



SCHOOL OF COMPUTATION, INFORMATION
AND TECHNOLOGY

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Volumetric Path Tracing inside the Unity Game Engine Extended

Lucas Denhof





SCHOOL OF COMPUTATION, INFORMATION
AND TECHNOLOGY

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Volumetric Path Tracing inside the Unity Game Engine Extended

Volumetrisches Path Tracing in der Unity Game Engine Erweitert

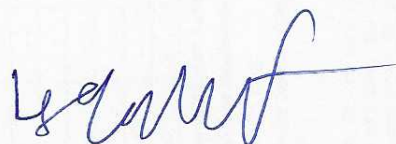
Author:	Lucas Denhof
Supervisor:	M.Sc. Constantin Kleinbeck
Advisor:	Prof. Dr. rer. nat. Daniel Roth
Submission Date:	3rd November 2025



I confirm that this bachelor's thesis in informatics: games engineering is my own work and I have documented all sources and material used.

Munich, 3rd November 2025

Lucas Denhof

A handwritten signature in blue ink, appearing to read 'Lucas Denhof', with a long horizontal stroke extending to the right.

Abstract

This thesis continues builds on previous work in making a volumetric path tracer in the Unity game engine. With the goal of better integrating the path tracer into Unity, features such as occlusion, support for multiple volumes, visibility outside of play mode and in the scene view have been added, as well as various other bug fixes and quality-of-life improvements. The tool aims to allow easy and highly detailed viewing of three-dimensional volumes, especially CT and MRI scans, which can be simply imported as DICOM files. By making it in Unity it will be highly portable and have cross-platform functionality, allowing interaction in nonstandard ways, for example in VR. This paper documents the various methods used, the problems encountered along the way, and their solutions.

Kurzfassung

Diese Arbeit baut auf früheren Arbeiten zur Erstellung eines volumetrischen Pfadverfolgers in der Unity-Spielengine auf. Mit dem Ziel, den Pfadverfolger besser in Unity zu integrieren, wurden Funktionen wie Okklusion, Unterstützung für mehrere Volumen, Sichtbarkeit außerhalb des Spielmodus und in der Szenenansicht hinzugefügt, sowie verschiedene andere Fehlerbehebungen und Verbesserungen der Benutzerfreundlichkeit. Das Tool soll eine einfache und hochdetaillierte Betrachtung dreidimensionaler Volumen ermöglichen, insbesondere von CT- und MRT-Scans, die einfach als DICOM-Dateien importiert werden können. Durch die Entwicklung in Unity ist es hochgradig portabel und plattformübergreifend einsetzbar, was eine Interaktion auf nicht standardmäßige Weise ermöglicht, beispielsweise in VR. Diese Arbeit dokumentiert die verschiedenen verwendeten Methoden, die dabei aufgetretenen Probleme und deren Lösungen.

Contents

Abstract	iii
Kurzfassung	iv
1 Introduction	1
1.1 Goals	2
2 Related Work	3
3 Methods	5
3.1 Volume importing	5
3.2 Scene view and edit mode	5
3.3 Occlusion	6
3.4 Depth sorting	8
4 Results	9
4.1 Image quality	9
4.2 Transfer functions	11
4.3 Benchmarking	13
5 Discussion	19
5.1 Summary	19
5.2 Interpretation	19
5.3 Limitations	19
6 Future Work	21
6.1 Color changing when scaling	21
6.2 White volumes in edit mode	21
6.3 Depth buffer noise	21
6.4 Flickering game view	22
6.5 More issues	22
7 Conclusion	23
8 AI Usage	24
Bibliography	25

1 Introduction

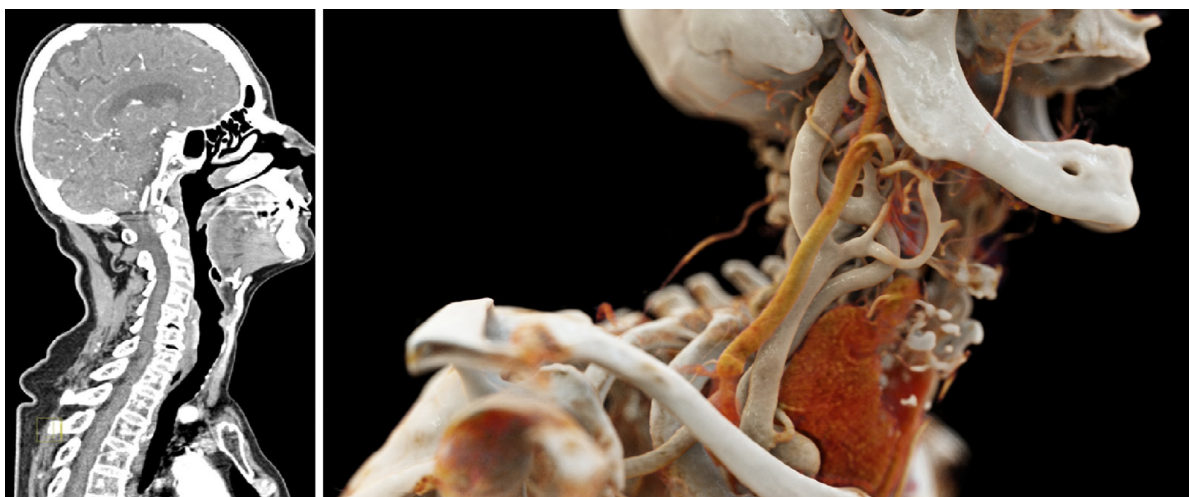


Figure 1.1: Left: Original CT data. Right: Cinematic rendering of the same dataset[1].

For a long time, doctors have been using two-dimensional slices of MRI scans to glean information out of the injuries their patients have suffered. However, this approach is flawed: We are three-dimensional creatures, and so are our injuries. The absence of an extra dimension is limiting doctors and their diagnoses.

Why not leverage the vision we have evolved with, and use realistic three-dimensional renders to create more intuitive and helpful tools for doctors? Such photorealistic renders will aid with our natural object recognition and depth perception, helping doctors reach the correct diagnosis. These images may also prove to be more intuitive for people who aren't familiar with interpreting 2D scans, such as young doctors or medical students, or even the patients themselves.

As it stands, two-dimensional renderers have been used for decades and many doctors have familiarised themselves with these slices. Three-dimensional renderers have also existed for some time, but they didn't always look photorealistic. Only in recent years have graphics cards, that are capable of rendering complex ray-traced 4K scenes in realtime or even faster, become so ubiquitous. The quality and interactivity of renderers, powered by these graphics cards, warrents investigating photorealistic renderers as a supplement, or alternative to the currently used two-dimensional renderers.

This thesis aims to investigate the feasibility of using volumetric path tracing to create three-dimensional renders. We seek to integrate them into the Unity game engine, allowing for easy, cross-platform, and real-time use in a variety of applications, perhaps even VR.

Standard medical formats for CT and MRI scans, such as DICOM, are to be easily imported, and the transfer function, which maps tissue to color and opacity, should be freely editable.

By doing all this, we wish to provide a simple but powerful tool for the medical world. Our hope is to provide an easy stepping-stone for others who wish to do research in this area.

1.1 Goals

Our primary goal is to better integrate the previous work we are building upon into the Unity game engine.

A game engine is a much greater tool to work with than the alternative: Many renderers are shipped as standalone applications locked to an operating system or two, and can be frustrating for people to experiment with or expand upon. Unity offers a simplified interface, and is a software that people may already be familiar with. Having a renderer available there allows scientific experimentation and research into areas with significantly greater ease.

We recognize that the technology of the renderer, and the quality of an image are not the only points of concern, and that there must be ease of use for both developers and doctors alike, otherwise neither will use them. In the work we are here continuing, many basic features were missing that are absolutely necessary for normal use. These must be dealt with before a proper path tracer can become adopted by the people in the medical industry.

After shining a light on the previous work we are building upon, we will explain the details and thoughts behind all the features we implemented. Then we will talk about the quality of the images that can be made with the final renderer and how long it takes to make them. Once it has all been discussed, we will explore opportunities of future research, before finally concluding the paper.

2 Related Work

Hofmann and Evans [3] created the structure for a volumetric path tracer which was implemented in VolRen in OpenGL, and it is the core algorithm of much that will be built in this paper.

Their approach works by physically simulating the full paths of light rays. Doing so means the image will start noisy, but over time as more and more samples are made, the image quickly becomes less and less noisy.

Giese [2] worked on porting the shader code of the VolRen renderer into HLSL, so that it would work in the Unity game engine. Doing so offers great flexibility, allowing others to use the path tracer in a way that is more platform independent, as Unity can export not just to desktop computers, but also to phones, and even VR.

It also becomes easier for people to add the features that they require. Many other renderers are shipped as standalone software and are not easily extendible, but a game engine offers a more appealing interface, and will often abstract away a lot of the complexity. Giese's work is the primary basis upon which we started, and it is this work we have continued upon.

Lavik [6] created UnityVolumeRendering, a project which implements a ray marching algorithm in Unity. There are many similarities between our two projects, as both aim to offer rendering of medical volumes in a game engine for maximum portability.

The biggest difference between Lavik's work and ours, is the technology underlying the volume renderer: UnityVolumeRendering uses a ray marching algorithm, which is significantly faster, but much harder to implement and with not as much graphical fidelity. On the other hand, we use a path tracer, which offers incredibly realistic images and makes many graphical effects trivial to add, at the cost of rendering speed.

From here is also where we get many of our Unity specific features from, such as the transfer function editor, and the DICOM volume importing via SimpleITK, as well as the idea of making the renderer work both in the scene view, and outside of play mode.

A transfer function is commonly used when rendering medical data, because a CT or MRI scan will output a three-dimensional grid of densities. The job of a transfer function is to map these densities to relevant colors, for example making the dense bones white and the less dense skin see-through, so a doctor can investigate a broken arm.

Ljung et al. [7] report on the state-of-the-art research on transfer functions. They talk about transfer functions of one dimension, two dimensions, and even higher dimensions. Other topics they discuss include rendering in HDR, animating the data, generating transfer functions automatically, and more.

Comaniciu et al. [1] explore various techniques to improve medical imaging. Among other things, they discuss their Cinematic Rendering renderer, which creates impressive images using Monte Carlo path tracing. They also discuss the application of artificial intelligence in

medical imaging, for example to automatically identify and clearly differentiate organs, as well as scanning with minimally invasive procedures.

Kroes, Post, and Botha [5] show the approach they used for their Exposure Render application, which uses Monte Carlo ray tracing to render volumes, and thereby allows a large variety of effects to be easily and flexibly implemented.

They also mention how realistic images aid a person's depth perception and ability to gauge the size of an object. Additionally, they explore using a noise reduction filter to clear up the initial noisy samples, a noteworthy approach which could be applied to our renderer as well.

Sutherland et al. [9] discuss medical imaging in the context of virtual reality (VR) and augmented reality (AR). They talk about the history of VR and AR, and explain the limitations and benefits of using these technologies in the medical space. They explore uses for patients and for clinicians, using AR and VR for therapy or surgery, and even talk about 3D printing for use in surgical planning. Their work shows some of the benefits that being able to export to VR/AR could have.

TrueTrace[8] is a project that brings path tracing into Unity, even on low-end devices without GPUs and ray tracing cores. Unlike our work, which focuses on rendering three-dimensional volumes and everything inside of them, their work focuses on the standard meshes that are used in most of computer graphics, which define the outer shell of an object with a multitude of triangles.

Hofmann et al. [4] present their work on denoising Monte Carlo path tracing for volumes. They highlight the various difficulties that traditional denoisers face when applied to volumetric data, including their reliance on data like the depth and normal vector to be well-defined and smooth, when neither is the case in volumetric data. Their solution involves a learning based method in order to achieve denoising, and could prove useful as a powerful denoiser for future work in this project.

3 Methods

3.1 Volume importing

The first issues we fixed relate to the SimpleITK library, which is responsible for importing the DICOM volume files. Many smaller issues were fixed here, for example:

- Many methods were made async to stop Unity from freezing when bottlenecked by I/O tasks.
- Loading bars were added to communicate progress of long tasks.
- Proper error handling was added to avoid scenarios where the editor would stop responding.
- The folders at the location of the SimpleITK binaries would be created if they didn't exist.
- The logic of downloading and enabling the libraries were more clearly split.
- Code would now look for the names of the binary files to confirm their existence, not just whether any files existed at their location.

3.2 Scene view and edit mode

A much more challenging issue we wanted to fix, was that the shader only worked in the game view, and only while in play mode. These issues can be tackled by adding the [ImageEffectAllowedInSceneView] and [ExecuteInEditMode] attributes to the class. Unfortunately, much of the logic of the script still needed to be reworked, as these attributes are not simply plug and play.

There are many challenges with making a camera effect work in the scene view. First is that there is no clear differentiator between the game view and scene view camera. For most shader effects this is likely not an issue, but since we do not adhere to the standard render pipeline, it is for us. We render the volumes in compute shaders outside of the main render flow, since rendering these volumes can take much longer than a frame, and since we do not wish to slow down the rest of the application.

If we do not differentiate between the scene view and game view, we would see incorrect orientations and positions of the volumes. It would appear as if the volume in one window would have come from the camera of the other. The solution we found to differentiate between these

two views, lies in the fact that the scene view camera is always named *SceneCamera*. During the `OnRenderImage(RenderTexture source, RenderTexture destination)` Unity callback, we simply check whether `Camera.current.name` is *SceneCamera* or not and split our logic accordingly. We also put some extra logic in `OnValidate()` to warn a developer in case they troublingly named the game view camera *SceneCamera*.

The rendering in the scene view also had some other complications: Many times when the volumes needed to be reset, for example when the camera moves and sees them at a different angle, the `needsReset` bool would be set by the scene view camera. However, all changes would then be reset or ignored once the context returned back to the game view, as it seems component variables do not retain their changes when altered in the scene view. This issue was fixed by writing the values where they wouldn't be ignored or reset: `ScriptableObjects`.

We did also have some success using static variables, but in the end these weren't used because of other reasons, namely the fact that static variables aren't serialized by the Unity inspector. This means they aren't easily viewed or changed in the inspector, but more critically, they aren't stored and remembered as variables, which led to some subtle bugs.

3.3 Occlusion

Another of the more challenging features we implemented was occlusion. In the previous work, the rendered image was always placed on top of whatever else was being rendered, even if it should have been behind another object.

Since rendering the volumes can take longer than a frame, we realized that the depth testing should not happen when the volumes are being rendered. Rather, each volume should output its own depth buffer alongside the rendered image.

This is because the merging of the rendered volumes with the camera texture is not expensive, and can be done with each new frame's updated camera color and depth buffer, even while reusing the already rendered volume textures. This means that the volumes will always be correctly behind or in front of other objects, even if the rendered volumes themselves are several frames old.

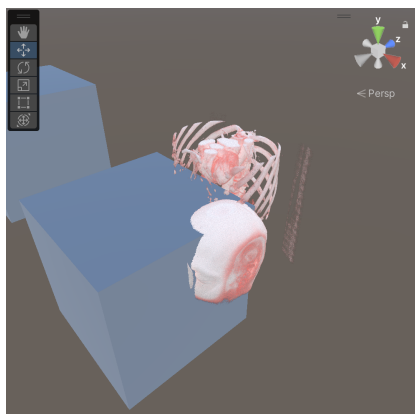


Figure 3.1: Two volumes partially occluding and being occluded by a standard unity cube.

The depth values of the volumes are calculated in the `trace_path` function the first time it intersects the volume. The distances between these intersections and the camera are then stored in an `RFloat` texture. This texture is later used to compare to the camera's depth texture and determine which pixels should be occluded or not.

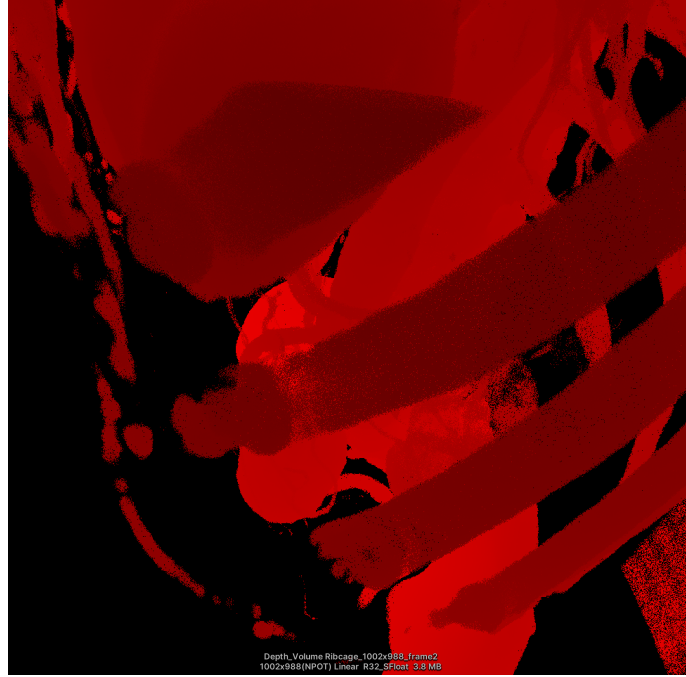


Figure 3.2: The `RFloat` texture the depth values are stored within.

Note that since these depth values are currently recalculated each time a new sample is rendered for the volume, there is some slight noise at the boundary between a volume and whatever object it is intersecting. Perhaps future research could explore how these depth values could be averaged over many samples, like the volume itself is.

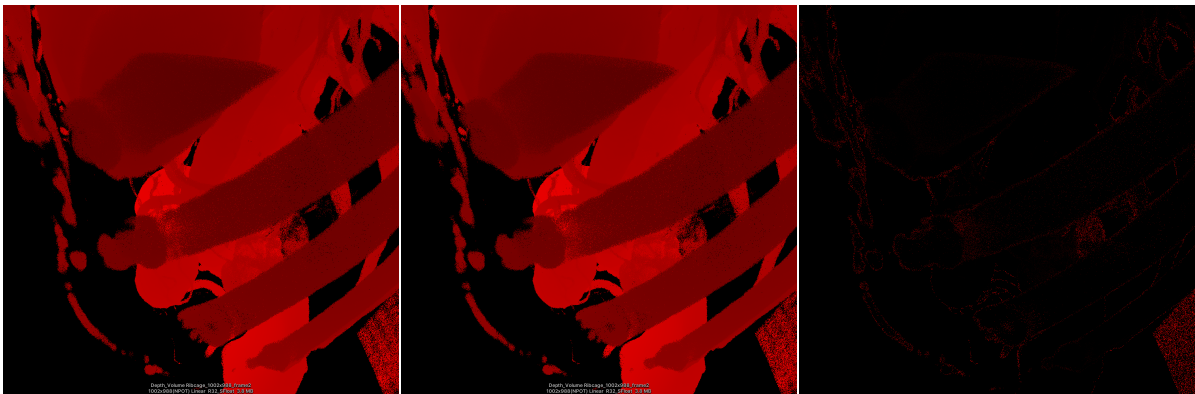


Figure 3.3: Two depth texture samples of the same image, and their difference.

3.4 Depth sorting

Another vital related feature we wanted to implement was the sorting of volumes based on their distance from the camera. Previously, each volume was rendered on top of the preceding one in the order that they were placed in a list. This lead to incorrect outputs, as seen in the right half of Figure 3.4, where the torso volume is rendered in front of the head, despite being behind it.

We fixed this issue by simply sorting the depths of the volumes at each pixel when blending the rendered volumes to the screen. This way, the final output had each volume correctly occluding the other volumes.

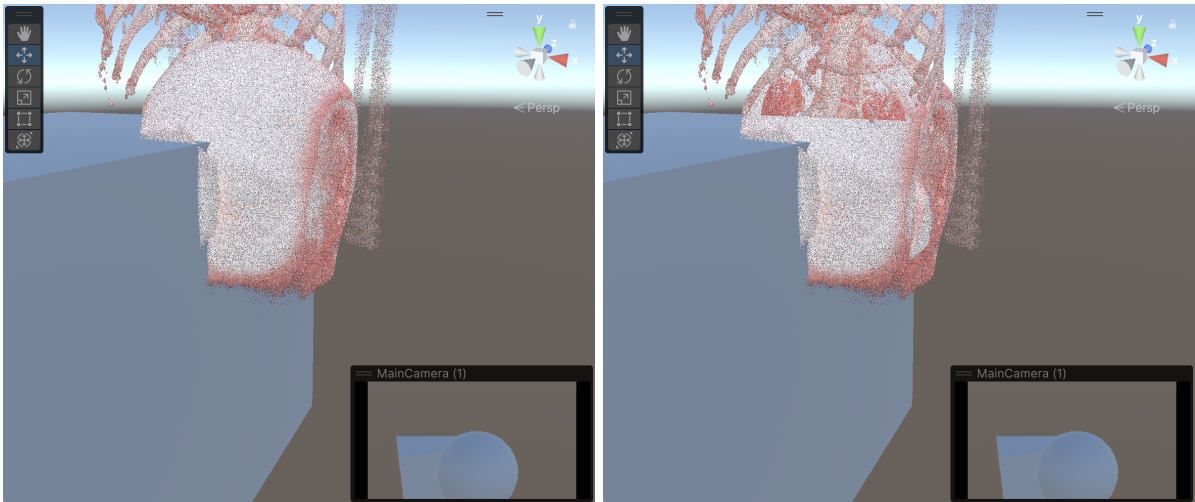


Figure 3.4: With depth sorting vs. without.

The sorting algorithm we chose for this was insertion sort, because we do not expect to be rendering more than just a few volumes. While the complexity of insertion sort is $O(n^2)$, it will still be faster than other algorithms with more favorable time complexities, since insertion sort has significantly less overhead than those other algorithms at this scale.

Since the sorting is done in a shader, we cannot dynamically allocate arrays of any length, so the maximum number of volumes supported this way is limited to a constant. We chose 64, which we do not remotely expect to be surpassed, though this constant can easily be increased if need be.

4 Results

4.1 Image quality

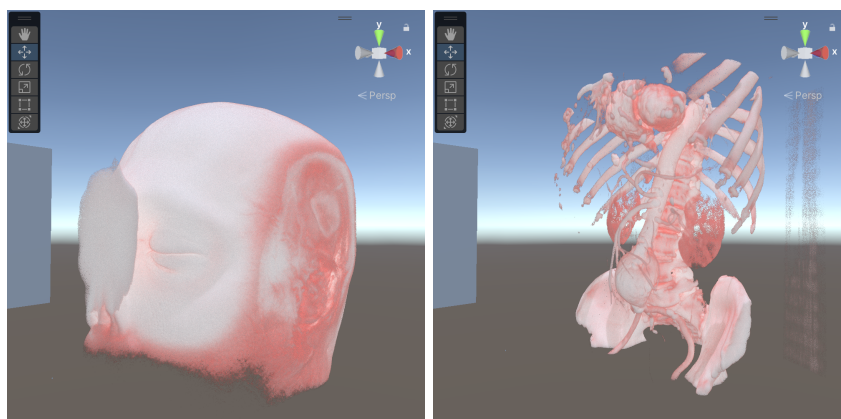


Figure 4.1: Head and torso volumes.

The output of the path tracer increases with fidelity the longer the camera and volumes stay in the same states. Once something changes, for example the position of the camera, the process must start again. The previous image is cleared, and rendered anew, getting better with each sample, as seen in Figure 4.2.

All in all, the path tracer has great flexibility and offers many variables that can be tweaked to achieve a desired image:

- Expected ones, such as the volume data itself, the transforms of the volumes and camera, and the projection matrix of the camera.
- The one-dimensional transfer function can be set for each volume independently.
- The global environment HDRI can be set, as well as the strength of the light it contributes.
- Gamma and exposure can be adjusted for the final image.
- Various volume specific variables, such as the albedo, phase, and number of light bounces, as well as a density multiplier.
- And many more minor variables that deal with performance or disabling some of these features.

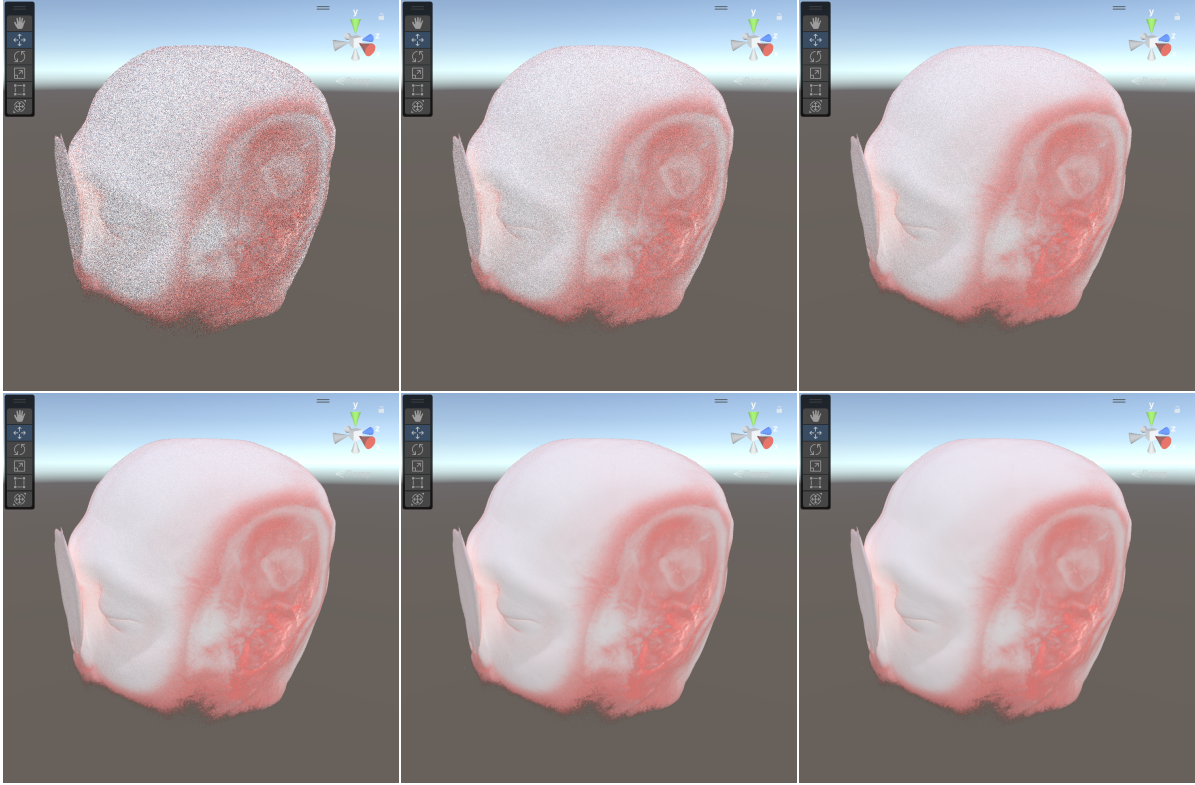


Figure 4.2: The same image with a different number of samples.

In order from left to right, top to bottom: 1, 5, 20, 100, 500, 1000, and 2000 samples.

It is quite useful that common consumer grade hardware can smoothly run a path tracer such as this one. One of the significant benefits of being able to use a fully fledged path tracer, as we do in this work, is the great quality of the output image. We get complex effects such as self shadows, as seen in Figure 4.3, effectively for free. Since we are going so far as to fully simulating the paths of every photon, it would be quite a challenge to get an image that is more realistic than this approach. Another benefit, is that the gates are opened to adding countless effects quite easily, such as depth of field, bokeh, or even motion blur.

Note that this does not mean that there are no artifacts. If you look in the left half of Figure 4.3, the voxel grid is visible. Such artifacts are for the most part a limitation of the CT and MRI scanners and the limited nature of the medium of the files they output, not so much the path tracer. Even with higher resolution, the voxel grid would still be visible if you zoomed in close enough. One could try methods such as trilinear interpolation to counter this, but even then, the problem would not completely go away.



Figure 4.3: A close up look at a volume. Note the self shadow of the arteries on the organ below. Also note the realistic lighting by the ribs.

4.2 Transfer functions

The transfer function editor we use comes from the UnityVolumeRendering[6] project. While their project had both a one-dimensional and two-dimensional transfer function editor, we currently only support one-dimensional transfer functions in our codebase.

The transfer function can be edited by adding control points which dictate the opacity. Neighboring control points are connected with an ease in-out curve. Both DDA and non DDA transfer functions are supported, though the latter takes longer to render. These transfer functions can be saved to and loaded from disk.

The following page shows a variety of images made by varying the transfer function. These have also been saved alongside the other volumes in the codebase.

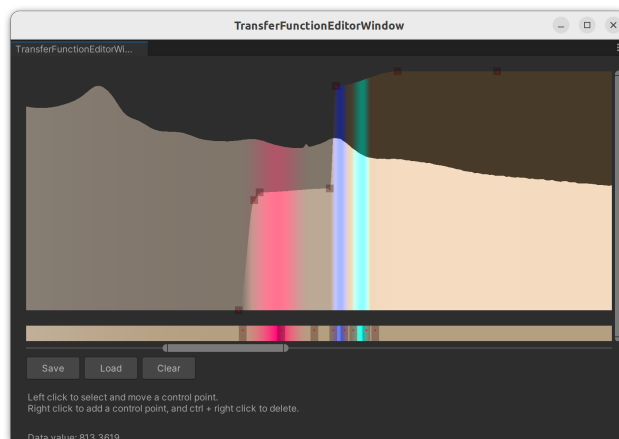


Figure 4.4: The transfer function editor.

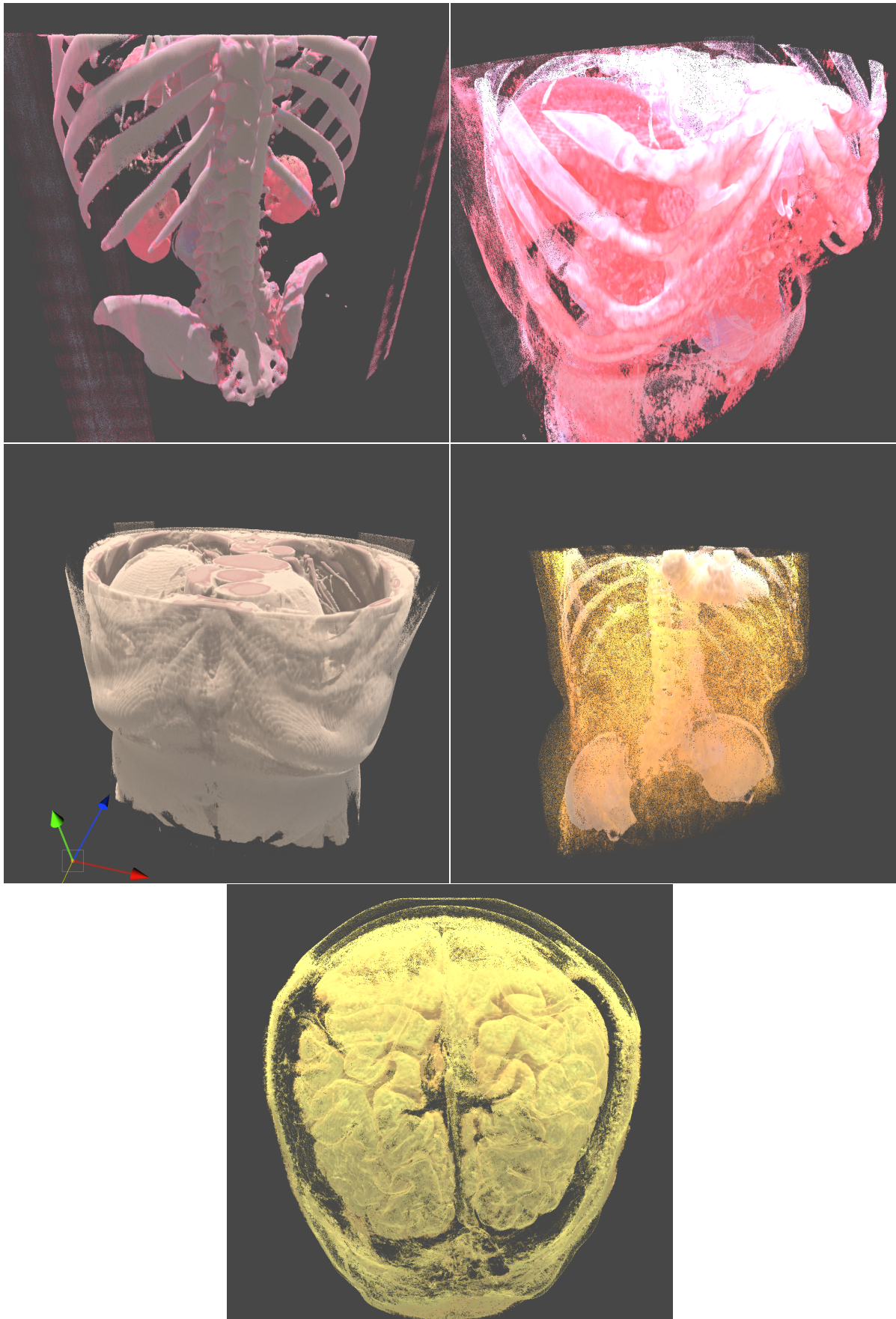

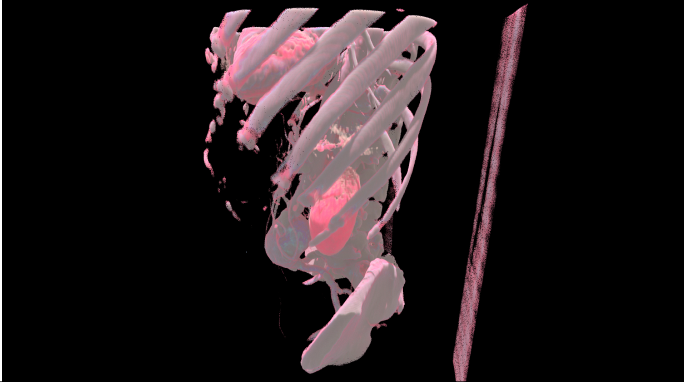
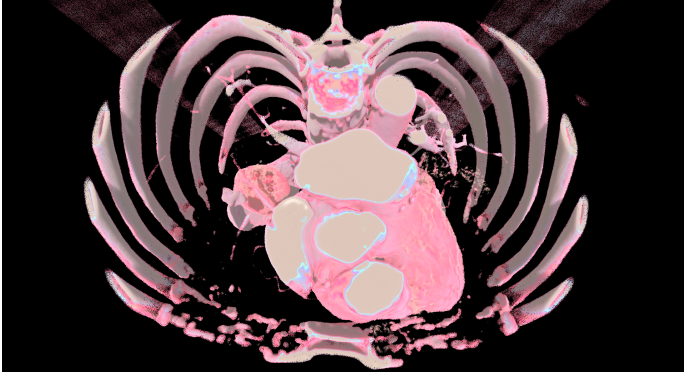


Figure 4.5: A showcase of various different transfer functions.

4.3 Benchmarking

Now we will cover the time it takes to actually render these images. All images in this section were rendered on a 6-core 3.9GHz 11th Gen Intel i5-11600K with an NVIDIA GeForce GTX 1660 SUPER and 16GB RAM running Ubuntu 22.04.5 LTS. The tests were made in the Unity editor (2022.3.62f2) in play mode. The game view window was in full screen, and volume rendering was disabled in the scene view by setting the maximum number of samples for the scene view to 1. The images were rendered for 1000 samples, and then their averages and standard deviations were calculated. The following images were all rendered at 1920x1080p.

Rendered Image	Average Sample Time	Standard Deviation
	152.643ms	3.093ms
	90.321ms	2.929ms
	110.359ms	2.794ms


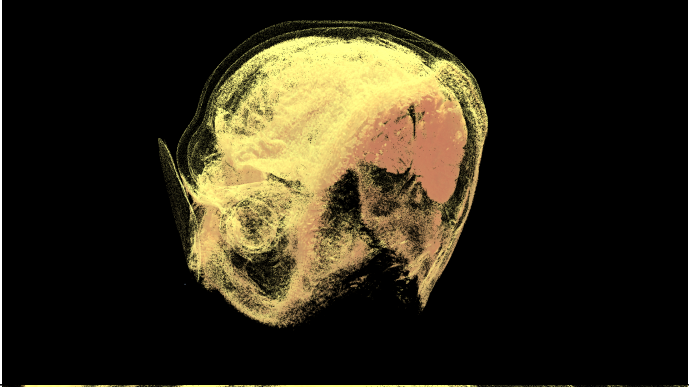
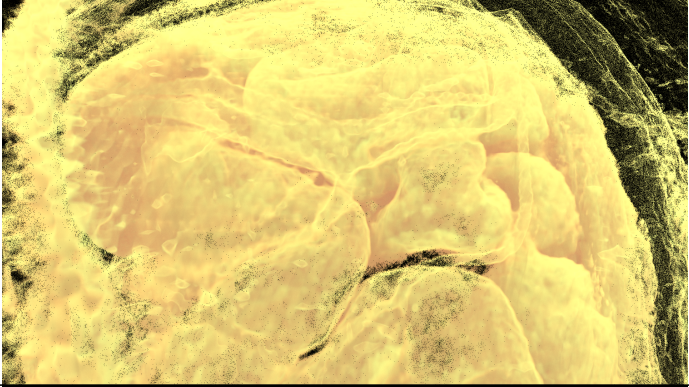
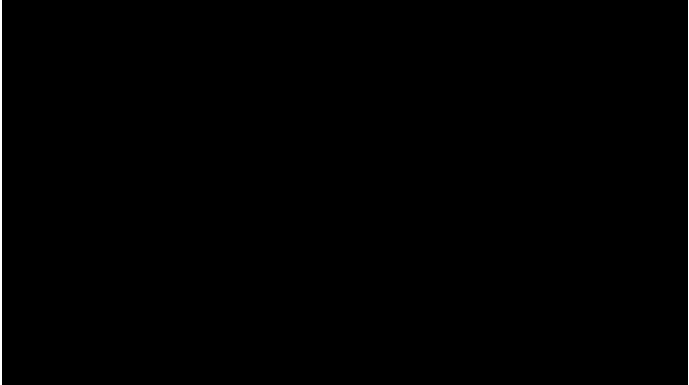
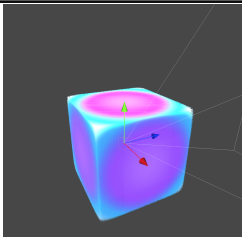
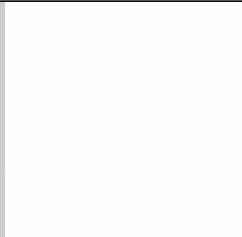
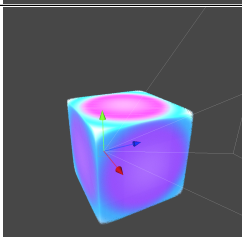
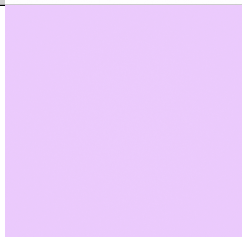
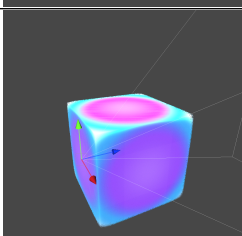
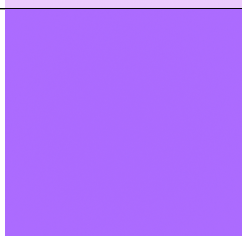
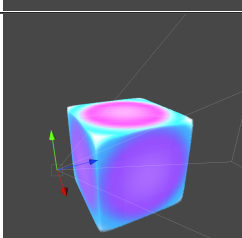
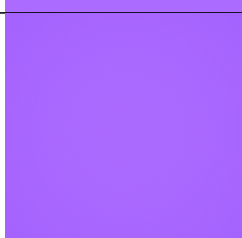
	356.065ms	5.657ms
	79.843ms	2.629ms
	205.593ms	2.590ms
 (Facing away from volume.)	5.568ms	1.737ms

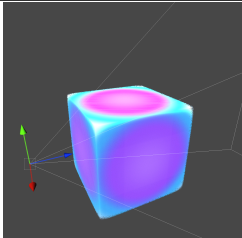

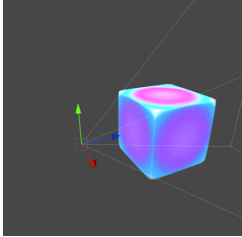

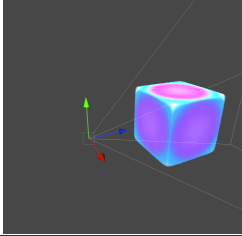
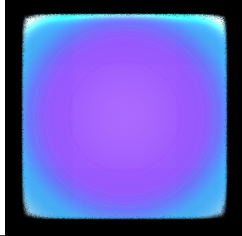
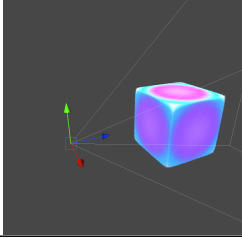
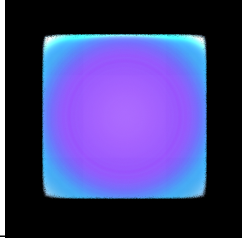
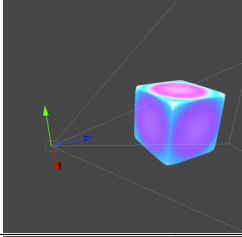
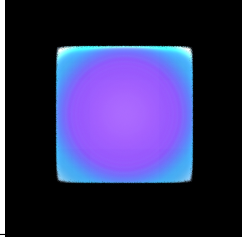
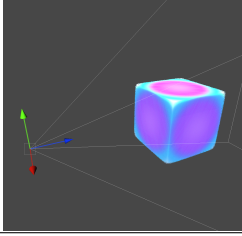
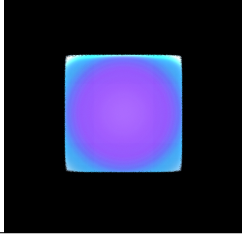
Table 4.1: Benchmarking various images.

Note that the last image in the table is in the same position as the first picture, but the camera is turned around, to act as a control. We also made a control where everything except the camera was disabled, with times of roughly 1.8ms per frame. Though please be aware that this measurement is not using the same system that timed the volumes in the table above, and thus direct comparisons should be avoided.

We noticed that the frame rate decreased dramatically the closer the camera was to the volume. In order to more concretely measure this, we used a test cube volume, with density proportional to the distance from its center. Its center has the maximum Hounsfield density, and its corners have the minimum Hounsfield density. We ran a series of benchmarks with the camera at regular distances from this volume. We also rendered these images at 1000x1000px resolution to simplify calculations. Note that the test volume has a side length of 256 units.

External View	Rendered Image	Distance	Average Sample Time	Standard Deviation
		0	33.709ms	1.786ms
		64	22.864ms	1.904ms
		128	17.015ms	0.965ms
		192	16.991ms	1.010ms

4 Results

		256	16.954ms	1.341ms
		320	18.766ms	1.087ms
		384	15.499ms	1.319ms
		448	11.804ms	1.308ms
		512	9.622ms	1.139ms
		576	8.434ms	1.219ms

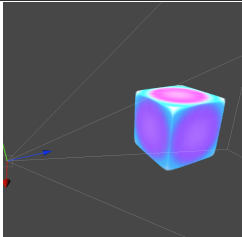
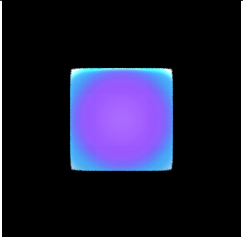
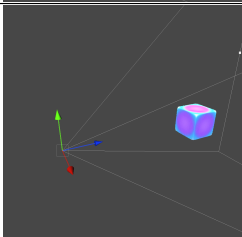
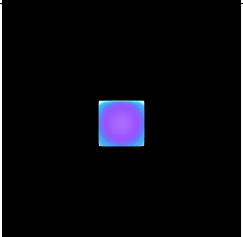
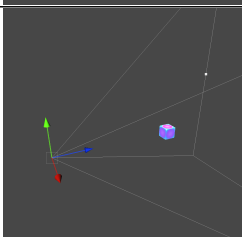
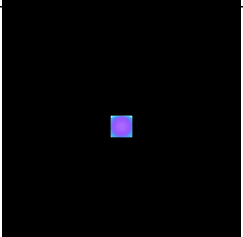
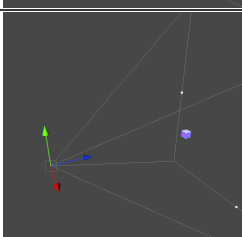
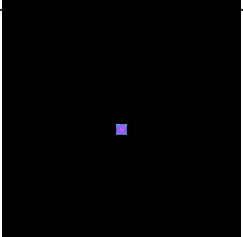
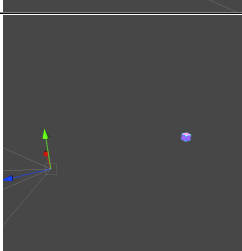
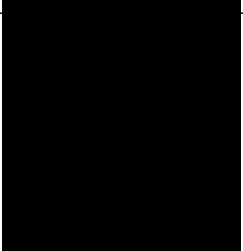
		640	7.591ms	1.175ms
		1280	5.455ms	1.097ms
		2560	4.918ms	0.902ms
		5120	4.837ms	1.126ms
		Facing away from cube.	4.223ms	1.010ms

Table 4.2: Benchmarking regular distances from the test cube.

As with the first table, we have the last image to see how long a sample takes when the volume is not in view by turning the camera around by 180° when 5120 units away. Similarly, we also made another control with 1000×1000 px, where everything except the camera was disabled. This yielded times of roughly 1.6ms per frame, though again, be aware that this measurement is not using the same system that timed the volumes, and thus direct comparisons should be avoided.

To make everything as simple as possible, we also made sure that the camera was facing perfectly forward and had no position offset, except that the z-coordinate was the negative of

the value of the Distance column.

Here we have plotted the values from the above table into a graph. Note that the y-axis does not start at zero.

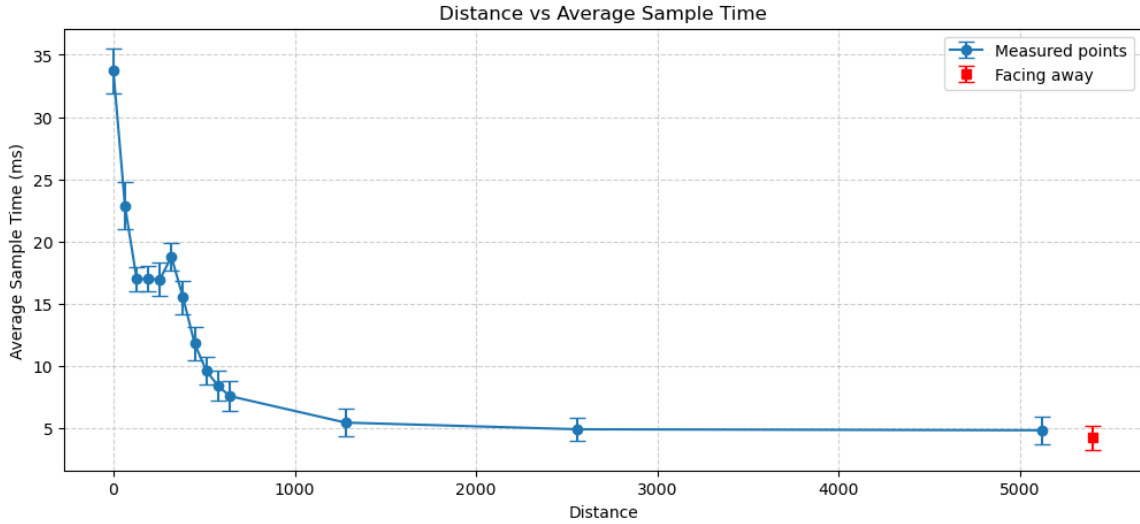


Figure 4.6: The data from Table 4.2 on a plot.

As we can see, the plot seems to show that the rendering time is roughly inversely proportional to the distance from the volume. Of note is the little spike at 320 units away. We expect that is the case because of the FOV of 60° , and the shape of the camera frustum: 320 units away is the first time the entirety of the cube is visible within the camera frustum (See the rendered image of 320 units away in Table 4.2).

5 Discussion

5.1 Summary

We have improved on many parts of integrating the path tracer into Unity. Volumes are now capable of occluding and being occluded by other volumes and GameObjects in the scene. Rendering is supported in the scene view now, not just the game view, and both also work when outside of play mode. Many quality of life features were added when importing volumes. The visual quality of the path tracer was shown, and we have made benchmarks to measure its performance.

As a whole, the path tracer can import DICOM files, allows for the editing of custom transfer functions, and can produce images of high quality that improve in real time. All this is available in the Unity game engine, allowing for easy experimentation and continuation, and opening the doors for builds to be made for a wide variety of platforms.

5.2 Interpretation

Overall, the renderer can produce high quality images, with effects such as self shadows. The rendering speed is almost real time. Ideally it would be just a bit faster, so that the image wouldn't lag behind by a few frames, but it is still fast enough to roughly place the camera in the desired position, and wait as more samples are quickly accumulated and the image becomes crisp.

There are some systems we would have liked to integrate into the path tracer, such as the light system, or reflection probes. There are also bugs we would have liked to fix, but time is unfortunately a limited resource, and the line to stop must be drawn somewhere. The features we did manage to implement are technically impressive in their own right, and will see heavy use from anyone working with this path tracer. We think it is fair to say our primary goal of better integration into Unity has been achieved.

5.3 Limitations

The biggest limitation of the path tracer as a whole, is the rendering speed. When compared to other renderers, this path tracer would likely take last place in speed, and that is because of two reasons:

One is because Unity is not completely focused on rendering. It handles many other functions that a standalone renderer doesn't, such as audio, scenes, game objects, and the rest of the boilerplate that makes it so useful to the developers who use it. Our non-standard

render pipeline likely also contributes to slowdowns here. The second, and bigger reason, is because the algorithm itself is slow. It takes many samples until a clear image is gained, and while looking around, one must be content with a noisy image.

As always there is a tradeoff, and the benefit in this case, is of course the flexibility that a game engine provides, as well as the high quality of the output. Technology is constantly improving, and it is likely that in future years, hardware will be fast enough that this limitation of rendering speed will stop being a problem, and we are left with only positives.

As it stands, this path tracer still needs some work before it likely sees use in the medical field. However, we believe that there are a great number of benefits to being able to offer full volumetric path tracing, and to do so in a way that is extremely flexible. Putting the path tracer in a game engine such as Unity means it will be available cross-platform on countless devices, from desktop computers to consoles, perhaps even VR or your phone. There are likely plenty of other use cases for a powerful volumetric path tracer. Who knows what other use cases people will come up with?

6 Future Work

6.1 Color changing when scaling

There are some unusual effects that occur when you change the scale of a volume. When the scale increases, the volume appears redder, when the scale decreases, the volume appears whiter. Though this appears to be a similar effect to changing the density scale value on a volume, there is no direct correlation between the two. Fixing this issue is important, because the volumes are of a significantly larger scale than most Unity objects, roughly 1 meter = 2000 units, where many Unity systems, for example the physics system, use 1 unit = 1 meter.

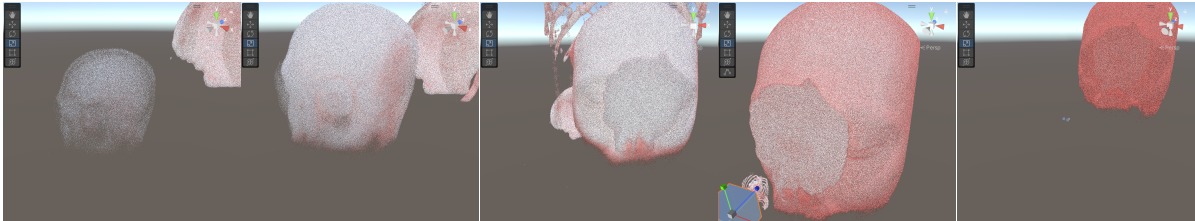


Figure 6.1: The scales from 0.01x to 100x times the original scale.

6.2 White volumes in edit mode

One issue that continues to elude us, is the fact that volumes are completely white when outside of play mode. It does not appear to be related to any of the values of the object, such as density scale. The best hypothesis we have is that some part of the render pipeline doesn't behave the same way in play mode and edit mode, perhaps coroutines, graphics fences, or compute shaders. Also note how the ribcage area in the right half of Figure 6.2 is filled in.

6.3 Depth buffer noise

Another area that could be improved relates to the depth buffer. At the edges of the volumes, there is some slight noise that changes every time a sample is rendered (See Figure 3.3). This is because the starting positions of the camera rays are slightly randomized within their pixel each sample, and the depth buffer is always that of the most recent sample. If the depth buffer would accumulate the distances over multiple samples and correctly average them, similar to what is currently done with the color buffer, then this issue would likely be resolved. It is unclear to us, however, how to correctly include ray misses into this averaging formula.

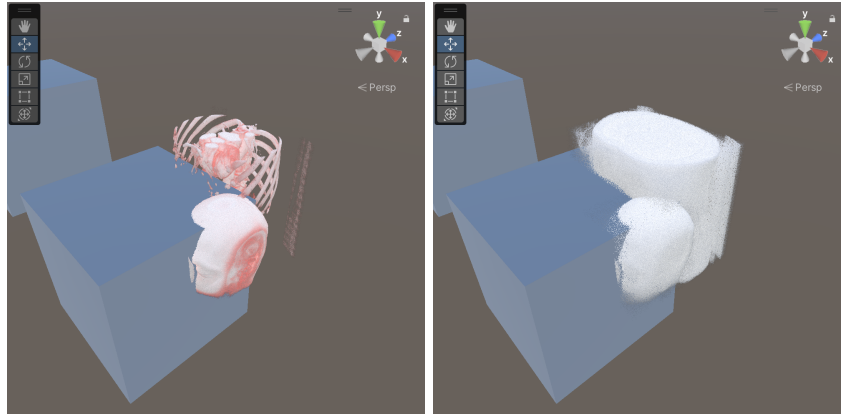


Figure 6.2: Left: Volumes while in play mode. Right: Volumes while in edit mode.

6.4 Flickering game view

A minor issue is also found in the game view, which flickers slightly when outside of play mode. We suspect that this happens because the screen is being updated too often, or is being updated at a time that is unexpected to the game engine somehow.



Figure 6.3: The game view during a flicker. Note that the rest of the scene is not visible.

6.5 More issues

Even further issues that need tackling can be found in the Future Work section of Simon Giese's paper[2] upon which we built upon. To summarize, these include: Fixing rendering issues of rotated volumes where axis-aligned bounding boxes (AABBs) cannot be used, adding UI to make everything accessible outside of the editor, experimenting with using the path tracer to render the rest of the scene, using reflection probes to get accurate lighting from the environment, integrating Unity's Light components into the shader, and applying denoisers for better or faster render passes.

7 Conclusion

In conclusion, we built upon previous work to better integrate the path tracer into Unity. We added features such as occlusion, depth sorting, and support for multiple volumes. We added support for making the path tracer work in the both the scene view, as well as when outside of play mode. We showed the various visual capabilities of the path tracer and ran benchmarks to see how long rendering takes. We fixed various bugs and added some quality-of-life features, and provided a valuable stepping stone for future work on this project.

8 AI Usage

Over the course of writing this document, LanguageTool has been used in order to correct various minor spelling and grammar mistakes. DeepL was used to translate the abstract and title into German.

ChatGPT and Claude have also been used sparingly while working on the codebase, and for help with some \LaTeX commands. ChatGPT also made the code for the plot in Figure 4.6.

Otherwise, **no** other artificial intelligence tools such as QuillBot, Grammarly, GitHub Copilot, DeepSeek, or similar have been used.

Bibliography

- [1] Dorin Comaniciu et al. “Shaping the future through innovations: From medical imaging to precision medicine”. In: *Medical Image Analysis* 33 (Oct. 2016), pp. 19–26. ISSN: 13618415. DOI: 10.1016/j.media.2016.06.016. URL: <https://linkinghub.elsevier.com/retrieve/pii/S1361841516300962> (visited on 06/01/2025).
- [2] Simon Giesse. “Volumetric Path Tracing inside the Unity Game Engine”. In: ().
- [3] Nikolai Hofmann and Alex Evans. “Efficient Unbiased Volume Path Tracing on the GPU”. In: *Ray Tracing Gems II: Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX*. Ed. by Adam Marrs, Peter Shirley, and Ingo Wald. Berkeley, CA: Apress, 2021, pp. 699–711. ISBN: 978-1-4842-7185-8. DOI: 10.1007/978-1-4842-7185-8_43. URL: https://doi.org/10.1007/978-1-4842-7185-8_43 (visited on 06/01/2025).
- [4] Nikolai Hofmann et al. “Neural Denoising for Path Tracing of Medical Volumetric Data”. In: *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 3.2 (Aug. 26, 2020), pp. 1–18. ISSN: 2577-6193. DOI: 10.1145/3406181. URL: <https://dl.acm.org/doi/10.1145/3406181> (visited on 05/27/2025).
- [5] Thomas Kroes, Frits H. Post, and Charl P. Botha. “Exposure Render: An Interactive Photo-Realistic Volume Rendering Framework”. In: *PLoS ONE* 7.7 (July 2, 2012). Ed. by Xi-Nian Zuo, e38586. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0038586. URL: <https://dx.plos.org/10.1371/journal.pone.0038586> (visited on 06/01/2025).
- [6] Matias Lavik. *mlavik1/UnityVolumeRendering*. original-date: 2019-02-11T16:16:35Z. May 30, 2025. URL: <https://github.com/mlavik1/UnityVolumeRendering> (visited on 06/01/2025).
- [7] Patric Ljung et al. “State of the Art in Transfer Functions for Direct Volume Rendering”. In: *Computer Graphics Forum* 35.3 (June 2016), pp. 669–691. ISSN: 0167-7055, 1467-8659. DOI: 10.1111/cgf.12934. URL: <https://onlinelibrary.wiley.com/doi/10.1111/cgf.12934> (visited on 06/01/2025).
- [8] Payton. *Pjbomb2/TrueTrace-Unity-Pathtracer*. original-date: 2022-02-20T06:33:01Z. June 2, 2025. URL: <https://github.com/Pjbomb2/TrueTrace-Unity-Pathtracer> (visited on 06/02/2025).
- [9] Justin Sutherland et al. “Applying Modern Virtual and Augmented Reality Technologies to Medical Images and Models”. In: *Journal of Digital Imaging* 32.1 (Feb. 2019), pp. 38–53. ISSN: 0897-1889. DOI: 10.1007/s10278-018-0122-7. URL: <https://pmc.ncbi.nlm.nih.gov/articles/PMC6382635/> (visited on 11/03/2025).