a conversion from textured polygon mesh to an SVO storing RGB color values for each non-empty voxel is also proposed in this thesis.

5.5.1. Building an octree

An octree can be build via constructing a root node, splitting it into children and recursively repeating this process until a termination criterion is reached such as a maximum depth.

The root node can be constructed by finding the smallest cube enclosing the model and placing it so that the center of the cube and the center of the model are in the same place. The cube should also be axis-aligned. For performance reason it makes sense to maintain a candidate list for reach node. This list contains every triangle that intersects the node. In case of the root node this means that every triangle is added to its candidate list.

During the recursive tree building each potential child of the current node is created. For each child an intersection test with all the triangles from its parent's candidate list is performed to see if any triangle intersects the new child. Any intersecting triangles are then added to the child's candidate list. If at least one child is not empty the parent node is actually split up by keeping the children in memory and storing their pointers in the parent node. The tree building then continues recursively with the children.

This algorithm ensures that only non-empty nodes (= cubes/voxels) are split up further and the lack of a valid child pointer indicates that a node is a leaf. The candidate list also allows to quickly decide with which triangles intersection tests have to be performed at all and so improves the performance of the algorithm.

5.5.2. Intersection test

Checking whether a triangle intersect a cube is far more involved than checking whether a ray intersects it. But triangle-cube intersection can be broken down into a series of ray-plane intersection tests for the purpose of creating an SVO.

The intersection of a triangle and a cube is plane segment in general. To determine the color values of each voxel later it is sufficient to know the positions of the corners of this plane segment without actually computing its other properties such as area or orientation.

Each triangle's position relative to any given cube is arbitrary. It may lie completely inside the cube. One or two vertices may lie inside the cube but not every one of them. Or all vertices may lie outside of the cube yet there may still be an intersection, e.g. a triangle which would cut the cube into two halves must be larger than the intersecting plane segment and may even be so large that none of its vertices lie within the cube itself.

To account for all these possible cases while only being interested in the corners of the intersecting plane segment the problem can be reduced to a series of ray-plane intersections. Each pair of vertices of a triangle forms a ray. Let the vectors $\mathbf{x}$ and $\mathbf{y}$ be such a pair. Then they define a ray in the form

$$\mathbf{x} + \lambda \cdot (\mathbf{x} - \mathbf{y})$$

with $\lambda \in [0, 1]$. These are rays which can be tested for intersection with a cube the same way as it is done during tree traversal when rendering an SVO. The only added criterion is that λ values outside of $[0, 1]$ can be ignored as they are not part of the ray formed by the vertices.

The same trick can be used to test the cube's edges for intersection with the triangle. Any pair of cube corners also form rays in the same way the vertices of the triangle do. But not all pairs have to be used to construct rays to intersect with the triangle. Only those pairs which do not require diagonal movement to get from one corner to the other one.

However the intersection of a ray with an arbitrary plane is not as easy to calculate as is the intersection with an axis-aligned cube. The most convenient way to do this is to use the point-normal form of a plane. If $\mathbf{x}$ is a known point on the plane and $\mathbf{n}$ is its normal, which can easily be computed with a cross product, then a plane is uniquely defined by all vectors $\mathbf{p}$ which satisfy the equation

$$(\mathbf{p} - \mathbf{x}) \cdot \mathbf{n} = 0.$$

The intersection with a ray

$$\mathbf{p} = \mathbf{y} + \lambda \cdot \mathbf{d}$$

is defined by the λ value which satisfies

$$(\mathbf{y} + \lambda \cdot \mathbf{d} - \mathbf{x}) \cdot \mathbf{n} = 0.$$

Solving this equation for λ results in

$$\lambda = \frac{(\mathbf{x} - \mathbf{y}) \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}.$$

If the denominator of this equation is zero the ray either lies within the plane, a case which can be safely ignored for the purpose of building an SVO, or it does not intersect it at all. For any non-zero denominator λ can be calculated and using this λ the actual point at which the intersection occurs can be computed.

A triangle is not a plane which extends into infinity. This means after finding the intersection another check has to be performed to find out if the intersection is actually within the area of the triangle. This is can be done easily with the parameter form of a plane. Let the vectors $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$ be the corners (vertices) of a triangle, then they span a plane in the form

$$\mathbf{p} = \mathbf{x} + \lambda \cdot (\mathbf{y} - \mathbf{x}) + \mu \cdot (\mathbf{z} - \mathbf{x})$$

with $\lambda, \mu \in [0, 1]$ and $\lambda + \mu <= 1$. This is a linear equation system with three equations, one for each dimension, and two unknown variables, namely λ and μ, if $\mathbf{p}$ is set to the point of the intersection. For any true triangle, meaning with $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$ being pairwise different, this linear equation system has a unique solution. After solving this equation system the properties of λ and μ can be used to determine if the intersection is actually within the area of the triangle.

This series of intersection tests will identify all corners of the plane segment that is the actual result of intersecting a triangle with a cube.

5.5.3. Computing the color information

Computing each node's color is a task that is best performed recursively as well. After the whole tree has been built each inner node determines its own color by averaging the color of all non-empty children. Empty leaf nodes do not actually need a color but they need to at least set a flag indicating that they are empty.

Computing the color of a non-empty leaf works by reusing some of the concept from the intersection tests. In order to take a filtered sample from a texture the proper texture coordinates have to be known. These can be calculated for each intersection by using the parameter form of a plane. The same λ and μ values used to determine whether an intersection is inside a plane can be used to calculate texture coordinates for said intersection. Each vertex of a triangle has texture coordinates associated with it. These coordinates can be used to span a plane

$$\mathbf{p} = \mathbf{x} + \lambda \cdot (\mathbf{y} - \mathbf{x}) + \mu \cdot (\mathbf{z} - \mathbf{x}),$$

too. Here $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$ are not the positions of the vertices but their texture coordinates instead. Using λ and μ calculated from the plane spanned by the positions allows one to calculate a texture coordinate $\mathbf{p}$ for each intersection.

With the texture coordinates obtained this way a filtered sample, e.g. using a simple bilinear filter, can be taken for each intersection. A non-empty leaf's color can then be computed as the average of all the samples.

5.6. Recap

This chapter introduced an actual hybrid renderer on a theoretical level. The renderer works by rendering each object using an approach appropriate for it. Animated objects which need to deform are rendered using the common rasterization of polygons. Everything else is rendered using ray traced voxels which are stored in an SVO in order to gain the advantages of ray tracing, namely easier lighting and rendering of volumetric effects, for as many objects as possible. Maintaining a depth buffer even when performing ray tracing allows for rendering objects in an arbitrary order and still get proper occlusion. Also post-processing algorithms may potentially take advantage of the depth buffer as its content is always in a valid state.

Also an algorithm for generating SVOs from textured polygon meshes has been proposed. This allows for easy acquisition of test data and may also be useful for content creators as they do not have to learn to use new tools but can create content such as meshes in their familiar tools and then convert those to voxel data.

6. Implementation

After explaining the hybrid renderer on a theoretical level in the previous chapter this chapter will describe its implementation. This includes discussing the technologies used and any supporting tools developed such as the tool for converting polygon meshes to SVOs.

The chapter will start by describing the tools and file formats used by the implemented software. This will also include details on what information is stored in memory for each object at runtime. After that follows a description of the hybrid renderer itself, which technologies it uses and what its features and limitations are. The chapter ends with a discussion of floating-point number precision, which turned out to be a problem in some cases, and a short recap.

6.1. Tools and file formats

In order to gain a better understanding of the implementation of the renderer itself it is helpful to understand the structure of the data files and how they were obtained.

6.1.1. File formats

In total five different types of file formats are used by the renderer and its supporting tools. Two formats for polygon meshes, two formats for textures and one format for storing an SVO.

The two polygon mesh formats are ASE and MD5. Both are in ASCII format and allow the raw data to be viewed in a common text editor. ASE is a format used by the 3D modelling software 3D Studio Max.[7] As mentioned in the previous chapter MD5 is the format used by the video game Doom 3 for its characters.

ASE specifies a mesh as a lists of triangles and vertices. Each triangle consists of three indices which point to the vertices defining the triangle. There are actually two lists of triangles and two lists of vertices. One pair of lists define the positions of each vertex while the other pair of lists describe the texture coordinates. This allows meshes to reuse vertex positions with different texture coordinates depending on which triangle the position is used for. ASE files also specify a lot of additional information such as normals, lighting parameters, etc. but those were mostly ignored by the supporting tools. Only the triangles, vertex positions, texture coordinates and texture file name were used. The renderer itself does not use ASE files directly at all. The texture information is stored in separate image files in the TGA format.

[7]`http://www.autodesk.com/products/autodesk-3ds-max/overview`

MD5 uses two ASCII files to store its information. One file, the so called mesh file, stores the geometry of the mesh together with information about texture mapping and the skeletal structure for animation. The other file, called anim file, stores the animation data. An MD5 mesh may be made up of several submeshes. The mesh file stores all these submeshes as a list. Each entry in that list consists of a texture name and three lists, triangles, vertices and weights. The triangle list which stores indices into the vertex list. Each vertex consists of texture coordinates and a pointer into the weight list which specifies a set of weights. The weights are responsible for defining a vertex' position based on the skeletal structure as described in chapter 5 when discussing the rasterization part of the hybrid renderer. The mesh file already fully defines an non-animated version of the polygon mesh. This default pose of the polygon mesh is called the bind pose and it is the same pose the model creator chose when creating the model. The anim file contains the frames of animation mentioned in chapter 5. MD5 also uses TGA images to the store the textures. MD5 does not store normals. These have to computed while loading the mesh.

These two mesh formats were picked because information on parsing them was easy to find on the internet as well as example meshes which can be used for free. For testing the hybrid renderer two models from a website called KatsBits[8] were used, namely a medieval character called Bob and an appropriate background looking like a part of a medieval castle. These two meshes are shown in figure 6.1.

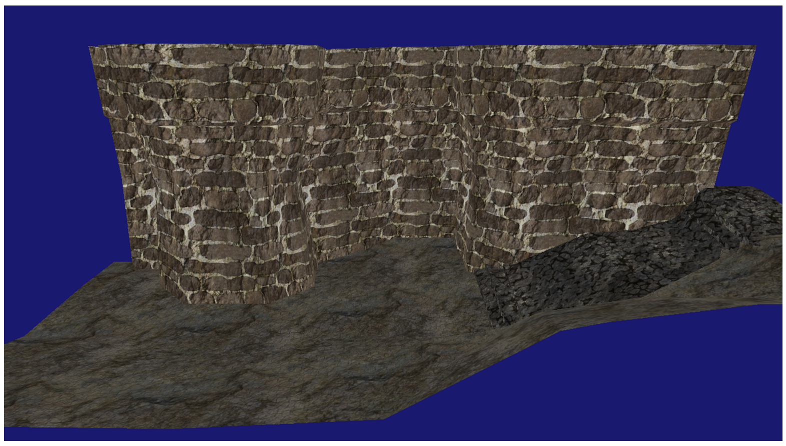
The two image formats used are TGA and DDS. Neither format was parsed manually. A free image loading library simply called stb_image.c[9] was used to load the TGA files and a library called DXUT was used to load the DDS files. DDS is a format specifically for use with DirectX as it natively supports texture compression formats that modern GPUs can decompress in real-time when reading texture samples.

The format used for storing SVO files was a self-developed one. A simple binary format was chosen. It begins with the size of the root node as 32-bit floating-point number. Since each node is a cube a single number is sufficient as the length and height of all sides is equal. The position of the center of the root is implicitly assumed to be $(0, 0, 0)$. By applying transformations at runtime the SVO can be moved to any arbitrary location as desired. The size is followed by the color and child information for the root. Each node is represented by two 32-bit words in the file. The first 32-bit word is an RGBA color value with the alpha channel being used to indicate that a note is empty ($\alpha = 0$). The second 32-bit word is the offset in bytes of the first child in the file. For example if an offset specifies address 200 (in decimal) then the first child can be found 200 bytes from the beginning into the file. All children are always written in the same order and in one $8 \cdot 64 = 512$ bit chunk. This way the order implicitly defines which part of the space belongs to which child. Offsets of 0 indicate that a node is a leaf.

[8] `http://www.katsbits.com/download/models/`
[9] `http://nothings.org/stb_image.c`

(a) Bob in bind pose

(b) One of Bob's textures


(c) background

(d) wall texture

Figure 6.1.: Example meshes and textures from KatsBits[8].

6.1.2. Tools

Two supporting tools to accompany the hybrid renderer were written. The first tool is a conversion tool from the ASE format to the MD5 format. This tool was written so that every other tool and the renderer only have to be able to parse one single format, namely MD5, and still be able to load every mesh. The conversion tool constructs a single, non-rotated joint in the origin of the coordinate system. Each vertex position from the ASE file is converted to a single weight in the MD5 file. Each unique pair of position and texture coordinates from the ASE file is converted to a vertex pointing to the appropriate weight in the MD5 file. During this conversion a mapping of (position, texture coordinate) pairs to vertex indices is created. This mapping is then used to create the triangle list for the MD5 file. No MD5 anim file is created so that the ASE model only exists as MD5 mesh in bind pose after the conversion is complete.

The second tool is the SVO construction tool. It reads the bind pose of an MD5 mesh and converts it to an SVO stored in the format described in the previous subsection. The tool can create octrees up to a specified depth and also measure the time required for its various steps.

6.2. The hybrid renderer

The main program, which was developed, is the hybrid renderer itself. It is a program being able to render objects via rasterization and via ray tracing. It uses to the system's GPU for fast rendering in both cases.

6.2.1. Technologies

The hybrid renderer was developed on a Microsoft Windows 8.1 Pro 64-Bit PC using Microsoft Visual Studio Ultimate 2013. The hybrid render and its accompanying support tools were written in C++. To access the GPU DirectX 11 was used. From this follows that all the shader code was written in the High-level shader language (HLSL). DirectX 11 was also used for most of the operations performed on vectors, matrices and quaternions as it offers a large collection of useful functions for creating and manipulating these objects. Textures and their required DirectX 11 API objects were created via DXUT[10], a helper library to make using DirectX 11 even easier. This library also provides GUI functionally which was used. The target platform of the hybrid renderer was a 64-bit Windows-based PC with a GPU that supports DirectX 11 and Shader Model 5.0. This choice allowed for an implementation with little technological limitations such as low memory limits or no control flow in shader code.

6.2.2. Rasterization

The rasterization part of the renderer is a straight forward implementation of rasterization. DirectX 11 API functions were used as much as possible to set up a render target

[10]`https://dxut.codeplex.com/`

(buffer representing the screen), depth buffer, vertex buffers, transformation matrices, etc. In the end each vertex processed by the GPU consist of a position, a normal and texture coordinates. A vertex shader written in HLSL transforms the position to screen space, the normal to world space and passes on the transformed vectors and the texture coordinates to the (non-programmable) rasterization part of the GPU. This part is responsible for performing depth tests, so that no pixel is overwritten if the new triangle is occluded by the existing data, updating the depth buffer and triggering a pixel shader, also written in HLSL, where appropriate. This pixel shader receives the information passed on from the vertex shader but in interpolated form. This means the pixel shader can use the texture coordinates it receives directly to read a filtered texture sample. This sample could then be modified depending on the interpolated normal and the lighting conditions. However no lighting is implemented and so the pixel shader simply outputs the sampled color. This also implies that only one texture is sampled from per polygon. No lightmaps or similar texture mapping techniques are used.

6.2.3. Ray tracing

Ray tracing of SVOs was implemented twice. The first implementation only utilized the CPU and was created for easy testing and debugging. It partitions the screen into four quadrants and uses one thread to render each quadrant. This significantly increases its performance on multi-core CPUs as long as the rendered object is in the center or near the center of the screen.

Later a GPU implementation of the ray tracing was created. It works by sending two triangles to the GPU which cover the entire screen. These triangles have texture coordinates set up in a way that the coordinates can be used to address individual pixels in the pixel shader used to render them. Both components of the texture coordinates range from 0 to 1. The first component addresses the horizontal axis with 0 being the left most column and 1 being the right most column of the screen. Analogously the second component address the vertical axis with 0 being the top most row and 1 being the bottom most row.

Both the CPU and GPU implementation implement ray tracing in the same way for each pixel. After the pyramid formed by the camera position and image plane corners was transformed by the inverse transformation matrix, a bilinear interpolation between the four image plane corners is used to calculate the starting point of the ray for each pixel. The difference between starting point and camera position defines the direction of each ray. With these two vectors, the starting point and the direction, the ray tracing is performed as described in chapter 5.

Both implementations are able to either perform early depth tests (before every down traversal iteration) or late depth tests (once an opaque leaf node is found). Either option has to be chosen when loading the SVO. This is due to the implementation on the GPU. It uses slightly different pixel shaders to implement each option and which pixel shader is loaded is decided when loading the SVO itself.

The proposed ray tracing algorithm requires a stack for increased performance. Managing a list is both expensive in terms of operations and memory. This is particularly bad for the GPU implementation as it needs to reserve this memory for all pixels at the same time because they are traced in parallel. Due to these reasons a single 64-bit variable (or two 32-bit variables as is the case for the GPU version) and an integer are used as a stack. Bit operations are used address individual segments of the 64-bit variable. Since each element on the stack, an identifier of a child, needs only three bits the variable is large enough to hold a stack of 21 values. This puts an upper limit on the maximum tree depth. A tree of that depth would still be able to store a voxel grid with more than two million (2^{21}) entries along each dimension which is still more than enough for current applications. The integer used in addition to the 64-bit variable holds the size of the stack.

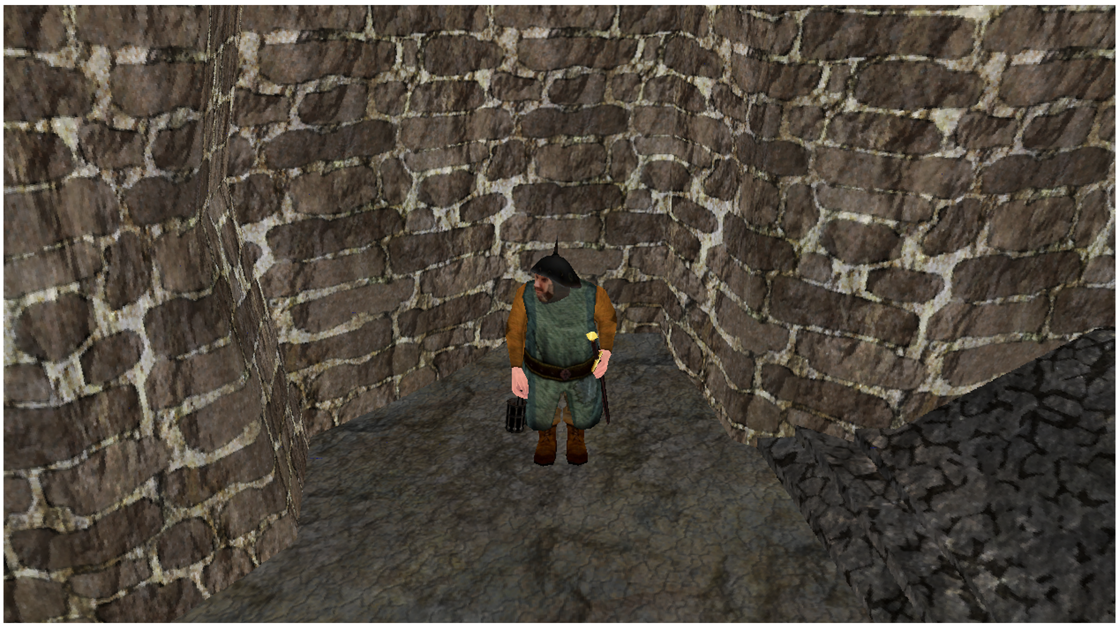
Updating the depth buffer works slightly differently for both implementations. The CPU version copies the depth buffer to its own memory space, updates it locally during the tracing process and then copies the updated buffer back to the GPU. The GPU implementation uses the depth buffer as a texture. For each pixel the current depth is read and potentially updated. Either the existing or the new depth value is then written to a separate buffer as the depth buffer can not be written to and read at the same time. After the ray tracing is finished the content of the buffer containing the new depth information is copied to the real depth buffer.

6.2.4. Depth visualizer

Another feature of the hybrid renderer is the ability to visualize the depth buffer. This feature is implemented mostly on the CPU as it is only relevant for debugging. Just like the CPU implementation of ray tracing a copy of the depth buffer in CPU memory is created. The CPU then computes the minimum and maximum depth values while ignoring the background, which is all pixels with depth equal to 1. These two values are then used to create an image which is white at pixels with minimal depth and black at pixels with maximum depth. All background pixels are fully transparent. This image is then used as a texture by a vertex and pixel shader combination which works similarly to the ray tracing GPU implementation. Two triangles, which cover the screen, are send through the GPU pipeline to copy the content of the texture onto the screen. The result of the depth visualizer is illustrated in figure 6.2. It was used during development to be able to visually assess the correctness of the content of the depth buffer after ray tracing.

6.2.5. Demos

The hybrid renderer includes four hard-coded demos. Each demo is a sequence of 101 frames with pre-defined camera positions and time values for the animation. These are useful in comparing the performance of the renderer with regard to different settings. Each demo represents a typical kind of camera motion. The last 50 frames of each demo are the first 50 frames in reverse order. The first demo is a panning motion from right to left and back again. The second demo is a zoom into the scene, in particular to a character. The remaining two demos rotate the camera. One demo moves the camera along a half circle with the location it is looking at fixed at the center. In the last demo


(a) Without depth visualizer

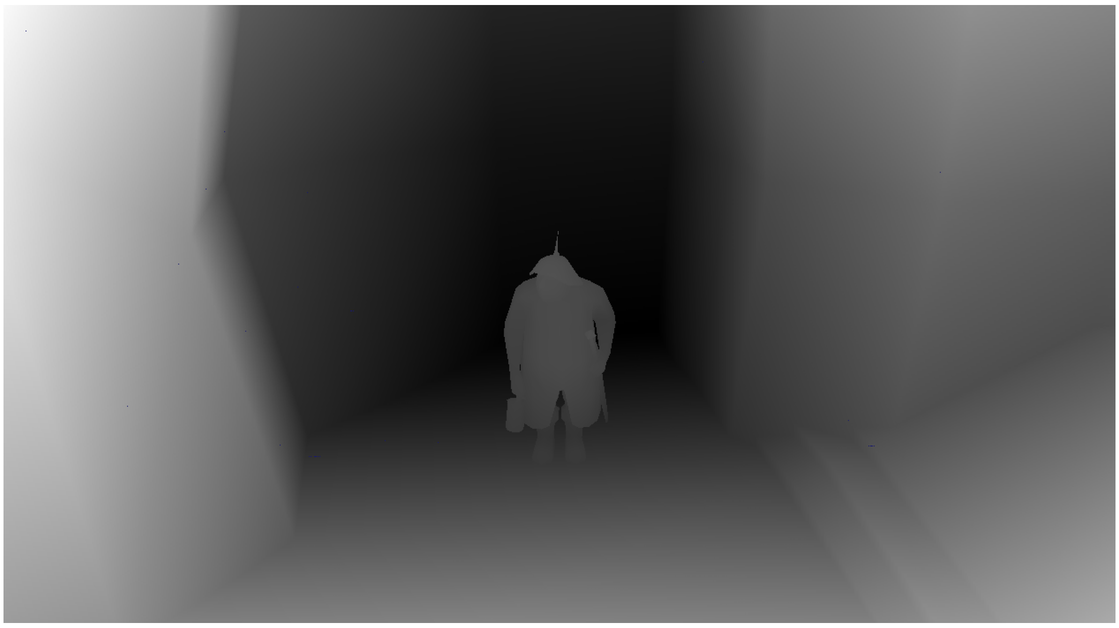

(b) With depth visualizer

Figure 6.2.: The same scene once with the depth visualizer turned off and once with the depth visualizer turned on.

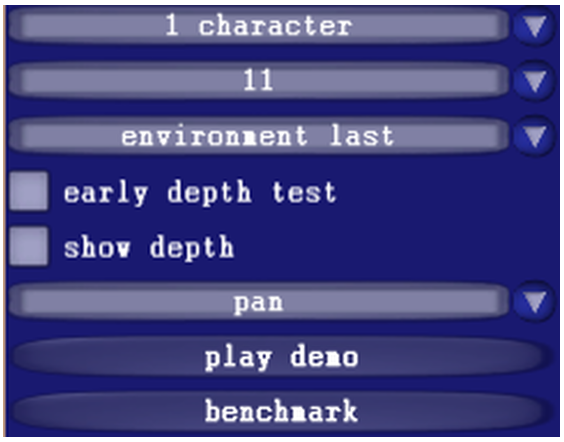


Figure 6.3.: The GUI which allows the user to change scene and render parameters at runtime.

the camera stays fixed in one location and rotates by 360 degrees.

6.2.6. Parameters and GUI

As its basic setup the renderer shows a scene with a single character in front of a background. The background is rendered as a ray traced SVO by the GPU while characters are rendered as animated polygon meshes. A GUI allows to the user to add more characters. The locations of these additional characters is fixed and some are intentionally placed behind the background object. Also the maximum depth of the background SVO can be chosen at runtime, as can the ability to render it via the CPU or to enable early depth tests. The order in which characters and background are rendered can be changed, too. The depth visualizer can be toggled on and off. Each demo can be played back. All of these options can be chosen at runtime via a GUI in the upper left corner of the screen as shown in figure 6.3. The only parameter which can not be chosen at runtime is the screen's resolution. This can however be specified as commandline argument when launching the program.

6.2.7. Benchmarking

The hybrid renderer can perform automated benchmarking. It does so by iterating over all the parameters available in the GUI. Each demo is played ten times with each possible combination of settings. During playback the times to render each frame are measured and logged. Since the CPU and GPU run asynchronously it is hard to measure the time individual operations need on the GPU. Measuring the time it takes to render the frames still allows one to make a relative comparison between features. For example the exact times to perform ray tracing are not known but by comparing runs with early depth test to runs with late depths the faster implementation can still be determined. Also repeating each run ten times and then averaging the times and removing obvious outliers allows to rule out outside influences, such as services running on the computer, on the measured times.

6.2.8. Limitations

The way the SVO is constructed and stored combined with the way it is rendered produces results similar to using a nearest neighbor filter to read samples from a texture without mip-mapping. One artefact produced by this is aliasing. This could be remedied by modifying the ray tracing algorithm to terminate at a lower depth depending on the current node's size with regard to the screen. Once an opaque node is reached that is roughly as large as one pixel from the point of view of the camera the tracing process should terminate.

Another artefact is that the voxels themselves become visible as cubes with a single color if the camera moves so close to the SVO that even the voxels at the highest depth become larger than a pixel. This could be remedied by blurring the picture at those locations. The proper regions to blur can be identified by using another buffer in which each ray writes whether the resolution of the SVO was sufficient for its pixel location or not.

Another obvious limitation is the lack of secondary rays. The scope of this thesis was to implement a proof of concept renderer first. If viable, future works can then add onto this implementation and evaluate the viability of hybrid rendering with regard to advanced lighting, volumetric effects or other advanced rendering techniques.

6.3. Floating-point precision

Floating-point numbers only have a limited precision as every programmer and every computer scientist knows. In many applications this does not pose a problem. During development of the hybrid renderer this turned out to be a problem in this case though. Sometimes the ray tracing algorithm would not terminate because it got stuck in an endless loop traversing down the same path of the tree over and over again due to not updating the point to traverse to properly. SVOs sometimes contained triangle-shaped holes due to the candidate list missing an entry which was mistakenly rejected at a lower depth. Figure 6.4 shows an example of this problem.

The solution was to avoid recalculating floating-point values when possible and to allow some slack at boundaries, e.g. by treating -10^{-7} as if it was larger than 0. These are standard tricks to avoid precision problems and in the case of creating and rendering SVOs it is actually necessary to use them.

Two places were identified to cause the most issues. One is the calculation of the point of intersection between a ray and an axis-aligned cube. When doing so the equation (or an analogous one for a different dimension)

$$y_1 = x_1 + \lambda \cdot d_1$$

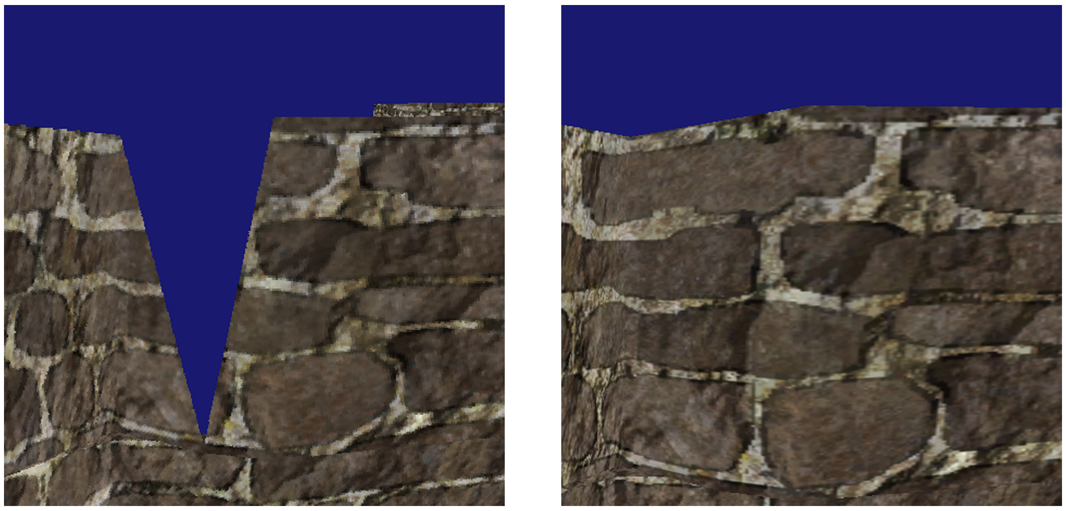

(a) Wall segment with triangle-shaped hole (b) Wall segment as it was intended

Figure 6.4.: Due to the limited precision of floating-point numbers some intersections are mistakenly rejected. This causes some candidate lists to miss an entry when building the SVO which in turn produces a hole in the object.

is used to calculate a value for λ. The resulting point of intersection should be set to

$$\begin{pmatrix} y_1 \\ x_2 + \lambda \cdot d_2 \\ x_3 + \lambda \cdot d_3 \end{pmatrix}$$

instead of

$$\begin{pmatrix} x_1 + \lambda \cdot d_1 \\ x_2 + \lambda \cdot d_2 \\ x_3 + \lambda \cdot d_3 \end{pmatrix}$$

and this point should be stored and reused as is when necessary. Even though both vectors are mathematically the same they may differ in the first component due to floating-point numbers not being infinitely precise.

The second place that caused a lot of problems was the calculation of the texture coordinates when building the SVO. In order to calculate those the parameter form of a plane is used to calculate the values λ and μ which have to satisfy certain properties such as being non-negative as explained in chapter 5. When checking those properties a small amount of slack should be allowed in order to not mistakenly remove triangles from a child's candidate list. While this alone does not fix all holes (other locations with similar boundaries checks are affected as well) it fixed the majority of holes.

6.4. Recap

This chapter explained how the hybrid renderer proposed in the previous chapter was implemented. Based on DirectX 11 the implementation is able to rasterize and ray trace on a GPU. After rendering each object, including those that are ray traced, the depth buffer is in a valid state reflecting all the content that is on screen. Floating-point precision, though not an issue in many applications, actually turned out to be a source of problems when creating and rendering SVOs.

7. Evaluation

This chapter serves as an analysis of the hybrid renderer's implementation. The benchmark features of the renderer and the SVO creation tool were run on two separate systems. The beginning of this chapter describes these two systems. This description is followed by details on the test data, namely how many polygons and vertices the original polygon meshes consisted of, how long it took to convert the background into an SVO and how much memory was required for each object. This is then followed by a rendering benchmarks and a discussion thereof. Like the preceding chapters the end of this chapter briefly summarizes its content.

7.1. Test setup

Both test systems ran Microsoft Windows 8.1 Pro 64-bit. One test system was build as a cheap low-end systems while the other was intended to be an expensive high-end system. Both systems were set up for regular use which means they had software installed such as third-party anti-virus software, third-party firewall software and other tools a user might use such as a browser, e-mail cient, office package and a tool to synchronize files with a cloud storage. During benchmarking no other software was running except for services which run as background tasks. The anti-virus and firewall processes were terminated completely for the benchmarks. The actual hardware configuration of each system can be found in table 7.1.

7.2. Polygon meshes

Two polygon meshes were used to construct all the test cases. Bob, the medieval character, was already available in the MD5 format and the accompanying background was converted into the same format. Table 7.2 shows how many primitives each mesh was made of in said format. It also shows how much memory was used on the CPU, including all pointers etc., and how much memory was used on the GPU. The GPU only needs parts of the data as animations are computed on the CPU and the modified vertices are then send to the GPU.

The background was not actually rendered as polygon mesh and the values in the table indicate how much it would have required if it was rendered as such. The implementation does not support instancing properly and so the costs for the character are per instance of him being loaded.

The Bob mesh uses five textures, one with a resolution of 512x512, one with 512x256 pixels and three textures with a 256x256 resolution. Using DXT1, a block compression

Component	**Low-end system**	**High-end system**
CPU	AMD A10-5800K (4x 3.8 GHz)	Intel Core i5-4670 (4x 3.4 GHz)
CPU RAM	2x 4GB DDR3 (2133 MHz)	2x 8GB DDR3 (1600 MHz)
GPU	AMD Radeon HD 7660D (800 MHz)	AMD Radeon R9 290 (1040 MHz; overclocked)
GPU RAM	*shared with CPU*	4GB GDDR5 (1250 MHz)
Mass storage	500GB SATA HDD (7200 rpm)	500GB SATA SSD

Table 7.1.: Test systems for benchmarking. The low-end system was also used for development. The GPU used in the high-end system is known to underclock itself if it gets too hot. It was sufficiently cooled to prevent this.

	Bob	**background**
no. of joints	33	1
no. of submeshes	6	3
no. of vertices	880	215
no. of triangles	1027	249
no. of weights	1264	194
no. of frames	142	0
memory (CPU) [in bytes]	219782	20652
memory (GPU) [in bytes]	34322	8374

Table 7.2.: The properties of the two polygon meshes which were used in creating the test scene.

format natively supported by DirectX 11 GPUs, 4 bit are required for each pixel. Also accounting for mip-maps this means the textures for this mesh need 384 kB of memory on the GPU.

The background uses two textures with 256x256 pixels and one texture with 512x256 pixels. Using the same compression and also generating mip-maps means that the textures for this mesh would require 171 kB GPU memory if rendered via rasterization.

7.3. Sparse voxel octrees

The background, as a static object, was converted into an SVO and rendered as such. For testing purposes Bob was converted as well. Each mesh was converted to SVOs with the maximum depths 8 to 11 ten times. Unless otherwise noted the results shown are averages of these ten iterations.

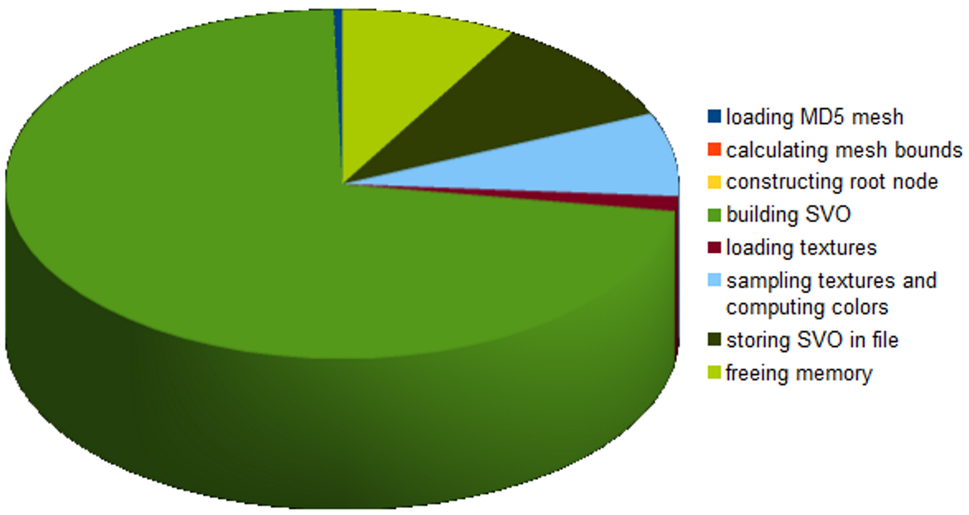
7.3.1. Creation time and file size

Figure 7.1 shows the time spent on the various steps during SVO creation in relation to each other. The majority of the time was spent on actually building the SVO which means on performing intersection tests and splitting up cubes into smaller ones. Computing the colors, storing the result in a file and freeing up the used memory were the only other steps which required any significant time. Each of those tasks required almost the same amount of time. The relative distribution of the time required for each task varies only little with regard to the maximum tree depth.

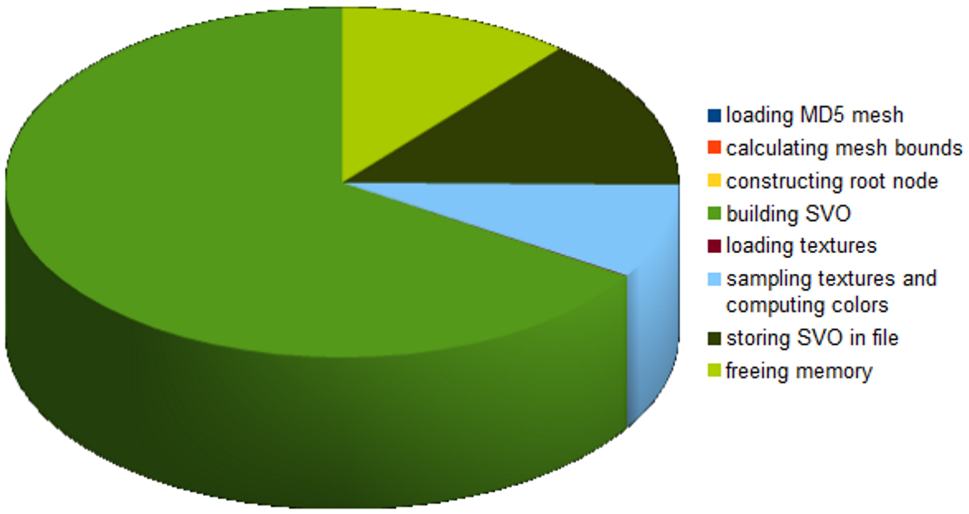
Table 7.3 lists the total amount of time required to build the SVOs as well as their file sizes. The time required to convert a polygon mesh to an SVO, especially for the more detailed SVOs, was high enough to require pre-computing the conversion. Storing each object only in their polygon mesh form and doing the conversion on start-up was no viable option even though there are only two different kinds of object in the scene. But the conversion times were still low enough to be done in a very reasonable amount of time when creating new content for an application.

Slow conversion means that the file sizes become very relevant as any application which wants to use SVO ray tracing has to include the pre-computed SVOs when being distributed. Looking at the file sizes, in particular those needed for the trees with the highest depth, this means that the total file size of the application grows rather quickly with each unique object added. This means large local storage costs for users and potentially long download times when distributing the application digitally, which becomes an increasingly popular method of distribution.

The file size also reflects the amount of memory required by the renderer at runtime. The renderer uses the exact same bit stream as is stored on the hard drive by loading the file as one large binary object. Individual 32-bit words of this binary object are then addressed as required during tree traversal. This object resides in the memory of the same processing unit that also performs the rendering. Modern GPUs have multiple GBs of RAM but even a handful of SVOs may already fill the RAM completely. This means an application which uses SVOs for more interesting scenes needs to find a clever



(a) maximum tree depth = 8



(b) maximum tree depth = 11

Figure 7.1.: Relative amount of time spent on the various steps during SVO creation of the background on the high-end system. The distribution is almost independent of the maximum depth.

mesh	depth	Low-end system	High-end system	file size
Bob	8	0.698	0.377	1.7
Bob	9	2.141	1.080	7.0
Bob	10	6.953	3.760	28.2
Bob	11	27.340	14.946	113.1
background	8	0.458	0.246	1.5
background	9	1.668	0.791	6.0
background	10	5.489	2.993	24.1
background	11	22.255	12.243	96.4

Table 7.3.: Total time in seconds required to build the SVOs on each system. The resulting file sizes in MB are included as well.

way of streaming only the relevant data to the GPU without copying every single object fully into the GPU's memory.

An interesting fact to observe is that conversion times and file sizes increase roughly by a factor of 3-4x when increasing the depth by one level. In particular the file sizes increase by a factor very close to 4x every time. This points at every other child node being empty with work (intersection tests, color computation, etc.) only continuing for four out of eight children each time a node is split up. This can be explained by the nature of the data being converted. Polygon meshes are sets of polygons with each polygon describing a 2D surface in a 3D space. This means each polygon could actually be described by an orientation, location and a quadtree. Quadtrees are 2D version of octrees and only have four children per node.

This leads to the question of how much redundancy there is in an SVO file. Using the free compression tool 7-Zip[11] this can be quickly and easily evaluated. Compression all eight SVOs to the ZIP format using the "Maximum" compression level results in a reduction in file size by 80.3%. Using the 7z format instead increases the reduction to 87.5%. On the one hand this means that an application using SVOs can actually be reasonably distributed in a compressed format. On the other hand it means that a lot of memory is actually wasted at runtime despite an SVO already being far more efficient than simply storing a full voxel grid. Using the values for depth = 11 from table 7.3 the SVOs already need less than 1% of the space a full voxel grid of $2048^3 = (2^{11})^3$ voxels would need.

7.3.2. Memory usage during conversion

While not systemically measured the memory usage of the conversion tool's process was observed during development. The memory usage can quickly rise up to several GBs, in particular when creating deep trees. When trying to create an SVO with maximum depth set to 12 the process actually began to swap pages from the physical memory to the hard drive on the low-end system. This is an indicator that the conversion algorithm, while

[11] `http://7-zip.org/`

reasonably fast in all evaluated test cases, could slow down significantly when creating even more detailed models than have been created for this evaluation due involvement of the slow mass storage device of the system. Further tests were not performed because the low-end system repeatedly crashed when trying to create more detailed trees and existing data already was sufficiently good to draw conclusions about the behavior at higher levels of detail. The reason for the crashes was most likely a defect in the HDD controller or the HDD itself as performing lots of I/O operations caused the system to become unstable even when using other applications.

7.4. Rendering

The core of this thesis is the hybrid renderer. The test scene shows an animated medieval character, Bob, on an appropriate medieval background. The test scene always renders one SVO for the background and one, three or eight characters in addition to that. In scenes with one or three characters all character are in front of the background. Since real scenes may include characters which are in the vicinity and so send to the GPU for rendering but who are occluded by other objects, choosing to render eight characters results in five of those characters being placed behind the background object. To measure the performance of the renderer the built-in benchmarking feature, which iterates over all possible settings and plays every demo with each combination of settings ten times, was used. The results presented in this section are always an average of the ten iterations unless noted otherwise. The demos are numbered 1 through 4 with 1 being the panning, 2 being the zoom, 3 being the rotation along the half circle and 4 being the stationary rotation as described in the implementation chapter.

In addition to evaluating performance a subjective image quality evaluation was done. The only parameter which affects image quality is the maximum depth of the SVO and so the evaluation was done by comparing screenshots rendered with different depth settings.

7.4.1. Performance

The performance was measured as elapsed time to render a frame. A measure for performance usually used by users, such as gamers, is the frame rate. Typically anything below 30 FPS is perceived as not smooth. This value is a little higher than frame rates common in cinema (24 FPS). This difference comes from the fact that 3D graphics rendered in real-time often lack motion blur and so are perceived as less smooth. Additionally some users, especially gamers who play action-oriented or rhythm games, even demand frame rates of 60 FPS and in rare cases even more than that. This means if a frame takes more than 33.3ms (30 FPS) to render the result is not acceptable as it lacks smoothness. Times below 16.7ms (60 FPS) are desirable.

Table 7.4 shows the worst case performance at the lowest detail level at a fairly low resolution of 640x480 pixels. When rendering the background first early depth tests are actually not useful. But they were enabled to test the worst case performance and because early depth tests and the render order have little impact on the performance

demo	low-end system (CPU)			high-end system (CPU)			low-end system (GPU)			high-end system (GPU)		
	min	avg	max	min	avg	max	min	avg	max	min	avg	max
1	79.5	300.4	447.5	49.8	157.9	232.5	16.4	73.8	122.7	1.3	4.1	7.5
2	268.8	432.6	545.0	130.6	201.2	260.8	52.1	109.1	165.3	2.5	5.4	8.4
3	271.1	329.2	411.6	133.1	164.6	200.4	78.6	96.4	115.7	4.0	5.0	6.8
4	258.6	368.6	517.3	133.7	178.4	230.5	38.4	72.6	114.5	1.9	3.9	6.6

Table 7.4.: The rendering peformance on both systems. Minimum, average and maximum times are shown for each demo and for rendering the SVO on either the CPU or the GPU. All times are in milliseconds. A resolution of 640x480 was chosen with the other settings set to SVO depth = 8, early depth testing, no. of characters = 1, background rendered first.

anyway as shown later. The results show that CPUs are not suitable for ray tracing SVOs at all. Even the high-end system could not achieve a smooth performance under the best of circumstances and an improvement by a factor of more than 5 would be necessary. While the low-end system struggled to achieve smooth frame rates even when rendering completely on the GPU, the results for the high-end system look promising. Even the worst times are far below the limit of 16.7ms indicating that there is room for more details. The difference between the GPU performance on both systems is rather interesting as well. The high-end system is faster by a factor of almost 20 even though it has neither 20 times the shader performance nor 20 times the memory bandwidth. The conclusion of this test is that anything but a high-end GPU is incapable of performing SVO ray tracing in real time at this time.

The next test focused on level of detail, namely resolution and tree depth. The results are displayed in table 7.5. Demo 2, the zoom, performed the worst out of all four demos regardless of resolution and tree depth. This may be due to the fillrate, how many texture samples can be read and how many pixels can be written to the screen, becoming a bottleneck once the camera is close to the character it zooms to. Aside from this outlier the other demos all had similar performance with the times increasing, as expected, with both the resolution and tree depth. On average smooth frame rates were achieved up to a resolution of 1280x720 for all SVO depths and with depth = 10 even achieving almost 60 FPS. On 1920x1080 only depths up to 10 resulted in smooth frame rates with demo 2 already stuttering at level of detail. The performance scaled quite well with the tree depth. Even though increasing the depth by 1 increases the size of the tree by a factor of four the rendering times only increase by a factor of less than 2, with an increase that is often 50% or less. Similarly resolution increases increase the rendering times by less than the increase in pixel count. Going from 640x480 to 1280x720 means rendering three times as many pixels and going from 1280x720 to 1920x1080 increases the pixel count by a factor of 2.25. But the rendering times only increase by factors of roughly 2. The good scaling may be explained by inefficiencies in the GPU pipeline. When increasing the work load the GPU has more opportunities to hide inefficiencies

resolution	**demo**	**depth = 8**		**depth = 9**		**depth = 10**		**depth = 11**	
		min	max	min	max	min	max	min	max
640x480	1	4.1	7.4	5.7	12.5	7.8	19.5	9.9	26.7
640x480	2	5.3	8.4	7.9	13.6	11.3	20.5	15.9	30.3
640x480	3	4.9	6.9	6.9	9.8	9.5	14.3	12.0	18.8
640x480	4	3.8	6.6	5.6	10.7	8.1	17.6	11.7	28.3
1280x720	1	8.4	14.5	12.0	23.4	16.3	33.9	21.6	47.5
1280x720	2	11.1	16.7	16.5	27.5	23.7	42.0	33.0	61.8
1280x720	3	9.5	11.2	13.2	15.7	17.7	22.2	23.5	31.2
1280x720	4	7.9	13.0	11.4	21.1	16.6	33.6	23.5	50.9
1920x1080	1	14.5	23.6	20.5	36.5	27.8	54.4	36.7	79.4
1920x1080	2	20.7	30.9	29.8	48.4	42.3	73.0	58.5	109.3
1920x1080	3	16.9	20.1	23.3	28.0	31.4	39.4	41.4	56.6
1920x1080	4	14.7	21.8	20.3	33.6	28.6	52.6	40.1	79.0

Table 7.5.: The rendering performance at different levels of detail. Average and maximum times in milliseconds are shown for different resolutions and different tree depths. All times are from GPU rendering on the high-end system with early depth testing disabled, rendering the environment last and three characters.

like latency by quickly switching between tasks and so keeping the shader cores occupied and getting closer to the peak memory bandwidth.

The last part of the performance analysis focused on the impact of the number of characters, early depth testing and the rendering order. The results are shown in table 7.6. With the standard deviation being 0.5ms or more the results do not differ significantly suggesting that ray tracing an SVO takes by far the majority of time with rasterized objects having little impact on the performance. This may be partially due to the low polygon count of each model and partially due to rasterization itself being significantly more efficient when no advanced effects such as lighting are implemented. It is interesting though that the early or late depth testing seems to have no significant impact on the performance. Figure 7.2 explores this in more detail. It shows the rendering time of individual frames of the second demo. As expected the graphs are symmetrical due to the demo itself being symmetrical. Surprisingly the two graphs are almost exactly the same even though early depth testing requires more operations per ray when no early termination is possible and potentially fewer operations when early termination is in fact possible. This points to memory bandwidth being the limiting factor.

7.4.2. Image quality

Image quality was only evaluated subjectively due to time constraints. Figures 7.3 and 7.4 show the same scene and camera angle with maximum tree depths at different levels.

early depth test	**demo**	**1 character**		**3 characters**		**8 characters**		
		bg. first	bg. last	first	last	first	middle	last
yes	1	21.8	21.7	21.8	21.6	21.8	21.6	21.6
yes	2	33.2	33.1	33.3	33.0	33.3	33.0	33.0
yes	3	23.7	23.6	23.6	23.6	23.7	23.6	23.6
yes	4	23.7	23.7	23.7	23.5	23.8	23.5	23.5
no	1	21.6	21.6	21.6	21.6	21.7	21.6	21.6
no	2	33.0	33.0	33.0	33.0	33.1	33.0	33.0
no	3	23.5	23.5	23.5	23.5	23.5	23.5	23.5
no	4	23.5	23.5	23.5	23.5	23.6	23.6	23.5

Table 7.6.: The rendering performance for different scene settings. Average times in milliseconds are shown for different numbers of characters, rendering order and early depth test on/off. The render order can be either rendering the background first, rendering it last or rendering it in between the three characters in front of it and the five characters occluded by it. All times are from the high-end system with GPU rendering at 1280x720 and depth = 11.

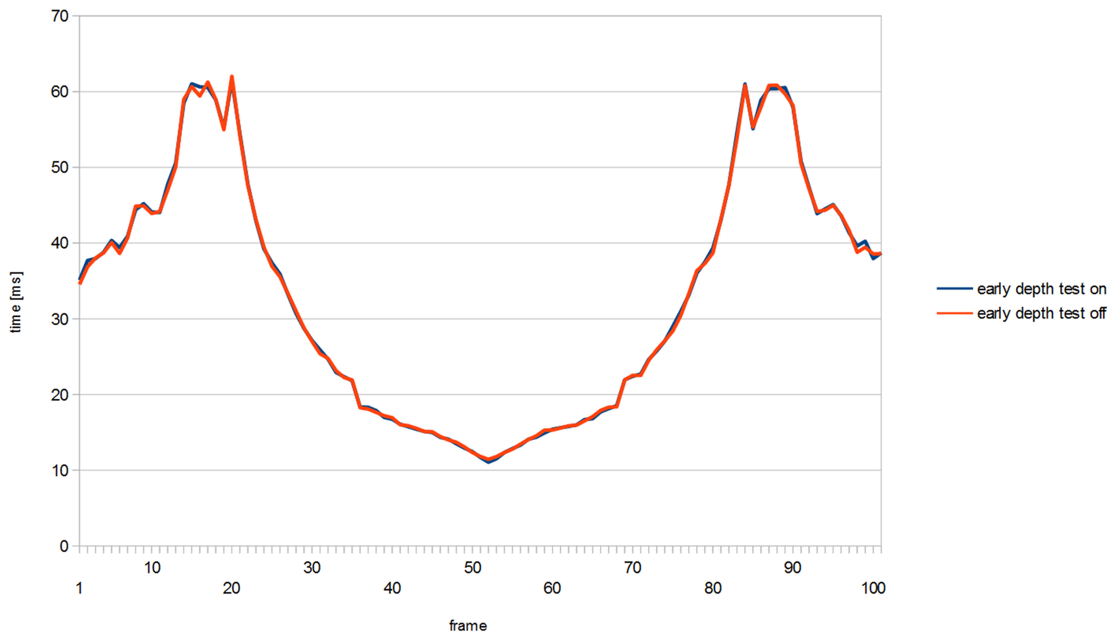


Figure 7.2.: Rendering times for individual frames of demo 2 (Zoom) at 1280x720 on the high-end system with GPU rendering, tree depth = 11, one character and rendering the background last.

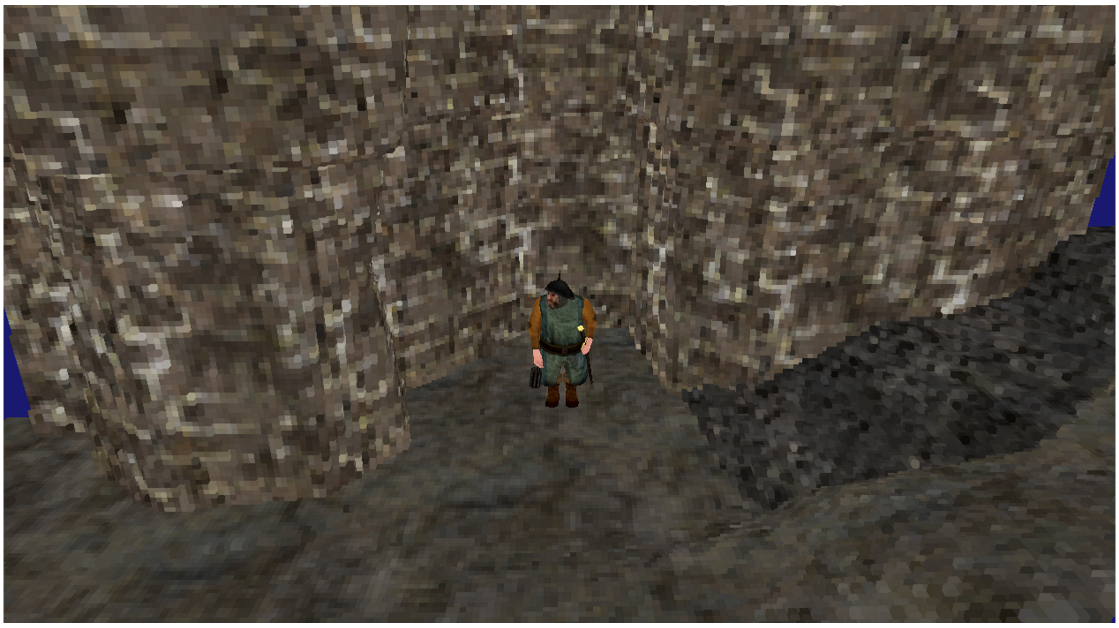
Output resolution was 1280x720 pixels. Depth = 8 is completely unacceptable. The cuboid shapes of the voxels are quite obvious and it is hard to make out any details in the environment. Depth = 9 already improves upon that significantly by making it possible to see details. Yet the cuboid shapes are still very visible. Increasing the depth one step further already resulted in a quite pleasing picture. Details are well-defined and the cuboid shapes are only visible to a trained observer. However aliasing also becomes visible on one surface (the wall to Bob's right that he is looking at) though it is not much of a problem. At depth = 11 the image appears even sharper with even better defined details. Cuboid shapes are gone with no additional aliasing appearing. The appendix does have an additional comparison gallery with the same results except that the cuboid shapes do not fully disappear at depth = 11 and that they are more obvious at depth = 10. The camera has been moved closer to Bob for the screenshots in the appendix.

The two highest depth levels deliver decent to good image quality. Yet depending on the camera's location even those levels of detail are insufficient as depicted in figure 7.5. Without any filtering the cuboid shapes of the voxels will eventually become visible.

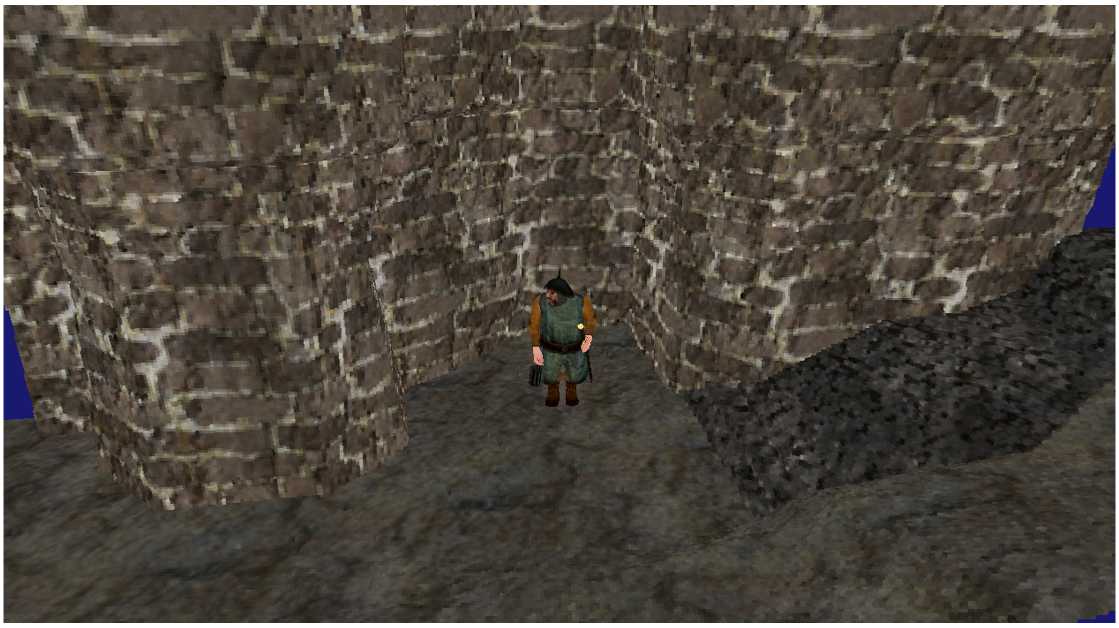
Even though there is aliasing it turned out to be a minor problem in the test scene.

7.5. Recap

This chapter evaluated the polygon mesh to SVO conversion tool as well as the hybrid renderer itself. The conversion tool performed well though there are indications of the performance degrading seriously at higher detail levels due to swapping to the hard drive. The renderer was able to achieve good performance even at HD resolutions with detailed trees. Image quality turned out to be mostly positive as well when the deeper trees were used. Memory usage for detailed SVOs is still a concern.

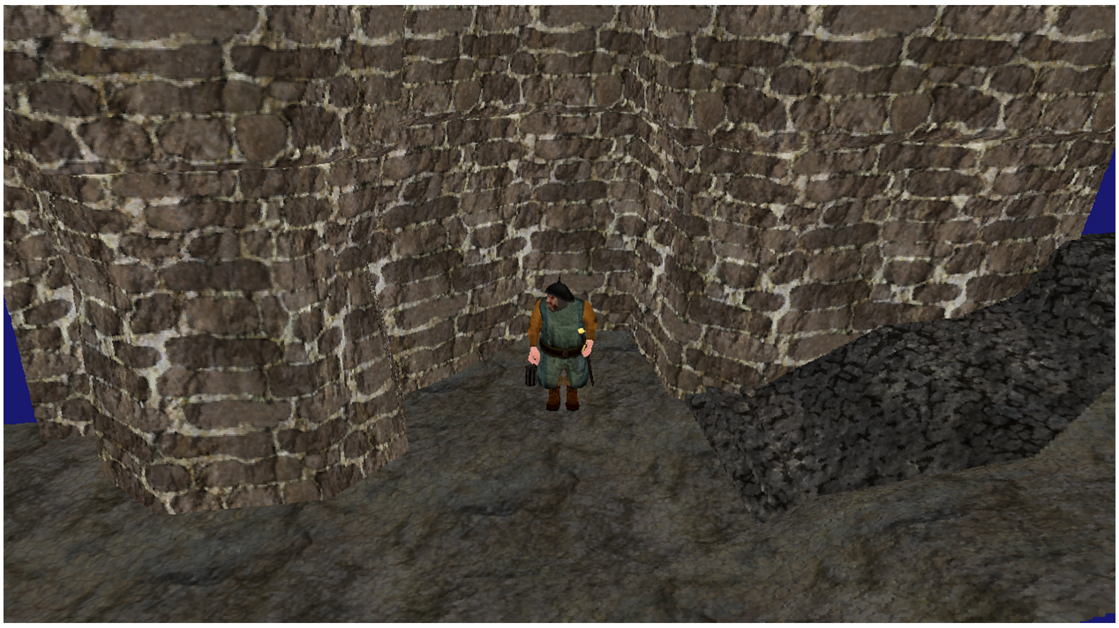


(a) maximum tree depth = 8



(b) maximum tree depth = 9

Figure 7.3.: Screenshots taken at 1280x720


(a) maximum tree depth = 10

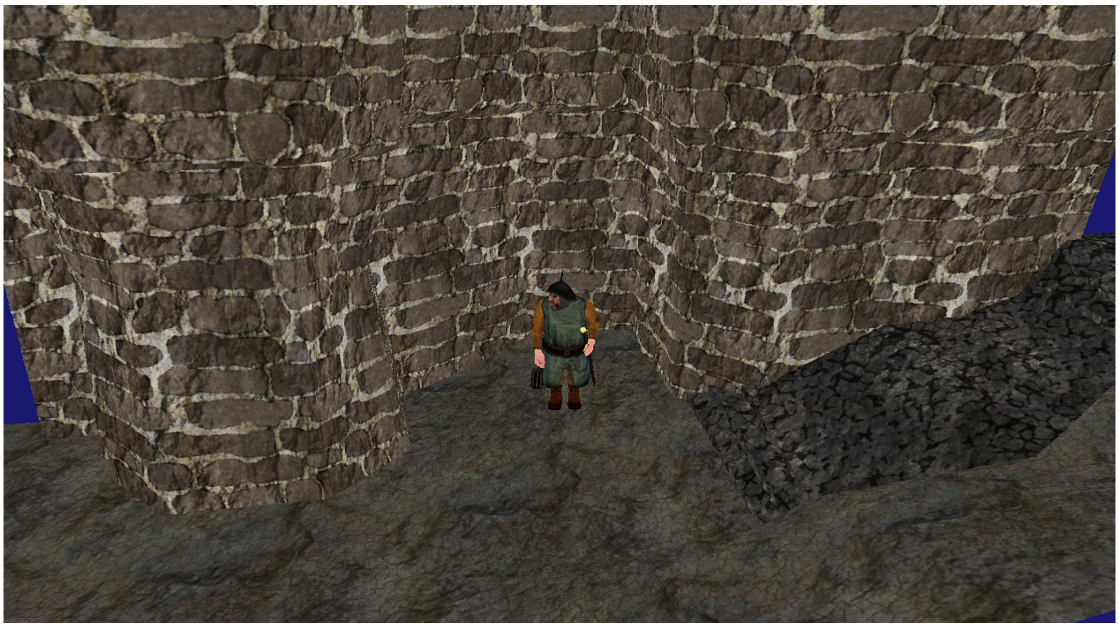

(b) maximum tree depth = 11

Figure 7.4.: Screenshots taken at 1280x720

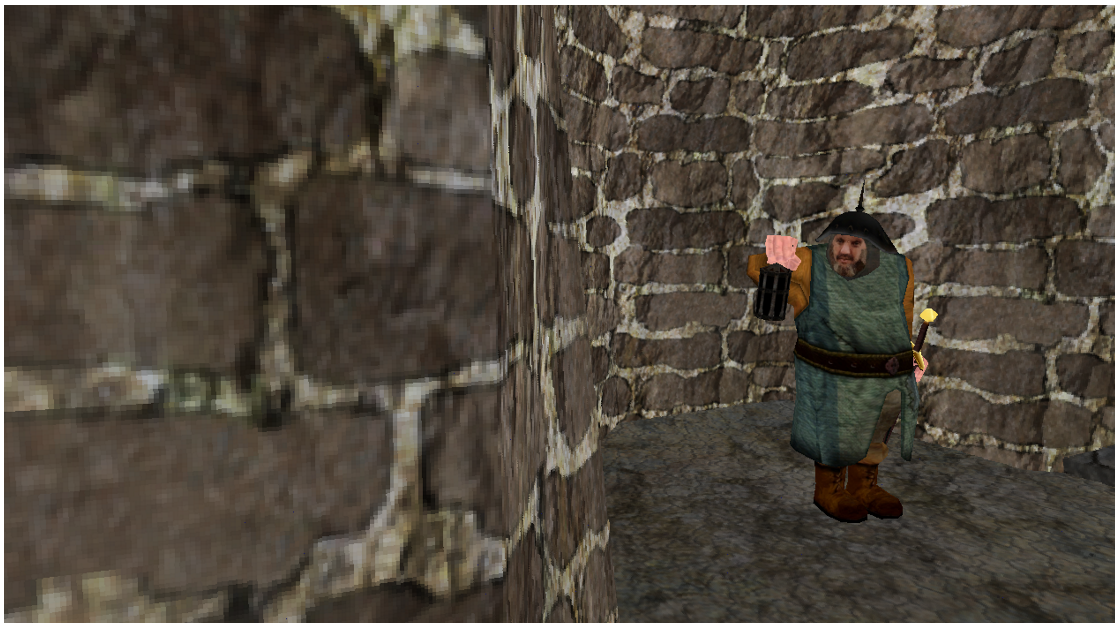


Figure 7.5.: Screenshot taken at 1280x720 with depth 11. Even the most detailed SVOs eventually reveal the cuboid shape of the underlying voxels when the camera gets too close (left side).

8. Conclusion

The final chapter summarizes the thesis and ends with making suggestions for future work.

8.1. Summary

This thesis discussed core rendering techniques used in modern 3D graphics. First it explained how modern GPUs render polygons via rasterization. Rasterization is the sampling of surfaces. In order for objects to appear in the proper position on screen such that a correct perspective look is achieved, each object is transformed across several coordinate systems or spaces to the so called screen space in which the final sampling is performed. After sampling shader programs are used to calculate the final color of each pixel based on data read from textures. The thesis then went on by explaining ray tracing. Ray tracing renders a scene by tracing the path of a ray into the scene and determining the first object it intersects with. Ray tracing does not rely on transformations between spaces but also performs shading in the same way as rasterization does.

The main part of this thesis then followed in the form of a proposal of a hybrid renderer. This render uses both rasterization and ray tracing depending on the kind of object to be rendererd. Each technique has its own advantages in particular when complexity of an implementation is considered. Rasterization can easily render animated deformable objects while ray tracing is well suited for realistic lighting and volumetric effects such as smoke or fog. The proposed hybrid renderer is able to use both techniques in any order while still producing correct results with regard to occlusion. It is also able to utilize different kinds of primitives for each object. Rasterization is performed on textured polygon meshes, i.e. primitives describing surfaces, while ray tracing uses Sparse Voxel Octrees (SVOs), i.e. primitives describing volumes. An accompanying tool to convert polygon meshes, which are commonly used by the industry to model objects, into SVOs was also proposed.

The proposed renderer and conversion tool have also been implemented with the important, non-obvious parts of the implementation being discussed in this thesis. The implementations have been evaluated and found to perform mostly well. There are still concerns about the memory usage of SVOs during rendering but performance was already acceptable on a high-end PC. Rendering via the CPU was found to be not an option but rendering on a GPU produced smooth frame rates. Considering that no advanced lighting effects etc. were implemented the image quality was promising as well. Detailed SVOs delivered sharp images with a lot of detail and little artefacts. Solutions for the remaining artefacts were hinted at but not yet implemented or tested.

8.2. Suggestions for future works

Obviously the rendering artefacts produced by the current ray tracer have to be improved upon if building on this work in the future. As suggested previously in this thesis an attempt at fixing the aliasing artefacts can be made by implementing an early termination for the ray tracing depending on the size a voxel would have on screen. Right now the SVOs are always traversed to the leaf nodes even if an inner node already would have roughly the size of a pixel on screen considering the distance the ray is currently at.

To avoid the cuboid shapes of the voxels from becoming visible a blur filter could be added. The same idea of the voxel's size on the screen could be used. If a ray terminates at a leaf node which is larger than a pixel a marker can be set in a different buffer. This buffer can then be used as a stencil to only blur the marked areas.

Instead of storing color values in the SVO texture coordinates could be stored. This could also be used as a remedy to the artefacts the current renderer produces at the cost of losing the "easy to filter" property voxels have.

Another obvious way to improve upon this work is to implement advanced effects such as realistic lighting and evaluate whether hybrid rendering still makes sense with this additional processing. While this thesis has shown that basic hybrid rendering is possible it is still not known if it is still viable when more advanced effects are computed and if the added benefit of easier implementation of advanced lighting via ray tracing is actually worth the cost.

The SVOs themselves also offer enough room for improvement. As shown in the evaluation a lot of spaces is actually wasted on empty nodes. A more efficient data structure, e.g. one which uses an extra field in which individual bits indicate the presences of children or pointers, could help to reduce both the file size and the memory usage during rendering.

Other ideas to improve SVOs include embedding quadtrees once a single flat surface is detected during octree creation. This could not only help with memory usage but also speed up traversal. Another potential optimization is to use wavelet decomposition when storing the SVO. Right now each node has a color and each child may have a different color, i.e. nine color values are stored. Using wavelets this could be reduced to eight values reducing the space required by color values by more than 10%.

This thesis only considers hybrid rendering based on objects. It may also be viable or even better to perform rasterization as the first step in rendering and then continue with ray tracing during shading to implement advanced effects. This would imply that only polygons are used as primitives due to the rasterization step but it may still turn out to be an interesting option.

Appendices

A. Projective transformations

Rasterization-based renderers rely on several spaces/coordinate systems and transformations between them. Those transformations, e.g. translation or rotation, are represented by matrices which is made possible through the use of homogeneous coordinates. This means that every vector $(\frac{x}{w}, \frac{y}{w}, \frac{z}{w})$ is represented by the vector $v = (x, y, z, w)$ with w being normalized to 1 after every operation that might affect this value. For each vector v and transformation matrix M the result of the transformation is determined by simply calculating $M \cdot v$.

All relevant transformations and what their matrices look like are explained in the following sections. More detailed explanations can be found in math text books which deal with projective geometry. Though knowing about projective geometry in that level of detail is not necessary when implementing rasterization. Books on computer graphics, e.g. [23], [9] and [21] as referenced in the beginning of the rasterization chapter, usually contain enough information on projective geometry and thus transformations to understand and being able to implement rasterization.

A.1. Scaling

Scaling a vector (x, y, z, w) to $(\lambda_x x, \lambda_y y, \lambda_z z, w)$ is achieved by the following matrix:

$$\begin{pmatrix} \lambda_x & 0 & 0 & 0 \\ 0 & \lambda_y & 0 & 0 \\ 0 & 0 & \lambda_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

A.2. Translation

Translating a vector (x, y, z, w) to $(x + v_x, y + v_y, z + v_z, w)$ is achieved by the following matrix:

$$\begin{pmatrix} 1 & 0 & 0 & v_x \\ 0 & 1 & 0 & v_y \\ 0 & 0 & 1 & v_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

A.3. Rotation

Rotating a vector (x, y, z, w) about the X axis by an angle α is achieved by the following matrix:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The same operation but with the Y axis as a rotation axis:

$$\begin{pmatrix} \cos\alpha & 0 & \sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Again, same operation but with the Z axis as a rotation axis:

$$\begin{pmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Arbitrary rotations can be achieved by chaining the above three rotations together via matrix multiplication. By choosing the proper order and proper angles any desired result can be achieved. It may be easier and more efficient to use an arbitrary rotation axis though. In such cases the matrix for rotating a vector about the axis $u = (u_x, u_y, u_z)$ (in 3D space; u must be a unit vector) by an angle α is as follows:

$$\begin{pmatrix} \cos\alpha + u_x^2(1-\cos\alpha) & u_xu_y(1-\cos\alpha) - u_z\sin\alpha & u_xu_z(1-\cos\alpha) + u_y\sin\alpha & 0 \\ u_yu_x(1-\cos\alpha) + u_z\sin\alpha & \cos\alpha + u_y^2(1-\cos\alpha) & u_yu_z(1-\cos\alpha) - u_x\sin\alpha & 0 \\ u_zu_x(1-\cos\alpha) - u_y\sin\alpha & u_zu_y(1-\cos\alpha) + u_x\sin\alpha & \cos\alpha + u_z^2(1-\cos\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

A.4. Perspective projection

As mentioned in the chapter on rasterization the sampling algorithm used actually calculates an orthogonal projection. To give it the appearance of a perspective projection an additional transformation is applied to vertices which are already in camera space. This transformation makes sure that the frustrum which contains every visible polygon becomes a cube whose center is the origin of the coordinate system and whose sides are all two units long.

Calculating this transformation matrix requires four parameters:

- the screen's aspect ratio $ar = \frac{\text{screen width}}{\text{screen height}}$

- the vertical FOV: the vertical angle α of the camera's view frustrum
- the location of the image plane $nearZ$ onto which the visible part of the scene is projected
- the distance $farZ$ of objects that are too far away to be visible

Introducing $farZ$ helps preventing visual artifacts when objects become too small. It also helps with performance as polygons which are outside the previously mentioned cube are not sampled from at all. The transformation matrix is as follows:

$$\begin{pmatrix} \frac{1}{ar \cdot \tan\frac{\alpha}{2}} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan\frac{\alpha}{2}} & 0 & 0 \\ 0 & 0 & \frac{-(nearZ+farZ)}{nearZ-farZ} & \frac{2 \cdot nearZ \cdot farZ}{nearZ-farZ} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

There is one problem with this matrix. For a proper projection the X and Y values have to be divided by the Z value. Unfortunately this can not be expressed by a matrix. To account for this the Z value is copied to the W value (last row in the matrix) and rasterization hardware performs this final operation before starting the sampling. At this point all visible points have X, Y and Z values in $[-1, 1]$. By taking the screen's aspect ratio and the desired vertical FOV into account all objects appear natural instead of appearing deformed after sampling.

B. Additional screenshots

Figures B.1, B.2, B.3 and B.4 are comparisons similar to the one used for evaluating the image quality in chapter 7. Refer to that particular section for details.

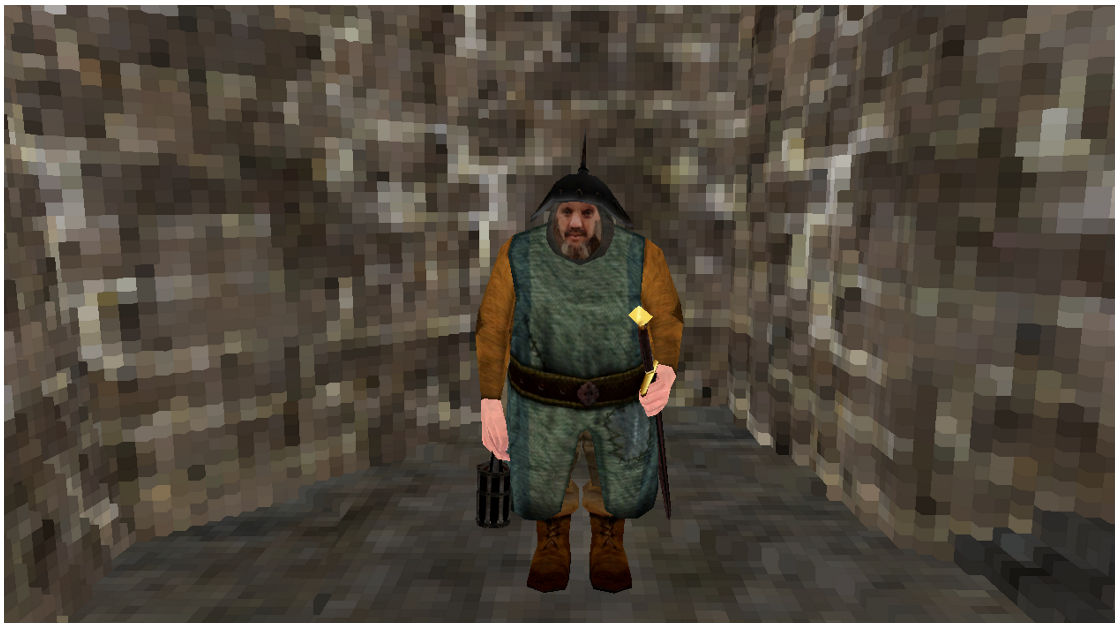


Figure B.1.: Screenshot taken at 1280x720, maximum tree depth = 8

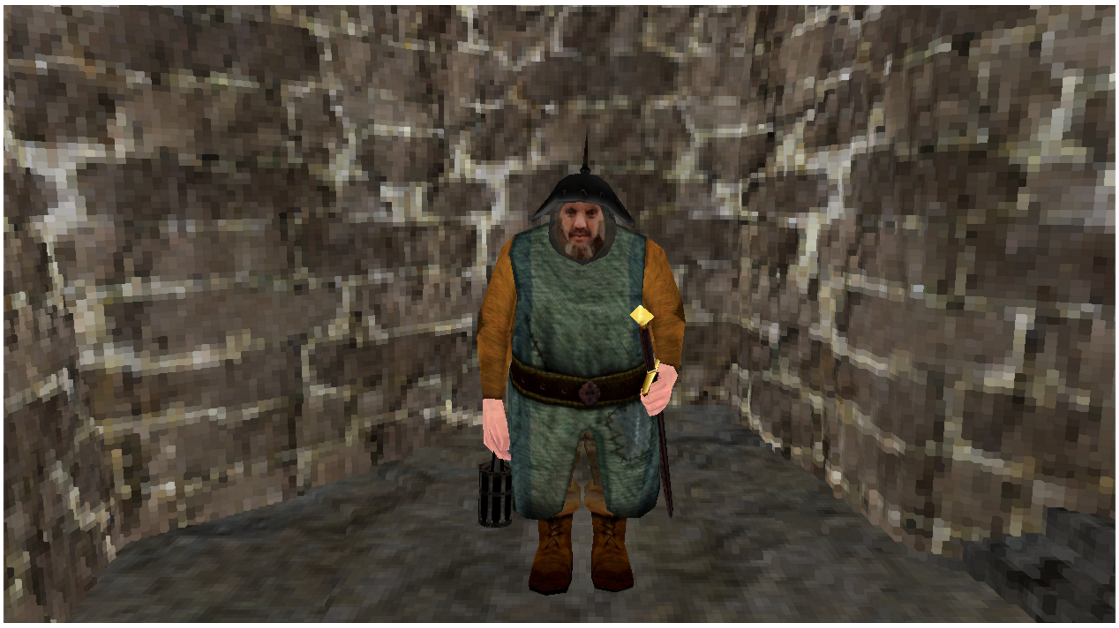


Figure B.2.: Screenshot taken at 1280x720, maximum tree depth = 9

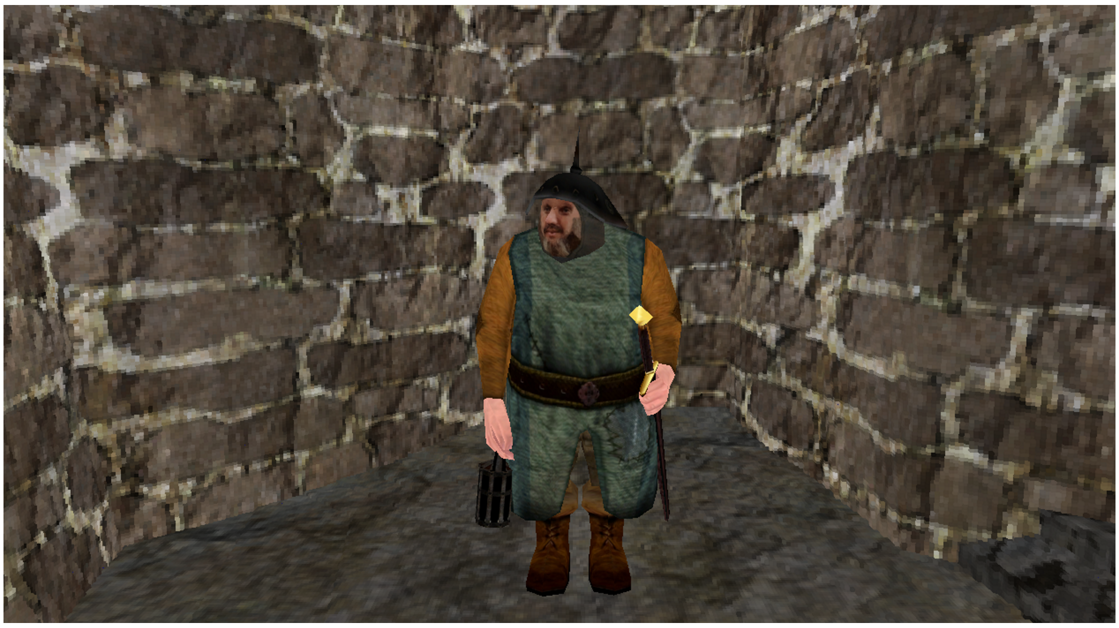


Figure B.3.: Screenshot taken at 1280x720, maximum tree depth = 10

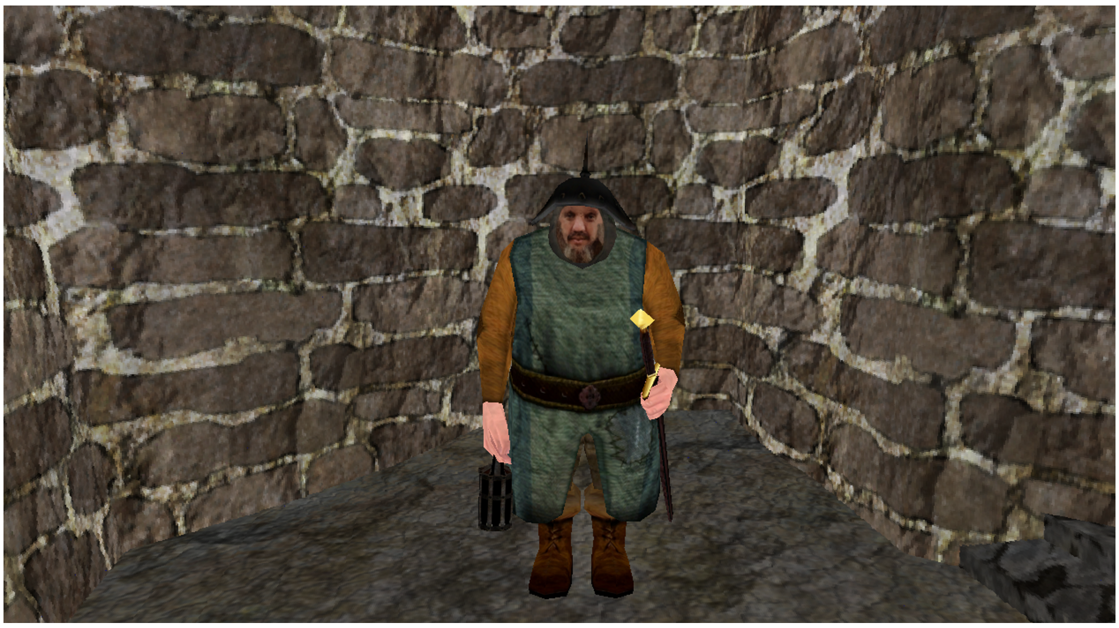


Figure B.4.: Screenshot taken at 1280x720, maximum tree depth = 11

Bibliography

[1] Epic games' tim sweeney explains lack of global illumination in unreal engine 4. `http://www.playstationgang.com/epic-games-tim-sweeney-explains-lack-of-global-illumination-in-unreal-engine-4/`. Online; accessed 28-February-2014.

[2] Outcast (video game) - wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Outcast_(video_game)`. Online; accessed 28-February-2014.

[3] Attila T Áfra. Incoherent ray tracing without acceleration structures. In *Eurographics (Short Papers)*, pages 97–100, 2012.

[4] Dennis Bautembach. Animated sparse voxel octrees. *Thd Thesis. University of Hamburg*, 2011.

[5] Louis Bavoil and Miguel Sainz. Multi-layer dual-resolution screen-space ambient occlusion. In *SIGGRAPH 2009: Talks*, page 45. ACM, 2009.

[6] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 15–22. ACM, 2009.

[7] Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt. The triangle processor and normal vector shader: A vlsi system for high performance graphics. *SIGGRAPH Comput. Graph.*, 22(4):21–30, June 1988.

[8] Kirill Garanzha and Charles Loop. Fast ray sorting and breadth-first packet traversal for gpu ray tracing. In *Computer Graphics Forum*, volume 29, pages 289–298. Wiley Online Library, 2010.

[9] J. Gomes, L. Velho, and M.C. Sousa. *Computer Graphics: Theory and Practice*. Ak Peters Series. Taylor & Francis, 2012.

[10] Samuli Laine, Timo Aila, Tero Karras, and Jaakko Lehtinen. Clipless dual-space bounds for faster stochastic rasterization. In *ACM Transactions on Graphics (TOG)*, volume 30, page 106. ACM, 2011.

[11] Samuli Laine and Tero Karras. Efficient sparse voxel octrees. *Visualization and Computer Graphics, IEEE Transactions on*, 17(8):1048–1059, 2011.

[12] Timothy Lottes. Fxaa. *NVIDIA white paper*, 2011.

[13] Josiah Manson and Scott Schaefer. Wavelet rasterization. In *Computer Graphics Forum*, volume 30, pages 395–404. Wiley Online Library, 2011.

[14] Martin Mittring. The technology behind the "unreal engine 4 elemental demo". *part of "Advances in Real-Time Rendering in 3D Graphics and Games," SIGGRAPH*, 2012.

[15] Benjamin Mora. Naive ray-tracing: A divide-and-conquer approach. *ACM Transactions on Graphics (TOG)*, 30(5):117, 2011.

[16] Jacob Munkberg and Tomas Akenine-Möller. Hyperplane culling for stochastic rasterization. In *Eurographics (Short Papers)*, pages 105–108, 2012.

[17] Hubert Nguyen. *Gpu Gems 3*. Addison-Wesley Professional, first edition, 2007.

[18] NVIDIA. Csaa - coverage sampled aa. `http://www.nvidia.com/object/coverage-sampled-aa.html`. Online; accessed 26-February-2014.

[19] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005.

[20] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-d shapes. *SIGGRAPH Comput. Graph.*, 24(4):197–206, September 1990.

[21] Peter Shirley and Steve Marschner. *Fundamentals of Computer Graphics*. Ak Peters Series. Taylor & Francis, 2009.

[22] László Szécsi and Dávid Illés. Real-time metaball ray casting with fragment lists. In *Eurographics (Short Papers)*, pages 93–96, 2012.

[23] John F. Hughes; Andries van Dam; Morgan McGuire; David F. Sklar; James D. Foley; Steven K. Feiner; Kurt Akeley. *Computer Graphics: Principles and Practice*. Addison-Wesley Professional, third edition edition, 2013.

[24] Sven Woop, Jörg Schmittler, and Philipp Slusallek. Rpu: a programmable ray processing unit for realtime ray tracing. In *ACM Transactions on Graphics (TOG)*, volume 24, pages 434–444. ACM, 2005.

[25] Cevat Yerli and Anton Kaplanyan. Future graphics in games. In *High Performance Graphics*, 2010. `http://www.crytek.com/download/Notes.ppt`. Online; accessed 28-February-2014.