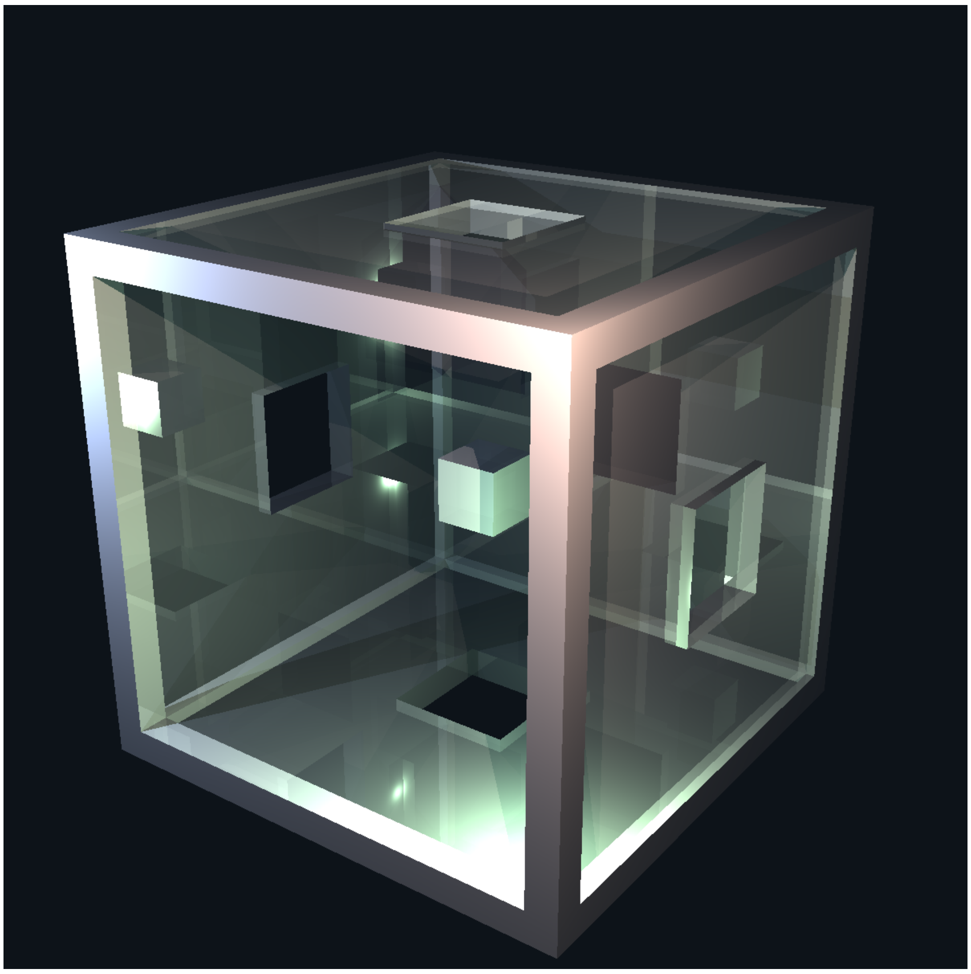
TSBK03 Project

Ray Tracing in Voxel Space

Axel Nordanskog

December 2019



1 Introduction

This project was done as a part of the course TSBK03 Advanced Game Programming at Linköping University in fall 2019. The aim of the project was to implement ray tracing as a way to visualize voxels. This was made using the ordinary OpenGL pipeline, primarily using a fragment shader, as opposed to other common ray tracing implementation methods and tools such as compute shaders [1], Nvidia CUDA [2] and OpenCL [3].

1.1 Requirements

The following mandatory and optional requirements were specified for the project:

Mandatory:

- M1: Ray tracing against voxel geometry.
- M2: Point lights – hard shadows.
- M3: Phong shading – ambient, specular and diffuse light.
- M4: Reflection rays – mirroring.
- M5: Refraction rays – transparency.
- M6: Voxelization of arbitrary polygon geometry.

Optional:

- O1: Light/shadows through semitransparent materials.
- O2: Octree geometry representation rather than an ordinary 3D grid.
- O3: Light sources with area – soft shadows.
- O5: Voxel cone tracing.
- O6: One of the following:
  - a: Marching cubes visualization of voxels.
  - b: Ray tracing against polygon geometry which corresponding voxels has been hit.

All mandatory requirements are met except for M6 (voxelization) due to time constraints. Work was done to successfully meet optional requirement O1 (semitransparent shadows) however, for reasons that will be explained in this report.

2 Background information

To follow the implementation described in this report, being familiar with the following basic theory is advised:

2.1 Ray tracing

Ray tracing is performed by casting *primary rays* from a view point through each pixel in the target image. Recursively, subsequent *reflective* and *refractive rays* are created at each hit in the scene as illustrated in figure 1. Additionally lighting is calculated for each light source that can be reached with a *shadow ray*, resulting in hard shadows. The resulting colors at subsequent ray hits are used to calculate the resulting colors at previous hits, resulting in reflections/mirroring and refractions/transparency. [4]

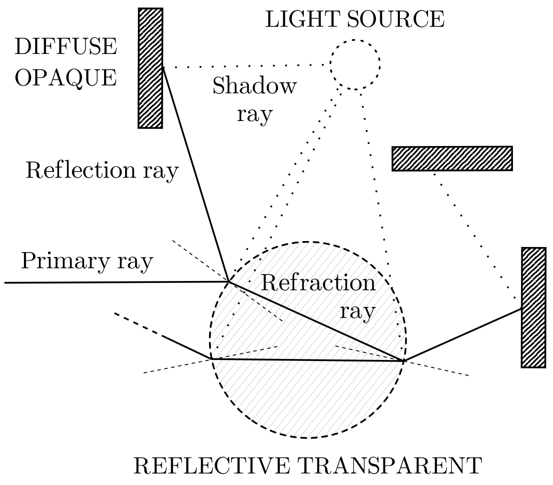


Figure 1: Illustration of ray tracing with a single primary ray splitting into reflection and refraction rays against a reflective transparent sphere before hitting diffuse opaque blocks. At each hit, shadow rays are cast towards a single light source to check of lighting should be calculated. The color at each hit is calculated from subsequent hits.

2.2 Refractions – Snell's law

When light passes from a medium to another, with different refractive indices n_1 and n_2, it refracts resulting in a different angle of refraction θ_2 from the initial angle of incidence θ_1. This relation is described in equation 1 and illustrated in figure 2. [5]

$$\frac{\theta_1}{\theta_2} = \frac{n_1}{n_2} \tag{1}$$

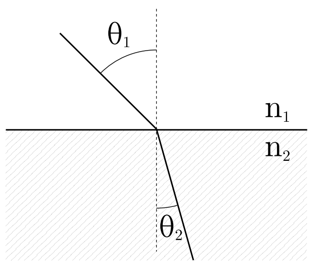


Figure 2: Illustration of a light ray refracting when passing between two mediums of different refractive indices.

2.3 Light falloff – inverse square law

Light intensity is inversely proportional to the distance from the light source squared [6]. This results in significantly higher light intensity near light sources.

2.4 Voxels

Like a *pixel* represents a value in a two-dimensional grid, a *voxel* represents a value in a three-dimensional grid.

3 Implementation

As stated previously, the implementation is primarily done in a GLSL fragment shader for the standard OpenGL pipeline. Camera movement, voxel generation and general overhead is implemented in C++ to be run the CPU. The implementation in its entirety is available on GitHub [7].

3.1 Tools

Specifically, OpenGL version 4.6, GLSL version 4.60 and C++14 version 201402L. The C++ compiler g++ version 9.2.1 has been used. The implementation was developed on Ubuntu 19.10 using an Nvidia GeForce MX150 graphics card with Nvidia 435.21 drivers. For window management and basic OpenGL utilities, the lab utilities for the TSBK03 course [8] are used. These have been somewhat modified however, including changes required for successfully compiling the code with g++. Since the code for the fragment shader is very long, it is split into multiple files that are included into a single final fragment shader file using GLSL Shader Includes [9].

3.2 Voxel generation and representation

The voxels to be visualized are generated on the CPU. This is done in a three-level nested loop going through each coordinate (x, y, z) in voxel space. Depending on a set of rules, different material indices are set for each voxel in a three-dimensional array representing the voxel space. The different materials are given in table 1.

Table 1: Materials used and their respective properties including Phong and ray tracing weights.

Material	Color	Diffusivity	Specularity	Reflectivity	Refractivity	Refraction index
Void	Black	0.0	0.0	0.0	1.0	1.0
Glass	White	0.2	0.7	0.3	0.8	1.5
Semi-solid	White	0.5	0.7	0.3	0.2	1.5
Solid	White	0.6	0.5	0.2	0.0	1.0

All the following percentages refer to percentages of the total radius of the voxel space. The resulting voxel space from the following rules is illustrated in figure 3:

- The outer layer of the voxel space is clad with glass panes with radius 87.5 % surrounded by a solid frame.
- Void holes of radius 25 % are at the center of each glass frame.
- At the front side of the voxel space, no glass pane is generated and the space is left void.
- The interior of the voxel space is all void except from a semi-solid cube of radius 12.5 % in the center.

The resulting array of integers is uploaded to a 3D texture on the GPU to be accessed through a `sampler3d` in the fragment shader.

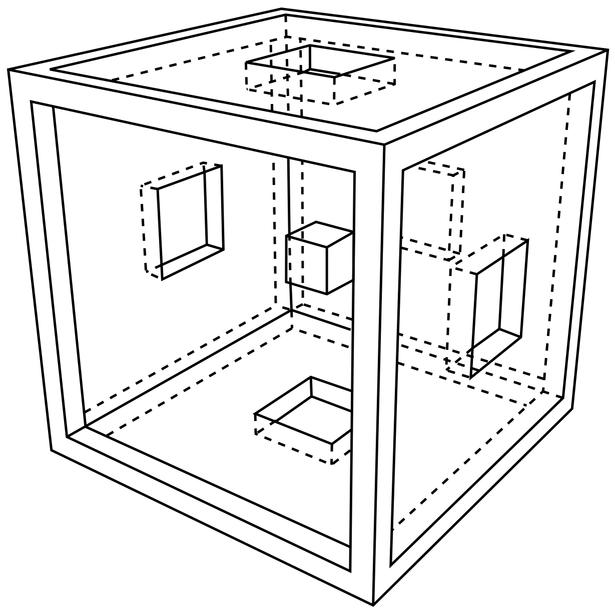


Figure 3: Illustration of the generated voxel space. Dashed lines represent edges that are visible through semi-transparent materials.

3.3 Ray tracing in GLSL

The implementation follows the ray tracing algorithm as usual. Primary rays are casted through the scene and if hit materials have a diffusivity or specularity value above 0.0, a shadow ray is cast towards each light source and for all reached light source, the Phong shading model is applied. The inverse square law is also applied before calculating the diffuse and specular lighting for a light falloff effect. If reflective or refractive materials are hit, new reflection and refraction rays are created, which directions are calculated using the `reflect` and `refract` GLSL methods respectively. The Phong shading model, reflection and refraction all require the normal of the hit surface. Calculations of surface normal has therefore been added to the ray marching and ray casting algorithms used, as described in sections 3.4 and 3.5 respectively.

The only polygons in the scene however are the two which make up the surface onto which the ray tracing results are drawn. The vertex shader for these polygons simply calculates the origin position of the primary rays based on the current camera orientation and screen width-to-height ratio. These positions are then passed to the fragment shader before which they are interpolated.

For each fragment, the fragment shader then calculated the direction of a primary ray, dependent on a uniform view position, before sending it out. GLSL does not support recursive function calls however, which is how the ray tracing algorithm is usually described when it comes to reflection and refraction rays. Instead, as suggested by Stack Overflow user Spektre [10], a standard GLSL array is used as a queue, stack and binary tree to schedule, process and apply ray tracing iterations. These iterations are represented as defined in listing 1.

Starting with the primary ray iteration placed in the array, all iterations in the array have their rays sent out in the order in which they were added. If a reflective or refractive surface is hit, more ray iterations are added to the array. Since this can result in up to two new rays for each cast ray, the maximum number of iterations, and therefore also the required size of the array, depends on the maximum recursion depths for reflection and refraction as described in the macro defined in listing 2. A short mathematical proof is given in equation 2. If an iteration results in the creation of new ray iterations, the array indices of these are stored in the iteration `struct` for later use.

Listing 1: Ray tracing iteration struct.

```
struct RaytraceIteration {
    Ray ray;
    int recursion_depth;
    int void_value;
    int refl_i; // Index of reflection ray
    int refr_i; // Index of refraction ray
    bool has_hit;
    RaymarchVoxelHit hit;
    vec3 color;
};
```

Listing 2: C/C++ macro for calculating the maximum possible number of ray tracing iterations.

```
#define MAX_ITERATIONS \
    ((1 << (min(MAX_REFLECTION_DEPTH, MAX_REFRACTION_DEPTH)) + 1) - 1 \
    + abs(MAX_REFLECTION_DEPTH - MAX_REFRACTION_DEPTH) \
    * (1 << min(MAX_REFLECTION_DEPTH, MAX_REFRACTION_DEPTH)))
```

$$2^0 + 2^1 + ... + 2^{d_{min}} + (d_{max} - d_{min}) * 2^{d_{min}} = 2^{d_{min}+1} - 1 + (d_{max} - d_{min}) * 2^{d_{min}} \tag{2}$$

After all rays have been cast, all subsequent iterations are stored after their parent iterations. The array is therefore iterated through backwards to determine the color at each hit. Shadow rays are cast and the Phong shading model is applied. What is interesting to note here is that all iterations have references to any resulting reflection and refraction iterations in the form of indices. These can and are therefore applied when parent colors are determined.

3.4 Ray marching in grid

To ray cast in voxel space, a ray marching in grid algorithm described in a paper by Amanatides et al. [11] is used. To implement details missing from the paper, a Lua implementation by Schumacher [12] was used as reference.

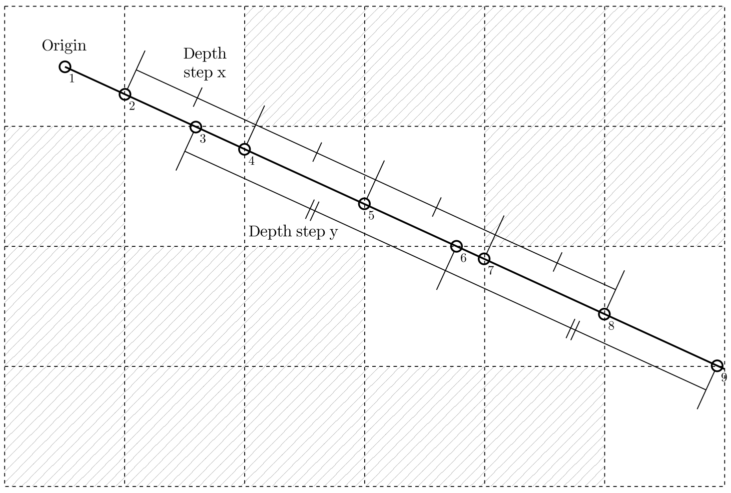


Figure 4: Illustration of the ray marching algorithm in a 2D grid. Circles represent hits/iterations and hit tiles are white. The numbers next to the hits represent their iteration depths. The depth steps for traversing along the x and y axis are marked out. Adapted from a screenshot by Schumacher [12].

The algorithm consists of stepping between integer voxel coordinates. The coordinates resulting in a minimum *next depth* are chosen. This depth is a floating point value representing the depth along the ray required to reach said coordinate. This is done by keeping track of the resulting next depth after traversing along the respective axes. The next depth for an axis changes by the *depth step* of said axis when traversing along it. This is illustrated in two dimensions in figure 4. [11]

In this implementation, next depths and depth steps are stored component-wise in `vec3`:s and are calculated as specified in listings 4 and 5 respectively. `r` is the `Ray`, as defined in listing 3, along which marching is done. `to_voxel` and `to_world` are functions converting between voxel coordinates and world coordinates. `depth` is the depth from the ray origin. For the ray marching, this is not necessary initially 0, as described in section 3.5

Listing 3: Ray representation used.

```glsl
// Ray with origin o, direction dir and inverse (1/dir) dir_inv
struct Ray { vec3 o; vec3 dir; vec3 dir_inv; };
```

Listing 4: Next depth calculations for all axes, in GLSL. Based on Lua code by Schumacher [12].

```glsl
ivec3 voxel_coords = ivec3(floor(to_voxel(r.o + depth * r.dir)));
vec3 init_offset = vec3(
    r.dir.x >= 0.0 ? 1.0 : 0.0,
    r.dir.y >= 0.0 ? 1.0 : 0.0,
    r.dir.z >= 0.0 ? 1.0 : 0.0);
vec3 next_depth = (to_world(voxel_coords + init_offset) - r.o) * r.dir_inv;
```

Listing 5: Depth step calculation for all axes, in GLSL. Based on Lua code by Schumacher [12].

```glsl
ivec3 voxel_step = ivec3(
    r.dir.x >= 0.0 ? 1 : -1,
    r.dir.y >= 0.0 ? 1 : -1,
    r.dir.z >= 0.0 ? 1 : -1);
vec3 depth_step = to_world(voxel_step) * r.dir_inv;
```

The last traversing direction is also stored so that it can be negated on a hit to achieve the normal of the hit voxel surface, which is required as described in section 3.3. This is an addition to the algorithm as it is described by Amanatides et al. [11] and implemented by Schumacher [12].

The primary hit condition for this implementation is traversing into a different material from which the ray started. This achieves desired results for primary, reflection and refraction rays. Initially, the hit condition was traversing into a non-void material. With refraction implemented however, rays can start inside e.g. glass, in which case they should not stop at glass but instead stop at all other materials, including void. Therefore, the hit condition was changed to traversing into a different material.

This change is not enough for shadow rays however when working with multiple transparent and semi-transparent materials. A shadow ray shouldn't stop at void when traversing through glass and neither should it stop at glass when traversing though void. This will result in transparent and semi-transparent materials acting as opaque materials when lighting is to be calculated. A result of this can be seen in figure 9e when compared to figure 9f. This is why the optional requirement O1 had to be met.

A secondary hit condition exclusive to shadow rays is clearly needed. A simple condition of only checking for opaque materials would fix this problem but materials with refractivity 0.9 would, like void, cast no shadow at all which is undesirable. The best solution would be to stop at opaque materials but also take all traversed materials into account so that the longer the ray has passed through semi-transparent materials, the darker the shadow will appear. It proved difficult however to write a formula resulting in an overall refractivity depending on different refractivities weighted with different traverse distances. Instead, a sufficient compromise was achieved by setting the the resulting overall refractivity to the maximum refractivity of any traversed material.

Ray marching for shadow rays should clearly also stop before the light source they march towards. This is achieved by returning a miss when the depth exceeds the distance to the light source. Unsurprisingly, exciting the voxel space also returns a miss.

3.5 Ray casting against AABB

The previously described ray marching algorithm assumes that the ray is cast from inside the grid. Instead, if cast from outside the grid, the ray will not hit anything inside the grid. This results in no voxels being rendered when viewed from outside the voxel space.

To add support for rendering the voxels from outside the voxel space, an initial ray cast is done against the voxel space' AABB (axis aligned bounding box). The resulting position and depth of this ray cast is then used as inital values for the ray marching as described in section 3.4.

Listing 6: AABB ray casting implementationin GLSL.

```glsl
/**
 * Sets information about the first side of the specified AABB, hit by the
 * specified ray, to the hit parameter.
 *
 * Returns true if there was a hit or false otherwise.
 */
bool raycastAABB(const Ray r, const AABB aabb, out RaycastAABBHit hit)
{
   vec3 dist_lo = (aabb.lo - r.o) * r.dir_inv;
   vec3 dist_hi = (aabb.hi - r.o) * r.dir_inv;
   vec3 dist_min = min(dist_hi, dist_lo);
   vec3 dist_max = max(dist_hi, dist_lo);
   float dist_max_min = min(min(dist_max.x, dist_max.y), dist_max.z);

   float dist_min_max;
   vec3 normal = vec3(0.0);
   if (dist_min.x >= dist_min.y) {
      if (dist_min.x >= dist_min.z) {
         normal.x = -r.dir.x;
         dist_min_max = dist_min.x;
      }
      else {
         normal.z = -r.dir.z;
         dist_min_max = dist_min.z;
      }
   }
   else if (dist_min.y >= dist_min.z) {
      normal.y = -r.dir.y;
      dist_min_max = dist_min.y;
   }
   else {
      normal.z = -r.dir.z;
      dist_min_max = dist_min.z;
   }

   if (dist_min_max > dist_max_min) {
      // No intersection
      return false;
   }

   float depth = max(dist_min_max, 0.0);
   hit = RaycastAABBHit(r.o + depth * r.dir, depth, normalize(normal));
   return true;
}
```

Listing 7: AABB ray casting result struct.

```
struct RaycastAABBHit {
    vec3  world_pos;
    float depth;
    vec3  normal;
};
```

The function defined in listing 6 is for the AABB ray casting. It sets an instance of the `struct` specified in listing 7 with information on the resulting hit. This implementation is based on two AABB-ray intersection implementations [13, 14] but with more descriptive variable names. The implementation used in this project also extends the previous implementations by additionally finding the normal of the AABB at the hit location. This is required, as described in section 3.3, for the voxel space's outermost surfaces. Without this additional requirement, the lines 15 through 34 in listing 6 could be reduced to `float dist_min_max = max(max(dist_min.x, dist_min.y), dist_min.z)`.

This intersection check is clearly faster than traversing while sampling a texture at each step. The implementation therefore runs significantly faster when the camera is placed outside the voxel space aimed away from it. In this case, the AABB ray cast intersection check will return false and a miss can be returned without even starting ray marching.

4 Interesting problems

Since the ray tracing had to be implemented from the ground up, the initial part of the project was plagued with multiple errors. This made it difficult to discern if the voxel generation and camera movements were working as intended. Below, some of these early problems are described and illustrated with reproduced screenshots.

4.1 Random misses/self-hits – spotty surfaces

Before the AABB ray casting was implemented, the voxel space was completely invisible when viewed from the outside, as described in section 3.5. When first implemented, the voxel space was made visible from the outside but only at random spots, as seen in figure 5. This was the result of random ray marching misses.

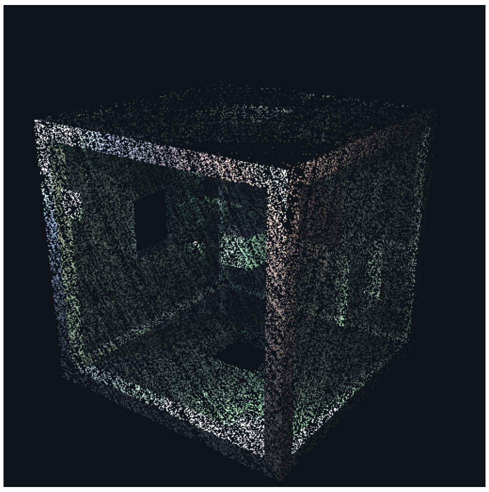


Figure 5: Rendering with AABB set to the exact dimension of the voxel space, resulting in random misses for subsequent ray marching rays due to precision errors.

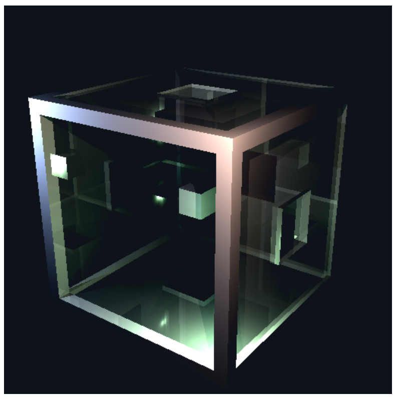
These misses were caused by the ray marching beginning at the calculated depth of the bounding voxel space surface. Theoretically, this depth value would be exactly at the surface but due to precision and rounding errors, this depth sometimes is too shallow. This leads to the ray marching missing as it starts outside the voxel space. This was solved by making the AABB slightly smaller than the voxel space with a margin of 0.0001 on each side.

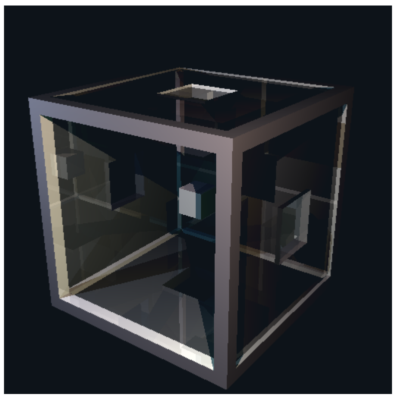
A similar effect was caused by shadow, reflection and refraction rays occasionally originating from inside the surfaces they were cast from, resulting in spotty lit surfaces, reflections and refractions respectively. This was fixed by offsetting the each ray's origin by 0.001 along the normal of the originating surface.

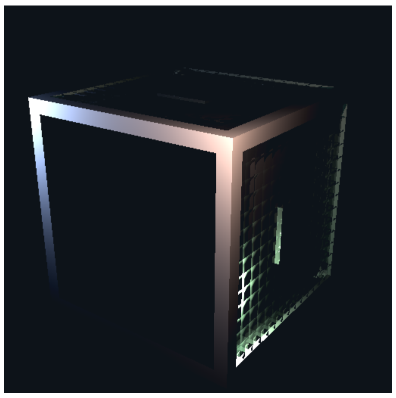
4.2 Incorrect depth calculations – visible planes and overestimated depths

When initially adapting the ray marching algorithm [11] and the Lua implementation [12], some experimentation was required since these were not completely understood. During this experimentation, an incorrect depth calculation method was implemented. `next_depth` was at the time incorrectly called `d_depth` due to misunderstanding the purpose of the variables in the reference algorithm [11] and implementation [12]. Thus, it was incorrectly used as follows: `depth += next_depth.x` when traversing along the x axis and correspondingly for the y and z axes.

This error was not discovered until shadows and light falloff were to be implemented. As illustrated using a fog effect in figure 6, this resulted in too high calculated depths inside voxel space. Planes between voxels were also visible, due by the the depth calculations getting more incorrect with each step between voxels. The error was fixed by changing the calculation to `depth = next_depth.x` for x and correspondingly for the y and z axes.


(a) Rendering with light falloff and correct depth calculations.


(b) Rendering without light falloff and correct depth calculations.


(c) Rendering with light falloff and incorrect depth calculations.

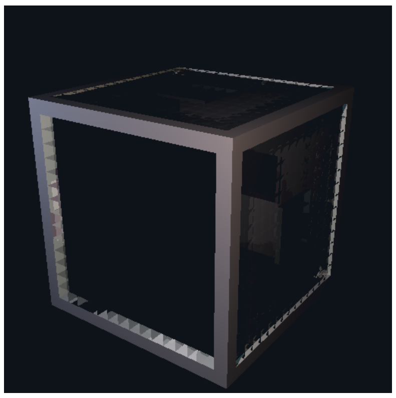
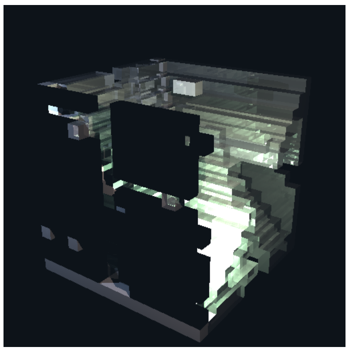

(d) Rendering without light falloff and incorrect depth calculations.

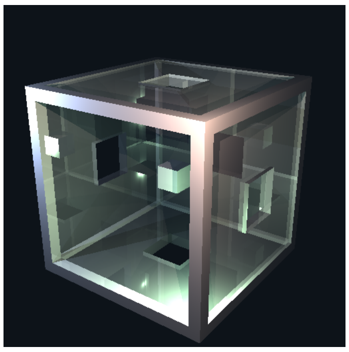
Figure 6: Renderings comparing the results of correct and incorrect depth calculations on a fog effect.

4.3 Non-power-of-two texture widths – heavy distortions

One initial problem was not caused by either incorrect voxel generation or rendering. It did however lead to extensive debugging attempts looking through and trying changing these aspects.

Initially, odd texture widths were used so that a single block could be placed in the center. It turned out however that widths of 3D textures need to be powers of two or else the data inside is distorted beyond recognition. As exemplified in figure 7, simply being one value off from a power of two results in heavy distortions. Examples of correct representations at different resolutions are given in figure 8.


(a) Width 15 (distortions).


(b) Width 16.

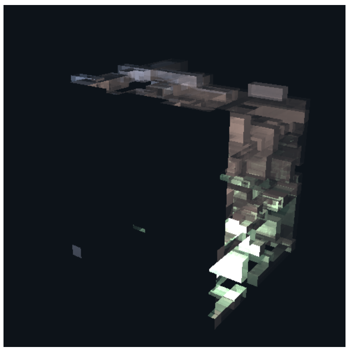
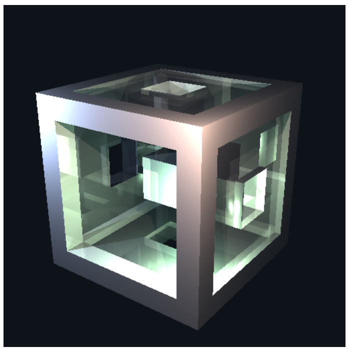
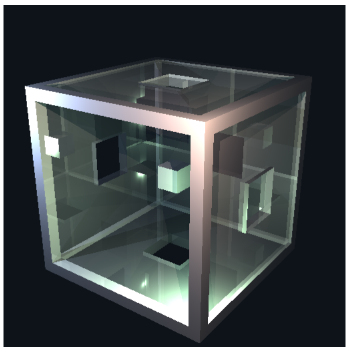

(c) Width 17 (distortions).

Figure 7: Renderings with different widths for the cubical Texture3D used for holding voxel data. Heavy distortions are visible for non-power-of-two widths.


(a) Width 8.


(b) Width 16.

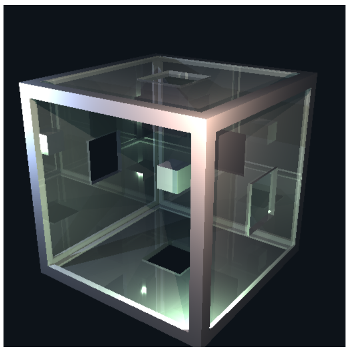
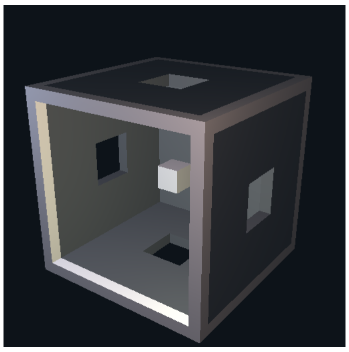
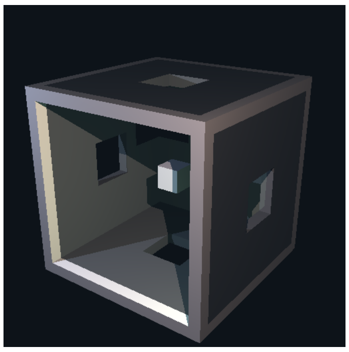

(c) Width 17.

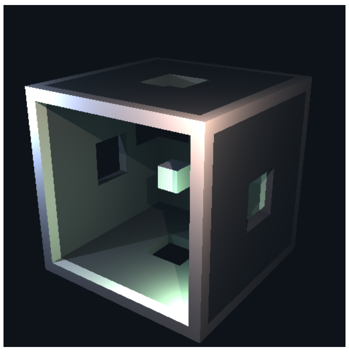
Figure 8: Renderings with different Texture3D widths, all being powers of two, resulting in correct representations. Note however that the voxel generation as described in section 3.2 has to be modified in order to generate the texture of width 8 correctly. Due to the low resolution, the radius, within which glass walls are generated, needs to be reduced from the usual 87.5 % to 75 % of the total voxel space radius. Otherwise, no solid outer frame is generated.

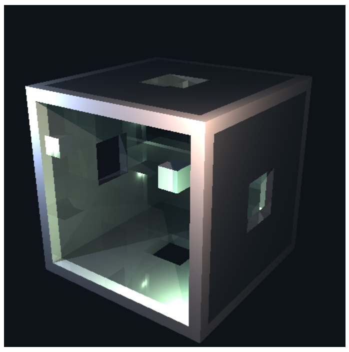
5 Conclusions

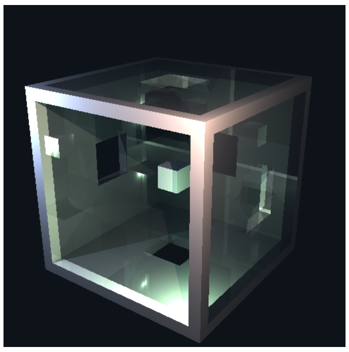
The visual results of the project are very satisfactory in my opinion. As showcased in figure 9, all mandatory requirements except from voxelization have been implemented. In the scene, there is a red point light source between the voxel space and the camera, a blue light off to the left side and a green one in the bottom corner closest to the camera, inside the voxel space interior. Specular light effects might not be noticeable in these screenshots, but they are in figure 10.

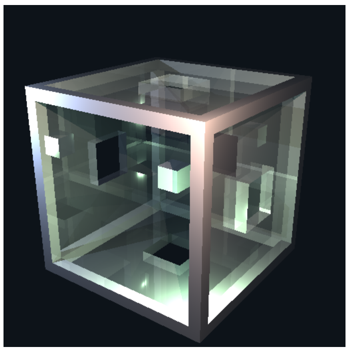

(a) Primary rays and Phong shading.


(b) Shadow rays.


(c) Light falloff.


(d) Reflection rays.


(e) Refraction Rays.


(f) Semi-transparent shadows.

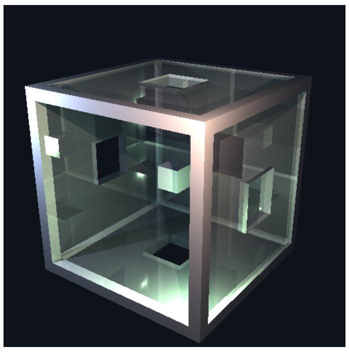

(g) Refraction indices.

Figure 9: Renderings showcasing different implemented ray tracing and lighting features. Subsequent renderings show all features of previous renderings as well their own specific described features.

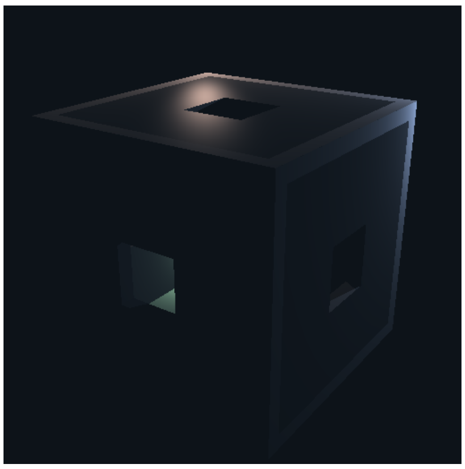


Figure 10: Rendering showing a red specular light effect on the top surface of the voxel space and blue one in the top right corner.

5.1 Performance

Something that should be obvious to anyone running the implementation is that it runs very slow. On the development setup used, single digit frames per second are achieved when rendering to a resolution of 512 px by 512 px with maximum recursions depths set to 4 for both recursion and refraction. For reference, these are the conditions during which all screenshots in this report were taken. The front cover image however was rendered at 1200 px by 1200 px.

The performance increases significantly when reducing the target resolution, recursion depths and texture resolution and decreased significantly when increasing these parameters. This hints at an obvious conclusion: nested loops of sampling textures is expensive. A possible fix could be to change to another fundamental data structure which required fewer ray marching iterations such as an octree, as suggested by the optional requirement O2.

5.2 What I have learned

Throughout this project, I have gotten a solid grasp on ray tracing, some details of which I had misunderstood beforehand. This is also my first time time working with 3D textures as well as voxels and visualization of them. I have also been forced to work with different code structures than I've been used to, with iteration forwards and backwards to achieve recursive effects without recursion being the primary example.

It was also interesting to get more experience with debugging graphical errors. This made up a majority of my initial time spent on this project, as the ray tracing and voxel generation were both built from the ground up with the only visuals being debug prints of the voxel array to the console and psychedelic, noisy, mind-bending visuals clad in debug colors displayed on screen. The voxel generation could definitely have been completed beforehand, which it was in a way. Due to using a non-appropriate texture width however (see section 4.3 and figure 7), the visual results would have appeared incorrect even with a perfectly implemented rendering method. Visualizing the 3D texture in another way, before moving on to ray tracing, could definitely have been done however, which I will learn from. I will also remember to always use texture widths that are powers of two in the future.

Something that I have not learned however, is what sources to cite for basic physics knowledge. Going back to the original source can be difficult when its from the 17th century and written in French or Latin.

References

[1] Arturo García et al. "Interactive Ray Tracing Using the Compute Shader in DirectX 11". In: *GPU Pro.* Ed. by Wolfgang Engel. 3rd ed. Boca Raton, FL, USA: CRC Press, 2012. Chap. 3, pp. 353–376. ISBN: 978-1-4398-8794-3.

[2] David Luebke and Steven Parker. *Interactive Ray Tracing with CUDA.* Nvision 08. Nvidia Research. 2008. URL: `https://www.nvidia.com/content/nvision2008/tech_presentations/Game_Developer_Track/NVISION08-Interactive_Ray_Tracing.pdf`.

[3] Masahiro Fujita and Takahiro Harada. *Foveated Real-Time Ray Tracing for Virtual Reality Headset.* 2014. URL: `https://pdfs.semanticscholar.org/e440/ec75cfb9b582dfc78a35b0ac7b2fb1d9281d.pdf`.

[4] Turner Whitted. "An Improved Illumination Model for Shaded Display". In: *Communications of the ACM* 23.6 (1980-06), pp. 343–349. ISSN: 0001-0782. DOI: 10.1145/358876.358882.

[5] René Descartes. "La Dioptrique". In: *Discours de la Méthode Pour bien conduire sa raison, et chercher la vérité dans les sciences.* 1637. URL: `https://gallica.bnf.fr/ark:/12148/btv1b86069594/f83.item`.

[6] Ismaël Boulliau. "In quo de coelo, seu universo, de sole, et de planetarum motu in genere disquiritur". In: *Astronomia philolaica*. Paris, France, 1645. Chap. 12, p. 23. URL: `http://diglib.hab.de/drucke/2-1-4-astron-2f-1/start.htm`.

[7] Axel Nordanskog. *TSBK03 Project – Ray Tracing in Voxel Space*. GitHub. 2019-11-12. URL: `https://github.com/axerosh/raytracing` (visited on 2019-12-29).

[8] Ingemar Ragnemalm. *Laboration 0*. 2019-09-04. URL: `http://computer-graphics.se/TSBK03/laboration-0.html` (visited on 2019-09-28).

[9] Tahar Meijs. *GLSL Shader Includes*. GitHub. 2018-04-23. URL: `https://github.com/tntmeijs/GLSL-Shader-Includes` (visited on 2019-10-29).

[10] Spektre. *Reflection and refraction impossible without recursive ray tracing?* Stack Overflow. 2017-07-17. URL: `https://stackoverflow.com/a/45140313` (visited on 2019-10-30).

[11] John Amanatides and Andrew Woo. "A fast voxel traversal algorithm for ray tracing". In: *Eurographics*. Vol. 87. 3. 1987, pp. 3–10. URL: `http://www.cse.chalmers.se/edu/year/2012/course/_courses_2011/TDA361/grid.pdf`.

[12] Joel Schumacher. *Ray Casting in 2D Grids*. 2016-02-10. URL: `https://theshoemaker.de/2016/02/ray-casting-in-2d-grids/` (visited on 2019-10-29).

[13] Hugh Kennedy. *ray-aabb-intersection*. GitHub. 2015-09-30. URL: `https://github.com/stackgl/ray-aabb-intersection` (visited on 2019-10-29).

[14] warvstar. *Fast glsl ray box intersection*. Reddit. 2018-06-02. URL: `https://www.reddit.com/r/opengl/comments/8ntzz5/fast_glsl_ray_box_intersection/dzyqwgr` (visited on 2019-10-29).