Contents lists available at ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Special Section on EG2022 Edu Best Papers

# Virtual Ray Tracer 2.0

Chris S. van Wezel, Willard A. Verschoore de la Houssaije, Steffen Frey, Jiří Kosinka [*]

*Bernoulli Institute, University of Groningen, The Netherlands*

A R T I C L E I N F O

*Article history:*
Received 11 November 2022
Received in revised form 15 December 2022
Accepted 12 January 2023
Available online 18 January 2023

*Keywords:*
Ray tracing
Computer graphics education
Visualization

A B S T R A C T

Building on our original Virtual Ray Tracer tool, we present *Virtual Ray Tracer 2.0*, an interactive and gamified application that allows students/users to view and explore the ray tracing process in real-time. The application shows a scene containing a camera casting rays which interact with objects in the scene. Users are able to modify and explore ray properties such as their animation speed, the number of rays and their visual style, as well as the material properties of the objects in the scene.

The goal of the application is to help the users – students of Computer Graphics and the general public – to better understand the ray tracing process and its characteristics. This includes not only the basics of ray tracing, but also more advanced concepts such as soft shadows. To invite users to learn and explore, various explanations and scenes are provided by the application at different levels of complexity, each with a step-by-step tutorial. Several user studies showed the effectiveness of the tool in supporting the understanding and teaching of ray tracing. The educational tool is built with the cross-platform engine Unity, and we make it fully available to be extended and/or adjusted to fit the requirements of courses at other institutions, educational tutorials, or of enthusiasts from the general public.

## 1. Introduction

In the area of Computer Graphics, ray tracing [1,2] is an important rendering technique. It is capable of producing realistic images and animations, albeit at a high computational cost. In real-time rendering applications, where performance is vital, ray tracing is often too slow and other rendering techniques such as rasterization [3] are used. While these techniques are fast, they often produce less convincing results. Because of this, ray tracing has seen much use in offline rendering applications such as animated films, but recent advances in graphics hardware are also making ray tracing suitable for real-time rendering applications [4], although often in combination with trained networks for real-time image denoising.

Ray tracing is, due to its prevalence and importance, widely taught in Computer Graphics courses. While the core idea of ray tracing is simple, more advanced ray tracing techniques, such as area lights and soft shadows, can become quite complicated. The understanding of these techniques can greatly be supported via visualization. Simple illustrations are frequently employed,[1] but they are 2-dimensional and static, which limits their effectiveness as an educational tool.

To address this issue, we developed Virtual Ray Tracer (VRT) [5], an interactive application that visualizes the ray tracing process. In this extended and adapted version of our original conference paper [5], we present *Virtual Ray Tracer 2.0*, an improved and extended version of the original VRT. Our novel contributions include:

- Area lights and soft shadows (via super/area sampling);
- Novel ray visualization styles (such as according to importance and/or color contribution);
- Bounding volumes for acceleration (dynamic axis-aligned bounding boxes and octrees);
- Gamification (with tailored tutorials per level/scene, reward badges, and more);
- A mobile and web version of the tool.

Our aim was to develop an improved 3D way to visualize ray tracing, make it widely accessible, and to evaluate the effect the application has on the learning process. The application has the potential to help computer graphics students understand ray tracing faster and better than they would without interactive visualizations or guided tutorials.

The main demographic for the application is the students following the Bachelor-level Computer Graphics course at the University of Groningen. The application will be used in future iterations of the course to help teach students about ray tracing. Nevertheless, the application is built to be accessible to as wide

* Corresponding author.
*E-mail address:* j.kosinka@rug.nl (J. Kosinka).

1 E.g. the Wikipedia illustration at https://commons.wikimedia.org/wiki/File:Ray_trace_diagram.svg#file.

https://doi.org/10.1016/j.cag.2023.01.005

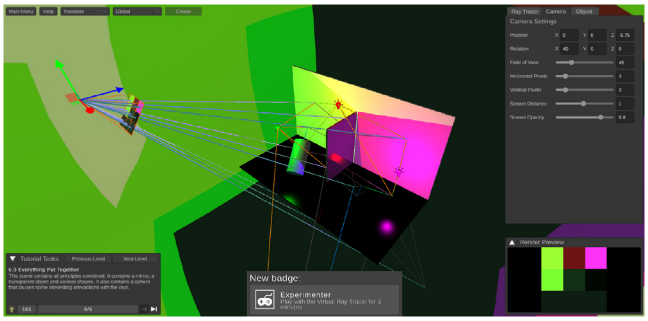


**Fig. 1.** A screenshot of Virtual Ray Tracer 2.0 showing a scene with UI elements enabled. The scene contains a virtual camera (with the translation gizmo enabled), a screen with 3 × 3 pixels, and several objects. Primary, reflection, and shadow rays are visualized (and animated) to explain the ray tracing process.

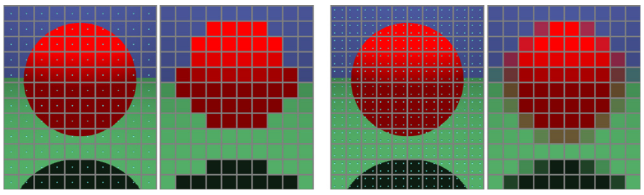

(a) No super-sampling (b) 2 × 2 super-sampling

**Fig. 2.** (a) Rendering with no super-sampling. Left is the scene with a screen overlay where each pixel has one ray in the middle of the pixel, right is the rendered image. (b) Rendering with super-sampling using 4 samples per pixel. Left is the scene with a screen overlay where each pixel has 4 rays distributed over the pixel, right is the rendered image.

an audience as possible, so anyone interested in ray tracing is enabled to understand and explore its underlying concepts.

We start by reviewing related work (Section 2) and providing relevant background information regarding VRT extensions (Section 3). In Section 4 we discuss the general design of the application while Section 5 introduces the gamification concepts that we adopted for our tool. Section 6 briefly outlines the structure of our implementation in VRT 2.0; please refer to the Appendix for an in-depth description and a detailed discussion of the changes in VRT 2.0 over the original VRT. In Section 7 we evaluate the results of the user studies we conducted. We state our conclusions and discuss potential future work in Section 8.

## 2. Related work

Our original VRT application [5] was designed to improve the learning process by providing a visualization of ray tracing techniques. While other applications visualizing certain aspects and metrics of ray tracing exist [6–9], these are not specifically aimed at education. The abundance of such applications does suggest that visualization is a useful tool in understanding ray tracing. Conversely, there are many applications aimed at teaching ray tracing, but they do not directly visualize the ray tracing process itself [10,11].

Nevertheless, the idea for an educational application that visualizes ray tracing is not entirely unique. One of the first implementations comes in the form of a set of Java Applets developed in 1999 [12]. Unsurprisingly, its age means that the application is rather simple by today's standards and thus unlikely to be useful for teaching ray tracing today. This is cemented by the fact that the application seems to be no longer available online. Its age and unavailability also mean that it is not possible to extend the application to meet modern standards of graphical fidelity and interactivity.

A more recent application similar to ours is the Ray Tracing Visualization Toolkit (rtVTK) [13]. As the name suggests, rtVTK is a toolkit for the visualization of ray tracing. The main goals of rtVTK are to aid in the development of ray tracing applications and to help with ray tracing education. However, the authors themselves admit that rtVTK may be too complicated for the latter. The main issue is that rtVTK requires users to hook up the toolkit with their own ray tracing application. While this approach means the toolkit can be used in nearly any ray tracing application, it is not exactly trivial. For an educational application meant to be accessible to as many people as possible this is not ideal.

Our VRT [5] filled this gap by providing a dedicated and user friendly educational application that visualizes the ray tracing process. The present VRT 2.0 builds on this by adding several features and extensions to VRT, including aspects of distributed ray tracing [14,15] to provide but also visualize the processes behind area lights and soft shadows, acceleration spatial data structures [1,2,16], and concepts of gamification [17,18] to make the tool fun to use and engaging, among other improvements, as discussed in detail below.

## 3. Background

Here we briefly review relevant background information concerning new concepts that we have integrated into VRT 2.0: distributed ray tracing (Section 3.1) and bounding volumes for acceleration (Section 3.2).

### 3.1. Distributed ray tracing

The term Distributed Ray Tracing was first coined by Cook in 1984 [14]. The concept entails integrating (or distributing) rays over some domain to create certain effects that are typically not available with standard ray tracing. For example, soft shadows can be created by distributing rays over the area of an area light source (as opposed to hard shadows created by point lights). In theory, to render an image with distributed ray tracing, an equation which includes integrals must be solved, such as those involving the rays distributed over a certain range. In practice, these integrals are often approximated using the Monte Carlo method [15]. By taking appropriately distributed and weighted samples (rays) from the given range, the exact value of the integral at hand is approximated. For soft shadows, this entails shooting rays to different points on an area light source and averaging the result. The downside is that many samples are needed for an accurate approximation, but the upsides are that the method itself is quite simple and that its computational cost can be adjusted depending on the desired quality. In VRT 2.0, we provide two examples of distributed ray tracing: super-sampling (Section 3.1.1) and soft shadows (Section 3.1.2), as discussed next.

#### 3.1.1. Super-sampling

Super-sampling can be used to avoid aliasing, the effect when edges appear jagged due to a low image or screen resolution. It reduces this type of aliasing with an intuitive approach: shoot (distribute) more than one ray through each pixel and average the result. The normal rendering approach in ray tracing is to determine the pixel color by shooting a ray through the middle of the pixel. This approach is illustrated in Fig. 2(a).

With super-sampling, we do not shoot a single ray through the (middle of the) pixel, but multiple rays distributed over the pixel's area. Every ray counts as a sample, and each ray contributes $1/s$ to the pixel's final color, where $s$ is the number of samples per pixel. There are many different algorithms to determine how

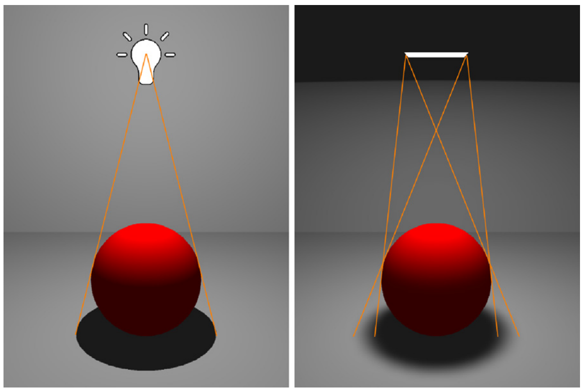


**Fig. 3.** Hard shadows produced by point light sources (left) versus soft shadows arising from area light sources (right).

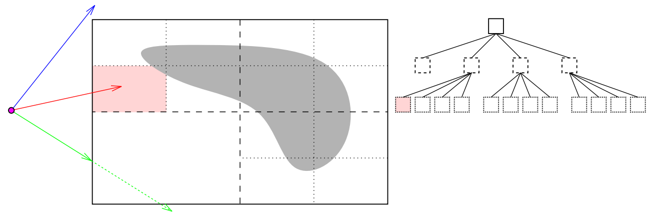


**Fig. 4.** A 2D illustration of the axis-aligned bounding box and octree concepts. The given shape (in gray) is to be intersected with several rays originating at the eye/camera (magenta). The blue ray misses the bounding box (solid rectangle), and thus the shape. The green ray hits the bounding box. When using an octree (here with levels 0 (solid), 1 (dashed) and 2 (dotted)) within the bounding box, the green ray is deemed to miss the object at level 1. The red ray intersects a cell at level 2. The parts of the object (here line segments, in 3D typically triangles) overlapping with the cell highlighted in pink are tested for intersection with the (red) ray, leading to the ray-object intersection.

to distribute the rays over the pixel's area [2]. For education purposes, we chose to go with uniform sampling with 1, 4, or 9 samples per pixel (set by the user). This means that all samples are spread out evenly, not only within each pixel but across the entire image, as can be seen in Fig. 2(b).

### *3.1.2. Soft shadows*

There are two major types of shadows: hard shadows created by point light sources, and soft shadows created by area light sources. Hard shadows have hard edges; a light ray either reaches a point light (no shadow) or it does not (full shadow). There is nothing in between. Hard shadows are often seen as unrealistic because almost all shadows in real life are soft shadows. However, they are realistic enough in some scenarios, simple to understand, and more efficient to compute than soft shadows.

Soft shadows arise from a light source that does not illuminate from a point, but from an area: an area light. An area light source may be fully visible or not visible at all from a point in the scene. Or, it can be partially visible, thus giving rise to a soft shadow. This concept is visualized in Fig. 3.

To compute soft shadows, we need to determine how much of the light's area is visible from a particular point. To get the exact answer, one needs to compute an integral over the light source's area. As mentioned above, this is difficult. Instead, we resort to distributed ray tracing again and approximate the result by taking $s$ uniformly spread samples over the area light(s) in the scene with $s \in \{2^2, 3^2, \ldots, 10^2\}$. $s$ can be set individually for each area light by the user.

## *3.2. Bounding volumes for acceleration*

A myriad of bounding volumes and their hierarchies exists to accelerate ray tracing [19]. The overarching principle is simple: given a complex object composed of many triangles, a standard ray-object intersection routine needs to check each triangle individually for ray intersections. In case a bounding volume of the object is known, it is sufficient to test only the bounding volume for ray intersection to register a miss. We bring two such volumes to VRT 2.0: axis aligned bounding boxes (Section 3.2.1) and octrees (Section 3.2.2).

### *3.2.1. Axis aligned bounding boxes*

The axis-aligned bounding box (AABB) of a 3D object with $(x, y, z)$ positions has two of its diagonally opposite corners at $(x_{\min}, y_{\min}, z_{\min})$ and $(x_{\max}, y_{\max}, z_{\max})$. AABBs spatially encapsulate the object within them. Computing whether a ray intersects a bounding box or not before checking for intersections with the object is key to accelerating intersection tests between rays and objects. If the ray does not intersect the AABB, then we can trivially ignore its intersection calculations for the object within: a ray that misses a bounding box cannot intersect anything encapsulated by that same bounding box (see the 2D illustration in Fig. 4).

### *3.2.2. Octrees*

An (axis-aligned) octree is a tree data structure for spatial partitioning of a 3D object. Each node in the tree can have either 8 or no children. At level 0, it is the AABB of the 3D object. Deeper levels are obtained by recursively splitting a node into 8 children provided it overlaps with the object. The recursion stops when a given depth is reached (or when the number of triangles in all cells is below a given minimum).

This way, just like with AABB acceleration, we check for ray and octree nodes (boxes) intersections first. The following two cases can arise: (1) The ray misses the octree (root node) completely. The ray thus misses the object. (2) The ray hits an octree node. The ray is checked, recursively, for intersection with the children of that node. Empty nodes lead to no intersection checking and if only those are encountered, the ray again misses the object. The parts of the 3D object overlapping with the non-empty nodes are checked for intersection.

This accelerates the intersection finding process because not all rays need to be checked for intersection with the (complex) object.

## 4. The application

The default setup for the application is a scene with some objects, lights and exactly one camera from which rays are cast. Fig. 1 shows one such scene. Rays slowly shoot from this camera, one ray through each pixel of the camera's screen. When a ray intersects an object in the scene, a shadow, reflection and/or refraction ray may be traced from that intersection point. Each different type of ray has its own default color. For example, reflection rays are blue while refraction rays are green. Several example scenes are shown in Fig. 5.

The user can interact with the objects, lights and camera in the scene by clicking on one to select it. This opens a properties panel. Properties of the selected object changed by the user are immediately reflected in the scene visuals and the rays being traced. The results of the visible rays are displayed on the camera's screen and in a preview window (bottom right in Fig. 1).

Again, this only shows the results of the rays being visualized, so it is very low resolution. The user can press the *Render* button in the general properties panel to open a high resolution ray traced rendering.

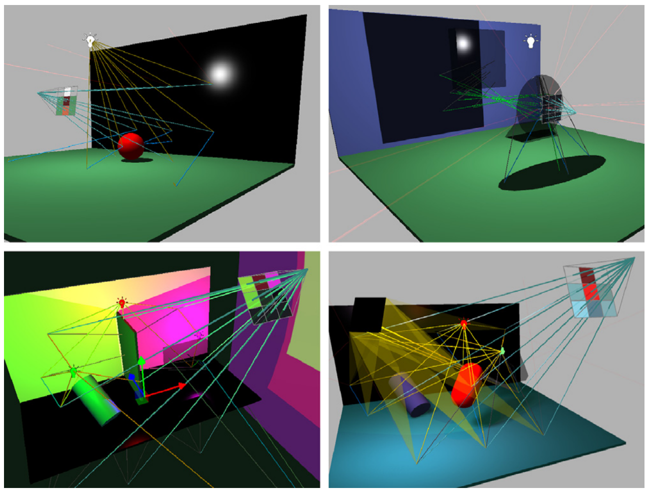


**Fig. 5.** Example scenes in Virtual Ray Tracer 2.0. Each scene comes with a virtual customizable camera and screen. A scene can contain an arbitrary number of shapes and (area) light sources with adjustable attributes (material/color, ambient/diffuse/specular coefficients, etc.). The traced rays, by default color-coded based on their type (primary, shadow, etc.), are shown as well.

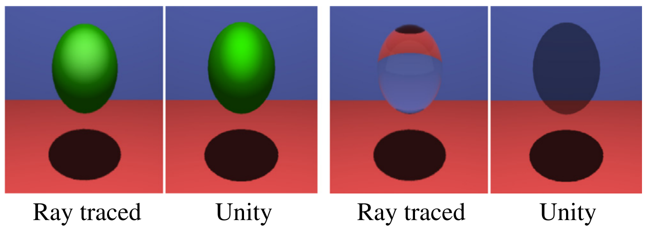


**Fig. 6.** Different materials in the ray traced renderings and in the rasterized Unity scene. Diffuse materials look nearly identical (left pair), but transparent materials are only roughly approximated in Unity (right pair).

### 4.1. Visuals

The core of the application is the visualization of ray tracing. This comprises the ray-traced scene from the point of view of the virtual camera (shown within the scene on the camera's screen and in a separate widget, see Fig. 1), the scene itself from the user's point of view (Section 4.1.1), various light source types (Section 4.1.2), the visualization of the rays traced in the scene (Section 4.1.3), and the acceleration data structures (Section 4.1.4).

#### 4.1.1. Scene visuals and rendering

Ideally, the scene visuals should match the final ray traced image when viewed from the camera's viewpoint. At the same time, it is crucial in our context that the application remains responsive at all times even for high ray counts on current commodity hardware. We have therefore opted to render the scene visuals via rasterization, which is computationally much less demanding than current real-time ray tracing methods [2]. This yields smooth and highly interactive operation on current commodity hardware – which students and the general public can be expected to have – while still conveying the visual appearance of the result. This intuitively allows users to immediately get an impression of the result of the setup within the scene without needing to view a separate window. A secondary advantage of this is that the application can be used to explain the differences between rasterization and ray tracing (something very handy in Computer Graphics courses).

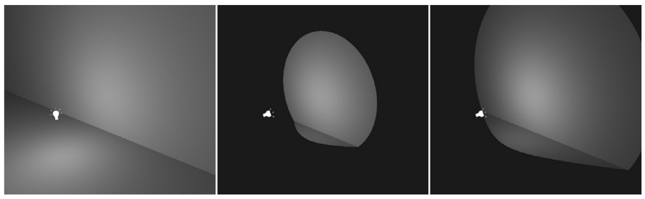


**Fig. 7.** A point light source (left) versus a spot lights source with a 90° spot angle (middle) and a 120° spot angle (right).

By employing a custom shader based on the same Phong illumination model [20] used by the ray tracer, we can make diffuse reflections and specular highlights look nearly identical. The visuals based on tracing recursive rays such as reflections and refractions can generally not be replicated as easily; see Fig. 6. The only exception is shadows where we use the built in support in Unity.

An alternative would be not to try to match the ray tracer visuals, e.g. by rendering each object with the same uniform color, regardless of the material settings provided to the ray tracer. Then there would be no confusion about some materials not lining up with the ray traced image. However, this would make editing object materials much worse, because none of the changes would be reflected in the scene visuals. This is arguably more confusing than the acceptable differences between the rasterized and ray traced visuals.

#### 4.1.2. Lights

VRT 2.0 supports a variety of light sources. While the original VRT offered only point light sources, VRT 2.0 provides also spot light sources (with angular and distance attenuation) and area light sources.

*Point light sources* are the simplest kind of light sources. They are specified by their position in the scene and their intensity. They are a close approximation of light sources like our Sun; see Fig. 7, left.

*Spot light sources* are also specified by their location and intensity. But on top of this, they are defined by additional properties, which make them behave like physical flashlights producing a cone of light. The axis of the cone is defined by their position as well as *direction*. This cone can be narrow or wide; this is controlled by *spot angle*. An example can be seen in Fig. 7. As the light source itself is an infinitely small point, just like the point light, light rays work exactly the same as with point lights, up to one difference: if a light ray falls outside the cone of the spot light, the light source is ignored (the origin of the light ray is not lit).

*Area lights* have some things in common with spot lights: they also have a direction and an 'angle', but this angle is fixed at 180°. Unlike the other lights, the area light is no longer an infinitely small point, but an area. As this is a better approximation of most real-world lights, it greatly adds to the realism of scenes by giving rise to soft shadows (Section 3.1.2).

An area light is visualized by a simple rectangle with the front side colored as the light's color and the back side colored black. An example can be seen in Fig. 3, right. As for the rays, we consider two options. We either show every ray as an individual ray from the point to the sample location, or one big 'cone' from a point to the area light source. The latter approach can dramatically reduce the number of rays, and thus improve performance and avoid visual clutter. In contrast, the former approach clearly visualizes the sampling process. We chose to go with a combination of the two. Up to $4^2 = 16$ samples, all rays are visualized by individual rays. From $5^2 = 25$ samples upward, if

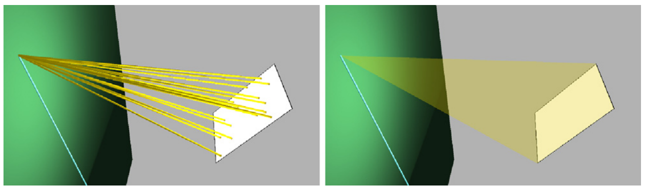


**Fig. 8.** An area light sampled using 16 of fewer samples visualizes every ray (left) and an area light with 25 or more samples shows only a single area-ray (right).

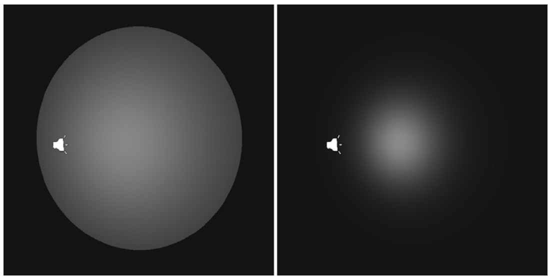


**Fig. 9.** No angle attenuation (left) versus angle attenuation (right).

all rays reach the light or all do not, they are transformed into a pyramid which we call an *area-ray*; see Fig. 8. If some rays reach the area light and some do not, we show the rays individually.

To further increase realism, we add light attenuation in VRT 2.0. Light attenuation means that light intensity gradually decreases according to some formula. We use two types of light attenuation: distance attenuation and angle attenuation.

*Distance attenuation* is a simple way of attenuating light according to the inverse square law. The intensity of the light decreases proportionally to the square distance from the light source. This can be seen from the formula of the surface area $4\pi r^2$ of a sphere with radius $r$. As the light travels farther (radius increases), the area covered by the exact same number of light particles (photons) increases quadratically.

*Angle attenuation* is a type of attenuation that applies only to light sources with a direction: spot and area lights in our case. It again models real flashlight behavior, this time by lowering the spot/area light's intensity according to the angle between the light's direction and the direction to the scene point in question from the light. This is shown in Fig. 9.

### *4.1.3. Ray visualization*

In order to visualize ray tracing in an intuitive and appealing way, we need to carefully consider how we draw the rays. It needs to be clear where a ray is coming from, where it is going, what kind of ray it is, and potentially also what sort of information (color) it contributes. All this combined with the fact that it should be possible to draw many rays at once makes for an interesting design and engineering challenge.

The core idea behind the ray design is simplicity. We want the user to be able to clearly see the rays, but we do not want to make the rest of the scene difficult to see. This becomes especially true when the number of visualized rays is large. Therefore, the rays should have a simple shape and material.

The simplest shape for a ray is a cylinder. We could argue that a cylindrical arrow is a better option because, while more complex, it also indicates the direction of the ray. However, we believe that animating the rays is a more natural way of showing the direction of a ray, while keeping the actual ray object simple. What we mean by animation is that we gradually extend rays from their origin towards their end point. Once a ray has reached

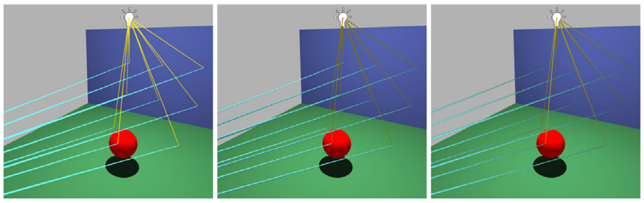


**Fig. 10.** Left: Rays unaffected by lights. The rays look 2D, are too prominent, and their positions are difficult to determine. Middle: Ray lighting in the original VRT tool. Most rays look 3D and their positions are easily determined. However, light rays still look 2D. Right: Ray lighting in VRT 2.0. All rays look 3D due to a point light source at the user's eye. This light source influences only the rays.

its full length, its child rays (such as reflection rays) start their animation. We do things this way as it clearly demonstrates the recursive nature of ray tracing (see the accompanying video).

We decided to shade the rays based on the scene's lighting conditions combined with an ambient color (VRT versions), and a point light source at the user's eye (added in VRT 2.0) that influences only the rays, which are themselves not influenced by any other (user-specified) lights in the scene. This provides a better understanding of their position and orientation in the 3D scene. The ambient color was added to make the rays easily visible even when there is no lighting in the scene, and the extra illumination from the eye provides a 3D look to light rays, especially in scenes with only a single point light source. To better distinguish between different kinds of rays, rays are by default colored based on their type. The differences among the three approaches is shown in Fig. 10.

One of the consequences of showing all rays for area lights, to visualize super-sampling, and screens with lots of pixels is potential visual clutter. To avoid this, VRT 2.0 introduces several ray visualization styles. The original VRT tool provides only limited options: rays have a color corresponding to their type; see Fig. 11, far left. All rays have the same radius, controllable by the user, and only rays that do not hit anything can be hidden. We have added several additional options to change the rays' appearance to help the user figure out what is happening in the scene, as follows.

Some rays are really important as they greatly impact their pixel's color, and some are negligible, contributing nearly nothing. Especially with higher levels of recursion, there may be a lot of these rays that have barely any impact. To declutter the scene, the user may want to hide these rays. To do this, we have added the option to *hide negligible rays*. This hides rays that contribute less than a set threshold.

By making rays transparent, we can show lots of rays whilst still being able to see the scene. How transparent a ray should be is subjective to how visible the user wants the scene to be and how visible the rays should be. We have added the option *ray transparency* and made the level of transparency controllable by the user.

As already mentioned, some rays are more important than others. Another option to distinguish among these rays is to adjust their size, which is in this case their radius, depending on their contribution. We have added *dynamic ray radius* as an option to make the ray's radius depend on its contribution. This maps every ray's contribution to a radius between a certain minimum and maximum radius set by the user.

Lastly, it is instructive and interesting to the user to be able to see where the pixel's color comes from. To this end, we have added the possibility to change the ray's color to the *color it contributes* to the pixel's color. This way, the user can visually see how the pixel gets its color. Several combinations of these ray visualization options are shown in Fig. 11.

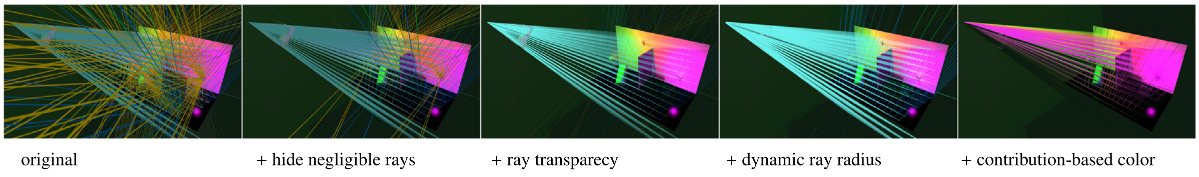


**Fig. 11.** Various ray visualization styles. From left to right (each time, an additional option is enabled): All disabled (the only option available in the original VRT tool), hide negligible rays, ray transparency, dynamic ray radius, contribution-based ray color.

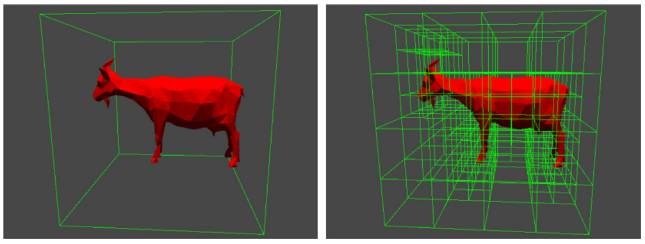
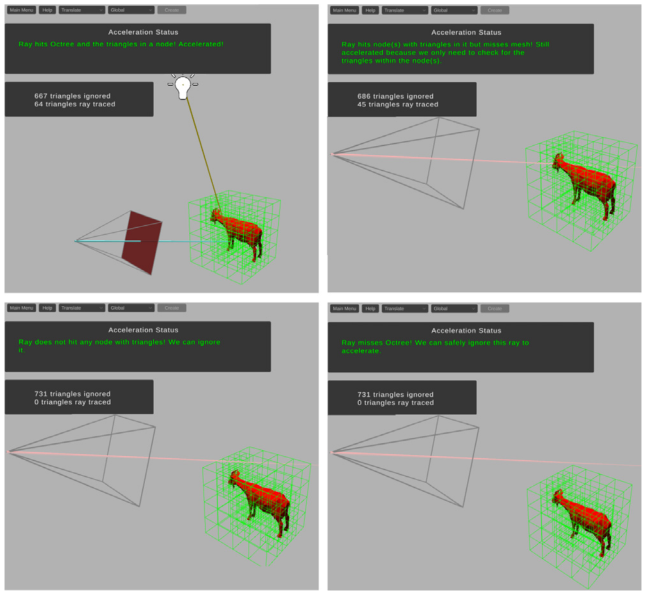


**Fig. 12.** The axis-aligned bounding box and level 0 octree (left) and level 3 octree (right) of a goat mesh. Note the adaptivity of the octree: empty regions are not subdivided; cf. Fig. 4.

### 4.1.4. Acceleration structures

The axis-aligned bounding box and the octree data structure are represented as wire frames around the object they bound (see Fig. 12). The acceleration status is reported to the user, as can be seen in Fig. 13 in the case of an octree built around a goat mesh. The user is informed not only of the case that arose in the intersection test (such as a miss), but also of the number of ray-intersection computations performed and avoided. For educational purposes and the sake of clarity, the screen is limited to only one pixel, i.e., only one primary ray is used in the scene.



Acceleration Status
Ray hits Octree and the triangles in a node! Accelerated!
667 triangles ignored
64 triangles ray traced

Acceleration Status
Ray hits node(s) with triangles in it but misses mesh! Still accelerated because we only need to check for the triangles within the node(s).
686 triangles ignored
45 triangles ray traced

Acceleration Status
Ray does not hit any node with triangles! We can ignore it.
731 triangles ignored
0 triangles ray traced

Acceleration Status
Ray misses Octree! We can safely ignore this ray to accelerate.
731 triangles ignored
0 triangles ray traced

**Fig. 13.** The various feedback provided to the user of VRT 2.0 in a scene with an octree of a goat mesh (with 731 triangles in total) depending on the mutual position of the single ray in the scene, the octree, and the mesh within it. In each, the user learns how many ray-triangle intersections tests have been avoided thanks to the octree.

## 4.2. Settings and controls

One of the most important features of the application is its interactivity. We want the user to be able to experiment with different settings and to learn ray tracing and its components that way. To this end, we allow users to change a wide variety of settings and properties of objects in the scene, and to add or remove a predefined set of objects to/from the scene (see the accompanying video).

We aim to make our application as accessible and intuitive as possible with our user interface (UI) design. Most of our UI components are on the right side of the screen. This means that the scene is always visible so that any changes made in the properties panel have an immediate and obvious effect. Another important concept is documentation. All elements in the properties panel have tooltips, and most aspects of the application are explained in the help panel. It is important that this information is easily accessible, but only visible on demand. Further, the properties panel's contents change based on the user's selection. For example, when the camera is selected, only its settings are displayed.

Nevertheless, the best UI is, arguably, no UI. If something can be done intuitively just through keyboard and mouse input, it is almost always better than designing UI components for it. A good example of this are camera controls as well as positioning, rotating and scaling objects in the scene. By placing shapes on a selected object that can be clicked and dragged, we can translate, rotate and scale the object in an intuitive way. This technique is commonly employed in the scene editors of game engines (such as Unity's editor). The shapes are often called transformation gizmos. Fig. 14 shows what this looks like in our application. Because clicking and dragging may not always have the desired precision, we still have position, rotation and scale UI components in the properties panel, but in most cases the gizmos are a much more direct and intuitive way to transform an object.

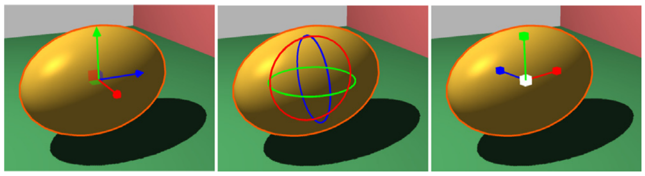


**Fig. 14.** Transformation gizmos. From left to right: Translation gizmo, rotation gizmo, and scale gizmo.

In addition, toggles for each type of light can disable those types of lights completely. This allows the user to experiment, for example, with different light types at the same positions. We also added a button that 'flies' the user to the virtual camera that the ray tracer uses. This allows the user to compare the rasterized scene with the ray traced scene.

As rendering a scene with an area light and super-sampling can take longer, we added a progress bar to the render screen so the user can estimate how long they have to wait before the render is done.

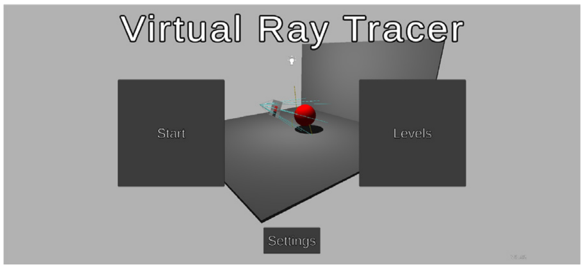


**Fig. 15.** Home screen of the VRT mobile application [21].

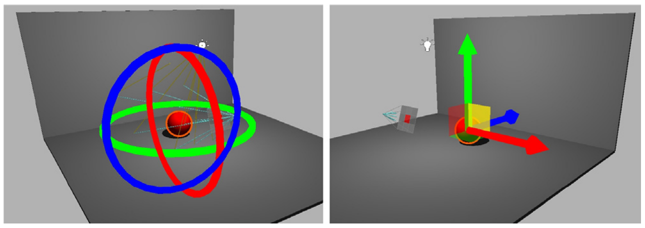


**Fig. 16.** Gizmos of the Virtual Ray Tracer mobile application; cf. Fig. 14.

### 4.2.1. Mobile controls

Besides desktop versions for all major operating systems, VRT 2.0 is also available as a web tool [22]. In terms of controls, the web port did not require any changes. However, the same cannot be said of our mobile port for Android [21] (note that due to computational and screen space limitations, this port currently supports features of the original VRT tool only). We now elaborate on the needed changes.

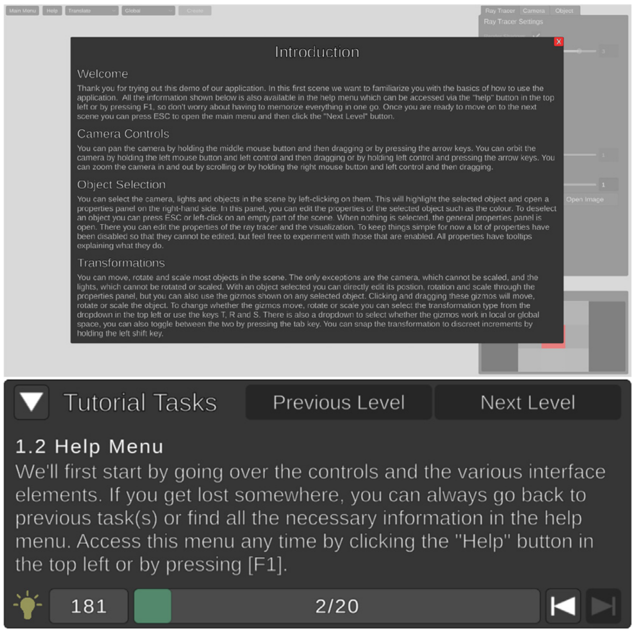
The WebGL version of our tool does not work particularly well (or not at all) on mobile devices [23], and features like keyboard input are (typically) not available. All UI components had to be adapted to make the application readable and usable for mobile users. The home screen is shown in Fig. 15. In general, the buttons, menus, bars, and hitboxes have been made bigger (relative to screen size), and the color picker has been adjusted for mobile screen use as well. Another modification concerns the gizmos used to translate, rotate and scale objects within the scene: they have become larger and are in turn easier to use on mobile devices; see Fig. 16.

Besides all these UI changes, input methods [24] had to be changed, too. The application has three ways to move within a scene: panning, zooming and orbiting. Panning on the mobile version of VRT is done using one touch and dragging this touch across the screen. Zooming within the mobile version of VRT is done with the use of two touches. Orbiting is accessed by double tapping the screen and holding the second tap, and then dragging that touch in the desired direction.

## 5. Gamification

The main idea behind gamification is for users to engage in specific actions and behaviors in return for rewards. When our brain expects these rewards, it will release dopamine, which gives feelings of pleasure. We have thus decided to gamify VRT to ensure its users are extrinsically motivated [25] to engage with the tool in order to receive these rewards, and in turn learn about ray tracing in the process. Intrinsic motivation is harder to obtain [26]. In our case, we assume that users are already intrinsically motivated to learn about ray tracing, in which case extrinsic-based gamification can be very effective [18].

Following the concept of BLAP (Badges, Levels/Leaderboards, Achievements, and Points) gamification [18], we have added the following features to VRT: a step-by-step tutorial (Section 5.1), points and unlockable items (Section 5.2), and badges (Section 5.3).



Introduction

Welcome

Thank you for trying out this demo of our application. In this first scene we want to familiarize you with the basics of how to use the application.  All the information shown below is also available in the help menu which can be accessed via the "help" button in the top left or by pressing F1, so don't worry about having to memorize everything in one go. Once you are ready to move on to the next scene you can press ESC to open the main menu and then click the "Next Level" button.

Camera Controls

You can pan the camera by holding the middle mouse button and then dragging or by pressing the arrow keys. You can orbit the camera by holding the left mouse button and left control and then dragging or by holding left control and pressing the arrow keys. You can zoom the camera in and out by scrolling or by holding the right mouse button and left control and then dragging.

Object Selection

You can select the camera, lights and objects in the scene by left-clicking on them. This will highlight the selected object and open a properties panel on the right-hand side. In this panel, you can edit the properties of the selected object such as the colour. To deselect an object you can press ESC or left-click on an empty part of the scene. When nothing is selected, the general properties panel is open. There you can edit the properties of the ray tracer and the visualization. To keep things simple for now a lot of properties have been disabled so that they cannot be edited, but feel free to experiment with those that are enabled. All properties have tooltips explaining what they do.

Transformations

You can move, rotate and scale most objects in the scene. The only exceptions are the camera, which cannot be scaled, and the lights, which cannot be rotated or scaled. With an object selected you can directly edit its postion, rotation and scale through the properties panel, but you can also use the gizmos shown on any selected object. Clicking and dragging these gizmos will move, rotate or scale the object. To change whether the gizmos move, rotate or scale you can select the transformation type from the dropdown in the top left or use the keys T, R and S. There is also a dropdown to select whether the gizmos work in local or global space, you can also toggle between the two by pressing the tab key. You can snap the transformation to discreet increments by holding the left shift key.

Tutorial Tasks | Previous Level | Next Level

1.2 Help Menu

We'll first start by going over the controls and the various interface elements. If you get lost somewhere, you can always go back to previous task(s) or find all the necessary information in the help menu. Access this menu any time by clicking the "Help" button in the top left or by pressing [F1].

181 | 2/20

**Fig. 17.** Top: The pop-up message in the first level of the original Virtual Ray Tracer tool. Bottom: The step-by-step tutorial in the new version of Virtual Ray Tracer, in this case displaying the Help Menu task of the first level. These tasks appear in the bottom left part of the screen; see Fig. 1.

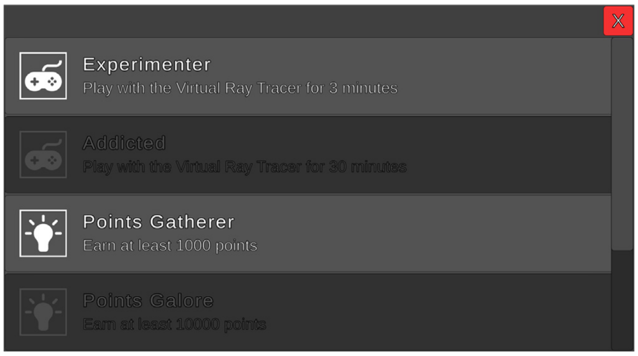
### 5.1. Step-by-step tutorial

The original VRT tool includes a pop-up message with explanations about the application itself or ray tracing before the start of each level. Fig. 17, top, shows one of these pop-up messages.

These messages might be skipped because of their length, causing users to miss important concepts. Others might read them but still miss important information because they failed to remember (some of) it. Replacing these pop-up messages with a step-by-step tutorial addresses these problems and provides additional benefits. The lengthy text is split into small, individual steps, which makes the users feel like they never have to read large amounts of text. Furthermore, the complementary concepts can be separated from the critical concepts. The critical concepts are made mandatory and the complementary concepts remain optional, reducing the amount of required reading even further.

Additionally, the steps that explain certain concepts can be transformed into small tasks by adding extra actions that the user needs to complete before being able to continue to the next step or task. This way, concepts can be practiced immediately, making users more likely to remember them. If users do get lost somewhere, they can always go back to a previous task to refresh their memory. Lastly, a progress bar is shown at the bottom. Besides giving the user an idea of their current progress, it also gives the user motivation to fill in the entire progress bar. An illustration of the step-by-step tutorial in VRT 2.0 is shown in Fig. 17, bottom.

It displays the task name and description as well as the current level and task number. The progress bar shows the current progress in both visual and textual form. The step-by-step tutorial also contains buttons to go to the previous/next level and the previous/next task. As can be seen in Fig. 17, bottom, most of the buttons are in a disabled state. This is the first level, so going back



Experimenter
Play with the Virtual Ray Tracer for 3 minutes

Addicted
Play with the Virtual Ray Tracer for 30 minutes

Points Gatherer
Earn at least 1000 points

Points Galore
Earn at least 10000 points

**Fig. 18.** Badges in the new version of Virtual Ray Tracer. The first and third badge have already been unlocked.

is not possible. The mandatory tasks for this level have not been completed yet, so the user cannot go to the next level either. The user is not allowed to go to the next task yet as they first need to complete the task mentioned in the task description. We have also added a free (cheat) mode option that allows the user to skip across tasks and levels. This allows VRT 2.0 to be used in a demo mode, such as at public engagement events or during lectures, and to avoid potential frustration of returning users.

### 5.2. *Points and unlockable items*

Points in VRT 2.0 can be earned by completing the mandatory and optional tasks in the step-by-step tutorial. Tasks that are more difficult and take longer are rewarded with more points, giving users more satisfaction when finally completing them. Optional tasks are also rewarded with more points to incentivize users to complete those tasks as well.

Of course, having lots of points would not be very rewarding if they could not be used somewhere. VRT 2.0 offers a sandbox level where users are able to add objects themselves to create their own scenes. These objects are locked until enough points have been gathered, after which they become available.

### 5.3. *Badges*

Badges are earned by playing with VRT 2.0 for a certain amount of time, gathering a certain number of points, or creating a certain number of objects in the sandbox level. The tool has a notification menu (see Fig. 1, bottom middle) that shows a pop-up notification whenever a new badge has been earned. Fig. 18 shows the overview of badges in VRT 2.0 available via the main menu.

In addition, we have added various sounds (such as for earning badges) and animations (e.g. fading out/in when transitioning between levels) to VRT 2.0 to create a more fun and engaging experience.

## 6. Implementation

We now present an overview of the implementation of VRT 2.0. Readers interested in particular details of our implementation can find them in Appendix.

The application is implemented in Unity.[2] Unity is a freely available game engine that can be used for 2D and 3D applications. It is widely used, well documented, and there are many resources online that explain and discuss its various components.

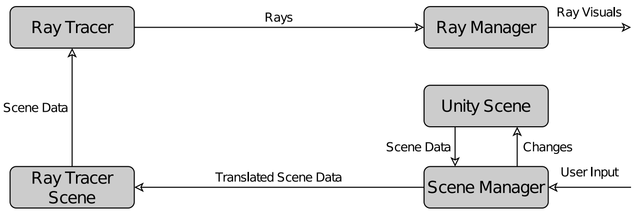


**Fig. 19.** The design of the general application structure.

This is important for extensibility. Unity also provides build support for a wide range of platforms, which allows us to make the application widely available [27].

Conceptually, the application consists of two core components: the ray tracer and the 3D Unity application. The ray tracer takes information about a scene and produces a list of rays traced in that scene from the camera. The Unity application renders the scene from a different perspective (that of the user in the interactive application), crucially also incorporating the rays generated and followed by the ray tracer.

In Unity a scene contains all information regarding the current state of the application, among others including UI elements and rays. Most of this information is irrelevant to the ray tracer: it only needs to know about the camera, the objects in the scene, and their materials. As already mentioned, the Unity scene can change dynamically based on user input.

Our design to meet these requirements is illustrated in Fig. 19. The Unity application takes input from the user and changes its internal scene. These changes then need to be sent to the ray tracer, which uses its own, simpler scene representation (*Ray Tracer Scene* in Fig. 19). For this, we include a translation layer that converts the Unity scene to the format of the ray tracer, which then outputs a list of rays for the Unity application to draw (via the *Ray Manager*).

Our interactivity and performance requirements mean that the translation from the Unity scene to the ray tracer needs to be fast. For this, we have implemented the ray tracer using Unity's built in ray casting utilities. These utilities work directly with Unity's internal scene representation, so the translation layer is comparably lightweight: it essentially comes down to filtering for the objects relevant to the ray tracer.

Implementation details regarding the scene and ray managers, the ray tracing process, ray visualization, acceleration structures, lighting, gamification, and VRT ports to web and mobile can be found in the Appendix: Implementation Details.

## 7. Evaluation

For performance evaluation, we used a laptop running Windows 10 with an Nvidia GTX 1050 GPU and an Intel i7-7700HQ CPU. With this setup, our application runs at highly interactive frame rates beyond 100 fps for a virtual screen size of $8^2$ (beyond which the presentation becomes significantly cluttered and some of the novel VRT 2.0 ray visualization/hiding mechanisms need to be enabled). On low-end systems of some users, lower (yet still interactive) performance has been noted in the context of distributed ray tracing (see below). A detailed performance evaluation can be found in [5].

A series of separate user studies has been conducted to evaluate the basic tool as well as novel components in VRT 2.0. Below, we describe the setup and discuss the results of each (the respectively reported participant numbers include only those

2 https://unity.com/

with successful completions). We present here only overviews of these studies; please refer to the cited works for details.

*Original VRT user study* [5]

In this study we had 17 participants, from which 8 had previously followed the Computer Graphics course. Of the 9 that had not, 6 were members of the general public with no formal education in Computing Science or a related field. The participants were asked to fill in a survey with a set of questions targeted at education potential (what they learned) as well as technical aspects (quality of visuals, ease of use, etc.), and were also given the opportunity to provide general comments.

The participants generally think that the application helped them to understand ray tracing better. All computer graphics students would consider the tool to be helpful for future students of the course. There were numerous positive comments about the scenes and the visualization, e.g.: "*I really liked how you could see the rays being traced, how they bounced off of different objects, and how they refracted. It was a great visual showcase of how ray-tracing works, and I think students should be able to better learn the concepts using it.*"

However, students with prior experience in ray tracing would have liked to see more detailed, mathematical explanations of underlying concepts in the application. One such participant said: "*In context of the CG course I think this would be nice to see as an introduction to the corresponding topics (shadows, reflections etc.) however it does lack a bit in the mathematics part of ray-tracing.*"

Almost all participants considered the application to be easy to use. However, it was mentioned by some that they found some aspects of the application confusing or difficult to understand. From detailed feedback we saw that controls and user interface are rated the lowest (with the most potential for improvement). However, the majority of users still thought they were at least acceptable.

Overall, the studies show that the tool achieves its educational purpose concerning computer graphics students and also allows to reach the general public.

*Distributed ray tracing* [28]

This user study considered 61 members of the general public as well as 31 high-school students. We asked the users in six questions to indicate on a scale of 1 to 5 how much the tool helped with understanding a certain concept. In addition, two open questions concerned how the tool helped (or not). Lastly, we asked the participants what they thought of the complexity of the tool and if they had any other additional feedback or general or technical remarks.

For participants belonging to the general public, the differences between the types of light sources and the idea behind soft shadows were understood pretty well. However, some struggled with grasping the concept of light attenuation. In general, advanced ray visualization options were considered to be quite helpful, especially ray transparency and contribution-based ray color. Users with a Computing Science background generally found the extensions to be very usable and the underlying concepts to be easy to understand.

For the group of high school students, the tool generally helped them to understand ray tracing quite well, although some said that in some cases they struggled with the complexity and understanding the used terminology. Advanced ray visualization was particularly appreciated, as has already been noted for the other groups as well. Some participants remarked occasional performance issues on older, lower-end computers.

In summary, this extension successfully helps users to understand distributed ray tracing. The evaluation showed that the target audience of Computer Graphics students can study with the tool very well, also other groups can effectively learn ray tracing from this application, given some basic knowledge and interest. The two main obstacles that were observed concerned the provided descriptions (especially the use of terminology) and the interaction with the tool (this was found to heavily depend on prior experience with 3D environments in particular).

*Acceleration data structures* [29]

We conducted a smaller study with three computer graphics students who were already familiar with the basics of ray casting. We asked for comments from the participants via four questions concerning the visual clarity of the data structures, interaction with rays and objects, additionally displayed information, and overall learning experience.

The participants noted that the acceleration data structures are clearly visualized. Focusing on a single ray and single object helps in applying the concepts in a smaller scope. The acceleration status display works well in illustrating the different cases that can occur when accelerating ray tracing. However, it was mentioned that the octree structure does not show clearly at certain angles, as the lines of wire cubes drawn in the denser areas of the octree overlap. It was further suggested that changing a 3D object of the scene and observing the impact of this change could be quite helpful. Feedback further indicates that more complex setups considering multiple rays or objects would also be regarded as beneficial.

*Gamification* [30]

The user study was conducted with 30 participants from the general public. We provided a web build for both the original and new version of VRT, and asked the participants to equally familiarize themselves with both of them for 10 min before answering multiple choice questions (between two and six response choices, depending on the question). These asked for feedback regarding technical aspects, the educational benefit, and the entertainment value. This was followed by an opportunity to provide open comments.

In this study we evaluated the differences between the original (non-gamified) to a new gamified version of VRT. The majority of users (73%) preferred the new version for understanding ray tracing better, while only 13% preferred the original version (the remainder stated that they had no preference). The users that did prefer the original version in this regard mentioned the (lack of) freedom in the new version compared to the original version. Most users found the gamified version to be easier (77%) and more fun (63%). The percentage of users that experienced issues with the respective tool decreased from 37% in the original version to 20% in the gamified version. Considering all factors, generally the majority (77%) preferred the new version overall. The feedback showed that participants – especially non-experienced users – found the step-by-step tutorial particularly helpful.

*Mobile version* [31]

The user study had 15 participants who belonged to one of two distinct age groups: eleven were 19–22 years old (students of Computing Science and related fields), and the remaining four were members of the general public, aged 45+. Notably, the participants from the second group had less experience with computers or mobile devices. This difference is clearly reflected in the user experience regarding ease of use (see below). The participants were asked to comment on the user interface (menus and panels) and interaction modalities (touchpad navigation) as well as to provide general feedback about the Android application.

The participants were generally positive about the mobile adaption of Virtual Ray Tracer. The study particularly focused on the user interface (UI) that has been revised to be more suitable for touchscreen usage. Most of the UI was received well and was noted to be clear and easy to navigate. 60% of the total participants were able to operate the tool without issues. The others (40%) mentioned they had minor difficulties with either selecting objects or gizmos. A few participants mentioned the orbiting to be a bit difficult, but most of the users had no particular problems with the application after getting used to it. One participant noted that it was hard to see objects while making adjustments.

The main menu and help menu were also revised for the mobile version and generally well-received by all of the participants. Overall, they were found to be sized well, clear, and easy to navigate. Two older Dutch participants from the general public mentioned that they struggled due to the fact that descriptions are given in English, providing an indication that offering translated versions could help to make the tool more accessible for public outreach.

## 8. Conclusion

We have presented VRT 2.0, an improved and extended version of Virtual Ray Tracer [5], an interactive application that visualizes ray tracing. It was designed to help with teaching ray tracing and to be used in the Computer Graphics course taught at the University of Groningen, but hopefully also beyond that. We have evaluated the application and its novel features through several user studies. Regarding the educational potential of the application, the results are positive. It is clear that visualizing ray tracing can be of great help in understanding it better. It is also useful to be able to change ray tracer settings and the properties of objects in a scene with the visualization updating based on every change made. This interactivity allows users to experiment and see how various settings affect the rays being traced.

The novel features in VRT 2.0 include distributed ray tracing (showcased on area lights and soft shadows), acceleration data structures (axis-aligned bounding boxes and octrees), gamification of the tool, and its ports to web and mobile (Android). This has addressed most of the original feedback that we received on VRT [5], such as adding a step-by-step tutorial and options for decluttering the scenes with too many rays. And it makes our tool, which remains fully open source [27] (MIT licence), useful in teaching ray tracing beyond its basics, and also widely applicable and accessible.

We hope that this will enable a wider adoption of Virtual Ray Tracer, not only in university courses in graphics, but also at events aimed at graphics enthusiasts and the general public.

In the future, we plan to incorporate ray tracing also for rendering the scene visuals. The challenge here is to maintain high responsiveness of our tool on commodity hardware and in the web environment. At the same time, using the current rasterization approach brings secondary benefits: it provides a direct way of comparing advantages and disadvantages of the two rendering methods, something very useful in graphics education.

For students of the Computer Graphics course, it could be useful to extend the application in a way that allows them to adjust the ray tracing process itself. We also aim to conduct an extended study with a larger group of participants and evaluate various learning goals.

In terms of further features and extensions for VRT 2.0, the following directions sound promising and useful for graphics education: adding support for textures (potentially both raster and procedural), incorporating a pseudo-code representation of the internal ray tracer and (visually) stepping through it in the scene (something for advanced users and those looking to learn how to implement ray tracing), and ray marching (which we have already partly explored [32]).

Virtual Ray Tracer is fully and openly available on GitHub [27], which is also where all related material and future updates can be found.

## CRediT authorship contribution statement

**Chris S. van Wezel:** Software, Investigation, Writing – original draft. **Willard A. Verschoore de la Houssaije:** Software, Investigation, Writing – original draft. **Steffen Frey:** Conceptualization, Methodology, Writing – review & editing, Supervision. **Jiří Kosinka:** Conceptualization, Methodology, Writing – review & editing, Supervision, Project administration.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Data availability

It is an open-source project, fully available on GitHub.

## Acknowledgments

The original VRT tool [5] was based on the BSc theses of the first two authors at the University of Groningen in 2021. The new VRT version presented in this paper is further based on the BSc theses of Peter Jan Blok, Roan Rosema, Bora Yilmaz, and Jesper R. van der Zwaag at the University of Groningen in 2022.

## Appendix. Implementation details

We now present the implementation details of VRT 2.0, and in doing this also discuss parts which are shared with the previous version of VRT [5]. The overall program structure was discussed in Section 6. We now go into the details of the scene and ray managers, the ray tracing process, ray visualization, acceleration structures, lighting, gamification, and VRT ports to web and mobile.

### *A.1. Scene and ray manager*

The application is split into the ray tracer and the Unity application, whereas the Unity application provides the input for the ray tracer and also handles its output. To better separate these two components, we have written the Unity side of our code to contain two important manager objects: the *Scene Manager* and the *Ray Manager* (see Fig. 19).

The scene manager handles the input for the ray tracer. This means that it manages any changes that are made to the scene by the user and presents that scene data to the ray tracer. As discussed in Section 6, the ray tracer is implemented in Unity and can directly use Unity scene data, so all the scene manager has to do is to collect this data into one convenient scene object. This scene object is simply a list of references to objects in the Unity scene, but with one important addition: whenever an object property is modified, an event gets sent to inform listeners that the scene has changed. These events allow us to avoid unnecessary calculations when the scene has not changed.

The ray manager handles the output of the ray tracer. It requests a list of rays from the ray tracer and is responsible for visualizing those rays. At the start of the application, the ray manager obtains a reference to the scene manager and the ray tracer. It subscribes to the events the scene manager sends out whenever a change is made to the scene. The ray manager also listens for similar events sent out by the ray tracer whenever its settings are changed. When either type of event comes in, the ray manager makes the ray tracer produce a new set of rays. These new rays are then drawn in the Unity scene, as described in Appendix A.3.

### A.2. Ray tracer

Our ray tracer is based on Whitted's model [33] with Schlick's approximation for refraction [34]. As discussed in Section 6, we make use of built-in Unity functions for casting rays and determining object intersections. More precisely, we employ Unity's `Physics.Raycast` function.[3] It casts a ray from a given point and returns information about the first object with a `Collider` component it intersects. From this we determine the location of the intersection, the type of object that was intersected and its material, and all other information that is needed for ray tracing.

Note that our ray tracer's main output is a set of rays, not primarily an image. Of course, our ray tracer can still produce an image, but there is an entire set of functions for generating rays that is unique to our ray tracer. First of all, the rays themselves are stored in simple ray objects. These contain the ray's origin, direction and length, but also the ray's type and color. Rays have types for the purpose of visualization. For example, we want to be able to distinguish between a ray produced by a reflection and a ray produced by a refraction. We can thus color the rays drawn based on their type to make it clear to the user what each ray does in the scene. Besides its type, the ray object also contains the color it contributes to its pixel in the final image. This way we can create an image from a set of rays.

The ray tracer outputs the rays in a tree structure. Because rays are traced in a recursive fashion, this tree arises naturally: each recursively called trace function just adds its ray as a child of the ray of its caller. Each pixel thus corresponds to one tree with the root ray traced from the camera through the pixel. This means that the final output of the ray tracer is a list of ray trees (one for each pixel).

### A.3. Ray visualization

As described in Section 4.1.3, we animate the rays by elongating them in a recursive fashion. One advantage of this approach is its comparably simple implementation with our organization of rays in a tree structure (see Appendix A.2). For animation, we recursively traverse this tree until we find a ray that is not fully extended and increase its length by a small amount. Doing this in each frame until all rays are at their full length results in the desired animation. It is possible to reset the animation back to the start by going through the ray trees and setting each ray's length to zero. Note that this simple approach traverses each ray tree every frame, while only a few, not fully extended rays may be of interest in that frame. If needed this could be improved by maintaining a list of the currently active rays, but the induced extra cost is negligible overall.

While often a rather small number of rays is shown for the sake of visual clarity, there are situations where drawing hundreds – up to even thousands – of rays is beneficial. Most importantly, it shows how the rays, as a collective, bounce around in the scene, and it can allow the identification of individual rays with interesting behavior (e.g., a ray bouncing several times before leaving the scene). It can also help to convey a better impression of the amount of work involved in ray tracing a high resolution image.

Drawing a large number of rays does come at a significant computational cost though, so the ray drawing code needs to be well designed to handle dynamically generated rays. When the rays we need to draw have changed since the last frame, for example due to the camera being moved, we cannot simply move the existing ray objects in the same direction as the camera because the structure of the ray trees and the number of rays may have changed. Unfortunately, the simplest option to destroy the old ray objects and create new ones in the right positions is not feasible, as the Unity functions corresponding to these actions, `Destroy`[4] and `Instantiate`,[5] are not fast enough to handle hundreds of rays. This means that we need to reuse the already existing ray objects in the scene, even if the structure of the ray trees is different from what it was before.

The solution lies in noticing that there is a difference between the plain data rays produced by the ray tracer and the Unity scene ray objects used to visualize that data. We can keep the ray objects around when the ray trees change, but we need to update their positions and colors to match the new data, and we may also have to hide some objects if the total number of rays has decreased. This can be achieved through the use of a so-called *object pool*.

An object pool is a design pattern commonly used in Unity applications when a lot of instances of the same type of object have to frequently be created and destroyed. It works by keeping a large number of instances of the object in a pool. When a new object needs to be created we instead activate an unused one from the pool, and when it needs to be destroyed we deactivate it. Because activating and deactivating an object is much faster than creating or destroying it, this significantly improves performance.

In our application we store the ray objects in such a pool. When a new set of rays comes in from the ray tracer, we take one ray object from the pool for each ray, activate it, and set its position and color to reflect the ray. If there are ray objects in the pool that are still active from before but are not being used for the new rays, they are deactivated. This allows us to update hundreds of rays every frame while maintaining good frame rates.

In order to support all new ray visualization in VRT 2.0 (as described in Section 4.1.3, see Fig. 11), certain properties need to be added to the object that stores ray information (the `RayObject`). This includes the contribution of the ray on the final pixel color. This cannot be fully determined while tracing the individual rays.

The parent's color depends on the material interaction at the hit-point plus any color returned by child rays. So once a parent ray is done tracing all child rays, we can set the contribution for all child rays with respect to the parent. This is important because if each ray knows its contribution with respect to its direct parent, we can recursively determine its contribution to the final pixel color.

To make rays transparent, they need new materials that are transparent instead of opaque. This is in general more computationally expensive than opaque rays. However, we noticed that if we remove all components except the ambient component of the transparent materials, it makes them actually faster to render than opaque rays, and the lack of diffuse and specular reflections on transparent rays is barely noticeable. We have added two options to the ray tracer properties panel with which the user can control ray transparency: a toggle to enable ray transparency and a slider to set the level of transparency. If ray transparency is enabled, the `RayManager` will assign transparent materials to the rays instead of opaque materials. Every ray gets a unique material as the level of transparency depends on both the ray's

[3] https://docs.unity3d.com/ScriptReference/Physics.Raycast.html

[4] https://docs.unity3d.com/ScriptReference/Object.Destroy.html

[5] https://docs.unity3d.com/ScriptReference/Object.Instantiate.html

contribution to the pixel's color and how transparent the user wishes the rays to be.

To handle dynamic ray radii, we give the user three extra controls: a toggle to enable dynamic ray radius and two sliders for the minimal and maximal ray radius. If the user wishes the ray's radius to be dynamic, the aforementioned function returns a radius between the minimal and maximal radius based on the ray's contribution. Otherwise it returns the `rayRadius`. The way this is implemented gives the user another unique way of visualizing the rays, as the minimal ray radius does not have to be smaller than the maximal radius. If this is not the case, negligible rays are simply bigger than important rays, which gives the user the ability to easily find rays that contribute almost nothing.

The ray tracer already stores the color that each ray contributes to the pixel in each `RayObject`. Just like the `RayManager` can return a unique transparent material for each ray, it can also give a uniquely colored material for each ray, optionally also transparent. We add yet another toggle to the ray tracer properties panel to enable contribution-based ray colors. If this toggle is enabled, the `RayManager` will not return a material based on the `RayObject`'s type, but on the color it contributes to the pixel.

## A.4. Acceleration structures

The two acceleration data structures that we have incorporated into VRT 2.0 are axis-aligned bounding boxes (AABB; Section 3.2.1) and octrees (Section 3.2.2). They are utilized in levels (two for each) dedicated to acceleration as described in Section 3.2. In the tool, the ray traced objects within the scene are `RTMesh` objects with a `MeshRenderer` component, via which the bounds property is accessible as a `Bounds` class object. This gives us the AABB and also the root node of the octree of the object.

In order to utilize the AABB to accelerate ray tracing, we need to check whether a ray intersects the AABB before checking for intersection with the object itself. We also keep track of the number of AABB misses so that this information can be conveyed to the user. Compared to the AABB, the calculations using an octree are more complex. To make the abstraction clearer, we implemented three classes: `Octree`, `OctreeRoot`, and `OctreeNode`. The root node is basically the AABB itself, which is then recursively subdivided into nodes until a certain depth limit has been reached or the processed nodes are empty.

In both cases, the visualization (see Fig. 12) is handled by the Popcron Gizmos package [35], which allows for run-time drawing. To offer feedback to the user, there are three visualization components implemented in order to visualize how the AABB accelerates the ray casting: acceleration status bar; number of rays ignored bar; hitpoint sphere on the AABB. The acceleration status bar indicates to the user if the ray has been ignored, has hit the AABB, or has hit the object or not. It updates as the object or camera is moved around by the user. The number of rays ignored bar is responsible for displaying the number of rays that were not considered while rendering an image. Finally, the hitpoint sphere is seen on the AABB as a green dot only when the ray intersects the AABB. This helps with showing that the ray intersects the AABB, even though the location of the intersection point is not necessarily important for acceleration.

## A.5. Lighting

As described in Section 4.1.2, VRT 2.0 supports two new types of lights. The first step to this end was to generalize lights. While some light properties are shared (position and color), some, such as rotation, are not. To this end, we have introduced an abstract base class `RTLight` from the `RTPointLight` class, made the `RTPointLight` class extend the `RTLight` class, and moved all properties to the base class and left only point light specific properties in the `RTPointLight` class. We also introduced a new enum field `RTLightType` so when lights are accessed, their type can always be determined.

### A.5.1. Area lights

Unity's rendering pipeline used for VRT does not support real-time area lights. This is not an issue as we want to have direct control over the sampling of area lights for education and visualization purposes. Hence, we use a number of Unity's spot lights to approximate an area light. We first created an `RTAreaLight` object in Unity, a prefab that has an image of a rectangle.

Next, we introduced a `lightSamples` variable and make the script distribute Unity's spot lights over the area light's area. The `RTAreaLight` object itself has a position, rotation, and scale. If we add sub-objects, so-called children, they are also impacted by these properties. So if the `RTAreaLight` prefab is a 1-by-1 rectangle with the front-facing along the positive $z$-axis, we can simply place spot lights accordingly in this rectangle and they will automatically be positioned and rotated correctly in the scene. We uniformly distribute a total number of `lightSamples` $*$ `lightSamples` Unity spot lights over the area (see Section 3.1.1), make them face along the positive $z$-axis and they will illuminate the scene correctly.

The next step is to incorporate this in the ray tracer. We use jittered sampling to approximate the area light's illumination. With jittered sampling, we divide the area light's area in `lightSamples` $*$ `lightSamples` equally sized rectangles and take a random point within each rectangle. So when we ray trace an area light, we take a total of `lightSamples`$*$`lightSamples` samples from the area light and treat each sample exactly like we would treat a spot light, except this time we divide their intensity by the number of samples.

When $5^2 = 25$ or more samples are used and they all reach the light source, we combine them into a single area-ray, see Fig. 8. This reduces the number of rays significantly and improves performance. This also means that we had to change the `RayObjectPool` to support two different types of ray objects. As this area-ray is a big object, it is always transparent.

### A.5.2. Spot lights and attenuation

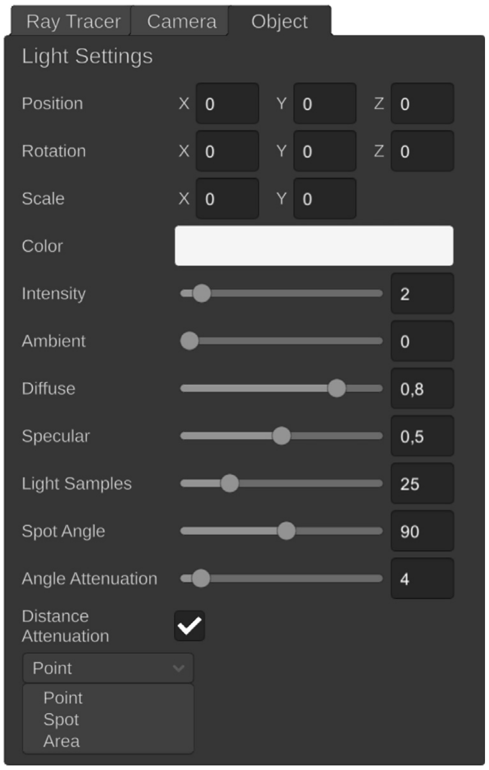
To model spot lights, we simply combined features from the point and area lights as detailed above. We opted to use $0.04 + 0.1d + 0.06d^2$ for distance $d$ attenuation. (Note that the typical scene's bounding box diameter is in the order of tens of units.) Light objects have their own properties panel that the user can access by selecting the light, and in order to better visualize the difference this makes, we added a toggle to this panel to enable/disable light-distance attenuation per light.

Angle attenuation only applies to the spot and area lights. As usual, we model this kind of attenuation by multiplying the intensity by $\cos(\alpha)^p$, where $\alpha$ is the angle between the light's direction and the direction to the point from the light, and $p$ is a non-negative exponent controllable by the user.

These two attenuation methods decrease a light's intensity, but there was no option to increase it. We allow for increasing the intensity of each light by simple multiplication via a slider in the light object properties panel. A complete overview of all the light properties can be found in Fig. 20.

## A.6. Gamification

The step-by-step tutorial is created as a Prefab with a Tutorial Manager script attached to it, and added to the Main Canvas. Tasks for each level are stored in a list along with helper functions


Ray Tracer | Camera | Object

Light Settings

| | | |
|---|---|---|
| Position | X 0 | Y 0 | Z 0 |
| Rotation | X 0 | Y 0 | Z 0 |
| Scale | X 0 | Y 0 | |
| Color | | | |
| Intensity | 2 | | |
| Ambient | 0 | | |
| Diffuse | 0,8 | | |
| Specular | 0,5 | | |
| Light Samples | 25 | | |
| Spot Angle | 90 | | |
| Angle Attenuation | 4 | | |
| Distance Attenuation | ✓ | | |

Point

Point
Spot
Area

**Fig. 20.** A screenshot of all possible light properties in VRT 2.0. Some properties are not visible or available for certain light types.

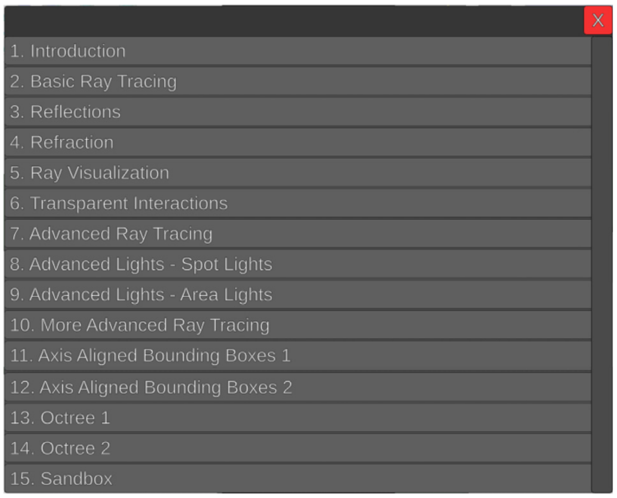

1. Introduction
2. Basic Ray Tracing
3. Reflections
4. Refraction
5. Ray Visualization
6. Transparent Interactions
7. Advanced Ray Tracing
8. Advanced Lights - Spot Lights
9. Advanced Lights - Area Lights
10. More Advanced Ray Tracing
11. Axis Aligned Bounding Boxes 1
12. Axis Aligned Bounding Boxes 2
13. Octree 1
14. Octree 2
15. Sandbox

**Fig. 21.** The complete list of levels available in VRT 2.0.

to, for example, increment the current level's task index. These tasks are then stored inside the Game Manager, so we can store the progress for each level when switching between levels (their complete list is shown in Fig. 21). This way, the Tutorial Manager is also able to retrieve the current task information and update the tutorial's UI accordingly.

Whenever a user has to complete a task by executing an action, we need some way of associating this action with the task. We do this by adding the same identifier to the task and the action. When a user executes a specific action (e.g. clicks on the Help button), we send this identifier to the Tutorial Manager. If the identifier matches the current task's identifier, the task is completed and the UI is updated.

Badges are stored in a list in the Game Manager with each badge containing a type, name, description, icon, and a number that determines when a badge is earned. In the badge collection menu, we simply create all available badges listed in the Game Manager's badge list. A badge that has not been earned yet is indicated by a disabled-looking state and barely readable text. In the badge notification menu, we continuously loop over each badge and check whether they have been earned since our last check. If so, the user receives a notification displaying the newly earned badge.

### *A.7. Web and mobile*

Unity supports porting an application to WebGL [23], a 3D graphics library/API which allows browsers to efficiently render 3D scenes. The rendering is client-based; the scene is usually downloaded from a server, after which the processing of the scene is done locally using the client's hardware [36]. The web build is available at [22]. The web port worked out of the box, and, unlike the mobile port, required no UI or other changes.

Porting to Android [21] also in principle worked directly, but it required several usability changes, as detailed in Section 4.2.1. As already mentioned, it is important to note that while the web version is based on VRT 2.0, the mobile version is, due to performance and screen-space limitations, based on the original VRT tool [5].

## References

[1] Haines E, Akenine-Möller T, editors. Ray tracing gems. Apress; 2019, http://raytracinggems.com.

[2] Marrs A, Shirley P, Wald I, editors. Ray tracing gems II. Apress; 2021, http://raytracinggems.com/rtg2.

[3] Marschner S, Shirley P, editors. Fundamentals of computer graphics. CRC Press; 2021.

[4] Deng Y, Ni Y, Li Z, Mu S, Zhang W. Toward real-time ray tracing: a survey on hardware acceleration and microarchitecture techniques. ACM Comp Surv 2017;50(4):58:1–41.

[5] Verschoore de la Houssaije WA, Wezel CSv, Frey S, Kosinka J. Virtual ray tracer. In: Bourdin J-J, Paquette E, editors. Eurographics 2022 - education papers. The Eurographics Association; 2022, p. 45–52.

[6] Simons G, Herholz S, Petitjean V, Rapp T, Ament M, Lensch H, et al. Applying visual analytics to physically based rendering. Comput Graph Forum 2019;38(1):197–208.

[7] Simons G, Ament M, Herholz S, Dachsbacher C, Eisemann M, Eisemann E. An interactive information visualization approach to physically-based rendering. In: Vision, modeling & visualization. The Eurographics Association; 2016.

[8] Spencer B, Jones M, Lim I. A visualization tool used to develop new photon mapping techniques. Comput Graph Forum 2015;34(1):127–40.

[9] Zirr T, Ament M, Dachsbacher C. Visualization of coherent structures of light transport. Comput Graph Forum 2015;34(3):491–500.

[10] Smyk M, Szaber M, Mantiuk R. JaTrac — an exercise in designing educational raytracer. In: Advanced computer systems: eighth international conference. Springer; 2002, p. 303–11.

[11] Vitsas N, Gkaravelis A, Vasilakis A, Vardis K, Papaioannou G. Rayground: an online educational tool for ray tracing. In: Romero M, Sousa Santos B, editors. Eurographics 2020 - education papers. The Eurographics Association; 2020.

[12] Russell J. An interactive web-based ray tracing visualization tool [Undergraduate Honors Program Senior thesis], Department of Computer Science, University of Washington; 1999.

[13] Gribble C, Fisher J, Eby D, Quigley E, Ludwig G. Ray tracing visualization toolkit. In: Proceedings of ACM SIGGRAPH. ACM; 2012, p. 71–8.

[14] Cook RL, Porter T, Carpenter L. Distributed ray tracing. SIGGRAPH Comput Graph 1984;18(3):137–45.

[15] Shirley P, Marschner S. Fundamentals of computer graphics. 3rd ed USA: A. K. Peters, Ltd.; 2009.

[16] Fujimoto A, Tanaka T, Iwata K. ARTS: accelerated ray-tracing system. IEEE Comput Graph Appl 1986;6(4):16–26.

[17] Reiners T, Wood LC. Gamification in education and business. Cham, Switzerland: Springer; 2015.

[18] Nicholson S. Strategies for meaningful gamification: Concepts behind transformative play and participatory museums. Mean Play 2012;1999:1–16.

[19] Meister D, Ogaki S, Benthin C, Doyle MJ, Guthe M, Bittner J. A survey on bounding volume hierarchies for ray tracing. Comput Graph Forum 2021;40(2):683–712.
[20] Phong B. Illumination for computer generated pictures. Commun ACM 1975;18(6):311–7.
[21] SVCG. Virtual ray tracer. 2022, Google Play, https://play.google.com/store/apps/details?id=com.RUG.VirtualRayTracer.
[22] van Wezel C, Verschoore W. Virtual ray tracer (web version). 2022, URL: https://wezel.github.io/Virtual-Ray-Tracer/.
[23] Documentation U. Building and distributing a WebGL project. 2022, https://docs.unity3d.com/Manual/webgl-building-distribution.html.
[24] Documentation U. Porting a project between platforms. 2022, https://docs.unity3d.com/520/Documentation/Manual/HOWTO-PortingBetweenPlatforms.html.
[25] Hennessey B, Moran S, Altringer B, Amabile TM. Extrinsic and intrinsic motivation. In: Wiley encyclopedia of management. John Wiley & Sons, Ltd; 2015, p. 1–4.
[26] Deci EL, Ryan RM. Self-determination theory. In: Handbook of theories of social psychology: volume 1. 1 Oliver's Yard, 55 City Road, London EC1Y 1SP United Kingdom: SAGE Publications Ltd; 2014, p. 416–37.
[27] van Wezel C, Verschoore W. Virtual ray tracer. 2022, URL: https://github.com/wezel/Virtual-Ray-Tracer.
[28] van der Zwaag J. Virtual ray tracer: distribution ray tracing [BSc thesis], University of Groningen; 2022, https://fse.studenttheses.ub.rug.nl/27881/.
[29] Yilmaz B. Acceleration data structures for virtual ray tracer [BSc thesis], University of Groningen; 2022, https://fse.studenttheses.ub.rug.nl/27838/.
[30] Blok PJ. Gamification of virtual ray tracer [BSc thesis], University of Groningen; 2022, https://fse.studenttheses.ub.rug.nl/27596.
[31] Rosema R. Adapting virtual ray tracer to a web and mobile application [BSc thesis], University of Groningen; 2022, https://fse.studenttheses.ub.rug.nl/27894.
[32] Bredenbals A. Visualising ray marching in 3D. Master's internship report, University of Groningen; 2022, https://fse.studenttheses.ub.rug.nl/27977/.
[33] Whitted T. An improved illumination model for shaded display. Commun ACM 1980;23(6):343–9.
[34] Schlick C. An inexpensive BRDF model for physically-based rendering. Comput Graph Forum 1994;13(3):233–46.
[35] popcron. gizmos ReadMe. 2019, https://github.com/popcron/gizmos/blob/master/README.md.
[36] Cantor D, Jones B. WebGL beginner's guide. Birmingham, UK: Packt Publishing; 2012.