15 Years Later: A Historic Look Back at *"Quake 3: Ray Traced"*

Daniel Pohl
Intel Corporation,
Konrad-Zuse-Bogen 4,
Krailling, Germany
daniel.pohl@intel.com

Selvakumar Panneer
Intel Corporation,
2111 NE 25th Ave,
Hillsboro, OR, USA
selvakumar.panneer@intel.com

Deepak S. Vembar
Intel Corporation,
2111 NE 25th Ave,
Hillsboro, OR, USA
deepak.s.vembar@intel.com

Carl S. Marshall
Intel Corporation,
2111 NE 25th Ave,
Hillsboro, OR, USA
carl.s.marshall@intel.com

***Abstract*—Real-time ray tracing has been a goal and a challenge in the graphics field for many decades. With recent advances in the hardware and software domains, this is becoming a reality today. In this work, we describe how we got to this point by taking a look back at one of the first fully ray traced games: *"Quake 3: Ray Traced"*. We provide insight into the development steps of the project with unreleased internal details and images. From a historical perspective, we look at the challenges pioneering in this area in the year 2004 and highlight the learnings in implementing the system, many of which are relevant today. We start by going from a blank screen to the full ray traced gaming experience with dynamic animations, lighting, rendered special effects and a simplistic implementation of the gameplay with basic AI enemies. We describe the challenges encountered with aliasing and the methods used to alleviate it. Lastly, we describe for the first time the unofficial continuation of the project, code named *"Quake 3: Team Arena Ray Traced"*, and provide an overview of the changes over the past 15 years that made it possible to generate fully ray-traced interactive gaming experiences with mass market hardware and an open software stack.**

***Index Terms*—ray tracing, computer**

I. Introduction

Real-time ray tracing has been a dream for computer graphics programmers for many decades. By physically simulating how light interacts with surfaces, ray tracing can produce outputs that can closely resemble how light illuminates surfaces in the real world. The key challenges that were faced in bringing ray tracing to real-time was the need for enormous data parallel computing power and memory bandwidth combined with advances on the algorithmic side. While GPUs were primarily designed to accelerate rasterization, many explored re-purposing the thousands of general-purpose execution units in GPUs to perform ray tracing in real-time. The introduction of highly parallel programming languages like CUDA, OpenCL and Direct Compute enabled developers to explore a sub-set of real-time ray tracing techniques on the GPU such as indirect lighting, approximated global illumination and environment effects. In 2018, Microsoft officially announced the graphics API *DirectX Raytracing (DXR)* [1] which unified GPU vendors to support a dedicated GPU-accelerated ray tracing pipeline. This led to games taking advantage of GPU-accelerated ray tracing to perform real-time global illumination, environment effects such as reflections, refractions and shadows at a quality level which is much closer to realism than mimicking these effects using the rasterization pipeline.

Figure 1. A blue light source is placed inside the quad damage item. Pixel-accurate real-time shadows are ray traced.

We want to use this moment to take a look back at the origins of ray tracing in games. We do this by going through a project which one of the authors of this paper developed during a bachelor's thesis at the University of Erlangen together with the Saarland University in 2004. At this time, we used the *Quake 3* game content from *id Software* in a novel game engine written from scratch together with a newly available real-time ray tracing library. The project was named *"Quake 3: Ray Traced"*, or Q3RT in short. A rendered image from it is shown in Figure 1. We share the challenges, learnings and benefits with many previously unreleased details, images and benchmarks.

Besides the historical documentation, we provide insights into what researchers and developers might encounter today in a similar matter with DirectX Raytracing as we did already one and a half decades ago. Furthermore, we disclose details about an unofficial and so far undocumented continuation of the project code named *"Quake 3: Team Arena Ray Traced"*.

In the following sections, we will start with the related work up until 2004 which was required to have the Q3RT project started. In chapter III, we give a short overview of the used OpenRT ray tracing library. This is followed by a description of our single PC hardware and a clustered network setup that were used during development. Chapter V outlines the motivation for starting this project. We continue with the first steps that describe the initial software setup and how rendered images were displayed. In chapter VII, we add static geometry from the game level into the engine. Following, different types of light sources with hard and soft shadows are discussed. In chapter IX, we add dynamics like texture animations, decals and player models to the ray traced game. In the section afterwards, we describe special effects like reflections, refractions, camera portals, ground fog and colored shadows and how they were enabled with ray tracing. We move on to chapter XI and describe the encountered issues with aliasing and which methods we applied against it. Next, we give details about the, so far, undocumented project continuation *"Quake 3: Team Arena Ray Traced"* with a ray traced water implementation and a test setup with one million reflecting spheres. Chapter XIII describes the achieved performance with the setup from 2004. After this, we share our learnings from the project. Before we finish with the conclusion, we provide a quick summary of what happened after the project until today and give a short outlook on potential future rendering architectures.

II. Related Work before Q3RT

The question of who invented ray tracing has been analyzed in an article by Hofmann [2] in 1990: going back to even before computers existed, famous artists like Leonardo da Vinci and Albrecht Dürer used between the years of 1480 and 1528 true perspective projections in their drawing. While this was done with paint instead of pixels in a frame buffer, these paintings could be seen as the first renderings with ray tracing.

Moving over to the electronic form of rendering, Appel [3] used in 1968 rays in the domain of computer graphics. In 1979, Whitted [4] used ray tracing for colorful renderings including simulations of rendered glass. Creating a single image with a resolution of 640×480 pixels and a total color depth of nine bits took between 45 and 120 minutes at that time.

In 1984, Cook et al. [5] described distributed ray tracing, allowing effects like blurred reflections, translucency, depth of field and motion blur. Glassner [6] optimized ray tracing for animations by using modified bounding volume hierarchies (BVH). Starting in 1995 with the military work from Muuss and Lorenzo [7], networked computers were used together to speed up ray tracing calculations. Parker et al. [8] scaled their isosurface rendering system across multiple shared-memory processors. Wald et al. [9] optimized ray tracing in 2001 for the usage of CPU-based SIMD vector extensions, offering another dimension of parallelization. A combination of scalable ray tracing across processors, SIMD and networked computers with flexibility in the rendering pipeline for different primitives and animated content was wrapped around the OpenRT ray tracing API [10], [11].

In 2004, the OpenRT rendering library was made available to us for the *"Quake 3: Ray Traced"* (Q3RT) project. During that time, the student Tim Dahmen was also exploring the usage of ray tracing in games with a newly created game named *"Oasen"* [12] in which an outdoor environment could be explored using a virtual magical flying carpet.

Parts of the work on the *"Quake 3: Ray Traced"* project have been written up in the bachelor's thesis of the student [13] in 2004, followed by a short summary in Schmittler et. al [12]. For our paper, we go in deeper with more details and with a retrospective look on that work with today's knowledge.

III. OpenRT

In 2002, OpenRT ray tracing library development was started at the Saarland University. At this time, OpenGL [14] programming was very commonly taught at universities and was the only available cross-platform low-level graphics API for the most common desktop and server computer systems. To let developers quickly adapt, one of the goals of OpenRT was to make the API as similar to OpenGL as possible.

OpenGL, at this time available in the version 1.5, had the option of using an immediate rendering mode. A triangle could be drawn as simple as in this code snippet:

```
glBegin(GL_TRIANGLES);
  glVertex3f(0, 0, 0);
  glVertex3f(0, 1, 0);
  glVertex3f(1, 1, 0);
glEnd();

glSwapBuffers();
```

Listing 1. Drawing a triangle in OpenGL 1.5

Many ray tracers, including OpenRT, were using acceleration structures like kd-trees or bounding volume hierarchies to store geometry. Therefore, the immediate rendering mode was not an available design option. Instead, a system comparable to OpenGL's display list was used where an object is described once and can later be instantiated.

```
int triangleObjID = rtGenObjects(1);
rtNewObject(triangleObjID, RT_COMPILE);
rtBegin(RT_TRIANGLES);
  rtVertex3f(0, 0, 0);
  rtVertex3f(0, 1, 0);
  rtVertex3f(1, 1, 0);
rtEnd();
rtEndObject();

int instID = rtInstantiateObject(triangleObjID);
rtSwapBuffers();
```

Listing 2. Drawing a triangle in OpenRT

For programmable shading, the OpenRTS ray tracing shading language was provided. It used a C++ interface with the

possibility of creating new rays during shading, e.g. for tests regarding shadows or for tracing reflections.

IV. Development System

The main development system used at that time was a PC with a single-core Pentium 4 (code named "Northwood"), clocked at 2.66 GHz with 768 MB memory. Due to the relatively low performance of that system, many parts of the interactive development were done in a rendering resolution of either 64×64 or 128×128 pixels as shown in in Figure 2. By just pressing a key on the numpad, it was possible to decrease or increase that resolution on the fly. For analysis of correctness in higher resolutions, an offline-calculated rendering in higher resolution with supersampling was created through a key shortcut. To test animations, videos were rendered over night in higher resolution. While the work was targeting future interactive ray tracing in games, the development on a single-core machine was sometimes rather limited.

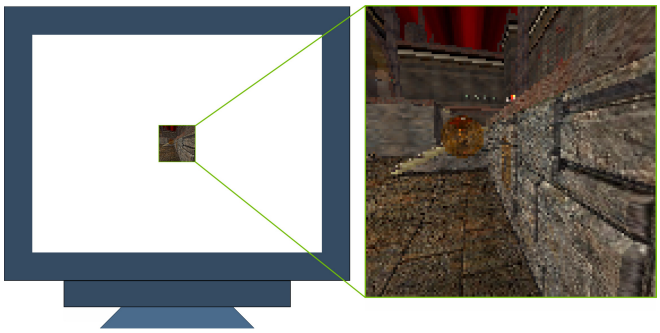

Figure 2. To give an understanding about the size of the interactive rendering resolution of 128×128 pixels, we show on the left such an image on a CRT monitor with a screen resolution of 1024×768 pixels. On the right, that image is enlarged and shows the pixelated content.

The true interactivity at higher resolutions came when using the PC cluster network at the Saarland University: 20 nodes with a dual processor system equipped with AMD MP 1800+ CPUs, clocked at 1.533 GHz, interconnected with 100 Mbit Ethernet. Due to the distance between the Q3RT development location and the cluster, this system was only used twice.

V. Quake 3: Ray Traced

The idea for researching the applicability of ray tracing for games emerged during a guest lecture from Professor Slusallek from the Saarland University. He presented recent advances in bringing ray tracing from an offline algorithm to real-time. The vision of getting this rendering fidelity combined with one of the most popular first-person shooters at that time led to the previously mentioned bachelor's thesis to research real-time ray tracing for games.

In 2004, Quake 3 was still one of the best-looking computer games. It had an advanced shading system, offered vertex animation for player models, curved surfaces, decals, volumetric fog, portals and many more features. It came with level editors and some of the internal file formats were unofficially documented on the Internet. This made it an exciting choice to investigate its applicability to real-time ray tracing.

The full game offered 30 levels. For the relatively short time frame of nine month for the bachelor's thesis, it made sense to limit the complexity by focusing on one of the levels exclusively. The level "q3dm7", short for "Quake 3 Deathmatch level number 7", was used. It is one of the larger levels with different interesting areas. Some experiments were still done with other levels which are shown in the Appendix.

VI. First steps

Because the source code of the original Quake 3 game was not available in 2004 yet, the Q3RT project was created from scratch in an empty Visual Studio 6.0 project. Libraries were used to quickly make progress: most importantly OpenRT for ray tracing as described before. For creating the display window, key inputs and mouse interactions libSDL was used.

The ray tracing frame buffer itself was an array in local PC memory, consisting of 32-bit RGBA data per pixel. The alpha channel was not used, but provided CPU-friendly data alignment. To display the frame buffer on the screen, for every rendered frame an OpenGL texture was created (or an existing one was modified with *glTexSubImage2D*). In 2004, many GPUs still had the limitations that textures had to be in a resolution of $2^n \times 2^m$. While with workarounds different aspect ratios could have been used as well, it was the easiest way to use resolutions like 128×128 in a 1:1 aspect ratio for this work. Once the texture has been updated, it was displayed on an OpenGL quad on the screen.

We learned that HUD, cross hair, and overlays in general were of higher quality and faster rendered on top of the image through the traditional OpenGL pipeline. They were never present in the ray traced world description where as 3D objects they might have interfered with ray intersections.

VII. Static Content

The first geometry to test if rendering works was just a single triangle. Once this was ray traced and showed up correctly through the OpenGL texture, the next step was to include a player camera model to simulate the WASD-movement known from first-person shooters. After completion, loading of the static Quake 3 geometry was addressed. The level files are stored in a .bsp file format which was unofficially documented on the Internet. The file extension .bsp hints at the usage of a binary space partitioning (BSP) tree [15] to optimize rendering on the GPU. The tree nodes contain information on the splitting plane within this volume, the bounding box of this volume and indices to the children. In the tree leaf nodes, information about the potentially visible sets [16] (PVS) were stored. The original rasterized rendering algorithm would determine in which leave node it currently is and then know which other nodes need to be rendered to not miss any geometry. The generation of the BSP tree with the PVS was done in a compiling step during level creation. To optimize performance further, professional level designers needed to place manually brushes into the scene in

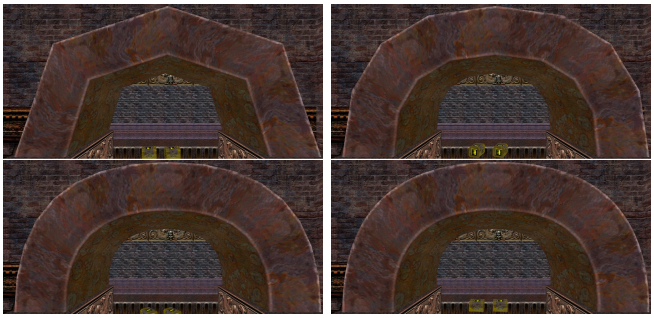

Figure 3. Top left shows the beziér patch at a resolution of 32 triangles. Top right uses 128, while bottom left has 512 and bottom right 7200 triangles.

the level editor with a hint for the PVS system to not continue to render beyond this volume.

Here is already an interesting difference between the original version for rasterization and engines using ray tracing. For Q3RT, we were not using any of the PVS data. It would even have been wrong to try to use it. For global effects like a reflection that bounces around, the PVS would not provide the full geometry that might be required to trace this ray.

Instead, we put the full level geometry into a single object for OpenRT. It internally built its acceleration structure for ray tracing on it. No further hints were required as the culling of other geometry happens implicitly on a per-ray level through the acceleration structure.

The number of triangles for the level q3dm7 is 20,000 which 15 years later appears very small and it surprises how much detail artists were getting out of this given the constraints. Besides the geometry we extracted so far, there is one more area which can increase the number of triangles. Quake 3 was one of the first games that supported rendering of beziér patches. Through changing parameters in the original game in the internal console, a different number of triangles were used for creating smoother curved surfaces. There are a total of 189 beziér patches in the level q3dm7.

For rendering curved surfaces like beziér patches, ray tracing has usually a great advantage [17]. A hit volume can be defined specifically for these patches. When tracing a ray inside such a volume, the mathematical equation can be used to calculate the exact hit point. This means that the visualization is always as smooth as possible and no single triangle-based surface becomes visible when looking very closely at these objects. However, this functionality was not yet supported by OpenRT in 2004. Therefore, we fell back to manually calculating the patches as triangle meshes during level loading. We tried various detail levels and measured the impact on the performance by adding additional geometry for the beziér patches on top of the 20K triangles of the level. We discuss the performance results in detail in section XIII. Renderings of the different beziér patch resolutions can be seen in Figure 3.

If a ray did not hit any target, the color was set to black. In DXR, this is now known as a miss shader. In game levels there is often a surrounding of a large skybox with cubical texture mapping on it. While the player moves through the

Figure 4. Level q3dm7 with all textures and the red, cloudy sky sphere. At this initial stage, there was only ambient lighting used.

scene, the sky is so far away that it appears fixed. In our case, if a ray hits outside the level, we used a sky sphere shader from OpenRT. Given the direction of the ray that does not hit any geometry, we calculate a texture offset into a spherical texture. The result is comparable to a skybox and shown in the red, cloudy area in Figure 4.

VIII. Lighting

Enabling dynamic lighting through ray tracing and having accurate shadows was one of our key goals. To achieve this, OpenRT and Quake 3: Ray Traced supported different types of light sources. The most relevant ones with their adjustable parameters were:

- *point lights:* position, RGB intensity, attenuation
- *directional lights:* direction, RGB intensity, attenuation
- *spot lights:* position, direction, RGB intensity, attenuation, falloff angle

The original Quake 3 game uses hundreds of *point light* sources in the level q3dm7. However, during compile time of the map, the illumination of these is baked into static light maps [18]. While this looks reasonably well, it does not allow for fully flexible dynamic illumination. Therefore, in Q3RT, we did not use the light map data, but enabled some of the point lights manually in the scene.

The *directional light* fits very well for simulating a distant light source on top of the level like the sun.

Quake 3 in its original form does not use *spot lights*. However, we added a flash light option for the player. From the estimated hand position, a spot light was used facing forward into the viewing direction of the player. At this time, such

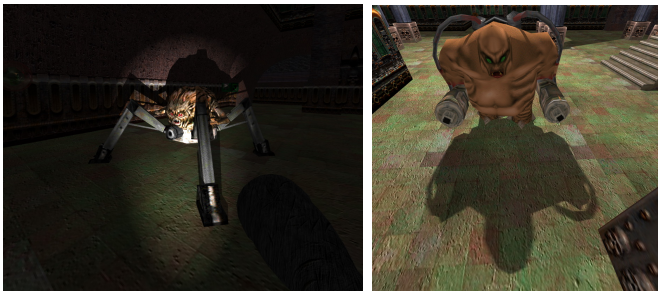

Figure 5. Left: flash light including casted hard shadows. Right: soft shadows.

Figure 6. Epsilon issues during casting a shadow ray. The ray self-intersects with the surface to be shaded and produces a flickering moiré pattern.

an effect of dynamically lighting the scene with a flash light including real-time shadows casted from it was not available to gamers yet. An example rendering is given in Figure 5 left. Such an effect was becoming available to gamers shortly after this work finished: in August 2004, the game Doom 3 was released. In it, the shadow volume [19] algorithm was used to enable such a flash light effect including real-time shadows.

"Where there is light, there must be shadow" is a quote from Haruki Murakami [20]. One very interesting aspect of ray tracing is the shadow generation. When we shade a surface, we can shoot a shadow ray to a light source and check if the ray reaches there or if it is blocked. In the first case, we illuminate the surface with the light properties, in the second we do not change the lighting at his point, which leads to a shadow. The implementations used at this time were not optimized well for supporting a multitude of lights. Even if one light was very far away and clearly out of reach to illuminate a surface, the shadow ray was still cast to the distant light source. The previously mentioned project *"Oasen"* solved this more elegantly by defining a bounding volume for the light source on which it can impact other geometry. By doing a check first against this bounding volume, hundreds of local light sources were efficiently used.

Shooting a single shadow ray to a light source leads to a hard shadow with a pixel-exact border at the surface. However, in the real world, most scenarios do not have such a hard shadow. Still, in 2004, this was a desirable result and equals the hard shadows that Doom 3 did in the same year with the shadow volume algorithm. One experiment in the Q3RT project involved shooting six shadow rays with an offset around the center of the light instead of a single one for a point light source. While this had certainly a much higher impact on performance, the results looked visually more pleasing compared to hard shadows. Soft shadows are shown in Figure 5 right.

As easy as it was to get shadows working in ray tracing compared to implementing various shadow mapping algorithms [21]–[23] with their specific optimizations or using shadow volumes, there is also a drawback. When the shadow ray is cast from the surface to be shaded, it needs to be offset by a certain epsilon value in its ray direction to avoid hitting the same surface again due to floating point inaccuracies. As the q3dm7 level spans from very large geometry to smaller models with their own internal acceleration structures, it did happen that the chosen epsilon value worked in one case but not another. We show this in Figure 6. While it seems this issue has in practice often been tweaked and afterwards ignored as a real problem, there is now in 2019 a publication on avoiding self-intersection during ray tracing from Wächter and Binder [24]. Another elegant way to avoid these issues would have been to store in the shadow ray the surface ID of the previous hit and exclude it during intersection tests later.

IX. Dynamic Content

In 2004, the thoughts on real-time ray tracing were that it is not suited for dynamics. While the term "dynamics" goes into different dimensions, as we will show in this chapter, the reference was made towards dynamically updating geometry as this would require a costly rebuild of the acceleration structures used for ray tracing.

The way Q3RT was setup as described up until here is having static geometry and some light sources, most of them fixed except the flash light. At this stage, the impressions of the renderings are a lifeless scenario like being the only person in a virtual environment with no changes at all over time. To counter this and make a lifelike impression as in the original Quake 3 game, various stages are involved in adding dynamics.

Quake 3 was one of the first games with its own shading language [25]. This allowed various effects on otherwise flat textures. For example, with just a few lines of shading code texture animations were created. Often, the texture coordinates were modified over time before sampling. This could be combined with blending multiple textures together to create the perception of a dynamically changing environment. For the level q3dm7, we ported these effects over into the shading language OpenRTS.

Next, we added support for decals. These are used in the game to dynamically add sprites or animation effects into the scene. One example is when shooting the virtual machine gun and hit decals show up at the environment. Those stay for a few seconds and then disappear, to not overload the rendering over time. The implementation is often in the form of a textured quad with transparencies around the effect area.

One interesting property of OpenRT is that once geometric objects are instantiated, they will remain automatically in all future frames unless they are manually removed. As shown in

the Appendix in Figure 18, one can easily "paint" the level by adding decals and never remove them. Of course, deletion of old objects was added afterwards into the rendering engine. From the ray tracing side, if a ray hits the quad with some transparent areas in it, it will shoot another ray behind that quad into the same direction. This can again lead to the previously mentioned epsilon issues regarding self-intersection.

The next dynamic objects are player models. In Quake 3, the animation for these are stored as pre-calculated keyframes. During rendering in the original game, the closest key frame according to the in-game time index is determined. The next closest is taken as well and the geometry is linear interpolated between those. In Q3RT, we tried this as well, but the frequent rebuilding of the acceleration structure of the player model was too costly and lowered the frame rate by a factor of 7 to 10. Therefore, during loading of Q3RT, we created a separate object for all possible keyframes of the player model. These were between 200 and 300 poses with around 1500 triangles. The acceleration structures were built during loading. In-game, we determined the closest key frame and instantiated it. In the next frame, we deleted the old instance and created the updated one. Figure 19 in the Appendix shows what happens if that deletion step is not used.

Using this method for player models meant that our animations were not as smooth as in the original game if both would be compared side-by-side at high frame rates. Newer approaches in the years afterwards provided much faster BVH buildings and therefore the option to just rebuild the interpolated player model at anytime. Other research was going towards mapping skeletal animation technologies directly into the acceleration structures [26]. Furthermore, refitting of BVHs became a viable method instead of a full rebuild [27], [28].

While the original Quake 3 game has a complex AI [29] system for moving the player models around, we took a simpler approach for Q3RT. Pre-defined way points were used for player models to move around in the levels. When a player model came too close to an AI-driven agent, the AI agent started firing into the direction of our own player. A basic game play logic was added with player health and damage given by various weapons and their projectiles.

X. Special Effects

With the combination of static and dynamic geometry, animation support and proper lighting, we explored several new special effects that were not in the original game and would be very hard to achieve without using ray tracing.

Reflections are very easy to use in a renderer with ray tracing. When a ray hits a reflecting surface, a new ray will be cast into the reflected direction, depending on the incoming angle and the normal of the hit surface. Besides showing the reflected color on the object like on a perfect mirror, there can also be a texture on the reflecting object which gets mixed together with the reflection. One of these examples is shown in Figure 7 left with the ammunition box for which we added reflecting properties, but also kept the original yellow-colored texture.

Figure 7. Left: ammo box with reflections. Right: multiple-bounce reflections.

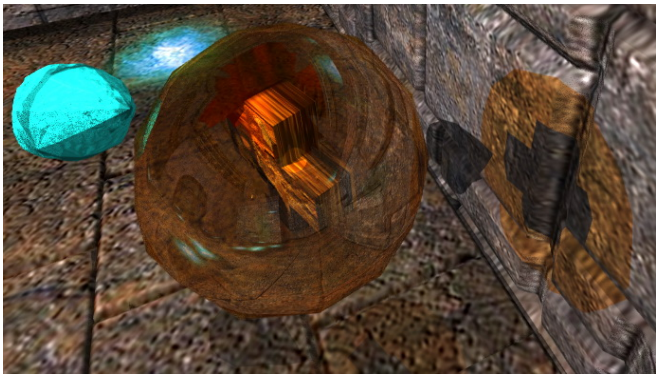

Figure 8. A glass shader on an orange sphere with reflections of the environment and refractions. Colored shadows on the wall on the right are cast through the glass using ray tracing.

As ray tracing works recursively through the shaders, there is no extra effort to be made by the developer to enable effects like a mirror inside a mirror including the multiple-bounced reflections that occur. While the concern might arise that these increase rendering time too much, it shall be noted that the impact is smaller than what people might expect. Because the reflection in the reflection is already decreasing in size compared to the original object, there are only very few rays that do a higher amount of multiple reflection bounces. Only these will require more performance, while the other ones are reasonably fast done with tracing. Other effects like lighting and shadows naturally work in the reflections as well without any additional development effort. An example of reflections in reflections can be seen in Figure 7 right.

Refractions and reflections together are found in glass. Ray tracers are known for the ability of rendering glass highly realistic. Besides the reflection ray on objects, a second ray for refraction gets traced as well, depending on the incoming angle and the refraction index of the surface or volume. An example is shown in Figure 8.

Camera portals are very easy to achieve with ray tracing. Once a ray hits the surface of the portal, another ray is set up. The new ray will have a positional offset and optionally a change in direction. It is traced from there and its shaded color value is used for the surface of the portal. This is shown in Figure 9 left. Even advancing to recursive effects showing portals in portals is possible with ray tracing at no extra development effort. Sample code for portals in OpenRTS:

Figure 9. Left: camera portal. Right: Ground fog through ray tracing.

```
Vec3D newOrigin = ray.hitPosition + portalOffset;

// shooting a new ray with the added offset
color = traceNewRay(newOrigin, ray.direction);
```

Listing 3. Creating the portal effect with the OpenRTS shading language

Ground fog is used in the original Quake 3 game. With ray tracing, there is also an easy method of enabling this. The ground fog is modeled in a non-visible volume. Once a ray hits this volume, it will prepare another ray. That second ray continues with a small offset into the same direction. After its shading color has been determined, the length of this ray will be used. It is put into an exponential function which determines a blending value between a fog color, like orange in Figure 9 right, and the original shaded color. This way, the further the ray traveled, the more fog will be applied.

Colored shadows cast by partially transparent objects can be done in ray tracing as well. In the regular, single-colored shadow casting with ray tracing, we test with a shadow ray if any object is blocking the path from the surface to be illuminated towards the light source. In case of decals with quads that have partially transparent pixels, we must already execute shading code on it to gather the information of transparency or opaqueness. In a scenario like shown in Figure 8, we give the shadow ray that tests the glass surface a color offset if it can reach to the light source through the medium. Even though the direct light from the surface to shade to the light source is blocked, the indirect light going through the glass will now contribute the new shadow color.

It shall be noted that for many of the described effects that require shooting a secondary ray like an additional reflection or refraction ray, the previously mentioned issues of self-intersection can happen. Two examples are shown in Figure 10.

XI. Aliasing and Image Quality

As typical for rendered content, aliasing can happen - this is the same if a ray tracer or a rasterizer is used. In the original Quake 3, trilinear texture filtering is used. Full scene anti-aliasing (FSAA) could be enforced through the graphics driver.

The version of OpenRT that was provided to us at the given time was limited to bilinear texture filtering. As a result, textured objects farther away were flickering much more compared to the original version. As a workaround, we implement shader-based trilinear filtering. First, the mipmaps [30] were created and made available as different textures into the shader.

Figure 10. Self-intersection issues with a glass shader on the left and on reflections on the right.

Figure 11. Visualization of mipmap levels based on distance.

Second, based on the distance of the original camera ray to the primary hit point, we determined the mipmap level. This is visualized with a color-coding in Figure 11. Last, with interpolation between the two closest mipmap levels, we achieved trilinear filtering. While this improved overall image quality, it shall be noted that this should usually be handled by the underlying ray tracing system instead of cluttering up the shading code. Furthermore, this way of using mipmaps is only a rough approximation. In the implemented form, it does not consider how large the surface is on which the texture is and what the texture and rendering resolution is. The correct way would be to use ray differentials [31] for this.

Full scene anti-aliasing was a commonly offered option in 2004 for rasterized games. It samples geometry in higher resolution, but shading is applied in the original rendering resolution. This is of course a trade-off between performance and image quality. For Q3RT, we experimented with supersampling [32]: multiple rays are shot instead of a single one for the virtual rendering camera. All rays will intersect the corresponding geometry and be shaded individually. The resulting color is then divided to create an average color between these samples. Multiple methods of how to sample the rays within a pixel are possible, e.g. using a regular grid, a rotated grid, random selections and various other stochastic methods. In Q3RT, we

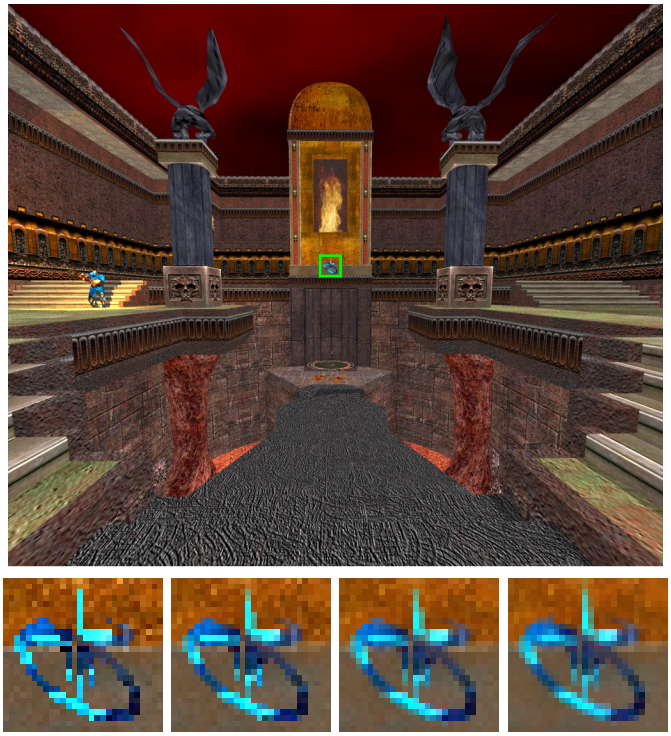

Figure 12. The top image shows the scene with 8× supersampling. The images on the bottom show a close-up of the marked green area from the top image. From left to right, these images use 1, 2, 4 and 8 samples per pixel.

used a randomized sampling pattern. The resulting renderings with different settings for 1, 2, 4 and 8 samples per pixel can be seen in Figure 12.

XII. Quake 3: Team Arena Ray Traced

Quake 3: Team Arena was an official expansion pack to the original Quake 3 games. From the graphics side, the interesting change was that much larger outdoor levels were supported in this release. After the official work on Q3RT ended with the handover of the student's bachelor's thesis, the student continued behind closed doors on the research on ray tracing on games. While a few screenshots of this continued work have been released, there was never an official mentioning on the details of this work.

The continuation did look at the next available content from id Software, which was the Team Arena pack for Quake 3. In the continued work, one of the large outdoor levels named "mpterra2" has been chosen for ray tracing. A look from high above the level with visualization for the triangle edges and other in-game views are shown in Figure 13. The size of the level spans multiple kilometers in each dimension, which was not very common for first-person shooters at this time.

Ray traced water was a new special effects added to the ray traced version of Quake 3: Team Arena. The code was derived from the glass shader. The similarities are that both need an additional reflection and an additional refraction ray. The refraction index for the water shader was changed to 1.33 instead of 1.5 for glass. Just applying the shader on a flat

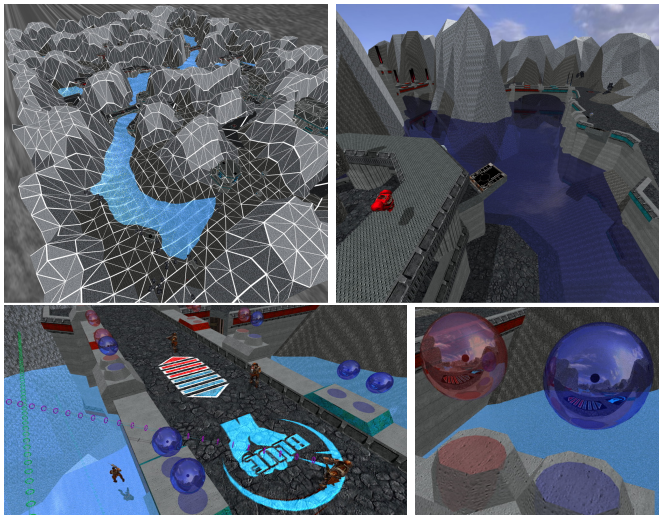

Figure 13. Team Arena level "mpterra2". Top left visualizes the triangle mesh. The other images show an initial implementation of a water shader, reflections in reflections and colored shadows from translucent objects.

Figure 14. Ray traced water shader with a normal map for a ripple effect.

surface in the world did not look very convincing. We added a normal map to the surface which simulated small ripples in the water. The normal map was animated over time, so the perception of ripples moving from wind was given. A small artistic fine tuning was done which increased the intensity of the blue color channel a bit more. An example with player models in the water is shown in Figure 14.

Having *a million reflecting spheres* in the level was the last test. The goal was to have a scenario in which ray tracing is the only feasible technology for rendering. At a certain distance to each other, about one million spheres where added into the level. As expected with ray tracing, reflections from other reflecting objects work naturally and there was no extra development required to have this working. Given the added complexity, interactivity was heavily reduced and only reasonably possible at a very low rendering resolution of 16×16 pixels. Nevertheless, we provide two offline-rendered images in higher resolution with supersampling in Figure 15.

XIII. Performance

The ray tracing performance is dependent on different factors: acceleration structure build time, the actual tracing of rays and

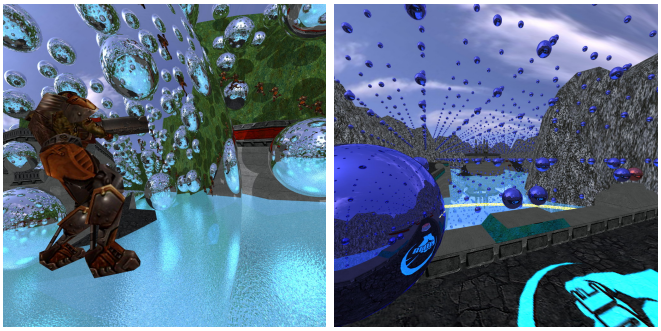

Figure 15. One million reflecting spheres in the Team Arena map

shading calculations including texturing.

The Quake 3: Ray Traced project was optimized to avoid the building of complex acceleration structures during runtime. The static geometry is only initialized once. The dynamic player models use pre-calculated acceleration structures for every animation step. The only change from frame to frame is instancing and deletion of dynamic objects like players and decals. This was implemented in OpenRT so efficiently, that no performance penalty was measurable from this.

For the tracing of rays, it was observed that, as an approximation, the number of computed rays had a linear impact on performance. However, not all rays perform equally. The primary rays shot from the camera shader showed a sublinear impact due to cache coherency. As a rule of thumb, when increasing the rendering resolution by a factor of 4, the performance impact was a reduction factor of about 3.6 to 3.8. Using supersampling and increasing the samples per pixel from 1 to 4 was also around that area with a performance reduction of about 3.4 to 3.7. The achieved frame rates on the single-core Pentium 4 system are shown in Table I.

Resolution	**spp 1**	**spp 2**	**spp 4**	**spp 8**
128×128	41.5	22.9	12.2	6.3
256×256	11.9	6.1	3.3	1.7
512×512	3.1	1.6	<1	<1

Table I
PERFORMANCE USING A DIFFERENT NUMBER OF SAMPLES PER PIXEL (SPP). VALUES IN FRAMES PER SECOND ON A SINGLE-CORE INTEL PENTIUM 4.

In the network cluster with 20 nodes, each equipped with a dual-core CPU, the frame rate was at 20 frames per second for a resolution of 512×512 pixels at $4\times$ supersampling.

As outlined in section VII, we investigated the impact of the different static, pre-calculated beziér geometry. This is again on the single-core PC. The frame rate was averaged over a walk through the level. We show the results in Table II.

As we can interpret from these measurements, there is of course an impact from increasing the overall geometric complexity. However, specifically the step going from a total of 117K to 1380K triangles had the impact of lowering the frame rate only by a relatively low 13%. The acceleration structures that are used for each tracing of a ray pay off very well: as the beziér patches are not fully visible all the time

	32Δ/p 20KΔ+ 6KΔ	128Δ/p 20KΔ+ 24KΔ	512Δ/p 20KΔ+ 97KΔ	7200Δ/p 20KΔ+ 1361KΔ
128x128	53.1	46.4	43.6	38.1

Table II
THE MAIN STATIC GEOMETRY OF THE LEVEL Q3DM7 HAS 20K TRIANGLES. DEPENDING ON THE DETAIL LEVEL OF THE PRE-CALCULATED BEZIÉR PATCHES, WE SHOW THE PERFORMANCE IN FRAMES PER SECOND. FOR EXAMPLE, HAVING 32 TRIANGLES PER PATCH, WE HAVE THE MAIN GEOMETRY OF 20K TRIANGLES ADDED WITH 6K TRIANGLES FOR PATCHES, RESULTING IN AN AVERAGE RENDERING FRAME RATE OF 53.1.

during the walk through, only the rays that cast into the highly detailed beziér patches cause a higher performance cost. The per ray geometry culling with such structures has a logarithmic performance impact depending on the number of triangles [33]. While rasterized games can use these structures as well, they are usually not down to a per-pixel culling level, but still require some larger chunks which might turn out to not be fully visible.

XIV. LEARNINGS

Now that we described all aspects of the Q3RT project, we want to discuss our learnings and impressions from it. One of the realizations during the project was that just because ray tracing is used, it does not magically improve the overall image quality by default as one might expect when previously viewing ray traced images from offline-rendered content only. The problems of aliasing still happen the same as with a rasterizer. In fact, rendering only the primary rays from the virtual ray tracing camera results in the same image as using rasterization. However, once specific effects like reflections, refractions and shadows are added, it became clear where the strengths of ray tracing are. Those effects are done with only a small amount of development effort and provide great, physically-based results. Furthermore, they are always per-pixel efficient. If only very few pixels of a reflecting object are visible, only those add an extra cost during rendering. This is in contrast with approaches like shadow mapping or reflection mapping, where the map needs to be created at a certain resolution even if only one pixel shows up in the final image using it. The combination of multiple effects, like reflections in reflections with the correct translucent colored shadowing worked flawless. No ordering of which effect to calculate first needs to be provided by the developer. It was also impressive to see the scaling of the ray traced rendering when more compute nodes were added. Ray tracing has been described before as an *embarrassingly parallel* algorithm [34]. Calculations of tracing one ray are completely independent of the other rays. Therefore, adding more computing nodes, more CPUs and other hardware units provides a great performance improvement.

The two most difficult aspects during the development of Q3RT were the low rendering resolution on a single-core PC. However, with the workarounds of quickly creating higher resolution screenshots offline and training of the human perception to this lower level of detail, this became less of an issue after a while. The other aspect that required tedious

tuning of epsilon values were the self-intersection issues as described in the chapters before.

Overall, for the student working on the Q3RT project, it was clear afterwards that ray tracing will play an important role in the future of interactive games. The development of hardware was already going to multi-core CPU architectures and highly parallel GPUs, which would help in scaling ray tracing up to real-time for consumers eventually. The open question was therefore not if it will happen, but when it would happen and be available to consumers.

XV. Fast forward to 2019 and Outlook

Today, 15 years later from our original real-time ray tracing project, there has been a tremendous amount of advancement in algorithms, software APIs and hardware for enabling real-time ray tracing. Through the years, we have seen many research projects on bringing ray tracing to modern game content: Quake 4 [35], Enemy Territory: Quake Wars [36] and the Wolfenstein [37] (2009) game were changed to use ray tracing. Bikker [38] looked into advanced effects like ambient occlusion with ray tracing and provided performance optimizations. McGuire and Mara [39] used screen-space ray tracing in 2014 as an approximation for rasterized content.

Developers targeting the professional rendering market were able to apply ray tracing into their engines by using highly optimized libraries like Intel Embree [40] or Nvidia OptiX [41].

With the release of DirectX Raytracing (DXR) in 2018, real-time ray tracing became available to the consumer and gaming world. In conjunction with that release, samples starting with ray tracing a simple triangle up to building a small game engine using DXR have been released [42]. We are starting to see many popular gaming titles using this technology in a hybrid mixture between ray tracing and rasterization. For example, *Battlefield V*, *Metro Exodus*, *Shadow of the Tomb Raider* and *Wolfenstein: Youngblood* all have some features that can utilize ray tracing in real-time. A next step to expect would be the development of games using additional global illumination features, similar to the *Quake 2 Vulkan Path Tracing* [43] project. Further along, we might be heading towards fully ray traced games with full real-time global illumination. However, there is still a need for more advances in algorithms and hardware before this can be achieved.

Recently, we have seen how machine learning being applied to interactive ray-tracing can denoise an image [44], [45] at similar quality to a highly sampled ray traced image with far fewer ray samples per pixel. Today's graphics architectures have compute cores for general purpose, rasterization, ray tracing, and AI. A combination using those can provide additional benefits for rendering like a rasterized image with ray traced special effects, where the lighting gets denoised through AI and the final image gets upsampled with AI to a higher display resolution. For future work, we could see more areas where AI can impact the rendering pipeline to help accelerate ray tracing like guiding importance sampling, applying super-resolution across frames, and enhancing acceleration structures.

XVI. Conclusion

We have shown how in the year 2004 we created a full ray traced game in the research project *Quake 3: Ray Traced*. We described the first footsteps of applying ray tracing to gaming. We demonstrated how we handled multiple light types, shadows, dynamics, and special effects. A look at aliasing and ray length-based mipmapping with supersampling was provided. We shared our learnings from the Q3RT project. We gave an overview to what happened after it until today with an outlook on future rendering architectures. Although we were limited by processing power and, in turn, screen resolution and features at the time, this provided validation that as algorithms and hardware advanced that real-time ray tracing was possible. Today, it is even included in several AAA game titles.

XVII. Acknowledgment

Many thanks to Jörg Schmittler for supervising the *Quake 3: Ray Traced* project together with Prof. Marc Stamminger and Prof. Philipp Slusallek. Additional thanks to Tim Dahmen and the whole OpenRT team for providing the ray tracing library.

References

[1] C. Wyman and A. Marrs, "Introduction to DirectX Raytracing", in *Ray Tracing Gems*, Springer, 2019, pp. 21–47.

[2] G. R. Hofmann, "Who invented ray tracing?", *The Visual Computer*, vol. 6, no. 3, pp. 120–124, 1990.

[3] A. Appel, "Some techniques for shading machine renderings of solids", in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, ACM, 1968, pp. 37–45.

[4] T. Whitted, "An improved illumination model for shaded display", in *ACM SIGGRAPH Computer Graphics*, ACM, vol. 13, 1979, p. 14.

[5] R. L. Cook, T. Porter, and L. Carpenter, "Distributed ray tracing", in *ACM SIGGRAPH computer graphics*, ACM, vol. 18, 1984, pp. 137–145.

[6] A. S. Glassner, "Spacetime ray tracing for animation", *IEEE Computer Graphics and Applications*, vol. 8, no. 2, pp. 60–70, 1988.

[7] M. J. Muuss and M. Lorenzo, "High-resolution interactive multispectral missile sensor simulation for ATR and DIS", in *Proceedings of BRL-CAD Symposium'95*, vol. 2, 1995.

[8] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P.-P. Sloan, "Interactive ray tracing for isosurface rendering", in *Proceedings Visualization'98 (Cat. No. 98CB36276)*, IEEE, 1998, pp. 233–238.

[9] I. Wald, P. Slusallek, C. Benthin, and M. Wagner, "Interactive rendering with coherent ray tracing", in *Computer graphics forum*, Wiley Online Library, vol. 20, 2001, pp. 153–165.

[10] I. Wald and C. Benthin, "OpenRT-A flexible and scalable rendering engine for interactive 3D graphics", 2002.

[11] A. Dietrich, I. Wald, C. Benthin, and P. Slusallek, "The OpenRT application programming interface–towards a common API for interactive ray tracing", in *Proceedings of the 2003 OpenSG Symposium*, Citeseer, 2003, pp. 23–31.

[12] J. Schmittler, D. Pohl, T. Dahmen, C. Vogelgesang, and P. Slusallek, "Realtime ray tracing for current and future games", in *ACM SIGGRAPH 2005 Courses*, 2005, p. 23.

[13] D. Pohl, "Applying Ray Tracing to the Quake 3 Computer Game", *University of Erlangen*, 2004.

[14] M. Woo, J. Neider, T. Davis, and D. Shreiner, *OpenGL programming guide: the official guide to learning OpenGL, version 1.2*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[15] H. Fuchs, Z. M. Kedem, and B. F. Naylor, "On visible surface generation by a priori tree structures", in *ACM Siggraph Computer Graphics*, vol. 14, 1980, pp. 124–133.

[16] D. P. Luebke and C. Georges, "Portals and mirrors: Simple, fast evaluation of potentially visible sets.", *SI3D*, vol. 95, p. 105, 1995.

[17] C. Benthin, I. Wald, and P. Slusallek, "Interactive ray tracing of free-form surfaces", in *Proceedings of the 3rd international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*, ACM, 2004, pp. 99–106.

[18] M. Abrash, "Quake's lighting model: Surface caching", *Graphic Programming Black Book*, 2000.

[19] J. Carmack, *John Carmack on shadow volumes*, http : / / fabiensanglard . net / doom3 _ documentation / CarmackOnShadowVolumes.txt, 2000.

[20] H. Murakami, *1Q84*. Random House, 2012.

[21] C. Everitt, A. Rege, and C. Cebenoyan, "Hardware shadow mapping", *White paper, nVIDIA*, vol. 2, 2001.

[22] R. Fernando, S. Fernandez, K. Bala, and D. P. Greenberg, "Adaptive shadow maps", in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ACM, 2001, pp. 387–390.

[23] M. Stamminger and G. Drettakis, "Perspective shadow maps", in *ACM transactions on graphics (TOG)*, vol. 21, 2002, pp. 557–562.

[24] C. Wächter and N. Binder, "A fast and robust method for avoiding self-intersection", in *Ray Tracing Gems*, Springer, 2019, pp. 77–85.

[25] P. Jaquays and B. Hook, "Quake 3: Arena shader manual, revision 10", in *Game Developer's Conference Hardcore Technical Seminar Notes*, 1999.

[26] J. Günther, H. Friedrich, H.-P. Seidel, and P. Slusallek, "Interactive ray tracing of skinned animations", *The Visual Computer*, vol. 22, no. 9-11, pp. 785–792, 2006.

[27] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast BVH construction on GPUs", in *Computer Graphics Forum*, Wiley Online Library, vol. 28, 2009, pp. 375–384.

[28] D. Kopta, T. Ize, J. Spjut, E. Brunvand, A. Davis, and A. Kensler, "Fast, effective BVH updates for animated scenes", in *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM, 2012, pp. 197–204.

[29] J. Van Waveren, "The Quake III Arena Bot", *University of Technology Delft*, 2001.

[30] P. S. Heckbert *et al.*, "Texture mapping polygons in perspective", Citeseer, Tech. Rep., 1983.

[31] H. Igehy, "Tracing ray differentials", in *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., 1999, pp. 179–186.

[32] F. C. Crow, "A comparison of antialiasing techniques", *IEEE Computer Graphics and Applications*, no. 1, pp. 40–48, 1981.

[33] J. Schmittler, S. Woop, D. Wagner, W. J. Paul, and P. Slusallek, "Realtime ray tracing of dynamic scenes on an FPGA chip", in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2004, pp. 95–106.

[34] B. Freisleben, D. Hartmann, and T. Kielmann, "Parallel raytracing: A case study on partitioning and scheduling on workstation clusters", in *Proceedings of the thirtieth hawaii international conference on system sciences*, IEEE, vol. 1, 1997, pp. 596–605.

[35] *Quake 4: Ray Traced*, http://www.q4rt.de.

[36] *Quake Wars: Ray Traced*, http://www.qwrt.de.

[37] *Wolfenstein: Ray Traced*, http://www.wolfrt.de.

[38] J. Bikker, "Real-time ray tracing through the eyes of a game developer", in *2007 IEEE Symposium on Interactive Ray Tracing*, IEEE, 2007, pp. 1–10.

[39] M. McGuire and M. Mara, "Efficient GPU screen-space ray tracing", *Journal of Computer Graphics Techniques (JCGT)*, vol. 3, no. 4, pp. 73–85, 2014.

[40] *Intel embree*, http://www.embree.org.

[41] *Nvidia OptiX*, http://developer.nvidia.com/optix.

[42] *DirectX Raytracing samples*, http://github.com/microsoft/DirectX - Graphics - Samples / tree / master / Samples / Desktop/D3D12Raytracing.

[43] *Quake 2 Vulkan Path Tracer*, http://brechpunkt.de/q2vkpt.

[44] C. R. A. Chaitanya, A. S. Kaplanyan, C. Schied, M. Salvi, A. Lefohn, D. Nowrouzezahrai, and T. Aila, "Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder", *ACM Transactions on Graphics (TOG)*, vol. 36, no. 4, p. 98, 2017.

[45] *Intel Open Image Denoise*, http://openimagedenoise.github.io.

APPENDIX

As an appendix, we provide more images that were created during the development. Not always everything worked on the first try or was fully implemented when taking these images. Furthermore, we show unreleased images from other levels of Quake 3, Quake 2 and the classic Wolfenstein game that were used for experimentation with ray tracing during the project.

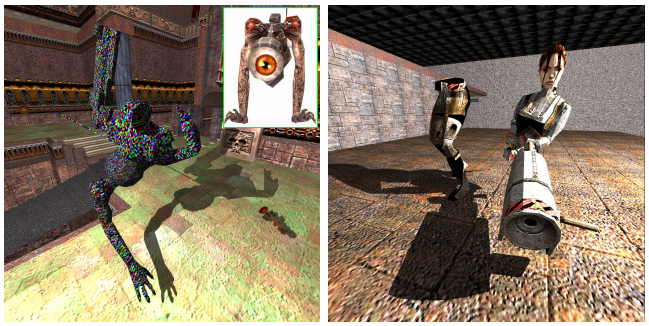

Figure 16. Left: invalid texture access for the Quake 3 model "Orbb". The texture visualizes random parts of the memory. Right: too large offset between the lower and upper body model.

Figure 17. Left: something went wrong during the mesh setup of the ammunition box. Right: during rendering the glass of the health sphere, rays got stuck inside due to self-intersection problems.

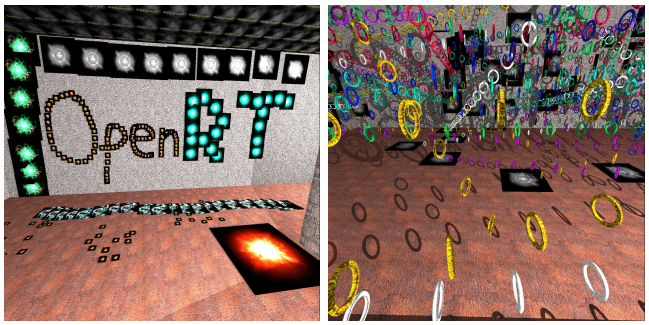

Figure 18. Left: shooting decals on the wall that last forever. Right: shooting rings of the virtual rail gun weapon without deletion.

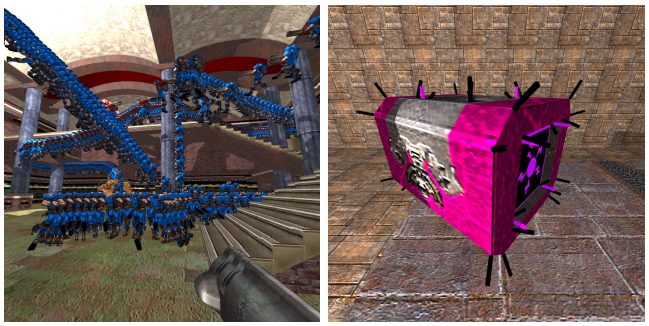

Figure 19. Left: moving character animations if they are not manually deleted. Right: visualizing surface normals through added geometry on models.

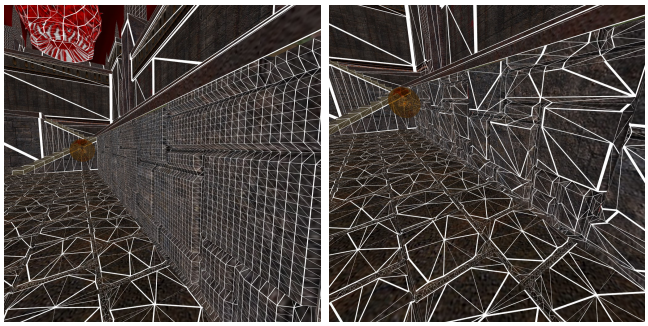

Figure 20. Both images show experiments replacing the original flat geometry with highly detailed meshes. The performance difference was only small.

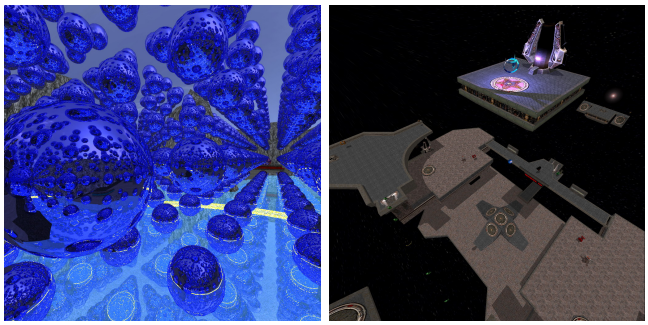

Figure 21. Left: the white pixels show areas where not enough ray tracing recursion depth was set for multiple bounced reflections. Right: the Quake 3 map q3dm17, ray traced.

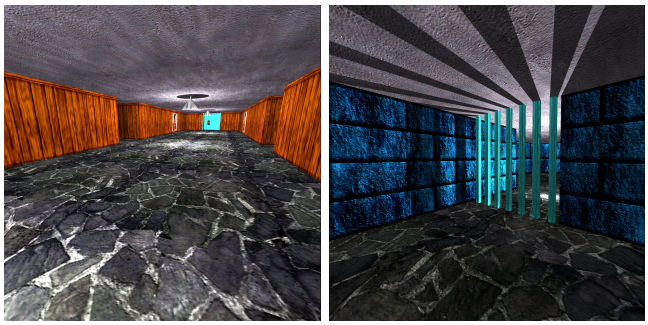

Figure 22. Both images show renderings of an enhanced level of Wolfenstein 3D (1992) with ray tracing.

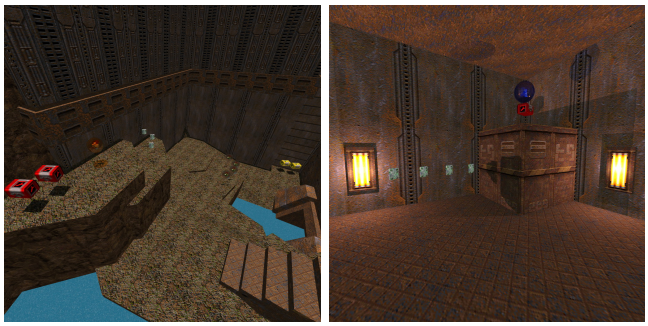

Figure 23. Both images show a modified version of the Quake 2 level q2dm1 using ray tracing.