Fast Walkthroughs with Image Caches and Ray Casting

(extended version)

Michael Wimmer[1] **Markus Giegl**[2] **Dieter Schmalstieg**[3]

Abstract

We present an output-sensitive rendering algorithm for accelerating walkthroughs of large, densely occluded virtual environments using a multi-stage Image Based Rendering Pipeline. In the first stage, objects within a certain distance are rendered using the traditional graphics pipeline, whereas the remaining scene is rendered by a pixel-based approach using an Image Cache, horizon estimation to avoid calculating sky pixels, and finally, ray casting. The time complexity of this approach does not depend on the total number of primitives in the scene. We have measured speedups of up to one order of magnitude.

CR Categories and Subject Descriptors: I.3.3 [Computer Graphics]: Picture/Image Generation - Display Algorithms, Viewing algorithms; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism - Virtual Reality.

Additional Keywords: virtual environment, ray casting, walkthrough, horizon tracing, real-time display, panoramic images, image-based rendering, visibility.

1 Introduction

In this paper, we present a new approach to the problem of interactively rendering large virtual environments, where the geometric complexity exceeds what modern graphics hardware can render interactively and the user has control over camera position and orientation. This is also commonly referred to as the 'walkthrough-scenario'.

Different solutions have been proposed for various types of virtual environments: Indoor scenes can be efficiently handled using portal rendering [Luebke95]. For sparsely populated outdoor scenes, the level of detail approach is viable, providing a number of representations for the same object with different rendering costs. Image Based Rendering has been proposed for very general, complex scenes. More recently, methods have been investigated to handle densely occluded, yet unrestricted scenes, for example urban environments. They are commonly referred to as 'Occlusion Culling'.

Especially in the case of densely occluded outdoor environments, the following basic observations can be made:

- A large part of the screen is covered by a small set of polygons that are very near to the observer (in the 'Near Field').
- Another large part of the screen is covered by sky.
- Pixels that do not fall into one of these two categories are usually covered by very small polygons, or even by more than one polygon.
- The number of polygons that fall outside a certain 'Area of Interest' is usually much larger than a polygonal renderer can handle - but they still contribute to the final image.

The main contribution of this paper is a new algorithm for accelerated rendering of such environments that exploits the observations listed above: the scene is partitioned into a 'Near Field' and a 'Far Field'. Following the ideas of Occlusion Culling, the Near Field is rendered using traditional graphics hardware, covering many pixels with polygons, whereas the Far Field is rendered using an alternative method: every remaining pixel undergoes a multi-stage Image Based Rendering Pipeline in which it is either culled early or sent to the last stage, a ray casting algorithm (see Figure 1).

Our method can be seen as a hybrid hardware / Image Based Rendering algorithm that uses a new way to obtain images on the fly with very low memory overhead. The algorithm is in its nature output-sensitive (compare [Sudar96]): by restricting hardware rendering to the Near Field, constant load of the hardware graphics pipeline can be achieved. The remaining pixels are also obtained in an output-sensitive manner: both the culling stage and the ray casting stage can be shown to have linear time complexity in the number of pixels only. Ray casting, if combined with an acceleration structure, is less than linear in the number of objects.

After reviewing previous work relevant to this paper, an overview of the system architecture is given

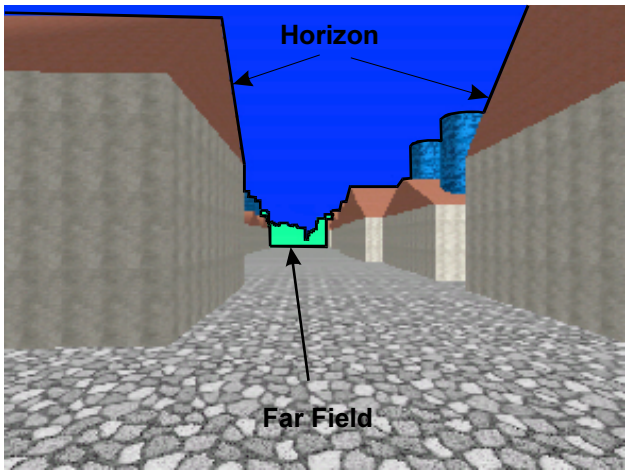

Figure 1: Ray casting is used to cover the pixels of the Far Field (beyond 100m). Pixels above the horizon are culled early

[1] Vienna University of Technology,
Email: wimmer@cg.tuwien.ac.at
Web: http://www.cg.tuwien.ac.at/~wimmer

[2] Ars Creat Game Development
Email: m.giegl@magnet.at

[3] Vienna University of Technology,
Email: schmalstieg@cg.tuwien.ac.at

and the various stages of and acceleration methods used in the algorithm are explained. This is followed by a discussion of the behavior of the system and the possible hardware setups where the algorithm works and a summary of the results we obtained in our implementation. Finally, conclusions and future avenues of research in this field are given.

2 Previous Work

Ray tracing is a well-known area of Computer Graphics. A good overview can be found in [Glass89]. The regular grid approach we use has been first introduced in [Fujim86]. Later, [Aman87] have introduced a different incremental traversal algorithm which proved to be the fastest for our purpose. On a theoretical analysis of the complexity of regular grids see [Cleary88]. Interesting extensions to grids for non-uniform scenes are discussed in [Klima97].

Image Based Rendering and Level of Detail rendering has been a hot research topic in recent time. One of the most well known commercial IBR systems is Quicktime VR [Chen95]. It is capable of displaying a panoramic view of an image acquired by a camera or by synthetic rendering. The viewer can rotate this panoramic image in realtime. Many papers have followed that improve on the basic idea. Different methods are used to interpolate between adjacent images to allow arbitrary viewpoints. Especially the problem of avoiding gaps and overlaps in the final image has received much attention in this area ([McMill95], [Shade98]).

[Heckb97] gives a good survey of current geometric Level of Detail algorithms. Hybrid traditional / Image Based Rendering algorithms have been proposed by [Shade96], [Schaufler96] and [Raff98]: they use graphics hardware to calculate Image Caches for objects on the fly. A similar idea, but taken further to allow affine transforms on the images done in special purpose hardware, is exploited by [Torb95].

[Teller91] describes a consistent way to precalculate 'Potentially Visible Sets' to accelerate architectural walkthroughs. [Luebke95] introduces portal rendering, where those PVS are calculated on the fly. More general scenes are made possible by the hierarchical z-buffer method [Greene93], where a hierarchical z-buffer is used to cull occluded geometry. An alternative occlusion culling method, Hierarchical Occlusion Maps, that does not use depth information, is shown in [Zhang97]. [Chamber96] already partitioned the scene into Near and Far Field. They use colored octree nodes to render the Far Field.

3 Overview of the System

The traditional polygonal rendering pipeline consists of three basic steps. Depending on the architecture, each of them may or may not be accelerated in hardware. What is obvious, though, is that the time complexity of rendering the scene is always linear in the number of primitives, because arbitrarily many objects may be

Figure 2: The traditional rendering pipeline consists of three steps

visible at any one time.

We introduce a second stage to the pipeline, which is not primitive-based, but purely image-based[4]. Each of the two stages is able to render the whole scene alone, but both would be equally overloaded by the whole database. Thus, rendering of the scene is distributed to the two stages by partitioning the scene into a Near- and Far Field. All primitives within a distance less than a certain threshold are sent to the polygonal renderer. The second stage passes over all the pixels and fills those that have not yet been covered by polygons in the Near Field

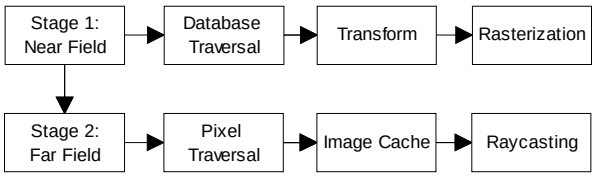

Figure 3: The extended rendering pipeline uses a polygonal pipeline for the Near Field and an image-based pipeline for the Far Field

with the appropriate color.

In the pixel stage, various mechanism exist to allow early exits before a ray is cast to obtain the color information:

- Pixels already covered by polygons are recognized by an opacity buffer that is created during the first stage.
- Pixels which fall into an area covered by sky are recognized by a Horizon Map created before the second stage.
- If a pixel fails those two tests, but its value is still within a valid error range, the pixel color is looked up in an Image Cache.

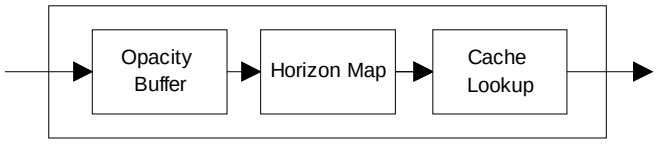

Figure 4: The Image Cache is composed of three stages

[4] unlike Impostors [Schaufler96], where the images used for Image Based Rendering still have to be created by polygonal rendering

Only if all these three tests fail, a pixel is sent to the final step in the pipeline, the raycaster. It is important to note that, while ray casting is a very costly technique in itself because it has to be done purely in software, the time complexity of the raycaster is still less than linear in the number of objects. Also, by restricting the number of polygons sent to the graphics hardware, the time spent in the polygonal graphics pipeline is bounded as well (given a reasonably uniform distribution of primitives in the scene). Thus, the overall algorithm can be said to be output-sensitive, i.e., its time complexity is linear in the number of *visible* primitives only, but less than linear in the *total* number of primitives.

4 Ray Casting for Image Based Rendering

4.1 Near Field / Far Field

When rendering a large scene, a huge number of objects can reside in the area defined by the viewing frustrum, but only a small amount of those objects actually contribute to the appearance of the image. Large, near objects usually have much more impact on appearance than small and far away objects.

It should therefore be possible to determine a set of objects that have the most contribution to the image, and only render this set of objects. The simplest way to achieve this is to select all objects within a maximum distance of the viewer. We call the space where these objects reside the 'Near Field'. This approach is very popular, especially in computer games, and it is often combined with fogging, so that the transition between the Near Field and the space where objects are simply not rendered is not so sudden.

Obviously, culling away all objects that do not belong to the Near Field introduces severe visual artifacts. We call 'Far Field' the space of objects beyond the Near Field, but not so far away as to be totally indiscernible. An important property of the Far Field is that it usually

- consists of much more polygons than the graphics hardware can render, but
- contributes to only very few pixels on the screen, because most of the pixels have already been covered by Near Field polygons.

To take advantage of this fact, a separate memory buffer, the 'opacity buffer' is used that records which pixels have already been covered by the Near Field for every frame. This is what has been demonstrated to work already in the 'Hierarchical Z-Buffer'-method, or for 'Hierarchical Occlusion Maps'.

The basic algorithm for our Image Based Rendering technique using ray casting is as follows:

1. Find Objects in the Near Field using the regular grid
2. Render those objects with graphics hardware
3. Rasterize them into the opacity buffer
4. Go through the opacity buffer and cast a ray for each uncovered pixel (enter the resulting color in a separate buffer)
5. Copy the pixels gained by ray casting to the framebuffer

4.2 Ray casting

We claim that ray casting is an appropriate technique for acquiring images for Image Based Rendering on the fly. This might seem strange at first glance, because ray casting (ray tracing) is known to be a notoriously slow technique. The reason for that is the high complexity: in a naive approach every object has to be intersected with a ray for every pixel, so the complexity is O(pixels * objects). In our approach, we want to cast rays into the scene through individual pixels and find the 'first hit', i.e., the first intersection with an object in the Far Field. This means no secondary rays have to be cast, and we are interested in so-called 'first hit' acceleration techniques.

From the first days of ray tracing, acceleration structures have been used to reduce the number of ray-object intersection tests for individual rays. The two most popular are Bounding Volume Hierarchies and Hierarchical Space Subdivision. Of all the methods proposed, the regular grid [Fujim86] approach is the most interesting for our purpose: space is partitioned into a regular grid structure, and all objects are entered into the grid cells with which they intersect.

It has been shown ([Ohta97], [Fujim86]) that theoretically, using an appropriate acceleration structure, the time complexity of ray tracing can be reduced to O(1), i.e., constant, in the number of objects (although this constant may be very large). In our experiments we have observed a sublinear rise in the time to cast rays into a very large scene.

The advantage of the regular grid is its speed. Also, given a more or less uniform distribution of objects, which we can safely assume for many types of virtual environments, the memory overhead is very low. Tracing through a grid is fast, using for example Woo's incremental algorithm [Aman97] which only requires few floating point operations per grid cell. If more objects are added, runtime behavior can even improve because rays will collide earlier with objects than if there were huge empty spaces in the grid.

The regular grid also provides a simple solution to view frustrum culling, which is necessary to quickly find the objects that have to be rendered in the Near Field.

For certain scenes, ray casting alone might already be sufficient and moderate gains can be observed. But generally, this still leaves too many pixels which have to be raycast, and while ray casting is relatively independent of scene complexity, casting a single ray is

expensive compared to polygonal rendering and thus only tractable for a moderate number of pixels. The following sections explain how Image Caching and Horizon Tracing can be used to drastically reduce the number of rays that have to be cast.

5 Image Caching

5.1 Panoramic Image Cache

Usually, walkthrough sequences exhibit a considerable amount of temporal coherence: the viewpoint changes only by a small amount between successive frames. We exploit this coherence in our system: instead of tracing every Far Field pixel every frame, we retain all the color values of the previous frame and try to retrace only those pixels that are outdated according to some error metrics.

The validity of pixels depends strongly on the type of viewpoint motion:

Forward/Backward motion: this makes up for a very large amount of motions in a walkthrough sequence. The farther away an object is, the smaller the amount of pixels it moves on the screen due to forward/backward motion. Many pixels will even remain at the same location, so just reusing the pixels from the previous frame is already a good first approximation.

Rotation: Rotation is quite different from forward/backward motion: reusing the contents of the framebuffer would indeed be a very bad solution, because all pixels would be wrong. But actually, many pixels are still valid, they have just moved to a different position. So what is needed is a method to store traced pixels that does not depend on the orientation of the viewer.

Panning (left/right, up/down): This type of movement is similar to rotation in that most pixels move to a different place.

Our assumption is that forward/backward motion and rotation will be the major types of motion in a walkthrough sequence. We therefore choose a representation which is independent of viewpoint rotation: a panoramic image cache.

Panoramic Images have been demonstrated to be a very efficient tool to store full views of a scene where rotation is allowed. We use the panoramic image cache not for presenting a precomputed panorama as for example in the Quicktime VR system [Chen95], but we use it as a rotation-independent Image Cache.

When a ray is cast through a pixel on the (flat) screen, its position on the (curved) map is calculated and the color value obtained by the ray is entered in this position. If, at a later time, another screen pixel projects to the same position in the curved map, its value can be reused if it is still good enough.

The major advantage of using a panoramic image as an Image Cache is that the validity of pixels stored in the map is invariant under rotation. This means that, as long as the viewpoint does not change, all image elements already calculated and stored in the map can be reused in the new image, provided they still fall into the viewing frustrum after rotation. This speeds up rotation considerably: only the very small amount of pixels that appears newly at the border towards which the viewer is rotating has to be retraced. All other pixels can be reused, and their values will be correct.

In the case of forward/backward movement, the behavior of the map resembles that of a normal, flat image map: reusing the previous panoramic map will be a good approximation to the image and many pixels will be in the correct location. Panning causes more pixels to be invalidated if no costly reprojection is used.

5.2 Cache Update Strategy

Assuming that pixels which have been traced in a previous frame are retained in an Image Cache, the algorithm has to decide which pixels are considered good enough according to a certain error metric, and which pixels have to be retraced. In an interactive walkthrough system, the decision can also be based on a given pixel-'budget' instead of an error metric: which

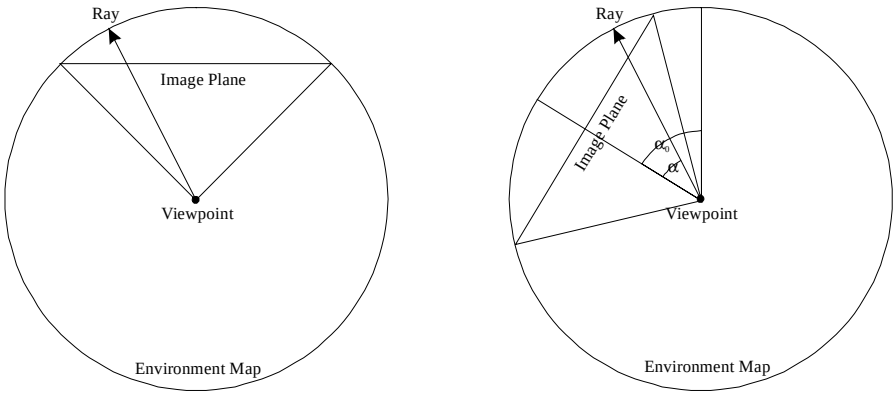

Figure 5: Indexing into the Panoramic Image Cache: Given a pixel on the image plane, an angle α can be calculated with respect to the image plane center. This angle does not depend on the initial viewer orientation α_0, therefore it can be precomputed and stored in a lookup-table. So, indexing into the Image Cache consists of looking up α in a table, adding the viewer orientation α_0 and rescaling this value to fit the resolution of the Image Cache

pixels are the most important ones to retrace, given a maximum amount of pixels available per frame.

As with any speedup-algorithm, worst-case scenarios can be constructed that do not benefit from the algorithm. In such a case, our approach allows progressive refinement by iteratively retracing all necessary pixels every frame. As soon as the scenery gets more suited to the algorithm or the observer does not move for a short moment, the system is able to catch up again.

To select an appropriate set of pixels to retrace, we assign a *confidence value* to each pixel in the map. The pixels are then ordered according to their confidence values and tracing starts with the pixels that have the lowest confidence, proceeding to better ones until the pixel budget is exhausted. Finding a good heuristic for the confidence value of a pixel is not trivial. We have chosen the following approach:

Every frame in which the observer moves more than a certain distance (rotation is not taken into account, as the Image Cache is rotation-independent), a new *Confidence-Record* is created, which contains the current observer position. All pixels which are traced during this particular frame are assigned a pointer to the current Confidence-Record (pixels which have not been traced at all point to a 'Lowest-Confidence'-Record).

After a few frames, there will be a certain amount of Confidence-Records (as many as there were distinct observer positions), and each pixel in the Image Cache will reference exactly one of those Records. This information is used as follows:

During the polygon rendering stage, a new opacity-buffer is created. All pixels of this buffer are visited sequentially. If a pixel is covered by a polygon, it is ignored. If not, a lookup is done into the Image Cache to find out the Confidence-Record associated with this pixel. The Confidence-Record also contains a pixel counter which is then incremented.

```
CacheElement {
   Color;
   Pointer to ConfidenceRecord;
};
ConfidenceRecord {
   ObserverPosition;
   PixelCounter;
   CurrentConfidence;
};
```

Figure 6: The basic data-structure used for in the Cache and for keeping track of confidence values

After all pixels have been visited, each Confidence-Record contains the number of pixels that refer to it in its internal counter. Now, the observer position stored in each Confidence-Record can be compared to the current observer position and the distance between the two is remembered as the current confidence value of this Confidence-Record. All Confidence-Records are sorted according to this confidence value.

Scanning through the Confidence-Records from worst (farthest away) to best (nearest), we add up the counted pixels until the pixel budget is met. This gives a threshold distance: all pixels farther away than this threshold distance will be retraced. Nearer pixels will be reused from the Image Cache.

This is accomplished by going again through the opacity buffer, indexing into the Image Cache for every unoccluded pixel and casting a ray for the pixel if its distance (which can be found out by following its pointer to the associated Confidence-Record) is greater than the threshold distance.

The problem with this approach is that it occurs quite often that all pixels have the same confidence value: if the observer stands still for a while and then suddenly moves, all pixels will be assigned the same new confidence value. In this case, the confidence values are not a good indication of where ray casting effort should be spent. We therefore only trace every n-th pixel that has the same distance, such that the pixel budget is met. In the subsequent frame, the remaining pixels will then be selected automatically for retracing.

The distance between the current and previous observer position is used as an error-estimation, because the panoramic image map is rotation-independent, hence the only value that changes between frames with respect to the map is the observer position. A more elaborate scheme could also store orientations with each Confidence-Record and compare this to the current orientation. For reasons of efficiency, we have chosen the more simple approach of keeping the error estimation rotation-independent. It would be interesting to investigate whether performance improves if one takes additional information about the hit object or the distance to the intersection point into account.

To sum up, our update strategy makes sure that pixels are retraced in the order of their distances to the current observer position, taking into account a pixel budget that allows for 'graceful degradation' if the demand for pixels to be retraced is too high in a particular frame. Note that on average, the area left for ray casting only covers a small portion of the screen.

6 Horizon Tracing

Most virtual environments share the following properties: they have

- a polygonal floor
- either a polygonal ceiling or
- empty sky

For indoor-scenarios with a polygonal ceiling, the system as presented so far would already be sufficient, but a problem arises if there are large areas of empty sky. Theoretically, the ray tracing acceleration structure should take care of rays that do not hit any object in the

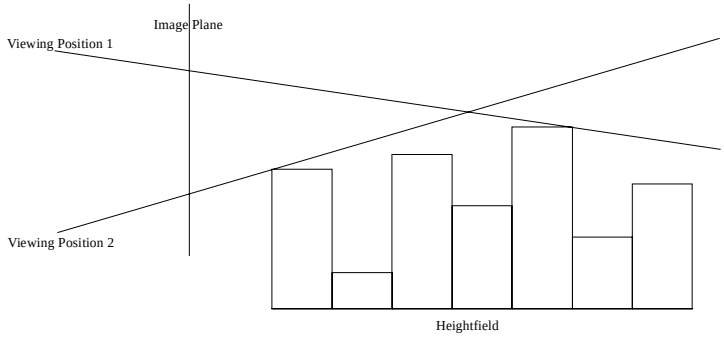

Figure 7: The image shows a cut through the heightfield along the path of one particular horizon ray cast from two viewing positions with different heights. Note that it is not always the highest point in the heightfield that determines the height of the horizon on the screen

scene. But in fact, even the overhead of just setting up a ray for every background-pixel is much too large as to be acceptable. The usual case in outdoor scenes is that between one third and one half of the pixels are covered by polygons. A very small part is covered by Far Field pixels that do hit objects, but the rest of the screen is covered by sky.

If it were possible to find out where the sky actually starts, most of the sky pixels could be safely ignored and set to a background-color or filled with the contents of a static environment map.

We assume that the viewer only takes upright positions, i.e., there is no head tilt involved. This is a reasonable restriction in a walkthrough situation. Then, we observe that the screen position where the sky starts only depends on the x-coordinate in screenspace, i.e., on the pixel column. So, for every pixel column we have to find out the y-coordinate of the horizon.

This, again, is a problem that can be solved by ray tracing, but in 2-dimensional space. In addition to the 3D regular grid that is used for tracing pixels, a 2D regular grid is created that contains the height value of the highest point in each grid node - a 2-dimensional heightfield.

For every frame, a 2-dimensional ray is traced through this heightfield to find the grid node that projects to the highest value in screenspace (note: this need not be the highest point in absolute coordinates!). All pixels with a height above this value can be ignored and set to the background color.

Our results indicate that the reduction in the number of pixels to trace was so substantial that the total time spent ray casting and the time spent horizon tracing were comparable. This makes horizon tracing itself a further candidate for acceleration.

One way to speed up horizon tracing is to carefully adjust the resolution of the height field. As opposed to pixel ray casting, rays cast through the heightfield have to travel through the whole 2D grid so as to find the point whose projection has the highest y-value on the screen. Whereas the 3D grid profits from higher resolution because of improved intersection culling, it is detrimental for horizon tracing because of the large number of grid cells that have to be visited. Even though a coarser grid tends to overestimate the horizon height, the speedup gained through faster horizon tracing makes up for this.

Another way to speed up horizon tracing is to apply the principle of graceful degradation to the Horizon Map in the same manner as to the Image Cache: as long as the viewer is moving, the horizon is subsampled and the locations between samples are filled with the maximum of the adjacent samples.

7 Implementation and Results

The algorithms described in this paper have been implemented and tested in an application environment for creating professional computer games. The system was tested with a Pentium 233MMX processor, which is moderately fast for a consumer PC. The 3D board used was a 3DFX Voodoo Graphics, a reasonably fast board for PC-standards. Even better speedups might be possible using a faster main processor. The implementation is still very crude, and it is likely that additional performance gains can be achieved by careful optimization of critical per-pixel operations.

One problem that has to be solved is how to create an occlusion map, and how to reuse it for rendering. Surprisingly, graphics hardware is not of much help in this case: transfers from frame buffer memory to main memory are usually very slow, except in some specialized new architectures which incorporate a Unified Memory concept (e.g., the Silicon Graphics O2 [Kilg97]).

Therefore, while the Near Field is rendered in the frame buffer using graphics hardware, we create the 1-bit opacity buffer with a very fast software renderer, taking advantage of the fact that neither shading nor depth information is required for the opacity buffer.

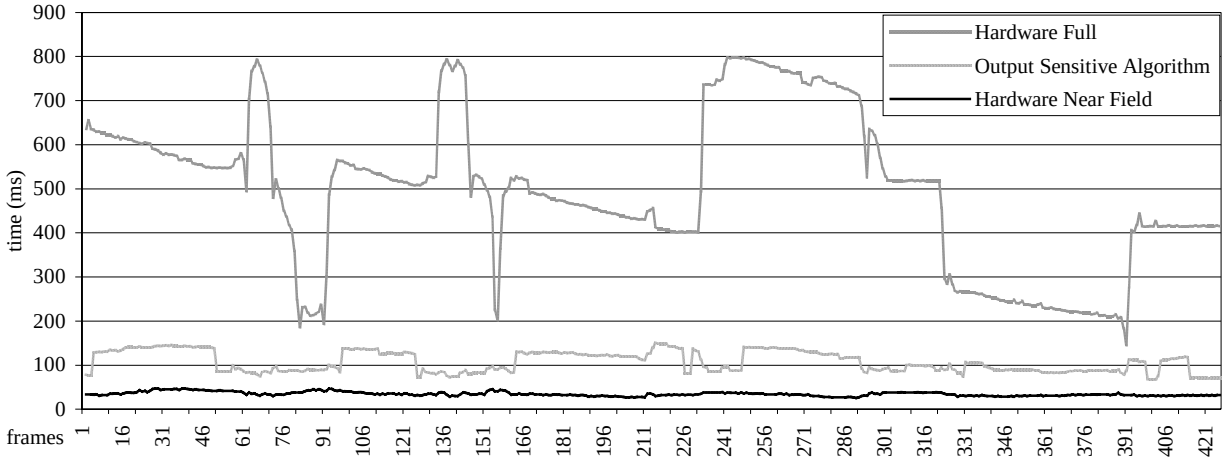

Figure 8: The chart compares full hardware rendering (backplane set to infinity), our new output sensitive algorithm and hardware rendering (Far Field not rendered) with the backplane at 100m. The image resolution was 640x480 pixels for all tests. The average frame rates were 2.0 fps for full hardware rendering and 9.25 fps for the new algorithm, so the speedup is about 4.6

Our current implementation is limited to upright viewing positions only. This restriction is inherent to the horizon tracing acceleration, and we believe that it does not severely infringe on the freedom of movement in a walkthrough environment. With respect to the Image Cache, a spherical map could easily be used instead of the cylindrical map that we chose to implement, allowing the viewer to also look up and down.

The first graph (figure 8) shows the time taken to render each frame of a recorded walkthrough sequence (about 400 frames) through a very large environment, a huge city (containing approximately 150000 triangles). Two of the series are for pure hardware rendering only, with the backplane set to infinity in one case and 100m in the other case. The Far Field is not rendered at all, and our algorithm disabled completely (so there is no overhead for tracing horizon pixels, creating or going through the opacity buffer etc.). It shows that up to a certain distance, graphics hardware can render the scene very quickly, but of course misses out on a considerable amount of the background. But if the whole scene is rendered indiscriminately, the hardware simply cannot cope with the amount of triangles, and the framerate drops to an unacceptably low value.

Obviously, these are the two extremes between which our algorithm can or should operate. It will certainly not get faster than just rendering the Near Field, but it should be considerably faster than rendering the whole scene with triangles only. The third series shows how our algorithm performs for the same walkthrough sequence with the backplane set at 100m. A speedup of up to one order of magnitude over rendering the full scene in hardware can be observed.

To give a feeling for the operating behavior of the algorithm, the second graph (figure 9) shows frame times for our algorithm with different Near Field sizes (i.e., the backplane set to different values). Increasing the back plane distance beyond a certain limit reduces performance, because more triangles have to be rendered, but they do not further reduce the number of pixels that have to be traced. Setting the backplane too near gives a very non-uniform frame-rate.

The third graph (figure 10) gives an impression of

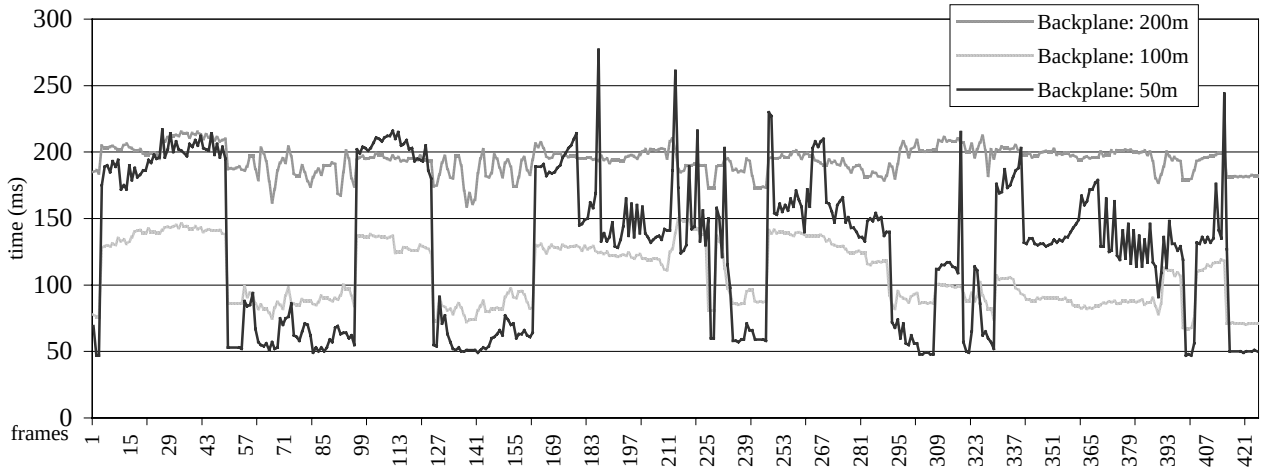

Figure 9: The chart compares the behavior of the algorithm with respect to the size of the Near Field at a resolution of 640x480 pixels

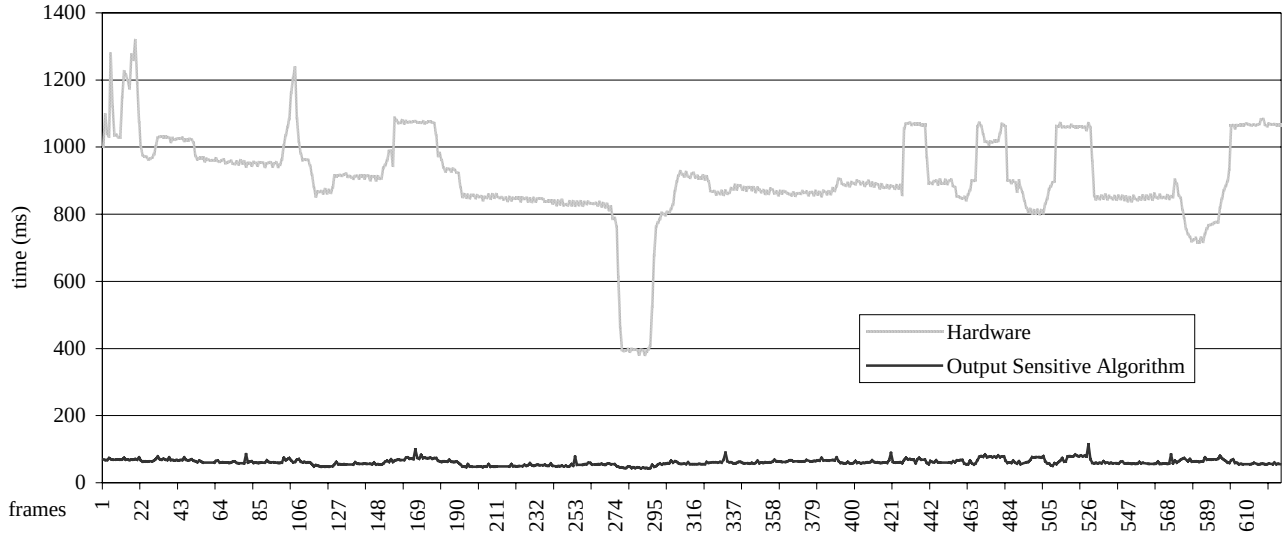

Figure 10: Frame times for a different walkthrough sequence at a resolution of 320x240 pixels. The average number of polygons in the viewing frustrum was higher than in the first sequence, making hardware rendering even slower. The average frame time for hardware rendering was 1.1 fps, for the new algorithm 16.3 fps, so the speedup is 14.8

what the algorithm is capable of if the screen resolution is reduced and the scenery is more complex (in this walkthrough sequence, almost all of the polygons were in the viewing frustrum most of the time, so view frustrum culling is not able to cull geometry). The following chart (figure 11) shows that the performance of the output sensitive algorithm is due to heavy occlusion in the walkthrough sequence:

Percentage of pixels...	Min	Average	Max
ray cast and hit an object	0,00%	0,13%	0,49%
ray cast and missed	0,00%	0,10%	0,38%
taken from the image cache	0,00%	0,13%	0,81%
culled by horizon tracing	7,38%	23,76%	39,21%
covered by polygons	60,37%	75,89%	92,62%

Figure 11: Illustrates that very few pixels have to be calculated using ray casting in a densely occluded environment

The images at the end of the paper show two views of the virtual city the walkthroughs were recorded in. The border to the Far Field is indicated by a line. To the left there is a view from an elevated position which does not satisfy the assumptions because there is no significant occlusion. The expected framerate for such a view is about 4 frames per second for a resolution of 640x480 pixels (triangles only would be about 1 frame per second). The image to the right represents a typical shot from a walkthrough sequence that does fulfill our assumption of dense occlusion.

8 Discussion

8.1 Scalability

The rendering algorithm described in this paper is applicable to a wide range of environments (see applications). The same is true for the type of platforms it can be used on. Originally, it has been designed with the consumer PC in mind, where almost every new PC is equipped with a 3D accelerator. These accelerators share a common property: they are very good at triangle rasterization, but the transformation step has to be done by the main CPU. Rendering scenes that contain a lot of primitives easily overloads the transformation capabilities of the CPU, and the 3D card is idle. Instead of transforming all primitives with the CPU, the algorithm can put this processing power to better use: by using the methods described, and some a priori knowledge about the scene, the 3D accelerator is used to quickly cover the Near Field with polygons, and the remaining CPU time is used for the pixel based operations.

The algorithm is not restricted to such a platform, though. As the power of the 3D pipeline increases, the size of the Near Field can be increased as well, thus leveraging the additional triangle processing power of the pipeline. More pixels will be covered by polygons, and even fewer pixels will be left to send to the ray casting step. This is especially true if the geometry transformation stage is implemented in hardware, as is the case in higher end PC solutions and midrange 3D workstations.

But even if a high end graphics pipeline exists, the ideas of this paper are valid: there will always be scenes too large and too complex to handle even with the best graphics hardware. Adjusting the size of the Near Field to the speed of the polygon pipeline provides a good parameter for tuning an application for speed on a specific platform.

This means that the approach scales very well with CPU processing power as well as with graphics pipeline speed, and the result is an output-sensitive algorithm that can be used in many different setups.

8.2 Aliasing

No speedup comes without a cost. There are two reasons why aliasing occurs in the algorithm: first, ray casting itself is a source of aliasing because the scene is point sampled with rays. The other reason is the aliasing due to the projection of the flat screen into a curved image map and back.

In both cases, antialiasing would theoretically be possible, but it would have a heavy impact on the performance of the algorithm, thus defying the purpose of the algorithm, which is to accelerate interactive walkthroughs.

9 Applications

There is a variety of applications where the algorithms presented in this paper could be applied. Foremost, there is:

9.1 Walkthroughs

Many types of virtual environment walkthroughs fulfill the basic preconditions the algorithm requires. Most and foremost, urban environments are ideal to showcase the points of this paper. Especially in a city, most of the screen is covered by the houses that are near to the viewer. But there are also several viewpoints where objects are visible that are still very far away - imagine looking down a very long street. Polygons cover the right, left and lower part of the image, a good part of the sky is caught by horizon tracing, and the remaining part can be efficiently found by ray casting. Note that under normal circumstances, such scenes are either excruciatingly slow to render, or the backplane distance is simply set so near that the result does not look very convincing.

Any other scenery which is densely occluded is also suitable. For example, walking through virtual woods is very difficult to do with graphics hardware alone - but with our algorithm, a good number of trees could be rendered in the Near Field, and the remaining pixels traced.

9.2 Computer Games

In recent times, first person 3D computer games have gained immense popularity. Many of them are restricted to indoor-environments, because portal rendering provides a very good solution for the complexity problem in this case. But few have ventured to outdoor scenarios, and most of those who have make use of heavy fogging to reduce the amount of polygons to render. Sometimes the back plane is not set much farther than 10-20 meters, which does not provide for a very realistic feeling. Using the described algorithm, the perceived backplane can be pushed back to the horizon, or at least a considerable distance further away, as the space between the previous backplane and the horizon can be covered by Far Field rendering.

Neither graphics hardware nor processing power will be lacking for computer games in the near future, as both are rapidly catching up with workstation standards. The benchmarks were done on a system whose performance is by no means ‘state of the art’ even for a PC environment (see results-section) on purpose, to show that good results can be achieved nevertheless.

9.3 Portal Tracing

Previous work [Alia97] has suggested the use of textures as a cache for rendering portals in indoor environments. Those textures are calculated by using the graphics hardware. We propose that under certain circumstances, it might be advantageous to use ray casting to trace through the portals: far away portals cover only a small amount of space on the screen, so there are very few pixels to trace, but the amount of geometry behind a portal can still be quite large, especially if portals have to be traced recursively. Of course in this case, the horizon tracing stage can be omitted.

9.4 Effects

A potential application for some of the ideas of this paper is to render certain special effects that do not require high accuracy: reflections on partly reflective surfaces can be adaptively raytraced using only a small number of rays - the effect would be visible, but one can avoid having to rerender the whole scene multiple times as is usually necessary for such effects.

10 Conclusions and Future Work

We have presented an algorithm which is capable of considerably speeding up rendering of large virtual environments. In scenes where our basic assumptions hold, speedups of an order of magnitude have been measured.

We believe that our way of partitioning the scene into Near Field and Far Field is a sound approach, as we have been able to demonstrate with examples. There is still a lot of work in carefully studying the behavior of the system with respect to scene complexity, overall ‘type’ of the scene and to the algorithm parameters. We plan to investigate ways to automatically determine such parameters as backplane distance, number of rays to trace per frame and grid resolution, so as to always provide near optimal performance.

There is no reason why this system could not be combined with other approaches like geometric level of detail or textured impostors. Especially the latter are very interesting for moving objects, as our algorithm as yet only deals with the static parts of a scene. It has to be pointed out, however, that many algorithms have very elevated memory requirements, which could pose a

problem for the type of scenes we imagine. The current algorithm is not very memory intensive as long as there is a certain amount of uniformity in the scene distribution.

Another interesting avenue of research is the use of graphics hardware for the image-based operations we have introduced. With systems that allow access to frame buffer and texture memory with the same speed as to the system memory, it might be possible to let the hardware do the reprojection of the environment map onto the screen. For example, a simple extension to our current system would be to use graphics hardware to create the opacity buffer.

11 Acknowledgments

This research is supported in part by the Austrian Science Foundation (FWF) contract no. p-11392-MAT. We would like to thank Ars Creat Game Development for letting us use the 'Ars Machina' framework of their upcoming computer game.

References

[Alia97] D. G. Aliaga, A. A. Lastra. Architectural Walkthroughs Using Portal Textures. IEEE Visualization '97, pp. 355-362, November 1997.

[Aman87] J. Amanatides, A. Woo. A fast voxel traversal algorithm for ray tracing. Eurographics '87, pp. 3-10, North-Holland, August 1987.

[Chamber96] B. Chamberlain et. al. Fast rendering of complex environments using a spatial hierarchy. Graphics Interface '96, pp. 132-141, May 1996.

[Chen95] S. E. Chen. QuickTime VR - An Image-Based Approach to Virtual Environment Navigation. Computer Graphics (Proc. SIGGRAPH'95), pp. 29-38, 1995.

[Cleary88] J. G. Cleary, Geoff Wyvill. Analysis of an algorithm for fast ray tracing using uniform space subdivision. The Visual Computer, 4(2), pp. 65-83, July 1988.

[Fujim86] A. Fujimoto, T. Tanaka. ARTS: Accelerated Ray Tracing System. IEEE Computer Graphics and Applications, 6(4), pp. 16-26, 1986.

[Glass87] A. S. Glassner (ed.). An Introduction to Ray Tracing. Academic Press, 1989.

[Greene93] Hierarchical Z-Buffer Visibility. Computer Graphics (Proc. SIGGARPH'93), 27, pp. 231-238, 1993.

[Heckb97] P. Heckbert, M. Garland. Survey of Polygonal Surface Simplification Algorithms. Technical Report, CS Dept., Carnegie Mellon U., to appear (draft May'97) (http://www.cs.cmu.edu/~ph/).

[Kilg97] M. Kilgard. Realizing OpenGL: Two Implementations of One Architecture. 1997 SIGGRAPH / Eurographics Workshop on Graphics Hardware, pp. 45-56, August 1997

[Klim97] K. S. Klimaszewski, Thomas W. Sederberg. Faster Ray Tracing Using Adaptive Grids. IEEE Computer Graphics and Applications, 17(1), pp. 42-51, January 1997.

[Luebke95] D. Luebke, Chris Georges. Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets. Proc. Symp. Interactive 3-D Graphics, ACM Press, April 1995.

[McMill95] L. McMillan, G. Bishop. Plenoptic Modeling: An Image-Based Rendering System. Computer Graphics (Proc. SIGGRAPH'95), 29, pp. 39-46, 1995.

[Ohta87] M. Ohta, M. Maekawa. Ray Coherence Theorem and Constant Time Ray Tracing Algorithm. Computer Graphics 1987 (Proceedings of CG International '87), pp. 303-314, Springer-Verlag, 1987.

[Raff98] M. Rafferty, D. Aliaga, A. Lastra. 3D Image Warping in Architectural Walkthroughs. IEEE Virtual Reality Annual International Symposium '98 (Atlanta, GA, 14-18 March 1998), 228-233

[Schaufler96] G. Schaufler, W. Stürzlinger. A Three-Dimensional Image Cache for Virtual Reality. Computer Graphics Forum (Proc. EUROGRAPHICS'96), 15(3), p. C227-C235, C471--C472, September 1996.

[Shade96] J. Shade et. al. Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments. Computer Graphics (Proc. SIGGRAPH'96), 30, pp. 75-82, 1996.

[Shade98] J. Shade et. al. Layered Depth Images. Computer Graphics (Proc. SIGGRAPH 98), pp. 231-242, July 1998.

[Sudar96] Output-Sensitive Visibility Algorithms for Dynamic Scenes with Applications to Virtual Reality. Computer Graphics Forum, 15(3), pp. 249-258, Blackwell Publishers, August 1996.

[Teller91] S. J. Teller, Carlo H. Séquin. Visibility preprocessing for interactive walkthroughs. Computer Graphics (Proc. SIGGRAPH '91), 25(4), pp. 61-69, July 1991.

[Torb95] Talisman: Commodity Real-time 3D Graphics for the PC. Computer Graphics (Proc. SIGGRAPH 96), pp. 353-364, August 1996.

[Zhang97] H. Zhang et. al. Visibility Culling Using Hierarchical Occlusion Maps. Computer Graphics (Proc. SIGGRAPH'97), 31(3A), pp. 77-88, August 1997.

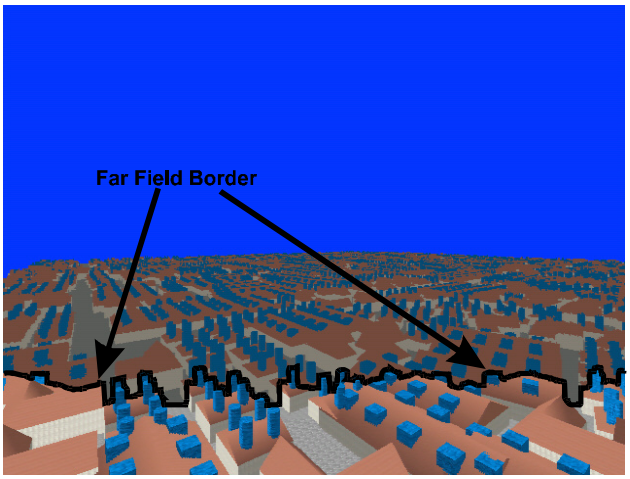

Figure 12: A view over much of the virtual city that was used for the walkthroughs

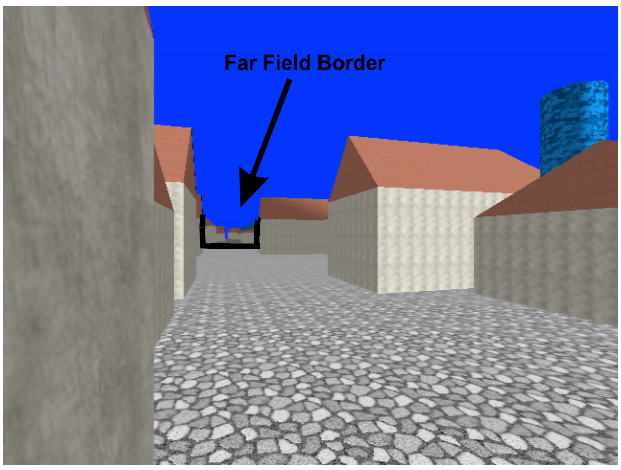

Figure 13: A typical view from a walkthrough sequence in the city