Using Graphics Hardware to Speed-up Visibility Queries

Nicolas Holzschuch, Laurent Alonso

Using graphics hardware to speed-up your visibility queries

Laurent Alonso
and
Nicolas Holzschuch
Équipe ISA, INRIA-Lorraine – LORIA

LORIA, Campus Scientifique, BP 239, 54506 Vandœuvre-lès-Nancy CEDEX, France

We present a visibility method that takes advantage of the graphics hardware to give fast answers to visibility queries. Our visibility method is designed to solve two types of visibility queries: point-based visibility queries, where several visibility queries share the same origin, and plane-based visibility queries, where several visibility queries have their origins on the same plane. Both occur frequently in global illumination algorithms.

Combining the speed given by graphics hardware with a software heuristic to avoid reliability problems, our visibility method is significantly faster than ray-casting, and still gives the same results.

1. INTRODUCTION

Visibility queries take up a significant part of total computation time in global illumination simulation algorithms, whether radiosity or ray-tracing. In hierarchical radiosity, up to 70 % of total computation time is devoted to visibility queries [Holzschuch et al. 1994]. It is very important to have a fast and reliable visibility method to do global illumination simulations on a complex scene. Modern graphics hardware provide fast access to the data computed by the graphics hardware, making it possible to use this data for visibility computations.

The results returned by the hardware are, by nature, of a limited precision. We need heuristics to predict where this limited precision can lead to false results and we must resort to different visibility methods at these points.

In this paper, we present a visibility method that takes advantage of the existing graphics hardware, is reliable and faster than ray-casting. Our method comes in two variations:

—a method for point-based visibility queries. In point-based visibility queries, we answer several visibility queries sharing the same origin in space. This occurs, for example, in ray-tracing for rays leaving the eye or a point light source.

—a method for surface-based visibility queries. In surface-based visibility queries, we answer several visibility queries where all the rays originate on the same

LORIA is UMR 7503 LORIA, a joint research laboratory between CNRS, Institut National Polytechnique de Lorraine, INRIA, Université Henri Poincaré and Université Nancy 2.

Name	Polygons	Description and Origin
Living-Room	288	A simple scene
Classroom	3138	One of Peter Shirley's scenes
Floor	4578	The unfurnished sixth floor of the Soda Hall model
Greek Temple	7778	A circular Greek temple from the Marmaria, Delphes
Office	9110	A furnished room from the Soda Hall model

Table 1. Characteristics of our sample scenes

surface. This occurs, for example, in hierarchical radiosity, when we are shooting radiosity from a polygon.

In order to demonstrate our algorithm, we used five sample scenes. They are depicted in figures 1 and 2 and their main characteristics are shown in table 1. The sample scenes are of various complexity, and behave differently in terms of visibility query. We will present the results by representing the sample scenes by their number of polygons. Polygon count is not the only parameter for visibility queries but it is useful for graphical representation.

2. PREVIOUS WORK

Most global illumination algorithms use ray-casting for their visibility queries, and many research was devoted to ray-casting acceleration, through hierarchy, spatial subdivision and surface caching.

Using the Z-buffer directly, the first implementations of the radiosity algorithm [Cohen and Greenberg 1985] did both visibility and form-factor computations in a single step. This method gives many sampling and aliasing artifacts [Wallace et al. 1989]. To avoid these problems, [Pietrek 1993] developed a form-factor computing algorithm using a Z-buffer combined with a heuristic to detect places where the Z-buffer would give faulty answers, and a ray-casting algorithm in those places.

Since the introduction of hierarchical radiosity [Hanrahan et al. 1991] and wavelet radiosity [Gortler et al. 1993], the visibility step is separated from form-factor computation. Most implementations use a Gaussian quadrature to compute the form-factor, combined with a ray-casting method to compute visibility.

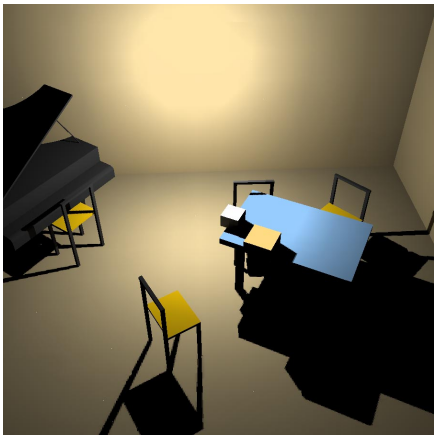
Our method is a fully separate visibility method, that takes advantage of the hardware, and can be easily combined with any illumination simulation method, such as light ray-tracing, eye ray-tracing, hierarchical radiosity or wavelet radiosity.

3. VISIBILITY METHOD FOR POINT-BASED VISIBILITY QUERIES

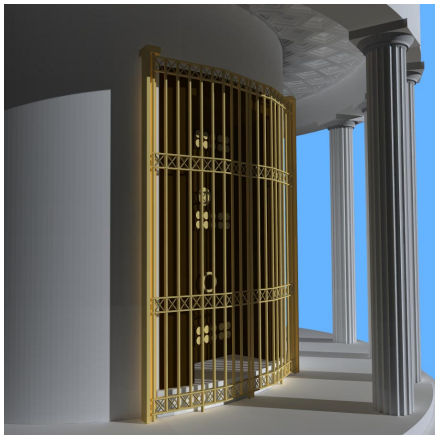
This section presents our visibility method for point-based visibility queries. Point-based visibility queries occur when we consider several visibility queries with the same starting point and varying ending points. They generally occur when we consider the interaction between a single point and several other objects, such as a point light source illuminating a scene, or the rays leaving the eye in eye ray-tracing.

The simplest hardware-based visibility method in this case is the object ID image. We make a rendering of the scene, using Z-buffer for hidden surface removal. It can be either a single object ID image (for eye ray-tracing) or a cube of object ID images (for a point light source).

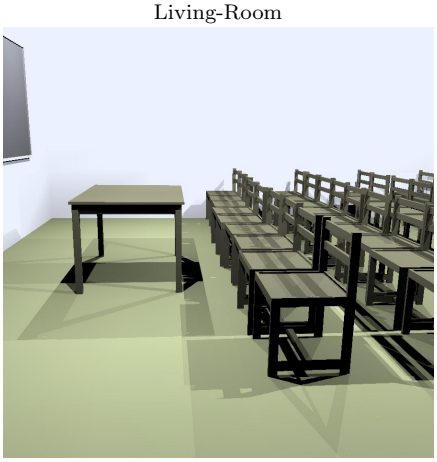
The rendering of the scene is done using a specific colour for each element in

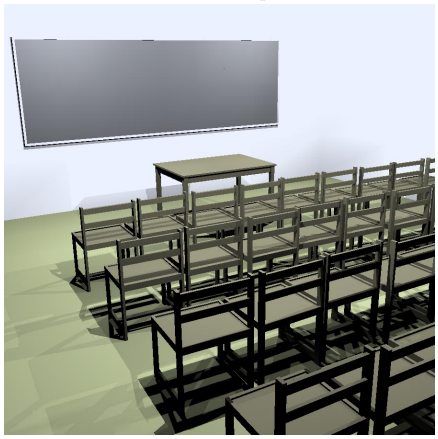


Living-Room

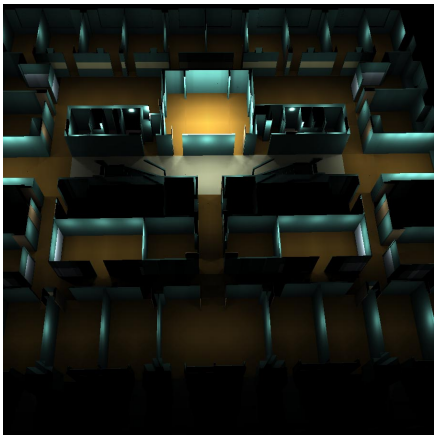


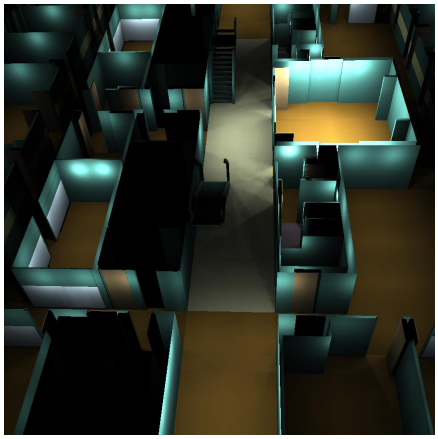
Greek Temple





Classroom





Floor: one unfurnished floor of the Soda Hall model

Fig. 1. Views of our sample scenes


Office: one furnished room of the Soda Hall model

Fig. 2. Views of our sample scenes (continued)

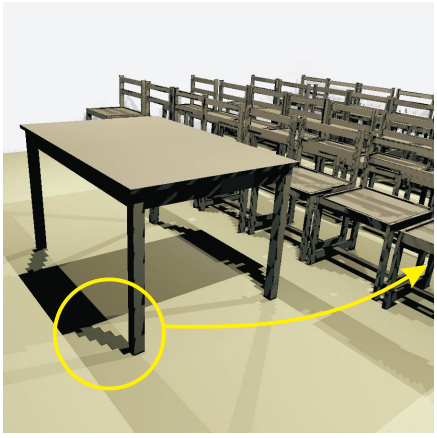
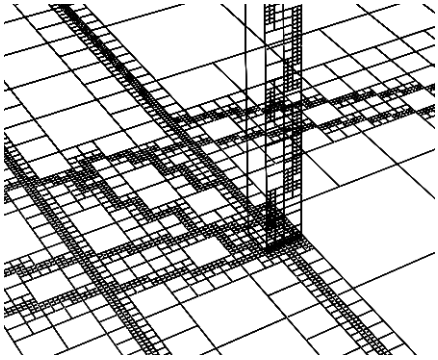



Fig. 3. Visibility with object ID images: result of false answers

the scene. We keep one colour for the background[1]. The first surface in a given direction is the surface corresponding to the colour of the pixel intersected in this direction.

Using ID images is both easy to implement and extremely fast. It is at least an order of magnitude faster than ray-casting. It is also very unreliable. We found that it gives an average of 5 % of false answers. Worse, these false answers result in large artifacts in the simulation and in unnecessary subdivisions (see figure 3).

These false answers are inherent to ID image sampling. We are sampling the scene using a regular grid (the image). The visibility queries are at different points than the original samples, which results in aliasing errors. These errors become important when the frequencies sampled are higher than the Nyquist limit, for example at a sharp edge between objects, or with an object small relative to the

[1]On 24-bit colour displays, this limits us to $2^{24} - 1 = 16,777,215$ different elements.

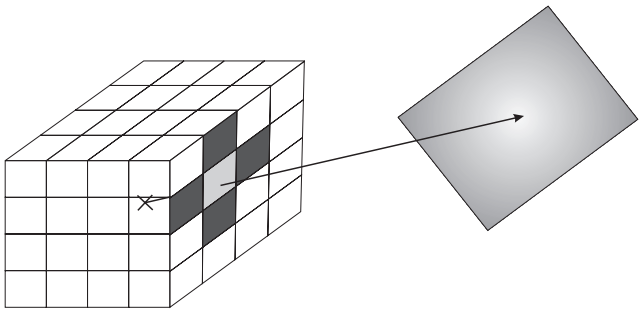


Fig. 4. Our visibility method: querying neighbouring pixels.

sample spacing. To reduce the number of aliasing errors, the only solution is to increase the sampling frequency, by increasing the size of the image, but graphics hardwares can only render images below a certain size[2].

3.1 Our point-based visibility method

Our visibility method uses an ID image for most of the visibility queries (or a cube of ID images) but uses an heuristic to detect places where the ID images will give unreliable answers. In these places, we resort to a more reliable visibility method, such as ray-casting.

Instead of querying just the pixel corresponding to the direction of the visibility query, we also query the four neighbouring pixels (see figure 4)[3]

—For all these pixels:
  —We retrieve the surface corresponding to the colour of that pixel.
  —We compute the intersection between this surface and the ray[4].
  —If there is an intersection, we compute the distance between the origin and this intersection.
  —We keep the surface with the smallest distance.
—If we have found at least one intersection, we return the surface with the smallest distance.
—If all the pixels did correspond to the same surface (or to the background) we conclude that the ID images give consistent information. If there was no intersection, we conclude that there is no surface intersected in that direction.
—Otherwise, we conclude that we cannot trust the answer returned by the ID images. We need a more reliable visibility method.
  —Usually, we just use ray-casting.
  —However, if we detect that a significant proportion of the previous visibility queries could not be resolved using the ID images, we enhance the resolution of the images.

[2]With current Silicon Graphics workstations, the maximum size is an image approximately a thousand pixels wide.
[3]With a cube, some of those neighbouring pixels can be different faces. To err on the side of caution, if the first pixel is on the border of a face, we also query the nearest pixel on the neighbouring face of the cube, as well as its neighbours.
[4]Each surface is tested only once, even if it appears in several pixels.

Enhancing the resolution of the images is done by zooming in on the problem area: we create a new image, centred on the problem area, with the same size than the previous ID images but twice the resolution. We then query this image as we would query the original images. The "zoom-in" image stays in use as long as the visibility queries fall inside it. As soon as a visibility query falls outside the new image, the "zoom-in" image is discarded and deleted from memory.

If the first zoom is not sufficient, and subsequent visibility queries still have consistency problems with the "zoom-in" image, we can zoom in on the problem again, and again. In that case, we build a stack of ID images, and we always use the ID image at the top of the stack. As soon as a visibility query falls outside this image, we discard it and fall back to the previous ID image on the stack, until we are back to the original cube.

If we have to use ray-casting, we advance the origin of the ray using the potential occluders returned by the ID images. For each of them, we had to compute the intersection between the ray and their supporting plane. From these intersections, we take the one that was closest to the origin of the ray, and use it as a new starting point for the ray. This scheme greatly reduces the length of space travelled by the ray, and hence the computation time. This is just a heuristic; it can happen that we advance the origin of the ray too far, thus missing an occluder. The heuristic works fine because most of the time, the occluders are grouped together; seldom do we have a single occluder floating alone in the scene. To be safe, we don't advance the origin to the nearest intersection, but only part of the distance – usually 90 %.

The ideas behind our visibility method are that:

—the ID images will give us rapid answers in places where there are no problems,

—we use ray-casting for small problems that concern few visibility queries,

—we use a better ID image for larger problems that concern several visibility queries. We assume that successive visibility queries correspond to points that are spatially close, and we exploit this spatial coherency.

3.2 Parameters and Sensibility to the Parameters

Our algorithm has two main parameters: the size of the ID images, and the proportion of previous visibility queries with problems that defines the threshold for zooming in.

Our experiences show that, on a Silicon Graphics RealityEngine II, we get our best results with a medium sized image (500×500 pixels) and a high threshold for zooming (0.8, that is 80 % of queries). It must be pointed out that the algorithm has a strong tolerance to these parameters: almost any setting (image size between 200 and 800, zooming threshold between 0.3 and 0.8) will give results close to the optimum.

The worse results are with a large image (800×800 pixels) and a low threshold for zooming (below 0.3). In that case, we zoom often, and each zoom is quite costly. The optimum set of parameters probably depends on the capacities of the graphics hardware and the speed of a single visibility query using ray-casting.

In our implementation, we only store what happened (whether we had to cast a ray or not) over a certain number of the previous visibility queries since the last zoom. The proportion of past visibility queries with problems is computed on

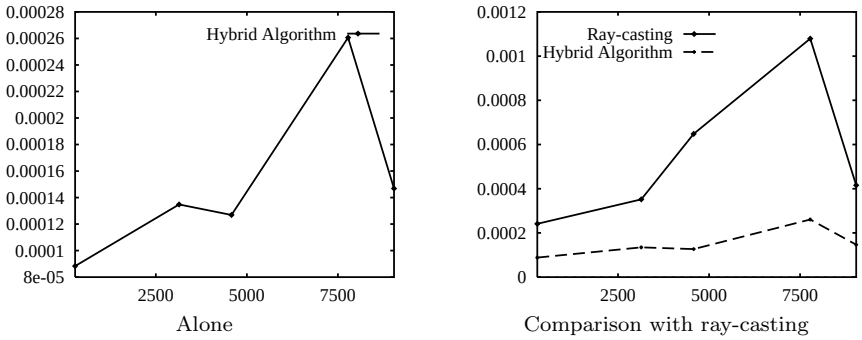


Fig. 5. Average time (in seconds) for a single visibility query for our algorithm, as a function of the number of polygons in the scene.

the visibility queries stored. We also only check whether we should zoom in once we have filled the storage capacity, to avoid early zooms due to a small number of visibility queries failing. The size of the storage is a minor parameter. Our experiences show that it has little influence on the timing results. We usually store 20 to 30 visibility queries.

3.3 Sample Data and Comparison

We have demonstrated our algorithm in a light ray-tracing application: for each sample scene, we set up a point light source sending light into the scene. Our sampling was divided in two steps:

—In the first step, we compute the illumination from the light source using ray-casting for visibility computations, and record all the queries and their results.

—In the second step, we conduct the same visibility queries, all together, for both visibility algorithms (ray-casting and our visibility method), check the results and record the time.

Conducting all the visibility tests together has two advantages:

—we can record the time for visibility tests without interference from other parts of the program. Due to their extreme shortness, visibility queries do not lend themselves well to profiling.

—we can record the time and reliability of visibility queries without having them interfere on the rest of the program. Since several decisions in an illumination simulation program depend — at least in part — on the result of visibility queries, using different visibility algorithms can lead to different computations being done, with different results in the end. As a consequence, the visibility queries called during the computations will not be the same, which has an influence on the time they require.

Figure 5 shows the average time for a single visibility query for our algorithm, both alone and in comparison with ray-casting.

The best thing with our visibility method is its reliability. On all our sample scenes, we had the same answer with our algorithm and with ray-casting for more than 99.9 % of the visibility queries.

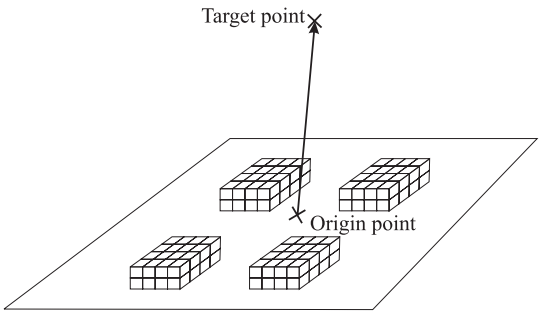


Fig. 6. We use four hemicubes surrounding the origin point.

4. VISIBILITY METHOD FOR PLANE BASED VISIBILITY QUERIES

The previous algorithm works with visibility queries sharing the same origin. Building ID images is a time-consuming task (even using graphics hardware) so each image must be shared by many visibility queries in order for a hardware-based visibility algorithm to be more efficient than straight ray-casting.

A specific case is several visibility queries having different spatial origins, but with all these origins located on the same surface. This happens frequently in radiosity, when a polygon is shooting light into the scene.

4.1 Description of the Algorithm

We start — as with the previous algorithm — by allocating a single colour to each object in the scene, and one colour to the background. We also define sampling points, located on a regular grid on the plane. These sampling points will be used as centres for hemicubes.

For each visibility query:

—We locate the sampling points that surround the origin of the visibility query. Usually four of them (but possibly two if the origin is close to the edge of the surface, or even just one).

—For each sampling point, we build a hemicube centred on the point (unless this hemicube has been built for previous visibility queries) (see figure 6).

—We query each hemicube for the pixel that corresponds to the direction of the target point, plus a few pixels along the epipolar line. Easiest is to take a 3×3 square, which always includes the epipolar line (see figure 7). If some of those pixels fall on another face of the cube, we query them as well.

—For each pixel:
  —We retrieve the surface corresponding to the colour of that pixel.
  —We compute the intersection between this surface and the ray.
  —If there is an intersection, we compute the distance between the origin and this intersection.
  —We keep the surface with the smallest distance.
  —Each surface is tested only once, even if it appears in several pixels.

—If we have found at least one intersection, we return the surface kept (the one with the smallest distance).

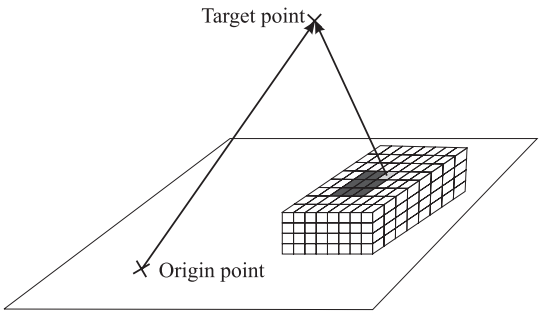


Fig. 7. We query the pixels in the direction of the origin point.

—If there were less than four hemicubes surrounding the origin point, and we haven't yet reached a decision, we resort to another visibility algorithm, such as ray-casting.

—If, for all the hemicubes, all the pixels did correspond to the background, we conclude that there is no surface intersected in that direction.

—If there were some occluders detected by the hemicubes, but none of them actually blocked the visibility between the origin point and the target, we compare the occluders lists that were returned by the hemicubes.
  —If the differences between them were small enough, we conclude that the information returned by the hemicubes was consistent, and that it must be visible.
  —If there were large discrepancies between the lists of occluders returned by the hemicubes, we conclude that the information returned was not consistent enough, and we resort to another visibility algorithm, such as ray-casting.

—If we have to use ray-casting, we advance the origin of the ray to the nearest potential occluder, in the same manner as described in section 3.1.

The idea behind our visibility method is that the hemicubes return a potential list of blockers, like the *tube* structure in [Teller and Hanrahan 1993]. However, this list of blockers is computed on the fly instead of being stored in a data structure. This is consistent with new wavelet radiosity implementations that avoid storing data related to the interaction [Stamminger et al. 1998]. Since the blocker list returned by the hemicubes can be incomplete, we supplement it with ray-casting.

4.2 Parameters and sensibility to the parameters

Our algorithm has three main parameters to consider:

—The size of the grid used for the sampling points.

—The size of the ID images in the hemicubes.

—The number of differences in the blocker lists for which we decide to cast a ray.

The key in choosing the first and second parameters is to have a reasonable accuracy, so as to avoid being forced to cast rays too often, and yet not have too many hemicubes built, so that the entire set of hemicubes can reside in memory. We found that relatively small hemicubes (with an image size between 50 and 100 pixels) are sufficient.

The number of sampling points is dependent on the geometry of the scene, and should be guided by the size of the features in the scene. The larger these features are, the further apart the hemicubes can be.

We found that, as a rule of thumb, for a simple scene (such as a room) a spacing corresponding to 15 to 25 hemicubes on the largest dimension of the scene was sufficient (that is to say, the distance between sampling points is the largest dimension of the room divided by 15 to 25). On more complex scenes, such as a floor of a building, the same heuristic expands to 15 to 25 hemicubes on the largest dimension of an average room. This heuristic obviously fails on scenes with both very large and very small features, such as the classical teapot in a stadium.

Too many hemicubes result in the visibility being slowed down as most of the time is spent computing hemicubes that will not be used, not to mention the extra memory costs. Too little hemicubes result in the visibility being slowed down as most of the time is spent sending rays in the scene.

Once again, we found that there is not a huge dependency in these parameters for our algorithm. That is to say, a normal variation in these parameters will usually result in small variations in the computation time for visibility queries. The largest threat is running out of memory, which makes the machine swap, completely ruining your computation time. The best idea to avoid swapping is to use smaller hemicubes (50 pixels wide).

Our current implementation keeps all the hemicubes computed in main memory until we move onto the next plane. A possible improvement would be to cache only a limited number of hemicubes. We would then delete the oldest hemicubes as newer hemicubes are being computed.

The third parameter is a little more complicated. The four hemicubes are seeing the scene with different points of view. Therefore, it makes sense to allow some differences between what they see. On the other hand, if the hemicubes return four completely different lists of occluders, it is a good indication that they are not seeing the complete set of occluders. We found that a total of eight differences (an average of two differences per hemicube) was giving correct results. Smaller values will mean more rays being casted, resulting in more precise results, at the expense of speed. Higher values will mean trusting the hemicubes more often, hence faster results, but some of them might be wrong.

4.3 Sample data and Comparison

We have demonstrated our algorithm in a hierarchical radiosity algorithm. We used the same protocol than for point-based visibility queries (see section 3.3).

Figure 8 shows the average time for a single visibility query for our algorithm, compared with ray-casting. Our method is approximately twice faster than ray-casting.

For one of our sample scenes, our method is much slower, being only marginally faster than ray-casting: the Greek temple is modelled mostly with long, thin parallel rectangles. These rectangles are not well suited for the regular sampling of an ID image. We get better results if we cluster these rectangles together, and apply our method to the clusters.

For the unfurnished floor of the Soda Hall, our method is 4 times faster than ray-casting. This scene is modelled with large rectangles having an aspect ratio

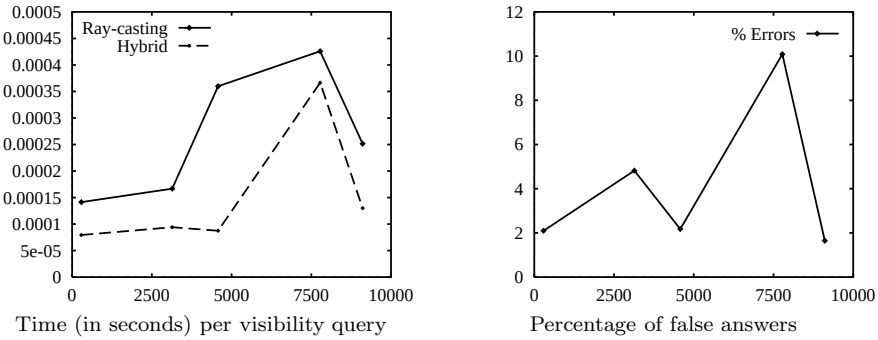


Fig. 8. Results for our method for plane-based visibility queries, as a function of the number of polygons in the scene.

close to 1. Such rectangles are well suited for the regular sampling in an ID image.

Figure 8 also shows the percentage of false answers returned by our method. While the number of false answers is not negligible, we must point out that these errors do not introduce significant differences in the radiosity solution. Each radiosity interaction requires several visibility tests, and combines their results together. Thus, false answers to visibility queries are generally cancelling each other out.

5. DISCUSSION

Our visibility method requires graphics hardwares with Z-buffer capabilities and 24-bit colours. We need hardware-based off-screen rendering capabilities, and fast access to the result of off-screen rendering. Such capabilities can be found in current graphics hardware, and are becoming more and more commonplace.

Our visibility method does not suppress aliasing problems related to the limited resolution of the sampling. It merely pushes back the point where they appear. However, we are pushing them quite far back. First, if the closest surface is on one the pixels queried, then this closest surface will be found and returned by our algorithm. For our algorithm to fail, we must have the closest surface that does not appear on *any* of the pixels queried. There can be two possible causes for that:

—the surface does not appear due to precision problems in the Z-buffer,
—the surface does not appear due to conflicts with other surfaces.

The first problem should be seen as something deeply inherent to our visibility method (and to any hardware-based visibility method). The second problem can be partially addressed using clustering. Some objects, such as spheres, cylinders, statues... are modelled with many small surfaces. Each of these surfaces is likely to be missed by the ID images, due to the limited resolution. However, if we render the entire object as a single object, with a single colour, we increase the probability that it will be detected. On the other hand, it will take longer to check whether there actually is an intersection.

Our algorithm for plane-based visibility queries can still be improved. The strongest point in our point-based visibility algorithm is the recursive zooming, which increases the resolution of the sampling in problematic areas. Zooming is more difficult in plane based queries, because there are several images to consider.

Finding a way to combine zooming with plane-based queries could greatly improve the algorithm. Also, we could use Image-Based Rendering techniques to interpolate between the ID images. It should even be easier since we have access both to the images and to the depth fields.

6. ACKNOWLEDGEMENTS

Permission to use the Soda Hall model[5] was kindly granted by Prof. Carlo Sequin.

REFERENCES

Cohen, M. and Greenberg, D. P. 1985. The Hemi-Cube: A Radiosity Solution for Complex Environments. *Computer Graphics (ACM SIGGRAPH '85 Proceedings) 19*, 3 (August), 31–40.

Gortler, S. J., Schroder, P., Cohen, M. F., and Hanrahan, P. 1993. Wavelet Radiosity. In *Computer Graphics Proceedings, Annual Conference Series, 1993 (ACM SIGGRAPH '93 Proceedings)* (1993), pp. 221–230.

Hanrahan, P., Salzman, D., and Aupperle, L. 1991. A Rapid Hierarchical Radiosity Algorithm. *Computer Graphics (ACM SIGGRAPH '91 Proceedings) 25*, 4 (July), 197–206.

Holzschuch, N., Sillion, F., and Drettakis, G. 1994. An Efficient Progressive Refinement Strategy for Hierarchical Radiosity. In *Fifth Eurographics Workshop on Rendering* (Darmstadt, Germany, June 1994), pp. 343–357.

Pietrek, G. 1993. Fast Calculation of Accurate Formfactors. In *Fourth Eurographics Workshop on Rendering*, Number Series EG 93 RW (Paris, France, June 1993), pp. 201–220.

Stamminger, M., Schirmacher, H., Slusallek, P., and Seidel, H.-P. 1998. Getting rid of links in hierarchical radiosity. *Computer Graphics Journal (Proc. Eurographics '98) 17*, 3 (September), C165–C174.

Teller, S. and Hanrahan, P. 1993. Global Visibility Algorithms for Illumination Computations. In *Computer Graphics Proceedings, Annual Conference Series, 1993 (ACM SIGGRAPH '93 Proceedings)* (1993), pp. 239–246.

Wallace, J. R., Elmquist, K. A., and Haines, E. A. 1989. A Ray Tracing Algorithm for Progressive Radiosity. *Computer Graphics (ACM SIGGRAPH '89 Proceedings) 23*, 3 (July), 315–324.

[5]The Soda Hall model is available on the web, at `http://www.cs.berkeley.edu/~kofler`.