Ray Tracing and Gaming - One Year Later

Author: Daniel Pohl
Date: Jan 17, 2008

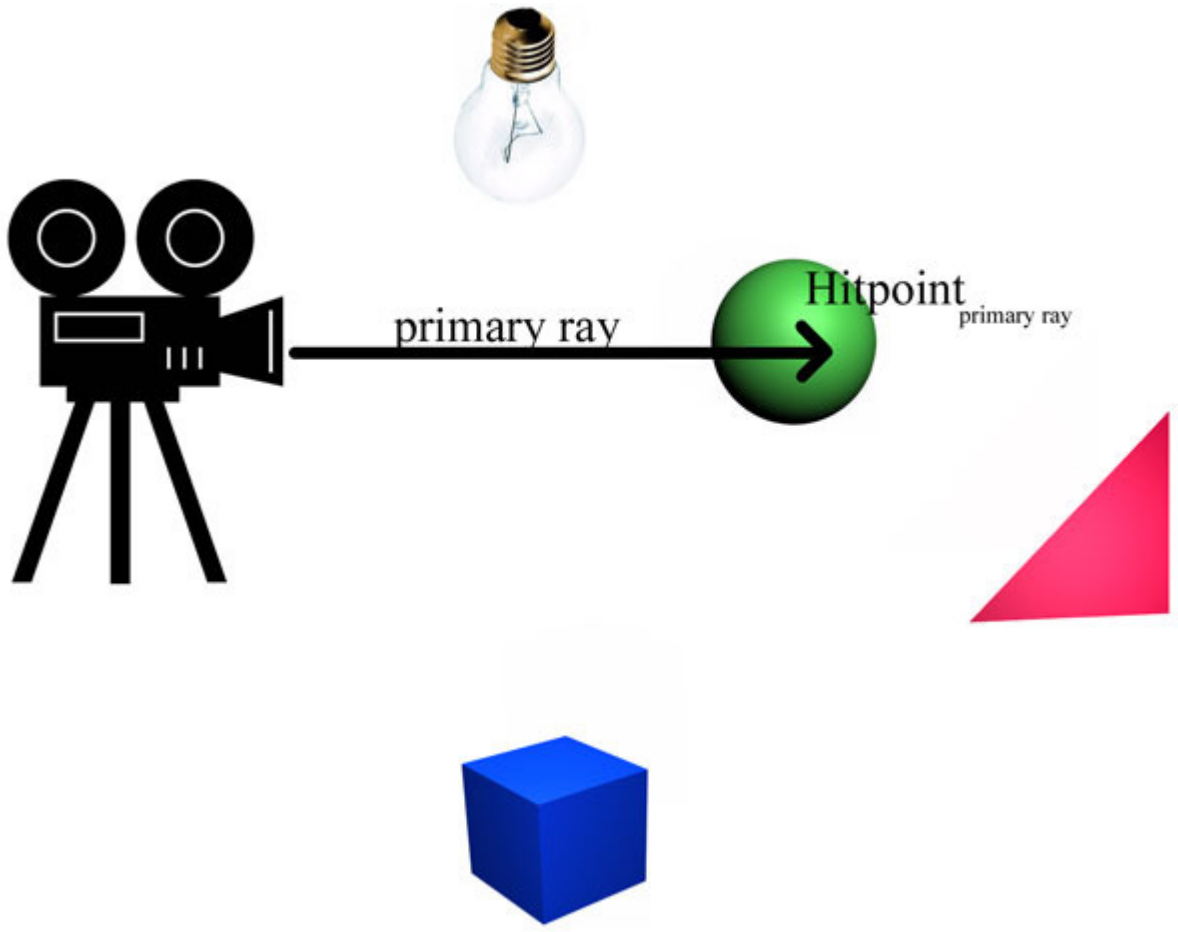
One Year Later

This article is intended to be a follow up article to one released about a year ago through PC Perspective. A lot of things have changed since then. Real-time ray tracing for desktop machines is just around the corner.

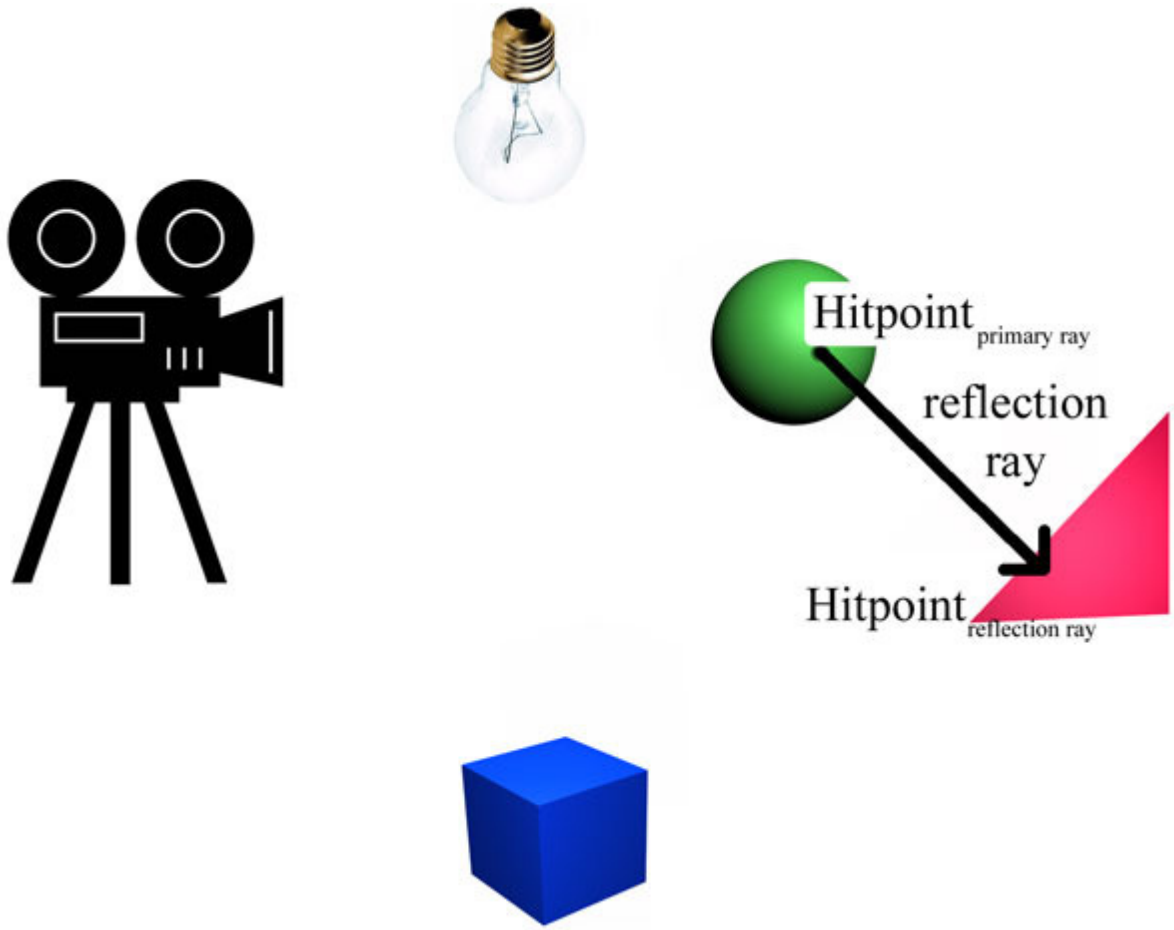
Introduction

If you are new to the topic of ray tracing you might want to read through this section.

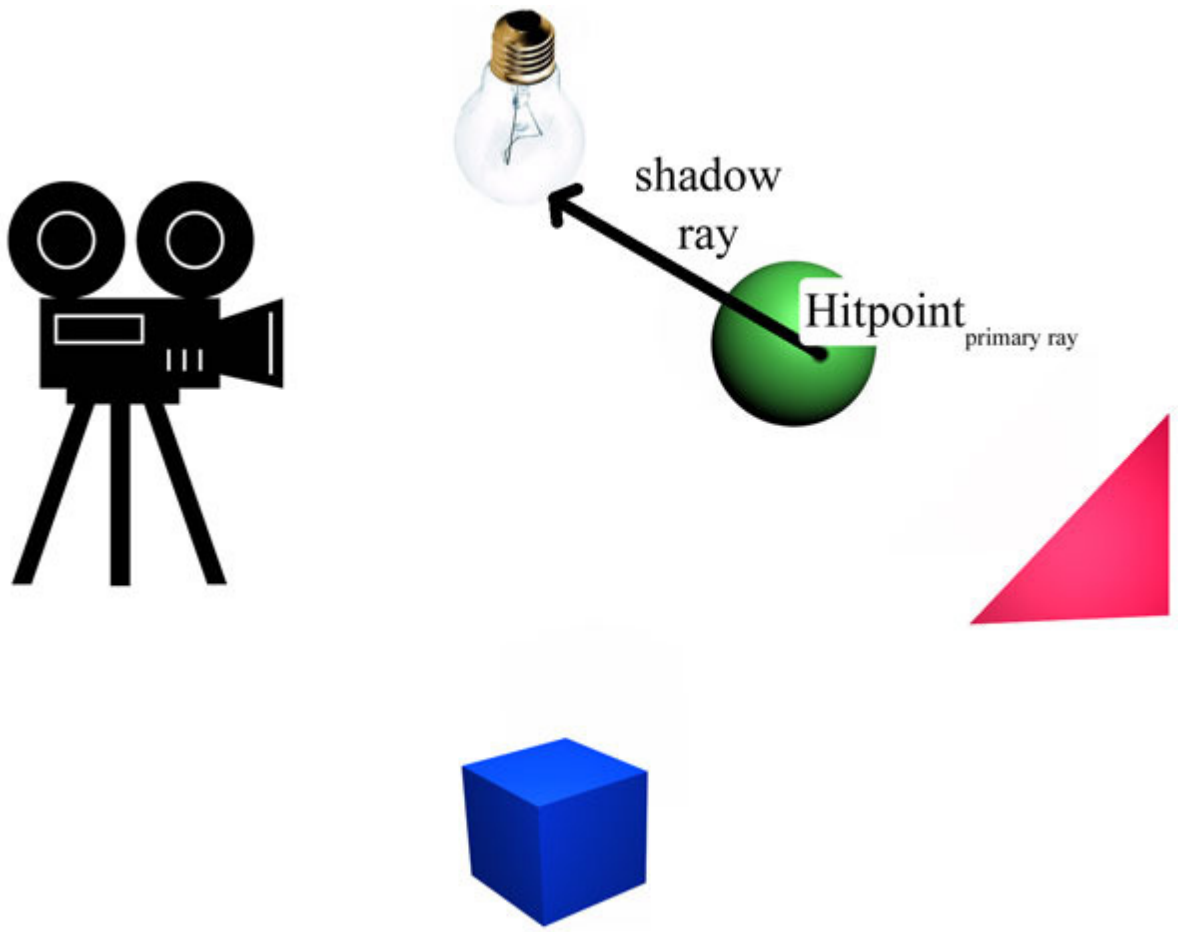
Ray tracing is a rendering technique that generates a 2D image out of a given 3D scene. This is done by simulating the physics of light propagation using rays. The algorithm shoots, for every pixel on the screen, a so-called "primary ray" from the perspective of the eye of the viewer. The ray tracing algorithm then determines which object is hit first on the path of the ray.



At that hit-point a shader program is invoked which could, for example, cast another ray to simulate (say) a reflection at a mirror.



Through so-called "shadow rays" it can be easily determined if a given pixel is lit or in shadow. If a ray from the point in question can be shot to the light source without being blocked, then it can be concluded that light reaches that point. In the other case – when it is blocked – then we can conclude that the point is in shadow.

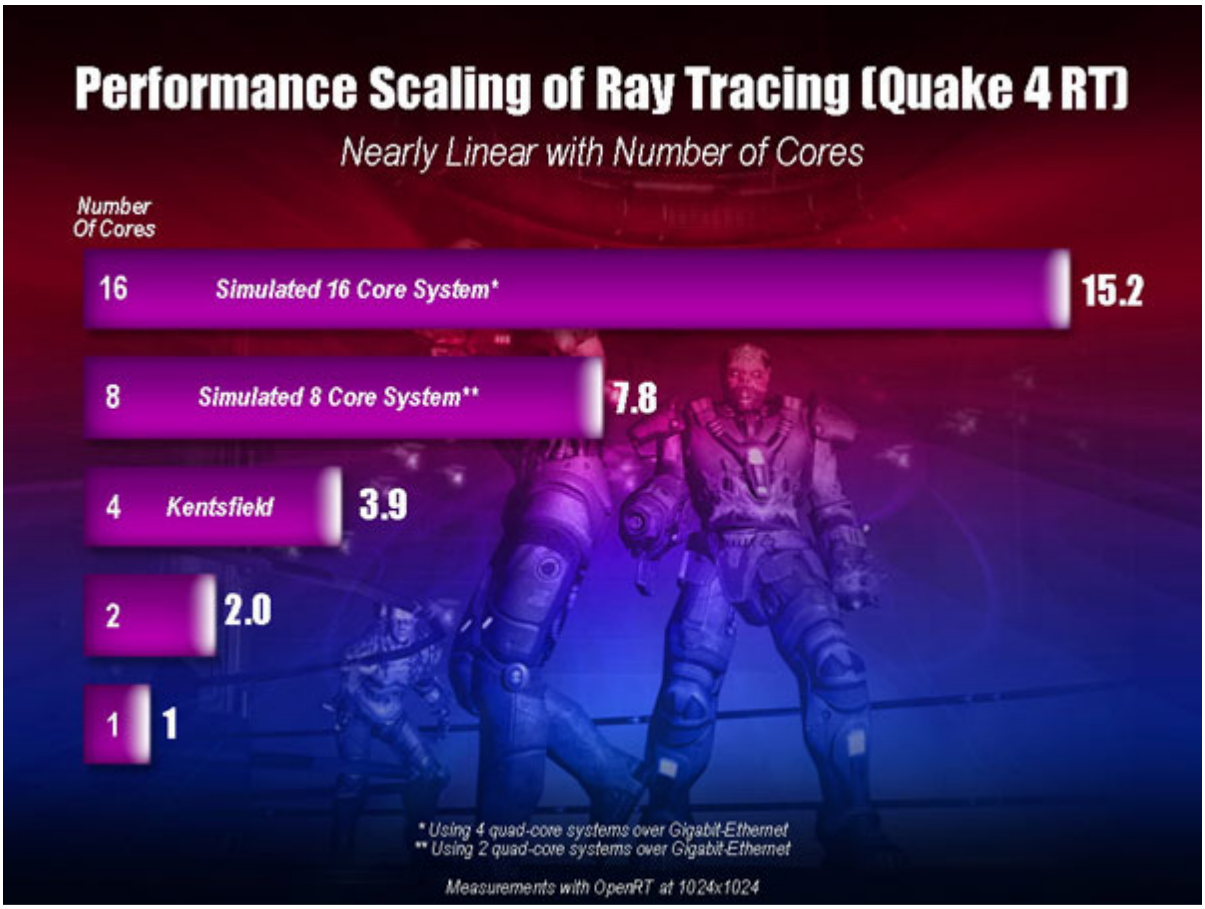


The ray tracing approach I am reporting about in this article is calculated completely on the CPU. No graphics card is needed to create the image. (Once created, we merely transport the pixels to the graphics card to have it paint the image onto the monitor).

Another approach to generate a 2D image out of a 3D scene is called "rasterization" and is currently performed by special-purpose hardware graphics cards. Currently, this the standard way that games "render" the images you see using standard libraries like DirectX or OpenGL.

Progress

After my last article about ray tracing and gaming hit the web I did some more research on scaling of ray tracing with the number of CPU cores. In order to simulate a 16-core machine I took four quad-core PCs and connected them over a Gigabit-Ethernet to combine their power. Because my project used the ray tracing library OpenRT from Saarland University which supports distributed rendering, this was quite easy to achieve.



Performance scaling of ray tracing with different number of cores

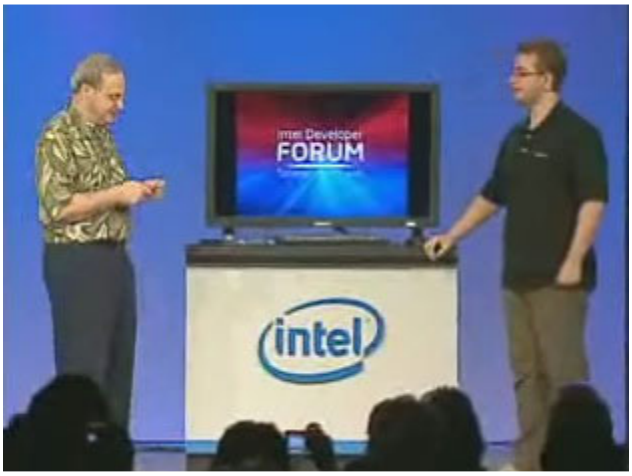
The results were amazing: If you use a 16-core machine instead of a single-core machine then the frame rate increases by a factor of 15.2!

After all the attention the previous ray tracing article got around the world I was contacted by several companies interested in this technology. One of them was Intel. They told me they would have a real-time ray tracer that would be around 10× faster than everything else that has been published so far. These performance numbers were already written down in some research papers, but I did not trust them without seeing it myself. So I went over to Santa Clara to get a live demonstration of it.

Then I saw it trace,
now I'm a believer!

So now I am a full-time research scientist working for Intel on ray tracing. As it turns out, Intel's labs are very interested ray tracing because it is so well suited to general purpose CPUs. Our ray tracing research is just a part of an overall program here called "tera-scale computing" aimed at scaling CPUs from a few cores to many (meaning tens or hundreds).

Joining Intel also gave me the chance to demonstrate Quake 4: Ray-Traced with the new Intel ray tracer on several occasions such as the Games Convention 2007 in Leipzig and Intel Developer Forum Fall 2007 in San Francisco. At the latter event it was even featured in the keynote from Justin Rattner (Intel CTO) about virtual worlds.



At HD resolution we were able to achieve a frame rate of about 90 frames per second on a Dual-X5365 machine, utilizing all 8 cores of that system for rendering.

More details about the presentation at Fall IDF 2007 can be found in Ryan Shrout's article at PC Perspective.

Ray tracing faster than rasterization: Example 1

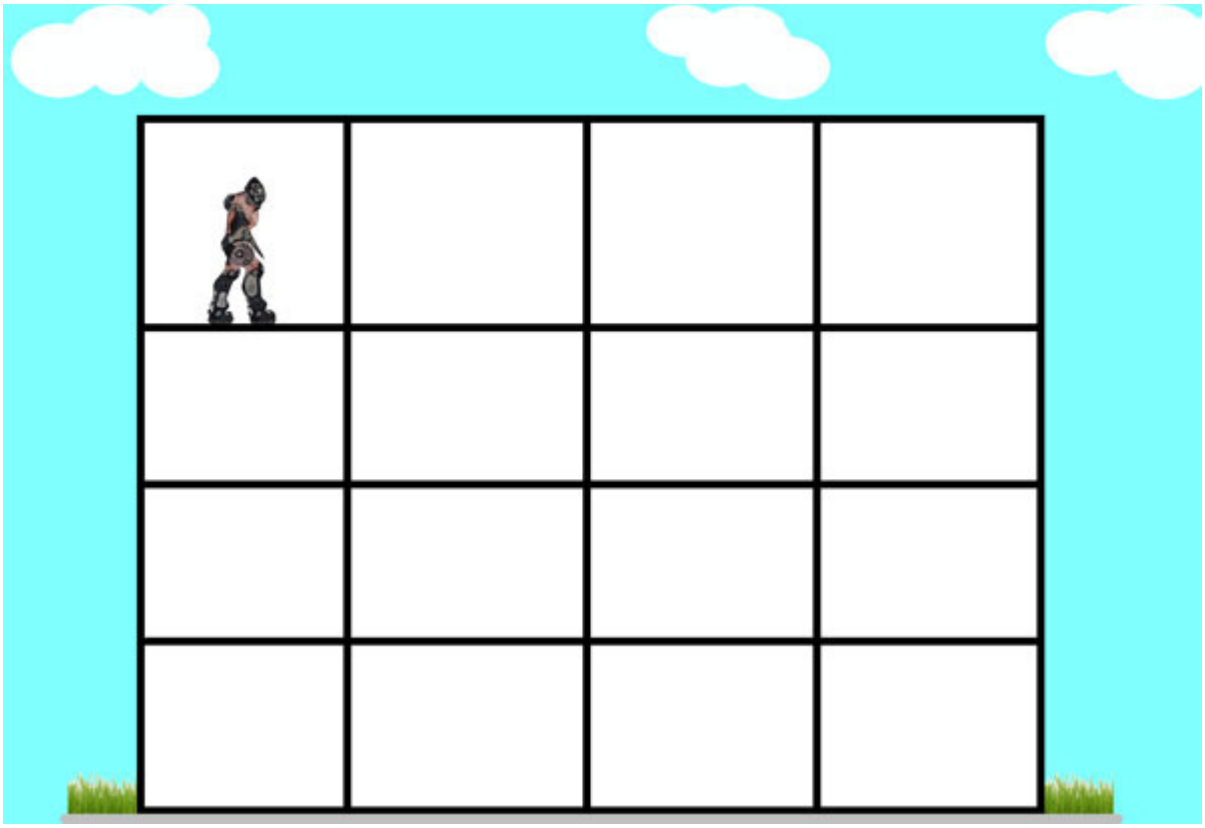
I have been asked several times when ray tracing will become faster than rasterization (the current GPU approach for rendering). There are cases in which it is faster on a single desktop machine TODAY!

Example 1: High number of triangles

Ray tracing uses "acceleration structures" that sort all the geometry of a virtual world according to their position in space. There are several different acceleration structures, but the most common ones used in ray tracing are Uniform Grids, BSP-trees, kd-trees and Bounding Volume Hierarchies (BVHs).

As described in the introduction, when a ray is shot, we must find the first object that is hit by that ray. The general idea of spatial partitioning / acceleration structures is best described through a small example:

Imagine you have a building with 4 levels and on each of those there are 4 rooms, if a player is located in the top-most, left-most room as shown below:

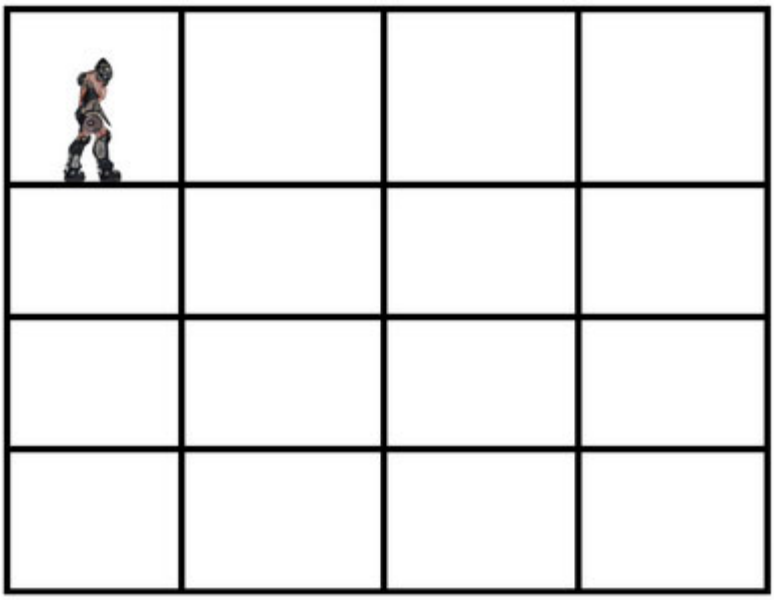


Artistic representation of a building with 4 levels with each 4 rooms

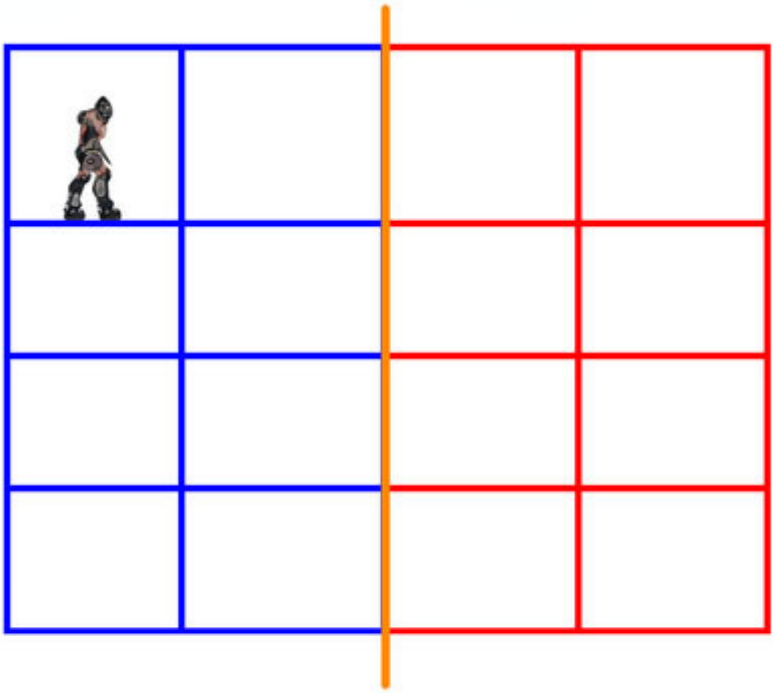
When ray tracing, we want to figure out whether a piece of geometry is hit or not. In ray tracing speak we talk about a "camera" and when we shoot rays through it to determine what is visible we refer to these as "eye rays" or "primary rays". Naively, we could just shoot eye rays everywhere and see what gets hit and what does not. Clearly, in the above example 15/16 of these rays would be wasted. Thinking about this situation a little, it seems obvious that it is very unlikely that the camera will "see" anything in the right-most room on the lowest-level, so why even bother checking if a ray hits any of the geometry from that area? How can we avoid such redundant work?

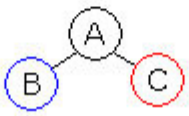
With hierarchical spatial acceleration structures we can just look at a convenient representation of an area and determine if there is any interesting detail there worth investigating further. If not for a given area or volume, we can save a lot of computational effort. This way we can focus computational resources on the areas where the geometrical details are located.

One advantage of such hierarchical data structures is that they have the effect of changing a "linear search" – check everything (every time) to see if there is a match of interest - to a "logarithmic" search – check the highest level which represents the bounds of a large area, if and only if there is detail of interest in this area, then proceed to the next level. In the above example, the "top" level might represent the 4×4 grid...

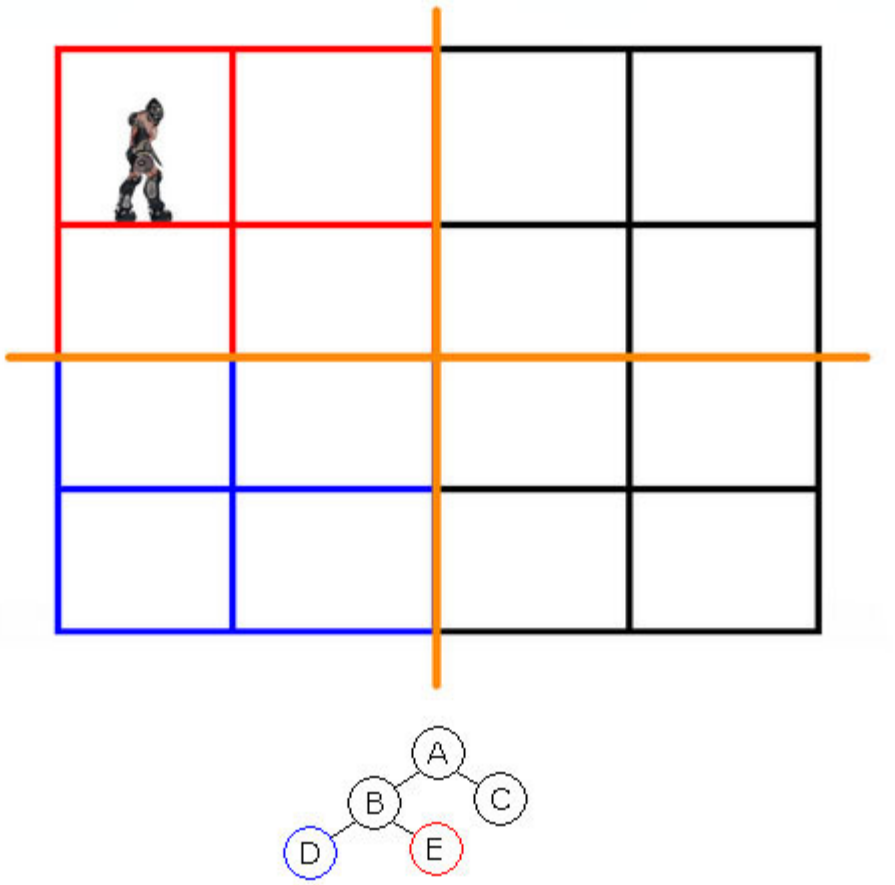


...the next level might split the grid right down the middle top to bottom...



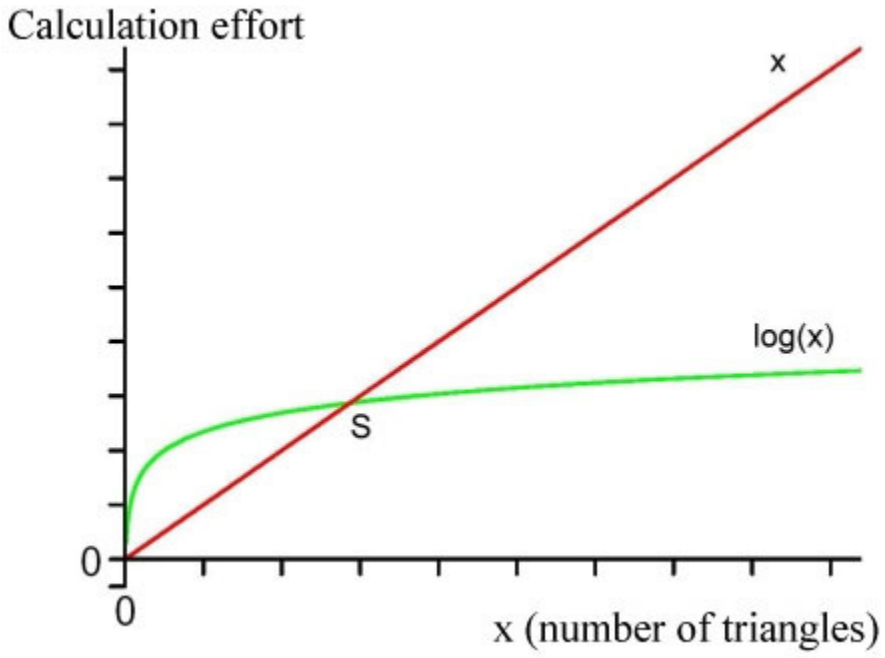


...on the left side that half might be split in the middle from side to side etc.



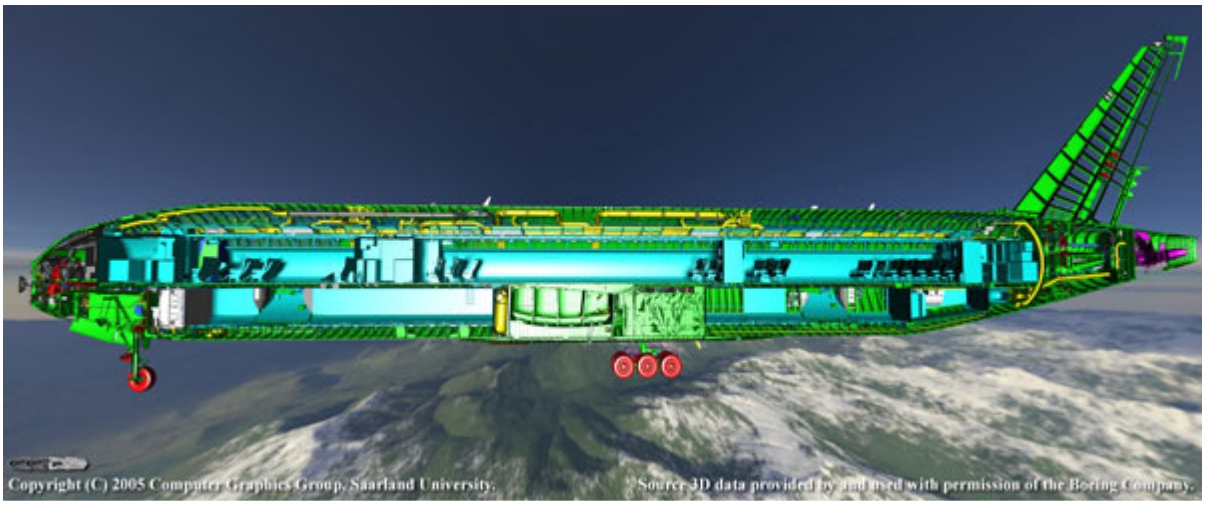
In the linear case we check every square in the grid – 16 units of work. In the logarithmic case, in step one, we eliminate the right half; In step two we eliminate the left, bottom half; In step three, the right half of the remaining quartet, and; In step 4, we determine which remaining half the interesting detail is in – so 4 steps instead of 16. If we increased the number of grid cells to 32, the result would be 32 checks in the linear case to 5 checks in the logarithmic case and so on. One way to think of this is that if we increase the complexity of a scene 10 times, using a hierarchical acceleration structure increases the cost of finding something by only 2×. Contrast that with the traditional rasterization approach, it is condemned to use a linear approach, so if we increase complexity by 10× the cost goes up by 10×.

This behavior is represented in this diagram:



The green curve represents the logarithmic behavior of ray tracing when the number of triangles are increased, the red line represents the linear behavior of rasterization. As you can see, initially for ray tracing (when the polygon count is low) ray tracing performance is at a disadvantage compared to rasterization, but quickly the two curves meet, and from that point on, as complexity increases ray tracing is ALWAYS faster than rasterization. This cross-over point depends on many factors: the performance of the CPU, the performance of the GPU etc, but this trend is a mathematical certainty, a logarithmic curve will always intersect a linear curve and the logarithmic curve will always win! Due to the linear scaling of ray tracing performance, doubling the number of CPUs would shrink the height of the green curve by half, moving the intersection point (S) closer and closer to 0, ie throw enough CPU cores at the problem and Ray Tracing would always be faster than Rasterization using a GPU.

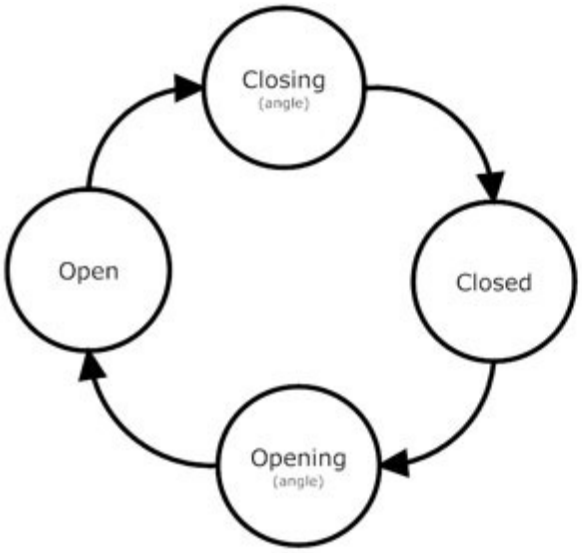
One example that clearly is above that point is the Boeing 777 model with 350 million triangles. This extreme highly detailed model that includes every screw of the plane has been provided by Boeing to the research community in order to experiment with methods on how to render it.



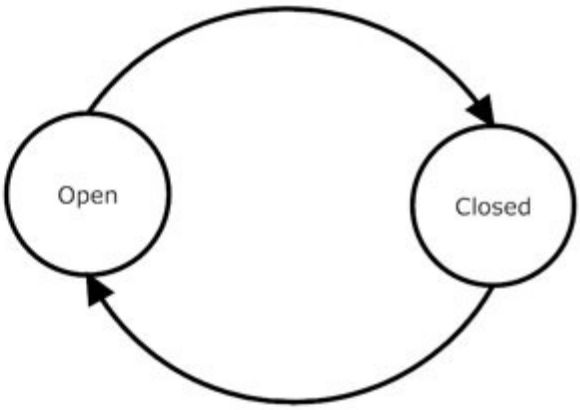
Ray tracing has been proven to be the right solution for that (http://graphics.cs.uni-sb.de/Publications/). Even back in 2004 the research group from Saarland University was able to render the model with 3 to 7 frames per Second at 640x480 pixel resolution with a dual-core CPU from that year.

The question may arise, why can't these acceleration structures be used in rasterization? Well, they are, to a certain extent, but at a lower precision. Some games have one such coarse-grained structure for rendering graphics, a different one (at a different resolution) for collision-detection and yet another one for AI (sometimes created by different third-party infrastructure suppliers). Besides taking up more memory than necessary, one problem in using these three separately computed data structures is the effort to keep them consistent. See below for an example of what happens when the information from the collision detection structure differs from the rendering structure.

In the game "Oblivion" two different structures are used for graphical rendering and for collision detection. The process of a slowly closing door, changing its angle from frame to frame is clearly visible to the player. Therefore four states can be speculated: Open, closing, closed and opening. But the structure for collision detection engine does not update the dynamic movements with fine details such as the angle that the door is currently at. Therefore other NPCs like "Velwyn" can detect only two states of the door: Open; or Closed.



Graphical Rendering State Diagram



Collision Detection State Diagram

What happens in the game as a result, is that Velwyn approaches the player while is obviously closing, but Velwyn only gets information that the door is either open or closed. So we can end up in the situation depicted below where the character ends up mixed up with the door.

Velwyn stuck in the door in Oblivion (2006)

(Of course ray tracing can also be used for collision detection to avoid those problems but that is another story.)

But even if the acceleration structure for graphical rendering is consistent with the rest then there is another problem: In ray tracing we are testing a ray in a per-pixel exact method against the triangles. In rasterization there are no rays therefore the relevant area / volume of the structure can only be approximated. This can be done by different methods. Let' have a short look at two of them:

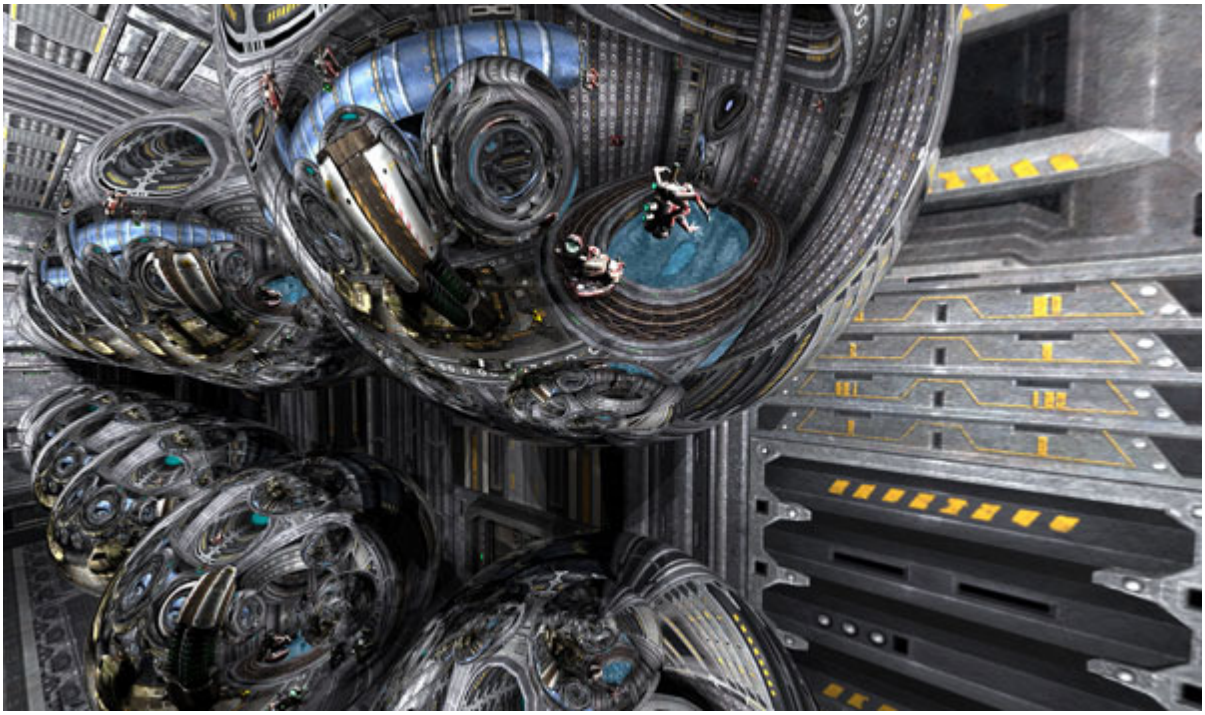
- Time consuming pre-calculations resulting in statements like "When I am in this room 1, then I could potentially see into room 2 and 3, but not room 4." For more information have a look at the Wikipedia entry for "Potentially Visible Set"
- Manually placed hints from a level designer for the engine known as visibility portals. Those consume a lot of time for the artists, e.g. Quake 4 has 3,200 of them. So far automatic algorithms for this have not made it into the practical world of game developers. The evaluation of the portals during the rendering process leads to complicated multi-pass techniques: First the scene is rendered with placeholders for these portals. Once it is detected that one of the placeholders is visible then this part of the scene is rendered again in full detail. More information can be found here.

Ray tracing faster than rasterization: Example 2

The second example is a case study on multiple reflections in reflections on spheres and tori.

A Torus is the mathematical term for a geometric object that has the shape of a donut. The plural of a Torus is some Tori.

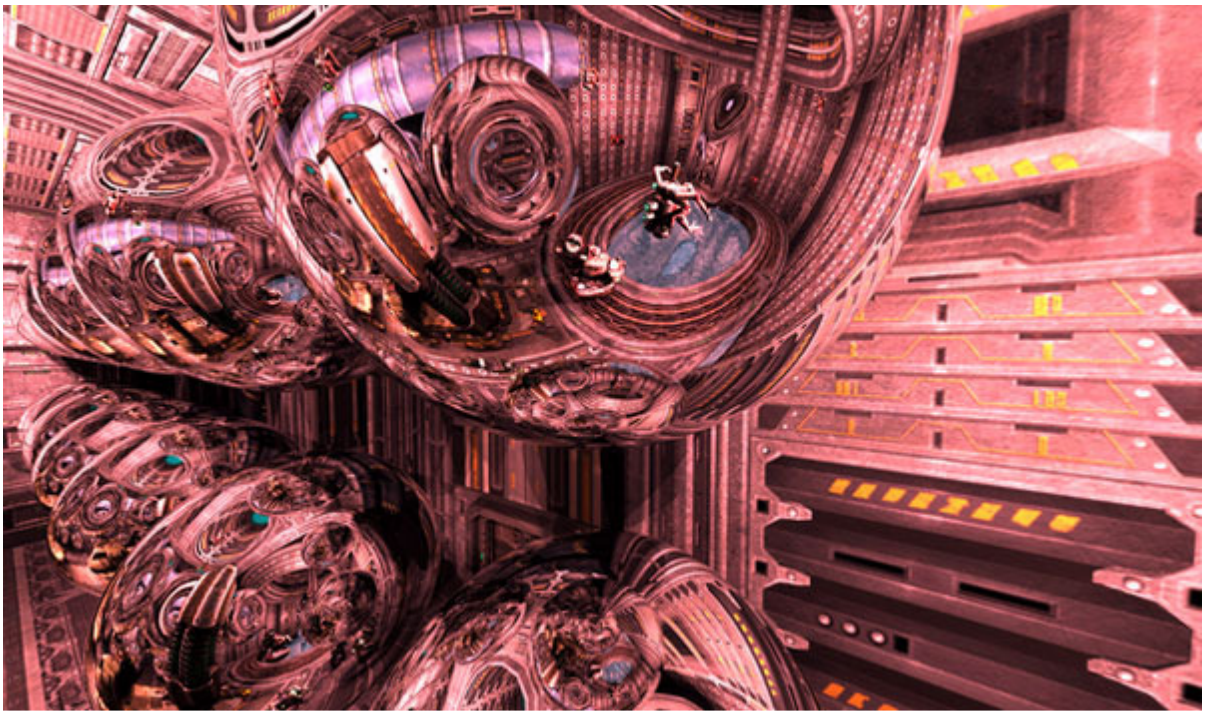
Mmm… Donuts!



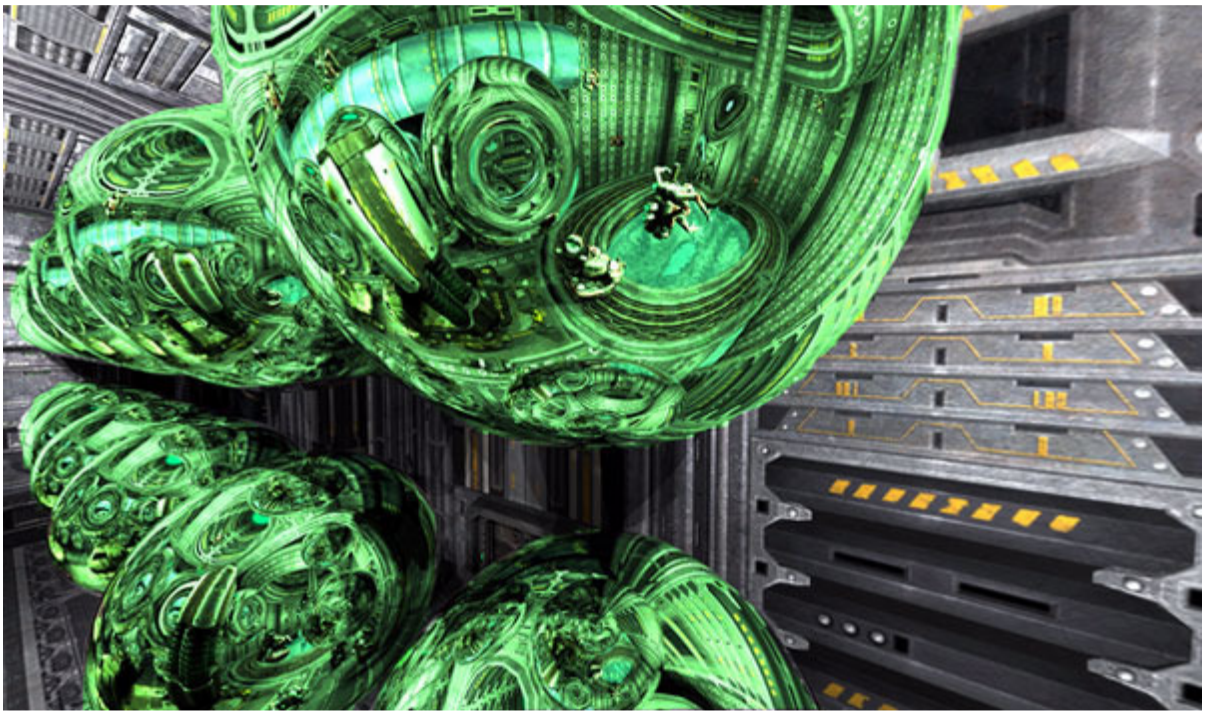
Multiple reflections in reflections - Click here for 80MB PNG file (10200x6000)

If we could just pretend for a moment, and ignore the fact that is scene would be impossible to render this scene using rasterization, not only is it possible to do this correctly with ray tracing, it's actually much cheaper than you might think! Ray tracing efficiently and accurately calculates all reflections on an as-needed basis. That means that only what is actually visible will be actually calculated. Consider the following sequence of images to see how the rays are actually used:

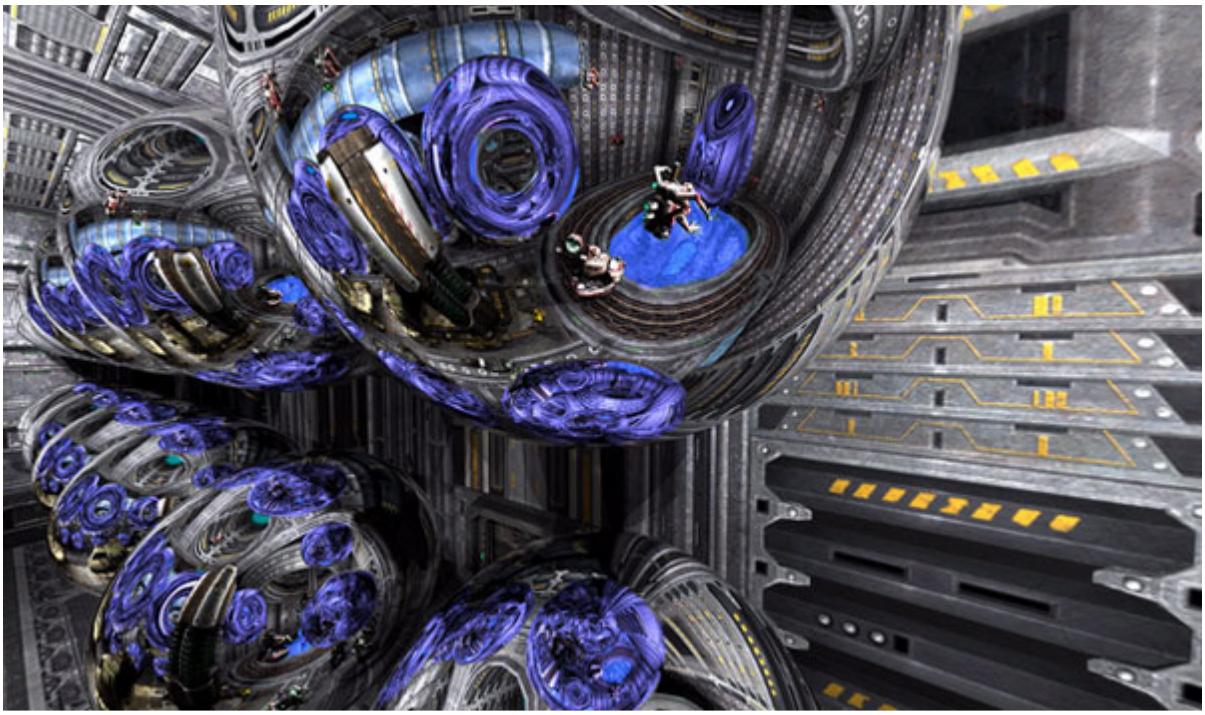
Step 1: "Eye" rays – at least one per pixel - (Tinted Red)



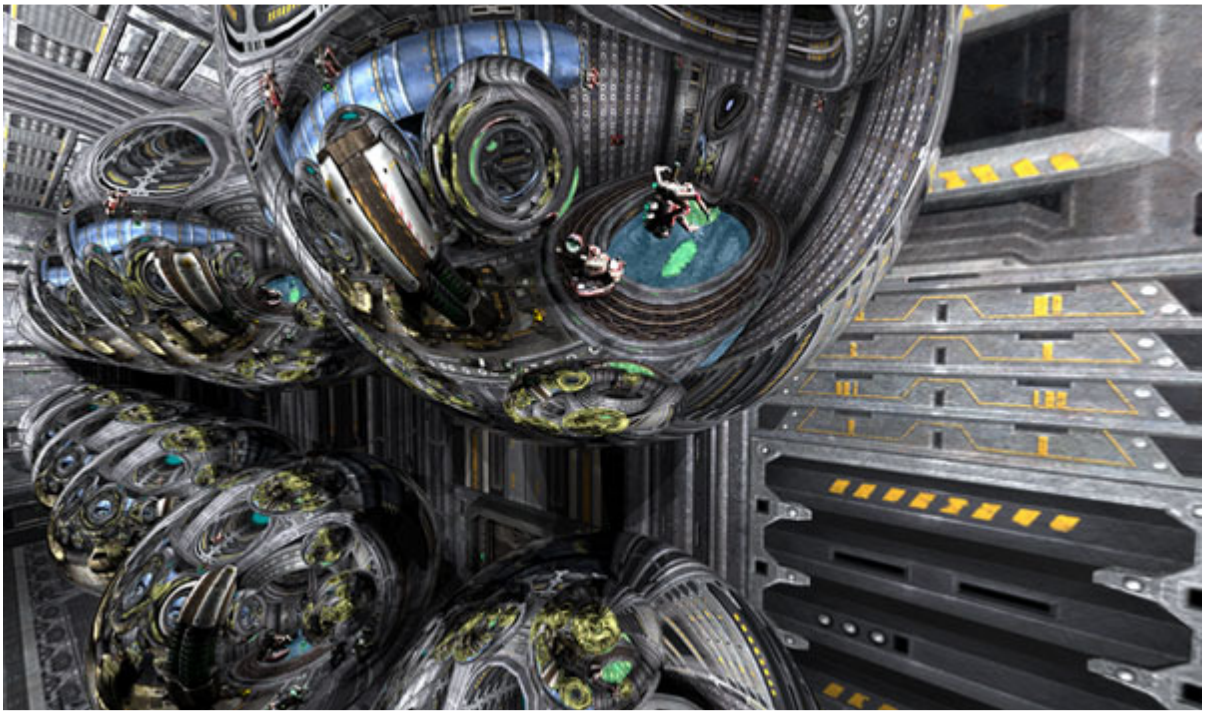
Step 2: Primary reflections (Tinted Green)



Step 3: Secondary reflections (Tinted Blue)



Step 4: 3rd order reflections (Tinted Yellow)



Step 5: ...

So clearly, with each iteration, the number of subsequent reflection rays decreases

dramatically, therefore even a complex scene with many reflections represents only a small incremental cost with ray tracing.

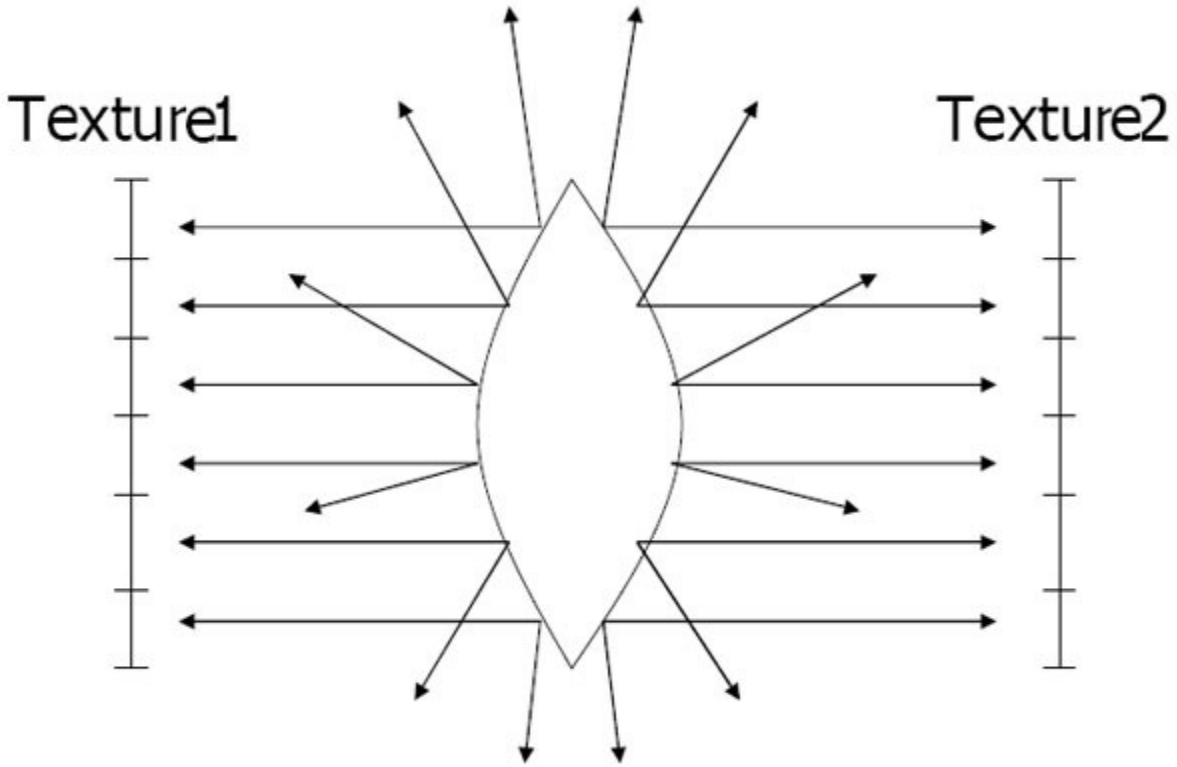
How are reflections handled in rasterization?

In order to simulate reflections, the scene is pre-rendered from at least one camera position and saved to a texture map (called a reflection map, sometimes a cube map) which is then applied as a layer of texturing in a subsequent rendering of the scene, a similar method is used to create shadow maps, each of these passes requires rendering the scene before you really render the scene!

There are different cases and methods:

- Below is an example of a one-pass reflection map texture notice the fish-eye lens distortion effect.

- Here is a 2-pass reflection mapping which results in parabolic distortion.



- Here we have 6 passes to create a cube-map.

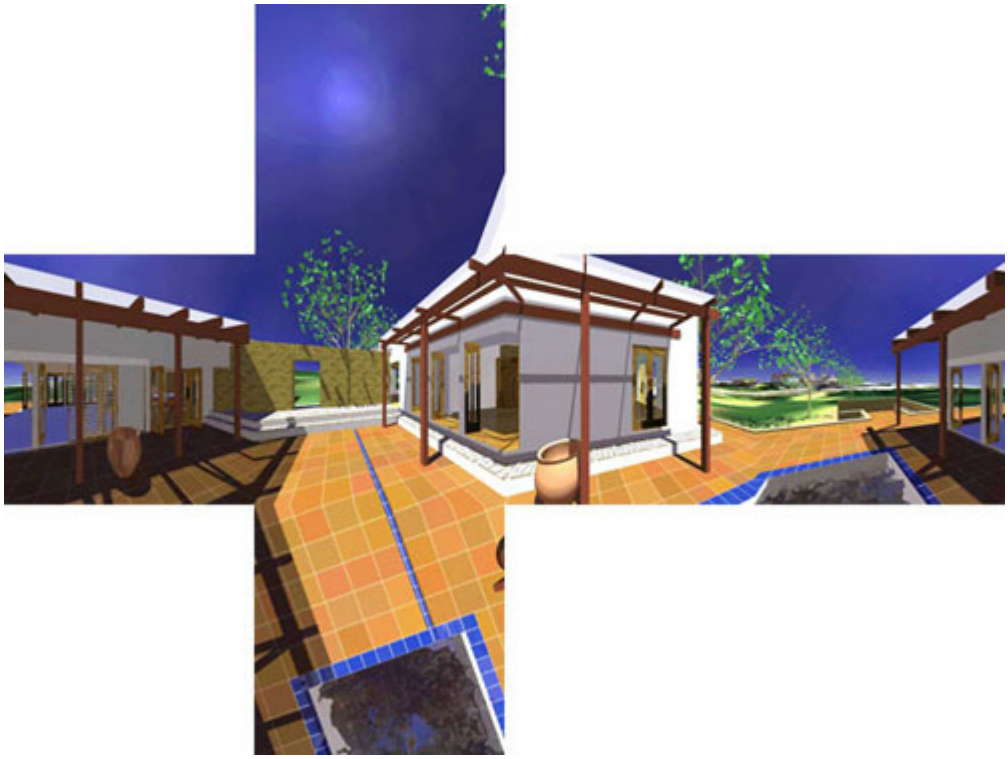


Image copyright Paul Bourke
(http://local.wasp.uwa.edu.au/~pbourke/projection/cuberender/)

- There are several more algorithms that result in only minor improvements over these techniques

Limitations of the rasterization approaches:

- Rendering into a texture can lead to visible jaggies due to incorrect sampling issues (an object that may have been far away from the camera in the reflection map calculation pass, and hence represent a tiny part of the reflection map, might end up being much closer to the camera during game play and hence get distorted).

- Rendering into a texture consumes time. Worst case: In the final image only one pixel of the reflection is visible, but to create the cube map approach the scene has been rendered additionally six times.

- How to handle multiple reflections? Very difficult to achieve, and in most cases just hopelessly impossible. It requires pre-calculating the reflections then rendering the scene, then evaluating the reflections in reflections and then re-rendering the scene and...

- As these reflection maps are not based on physical laws therefore the reflections are not correct anyway.

What all those approaches try to do is an approximation of ray tracing.

As mentioned in the first example: The speed of ray tracing scales logarithmically with the number of triangles. This applies, of course, also for all reflection rays. So in a scene with a very high poly count and with reflections ray tracing can take even greater advantage of this situation (ie single rendering pass, only pay for reflections where they actually happen).

Special Effects: Portals

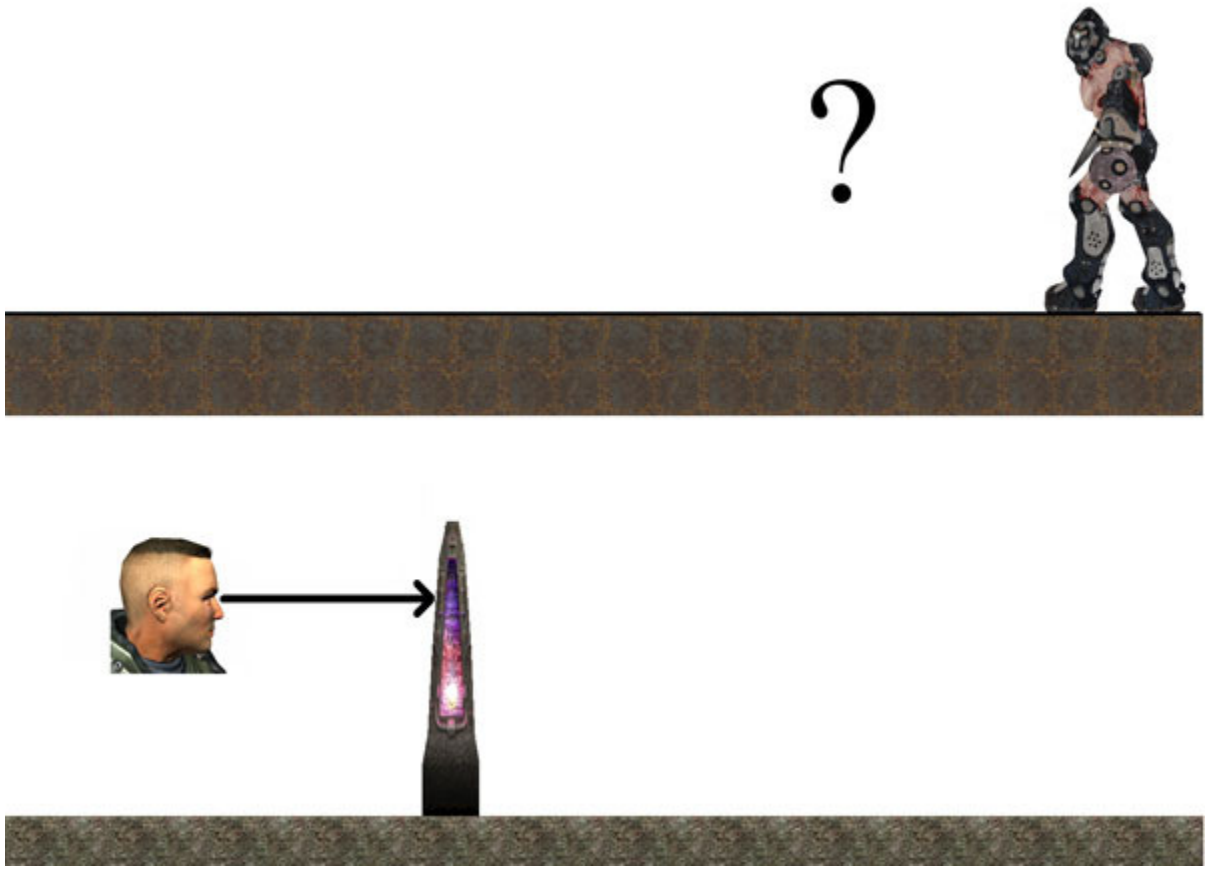
In cinematic image rendering, ray tracing is often used because of its unique capabilities to deliver special effects robustly, where other algorithms fail or are unreliable. The following section will cover two special effects that are relevant to gaming and could increase the fun factor significantly: Camera portals and reflections.

Special effect #1: Camera portals

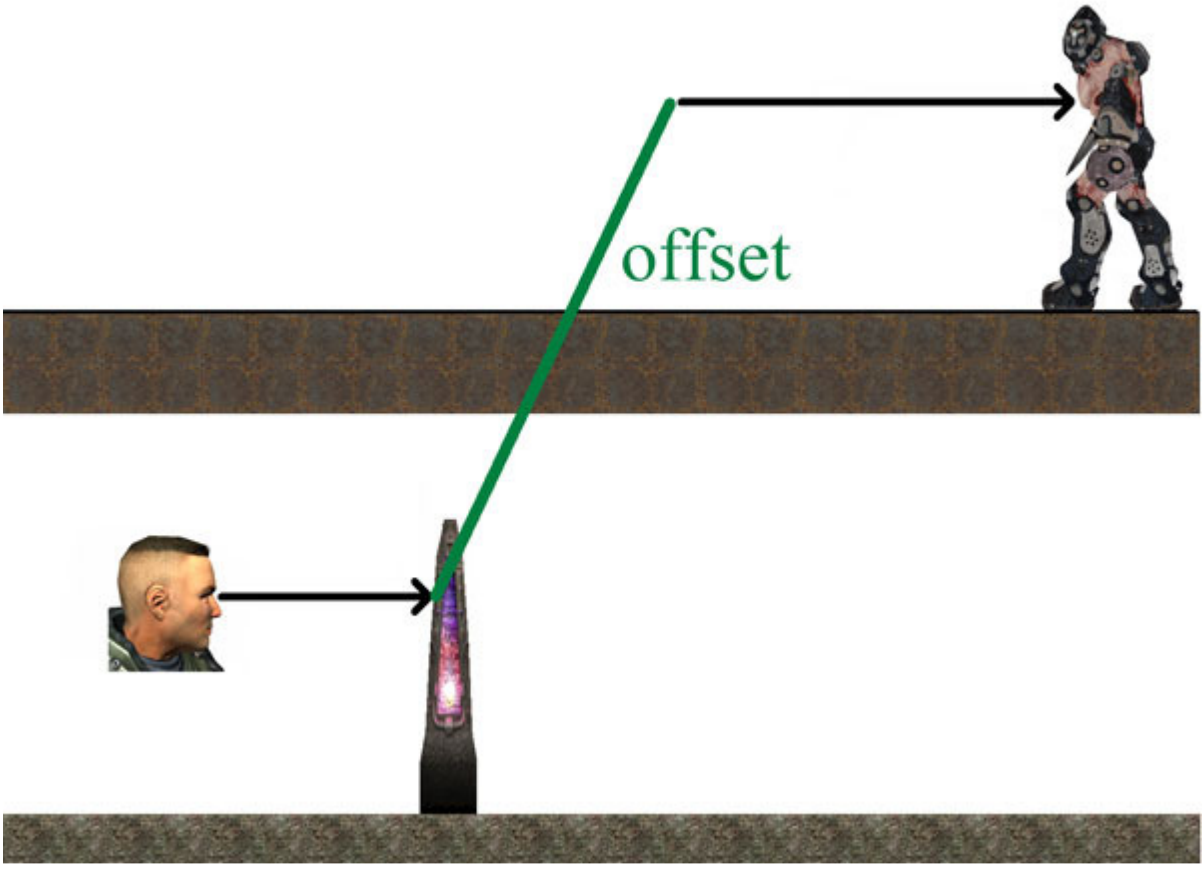
A common feature of 3D shooter games is the camera portal. It tempts the player to see into another part of the scene despite the fact that the player does not have the necessary privileges to participate there.

Camera portal in Quake 3: Arena (1999)

It is interesting to see how easy and efficient this effect can be created using a ray tracer. Let's take a look at the scene setup:



The player is standing on the lower level. In front of him is a camera portal that he is looking through. On the upper level there is an interesting potential enemy character. What we want is pretend that the portal magically teleports us to the upper level. For ray tracing this simply means that we just need to offset the "eye" rays we shoot that hit the camera portal to a new origin and then to re-project it from there as shown below:



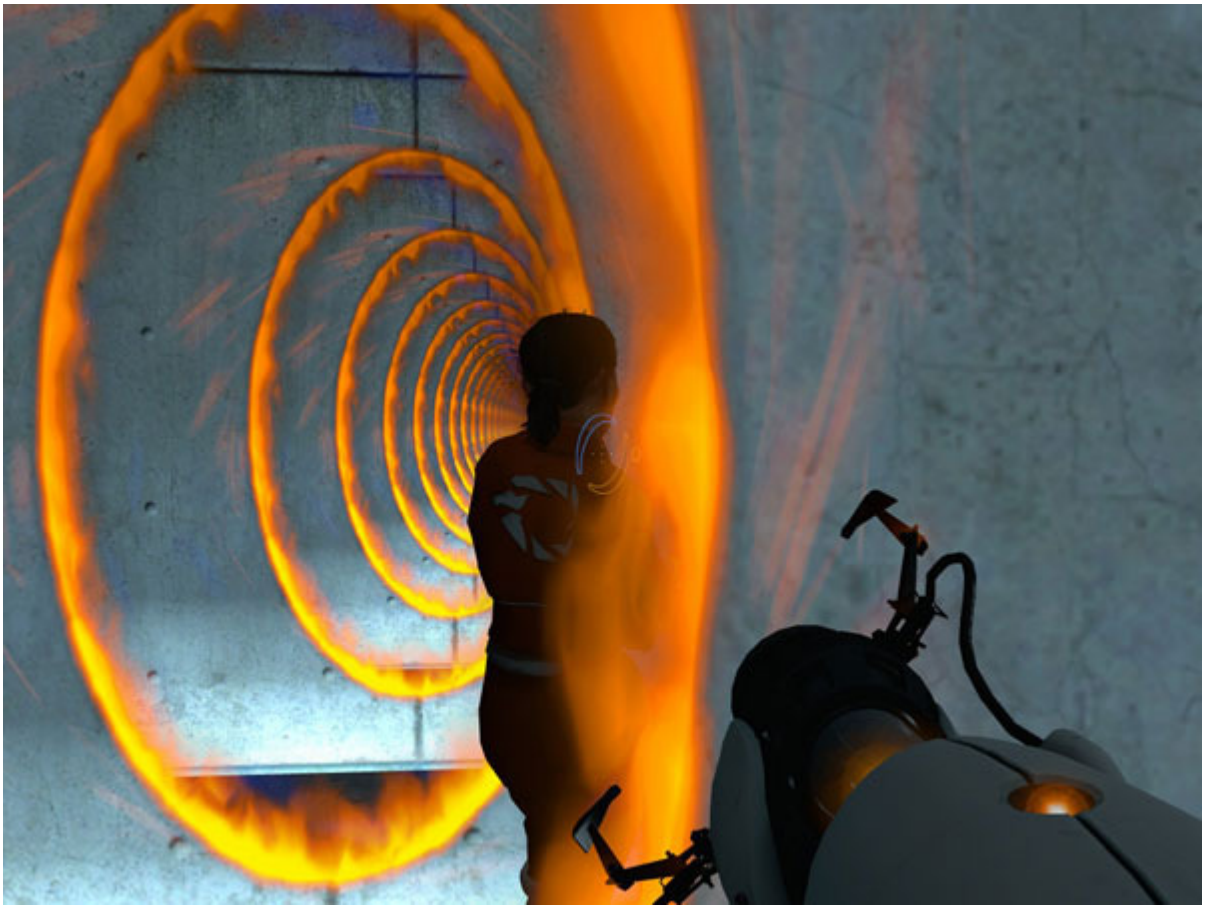
In programming code it looks like this:

```
// finding out at what point in space the ray hit that called
// this shader program
Vector3D hitPosition = getHitPositionOfRay(ray);

// adding a 3D offset to that hitPosition
hitPositon += cameraPoralOffset;

// shooting a new ray from the hitPosition with the offset
// into the same direction as the ray that has called this
// shader program
finalColor = traceNewRay(hitPosition, getDirectionOfRay(ray));
```

Voilà, after only three lines of code you got a camera portal. Here it is applied to Quake 3: Raytraced:

Camera portal in Quake 3: Raytraced (2004)

In October 2007 Valve released a game called "Portal". This game has an innovative game concept: By strategically manually placing portals in a game level, the player is able to magically travel to various locations and solve puzzles. From a technical standpoint it is very interesting to see how they implemented the portal-in-portal effect.

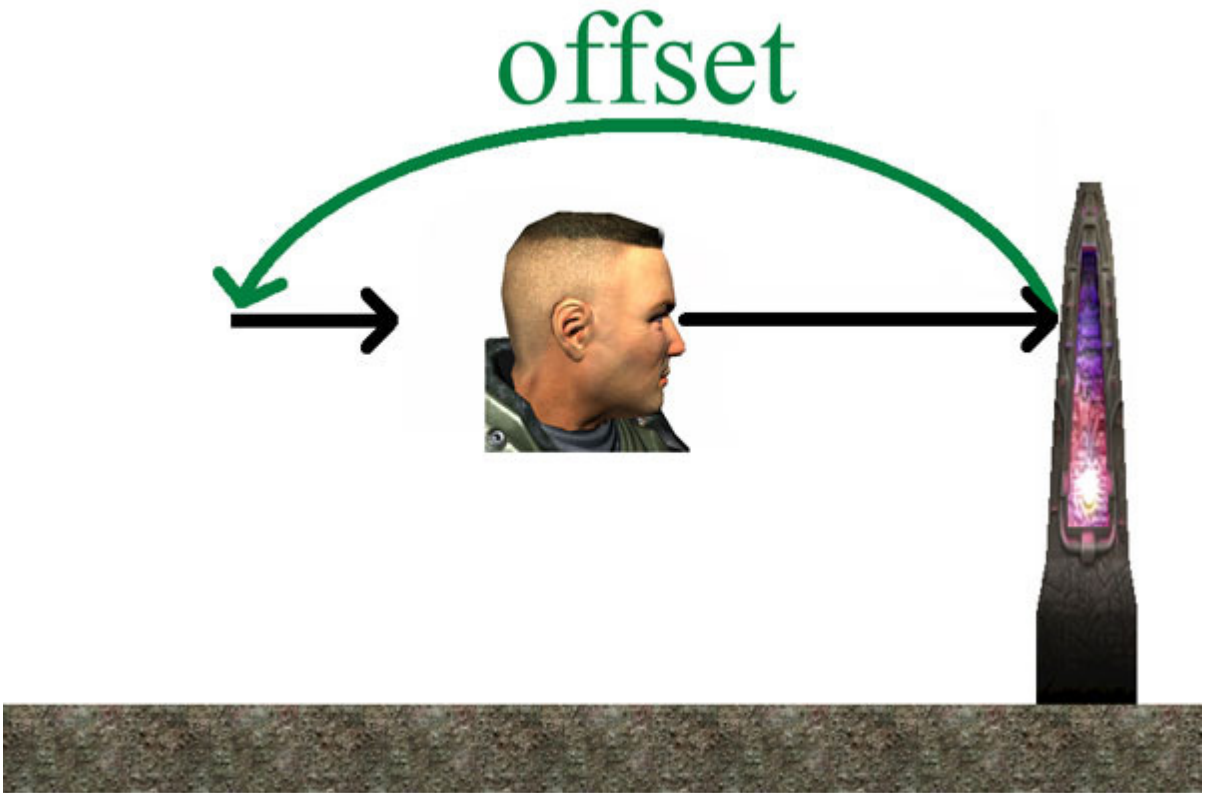


The game "Portal" from Valve (2007)

The implementation of this effect is pretty robust. The game is designed in a way that these portal-in-portal effects never become so close that the recursion depth becomes excessive. Nevertheless, with some experimentation and testing, one is able to detect the limitations of this approach as in the following screen-shot where the recursion depth reaches its pre-determined limit. This can be seen in the center of the camera portal where the box should have been repeated more often.

The visual representation of the box stops repeating itself close to the center of the portal

It is interesting to note how the code for the ray traced version needs to be changed in order to also allow camera portals in camera portals: No changes at all! It still runs with exactly the same code. Just the offset parameter needs to be changed so it looks in front of the camera portal as illustrated here:



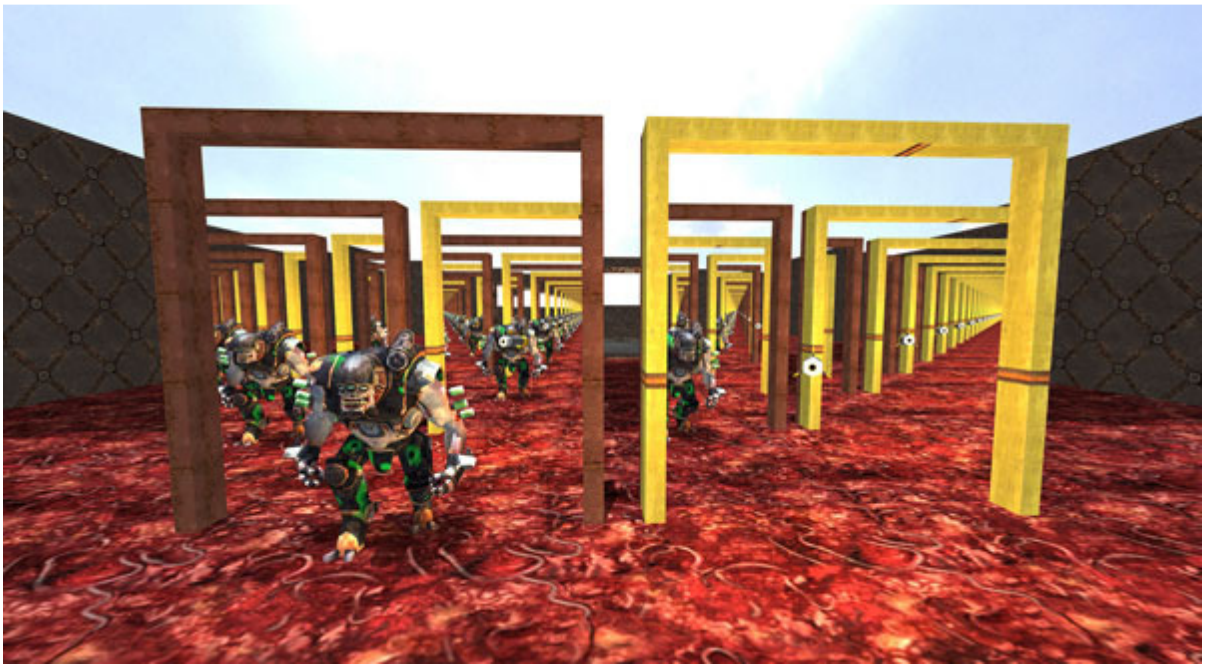
In order to have such an effect in a rasterizer, the image with a "blank" camera portal must be rendered to a texture first. Then a part of this texture needs to be carefully applied to the blank spot. Then again a bit smaller version over the center of the area that was a blank spot before and again and again…

Rendered using the three lines of code from above in a ray tracer, it looks like this:

Scene from Quake 4: Ray-Traced. In the center the # portals in portals is ~130

As noted above, the portal effect is efficient insofar as only those pixels affected "pay" with additional rays to achieve the effect, the exact same applies to reflections. And again, the same principle with reflections applies such that the number of pixels effected at every subsequent recursion gets fewer and fewer, only a tiny percentage of the rays end up having really deep recursions and therefore high calculation costs.

You can afford to do amazing things - like have two camera portals side by side and have both of them visible within each portal – mind blowing stuff! Guess what: Same code!



Q4RT: Two camera portals in camera portals

Special Effects: Gameplay reflections

Special effect #2: Increasing gameplay using reflections to detect incoming enemies

For safety reasons, it is pretty common to hang mirrors on a wall at corners so that people don't run into each other when they are in a hurry to get to lunch for example.

Similarly, hardcore 3D shooter players would like to know what is around corners. Maybe there is an enemy approaching? Perhaps, an ally? With ray tracing it is a very easy task to just place such a mirror in place and enjoy the increased possibilities in game-play.

Quake 4: Raytraced – Putting a mirror in the corner to see into another hallway

Here is a video where the additional information about an approaching enemy in another hallway, gathered through such a mirror is taken advantage of. Only in this way would the

player be able to predict the arrival of the monster and to place a rocket into it's direction at the right time.

Download 1280x720 XviD video here
Download 1280x720 WMV video here

Hybrid Rendering: Combining Ray tracing and rasterization

Hybrid Rendering: Why it is a bad idea and why it will be done nevertheless.

How will ray tracing for games hit the market? Many people expect it to be a smooth transition – raster only to raster plus ray tracing combined, transitioning to completely ray traced eventually. They think that in the early stages, most of the image would be still rasterized and ray tracing would be used sparingly, only in some small areas such as on a reflecting sphere.

It is a nice thought and reflects what has happened so far in the development of graphics cards. From the initial fixed function pipeline some minor parts were opened up to programmers. Going from very hard to program "register combiners" to vertex and pixel shaders. Then these were improved over a long period of time in very small steps, resulting in a smooth transition from one advancement to another.

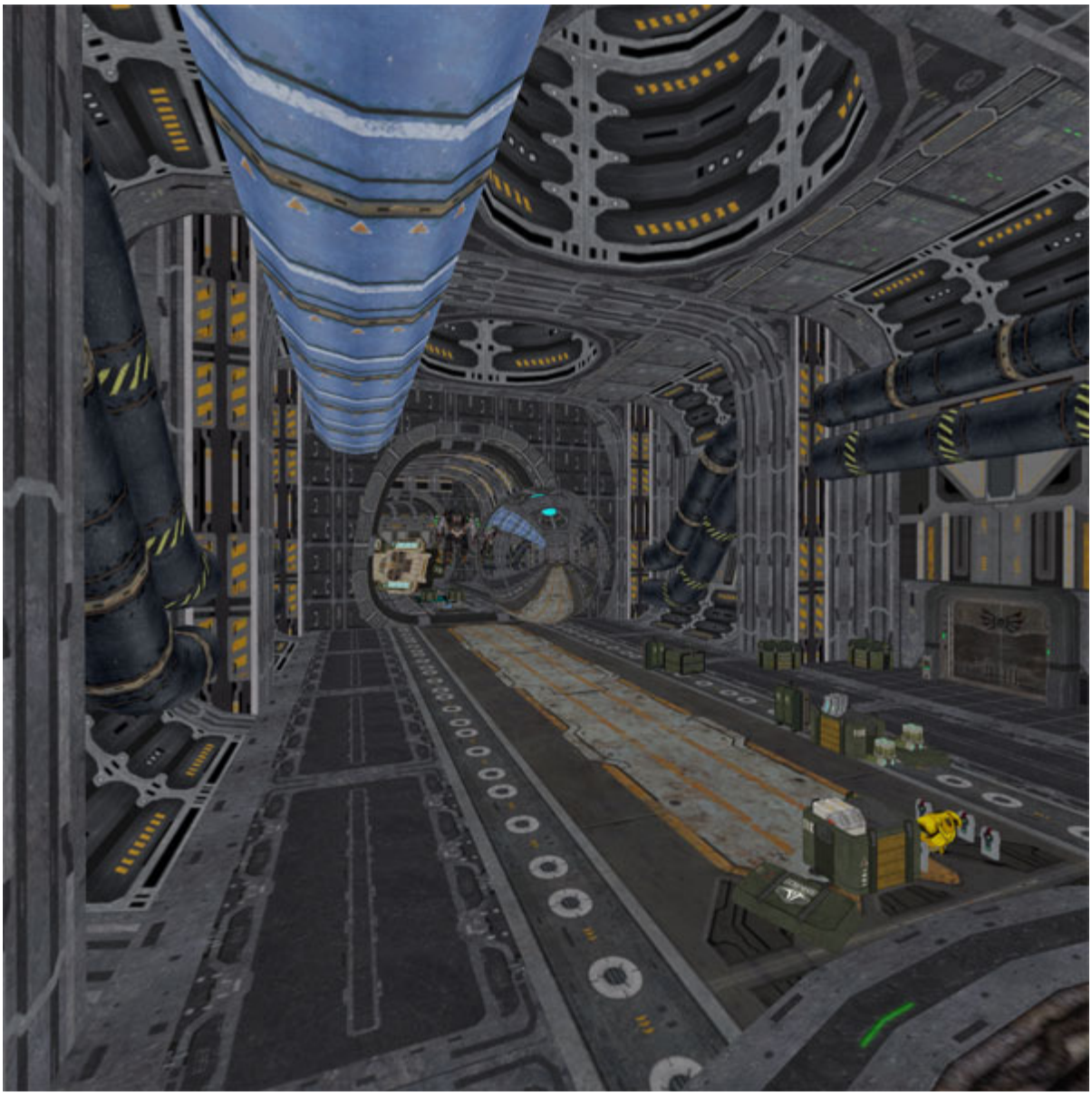
Pixel shader version	DirectX version	Year
1.0/1.1	8.0	2000
1.2	8.0a	2001
1.3	8.0a	2001
1.4	8.1	2001
2.0	9.0	2002
2.0a	9.0b	2003
2.0b	9.0b	2003
3.0	9.0c	2004
4.0	10	2006

Source: http://de.wikipedia.org/wiki/Pixel-Shader

So it is understandable that people have it in their minds that ray tracing should be introduced in small steps. The only problem is: Technically it makes no sense.

Scene example 1

Let's assume a typical game scene setup from Quake 4 with one additional reflecting sphere that is far away like on this screenshot:



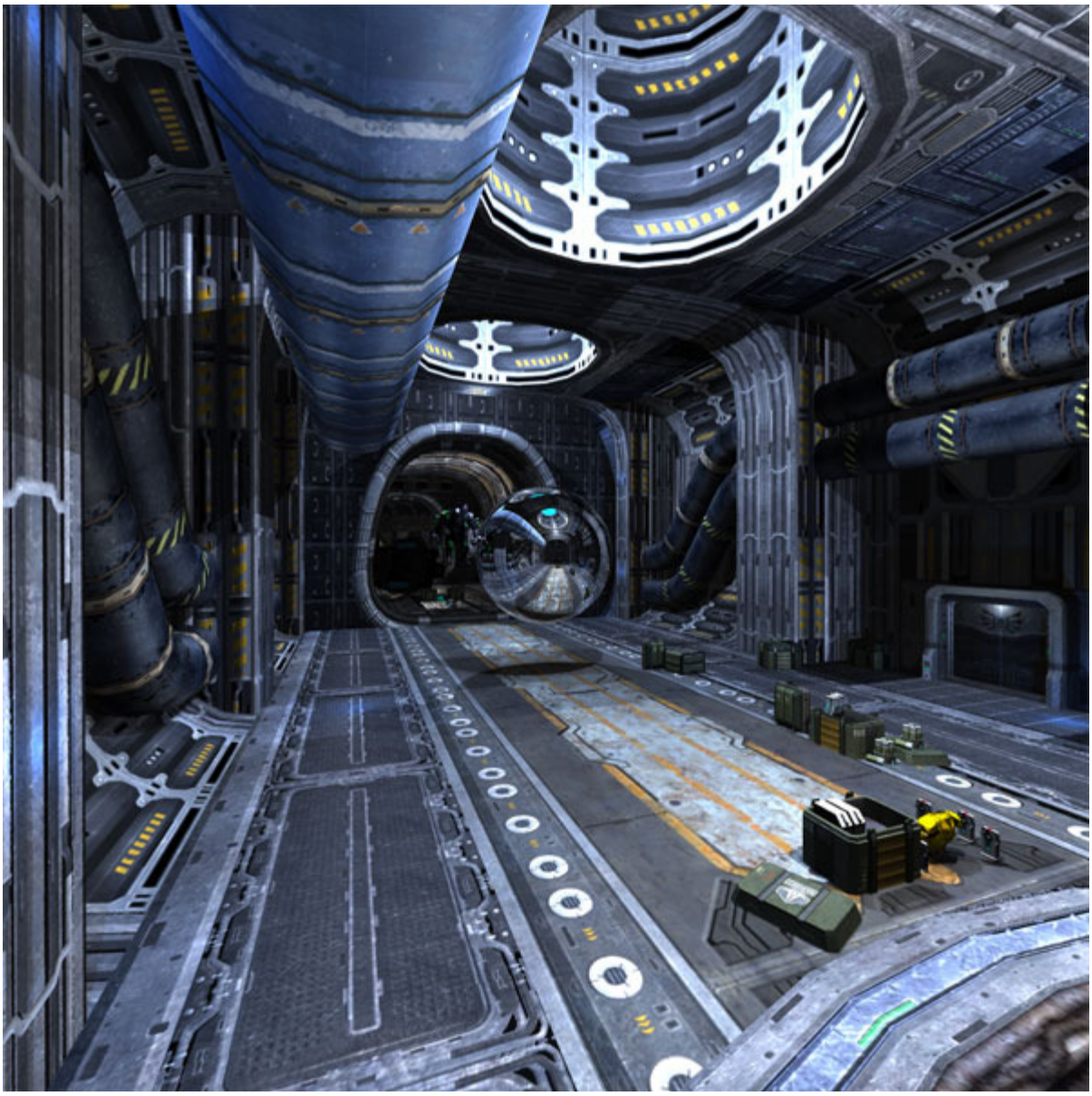
One could render everything but the sphere with rasterization and then do only ray tracing for the sphere. In order to do that efficiently one would still need:

- A fast ray tracer
- Up-to-date acceleration structures (e.g. BSP-tree, BVH-tree) for the entire scene

If the world was complex you would potentially have to rasterize far more triangles than the number of rays that would be used instead, but let's put this aside for now. In this scenario we "saved" from having to calculate for all the primary rays under the assumption that the rasterization algorithm will resolve all the primary visibility queries.

Scene example 2

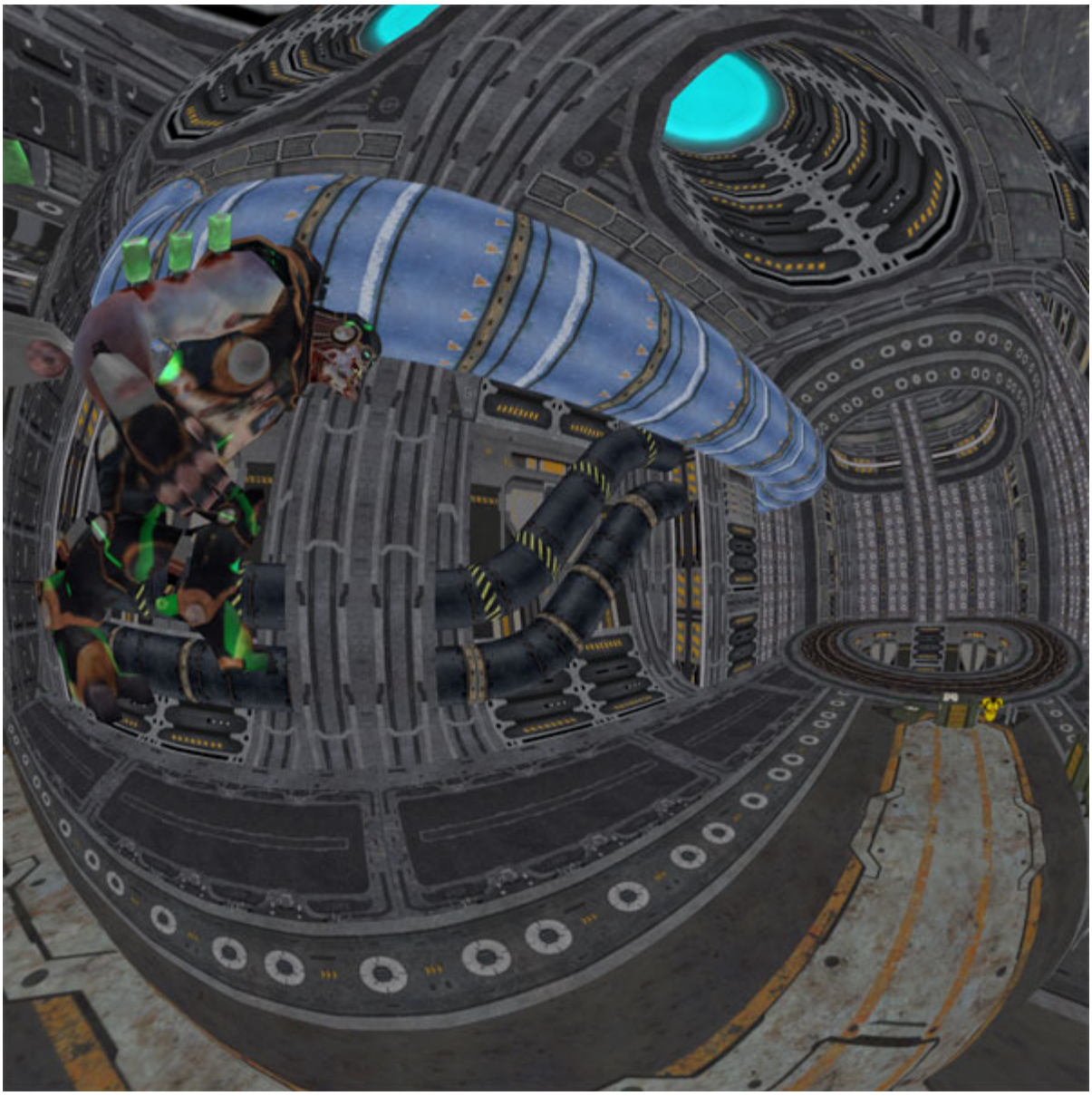
The upper screenshot looks quite boring, because it has no lights and no shadows in it. In the lower example each pixel is being lit by an average of 3-5 light sources simultaneously.



If we use ray tracing to render all the shadows accurately, then each pixel would ordinarily require one visibility ray and 4 shadow rays. If we were to use rasterization to achieve the same quality image, it would only save us the 1 visibility ray per pixel. Rasterization techniques simply are not robust enough to achieve the same kind of accuracy and fidelity as ray tracing for shadows, reflections and refractions. If you imagine a trend toward higher and higher quality, the savings of using rasterization for the visibility determination trends towards insignificance.

Scene example 3

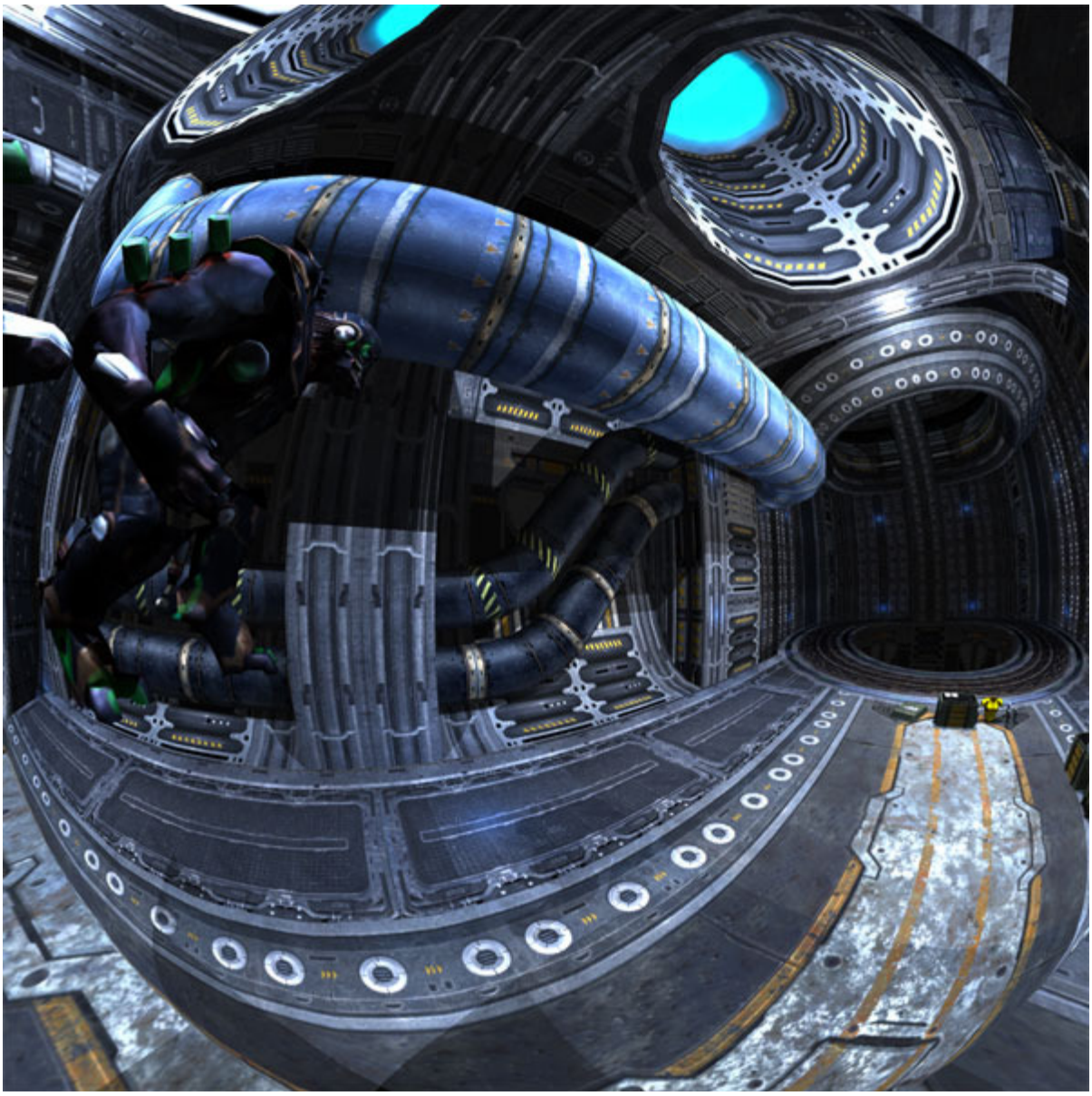
The previous examples showed the reflecting objects from afar. But what if one zooms in on such an object so that it fills almost the entire screen? In a game environment where the player is capable of moving around freely, such situations are unavoidable.



Now about 90% of all pixels need to be ray traced. Therefore the hybrid rendering approach could only "save" 10% of the primary rays.

Scene example 4

I'll leave it to your imagination to figure how this approach would work out when the reflection would now also contain lights and shadows…



This leads us to the conclusion that a hybrid rendering approach for rendering the 3D world might not be the best idea. The only useful application of such a hybrid approach that I can think of would be for layering 2D data like a HUD, a console or maybe even a 3D perspective of a cockpit, but not for the actual 3D scene computation.

Fonts and crosshair layered over the ray traced image with semi-transparent quads in OpenGL

Nevertheless I would not be surprised to see demos of this approach from people who want to push the idea of hybrid rendering. But I doubt this will make it in a really convincing way into a commercial game.

Stay tuned for more information and thank you for reading this article!

About the author

Daniel Pohl is a Research Scientist at Intel Corporation currently based in Germany. He joined Intel's graphics group after completing his master's degree in computer science at University of Erlangen. During his time of study he adapted the ray tracing algorithm to the games Quake 3 and Quake 4. His main areas of expertise are 3D graphics and game development.