6
Performance Results

In this chapter we present a performance evaluation of our proposed ray-tracing solution. Section 6.1 presents a synthetic analysis of GPU grid reconstruction performance, comparing to an equivalent CPU implementation. Following that, Section 6.2 evaluates our ray-tracing algorithm using static scenes, with no grid rebuild. In this case, we are also interested in measuring ray-tracing scalability and flexibility, in order to consider this technique for rendering CAD models.

Afterwards, in Section 6.3, we perform a full evaluation of the integrated ray-tracing system, using dynamic scenes to identify the possible benefits and limitations of our method. Finally, Section 6.4 presents a detailed comparison of our results with other state of the art research.

In all our tests, we have used a Core 2 Duo 3.0GHz CPU with an Nvidia 8800 Ultra graphics card. All scenes were rendered at 1024 x 1024 screen resolution.

6.1
Grid Construction

The Uniform Grid reconstruction procedure has been evaluated against a similar CPU implementation. The test scene consists of several triangles randomly distributed inside a box with dimensions [-50, -50, -50] x [50, 50, 50]. Each triangle has a randomly determined size, obtained by varying the radius of its enclosing bounding sphere from 0.2 to 1.0.

Grid resolution is determined using Equation 3-1. Therefore, increasing the number of triangles not only increases the amount of data to be stored, but also the number of cells in the resulting grid.

# Triangles	$5K$	$10K$	$50K$	$100K$	$200K$	$300K$	$400K$	$500K$
CPU	0.8	1.5	10.2	33.8	94.1	167.4	253.0	353.6
Our Method	1.8	2.3	7.1	14.7	31.3	75.2	121.2	178.6

Table 6.1: Time in milliseconds to rebuild the entire grid structure.

As can be seen in Table 6.1, the GPU solution suffers from the overhead of the graphics API for small scenes (10K triangles or less). However, this procedure scales better than its CPU counterpart, obtaining faster grid rebuild times for scenes with 50k triangles and more. Notice that table values do not scale linearly, as was expected since both the number of triangles and the number of cells increase with larger scenes.

6.2 Static Scenes

In order to measure ray-tracing performance independently of grid rebuild, we have devised a number of static scene tests. In these, the Uniform Grid is built only once during initialization. The first test measures performance of primary rays only, by using a simple grey-scale shader with no additional texture accesses. In the second test, we use additional material information obtained from several textures to perform lighting computations, and includes tracing shadow rays from a single point light. Finally, the third test case further enables reflection rays for the entire scene.

6.2.1 CAD Models

The first three scenes consist of different CAD models, as shown in Figure 6.1. The first model, called "Boat", is made of 50K triangles with no textures. The second model is a small oil platform called "MonoBR", made of 112K triangles, including a few textured materials. The third model is a complex section of the "P-40" oil platform, with more than 470K triangles. Table 6.2 summarizes ray-tracing performance for all test cases.

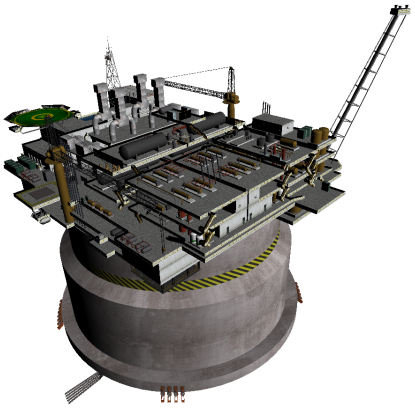
Scene	# Tris	Simple shading	+One Light	+Reflections
Boat	50K	21.3	11.7	4.1
MonoBR	112K	11.6	5.6	1.4
P40	470K	14.1	7.8	1.5

Table 6.2: Performance in frames per second (fps) for different static scenes.

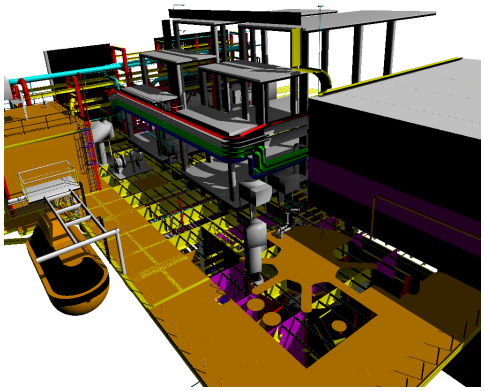
An evident result is the scalability of the ray-tracing procedure according to scene size. Even the "P-40" model with about half a million triangles can be rendered at almost the same performance as the smaller "Boat". In fact, using the point of view in Figure 6.1(c), it even outperforms the "MonoBR" scene.

This brings forth another advantage of the ray-tracing technique: automatic occlusion culling. In a CAD model, it is common for most of its

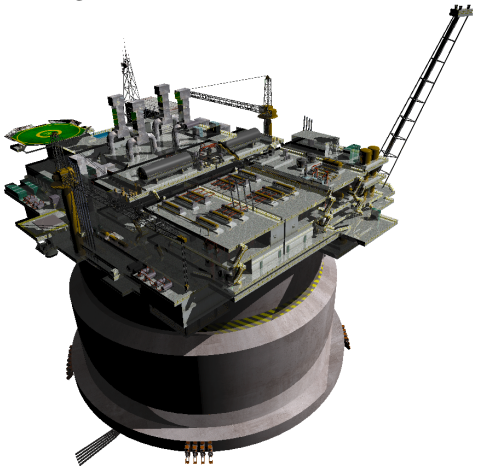
6.1(a): Boat with shadows and reflections.



6.1(b): MonoBR model with simple shading.



6.1(c): P-40 model with over 470K triangles.



6.1(d): MonoBR with shadows and reflections, notice added 3D perception and realism.

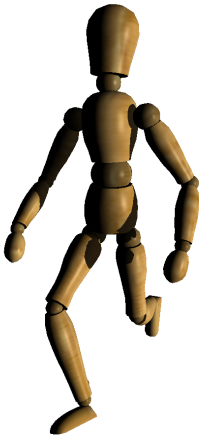
Figure 6.1: Ray tracing CAD models at interactive rates, including shadows and reflections.

complexity to lie inside the 3D structure. This means that several primitives are hidden away by the outside surfaces, which act as one large occluder. The ray-tracing algorithm automatically discards these hidden primitives, since it finds the nearest triangle intersection by performing successive tests in increasing depth order.

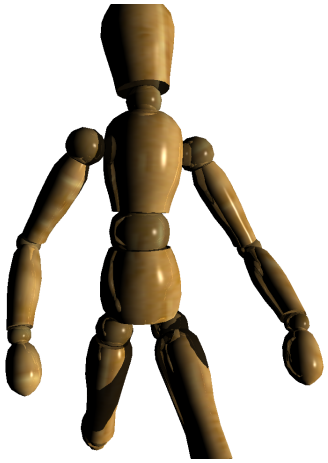
Usually, CAD models contain little or no material information. The ray-tracing algorithm is capable of adding realism to the scene, not only by providing illumination effects such as shadows but by enhancing material properties such as metal reflectiveness. These observations make ray-tracing a good candidate for visualizing this kind of models.

6.2.2 Benchmarks

The second batch of tests consist in a known benchmark for ray tracing dynamic scenes. All models contain detailed material information, including several high-resolution textures, as can be seen in Figures 6.2 and 6.3. In this case we seek to evaluate the pure ray-tracing performance, disregarding the acceleration structure rebuild. We use one animation key-frame from each model, building the grid structure only once. The results in Table 6.3 will help identify the bottleneck during the final rendering of the entire animation.



6.2(a): Wood-doll model with shadows.



6.2(b): Reflections in wood-doll makes it appear polished.

6.2(c): Hand model with shadows.

6.2(d): Ben rendered with shadows.

Figure 6.2: Deformable meshes used in static and dynamic scene tests.

The first model is a simple "Wood-doll", made of 5K triangles. The second one is a "Hand", modeled with 16K triangles. A runner character of 78K triangles is the model called "Ben". Another model consists of several "Marbles" that add up to almost 9K triangles. The fourth scene is made of five wind-up "Toys", totalling 11K triangles. Finally, the largest scene is a faerie model inside a "Forest", consisting in about 174K triangles.

Scene	# Tris	Simple shading	+One Light	+Reflections
Wood-doll	5K	71.5	54.6	14.6
Hand	16K	42.4	26.5	6.8
Ben	78K	19.6	16.5	4.4
Marbles	9K	108.9	88.1	16.5
Toys	11K	53.7	33.4	8.2
Forest	174K	6.3	3.4	0.7

Table 6.3: Performance in frames per second (fps) for a single key-frame of the benchmark scenes.

As shown in Table 6.3, the two simplest scenes, "Wood-doll" and "Marbles", can be rendered with shading and shadows at speeds above 50 fps. With the other less simple models, "Hand" and "Toys", our implementation can still achieve rendering rates of around 30 frames per second. The more complex "Ben" model can be rendered at about the same speed as the "Boat" model, similar in size, evaluated in the previous section.

The "Forest" scene is a classic worst-case scenario for the Uniform Grid: a complex object (the faerie) at the center of a larger but simple scene (the background). This is commonly known as the *teapot in a stadium* problem.

In these cases, grid resolution should be high to avoid cells with a large number of triangles from the small complex object. However, it cannot be set too high otherwise it would hamper ray-traversal performance through large regions of empty space. Also, memory consumption from a large grid can become a limiting factor.

Nevertheless, our implementation is capable of achieving interactive rendering rates, except when enabling reflections in the more complex models.

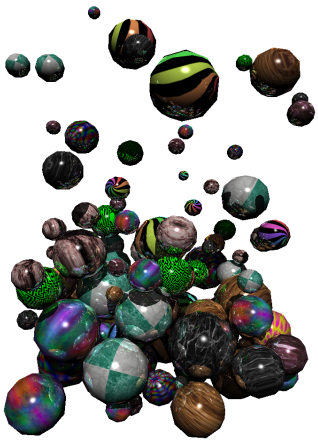
6.2.3 Discussion

The performance results from the CAD models and the benchmark scenes are consistent with the following observations:

1. Activating shading computations as well as shadow rays decreases rendering performance by at most a factor of two.

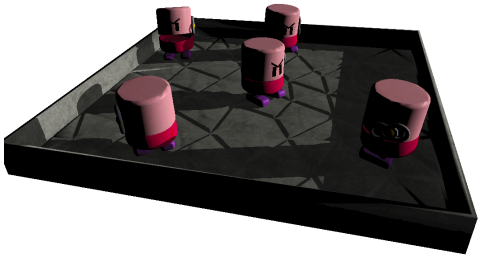
2. Tracing reflection rays further reduces these values by a factor of four.

The first observation is not only due to the cost of tracing additional shadow rays, which effectively doubles the number of rays being traced per frame, but also due to additional memory operations required for shading computations. Considering these two factors, performance is above what one

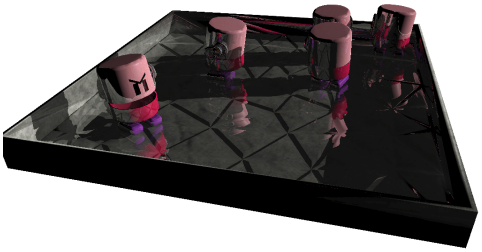


6.3(a): Several marbles rendered with shadows and reflections.

6.3(b): Faerie dancing in the Forest scene.



6.3(c): Toys scene with shadows.



6.3(d): Same scene using reflective materials.

Figure 6.3: Scenes with unstructured movement used for static and dynamic test cases.

would expect. We have further verified that using optimized traversal and intersection routines for shadow rays have significantly reduced their overall impact in rendering speed.

Furthermore, performance with one level of reflection is interactive for scenes with less than 100K triangles. A simple explanation is that each additional reflection ray performs the entire shading computations once more, while also spawning additional shadow rays. In effect, this test has twice the number of rays being traced per frame and also twice the number of shading procedures being performed.

6.3 Dynamic Scenes

The main goal of this work is to ray-trace dynamic scenes, including illumination effects, at interactive rendering rates. We have evaluated the

same six benchmarks, from Section 6.2.2, but this time we render their entire animation sequences. The "Wood-doll", "Hand" and "Ben" models consist of deformable meshes that do not move along the scene. Meanwhile, the "Marbles", "Toys" and "Forest" scenes combine mesh deformation with unstructured movements, comprising real-world test scenarios.

Scene	# Tris	Simple shading	+One Light	+Reflections
Wood-doll	5K	68.6	52.6	13.1
Hand	16K	41.2	25.2	6.4
Ben	78K	17.5	13.3	3.1
Marbles	9K	103.2	84.3	15.3
Toys	11K	52.7	28.9	7.8
Forest	174K	3.1	2.8	0.4

Table 6.4: Performance in frames per second (fps) for the entire animation of each benchmark scene.

When rendering the entire animation, it is necessary to fully rebuild the grid structure each new key-frame. Comparing the values in Table 6.4 with Table 6.3, it is clear that this reconstruction procedure has little to no impact in rendering performance. The frame-rate with full animations is up to 10% smaller than when rendering a single key-frame. Only the "Forest" scene has suffered a greater slowdown, from 30% to 50% depending on the test case.

6.4 Comparison with Related Work

In this section, we seek to evaluate our work in relation to other state of the art research. We have chosen the test case with fully animated scenes including shading, textures and shadows. Table 6.5 includes performance figures from our technique, as well as other four related work.

The first work is a CPU implementation that uses a Uniform Grid to trace packets of rays [Wald et al. 2006]. The second uses a deformable bounding volume hierarchy to adapt to scene movement [Wald et al. 2007]. Another proposal uses a multi-core CPU to build a kd-tree structure in parallel [Shevtsov et al. 2006]. Finally, the fourth related research is an equivalent solution to ours, but one that uses a kd-tree as acceleration structure inside the GPU [Zhou et al. 2008].

Values in Table 6.5 demonstrate the efficiency of our grid rebuild and ray-tracing algorithms. Related to the work in [Wald et al. 2006, Wald et al. 2007], we are capable of achieving up to four times faster rendering speeds. Only when compared against a more optimized structure, as the kd-tree used

Scene	Our Method	CPU Grid	CPU BVH	CPU kd-tree	GPU kd-tree
Wood-doll	52.6	35.1	*n/a*	*n/a*	*n/a*
Hand	25.2	15.9	*n/a*	*n/a*	*n/a*
Ben	13.3	8.9	8.5	*n/a*	*n/a*
Marbles	84.3	19.6	16.2	*n/a*	*n/a*
Toys	28.9	9.4	10.5	23.5	32.0
Forest	2.8	1.3	2.3	5.8	6.4

Table 6.5: Frames-per-second comparison between our method and two state of the art research, using complex shading and shadow rays.

in [Shevtsov et al. 2006, Zhou et al. 2008], our method achieves similar or slightly lower frame rates.

In the more sparse "Toys" scene, we are able of surpassing the work in [Shevtsov et al. 2006], but still achieve inferior performance than results in [Zhou et al. 2008]. In the "Forest" scene, this situation is aggravated: our implementation achieves about half the performance values than these two related research. This once more indicates a limitation in the Uniform Grid traversal (as discussed in Subsection 6.2.2).

To try and identify this possible bottleneck in our implementation, we have split the total frame time of our solution into: key-frame upload time, grid rebuild time and ray-tracing time. Table 6.6 below summarizes our findings with the complete shading algorithm and shadow rays. For comparison purposes, we have included the structure rebuild times from both [Wald et al. 2006] and [Zhou et al. 2008].

Scene	Key-frame Upload	Grid Rebuild	Ray-Trace	CPU Grid Rebuild	GPU kd-tree Rebuild
Wood-doll	1.1	2.3	15.5	1.0	*n/a*
Hand	2.7	4.1	36.2	5.0	*n/a*
Ben	12.4	10.4	81.1	14.0	*n/a*
Marbles	1.6	1.8	8.6	2.0	*n/a*
Toys	1.8	3.0	34.9	4.0	12.0
Forest	27.1	54.8	295.4	68.0	77.0

Table 6.6: Analysis of times in milliseconds from our proposed implementation, compared to related work.

From these results, we can conclude that uploading new key-frame data to the GPU is not the current bottleneck. Even though in the "Forest" scene a time of 27 ms starts to detriment overall performance, it is still not the most time consuming step: the ray-tracing procedure is the major factor to slowing down rendering rates.

Comparing our grid rebuild times with the ones from [Wald et al. 2006], we find that our method is considerably faster except for the very simple "Wood-doll" scene. This can be easily explained as the overhead of the GPU implementation, which has already become evident in Section 6.1.

As expected, the kd-tree rebuild times from [Zhou et al. 2008] are higher than our Uniform Grid implementation. Since our overall performance is worse in the "Toys" and "Forest" scenes, we can conclude that our ray-tracing procedure is taking most of the rendering times. Indeed, Table 6.6 show that for all the other test scenes our current bottleneck is the ray-tracing step, which can take up to ten times longer than the grid structure rebuild.