



# Intel<sup>®</sup> Technology Journal

Compute-Intensive, Highly Parallel Applications and Uses

## Ray Tracing Goes Mainstream

# Ray Tracing Goes Mainstream

Jim Hurley, Corporate Technology Group, Intel Corporation

Index words: ray tracing, global illumination, ambient occlusion, immediate mode API, retained mode API, photo-realistic rendering, physically correct photo realistic rendering, occlusion culling

## ABSTRACT

We present an introduction to the rendering technique known as “ray tracing.” We propose that its performance has reached the stage where it is feasible that it will take over from raster graphics in the near future for interactive gaming and other application domains. We investigate various aspects of ray tracing and compare and contrast them with the raster equivalent. Finally, we analyze ray tracing’s platform requirements and scalability potential.

## INTRODUCTION

Ray tracing is the act of tracing the trajectory of a ray from one point to another to determine if anything is hit and the distance to the nearest hit point. Although for our purposes ray tracing can be thought of as a “workload,” in the larger graphics world, ray tracing is considered to be a tool. Rendering systems use a variety of such tools to achieve their goals. In almost all graphics workloads, the rendering portion consumes >90% of the available resources. Nowadays, more and more of the techniques used in photo-realistic imagery are based on ray tracing.

## HOW RAY TRACING IS USED

The following is a partial list of how ray tracing is used:

- Visibility testing is used to determine if there is an unobstructed path from A to B. “Eye rays” are shot from a camera to determine what can be seen. This is known as inverse/reverse ray tracing.
- Illumination testing is used to determine if there is an unobstructed path from A to a light source. This enables us to determine very precise and accurate shadows and illumination.
- Perfect reflection and refraction: subsequent ray trajectory based on the properties of a material struck by a visibility ray. (Very few real materials exhibit such perfect properties.)
- Diffuse or anisotropic reflection and refraction gives a more realistic determination of the consequences of a ray intersecting with a realistic material. Typically,

a “shader” is invoked whenever a ray strikes a surface. This shader then determines some distribution and “weights” of subsequent reflected and refracted rays in various directions.

- Light transport determines how light flows from the various light sources in an environment to one or more “cameras.” Recall that that we mentioned that “eye rays” are shot from the camera into the scene to determine what is visible. These “forward” rays travel in the opposite direction. Some light will flow in such a way that it does not impact the image seen by the camera (i.e., may or may not bounce off various surfaces in the scene, but nothing that the camera sees is directly or indirectly effected); other light may directly influence the image seen by the camera, and some other light may indirectly influence the image seen by the camera.
- Ambient occlusion is a “trick” used extensively by Pixar and other movie-production studios. An assumption is made that the lighting environment consists of a horizon-to-horizon hemisphere of uniform illumination intensity. Whenever an “eye ray” intersects a surface, some number of visibility rays are shot over a hemisphere sample space. The rays are shot to determine how “exposed” the intersection point is to the “sky.” The farther more of the rays travel before hitting anything, the more exposed the intersection point. The “weight” of these feeler rays is used to determine an ambient intensity for the intersection point. Ambient occlusion produces a pleasing look to the image because of its soft intensity transitions. Note that no consideration of material properties, or actual light placement etc. is taken into account. Usually an ambient occlusion map is produced as a result of this pass, and this map is considered the base layer to which other lighting layers are added; sometimes, this pass alone is performed.



**Figure 1: Image demonstrating Ambient Occlusion.**  
Copyright © 2003 Pixar [1]

## HOW RAY TRACING IS USED WITH OTHER TOOLS

Photo-realistic rendering, the ultimate goal of all graphics, is achieved by using a variety of tools, several of which use ray tracing directly or indirectly. Ray tracing can easily be used to determine the direct illumination of a given scene. Indirect illumination is more difficult, as not only light paths are important, material properties play a significant role also. However, even in indirect illumination, ray tracing techniques can be employed. Some techniques effectively allocate a budget of rays dedicated to forward tracing from light sources; the remainder of the rays are used in the usual fashion of inverse tracing from the camera. Forward ray tracing is used to trace the path of light as it emanates from various light sources, strikes various surfaces in the scene, and reflects, refracts, etc. from surface to surface. As the rays land on various surfaces, material shaders are invoked that determine how the light energy is absorbed, reflected, refracted, scattered, etc. from the surface, and at each such spot the color at that point is stored in a cache. Subsequently, during the inverse tracing phase, the cached photons are effectively treated as a larger collection of light sources. Ray tracing itself does not solve the problem of creating *photo-realistic* images; however, it is an important tool that is used extensively in conjunction with a wide variety of other tools.

True photo-realistic rendering requires solving a “global illumination” problem, namely that everything in a scene affects everything else in the scene (to some degree), and that indirect illumination is vitally important, even though it might only have a subtle effect on the final image. It is this subtle effect that makes the difference between a false-looking image and one that looks “real.” (The goal is to achieve the effect called “suspension of disbelief”; in

other words, the imagery created can be intended to be a cartoon or “live action”).

## Raster Graphics and Ray Tracing

Ray tracing and “raster graphics” can be used to attempt to solve the above-mentioned global illumination problem. In fact, ray-tracing techniques can achieve the exact same results as raster-based techniques (including all the approximations and tricks that raster solutions typically require); however, it does not work the other way around. Both of these approaches have their advantages and disadvantages, and both work in very different ways with very different system implications which we summarize here.

### Raster Graphics

The primary differentiating factor between raster- and ray-tracing approaches is that a ray tracing approach enables one to solve a global problem, while a raster-based approach seeks to achieve similar results by solving a local problem. Raster graphics attempts to render an image efficiently by making certain convenient assumptions. In particular, it treats triangles as if each triangle is entirely independent of every other triangle. Raster graphics hardware is capable of achieving extremely high throughput. However, the triangles are not independent, and in fact, this presumption places a lot of restrictions on the rendering system. Modern raster graphics APIs work in “immediate mode” where there is an expectation that the raster engine renders each triangle or command upon receipt. There is a concept of the current state and the current triangle. This is the Graphics Processing Unit’s (GPU) view of the entire world; it has no idea if or what comes next. Modern raster systems are extremely efficient, and some of the above-mentioned limitations can be worked around some of the time: for example, by rendering in multiple passes, or by employing a variety of approximations, tricks, etc. to leverage the tremendous performance of these devices. However, these approximations and multi-pass approaches impose limitations that Independent Software Vendors (ISVs) either have to live with or learn to avoid.

Raster and ray-traced systems have different cost functions and scaling characteristics, listed below. In general, due to the way that raster systems work, they process every pixel of every submitted triangle to determine the final image that needs to be displayed. Raster graphics performance scales strongly with the number of triangles and pixels that have to be processed for a given image, so cost scales roughly linearly with viewport size and overall scene complexity, as follows:

- If a scene requires 100M triangles to be submitted for rastering, and each has an average of 10 pixels, the

raster system has to process 1B pixels. That is if it only takes one rendering pass.

- ISVs very carefully manage the complexity of their content, and various occlusion-culling techniques are used to minimize the quantity of submitted triangles.
- Because geometry must be grossly simplified to avoid the above performance problems, various techniques are used to create the illusion of more detail than what actually exists.
- Texture mapping is used to simulate extra detail. In fact, multiple layers of textures are often used.
- As mentioned earlier, part of the cost function is related to the number of pixels that have to be rendered. Often, multiple textures get applied per pixel, and each layer of texture can require many samples from the texture buffers; consequently, the bandwidth requirements of raster-based solutions can be astronomical.

Raster graphics assumes that the triangles in a scene are independent of each other. This allows hardware to process each triangle independently and even to process multiple triangles and pixels simultaneously. But, in fact, the triangles are not independent: triangles can cast shadows on other objects or other triangles in the same object, but also triangles can be translucent, and they can reflect and refract light, and so on. Raster systems can get around some of these limitations using a variety of tricks. The tricks, such as those listed below, usually work under certain circumstances, and those situations where they fail must be avoided in order to preserve “suspension of disbelief.”

- Complex lighting effects, caused by objects reflecting and refracting light, can be simulated by running a real offline global illumination solution and extracting the results and storing them in maps. This can lead to plausible images being generated. However, these light maps are captured at a point in time, and with a particular arrangement of all the lights and objects in a scene, that they do not accommodate dynamic lighting situations.
- The Z buffer is used to perform a binary test to decide if a point on the screen covered by a new triangle is closer to the camera than the same point on the screen covered by a previous triangle. However, this makes the assumption that all triangles are opaque. Where translucency is involved, the translucent objects have to be separated from the rest of the objects, and all the objects need to be rendered in a particular order to avoid artifacts.
- To compensate for lower triangle counts, various bump maps and normal maps are used to create the illusion of increased complexity. Texture maps were originally intended to represent the micro-detailed

texture of a surface; however, raster solutions often use texture maps to represent macro-level features. Lower precision models cause lots of problems: silhouette edges are blocky, and it is very difficult for artists to get the look and feel of what they are striving for. When the models are viewed from shallow angles, it becomes apparent that the surface details are not really there; the ISV has to work hard to understand all the limitations and to avoid those situations where these techniques break down.

- Shadows are another issue. Firstly, shadows are critical; images without shadows appear to have objects floating in space. Shadows are important visual clues that help associate an object with the surface that it is on or above, etc. Raster solutions don't really handle shadows because of the independent triangles presumption; instead raster techniques emulate shadows. There are a variety of ways of doing this: some require rendering the scene many times from the point of view of each light, and some involve determining the silhouette edges of an object from a particular light's point of view and casting rays through these silhouette edges to form so-called shadow volumes, etc. Each of these techniques works after a fashion, but all have various artifacts and restrictions. In fact, in one raster-based game we investigated recently, we found that there were five different shadow algorithms in use, and the ISV had to pick which one to use, on an object-by-object basis.

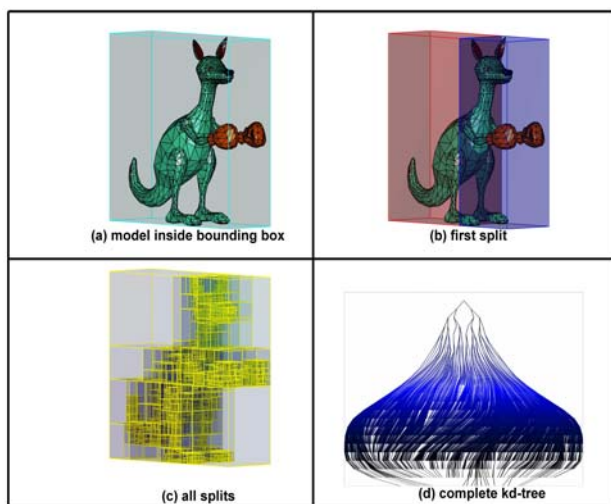
## RAY TRACING

Ray tracing, and by extension, global illumination does not suffer from these limitations just outlined, but it has its own set of problems that we discuss here. Firstly, there are fundamental differences between raster and ray-tracing approaches. Because ray tracing takes a global approach, the natural interface to it is different than that used with raster graphics. Recall that raster graphics use an “immediate mode” interface, and is only aware of a current state and a current primitive at any point in time. Ray tracing, on the other hand, needs a “retained mode” interface where random access to the whole scene is required, and when a visible triangle is determined, a specific shader needs to be invoked on demand.

Ray tracing relies on a so-called “acceleration structure”; this is organized as a spatial partitioning or indexing structure. Any given object or scene is decomposed into regions of empty space and finer and finer partitioning of filled space. This structure allows us to efficiently determine the path a ray would take through space, and to test it only against those triangles that would be in the vicinity of the ray's trajectory. Otherwise, we would need to test every ray against every triangle to test for any

possible intersection. If the scene is composed of rigid bodies, or articulated rigid bodies, the acceleration structure can be constructed in advance and loaded into the application along with the scene's geometry. If the scene is dynamic, then the acceleration structure may have to be built on the fly, per frame, which can be very expensive. We have found that kd-trees make the best acceleration structures. If one needs to build an acceleration structure, with a very large model, the algorithm (greatly simplified) goes something like this:

- Read in *all* the vertices, think about them for a while, then propose a (single) split based on some cost function. This effectively gives you two sub-trees.
- Repeat the previous step for each and every sub-tree you create until certain termination criteria are met.
- You will notice that, at the higher parts of the tree, traditional caches won't help much. However, once a certain threshold is reached, where the entire sub-tree fits, constructing the lower parts of the tree gets easier and easier.
- Conversely, you can see that if you are given the higher parts of the tree, building the rest of the tree is relatively inexpensive. Unless the object literally comes apart at the seams, the top-most parts of the tree rarely ever change.
- Often, in a gaming scenario, for example, other parts of the application require some sort of spatial partitioning structure (such as the occlusion-culling engine, the physics engine, the collision detection engine, etc.). If all of these used the same structure that the ray-tracing engine requires, even if at lower precision, this would greatly facilitate those cases where the tree needs to be built on the fly.



**Figure 2: An example of a model, and how it is decomposed into an acceleration structure**

Building these trees, and building them well is a huge topic all by itself. We have implemented algorithms that result in incredibly good trees, but it would take a separate white paper to describe the techniques used in detail. Also, separate research has begun in areas related to lazily building such trees (only build what's needed, when it's needed) and building them to a sufficient (i.e., minimal) level of detail. Finally, the very best acceleration structures are built using a cost function that trades off a given platform's computation and memory system's characteristics: these structures need to be built on the platform that they will get used on.

The next issue to tackle is the platform cost of tracing a ray. This requires traversing the acceleration structure in a serialized sequentially dependent fashion until the ray finds a leaf node. Every time we traverse the structure it will be for a different ray, but if you shoot rays in a spatially coherent fashion, most likely the rays will take the same path through the structure. The (simplified) traversal algorithm is as follows:

- Test the ray against the split plane defined for the current volume of space.
  - Perform a simple test (a few simple ops and a compare).
- Determine if the ray goes cleanly to one side or the other of the split plane, or passes through it.
  - Perform a data dependent unpredictable branch.
  - Go to the “left,” “right” or both sub-node(s).
- Move onto the next node(s) and repeat.

Once a leaf node is encountered, we need to perform a computationally intensive ray-triangle intersection test for every triangle in the node. Even then, the ray might miss all the triangles there (hence it is best that leaf nodes hug the boundary of an object as tightly as possible). We have optimized these algorithms extensively, and we have optimized the acceleration structures to minimize the number of traversal steps to a leaf node, greedily accounting for as much empty space as possible, and we have minimized the number of triangles in each leaf node, for **any** given platform. Remember, it is cheaper to test a ray against an empty space than to test it against a bunch of triangles in a cell and find out that the ray misses them all. We have figured out how to shoot arbitrary groups of rays as a beam, performing most of the traversal using the beam instead of all the rays in the beam. We use vector approaches where they make sense: testing the four planes that represent the limits of a beam against each split plane, testing the triangles in a leaf node against each ray that gets that far, etc.

The result of using beams is that a lot of the time, the beam finds a very deep “entry point” in the kd-tree for all

the rays in the beam. Often, this entry point is at the leaf node itself, meaning that the bulk of the work is now computational. Also, as we report later, we find that we get really excellent cache hit rates, which dramatically lowers the external bandwidth cost of using ray tracing (assuming traditional CPU-style cache hierarchies).

However, the really exciting news is how the ray tracing workload scales. It is strongly effected by the number of rays shot in a scene and weakly effected by the complexity of the scene, which is different than raster graphics. Recall its performance scales with the number of triangles and pixels rendered, which is a function of the scene complexity and the overall viewport size. In contrast, ray tracing scales linearly with the number of rays shot and only logarithmically with the complexity of the scene. For a fixed resolution image, the cost of raster graphics doubles (roughly) as the complexity of the scene doubles; for ray tracing, you would have to increase the *viewed* scene complexity by 10x to double the cost. We have found that even with today's hardware (HW) raster accelerators, a single CPU running software (SW), ray tracing will catch up with the HW raster engine around the 1M triangles per scene mark, and will always outperform it above that. We have also discovered that the performance of ray tracing scales linearly with the number of CPUs. Another observation is that the performance of ray tracing is not so much overall scene complexity dependent, but dependent upon the *visible* complexity of the scene. Imagine a world where there are millions of triangles, but only 50K are visible at a time (such as a building). Given the above mentioned beam concept, the ray-tracing algorithm will quickly zero in on just those parts of the overall structure where the visible triangles are, effectively, shrinking the tree to just that part. Therefore, ray tracing performance scales with the observed complexity, rather than the overall complexity. Effectively, the acceleration structure behaves like an infinite level of detail occlusion-culling mechanism.

### ISV Implications

ISVs can build complex models, without having to fake details, or have harsh polygonized outlines, silhouettes, etc. Moreover, they do not have to compromise by trading off model and scene complexity against overall performance. And finally, turning on more and more features just results in more rays:

- Visibility is determined by shooting  $(1 - n)$  "eye" rays per viewport pixel (depending on how much anti-aliasing is needed).
- Infinitely precise shadows cost 1 ray per eye ray/triangle hit point, per light.
- Exact reflections, refractions are 1 ray each (times the number of subsequent bounces that are permitted).

- For translucency, keep refracting rays through translucent surfaces until an opaque one is hit, or some saturation point is reached.
- For anisotropic reflections/refractions. a budget of subsequent sample rays per surface struck must be allocated, which can be bound by the limits of new rays per bounce, and the number of bounces, etc.
- For global illumination, use more rays: forward rays and regular rays. In essence, everything boils down to rays.

All of the above work exactly as expected: there are no corner cases where they don't. The cost of a scene can be calculated very accurately and parameters can be tweaked to hit frame rates. The platform implications boil down to how coherent those rays are: eye rays can be engineered to be highly coherent as can shadow rays. This is less so for reflected and refracted rays. Not only can an ISV budget exactly what he or she can (or cannot) afford given a target platform and other application parameters such as model complexity, etc., but an ISV can safely use these features individually or in combination:

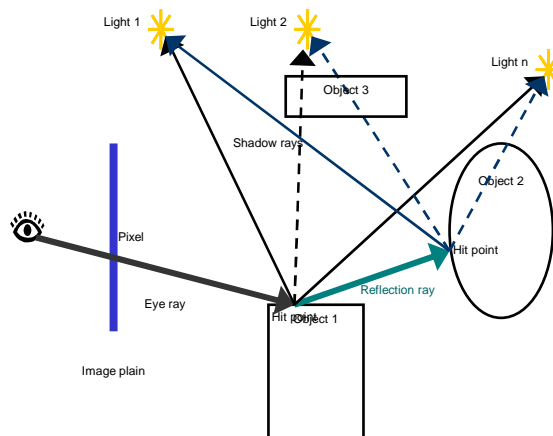
- For one thing, shadows are no longer an issue: they are on all the time for everything. There is no need for tricks, or hacks. Different algorithms don't need to be chosen selectively for individual objects, and there is no need for multi-pass approaches. Moreover, those shadows are exact, perfect, and always right, and everything shadows everything else: a character's nose casts a shadow on its face, and dimples in the nose have shadows inside them. If the ISV wants a shadow to fall a particular way, he or she can engineer that by selecting a mask to indicate which lights to seek shadows from.
- There are no multiple passes of rendering from virtual cameras behind reflective surfaces, and no need to texture the result into the rendering pass. No environmental maps are needed that only comprehend infinitely distant scenery. And, multiple reflections work—naturally.
- Translucency is no longer a problem. There is no need for sorting of geometry or for careful ordering of what gets rendered first. It all just works. Even if the desire is to have everything be translucent and there are sufficient rays available, this can also be done.
- All of the above-mentioned techniques work together in one pass without the need for complex sorting of what needs to happen first, or for what limitations exist.

Everything else that gets used in raster graphics, such as multiple texture mapping for simulation of fine detail surface texture, still works, and all the sampling and

filtering that goes with that. All the hacks and tricks used to simulate complex lighting like pre-rendered global illumination solutions etc. can still be used. In fact, you can create exactly the same results as a raster platform using ray tracing instead. However, unless *all* the hacks and tricks that simulate complexity, shadows, and complex lighting are used, then fine details will go missing: triangles that appear to have surface detail and shadows, when rastered, will actually appear flat and featureless and have weird colors that don't appear to belong there when ray traced, if all the aforementioned tricks are not used.

## RAY TRACING PERFORMANCE

The basic core of a ray-tracing engine is the act of shooting a ray, and that, in turn, depends heavily on three key algorithms: acceleration structure traversal, ray triangle intersection tests, and an arbitrarily complex “shader,” invoked if there is a ray-triangle hit. We determined that we needed to establish a “benchmark” by which we could gauge performance: the number of ray-segments per unit time. A ray segment is one leg of the journey of a ray. Each “bounce,” if you will, also each ray shot at a light to determine if the current spot is in shadow, is also a ray segment. We determined that overall performance could be characterized in terms of some aggregate number of ray segments per second; there were three main dimensions:



**Figure 3: Diagram illustrating how a single pixel's rays decompose into eight ray segments**

1. The number of rays shot per pixel—a quality metric. More raysegs per pixel account for more and more lights, bounces, sampling, etc.
2. The number of pixels per frame (grows in discrete steps, 640x740 to 1024x768 ...etc.).
3. The number of frames per second (30fps for movies, 75fps for games).

We determined that 450M raysegs/S was the threshold where real-time ray tracing becomes interesting. We

assumed a frame rate of 30fps, an image size of 1M pixels, and 15 raysegs per pixel. This could quickly escalate (linearly) if 75fps, 3M pixel displays, and higher quality is taken into account. We measured the computation and raw and external bandwidth required for a variety of scenes (recall that performance is viewport and scene complexity dependant) and found that the per ray segment cost was 1500-3000 FLOPS and 600-1400 raw bytes, with cache hit ratios of 300-1200:1. We were able to achieve up to 100M raysegs/S for simple models and large viewports on desktop machines (a 3.2 GHz Pentium® 4 processor), and measured linear performance scaling up to 128 CPUs in cluster configurations.

## Scalability of the Ray Tracing Algorithm

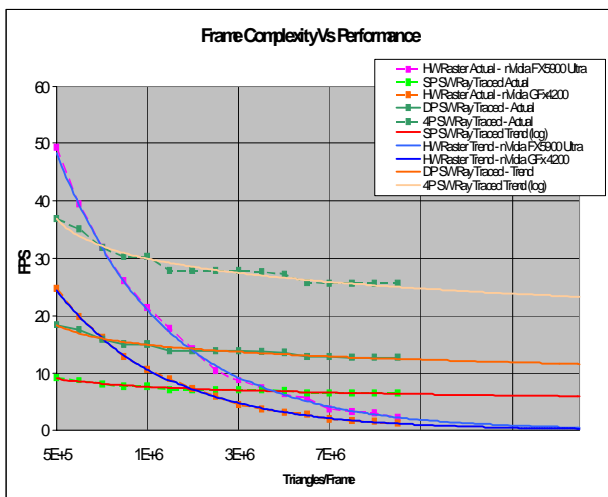
Ray tracing scales strongly with the number of actual rays that need to be traced and weakly with the overall complexity of the scene being rendered. The number of rays that need to be traced depends on the following conditions:

- The raw dimensions of the viewport (i.e., the number of pixels). Over sampling or anti-aliasing would increase the number of rays per pixel shot into the scene.
- The frame rate (24fps for movies, 72-75 Hz for game content).
- The number of active lights. If a ray hits an object, rays are shot from the hit point to all of the active lights to determine if there is an unobstructed path.
- The number of bounces allowed per ray. Some engines place a cap on the number of subsequent bounces caused by reflection or refraction, as each such bounce contributes less and less to the value returned to the eye.
- The sophistication of the shaders invoked when a ray hits a triangle. Most surfaces are not perfect reflectors or refractors; instead a shader invoked at an initial hit point may choose to shoot many secondary rays in a non-uniform distribution, thereby integrating the resulting returned values. Similarly, the first set of secondary rays can spawn its own second set of secondary rays, etc.
- Whether or not global illumination techniques are employed. Some techniques take a budget of rays and shoot them from the lights into the scene letting them bounce around, thereby caching the light intensities of the various hit points. A subsequent traditional ray tracing pass might then consider these cached hit points as additional light sources, for example.

® Pentium is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Ray tracing scales weakly with the complexity of the scene because of the acceleration structure. Due to the structure's hierarchical nature, ray tracing reduces the cost of finding a ray triangle intersection to  $O \log N$ . As the size of the model increases linearly, the overall size of the acceleration structure can grow linearly also, but the cost of finding an intersection doesn't.

Scalability studies for raster graphics and ray tracing graphics have been performed. In general, the cost of raster graphics processing is linear with the number of pixels to be processed. The cost of ray tracing scales linearly with the number of rays shot, so roughly one can claim that ray tracing performance scales linearly with viewport size.



**Figure 4: Frame complexity vs. performance of SW ray tracing vs. HW raster**

If the viewport size remains fixed, then the ray-tracing performance scales logarithmically with the complexity of the scene. This means that if you compare a HW raster engine and a SW ray-tracing engine using the same input for both engines, although the HW will initially beat the SW, the SW will eventually catch up with the HW. In fact we measured this and found that the intersection point is in the vicinity of the 1M triangle range, i.e., when the scene complexity exceeds 1M triangles, a SW ray-tracing solution will always outperform a HW raster solution.

Figure 4 is a log/linear chart. We created many versions of the same model at various resolutions from 10K to 10M triangles, and we fed them into an nVidia GeForce FX 4200 (lower blue curve) and an nVidia GeForce FX 5900

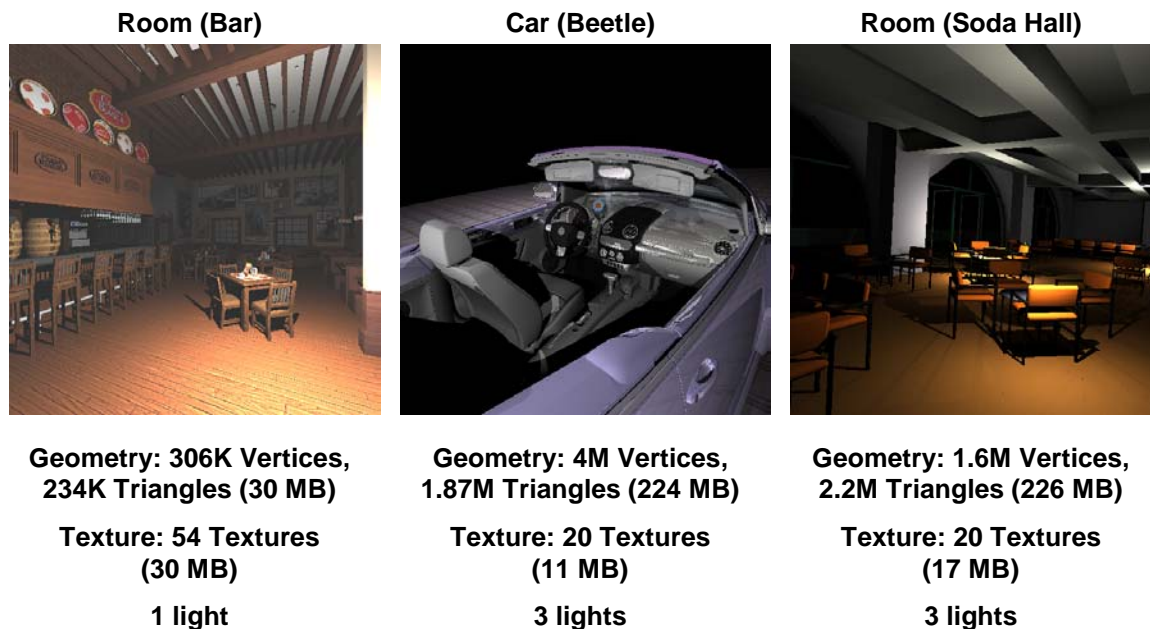
Ultra (upper blue curve). A SW ray tracer running on the same 3.2 GHz Pentium 4 processor (with a 512 KB L2 cache) was used to drive the HW cards (performance above the 10M triangle mark is extrapolated from curves fitted to the measured data). The curved blue lines show the performance of the HW cards: performance declines at  $1/X$ . The (almost) straight yellow/orange lines represent SW ray-tracing engine performance on a single processor and 2- and 4-way Symmetrical Multi-Processor (SMP) systems. Performance declines at  $1/\log(X)$ .

**Scene and Visibility Complexity Implications**

Figure 5 shows the three scenes used in this analysis of the ray-tracing algorithm. The scenes are carefully chosen to span a wide range of visual complexity. One can see that the level of detail as well as the quality of the images are at least as good as those produced by the high-end raster graphics engine today.

The first scene is a typical bar illuminated with one light. The model consists of about 250K triangles with about 300K vertices. The size of the acceleration structure that represents the spatial distribution of detail is 18 MB, and the scene data itself is 29 MB. The model has 54 different textures, and the texture maps to a total of 30 MB in size. Given the camera placement in this scene, the majority of the room's details are visible. There are no reflections or refractions in this scene. The second scene is a top view of a VW Beetle illuminated with three lights. The model consists of 1.87M triangles with about 4M vertices. The size of the acceleration structure is 127 MB, and the scene data itself is 224 MB. The image uses 20 texture maps totaling 11 MB in size. This particular view of the car was chosen because most of the geometric detail is in the interior of the car. Reflections are enabled with a maximum of two reflections per ray. The third scene is of the inside of a room in UC Berkeley's Soda Hall. The model consists of 2.2M triangles with about 1.6M vertices.

The size of the acceleration structure is 148 MB; the scene data is 226 MB. The model has 20 texture maps totaling 17 MB in size. This model was chosen because despite the fact that the whole model has 2.2M triangles, only a small percentage is visible at any time. If the camera is outside, we can see only the shell of the building; if it is inside the building we can only see the details of that particular area. In this particular room there are three lights. The full model has 1300, but the rest are disabled. Reflections and refractions are disabled.



**Figure 5: Scenes with different visual complexities used in this ray-tracing analysis**

Table 1 shows the raw FLOPs and bandwidth required per ray segment across the various models all rendered at the same 1024x1024 resolution. As you can see, there is some correlation between overall model complexity and required performance as we go from the bar scene (250K Triangles) to the beetle scene (~2M Triangles). Although complexity increases 10x, the computational and bandwidth costs only increase ~2x.

**Table 1: Computation and memory requirements**

Measured Data	Room (Bar)	Car (Beetle)	Building (Soda Hall)
Flop / RaySeg	1518	2954	1488
Byte / RaySeg	586	1382	793
Byte / Flop	0.386 / 2.59:1	0.467 / 2.14:1	0.533 / 1.87:1

**Table 2: Bandwidth at each level of memory hierarchy**

Measured Data	Room (Bar)	Car (Beetle)	Building
Core to L1 BW	1,300.0 GB/s	920.0 GB/s	1,260.0 GB/s
L1 to L2 BW	66.5 GB/s	57.6 GB/s	49.5 GB/s
External BW	6.1 GB/s	12.7 GB/s	1.1 GB/s
Raw / Ext BW	216:1	72:1	1141:1

However, as we go from the Beetle to the soda hall, we note that even though both models are about equally complex, because only a fraction of the scene is visible in any frame, the cost is dramatically lower for the soda hall. Table 2 shows the raw and cache filtered bandwidths required for the same scenes assuming each is rendered at 30fps. Here we see that the cache hit rate seems to correlate with the observed model complexity, so the soda hall scene shows the best performance thanks to the beam effect zeroing in on the portion of the acceleration structure that is effectively used by the ray tracing algorithm.

**Performance Scalability Studies**

Performance data was collected on 2-, 4- and 8-way SMP machines and for large cluster configurations with up to 128 nodes. As you can see from Figure 7, performance scales linearly with the number of processors for the SMP systems, and from Figure 8 you can see a similar story for clusters of systems with up to 128 nodes. We also measured performance with and without hyper-threading on the SMP systems. From Figure 6, you can see that we get a >25% overall performance improvement across the board when hyper threading is turned on.

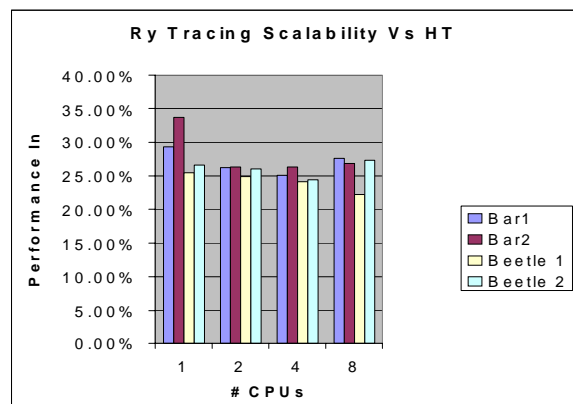


Figure 6: Hyper-threading effect on ray-tracing performance for 1-, 2-, 4- and 8-CPU systems

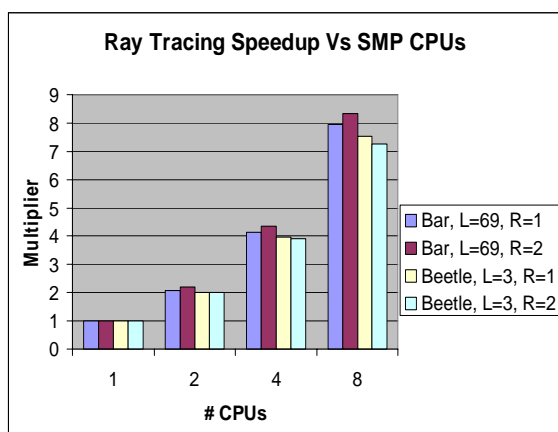


Figure 7: Scaling of ray-tracing performance for 2-, 4- and 8-way SMP machines (L = Lights, R = Reflections)

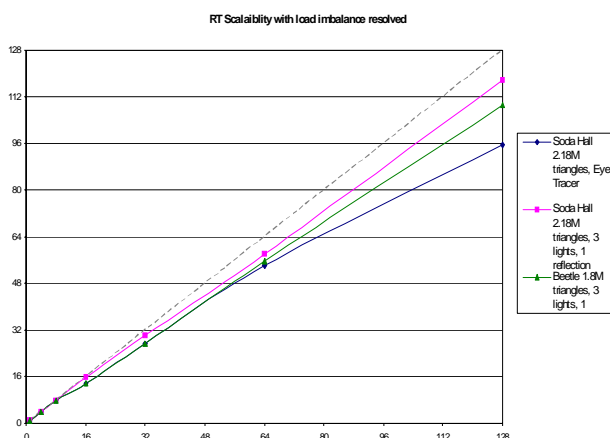


Figure 8: Scaling of ray-tracing performance for cluster systems with 1-128 CPUs

## CONCLUSION

Ray tracing has long been considered too expensive for mainstream rendering purposes. Movie production studios have only recently begun the transition to using it; however, the true cost of ray tracing has been very poorly understood until recently. It is now poised to replace raster graphics for mainstream rendering purposes. Its behavior is very well suited to CPU processors, and scales well with hyper threading and multi-processor configurations. The traditional cache hierarchy associated CPUs is very effective at managing the external memory bandwidth requirements. For ISVs, a transition to ray tracing is a huge step forward freeing them from all the limitations imposed on them by today’s raster-based approaches. Ray tracing is one tool that can enable ISVs to aspire to achieving high fidelity photo (or cartoon) realistic imagery.

## ACKNOWLEDGMENTS

Various people contributed to the materials that went into this report including Gordon Stoll, Alex Reshetov, Victor Lee, and Alexei Soupikov. If I left anyone out, please accept my apologies in advance.

## REFERENCE

[1] P.H. Christensen, D.M. Laur, J. Fong, W.L. Wooten, D. Batali, “Ray Differentials and Multiresolution Geometry Caching for Distribution Ray Tracing in Complex Scenes,” Eurographics 2003 conference, Computer Graphics Forum, Volume 22, Issue 3 (September 2003).

## AUTHOR’S BIOGRAPHY

Jim Hurley is a principal engineer in the Application Research Lab of the Corporate Technology Labs at Intel. He has been working in the area of computer graphics for over 20 years. His e-mail is jim.hurley at intel.com.

Copyright © Intel Corporation 2005. This publication was downloaded from <http://developer.intel.com/>.

Legal notices at <http://www.intel.com/sites/corporate/tradmarx.htm>.

**THIS PAGE INTENTIONALLY LEFT BLANK**

For further information visit:

[developer.intel.com/technology/itj/index.htm](http://developer.intel.com/technology/itj/index.htm)