Sampling Visibility in Three-Space

Martin Blais   Pierre Poulin

Département d'Informatique et de Recherche Opérationnelle
Université de Montréal [*]

Abstract

We present a visibility algorithm that is based on sampling the scene for visible surfaces. The subset of lines for which the visibility is pre-calculated is defined by a two-plane parameterization. Each sample represents the portion of geometry that is visible for the lines through that sample's region. Lumping geometry together in grid cells allows achieving visibility at a coarser level, and improves the solution. A rendering algorithm that uses this visibility pre-calculation is presented. Using this method, most hidden geometry is culled and a speedup in rendering time is obtained.

Some problems remain, however, and are discussed here. Particularly, some cells are left unrendered because they are missed by the discrete sampling procedure, thus potentially creating holes in surfaces. Techniques to improve the solution are proposed. In particular, modifying the pre- and post-filtering algorithms used in creating/resampling the field helps remove some of the artifacts. Nonetheless, the algorithm remains useful for cases where only coarse visibility is needed.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling - surfaces and object representations.

Keywords: Visibility, Occlusion, Culling, Image-Based Rendering

1 Introduction

Determining visibility has been an important topic in computer image synthesis since the advent of raster graphics two decades ago. The need for faster computation of the visible parts of 3D geometric representations of objects has led to research in several directions, each of which has had its own successes and problems. The method we present here combines results from the emergent field of image-based rendering and visibility algorithms to produce a new method for pre-computing visibility for a scene.

We modify the light field rendering technique of Hanrahan and Levoy [9] to be used to identify visible surfaces. Radiance values in the field are replaced by indexes to chunks of geometry. We call this structure the "visibility field". We use the fact that the visibility of surfaces does not change along a line in free space, just as for radiance. The visibility field is computed offline, and is resampled at render time to find out which surfaces are visible from the camera viewpoint. These surfaces are then treated by a classical rendering and visibility algorithm. Thus most of the invisible surfaces do not have to be considered for rendering. The reduced amount of scene geometry to be rendered implies possibly faster renderings.

2 Related Work

2.1 Visibility Culling

The visibility determination problem has been one of the primary areas of research in computer graphics since the early 1970's. Many simple hidden-surface removal algorithms have been proposed and are widely in use. However, as we are dealing with increasingly large model databases, more complex algorithms are becoming necessary to reduce the cost of visibility computation in the context of rendering.

Many techniques implement visibility culling from image-space structures (the z-buffer is one such example). One such recent method [14] uses hierarchical occlusion maps created from a subsets of objects to cull away most of the geometry. That technique is similar to the technique presented here, in the sense that it is based on sampling visibility "images". Another similarity is that this technique provides the possiblity of performing *approximate visibility culling*, which is what our method performs.

Recently, characterizations on the exact content of the visibility function have been developed [4]. The visibility function contains many discontinuities. Detecting the important discontinuities in this function is very important for radiosity solutions. A method for pre-computing global visibility for a scene has been proposed by Durand *et al.* [5]. Their method computes the *exact* visibility from one point to all other points in space. Although their structure may be computed lazily, it is not suited for fast or real-time rendering.

The approach we suggest here relies on sampling the visibility function rather than computing all the possible visibility changes.

2.2 Image-based Rendering

Image-based rendering systems create new views of a 3D environment from a set of pre-acquired images—that is, no information other than the images and their geometry is known about the scene being rendered. These new rendering methods offer the advantage that model complexity is unbounded, and rendering time is independent of scene complexity.

The idea of capturing all the incoming light at a point in space in an environment map [7] [1] is a forerunner to the more recent approaches of image-based rendering. It was used to efficiently compute reflections on shiny surfaces. It was shown that although spherical projections of the scene would be most appropriate, that it is more efficient to use a cube (six planes) to store the environment map. Panoramic images (cylindrical

[*]Département d'Informatique et de Recherche Opérationnelle, Université de Montréal, C.P. 6128, Succ. Centre-Ville, Montréal, Québec, Canada, H3C 3J7
Email:{blais|poulin}@iro.umontreal.ca

[2], or spherical [13]) have also been used to store environment maps to enable one to explore a scene from a fixed viewpoint. A shortcoming of these methods is that the view or shading point is fixed.

To address this problem, many techniques using the forward mapping approach have been developed [3] [11]. These methods often rely on a disparity map to project the original images' pixels on the viewplane. These methods suffer from the problem of *holes*. Hidden regions in the original images become visible when the viewpoint is changed and no information about those regions is known. Several interpolation techniques have been devised to try to fill in those gaps.

A slightly different approach, that relies on a more complete sampling of the light field, has been proposed recently [9] [6]. Dubbed *light field rendering*, or *Lumigraph*, these methods are not constrained to using only a few selected images. Instead, a parameterization of a subset of lines in space is established and discretized to contain radiance values for many positions/directions. Rendering is achieved by querying and interpolating the radiance values stored in the parameterization from the camera viewpoint. Appropriate filtering, both when creating the field and when resampling from it, is crucial for the success of this method. Since a large amount of samples is needed to produce decent images, a good compression technique is required to limit the field storage size.

A related rendering technique (*Lucifer*) has also been proposed by Lewis *et al.* [10] for propagating light between regions in a scene. The technique iteratively propagates and interacts the light field in/with scene voxels. They propose using a wavelet representation of the light field for each of these voxels.

In the *light field rendering* approach, it was shown how to effectively sample and filter the four-dimensional light field in space. In particular, two-plane parameterizations [8] are used for efficiently accessing sample values of the light field. The intersection of a line in space with the two fixed parameterization planes (uv and st planes) is determined, yielding (u, v, s, t) values which directly index into the field array. This method is both simpler and more computationally efficient than using Plücker coordinates or position/angle representations because it only involves linear operations.

2.3 View-based Rendering

Another technique in-between image-based rendering and model-based rendering has been proposed by Pulli *et al.* [12]. This technique first precomputes visibility at a *fixed* number of viewpoints around an object by keeping texture-mapped partial submeshes of the model for each of these viewpoints. To render a view from a vantage viewpoint, the three closest views are chosen and the three corresponding submeshes are rendered from the new camera viewpoint.

The three renderings are then combined according to a weighting scheme ("soft z-buffering"), where the following parameters are taken into account: the position of the camera relative to each sample viewpoint, a measure of the surface sampling density, and feathering at the meshes' edges. Most of these operations can be done in hardware, so it is quite efficient.

However, there is a lot of redundancy in the information that is kept (the submeshes), if the meshes overlap.

3 Sampling Visibility

The method presented here consists of using a two-plane parameterization to sample the visibility of scene geometry. Each scene object is divided in basic blocks of geometric primitives. We sample the scene by raytracing all lines subtended by the two planes, storing which primitives are hit from any one ray, thus creating a 4D structure similar to the light field structure used in light field rendering [9]. We render a view by first resampling from the *visibility field* to mark the visible cells, then we use a standard rendering method to render only those cells that are marked.

Equivalence between the light field and the visibility field is discussed in section 3.1 and 3.2, and filtering issues are discussed in sections 3.3 and 4.1. Section 4 describes the rendering algorithm in detail, and section 5 presents our implementation. The results and a discussion are presented in sections 6 and 7.

3.1 Field Complexity

It has been shown that the plenoptic function reduces to a 4D light field in space free of occluders [9]. The plenoptic function is a 5D function in space (radiance values for all x, y, z, θ, ϕ). However, we can simplify it to a 4D function by assuming that we are representing the function in space free of occluders (this, in conjunction with the fact that radiance is constant along a line).

The same principle holds for visibility: in space free of occluders, the part of an object's surface that is visible along a line is constant. The visibility field can thus be represented as a 4-dimensional field.

3.2 Visibility along a Ray

We need to define how visibility along a ray will be represented. We assume that the scene is composed of polygon meshes. We chose to represent it as the "geometry element" that is first encountered by a ray. This "geometry element" could be defined as either

1. The object (polygon) that is directly hit by the ray ;
2. A grid cell that contains the polygon that is directly hit by the ray ;
3. The set of objects hit by a cone in the direction of the ray.

Each of these choices has its advantages and associated problems. The first method, which consists of associating a polygon id with the sample, will minimize the amount of multiply-referenced geometry, because few rays will hit the same polygon (see figure 1). We want to avoid multiply-referenced geometry elements, because the more we have similar adjacent geometry elements in the visibility field, the more time will be wasted in the resampling algorithm, marking the same geometry elements for rendering which, in such a case, is an indicator that visibility sampling could have been achieved with a lower sampling resolution. However, since the individual geometry elements (polygons) can be quite small for detailed models, the sampling rate has to be quite high in order to cover the whole visible geometry (see section 7.3).

The second option, lumping together parts of the visible geometry in grid cells, alleviates the problem of missing geometry because of insufficient sampling by creating larger geometry elements. However, a problem with this approach is that

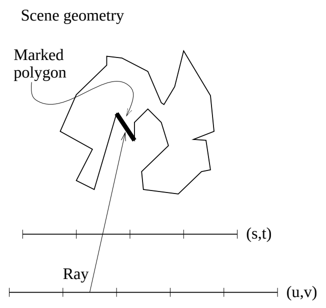

Figure 1: Sampling polygon ids

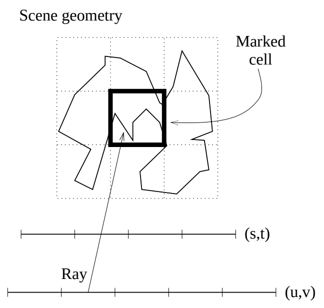

Figure 2: Sampling object grid cells

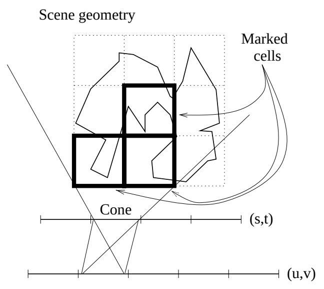

Figure 3: Sampling the cone of cells

the size of the grid cells depends on the position of the objects in the scene, i.e. after projection, they may be very small.

The third option, keeping ids to all the visible objects or grid cells enclosed in a cone originating at the ray, resolves the problems of the preceding two options at the expense of keeping multiple object ids for a single sample of the visibility field. A problem with this method is that it requires a much more complex data structure: the samples may be of varying size. It is thus hard to bound the size of this structure, since it depends directly on local scene complexity.

There is a tradeoff between the quantity of geometry that is referenced in the parameterization and the quality of the visibility estimation. If geometry is referenced in large lumps, then we may be rendering more geometry than is actually necessary. On the other hand, if the geometry is finely subdivided, then the visibility sampling might miss some important geometry elements.

In our implementation, we have chosen the use a compromise between the second and third technique. We divide each object in grid cells and each sample in the visibility field contains the id to this object and cell. Furthermore, each sample may contain more than one cell/object id, but only up to a *fixed* number. We allocate the field by setting the maximum number of ids for each sample, so that the structure remains a simple 4D array in memory.

Note that we chose to subdivide each object separately, rather than subdividing the scene as a whole. Subdividing the whole scene, regardless of objects, would also be a working solution (see section 7).

3.3 Pre-filtering

In the context of light field rendering, pre-filtering the input is necessary to reflect all of the possible values that a particular radiance sample represents. This is also true for the visibility field, and it corresponds to keeping all of the geometry elements visible from within a cone passing through the sampling grid elements in the (u, v) and (s, t) planes (see figure 3).

The visibility field is created by casting rays in the scene through all of the possible u, v, s, t values on the parameterization planes. At the time of creating the visibility field, the maximum number of cell ids n, that the field will contain is fixed. For a sample, we supersample the scene by using a grid of 3 × 3 positions on both the uv and st planes, thus yielding 81 different rays. Since a limited number of cells are available for the sample, we rank the cell ids by *sorting them* according to the number of times they occur among the 81 rays. We then store the n first cell ids in the sample. If there are less than n cells hit, we store the remaining cell ids for this sample as null ids (empty).

4 Rendering Algorithm

We render a novel view of the scene by resampling the visibility field by using rays from the new camera viewpoint, marking those cells that are encountered as visible. Then we render only the marked cells, with a traditional rendering method (in our case, z-buffering with SGI hardware). The procedure is depicted in figure 4.

```
procedure Render (grid cells) is ;
begin
    Project the $(u, v)$ and $(s, t)$ slab planes and rasterize intersection
    for each pixel in intersection loop
        Resample the visibility field to find which cells are visible
        Mark the grid cells associated with the sample
    end loop
    for each cell in the grids loop
        if the cell is marked then
            Render the cell's geometry (with OpenGL)
        end if
    end loop
end Render
```

Figure 4: The rendering algorithm

In comparison with the usual straightforward rendering method—that is, rendering all of the geometry in the scene—our rendering technique has the additional cost of rasterizing and resampling the grid. There is a gain in rendering speed if

$$K_{\text{rasterize+resample}} + t_{\text{render visible cells}} < t_{\text{render all cells}}$$

where $K_{\text{rasterize+resample}}$ is constant time for a fixed slab resolution and subsampling level for resampling. Also, we will compare with *visibility tracing*—that is, applying the ray casting technique to find visible cells at rendering time (see section 6).

The extent of this gain will depend on the following factors:

- **The amount of *total cell occlusion* in the scene:** if there are a lot of completely hidden cells, we avoid rendering them. The more occlusion there is between grid cells, the more we gain, because the technique essentially culls the hidden grid cells ;

- **The amount of *partial cell occlusion* in the scene:** if a lot of partially visible cells are present, then we end up sending the hidden geometry in them to the rendering pipeline as well. This can happen, for example, through holes in the scene, or near silhouette edges ;

- **The distribution of geometry in the scene:** if there is a lot of self-occlusion *within* the cells, we end up rendering the hidden geometry in those cells too ;

- **The number of geometric primitives (polygon count):** the cost of rasterizing and resampling the slabs is constant and is thus amortized over the total rendering cost. If we have a lot of primitives to render, the cost of this resampling procedure is negligible over the gain in culled geometry ;

- **The resolution of grid subdivisions:** if we use finely divided grids, it is more costly to go through all of them to check for marked nodes for rendering.

4.1 Post-filtering

In light field rendering, post-filtering is applied when resampling from the field: the neighboring samples may be linearly interpolated to avoid the aliasing effect induced by the discretization of the slab planes.

Similarly, we can use the neighboring samples in the visibility field to improve our visibility approximation. We optionally consider the 16 neighboring samples of the (u, v, s, t) ray intersection when resampling, and mark the associated cells as visible. In doing so, we often mark cells that were "missed" by the point-sampling procedure as visible. However, we probably also increase the amount of hidden geometry that is sent to the rendering pipeline.

5 Implementation

The software was programmed in C++, and running times were computed on an SGI R10000 with Solid Impact graphics and 128 MB of memory. Compression of the visibility field was not implemented. Although the method could be generalized to any kind of geometry, our implementation only supports polygonal geometry (triangle and quad meshes).

Each slab sample contains an array of one or more of the following structure (see figure 5): an id to the mesh that the visible grid cell is in (-1 for an empty cell), and the coordinates of the grid cell in that mesh.

```
typedef struct {
    unsigned char meshidx ; /* mesh index */
    unsigned char cx ; /* cell coordinates */
    unsigned char cy ;
    unsigned char cz ;
} VisFieldSample;
```

Figure 5: Structure of visibility field sample

The system is able to run at interactive rates and can support multiple polygonal meshes (each with its own grid of cells), and multiple visibility fields ("slabs").

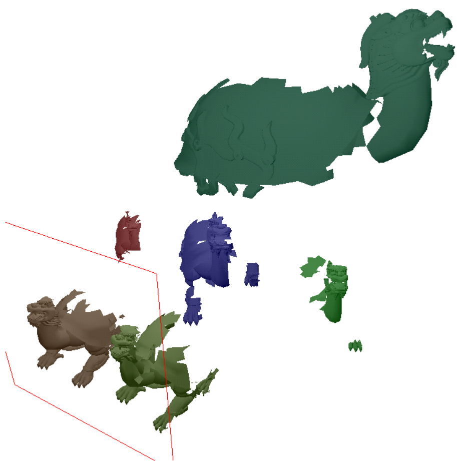

Figure 6: Side view of non-culled cells as computed from a viewpoint in front of a scene of 6 turtles

6 Results

We tested our algorithm on a few scenes containing moderate occlusion. A normal rendering of the *turtle* test scene is shown in figure 12 (all geometry rendered).

An image produced by rendering only the cells marked by our algorithm, using no interpolation, and only one cell id per sample, is shown in figure 13. The uv plane is very large and placed at infinity, and the st plane close to the object. Both planes are discretized with a 16×16 resolution. Resampling is done by subsampling the viewpoint by using 1 out of 4 pixels only (half-size image).

Notice the holes in this figure: they correspond to missing cells. We observe that marking neighboring samples' cells ("interpolation", see section 4.1) removes some of these holes, but not completely (see figure 14). Also, using more than one cell id per sample greatly improves the visibility estimation (see figure 15, which uses 5 cell ids per sample).

In figure 6, the scene is shown from a different viewpoint, with only the geometry that would be rendered from a viewpoint in front of it, to clearly show that occluded cells of the model are not being rendered.

Results on other test scenes (*cars* and *bunny*) are available online at our web page associated with this paper [1].

Statistics for our algorithm are shown in figures 8 and 9, 10, 11. Notice that since the trivial *bunny* scene contains only one convex object, only about half of the geometry is culled. Much less cells are rendered for scenes with even moderate occlusion.

We compare our method with two different techniques for visibility calculations:

- *Z-buffer rendering*, by sending all of the geometry to the rendering pipeline (without using display lists) ;

- *Visibility tracing*, which consists of casting rays from the camera viewpoint *at render time*, to determine which cells are visible, and then rendering those with a typical z-buffer algorithm.

[1]URL: http://www.iro.umontreal.ca/labs/infographie/papers/

Comparing our algorithm with visibility tracing allows assessing the performance of the slab parameterization. If the set of lines from which the visibility field is computed is too sparse or badly chosen for most viewpoints, the visibility tracing algorithm with subsampling will give better results (less holes) in lesser time. However, considering the extra cost of performing the tracing, a good parameterization will deliver better results faster than the visibility tracing algorithm.

It is difficult to directly compare our results with the visibility tracing times obtained, but for most examples, the visibility tracing technique gave roughly comparable results in slightly more time (a few seconds) when subsampling at around 1 out of 10^2 samples (with a few holes remaining). Using a resampling procedure without subsampling was not practical (it took several minutes to compute one image).

The results are analyzed with regards to the following criteria:

- Quantity of geometry rendered (number of grid cells). See figures 8, 10 and 11. For the *turtle* scene, these show that typically less than 800 of the total 4680 cells which contain geometry are rendered with our algorithm (less than 20%). Furthermore, as figure 8 shows, there is not a substantial amount of degradation as we subsample less pixels (especially if we use more than one cell id per sample). Note that these results assume that most cells contain approximately the same amount of geometry. We could be more precise and count the actual triangles/quads that are rendered ;

- Rendering time with and without visibility culling (*without* display lists). Rendering the grid cells cannot be achieved with display lists (unless one would have a display list per cell, which is not practical—the number of display lists supported by the hardware is limited). Figures 9 and 10 show the rendering times obtained with and without the culling algorithm. For the *turtle* scene, we observe a speedup when we subsample 1 out of 9 pixels (compare with no subsampling, which mostly does not achieve speedup), even with uv and st interpolation. As pointed before, subsampling does not increase the amount of error (missed cells) substantially, but decreases the resampling time $K_{\text{rasterize+resample}}$.

The display list mecanism offers great speedup for the naive rendering strategy, but its speedup is linear with the number of primitives sent to the pipeline. For this reason, we can expect that using our algorithm, with the reduced number of geometric primitives to render, there is a number of geometric primitives at which we still gain over that rendering strategy, even with display lists. Furthermore, typical, more complex scenes will usually exhibit more occlusion, hence further reducing our algorithm's relative rendering time when more primitives are present in the scene.

The results were recorded over an animation of the camera. For all renderings, we have kept the camera within the region where the geometry is not "clipped" by the parameterization, so as to not bias the results with partially rendering the geometry because of the lack of slab samples for a particular viewpoint (we can only render geometric primitives through the (u, v) and (s, t) slab planes intersection). This is not really a limitation of the technique, because we could use multiple slabs to cover visibility from all viewpoints around the scene.

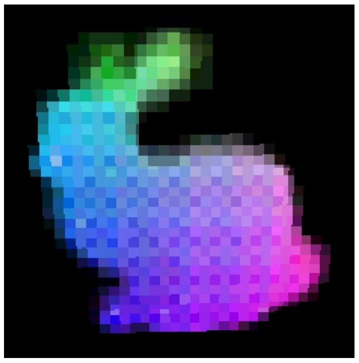

Figure 7: Equivalent resolution light field rendering with no interpolation.

6.1 Resolution

Observe that we used some rather low resolutions here, compared to the resolutions typically needed for light field rendering. The resolutions one needs to use depend on the grid resolution, the placement of the parameterization slabs, the placement of objects with regards to the slabs, and the output image resampling resolution (subsampling). To get an idea of the light field that would be captured with such small resolutions, see figure 7. This figure exhibits a light field of a bunny with corresponding resolution of $8 \times 8(u, v), 32 \times 32(s, t)$, shown with no interpolation.

7 Discussion

In this section, we describe some of the problems inherent with this technique.

7.1 Memory Size

As for light field rendering, the visibility field structure occupies a large amount of memory and is costly to compute. Hence it is not practical to pre-compute the visibility field for multiple frames of an animation where some objects move: the technique is only useful for static scenes where only the camera moves.

However, we expect much more coherence in the visibility field than in the light field. Adjacent samples often have the same values because they index the very same cells. Compression techniques should thus be expected to offer a great reduction in memory size.

7.2 Partial Occlusion

A problem with sampling the visibility function in the domain of lines in 3-space is that since it contains many discontinuities, it is difficult to capture all the necessary directions/positions. For example, if there are some small holes in an object, through which another object is visible, if the holes are sufficiently small, and happen not to have a single sample ray cast through them, we will never render the surfaces behind it. Also, since we use a limited number of cell ids per sample, a ray that would hit the back object might not be considered important enough to be included in the sample.

7.3 Small Cells

Similarly to the partial occlusion problem, if some visible cells contain only a few small polygons, they might be left unrendered because none of the sample rays of the visibility field has hit them. This problem can potentially lead to holes in the model. Supersampling and using multiple cell ids helps reduce this effect, but does not completely solve the problem.

Moreover, when rendering from viewpoints far from the visibility field parameterization planes, less query rays from the camera hit the planes, and so only a subset of the cells may be rendered, thus leaving gaps in the image. This last problem is somewhat alleviated by the fact that the images produced are smaller (and in a sense, could be considered as a form of "buggy" level-of-detail approximation).

7.4 Positioning of Planes

The success of the technique also depends on how we place the parameterization planes in space. Different plane positions will cover different subsets of lines and might be more or less efficient for visibility computation for certain viewpoints. This has been further studied in other works [8] [9].

8 Conclusion and Future Work

We have presented a new method for pre-computing visibility for a scene from a set of locations in 3D. This method has shown to improve rendering speed by selecting only potentially visible surfaces to render (thus culling away large amounts of the geometry). The method is based on using the two-plane parameterization presented in the field of image-based rendering for sampling the 4D visibility function in space.

However, the technique exhibits some problems. First and foremost, it cannot compute exact visibility, that is, some cells may remain unrendered and cause holes to appear in the model, or not show through behind small holes in the model. It would be necessary to find a way to improve the method in order to have an exact solution. Depending on the scene geometry, the holes might be more or less important. Keeping into account all the cells in each sample cone requires a complex data structure, but it was easy to extend the structure to use up to a finite number of cell ids per sample.

One aspect that could greatly improve the solution obtained is the use of interframe coherence. For one frame, we could keep the last few frames' visible cells as marked.

Another improvement that could be brought to this technique is to pre-filter hierarchically the visibility field, similarly to a mipmap, but in four dimensions. Doing this could help avoid the problem of undersampling the field when the viewer is far from it (section 7.3). Since the visibility field is a 4D structure, mipmapping would cost only $1/16$ times the size of the full field, which is negligible compared to the benefits it would provide.

Also, we could try to improve the grid structure itself by using an octree, whose cell sizes would be based on the distance of the objects from the slabs. We could scale the cell sizes appropriately to minimize holes. However, access to the octree nodes is more slightly costly than for a grid (if we want to keep them in optimal space).

Compression of visibility fields has not been investigated. Many adjacent samples index to the same cells. Therefore, we can expect that a high compression ratio would be possible.

Finally, more analysis could be done to assess the usefulness of the method. It is hard to judge the efficiency of the algorithm in filling in holes and silhouettes from a ratio/count of cells, because this ratio highly depends on the amount of occlusion in the scene and the viewpoint. Rather, we could provide a metric based on image-space differences (pixel comparisons), to see how much the image rendered with our algorithm differs from a rendering of all the geometry.

9 Acknowledgements

We would like to thank the Stanford graphics lab for putting their light field package available on their web page, as well as some cool models. Code snippets for rasterization and resampling were taken from *lightpack*. Some of the models (*cars*) came from the Viewpoint Datalabs and Avalon archives. Thanks to Lifeng Wang for providing models from the Yuan Ming Yuan garden project (*turtle* scene).

Also, thanks to Alain Fournier for the discussions on comparing the results with visibility tracing. This research work was supported by FCAR.

References

[1] BLINN, J. F., AND NEWELL, M. E. Texture and reflection in computer generated images. *Communications of the ACM 19* (1976), 542–546.

[2] CHEN, S. E. Quicktime VR - an image-based approach to virtual environment navigation. In *SIGGRAPH 95 Conference Proceedings* (Aug. 1995), ACM SIGGRAPH, Addison Wesley, pp. 29–38. held in Los Angeles, California.

[3] CHEN, S. E., AND WILLIAMS, L. View interpolation for image synthesis. In *Computer Graphics (SIGGRAPH '93 Proceedings)* (Aug. 1993), vol. 27, pp. 279–288.

[4] DURAND, F., DRETTAKIS, G., AND PUECH, C. The 3D visibility complex: A new approach to the problems of accurate visibility. In *Eurographics Rendering Workshop 1996* (New York City, NY, June 1996), Eurographics, Springer Wien, pp. 245–256.

[5] DURAND, F., DRETTAKIS, G., AND PUECH, C. The visibility skeleton: A powerful and efficient multipurpose global visibility tool. In *SIGGRAPH 97 Conference Proceedings* (Aug. 1997), ACM SIGGRAPH, Addison Wesley, pp. 89–100. held in Los Angeles, California.

[6] GORTLER, S. J., GRZESZCZUK, R., SZELISKI, R., AND COHEN, M. F. The lumigraph. In *SIGGRAPH 96 Conference Proceedings* (Aug. 1996), ACM SIGGRAPH, Addison Wesley, pp. 43–54. held in New Orleans, Louisiana.

[7] GREENE, N. Environment mapping and other applications of world projections. *IEEE Computer Graphics and Applications 6*, 11 (Nov. 1986).

[8] GU, X., GORTLER, S. J., AND COHEN, M. F. Polyhedral geometry and the two-plane parameterization. In *Eurographics Rendering Workshop 1997* (New York

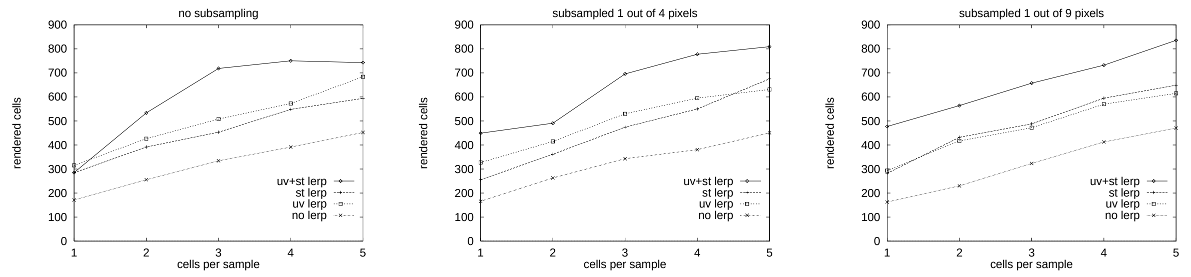

Figure 8: Nb. of rendered cells for *turtle* scene, $uv = 16 \times 16$, $st = 16 \times 16$, 231060 triangles, **4680 cells**

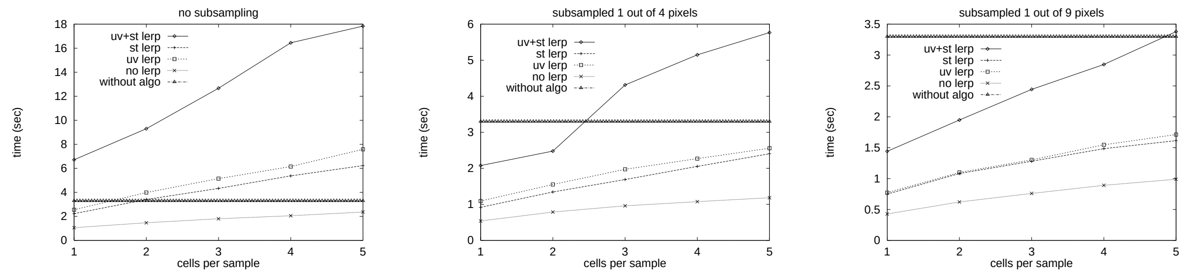

Figure 9: Rendering time for *turtle* scene, $uv = 16 \times 16$, $st = 16 \times 16$, 231060 triangles, 4680 cells, time to render all geometry: 3.3 secs.

City, NY, June 1997), Eurographics, Springer Wien, pp. 1–12.

[9] LEVOY, M., AND HANRAHAN, P. Light field rendering. In *SIGGRAPH 96 Conference Proceedings* (Aug. 1996), ACM SIGGRAPH, Addison Wesley, pp. 31–42. held in New Orleans, Louisiana.

[10] LEWIS, R. R., AND FOURNIER, A. Light-driven global illumination with a wavelet representation of light transport. In *Eurographics Rendering Workshop 1996* (New York City, NY, June 1996), Eurographics, Springer Wien, pp. 11–20.

[11] MCMILLAN, L., AND BISHOP, G. Plenoptic modeling: An image-based rendering system. In *SIGGRAPH 95 Conference Proceedings* (Aug. 1995), ACM SIGGRAPH, Addison Wesley, pp. 39–46. held in Los Angeles, California.

[12] PULLI, K., COHEN, M., DUCHAMP, T., HOPPE, H., SHAPIRO, L., AND STUETZLE, W. View-based rendering: Visualizing real objects from scanned range and color data. In *Eurographics Rendering Workshop 1997* (New York City, NY, June 1997), Eurographics, Springer Wien, pp. 23–34.

[13] SZELISKI, R., AND SHUM, H.-Y. Creating full view panoramic image mosaics and environment maps. In *SIGGRAPH 97 Conference Proceedings* (Aug. 1997), ACM SIGGRAPH, Addison Wesley, pp. 251–258. held in Los Angeles, California.

[14] ZHANG, H., MANOCHA, D., HUDSON, T., AND III, K. E. H. Visibility culling using hierarchical occlusion maps. In *SIGGRAPH 97 Conference Proceedings* (Aug. 1997), ACM SIGGRAPH, Addison Wesley, pp. 77–88. held in Los Angeles, California.

Interp.	Cells	% cells	Time	% time
none	231.4	17.0%	142 ms/fr.	31.1%
uv	283.8	20.8%	181 ms/fr.	39.6%
st	251.6	18.4%	194 ms/fr.	42.4%
uv+st	341.4	25.0%	385 ms/fr.	84.2%

Figure 10: Rendering statistics for *cars* scene, $uv = 12 \times 12$, $st = 48 \times 48$, 60064 triangles, 1364 cells, 1 cell id per sample, time to render all geometry: 457 ms/frame.

Interp.	Cells	% cells
none	91.2	38.8%
uv	104.9	44.6%
st	91.3	38.9%
uv+st	104.5	44.5%

Figure 11: Rendering statistics for *bunny* scene (no occlusion), $uv = 8 \times 8$, $st = 32 \times 32$, 3444672 triangles, 235 cells, 1 cell id per sample

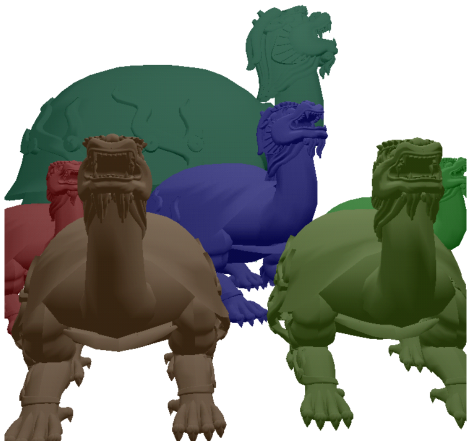
Figure 12: Normal mesh rendering

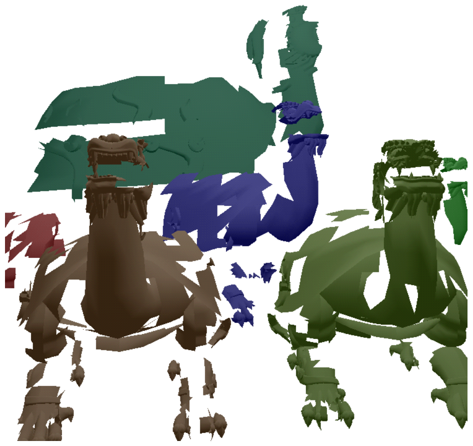
Figure 13: Grid cells rendering: No interpolation

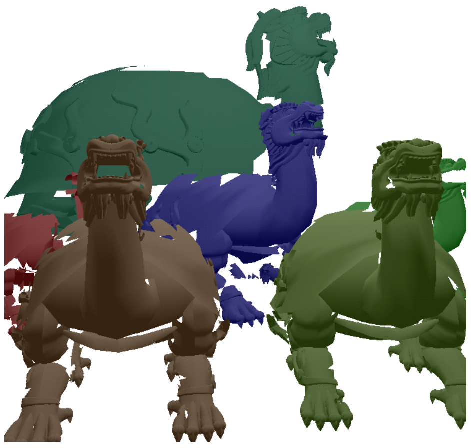
Figure 14: Grid cells rendering: UV+ST interpolation

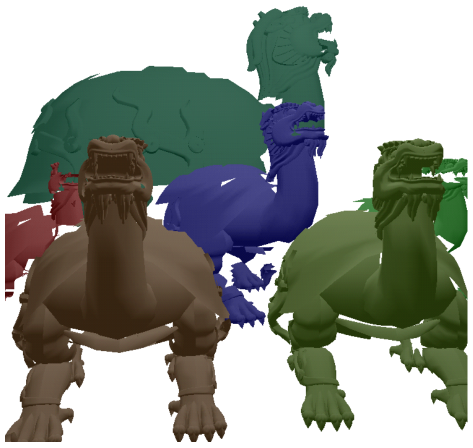
Figure 15: Grid cells rendering: using 5 cells per sample