Visualising Ray Marching in 3D

Anton Bredenbals, Supervisors: Jiří Kosinka and Steffen Frey

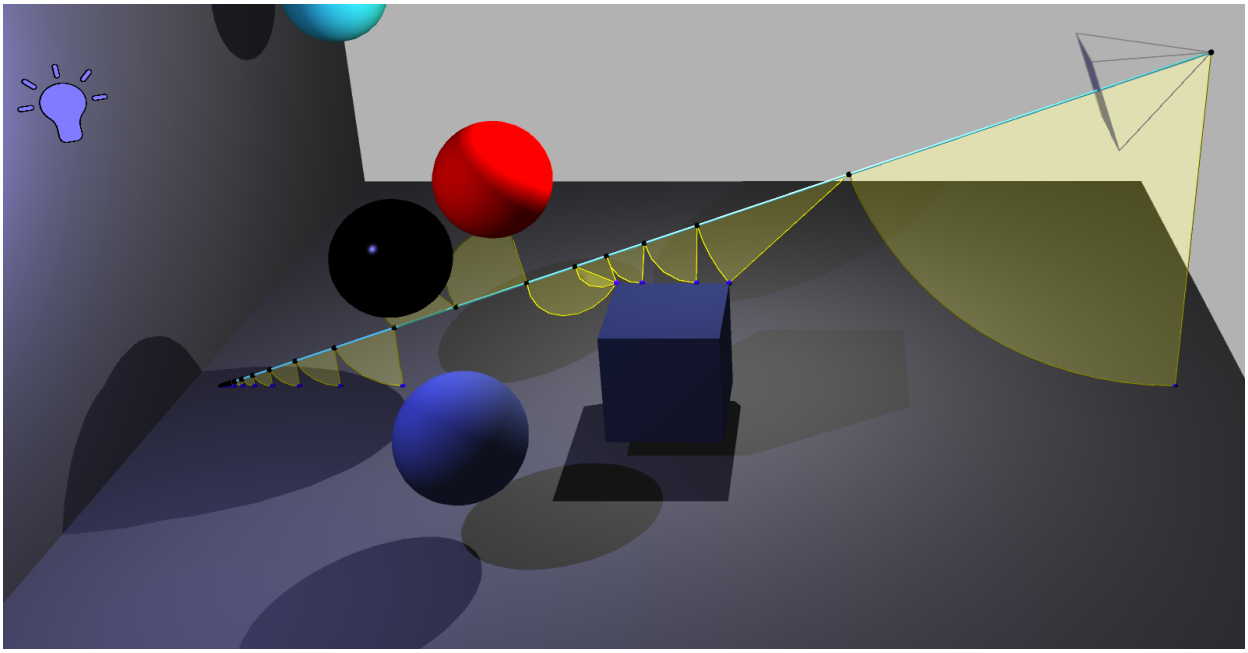


Fig. 1: Visualisation of ray marching in Virtual Ray Tracer.

Abstract—Ray marching is a commonly used technique in rendering digital scenes. Despite its similarities to ray tracing and its easier implementation in practice, it is not commonly taught in basic computer graphics courses. This may also be due to it being more challenging to be properly visualised in 3D. Currently, there are existing applications which greatly visualise ray marching in 2D; but in 3D, especially with regards to education, there are no widely available applications. This is why, in this report, I discuss the development of an extension to the open-source "Virtual Ray Tracer" application, which hosts a number of levels from basic to more advanced ray tracing visualised in 3D. Now it will be supplemented by a basic and an advanced ray marching level, allowing for real-time interaction with the ray marching process. The user has multiple different options for visualisation, which can be combined as it suits them. The "Virtual Ray Tracer" platform already allows the user to interact with the simulation by editing the objects and their materials in various ways and by editing the ray and camera settings. The goal is to teach the user the differences and similarities between ray tracing and ray marching in a fun and interactive way.

Index Terms—ray marching, rendering, education, interactive learning

1 Introduction

Ray marching is an increasingly important rendering technique in computer graphics. Rendering is a wide field with different requirements from scientific visualisation to real time applications. Since one of its first mentions in 1989 by K. Perlin [3], ray marching has seen a few improvements like the inclusion of signed distance functions (SDFs). These allow to dynamically use the biggest step sizes possible, instead of constant step sizes. Together with the rising computing power over the decades this led to modern rendering being able to include ray marching even for real time applications [11].

Ray marching is intertwined and closely related to ray tracing as they are both image order rendering techniques. To determine the colour of each pixel, a ray is cast from the camera through each of the pixels into the scene. These rays then intersect with objects in the scene and get reflected, refracted and more rays are cast in directions of the light source to get all relevant components of colour for that specific pixel. This procedure is fundamentally the same in both techniques. The main difference is in the way those intersections are computed. While ray tracing generally uses implicit object equations to directly compute an intersection, ray marching follows an iterative approach build on either set step sizes, or as we will use in this paper, step sizes computed by the SDFs of the objects in the scene. This makes ray marching more robust for different object shapes, as implicit equations do not exist for all shapes of objects. However, ray tracing is usually faster due to its non-iterative nature.

Because of its rising popularity, ray tracing is a staple in computer graphics courses at most universities. Despite its similarities, ray marching is less commonly taught. This in turn means that online or especially learning resources for ray marching are scarce. 3D visualisations of the iterative process are rare.

- *Anton Bredenbals is a student at the University of Groningen. E-mail: a.bredenbals@student.rug.nl. Student number: s4934059.*
- *Jiří Kosinka and Steffen Frey are with the SVCG Group at the University of Groningen. E-mail: j.kosinka—s.d.frey@rug.nl.*

Due to this, I decided to look into the virtual learning environment "Virtual Ray Tracer" (VRT) [7,9,10] and add ray marching functionality to it. VRT is a 3D learning application created in the game engine Unity[1] [5]. Before, VRT consisted of different levels from basic to advanced ray tracing that offered visualisation of a scene. A scene includes a camera from which rays are visually cast into the scene where they bounce off of the objects they hit. Once a base of understanding for ray tracing is available, making the student understand the difference between the methods would only be another rather small step.

My contribution is a level dedicated to ray marching. This includes most importantly new visuals and animations to make the iterative process of ray marching as intuitive as possible.

The coming sections include the following: In the next section I introduce relevant related work. In section 3 I elaborate on the concept behind the visualization and the individual elements, after introducing ray marching in more detail. Section 4 includes relevant points about the practical implementation of the concepts. In section 5 I discuss the results of developing the ray marching level. Section 6 includes ideas for future work and in section 7 I conclude my work.

2 Related Work

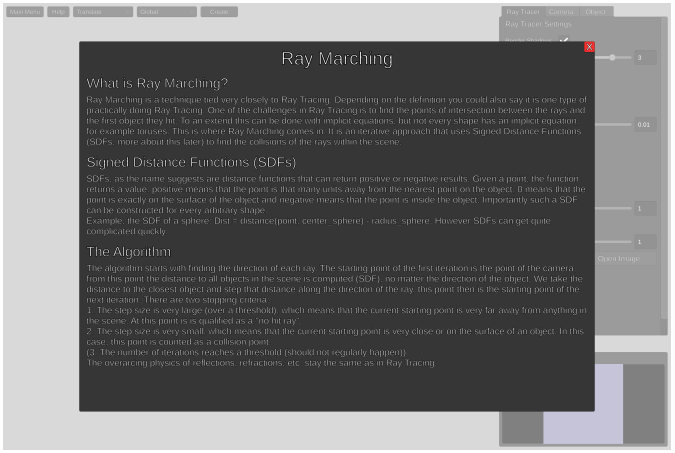
As mentioned previously, my work is based on VRT developed by van Wezel and Verschoore de la Houssaije [7,9,10]. VRT is an open source interactive learning application focused on ray tracing [8]. The user can choose from a variety of levels, each expanding on the knowledge acquired in the previous levels.

Each level is a 3D scene consisting of the virtual "raytracer camera" including a virtual screen, one or more light sources and the objects in the scene. The first levels restrict the user in freedom by not allowing creation of new objects and certain variables such as material properties cannot be edited. This is done so the user can focus on the most important underlying features. Later levels allow for more and more customization and there is a sandbox level that has all option unlocked.

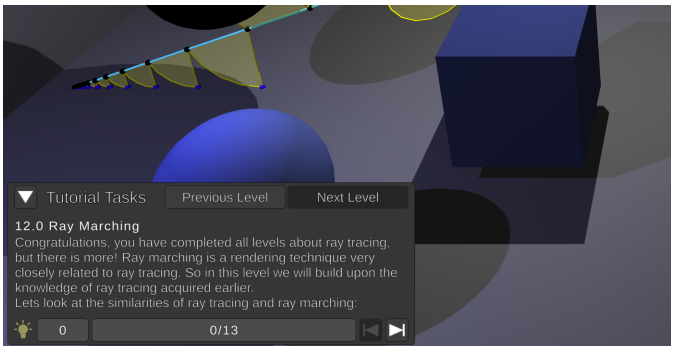
The user can interact with the scene in different ways: Objects in the scene can be rotated, translated and scaled freely and their material is editable in colour and other modifiers such as coefficients for specular, diffuse, etc. The ray tracer can be configured in the depth of recursion and shadows can be turned off or on. For the camera, the user can, amongst others, decide how many pixels the virtual screen has, whether and how the rays are animated, and rotate and translate the camera. Further, a higher resolution image can be rendered of the current configuration and point of view of the virtual camera, however this is not possible in real time.

VRT is implemented using the game engine Unity [5] which comes with the default scripting language C#. For me, the decision to use Unity was straightforward, as van Wezel and Verschoore de la Houssaije created the project in Unity. Their choice was mainly between Unity and Unreal, which are both very well documented but with Unity being a bit more beginner friendly, using C# instead of C++ and requiring less disc space to run. Unity is a widely known and used 3D game engine. As such it offers a lot of functionality and infrastructure that is of use in such a project. Creating and designing scenes is very easy and straightforward, Unity delivers meshes for basic shapes like spheres, cuboids and cylinders as well. Unity supports C# scripts and allows intuitive handling for those as the user can apply the scripts directly to object instances in the scene. For this, Unity provides the base class *MonoBehaviour* that can be extended. In the package, Unity also provides a large set of useful functions for scripting from *GetConponent<>* to *closestPoint*.

Van Wezel [7] and Verschoore de la Houssaije [9] conducted a user study and lowest scoring was the "user interface and controls". Based on that, another project by Peter Jan Blok [1] is ongoing in parallel, adding updated gamification and improvements to the interface and user experience in general. In detail, this means replacing the introduction text of each level, which is currently quite unintuitive as it is one big window with all information compiled. As can be seen in Figure 2, this is replaced by a step by step tutorialisation that updates semi



(a)



(b)

Fig. 2: The old (a) and new (b) tutorialisation of VRT.

automatically. The user is required to complete the main steps of each level's tutorial before the following level can be accessed. The first level is an introduction to the controls of the program, a "cheat mode" is available in the options to circumvent this restriction. In the same step an achievement system that rewards the player with badges, for example after playing for set amount of time, was introduced and the render image view now has a loading bar.

Another parallel project by Jesper van der Zwaag [6] extends VRT to also include distributed ray tracing, sometimes also called stochastic ray tracing. It introduces different kinds of light sources. The basic point light are light sources that emit light from a single point equally in all directions; this was the only light source available before. Spot lights emit light from a single point as well but have a direction similar to flash lights. The last type of light sources are area lights which emit light from a whole area. This is essential for distributed ray tracing as it allows for the more realistic smooth shadow transitions, the so called penumbra. Van der Zwaag also introduced more lighting options to make the light more realistic.

The last current addition to VRT, by Roan Rosema [4], is in the form of a web and a mobile version of VRT. The web version is quite easy to set up, as the project is created in Unity. Unity has built-in functionality to build the project for WebGL, which can directly be supplied to GitHub pages. The mobile version is currently only available in the Google Play Store[2] and will probably come to iOS in the future. For the mobile version, all the controls and the interface have been overhauled.

[1] https://unity.com/

[2] https://play.google.com/store/apps/details?id=com.RUG.VirtualRayTracer

3 CONCEPT

In this section I introduce the ideas of how to visualise ray marching and explain the goal of each visualisation. For this I first introduce ray marching in more detail. Ray marching is a rendering technique and as such it renders a 2D image, given a 3D scene and a camera position in it. Ray marching specifically starts off by computing the colour each pixel should display separately. For this a ray is cast through each pixel, starting from the camera position; this uniquely determines their direction. For each ray, the starting point of the first iteration is the point of the camera. For each iteration, the SDF of each object with respect to the starting point is evaluated. The smallest of those distances is then taken and stepped along the direction of the ray to find the starting point of the next iteration.

The algorithm has different break conditions, namely: If the distance to travel in one iteration is larger than a certain threshold, 99 in this case, the starting point is far enough away from every object in the scene to declare that ray a "no hit ray". This threshold should be bigger than any reasonable distance within the scene. In this case it is multiple times larger than the whole occupied scene. If the distance is smaller than a very small value ε, 0.001 in this case, the current starting point is counted as on the surface of the object and thus as an intersection. After a high number of iterations, in this case 250, the algorithm should also stop as a safety measure against too long looping times or even (close to) infinite loops. This case should not come up regularly, but the closer a ray passes by one or multiple objects, the more iterations it needs as the step sizes decrease with that distance. Generally it is a good practice to have such a threshold in place for potentially infinite loops, in case of unfound bugs or glitches.

If an intersection is found, the algorithm continues in the same way as ray tracing: The material of the hit object will be considered to determine transparency, the diffuse and specular coefficients, etc. The normal of the intersected triangle on the hit object will be used to compute the reflecting and refracting ray if necessary. More rays are cast in the directions of each of the light sources. For each of these rays the procedure lined out above is repeated. Light rays have an additional stopping criterion, if the light source is reached, the algorithm stops and determines that the point hit before does not lay in shadow with respect to that source.

To visualise ray marching and especially its iterative nature, the visualisation consists of four separate parts. The first option is only available in the animated case, where rays move further into the scene over time. This is an expanding sphere around the starting point of the current iteration. The sphere will disappear as soon as a new iteration begins, to reduce visual clutter. This concept can be seen in many 2D ray marching visualisations, where it appears as circles around the starting points, as seen in Figure 3. But it is more difficult to make it appealing in 3D. It simulates that the distance to the nearest object equals the size of the circle/sphere and thus, the circle/sphere will intersect with the nearest object and the point on the ray where the new iteration begins. The sphere can get quite big and should thus be transparent to not occlude the rest of the scene.

Two simpler visualisations indicate the starting point and the nearest object of each iteration with small pointers, spheres in this case. This should help the user identify the iterations on the ray itself and the nearest object respectively. To group the starting point with the nearest object another option is available to draw a line, a thin ray, between each starting point and the corresponding nearest point on the nearest object.

Lastly, an option for arcs is available that span a part of a circle between the nearest object and the starting point of the next iteration. This is meant to help the user understand the connection between these two: The next iteration's starting point is exactly as far away on the ray as the nearest object's distance is.

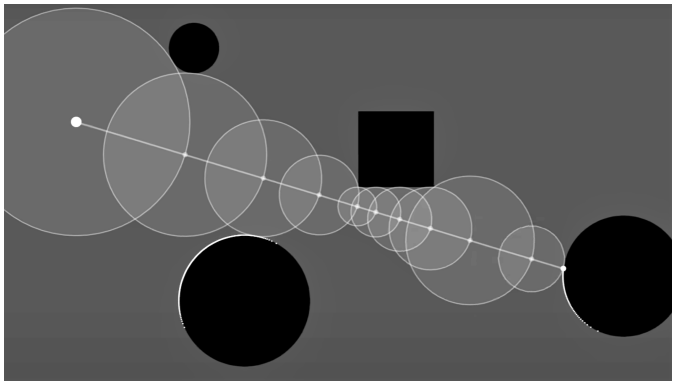


Fig. 3: A typical 2D visualisation of the ray marching process. Taken from `https://youtu.be/Cp5WWtMoeKg`

4 REALISATION

Next, I elaborate the important choices made during the implementation of the concepts, starting with general decisions made and then getting into details about the concrete different visualisation options.

I made the decision for the code base that most of my ray marching specific additions should have their own associated classes to not slow down the application in the other levels with loading unnecessarily big classes. New classes were created for some of the visualisations, mainly *sphere objects* for the spherical visualisations and *meshObjects* for the transparent meshes filling the arcs. The general architecture here was kept similar to the *rayObjects* with *ObjectPools* speeding up the initialisation of a large amount of objects. In some cases it made sense to create derived classes from their ray tracing focused parents. Examples are *RayMarchingManager* and *UnityRayMarcher*. Both of these classes have a central role in the program. The *UnityRayMarcher* extends the *UnityRayTracer* and overrides the ray tracing functions with ray marching specific ones as well as adding the *RayMarch* function itself. The *RayMarchingManager* overrides the ray tracing specific visualisation functions with those that also visualise the extra ray marching information. In some cases I decided to add ray marching specific methods or extensions to the base classes, if they were minor. For example, *RTMesh* now has a *distanceToPoint* function which implements the SDFs of the respective shapes and a *normalRM* function to find the normal of the intersection.

Ray marching requires explicit SDFs for every type of object. This is not fully realised in VRT yet, only non deformed spheres and non rotated cuboids have a functioning SDF implemented. However, it is still good for the user to be able to choose from more kinds of objects in the level, so for the other options I used the unity function *meshCollider.ClosestPoint* to simulate the SDF. The *normalRM* method returns the normal of the surface that is closest to the given point. The normals are calculated in different ways, for spheres the normal is smooth by default as it is simply taken as the direction from the sphere center to the given point, which results in smooth/Phong shading. However, for the other object types the normal of the hit triangle is used which results in flat shading. The differences can be seen in Figure 4.

I also decided to change the basic layout of the 3D scene to focus on the differences from ray marching to ray tracing. For this, I decided to reduce the default pixel number in the level from 9 (3x3) to 1, because with all the different visualisations on top of the ray animations, the scene gets crowded by visualisations quickly which in turn will have a negative effect on understanding. Having one ray as an example is easier to understand as the visualisations for multiple rays would interfere with each other to reduce visual clarity. Having too many objects for visualisation in the scene also reduces the performance significantly. Further, the concept of the screen with many pixels and each having their respective ray computations is touched upon in previous levels a lot and should be clear by this level. However the level still allows the user to edit the screen resolution to their liking.

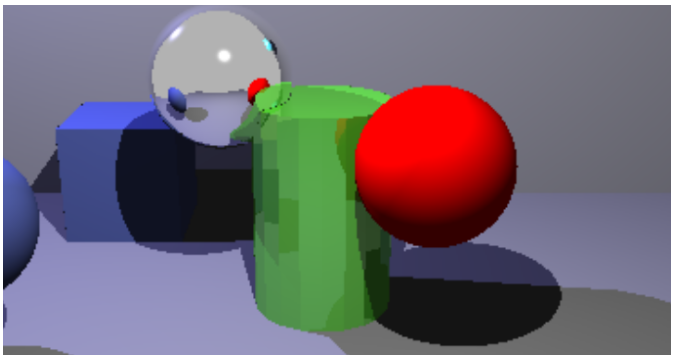


Fig. 4: Flat shading on the cylinder vs smooth shading on the spheres.

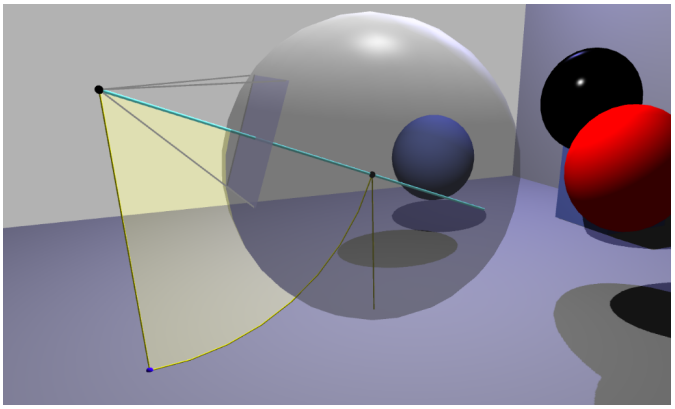


Fig. 5: The animated visualisation with expanding spheres.

For the expanding spheres visualisation, I decided to only make it work in the animated setting, as having too many of these spheres in the scene at the same time detriments from the understanding as they are occluding big parts of the scene. Having them only available in the animated mode allows to only show the sphere that corresponds to the iteration that is currently in the process of being drawn. This can be seen in Figure 5. The sphere is transparent as bigger iterations steps result in very large spheres.

For the iteration indicators (indicator dots) on the ray and the nearest object I decided to use very small spheres; black ones on the ray and blue ones on the nearest objects.

The indicator lines between each iteration's starting point and its respective nearest object are yellow, the same colour and thickness as the arc indicating the same distances between the nearest object and the starting point of the next iteration. The arcs are built out of many small ray segments. How many ray segments are needed is simply determined by recursively splitting the direct distance between the two points the arc should connect until each segment is shorter than a set threshold. To reduce the impact that very big arcs, for example from no hit rays, would have on the performance, the maximum number of pieces an arc can have is limited to 100. Additionally, the area enclosed by the indicator lines and the arcs is filled with a transparent plane, as can be seen in Figure 6.

Finally I also merged the features implemented by van der Zwaag [6] and Blok [1] into my version of VRT. For the ray marching level, this is unfortunately not as straightforward as for most of the other levels. This is due to the decision to have mostly separate classes for ray marching, which means most new features need to be edited in separately for the ray marching level. That said, the additions that are compatible with the ray marching level are working there as well, for example the loading bar for the render image page, some extra options options in the ray tracer menu, including the *Fly to virtual camera* button. And the tutorialisation now also works with the new tutorial boxes.

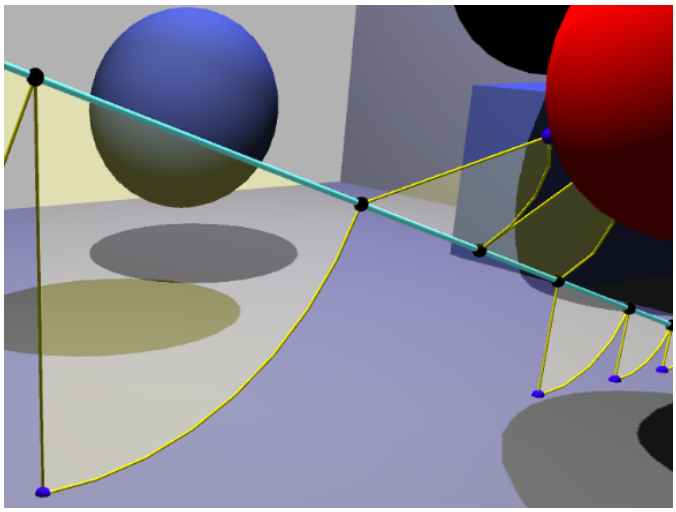


Fig. 6: The visualisation with indicator spheres, lines and arcs turned on.

5 Results and Evaluation

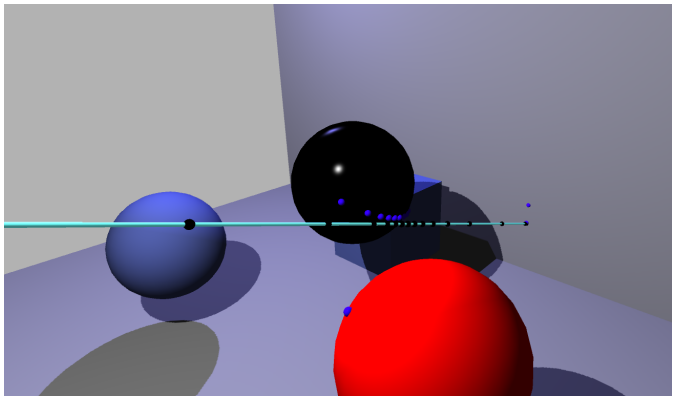
Taking into account the points mentioned before, I developed the application capable of the visualisations that can be seen individually in Figure 7. The first idea for visualisation were the expanding spheres that can be seen in Figure 7d. The idea was based on the circles used in most 2D visualisations like in Figure 3. Turning this into a 3D visualisation that does not take over the whole scene was a difficult task. Making the spheres transparent is a must and further I decided to only show one sphere at a time. This came with the decision to have this option only available in the animated case. I think this option adds a lot to the visualisation, as it shows the progress of ray marching over the iterations.

A valid point of critique here is that the constantly forward moving ray and expanding sphere is not how ray marching functions mathematically, as it computes each step and then steps forward. However, this is the same for ray tracing, where the intersection with an object is computed immediately. Still, many times in education, ray tracing is explained with the help of physical rays and those move through space with a constant speed and with that in mind, I think that the choice to have animated rays moving though the scene was a good decision for ray tracing. The logical expansion for ray marching are the growing spheres.

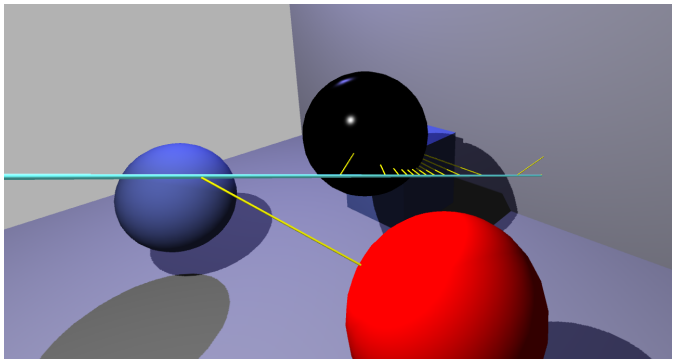
This leads to the static visualisations included: First, I implemented the indicator dots and lines, seen in Figure 7a and 7b respectively. These do a good job at showing the relevant steps within an iteration of the ray marching algorithm. However the shortcoming was the connection between the iterations.

Initially I did not have the indicator arcs seen in Figure 7c planned to be included. However, only looking at the first two static visualisation techniques, I felt like there was a disconnect between where one iteration ends and the next one starts. The idea that the distance to the nearest object is exactly the distance to the start of the next iteration was not clear. The indicator arcs were introduced for exactly that purpose and through that the iterative nature of the algorithm becomes clear.

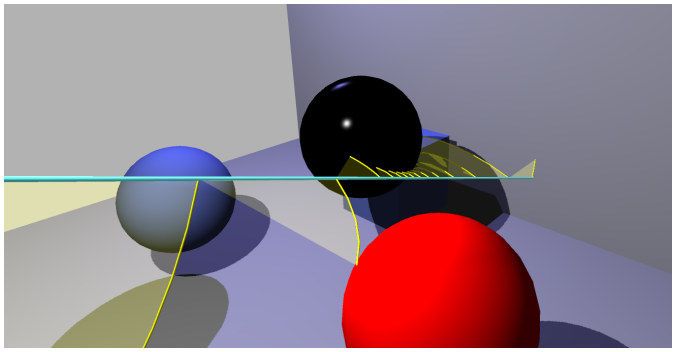
While currently only some shapes have an explicit SDF implemented in VRT, Unity delivers a workaround, *meshCollider.ClosestPoint(x)*, which returns the closest point on the mesh in the scene given a starting point x. However, since this function only works for convex shapes, all of the shapes in the scene must be converted to convex meshes, which can lead to unwanted behaviour elsewhere. With this in mind, I know that this is obviously not the best solution but having all shapes available in the simulation is definitely an upside worth it.



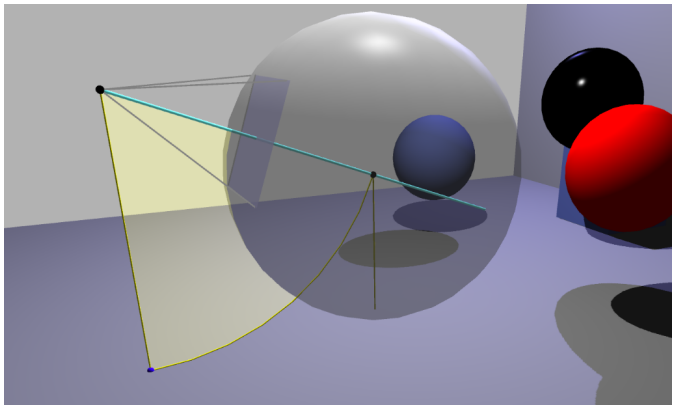
(a) Indicator dots, the black ones on the ray, represent the starting point of each iteration of ray marching, the blue dots the nearest point on any mesh in the scene from each starting point.



(b) The indicator lines connect the starting points and its corresponding nearest point on any mesh in the scene.



(c) The indicator arcs connect the nearest point on any mesh of the current iteration with the starting point of the next iteration, visualising that these have the same distance from the current starting point.



(d) The expanding spheres are only available in the animated mode and grow as the ray does. Only the sphere of the currently animated iteration is shown to reduce visual clutter.

Fig. 7: The four separate visualisations specific to ray marching.

With this, the biggest difficulties of visualising ray marching in 3D have been tackled. I created a coherent 3D visualisation of ray marching and a level around it that allows the user to concentrate on things that make ray marching different from ray tracing. Reducing the size of the virtual screen and hiding the reflection, refraction and light rays all contribute to making the scene less crowded and let the user concentrate on the important parts. This was necessary as the new visualisation techniques add a lot of objects to the scene. Having a higher number of rays shown at the same time would heavily reduce the visual clarity and in turn the user experience. Further, the number of objects present in the scene for visualisation purposes gets to levels that strongly reduce the performance of the application. In some cases the frames per second (FPS) are reduced to single digit numbers even on better hardware.

The gamification is a great addition which I happily added to my level as well. The result can be seen in Figure 2b. With this, the user is not hit by a wall of text at once in the beginning of the level and all the different options can be introduced one by one. This hopefully makes interacting with the program more enjoyable, as this was one of the demanded points of the original user study. Further, I also made a WebGL version of my project available under `https://abredenbals.github.io/Virtual-Ray-Tracer/` .

With this level in VRT, the differences between ray tracing and ray marching are shown well and with that the user gets a good basis of understanding of ray marching after learning about ray tracing already. This should make for a great addition to VRT as it builds on its solid base and adds more value to the application. With more concepts added to VRT, it becomes more and more enticing for lecturers in computer graphics to learn about VRT and incorporate it into their course. The other way around, this also makes the access to learning material about ray marching easier and thus will give more students the opportunity to learn about ray marching in the first place.

6 Future Work

With this project, a base was laid out for future projects to expand upon in different ways. I now outline a few ideas of what can be done. First, the base could be made more sturdy by introducing SDFs for more shapes to get rid of the Unity built-in workarounds. This can go together with performance optimisation, making the ray marching process itself more efficient but also making the visualisation more efficient by reducing the number of simple objects in the scene or finding more efficient ways to render them.

Further, one could expand on the number of levels about ray marching, showing off advanced rendering techniques and applications for ray marching. Ideas for this are SDFs for intersections of shapes, for example only rendering the shape of the intersection, if a sphere and cube overlap in this area. The smooth blending of nearby objects is another option that can be implemented. Both of these require changing or creating unique min/max functions that replace the usual min function with which ray marching determines the "jump" distance of the iteration. Lague [2] has created prototypes and open source code for this already. Another often shown use of ray marching is the creation of various kinds of fractal animations. Examples of this are animations from Auctux[3] and Lague[4] [2]. Some of these ideas might need real time capabilities, which are probably out of reach or alternatively video rendering capabilities to really shine.

Further, a new user study would really help to pick future routes for improvement. This user study should not be focused on a specific part but rather span the whole project and all additions made since the original study. Maybe this could be done after using the application in a computer graphics course. This would allow for a decently sized sample group of computer science students at least. Further, also interested people outside of the computer science studies should be asked.

[3] `https://youtu.be/SdNb7-I1TtA`
[4] `https://youtu.be/Cp5WWtMoeKg`

6.1 Known Bugs

Here is a list of known bugs that do not heavily impact the use of the application but could not be fixed in the time scope of the project:

- On some machines, the WebGL version can crash when rotating for periods longer than 5 seconds.
- The "Render Image" button in the ray marching level renders artifacts (black pixels), if the ray hits a cuboid orthogonal. Sometimes there are further artifacts in a (partial) circles around this. This is not reproducible in the lower resolution render preview.
- The *ObjectPool* classes used for the ray marching level have a memory leak associated with them, as excessive unused objects are not destroyed but stay in memory. This bug was found by Jesper van der Zwaag. A fix for the *rayObjectPool* was implemented by van der Zwaag as well, but the ray marching level as it currently stands cannot benefit from that fix without some tuning, as the subclasses for ray marching all need to be reviewed. The same fix could then be applied for the *sphereObjectPool* and the *arcMeshObjectPool* in a similar way.

7 Conclusion

The aim of the project was to integrate a suitable visualisation for ray marching into Virtual Ray Tracer. The main problem was expected to be the proper visualisation of the iterative nature. To properly visualise ray marching, I decided on four separate visualisations: indicator dots, indicator lines, indicator arcs and the expanding spheres. Together, these are able to convey the iterative nature of ray marching and visualise how the distance to the objects in the scene determines the step size. Having separate toggles for all four techniques allows for customisation for the users experience. I think the result is a clear visualisation for ray marching which focuses on the differences to ray tracing and the iterative nature. The later introduction of the indicator arcs was a very helpful addition especially to visualise the iterations. I also incorporated features from the ongoing parallel projects that extend VRT in different ways, many of which are aimed to improve the user experience.

Adding ray marching as another rendering technique to VRT allowed me to focus on the differences to ray tracing. Further, I could build the ray marching level into a solid base application that was already in place. Connecting ray marching this close to ray tracing hopefully also opens the doors for ray marching to be taught more regularly in computer graphics courses. This broader education would lead to a broader knowledge in the field of rendering and in the future bring more nuanced advancements to the field.

Acknowledgments

I wish to thank my supervisors Jiří Kosinka and Steffen Frey for their support. A special thanks to Chris van Wezel and Willard Verschoore de la Houssaije for developing Virtual Ray Tracer and making it publicly available.

References

[1] P. J. T. Blok. Unpublished manuscript. Bachelor's thesis, University of Groningen, The Netherlands, 2022.

[2] S. Lague. Coding adventure: Ray marching (02/04/2019). `https://youtu.be/Cp5WWtMoeKg`. YouTube, accessed: 09/07/2022.

[3] K. Perlin and E. M. Hoffert. Hypertexture. *SIGGRAPH Comput. Graph.*, 23(3):253–262, jul 1989. doi: 10.1145/74334.74359

[4] R. Rosema. Unpublished manuscript. Bachelor's thesis, University of Groningen, The Netherlands, 2022.

[5] Unity Technologies. *Unity Engine Manual*, 2019. accessed: 09/07/2022.

[6] J. R. van der Zwaag. Unpublished manuscript. Bachelor's thesis, University of Groningen, The Netherlands, 2022.

[7] C. van Wezel. A virtual ray tracer. Bachelor's thesis, University of Groningen, The Netherlands, 2021.

[8] C. van Wezel and W. Verschoore. Virtual-Ray-Tracer. `https://github.com/wezel/Virtual-Ray-Tracer`, 3 2022. GitHub, MIT license.

[9] W. A. Verschoore de la Houssaije. A virtual ray tracer. Bachelor's thesis, University of Groningen, The Netherlands, 2021.

[10] W. A. Verschoore de la Houssaije, C. S. v. Wezel, S. Frey, and J. Kosinka. Virtual Ray Tracer. In J.-J. Bourdin and E. Paquette, eds., *Eurographics 2022 - Education Papers*. The Eurographics Association, 2022. doi: 10.2312/eged.20221045

[11] K. Zhou, Z. Ren, S. Lin, H. Bao, B. Guo, and H.-Y. Shum. Real-time smoke rendering using compensated ray marching. In *ACM SIGGRAPH 2008 papers*, pp. 1–12. 2008.