MASTER'S THESIS | LUND UNIVERSITY 2015

Real-time screen space reflections and refractions using sparse voxel octrees

Filip Nilsson

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2015-09

Real-time screen space reflections and refractions using sparse voxel octrees

Filip Nilsson
ada09fni@student.lu.se

May 5, 2015

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Michael Doggett, michael.doggett@cs.lth.se
Examiner: Tomas Akenine-Möller, tomas.akenine-moller@cs.lth.se

Abstract

This thesis explores the data structure known as sparse voxel octree and how it can improve the performance of real-time ray tracing. While ray tracing is an excellent way of producing realistic effects in computer graphics it is also very computationally heavy. Its use in real-time applications such as games and simulators is therefore limited since the hardware must be able to render enough frames per second to satisfy the users. The purpose of an octree is to reduce the amount of intersection tests each ray needs significantly. This thesis will explain the many challenges when implementing and using an octree, and also how to solve them. This includes both how to build the tree using various tests and then also how to use it with a ray tracer to produce reflections and refractions in real time.

Keywords: Sparse voxel octree, ray tracer, reflection, refraction

Acknowledgements

I would like to thank my supervisor Ass. Prof. Michael Doggett for all the support and discussions we've had.

Contents

Chapter 1
Introduction

In the real world photons from light sources bounce around on surfaces and if they hit our eyes they tell us about their journey by giving us information about the colour of the surfaces along the photons' paths. This paints the picture of what we see, with shadows, reflections, refractions and all the other effects we see each day. In offline rendering such as for movies effects it's common to simulate all these effects by using ray tracing. To determine the colour of a pixel you send a ray through it and see what it hits in the scene. Because we have models of how light rays work in the real world, if we implement them for our own rays we can achieve realistic effects.

Ray tracing is very computationally demanding. A ray tracer in its most basic form will for each frame trace one ray through each pixel, and then do an intersection test against every primitive in the scene. These many calculations per frame makes it impossible to use ray tracing in real time in any practical sense.

In this paper we will talk about two major optimizations, the Octree data structure and voxelization, and a traversal algorithm for the rays that takes advantage of these two implementations.

1.1 Background

1.1.1 Ray Tracer

To simulate photon rays it might sound more accurate to have them originate from light sources and sending them out into the scene. There are two major problems with this idea. The first is that the camera is a close to infinitely small point, so probability of a ray hitting it is close to zero. The other is that every ray not hitting us is wasted. Since we already will need several hundred thousand to million rays that hit the camera, this is not a feasible solution. By instead using rays originating from the camera out towards the scene we in a sense only trace those photon rays that hit us, from their destination backwards towards

where they originated from. This way every ray traced is one that matters for the final image. To detect what a ray intersects with we use geometry maths and the fact that a ray is a line, and that all shapes in computer graphics consist of primitives, like triangles and spheres. If we know where a line starts and in what direction it has, we can find if it intersects a sphere, and where this intersection point is, by comparing the equations for both the line and the sphere and calculating for what values they are equal. It is not enough to find an intersection point though, as there might be other primitives that will be intersected earlier and block the view. The goal with a ray trace is therefore to find the closest object to the camera that the ray hits. If we know the closest intersection point, we can let the colour of that point be the colour of the pixel the ray was traced through.

History of ray tracing

Ray tracing was first presented by Arthur Appel [1] in 1968, in the form of ray casting. His algorithm was to trace rays into the scene to find the colour of the first point hit. This meant any shape or form could be rendered, not just planar ones.

Turner Whitted made the next big step in 1979 when he presented an extension of Appel's algorithm, today referred as Whitted Ray-Tracing [2, p. 23]. He proposed using up to four rays per pixel instead of just one. The new ones were the shadow, reflection and refraction rays, and the old one is called primary ray. When the primary ray hits a point the shadow ray tests if there's anything between the point and the light source, which means the point is in a shadow. If the surface is reflective or refractive the respective new rays continue into the scene and the colour of the surface they hit gets mixed with the colour of the original hit point.

Whitted Ray-Tracing is still used today in many applications, but it has the downside of only having hard shadows and perfect reflections and refractions. To achieve effects such as motion blur, depth of field, fussy reflections and soft shadows we need to use Distributed Ray Tracing [3], first presented by Robert Cook in 1984. If we want to render soft shadows we send several rays distributed around the light source direction from the hit point and then use the average result to see how much shadow the point should receive. The two biggest drawbacks of this technique are the computational heavy demands, as the number of rays quickly grow, and the lack of indirect light.

Today most advanced ray-tracers tries to solve the Rendering Equation, first presented by James Kajiya in 1986. The equation calculates not only the direct light received by a point, but also the indirect. Since Kajiya first presented his equations there have been many different versions of it. A common one is:

$$L_o(x, \omega_o, \lambda, t) = L_e(x, \omega_o, \lambda, t) + \int_\Omega f_r(x, \omega_i, \omega_o, \lambda, t) L_i(x, \omega_i, \lambda, t)(\omega_i \cdot n) d\omega_i$$

λ: wavelength of light.
t: time.
x: hit point.
ω_o: Outgoing light direction.
ω_i: Reverse direction of incoming light.
n: Normal of surface at point x.
L_o: The total light emitting with wavelength λ towards direction ω_o from point x at time t.

L_e: Directly emitted light of wavelength λ from point x, at time t, in direction ω_o.
Ω: The hemisphere around x.
f_r: Bidirectional reflectance distribution function, calculating how light behaves when reflected on an opaque surface.
L_i: Incoming light from ω_i to point x at time t with wavelength λ.

Since the equation has a continuous and not discrete integral component computers can only approximate a solution. It also doesn't take into account all visual phenomena of the real world, which needs to be added to the implementation if wanted. Real-time computer graphics is far behind offline rendering. Most ray tracers used are either ray casters or Whitted ray tracers. Effects possible with later techniques, like blur, can still be approximated via post-processing methods.

1.1.2 Octree

Tree structures has been used in many fields to help navigate through big amounts of data. We see it used in biology to give an easy overview of the evolutionary relationships between different species, or on websites where each page often is found under a parent category.

A tree starts with a root node at the top. This node in turn has a number of child nodes depending on the tree type. These children are called siblings because they share the same parent. Each node is located on a certain depth, which is the number of steps in the tree to the root node. A node with no children is called a leaf.

Trees are very popular in computer science, where big arrays of data is a common part of many programs and algorithms. The tree can either be an abstract data type or an actual data type. If it is concrete data type it might be a class with different variables representing the children and functions for finding information or adding new nodes. An abstract data type will in most cases be a big array formatted in a way to make tree searches possible. The tree used for this paper is abstract, so it will become clear how these work in later sections.

An octree is a tree where each node has a maximum of eight children. If it is sparse that means that a node can have fewer children, unlike a dense octree. This saves a lot of space as otherwise the tree would have to store a lot of empty nodes. The downside of using a sparse tree is that the formatting will be more complicated for searches. In a dense octree you can calculate where a node is without a traversal algorithm because you know for sure that each depth has 8^{depth} nodes. Since a dense tree can be a lot bigger in memory than a sparse tree they both have their uses.

1.1.3 Voxels

A voxel is a point in a regular grid. It can contain information about the volume it represents. Voxels are often used in real-time graphics to condense a lot of information in the scene into a grid with few enough points to achieve effects otherwise not possible. In this project a voxel replaces all the triangles in the cube space around its center point. It will contain information about the average colour, normal and more of all the triangles.

1.1.4 Contours

To improve the visual quality in the scene we have used contours to reduce the blockiness just rendering the cube spaces of the voxels. Contours are polyhedrons which are guaranteed to encompass all the primitives in a voxel. They can have as many sides as the creator wants, with more sides approximating the original models better but at the cost of computational power. A contour is made up by several planes that can be chosen based on different methods.

1.2 Problem formulation

As mentioned ray tracing is very popular in offline rendering to achieve realistic effects. In real-time graphics a lot of creative methods have been invented to estimate some of these effects in a cheaper way but with more unrealistic results. For reflections and refractions a rasterizer can't render what the camera can't see, so normally a static texture has to be used and sampled from to create the illusion of say reflections. As hardware improves and developers tries to improve the graphic quality in games ray tracing have started being more widely used in real-time too. Instead of completely abandoning rasterize rendering most games using ray tracers use a hybrid renderer which only use rays to achieve some effects that are still hard to emulate with the classic rendering.

In this paper we will explore how useful certain optimizations are for a complete ray trace renderer. The first one is voxelization, which will turn our scene of several thousand triangles into a fewer amount of voxels represented by contours. We want to compare the original bunny made up by triangles with the new voxel bunny to see what the visual cost of using voxels is. Voxelization will require several tests to determine which triangle goes into which voxel. We will see how fast this can be done, as that information can be useful for future research. To improve the performance we will use a sparse octree to store our voxels. We will use a traversal algorithm for the octree and rays to try to minimize the time it takes the system to find the first voxel a ray intersects with in the scene. We will measure the build time of the octree based on different depths of the tree, as well as how the triangle testing will be done.

Finally, we will use the octree, ray tracer and voxels to implement realistic reflections and refractions, two visual effects hard to achieve in real-time without ray tracing, and measure the frame rate in different situations and resolutions.

1.3 Related work

We used information from two papers to learn about voxelization. The first one was "Octree-Based Sparse Voxelization Using the GPU Hardware Rasterizer" [4] by Cyril Crassin and Simon Green which touches on a broad number of subjects relevant to voxels and helped us get a basic understanding of a lot of them.

The second paper was "Hybrid Computational Voxelization Using the Graphics Pipeline" [5] by Randall Rauwendaal and Mike Bailey. While it also contains a lot of general information about voxelizations, we mainly used this paper to learn more about the voxel-triangle intersection tests.

We're using a sparse octree structure made by myself, but it uses ideas found in another paper. This paper is "Efficient Sparse Voxel Octrees" [6] by Samuli Laine and Tero Karras.

My sparse octree traversal algorithm is an expansion on the algorithm presented in "An Efficient Parametric Algorithm for Octree Traversal" [7] by J. Revelles et. al. This algorithm has some important key differences from our expansion.

Previous work on reflections has been done by Morgan McGuire and Michael Mara and is presented in the paper "Efficient GPU Screen-Space Ray Tracing" [8].

Our work involves sparse octrees, but for related work made using the BVH structure instead "Real-Time Multiply Recursive Reflections and Refractions using Hybrid Rendering" [9] by P. Ganestam and M. Doggett is recommended.

For further reading on the triangle-box test involving the Separating Axis Theorem see "Fast 3D Triangle-Box Overlap Testing" [10] by T. Akenine-Möller.

1.4 Implementation

We used C++ to implement the majority of the code, including the voxelization and octree building. The octree traversal algorithm and ray tracer were implemented in OpenGL shaders using GLSL version 4.5. The buffer used for the octree and leaves data requires GLSL version 4.3 or later.

1.4.1 Voxelization

We've used the Stanford Bunny [11] for this project, a 3D model consisting of 35 000 vertices used for 70 000 triangles. This model is good to test certain visual effects since it has both thin, thick, smooth and uneven surfaces. The goal of the voxelization is to turn the scene into a grid of voxels, and if a triangle intersects with a voxel its values are added to that voxel. Since we will use a sparse octree there will only be a voxel in a grid slot if there are any triangles intersecting with it.

To find out if a triangle is in a voxel we use several cube-triangle intersection tests. The reason for using many tests is that they can be very cheap but with a higher chance for false positives, or more expensive and accurate. We use the cheap tests to rule out most of the triangles, and then use more expensive tests on the smaller amount of triangles left to get the correct result. We'll show the time saved doing this in the evaluation section.

The first test is a Axis-aligned bounding box (AABB) test. After we've parsed all the vertex and triangle data from the model file we construct an AABB for each triangle and store it in an array. This is done by finding the most negative and positive vertex for the triangle. This makes the test very efficient as we only need to see if the AABB and voxel, which is also axis-aligned, overlap in all three axes. If they do not we can rule out an intersection. Because all voxels share one corner, which is the center coordinate of their parent node, we can further improve the performance time of this test by testing a triangle against several children at the same time. For the worst case scenario when a triangle is in every child simultaneously we would normally need 3 tests per child to determine what voxels the AABB overlaps with. In our implementation we would need 14 comparisons for the same worst case scenario, which requires less than 60 % of the time. There's a

decent risk of a false positive with this test as the voxel can intersect the AABB but not the triangle, so more tests are needed.

input : AABB vector with its coordinates $(min_x, min_y, min_z, max_x, max_y, max_z)$
Parent node's center coordinates $(parent_x, parent_y, parent_z)$
output: Updated triangle lists for every child

for *every triangle in parent node* **do**
  if $min_x < parent_x$ **then**
    if $min_y < parent_y$ **then**
      if $min_z < parent_z$ **then** $Child8 \leftarrow triangle$;
      if $max_z > parent_z$ **then** $Child7 \leftarrow triangle$;
    end
    if $max_y > parent_y$ **then**
      if $min_z < parent_z$ **then** $Child6 \leftarrow triangle$;
      if $max_z > parent_y$ **then** $Child5 \leftarrow triangle$;
    end
  end
  if $max_x > parent_x$ **then**
    if $min_y < parent_y$ **then**
      if $min_z < parent_z$ **then** $Child4 \leftarrow triangle$;
      if $max_z > parent_z$ **then** $Child3 \leftarrow triangle$;
    end
    if $max_y > parent_y$ **then**
      if $min_z < parent_z$ **then** $Child2 \leftarrow triangle$;
      if $max_z > parent_y$ **then** $Child1 \leftarrow triangle$;
    end
  end
end

Algorithm 1: AABB - Voxels intersection test

The second test is a cube-plane intersection test, using the triangle's plane. The plane is also calculated before the testing begins. The three first coefficients are the same as the normal's values, which are calculated for several uses later. By taking the dot product of the normal and one of the triangle's vertices we get the fourth and last coefficient for the plane.

To test the cube and the plane we first find the most negative and positive corners of the cube. We then see if they're on opposite sides of the plane by calculating the distance between the plane and the points. If a point is below the plane the distance will be negative, otherwise positive. If the two points are on opposite sides that means an intersection. This test rules out most of the negatives, but there's a small chance that it gets a false positive.

The third and final test uses the Separating Axis theorem. We project the vertex and triangle on three 2D planes by either removing the x, y or z coordinate depending on what plane, XY, YZ or ZX, we want to project to. We then create 3 perpendicular axes from the 3 edges of the triangle, and project the voxel and triangle corners on each axis. If they do not overlap in at least one axis there exists a plane that can separate the voxel and triangle, meaning they do not intersect. If they overlap for every axis, for every 2D projection, we

input : Triangle's plane coefficients (a, b, c, d)
Center coordinates of child node $(center_x, center_y, center_z)$
Length from center coordinate to side, $length$
output: Intersection boolean

$p \leftarrow (center_x - length, center_y - length, center_z - length)$
$n \leftarrow (center_x + length, center_y + length, center_z + length)$

for *every dimension i of normal* **do**
  if $normal_i > 0$ **then** swap p_i and n_i's values;
end

if *distance between plane and point p is negative* **then** $Intersection \leftarrow false$;
else if *distance between plane and point n is positive* **then**
$Intersection \leftarrow false$;
else $Intersection \leftarrow true$;

Algorithm 2: Plane - Voxel intersection test

know for certain that the voxel and triangle intersect. Normally we would need to also look for an overlap using the voxel's edges as axes too, but we learned that they overlapped in the AABB test already.

1.4.2 Octree building

We start the octree building by first creating the root node. To calculate it's size we find the smallest and biggest coordinate value in each axis by comparing every triangle vertex and see which difference is largest. This will be the length of each side of the root node. We then split the root into 8 equally big children and pass along a list containing all the triangles in the scene. For each triangle in the list we test which of the eight child nodes it intersects. For each child it intersects with we add it to that child's triangle list. After testing all triangles we split up those children with triangles in them in 8 new children, which will have their respective list of triangles themselves.

This recursive method is done until we reach a specified depth of the tree. We then create a leaf, which is an array containing the average normal, colour, reflectivity and refractivity of all the triangles intersecting the node. This array is then inserted to a bigger array containing all the leaves. We return an index of the leafs position in this big array to its parent. After the parent has received a return value from all of its children it inserts an array of 9 values into an octree array. The first value is a information byte with each bit telling us if the i'th child exist. The 8th following values are pointers to the children. The node then returns a pointer to itself to the parent node.

The order of the children will matter depending on what you intend to do with the tree. For our tree to work with our traversal algorithm the first child must be the most positive one, and the eight the most negative. The exact order can be seen in figure 1.1 on page 14

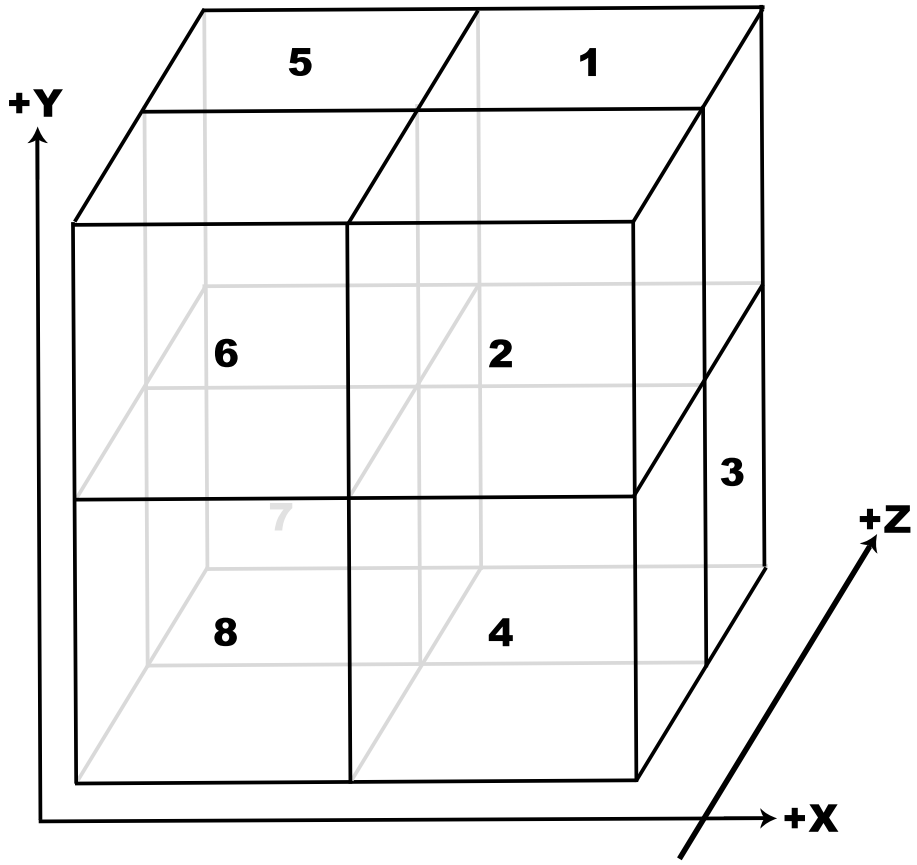


Figure 1.1: Child order

input : Center coordinates of child node $(center_x, center_y, center_z)$
Length from center coordinate to side, $length$
output: Intersection boolean

for $Planep \leftarrow XY, YZ, ZX$ **do**
  $v_p, t_p \leftarrow$ Project voxel corners and triangle vertices on plane.
  for *every edge e of triangle* **do**
    $axis \leftarrow \frac{e}{\|e\|}$
    for *every corner i in* v_p **do** $v_{ai} \leftarrow v_{pi} \cdot axis$;
    for *every vertex i in* t_p **do** $t_{ai} \leftarrow t_{pi} \cdot axis$;
    $min_t \leftarrow min(t_a)$
    $max_t \leftarrow max(t_a)$
    $min_v \leftarrow min(v_a)$
    $max_v \leftarrow max(v_a)$
    if $max_t < min_v$ *or* $max_v < min_t$ **then** $intersection \leftarrow false$;
  end
end
$intersection \leftarrow true$

Algorithm 3: Separating Axis Theorem intersection test

1.4.3 Ray tracing

Normally when a scene is rendered by the GPU the vertex shader first gets fed the data about the vertices of the scene, primitives are created and if a fragment covers a primitive it gets coloured by the fragment shader. This is known as rasterizer-based rendering. With a ray tracer we instead find if we hit something by a series of intersection tests between the ray and objects in the scene, so we do not need to create vertices for the primitives. Instead we create a canvas to cover the screen. This is a quad made in the vertex shader to exactly fit the screen. We also create a ray from the camera towards each corner of this quad. A ray is represented as:

$$r(t) = (x_o, y_o, z_o) + t * (x_d, y_d, z_d)$$

which can be written as:

$$r(t) = o + t * d$$

The first vector is the origin point of the ray, in our case the camera coordinates. The second vector is the direction, in our case the coordinates of one of the four corners. All positive values of t give us a point along the ray line, and a negative value give us a point behind the camera. In the fragment shader we use trilinear interpolation between the four rays using the fragment's position on the screen to get the ray direction for that specific fragment. We can then start our octree traversal. The first step is to test if the ray hits the

scene, which is done by calculating the distance to each plane of the root node's faces. This is done by solving the following equation for t (here for the YZ planes):

$$o.x + t * d.x = root.x \pm \frac{rootSize}{2}$$

When subtracting the rootSize you get the distance to the most negative plane, otherwise the most positive. If the distance to the negative plane is less than the positive for every axis, and no distance to a positive face is below zero we know that the ray hits the root node and scene. These 6 distances are kept and modified all the way through the traversal until we hit a leaf. Every time we reach a node in the octree we check for 3 cases:

1. Is the node behind us? This is just as with the root node done by testing if any of the most positive face distances are negative. The algorithm then goes back to the previous step and tests the next node in line.

2. If we've hit a voxel leaf. In our octree array indices to leafs are marked by being negative, while indices to nodes are positive so that we easily can test for this case. In this case we test if we hit a contour, which we will talk about in the next section. If there's an intersection we return the index to the voxel as well as the distance to the intersection point.

3. The node has children.

In the last case we have to find out in what order the algorithm should check the node's children. This is done by first calculating the distance to the middle point between the negative and positive face of the node's parent. We then have either the new negative or positive face, depending on where in the parent the child is.

By comparing the new negative and positive face distances we know in what order the children will be intersected by the ray. As an example, if the last negative face to be hit is the XY-plane we know that the ray will first hit the bottom of the node.

Once we know what child to go to we see if the child exist by checking the information byte previously mentioned in 1.4.2. If it does we continue by checking it. If that path does not lead anywhere we check the next sibling node in line. Once all possibly intersected siblings have been checked with no voxel intersection found we go back up one level in the tree, checking which of the parent's siblings are next in line. This way, we check all nodes intersected by the ray in the octree. If the ray does not hit anything we use a samplerCube to create a skybox. Otherwise we check if the voxel we've hit is recursive or reflective and spawn a new ray by using the voxel's normal and traverse through the octree with it. If the voxel is not reflective or refractive we apply a simple Phong shading algorithm to give a sense of depth and use the voxel's colour to colour the fragment.

1.4.4 Additional implementation additions

Negative rays

The octree traversal algorithm only works for positive direction rays so that we can make assumptions about the most negative and positive faces. To make this algorithm viable for

every ray we must mirror the scene if a ray is negative in any axis. The ray will be remade to positive with the following calculations (if x is the negative direction dimension):

$o.x = 2 * rootCenter.x - o.x$
$d.x = -d.x$

We do this for every negative direction dimension, and also create a byte for mirroring the octree. This byte is initialized as 0. If the x-direction is negative we bitwise *or* it with 4. If y is negative we *or* it with 2, and for z with 1. This byte is later used to get the mirrored child node in the octree by using *xor* with the child's index.

Triangle testing

The octree building time is significantly better if we only use the final test on the final depth level when creating the leaves. While we'll carry some false positives further down the tree than if we would use the SAT test on every level, not having to use SAT on the true positives more than once per possible voxel saves a lot of time. There could be scenes where this is not true, but we can not see any practical scenario with those types of scenes. If we would be interested in what triangles are in non-leafs we would have to use SAT on every level.

Another very big optimization in the testing is to before each SAT test check if any of the triangle's vertices are inside the voxel cube. If this is the case we're guaranteed an intersection, so we can skip the SAT test.

Contours

Without contours the octree would need to be very deep to not look blocky, as we would simply render a voxel as the cube of space it takes up. While creating the leaf in the octree we also create two planes, a positive and a negative one, that encompass all the triangles in the voxel the leaf represents. The planes are represented by their d-coefficient, as the rest of the coefficients can be found in the normal which is already calculated and stored. For each vertex of each triangle in the voxel we calculate the distance to the planes. If the distance to the positive plane is positive we move that plane so that it contains the vertex. If the distance to the negative plane is negative we move that plane to the vertex.

Once a ray has hit a non-empty leaf during ray tracing we test if we hit the polyhedron, which would mean the ray should stop. The first test calculates the intersection point between the ray and the voxel, and then calculates the distance between that point and the two planes. If the point is between the two planes we've hit the contour. Otherwise we calculate the distance to the intersection point between the ray and the first plane that will be hit. We know which of the two planes it will be based on the previous distance calculations, which both will be positive if the ray will hit the positive plane first, otherwise the negative one. A final comparison to make sure the distance to the intersection point is between the distance to the entry and exit point of the voxel acts as the final test for an intersection.

Buffers

To pass all the data about the leaves and octree to the shader we're using Shader Storage Buffer Objects (SSBO). This new buffer has been a core feature since GLSL 4.3. This means that current MacOSX systems (up to Yosemite) can't run it without using extensions since they only support GLSL 4.1. The extensions needed are:

- *ARB_shader_storagebuffer_object*
- *ARB_program_interface_query*

Other options for passing big amounts of data are textures and Uniform Buffer Objects (UBO). These have a lot lower size limits than an SSBO though. For comparison, an UBO has a size limit of at least 16KB, while an SSBO at least has 16MB but more typically on the order of the GPU memory [12]. For this project the difference meant that we could only create an octree of depth 4 with an UBO. We could work around the limit by splitting up our arrays to reach depth 5, but 6 would have been impossible. With SSBO there are no issues at all.

1.5 Evaluation

To test our implementation we rendered a scene with the Stanford Bunny submerged into a pool from 3 angles. We added code to measure the frame rate by for each frame adding to a frame counter. Once the counter hits 100 we check how much time has passed and can then calculate the frame rate. We think this keeps the timer's performance hit close to nothing.

We've tested both the octree build time and frame rate on level 1 to 8 in various ways. For our octree we tested the best way to perform the three intersection tests by using the tests either on every level or on the leaf level only. In the following table we've presented the build time in seconds for each level:

	1	2	3	4	5	6	7	8
AABB + Plane	3.071	1.216	1.185	1.249	1.647	3.335	17.35	54.17
AABB + Plane + SAT	23.81	36.13	45.40	54.40	66.21	95.46	197.3	472.2
SAT (leaf level)	36.25	14.83	9.539	9.960	13.85	78.06	191.2	427.7
Plane + SAT (leaf level)	34.60	14.13	8.870	9.593	14.52	80.69	204.7	493.1

Since SAT on leaf level only was the fastest we then used the extra test of checking if any vertex was inside the cube meaning we could skip the SAT test if it was. The build times we then got for every level is presented as:

	1	2	3	4	5	6	7	8
SAT (leaf level)	1.813	1.038	0.859	1.394	2.216	11.71	61.2	256.6

Testing the frame rate is both difficult and not as telling as in many other projects. When measuring rasterizer-based programs the framerate will mostly depend on what's in front of the camera only. With a ray tracer and octree the whole scene, even things not in front of the camera, will have an effect. Only looking at a flat surface in one scene and the same surface placed in another scene can give us different frame rates, even though the image

rendered will look identical. With that said we've compared 3 different camera positions in the scene. The first one is when the whole screen is filled with water (figure 1.2, p. 20). This is in most cases the worst possible situation seen to performance, as every ray will hit the water, requiring two more traversals for the reflection and refraction rays. The seconds camera position makes every ray hit the bunny from the side with as many voxel normals as possible pointing towards the camera, and will therefore require one traversal with no bad cases per pixel (figure 1.3, p. 20). In the last test we've placed the camera along the bunny's inside, which is hollow (figure 1.4, p. 21). With contours enabled a very taxing traversal situation is if a ray hits several non-empty leaves but doesn't hit the planes within them and have to continue in the tree. While writing and testing our code we discovered this was the situation requiring the most node visits, so it could be a contender with the water view for worst case scenario. We tested each situation on every level, with two different resolutions. In the following table we've presented the frame rate we got with an AMD Radeon HD 6900 series card:

	1	2	3	4	5	6	7	8
Water (640x480)	-	-	25	17	13	9.5	7.4	6
Water (1920x1200)	-	-	4	2.7	2.0	1.5	1.21	1.18
Bunny (640x480)	105	105	65	60	50	45	40	35
Bunny (1920x1200)	18	17	14	11	9.8	7.9	7.2	6.0
Parallel (640x480)	-	-	52	47	32	23	15	10
Parallel (1920x1200)	-	-	11	8	5.5	3.8	2.61	1.94

On level 1 and 2 it was not possible to get a full view of water and the bunny was no longer hollow, so tests could not be done.

As explained in the buffer implementation section we first used UBOs instead of SSBOs. When reading data from these arrays UBOs should be slightly faster or at least not slower [12], but no performance change was detected when comparing the two buffers with an octree with depth 5. This could be because the amount of accesses are not enough on that depth to make a difference. Since the UBO couldn't handle any deeper trees we do not think it's worth investigating though.

In the contour image we can one of the limitations of our traversal algorithm. Because some rays have to pass a very large amount of nodes they hit an iteration cap specified in the code. We chose to have this limit so that a few pixels alone wouldn't lower the frame rate. We left it in to better showcase the problem, but one could instead render the last voxel the ray hits before the cap as a cube instead, or increase the cap if the lower frame rate is acceptable.

1.6 Visual comparisons

Using a rasterizer to render the scene gives us an rendition of the bunny (fig 1.5), letting us compare the voxelized bunny to a perfectly correct one.

In figures 1.6 to 1.10 we can see the difference between rendering contours and voxel cubes on different levels.

From comparing these images we can see that the difference contours make is more

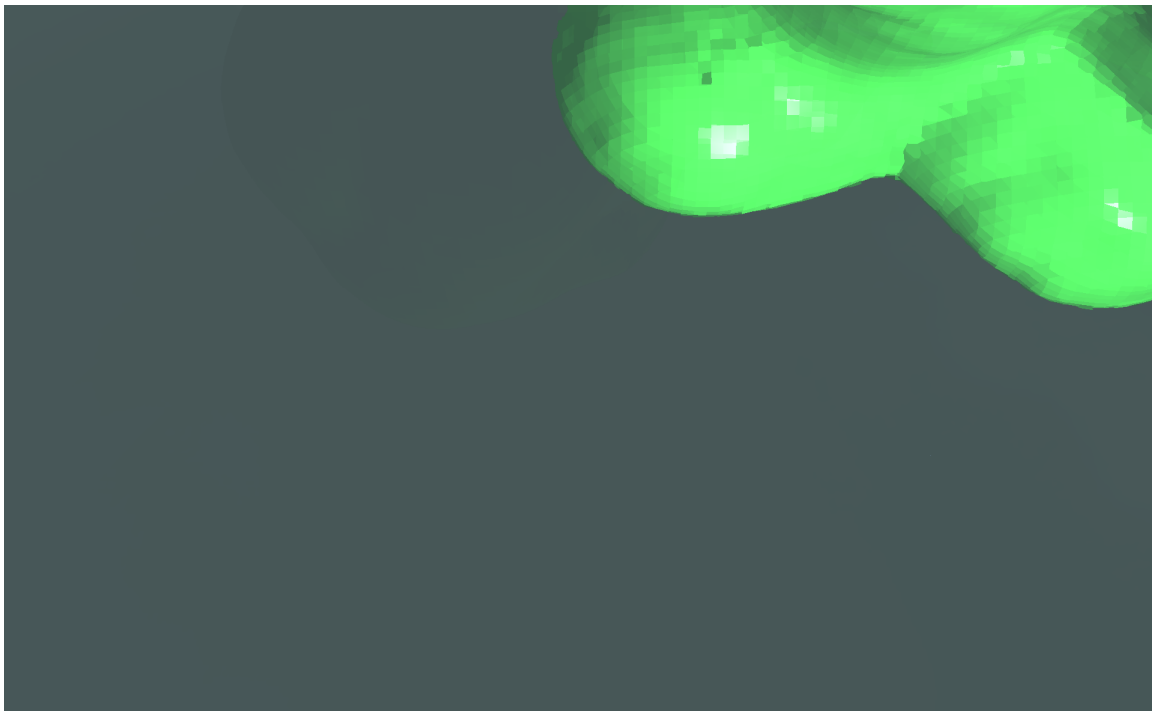


Figure 1.2: Water image

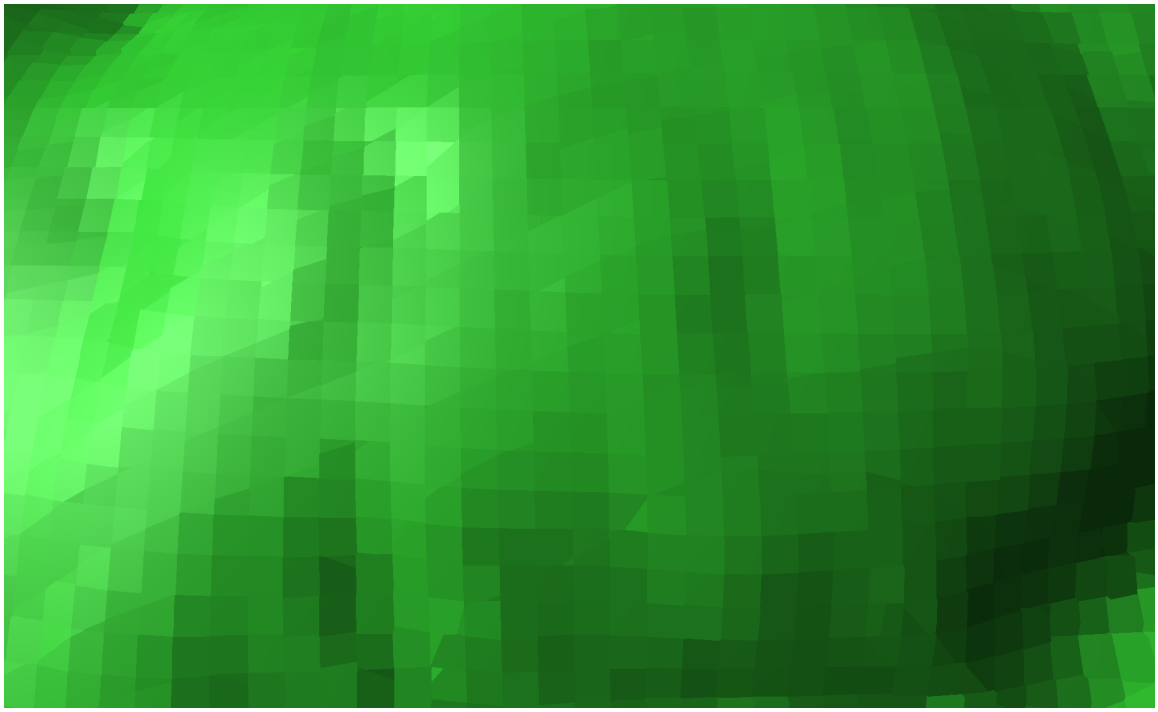


Figure 1.3: Bunny image

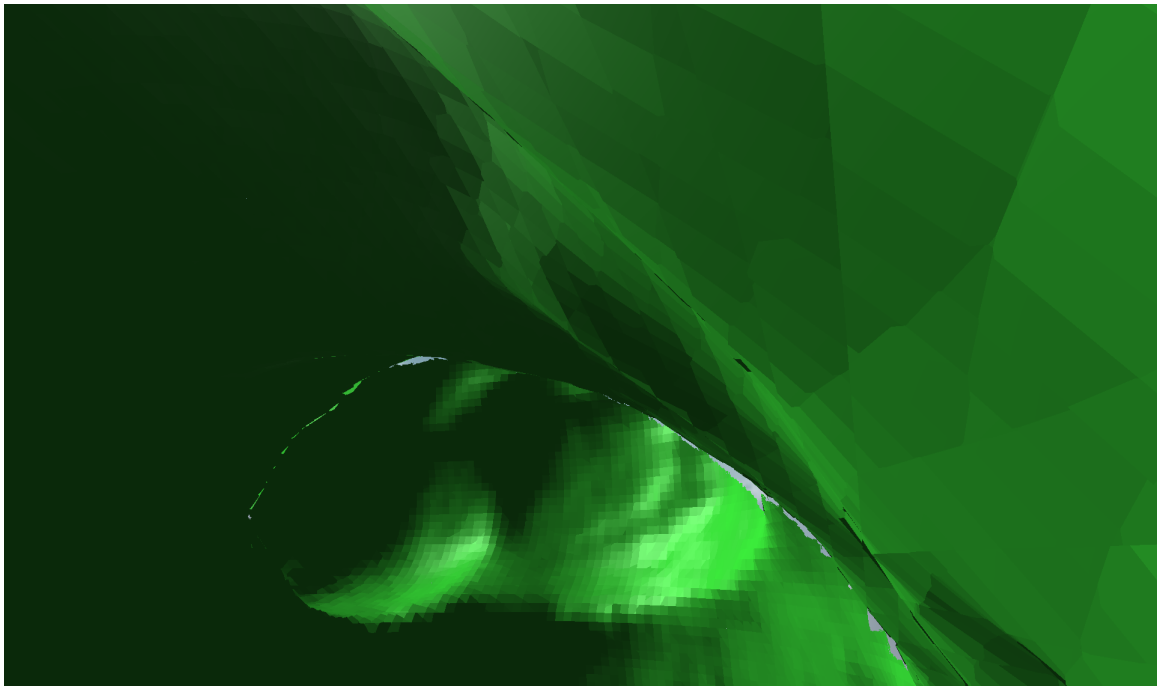


Figure 1.4: Contour image

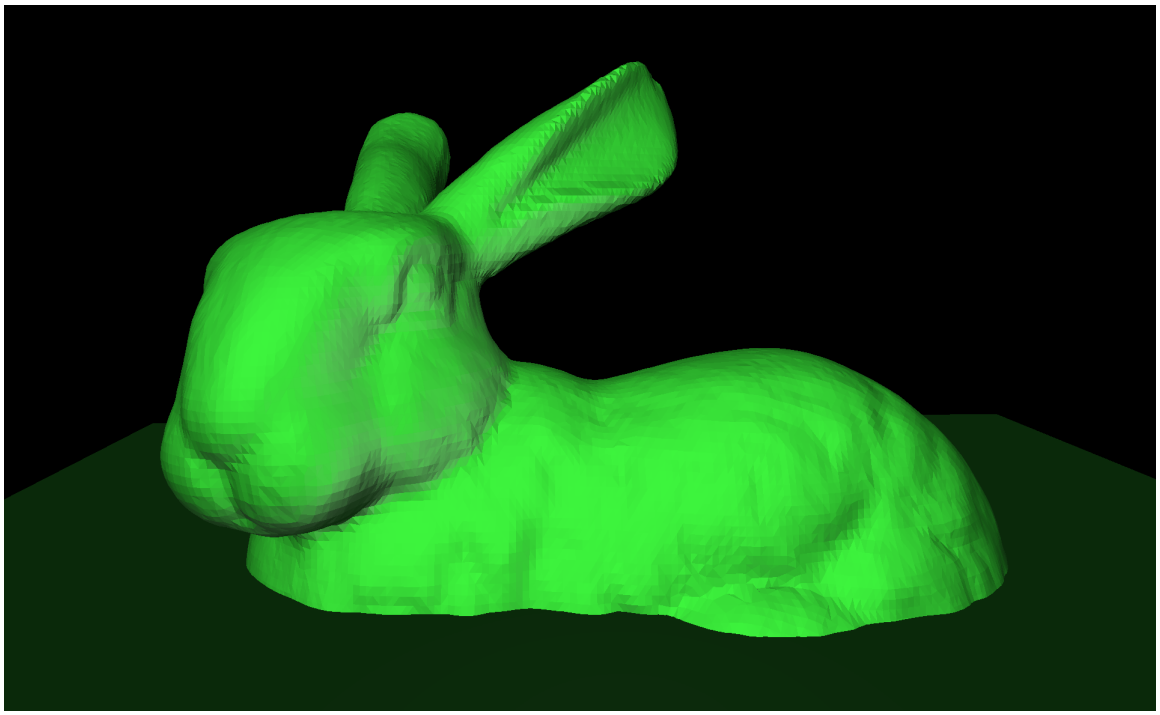


Figure 1.5: Rasterized bunny

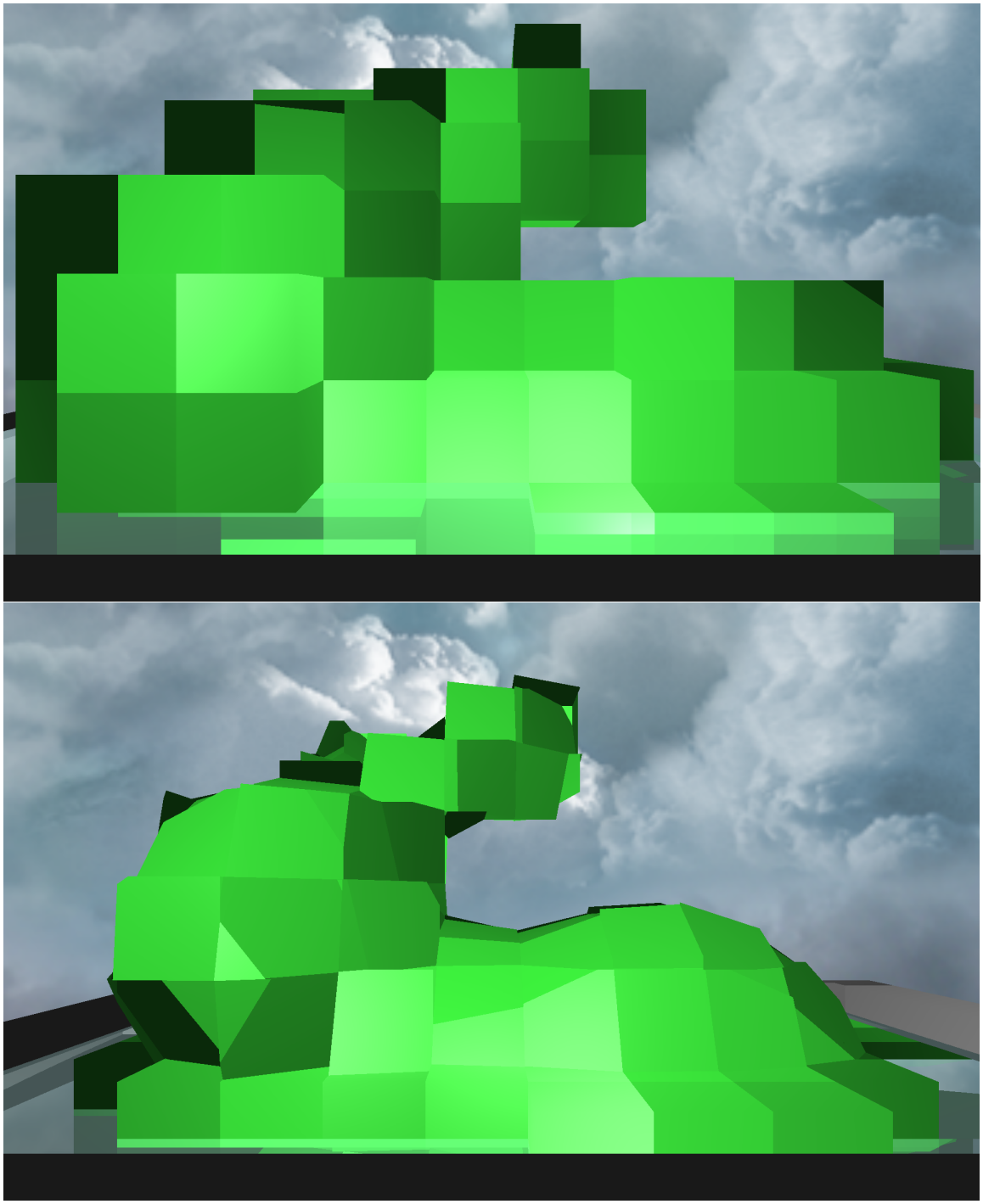


Figure 1.6: Depth 4 octree

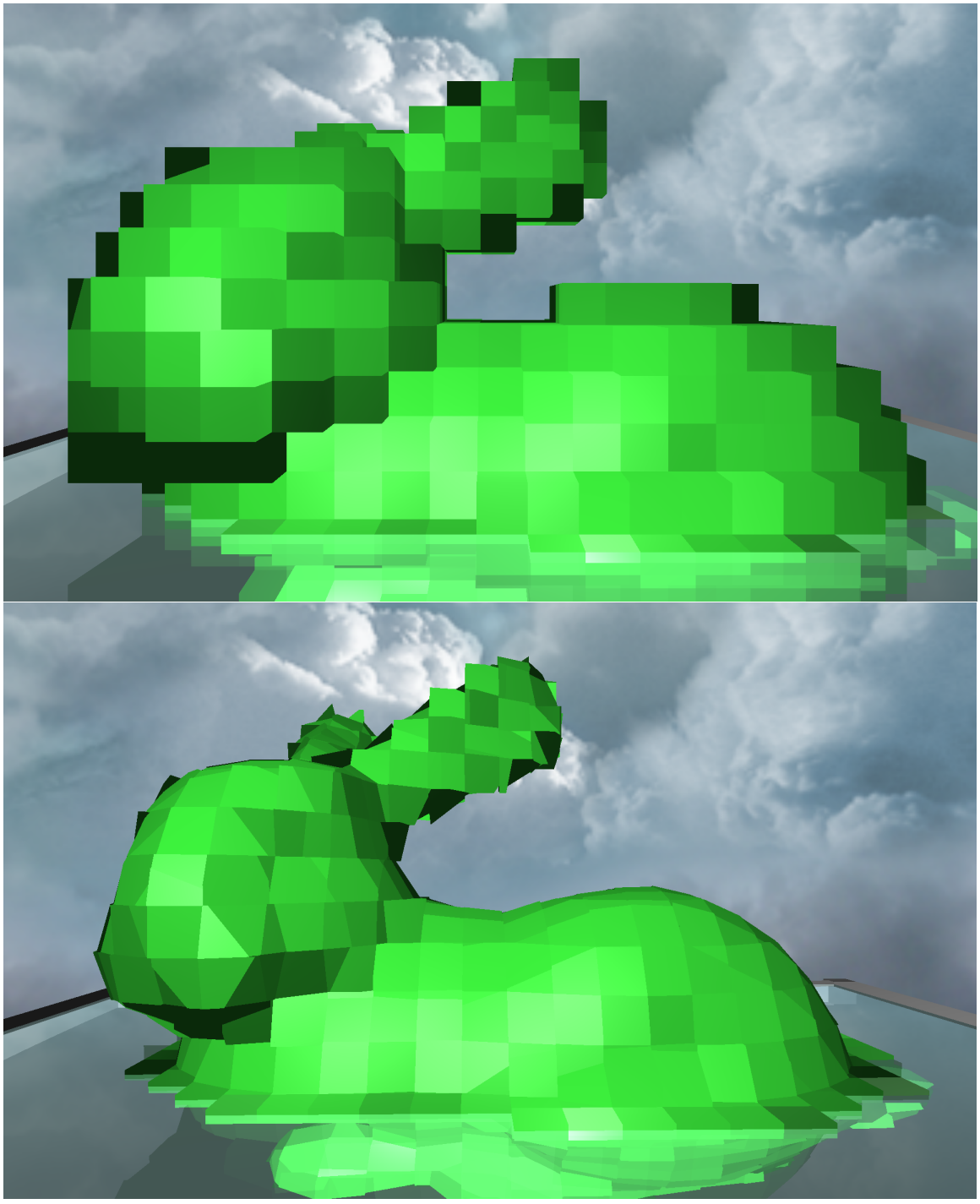


Figure 1.7: Depth 5 octree

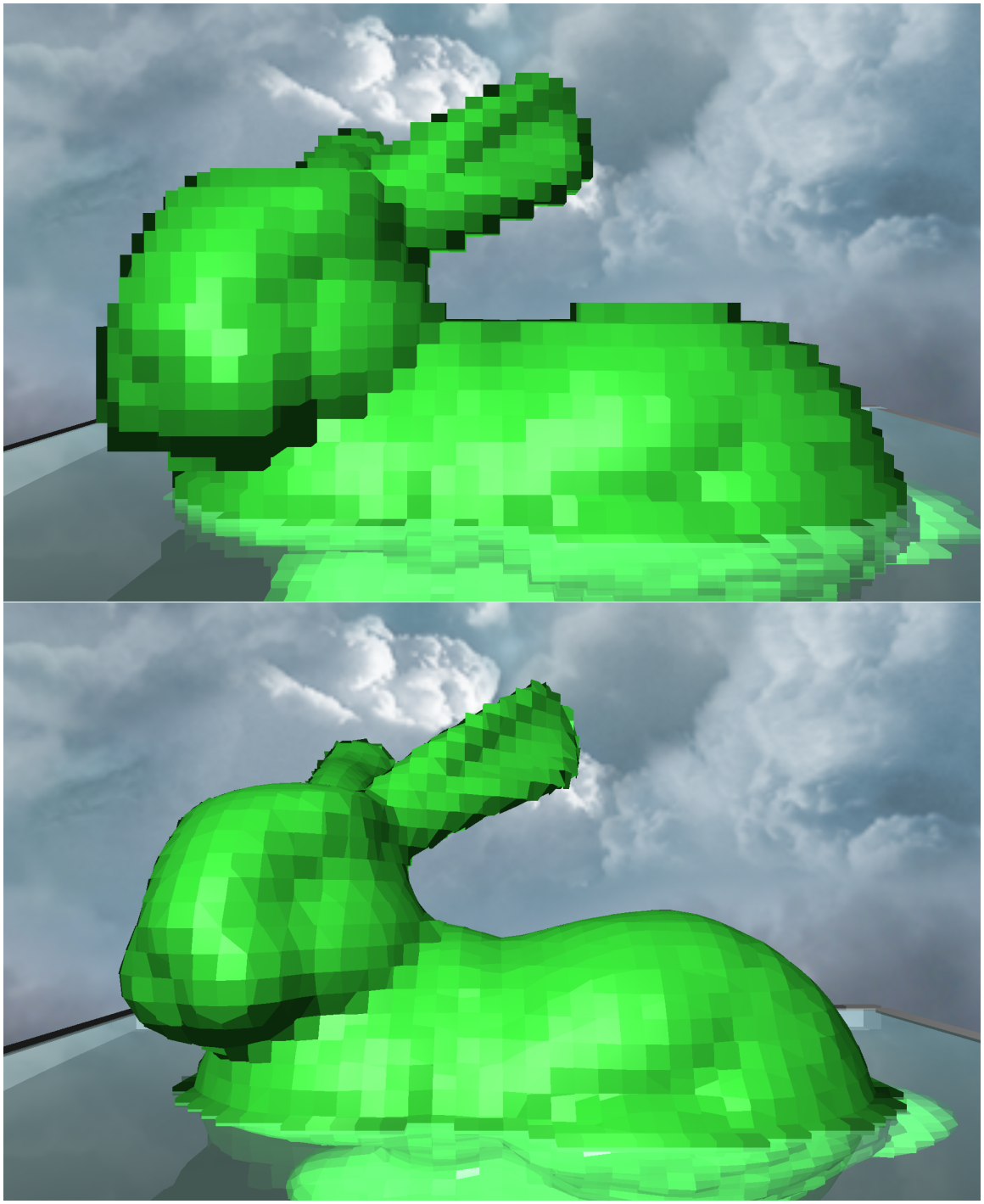


Figure 1.8: Depth 6 octree

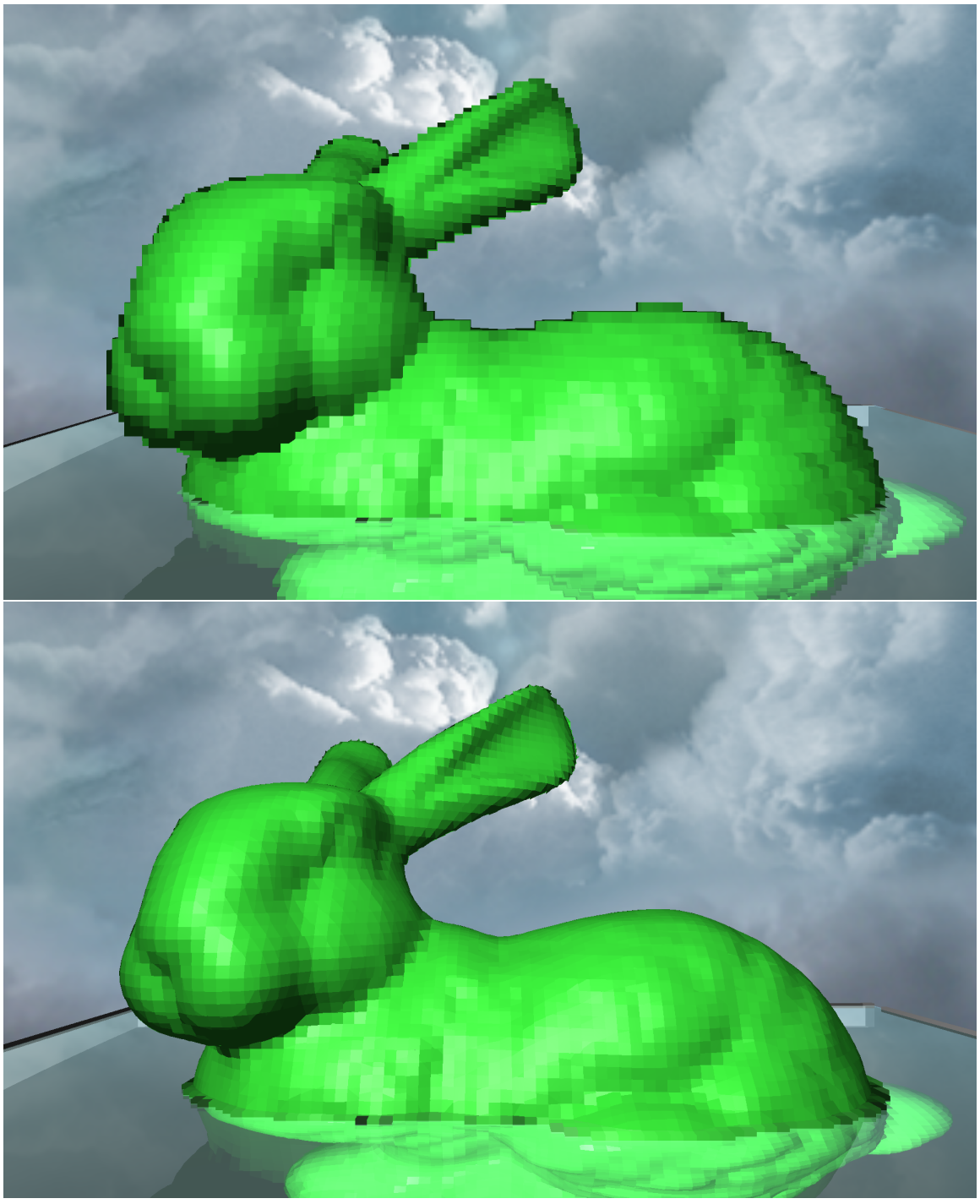


Figure 1.9: Depth 7 octree

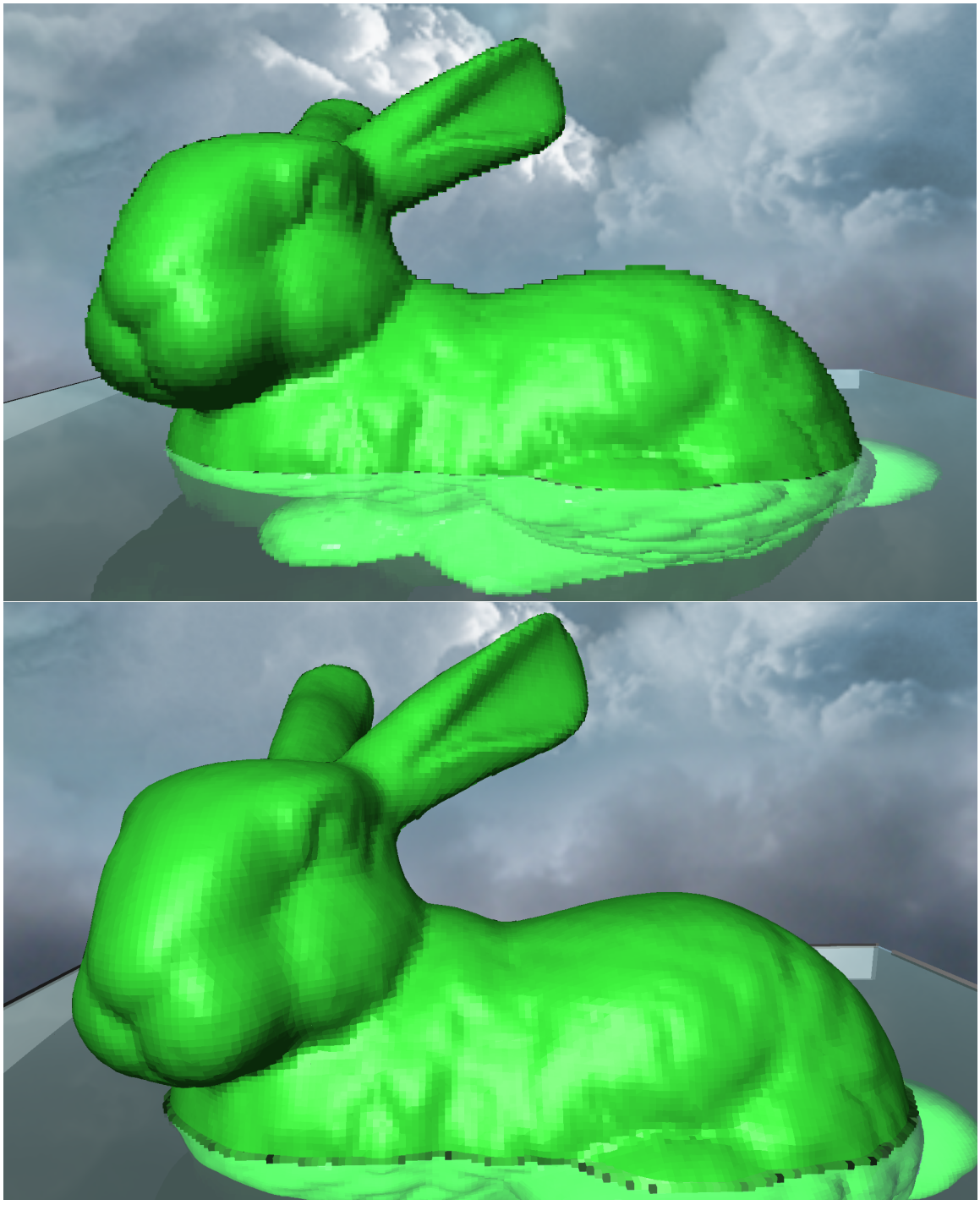


Figure 1.10: Depth 8 octree

noticeable on lower levels, and that on depth 8 the difference is mostly in the actual contours of the bunny. If we would use even deeper trees a possible optimization could be to only render contours if the angle between the ray direction and the plane is below a certain threshold.

We can also see how our reflections and refractions behave in figure 1.11. We've used Fresnel's equations to mix the contribution the reflection and refraction makes for the colour of the water. This makes the water look more realistic, as it reflects more if we look along the surface instead of directly down through it. We can also see the refraction in full effect as the pool looks to be shallower further away, even though the depth is really unchanged.

1.7 Discussion

1.7.1 Octree build time

As we can see in the evaluation section the optimal way to perform the intersection tests is to save the SAT test for the final level, while the plane-voxel test should be done on each level. This is because the AABB test is quite bad in accuracy and lets a lot of false positives through. The plane test on the other hand is still very cheap, as can be seen in the case where we completely skipped the SAT test all together, but still has a high accuracy. The false positives it lets through will be mostly noticeable in sharp surfaces. In our scene the major visual difference between skipping the SAT test or not can be seen in the ears, otherwise the smooth bunny still looks very good.

Since the octree is built before the renderer starts build time is not as important as the ray tracer performance, so there's no question in our minds that all three tests should be used. If we did not use contours but instead just rendered cubes, the difference between using the SAT test and not can hardly be seen in the rendered images. Since the octree is static it also only need to be computed once, and can then be stored as a string which can be parsed the next time you need the tree.

At the end of the project while trying out other models we noticed how much the build time would increase if the model had a very high resolution. To try to counter this issue we implemented the additional test on the leaf level in which we see if any vertex is in the voxel. This will make the voxelization skip the SAT test. For our Bunny model the reduction in build time was massive, staying around 90% for the 6 first depths and then quickly dropping to 40% for level 8. The true positive cases this extra test will miss are the ones where all vertices are outside of the voxel, but it will still be somewhere between them. Since this case is more rare in more detailed models, we estimate that this test will give better results for those.

There's not many things we think we can improve with the octree builder. Since the plane test is very close to the SAT test in terms of accuracy we don't think any tests between them would improve the build time. If we had to continue working on the octree we would spend more time trying to find a faster test than SAT that still had perfect accuracy, since that's the big time sink.

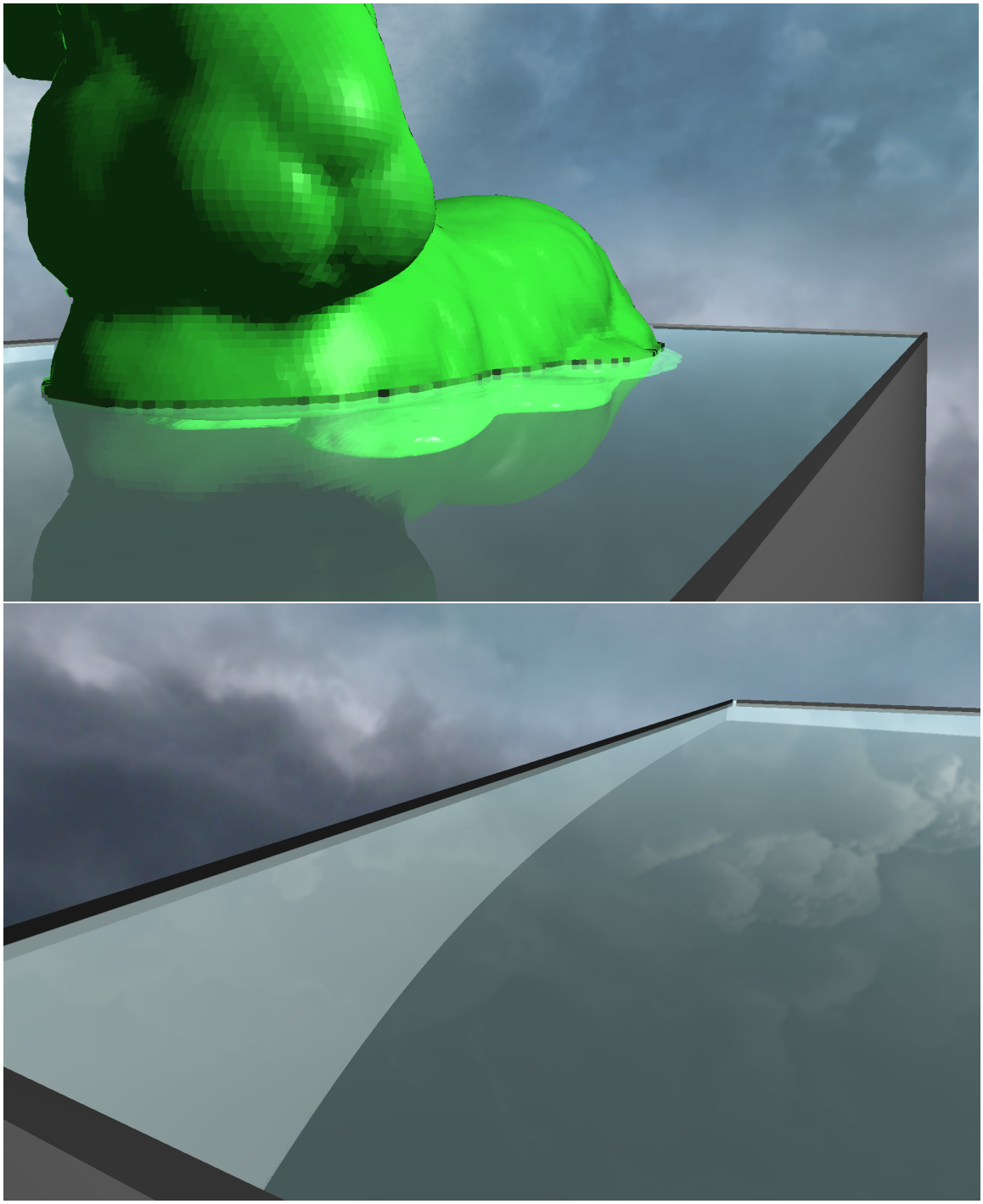


Figure 1.11: Reflections and refractions

1.7.2 Ray tracer

In this implementation we've used a whitted ray tracer without shadow rays. Adding shadow rays would half the frame rate in most situation as each ray would require two traversals instead of one. Adding distributed rays to achieve soft shadows, blur and other effects would be to costly and is better left to post-processing algorithms. As can be seen in the evaluation section the frame rate for a solely ray tracing renderer is highly dependant on the effects you choose to use. Contours were very cheap except in the parallel case, while reflections and refractions can be very heavy. Beyond requiring at least an extra traversal those two features also need to calculate all other effects one more time, like the contours used in this project.

The frame rate was not enough for real time in some of the cases, but there are a lot of improvements that can be done to vastly improve the performance. Many games utilizing ray tracing today only use it for some parts of the screen, and let a rasterizer render the rest which saves a lot of time. There are also many other optimizations we would have wanted to try out. The rays are all independent of each other, but there are techniques which take advantage of the fact that nearby rays will almost take the same path, which could have the potential to vastly increase the performance [13]. We would also have liked to find a better way to implement reflections and refractions, since as mentioned they are very computationally heavy and mostly based on physics and with little optimizations done to take benefit of the octree.

1.8 Conclusions

From the work and research we've done we think ray tracing in real time is definitely possible, it just comes with a lot of ifs. The graphic card we used for testing the code is close to 5 years old, so with a modern card from the last year the performance would be a lot better. While hard to tell without specific tests we'd estimate that a Radeon R9 280X for example could double the frame rate, based on 3DMark benchmark tests. Overall we're pleased with the results from this project. The octree performed very well and is shown to work well with ray tracing, even on an GPU. We believe it can be one of the best optimizations possible for performance. Combined with the voxelizations it's very flexible, as only memory limits how much information the voxels can hold. The contour for example is something we added very late in the project and it only required additions to the code, meaning we didn't have to rewrite anything. The voxelization is shown to produce visually pleasing results, allowing the use of the octree without diminish the quality of the image.

Bibliography

[1] Arthur Appel,
Some techniques for shading machine renderings of solids, IBM Research Center. In: AFIPS '68 (Spring), Proceedings of the April 30–May 2, 1968, spring joint computer conference, p. 37-45. ACM, New York (1968)

[2] Borko Furht,
Handbook of Multimedia for Digital Entertainment and Arts, Springer Science & Business Media, 2010.

[3] Robert L. Cook, Thomas Porter, Loren Carpenter
Distributed Ray Tracing, Computer Division, Lucasfilm Ltd. In: SIGGRAPH '84, Proceedings of the 11th annual conference on Computer graphics and interactive techniques, p. 137-145. ACM, New York (1984).

[4] Cyril Crassin, Simon Green
Octree-Based Sparse Voxelization Using the GPU Hardware Rasterizer, Nvidia Corporation. In: OpenGL Insights. CRC Press (2012).

[5] Randall Rauwendaal, Mike Bailey
Hybrid Computational Voxelization Using the Graphics Pipeline, Oregon State University. In: Journal of Computer Graphics Techniques (JCGT), vol. 2, no. 1 (2013).

[6] Samuli Laine, Tero Karras
Efficient Sparse Voxel Octrees, NVIDIA Research. In: I3D '10 Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games, p. 55-63. ACM, New York (2010).

[7] J. Revelles, C. Ureña, M. Lastra
An Efficient Parametric Algorithm for Octree Traversal, University of Granada, Spain. In: WSCG 2000 Conference Proceedings (2000).

[8] Morgan McGuire, Michael Mara
Efficient GPU Screen-Space Ray Tracing, Williams College. In: Journal of Computer Graphics Techniques (JCGT), vol. 3, no. 4 (2014).

[9] Per Ganestam, Michael Doggett
Real-time multiply recursive reflections and refractions using hybrid rendering, Lund University. In: The Visual Computer (2014).

[10] Tomas Akenine-Möller
Fast 3D Triangle-Box Overlap Testing, Chalmers University of Technology. In: Journal of Graphics Tools, vol. 6, no. 1, p. 29-33. A. K. Peters, Ltd. Natick, MA, USA (2001).

[11] Stanford University Computer Graphics Laboratory
Stanford Bunny, Available at: `https://graphics.stanford.edu/data/3Dscanrep/`. [Accessed 13 April 15].

[12] Khronos Group
Shader Storage Buffer Object, Available at: `https://www.opengl.org/wiki/Shader_Storage_Buffer_Object`. [Accessed 13 April 15].

[13] Ryan Overbeck, Ravi Ramamoorthi, William R. Mark
Large Ray Packets for Real-time Whitted Ray Tracing, Columbia University, Intel Corporation, University of Texas at Austin, 2008. In: Symposium on Interactive Ray Tracing 2008, p. 41-48. IEEE (2008).

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTATIONSDAG 2015-04-29

EXAMENSARBETE Real-time screen space reflections and refractions using sparse voxel octrees
STUDENT Filip Nilsson
HANDLEDARE Michael Doggett (LTH)
EXAMINATOR Tomas Akenine-Möller (LTH)

Reflections and refractions using a real-time ray tracer

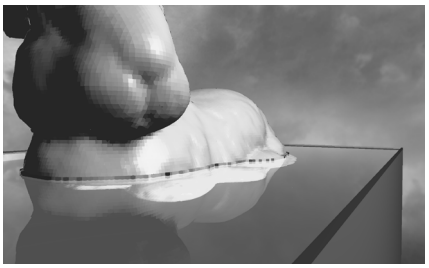
POPULÄRVETENSKAPLIG SAMMANFATTNING **Filip Nilsson**

To reflect and refract objects in a correct way ray tracing has to be used. This is very computationally demanding in real-time applications. However, by combining three different methods the performance can vastly improve.

What we see as reflections and refractions in games and other real-time applications is in most cases just an image being sampled to create an illusion of the effect. When rendering a frame on the screen only objects in front of the camera is computed, so it's very hard to reflect the rest of the scene. To do this we have to use ray tracing instead. This technique works by tracing rays through the camera into different parts of the scene. These rays can be compared to photon rays, but instead of coming from the scene we trace them backwards from the camera. We can then emulate real-world effects realistically by for example letting them bounce to get reflections. Ray tracing is very computationally demanding, as a single movie frame can take days to render. In real-time applications we normally want at least 24 frames per second to get smooth motions, so a lot of optimizations have to be made.

The octree structure

To see what a ray hits we perform intersection tests with objects in the scene. In the most basic form of ray tracing we test a ray against every triangle in the scene and see which one we hit first. This is unfeasible as a scene normally have thousands to millions triangles. We instead use an octree, which is a tree structure consisting of nodes containing voxels. A voxel is a cube that takes up space in the scene. The first node in the tree is the root node, which contains the whole scene. This node has 8 children, each taking up 1/8 of its space. The children in turn have their own 8 children, and so it continues until the tree has reached a certain depth. The tree is also sparse, meaning that if a voxel doesn't contain any triangles it will not be created, saving space.



The water both reflects and refracts the scene.

The voxelization process

To find out which triangles belong to which voxels three tests are performed. They vary in performance and accuracy, where they can be faster but might include triangles that are not in the voxel, or more expensive and accurate. The cheap tests are useful for ruling out the vast majority of triangles, leaving a smaller amount for the expensive ones.

On the lowest depth the triangles in a voxel are condensed into a polyhedron based on the average shapes, colours and other attributes of the triangles. This is all done once during the preparation part of the program.

The traversal algorithm

Once we start rendering frames we use an algorithm made for this specific type of octree to quickly find what voxel will be hit. This is done by first testing which of the 8 children of the root node that are intersected by the ray. In order of intersection their children are then tested in a similar way until we reach a node on the max depth of the tree that is intersected, and we render the polyhedron of that voxel.

This reduces the amount of intersection tests to a fraction of the total number of triangles, increasing ray tracing performance enough for real-time. Using reflections and refractions decrease the performance, but are visually correct. This work shows the use of real-time ray tracing to create more realistic reflections and refractions than typical rasterization algorithms can achieve.