Fast Ray Tracing of Sequences by Ray History Evaluation

Arno Formella
International Computer
Science Institute
Berkeley, CA, 94704, USA

Christian Gill, Volker Hofmeyer
Computer Science Department
University of the Saarland
66041 Saarbrücken, Germany

Abstract

We present a method to reduce the time needed to render a sequence of ray traced images. We exploit the temporal and spatial coherence between consecutive frames. The algorithm does not only inspect the image plane in order to find regions with minor changes; the object space is analyzed by evaluating the so called ray history. With this method we obtain a speedup in the range of 1.2 to 2.8 almost without visible loss of quality. The method is more or less independent of the used ray trace algorithm and can be implemented in many ray trace programs without too much effort.

1 Introduction

Ray tracing is a well known rendering technique which produces high quality images. It simulates geometric optics by tracing light rays backwards from the view point into the object space. So reflections at specular surfaces and refraction through transparent materials can be calculated. The main disadvantage of ray tracing lies in its excessive computational complexity: most of the time is spent in finding intersection points between millions of rays and thousands of objects. Nevertheless todays fast computers and advanced ray tracing algorithms allow to render single frames within minutes.

Ray tracing was first introduced by Appel in [3]. Early optimizations, such as recursion, bounding volumes and adaptive oversampling, which led to significantly better run times, have been studied by Whitted in [25]. Since then a lot of improvements to the basic algorithm have been developed. Excellent references to the ray tracing literature are gathered for instance in the bibliographies of Speer in [24] and of ACM SigGraph in [1].

We focus on a method to reduce the render time for a sequence of ray traced frames. This means that not a single frame is rendered faster, but information of previous calculations is re–used in consecutive, correlated frames. We use the coherence normally found in a typical picture. It can be observed that coherence occurs not only within one frame but also in a sequence of consecutive frames. Section 2 describes the different types of coherence which can be distinguished. In section 3 we introduce ray–object intersection trees and ray histories. Adaptive undersampling is described as well. Section 4 explains our algorithm. In Section 5 we summarize the results we obtained for a variety of films. Section 6 deals with the limitations of this method, because an undersampling algorithms almost always has its deficiencies. However, we also state some further improvements which may be worth while to be implemented in order to reduce aliasing effects. Section 7 concludes briefly.

2 Coherence

In a typical picture adjacent pixels are closely correlated. Often they have similar colors or at least they belong to the image of the same object. This coherence in image space remains visible during time looking at the same pixel in consecutive frames. Because often objects are moving slowly in space, their images stay at almost the same positions in the frames. Additionally, the perspective projection onto a two dimensional screen reduces the velocity which can be seen. For instance, an object moving towards the camera becomes only larger but does not change its position.

The spatial and temporal coherence in case of ray tracing means the following. Rays which are traced to calculate the color of adjacent pixels almost run in parallel or at least the ray–object intersection history is very similar, i.e. the tree of rays bounces between the same objects. In section 4 we address this point more precisely.

Naturally, the spatial coherence is the base of many subdivision strategies using hierarchical data structures. Glassner suggests in [10] the octree to speedup the process of finding the intersection points. In [11] he extends the data structure with a time axis to allow for animations. Ray tracing with ray classification (i.e. grouping rays according to their direction) is proposed by Arvo et al. in [4] as a very fast rendering algorithm. Introducing the time as an additional dimension to this approach is done by Groller et al. in [12].

The fact that a bunch of rays can be traced in parallel casting so–called beams is introduced by Heckbert et al. in [15] and expanded by Hanrahan in [14]. They use recursive subdivision of the beams to find edges of objects. Caches and breadth first search in the ray histories are added to exploit the parallel nature of the considered rays.

Marks et al. claim in [20] that the expected advantage of algorithms looking for coherence in an image finally may increase the run time, because the overhead consumes more time than the straight forward algorithm. Their analysis deals with scanline algorithms; however, they address ray tracing too and state that the resolution of the image has a great influence on the obtained speedup. A similar result has been obtained by Speer et al. in [23] with theoretical and empirical means. They say, that although similarity can be found between ray histories of adjacent pixels, it seems difficult to exploit this in an algorithm. The main obstacle is to ensure the validity of ray–object intersections.

Coherence for shadow casting is used by Haines in [13] in such a way that the last object which cast a shadow for one pixel is likely to cast a shadow for the next one as well. Pearce at el. use in [22] a whole set of small objects, which are defined through a grid in object space, as a shadow casting cache.

Often, almost parallel rays are intersected with the same object. This knowledge can be used to reduce the time spent in the intersection calculation. As examples for the case of parametric surfaces we refer to the works of Kijiya in [18], Joy et al. in [17] or Lischinski [19].

Badt suggests in [5] an image space temporal coherence algorithm to accelerate frame by frame rendering. The first frame is traced entirely. In the following frames, first only randomly or heuristically chosen pixels are traced. Their neighbors as well in time as in space are floodfilled with the color of the seed until a significant difference to the first frame is encountered. Then, real ray tracing is performed again. As an heuristic a reprojection algorithm is presented.

Another method to reduce the time for finding intersection points is introduced by Chapman et al. in [6]. Assume that all polygons and the camera are moving on curves defined through polynoms, if they move at all. Every ray describes a polynomial, maybe piecewise, curve on the polygon which it hits during the motion. The algorithm calculates for each pixel those curves which has to be displayed in the animation. Furthermore, all segments of the curves are detected, which are covered eventually by other polygons moving around. Once all visible segments are precalculated, rendering can be performed fast.

Pixel selected ray tracing is introduced in [2] and incremental ray tracing is presented in [21]. Those are the bases for our algorithm and thus explained in the next section in more detail. It is possible to include many of the optimizations cited above in this algorithm.

3 Ray History

The kernel of each ray tracing algorithm is the following. For every pixel of the image plane at least one ray is cast into the object space. If the ray hits an object three actions take place, which may depend on the properties of the material. In the first place, a transmitted and a reflected ray are cast according to the laws of geometrical optics. Usually this is implemented recursively. Secondly, a shadow testing ray is cast to every light source, which may illuminate the object. If such a shadow ray hits any object the intersection point under consideration is in shadow respective to this light source. Finally, the properties of the material at the intersection point (e. g. textures) and the recursively gathered information are combined to calculate the definite color of the pixel.

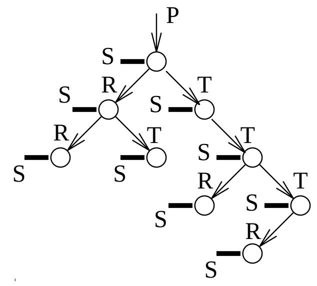

Figure 1: Ray–object intersection tree

Figure 1 shows a possible ray–object intersection tree, which is obtained by the recursion. P stands for a primary ray sent through a pixel. R and T denote the refracted and transmitted rays. S represents the shadow rays. If no optimization is implemented a shadow ray is cast for every intersection point to every light source. A node contains all necessary information for two purposes. It must be possible to compare two trees which are constructed at the same time for different pixels to exploit spatial coherence and it must be possible to compare two trees at different times for the same pixel to exploit temporal coherence.

Akimoto et al. use the ray–object intersection trees for adaptive undersampling in [2]. They decided recursively in four levels how to determine the color of the center pixel of a square if the four corners are precalculated (see figure 2). If different objects occur in the ray–object intersection trees of the corners real ray tracing is performed for the center (level 1). If only the shadow information is different, the precise intersection points of the actual rays with the objects and the shadows are calculated (level 2). Note that no objects must be found, the information in the tree is used instead. For the shadow rays at least a small cache of objects can be implemented. Properties of the material of the object might make it necessary to calculate the exact intersection point to determine the color (level 3). In the case that there is a plain material and the intensities of the corner pixels do not vary too much, the color of the center pixel is computed by interpolating the colors of the corners (level 4). We call this process *fast ray tracing*. The size of the inital square has a great influence on the quality of the image. They suggest a 5×5 pixel square.

Murakami and Hirota [21] extend the data structure of a ray–object intersection tree. Their ray tracer uses as subdivision method a regular grid as it has been introduced by Fujimoto [8]. They add to the ray–object intersection tree a voxel traversal history and intersection histories. The voxel traveral history is a list of all voxels traversed by the incoming ray of the node. An intersection history is a sortet list of all intersection points which are found in those voxels

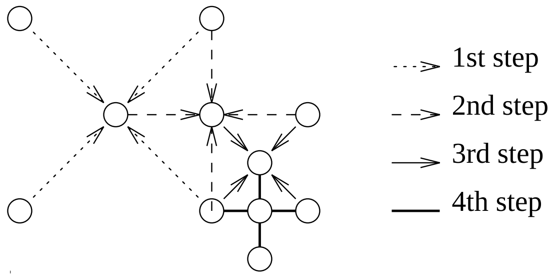

Figure 2: Adaptive undersampling

in the list which contain moving objects (note, that the shadow rays are not included at this point in their approach). If once for the initial frame all extended intersection trees are calculated, for the following frames only those rays must be updated, whose voxel traversal histories contain voxels with moving objects. In order to accelerate this search a hashing scheme is given, which maps the occuring lists to hash indices. During a preprocessing step all those lists, i.e. their hash indices, are flagged in the hash table which contain a voxel with moving objects. The update of the rays can be made fast, because the intersection histories contain all previous intersection points and additional tests must be performed only with the moving objects. The amount of memory to implement the data structures is very high and a moving camera can not be handled. Shadow rays are included in a similar manner without intersection histories.

We use a similar approach as in [2] to implement adaptive undersampling in time. We apply the concept of extending the ray–object intersection trees found in [21] as well, but reduce it to single pixels. The next section describes our algorithm. It depends on the basic ray tracing algorithm what information must be stored along with the ray–object intersection tree. We will not describe exactly what should be stored at each node of the tree in order to allow for a fast implementation of the algorithm in a particular renderer, we simply call such an annotated ray–object intersection tree *ray history*. A few hints are given in section 5 where our implementation is discussed.

4 Algorithm

The algorithm performs adaptive undersampling in time. For simplicity we consider only one pixel in the image plane. In order to generate a frame a loop over all pixels of the frame must be added. At this point oversampling can be handled in the standard way as well.

Let the film be composed of n frames which have to be rendered at times $t_1, \ldots, t_n$. Let k be the length of the time interval undersampling is started with. The film is divided into sequences of length $k+1$ in such a way that the first frame in each sequence is the same as the last frame in the previous one. This allows us to formulate the algorithm recursively. First, the ray histories and thus the colors at the beginning and at the end of the sequence are calculated. The ray histories are evaluated in order to calculated the rest of the sequence using fast ray tracing introduced in section 3. If the ray histories differ the sequence is divided into two sequences which are processed in the same way. The recursion stops if the sequence has length three. Clearly, we compute the ray histories only once for a pixel, the information is passed to subsequent steps of the recursion. Figure 3 illustrates the algorithm for $k = 8$ and $n = 17$.

If it happens that during fast ray tracing a ray does not hit the next object proposed by the tree the algorithm starts real tracing for that pixel. As an improvement one could continue with real ray tracing rather than starting with the primary ray, but this can be more complicated to be implemented in a given ray tracer. Up to now we have not used the interpolation level (level 4) introduced by Akimoto et al.

Clearly, the algorithm renders films with little or no geometrical changes much faster than a frame–by–frame ray tracer. So it can be used especially if the motion in the film is based on animations which do not change the ray paths such as color changes, texture animation or light variations. Nevertheless in our experiments we have found out that even with camera movements significant speedups can be achieved.

5 Results

Before we give some practical results, we will discuss what at most can be gained and what at most may be lost applying our method in a particular renderer.

The total run time T_{frame} of a ray tracing renderer to generate one frame can be divided into three parts:

$$T_{frame} = T_{pre} + T_{inter} + T_{post}$$

The time T_{pre} is the time spent in preprocessing the data, i.e. reading input data files and building internal data structures. T_{inter} denotes the time to find all objects which are hit by the rays in such a way that the intersection point is necessary during image generation. T_{post} is the time to calculate the precise intersection points and to apply the illumination model. Mapping textures on the surface can have a great impact on the postprocessing time; writing the image is covered by T_{post} as well. Note that T_{pre} does not depend on the number of rays. It depends more or less only on the number of objects. In contrary T_{post} does not depend on the number of objects but on the number of rays. The major part of the run time is covered by T_{inter} and clearly it depends both on the number of objects and the number of rays.

Now, building the ray–object intersection trees increases T_{inter}, which we denote with a factor $(1+\alpha)$. In order to generate the color of a pixel for every ray the ray history must be considered. This can be expressed by an increase of T_{post} denoted by $(1+\beta)$. If we initially generate a ray tree every k frames, we need — as best case — only once a compare of two trees for a sequence of k frames. The size of the trees is correlated to the number of rays, so we can add the compare time to T_{post}, let's say by γ. If we use a renderer with partial run times as stated above we get as

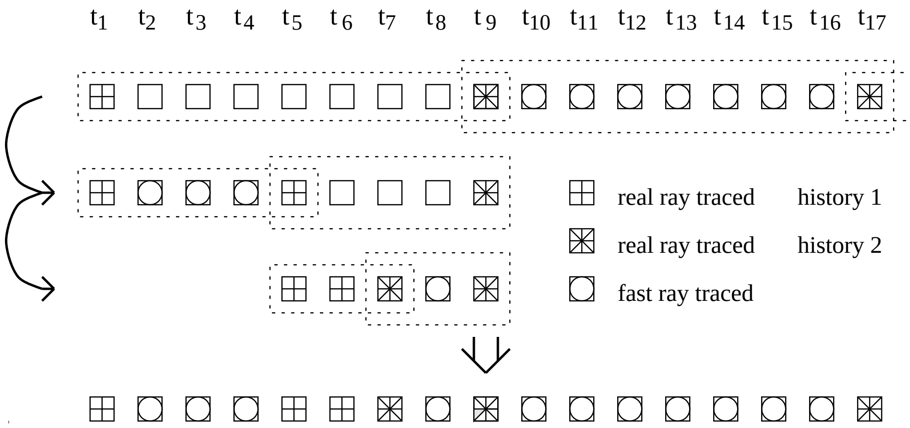

Figure 3: Illustration of the recursion of the algorithm

a lower bound of the run time for k frames

$$\begin{aligned} T_{seq} \quad &\geq \quad T_{pre} + (1+\alpha) * T_{inter} \\ &\quad + ((k-1)(1+\beta) + 1 + \gamma) * T_{post} \end{aligned}$$

We assume almost the same preprocessing time, because the additional data specifying the movements in the scene normally will be essentially smaller than the amount of static data (consider simple camera movements as an example). However, this must not be true in every case; it might be necessary to update the internal data structures of the renderer because objects have moved in space, which consumes time, too. We will not consider this in the analysis. The lower bound can be reached approximately if there is no geometrical change in the scene during the time interval of k frames.

In the worst case, which occurs if the ray history does not allow to calculate the intermediate $k-1$ frames faster than through real tracing, we get the following upper bound

$$T_{seq} \leq T_{pre} + k(1+\alpha/2)T_{inter} + k(1+\gamma/2)T_{post}$$

with the following explanations. Each pixel in the k frames is really ray traced, so time T_{inter} occurs k times. Additionally for half of them the ray–object intersection tree is constructed, so we add $\alpha/2$. Similarly for half of the pixels two trees are compared, which explains $\gamma/2$. β does not occur, because the ray history is not used.

It is well known that a tracer spends most of his time in finding intersection points. So let us assume that T_{inter} dominates with 80% the run time of the single frame renderer. The rest of the time we will distribute equally between T_{pre} and T_{post}. Those assumptions might not be true for a given renderer — in fact in our implementation T_{pre} is less than 2% of the total render time — but the following discussion shows how the speedup can be calculated if the particular run times of a single frame renderer are known. Further, let us assume that a single frame tracer needs only once the preprocessing time if k consecutive frames are rendered. The minimal and maximal speedup of our method can be estimated with

$$s_{min} \quad = \quad \frac{0.1 + k * 0.9}{0.1 + k(1+\alpha/2) * 0.8 + k(1+\gamma/2) * 0.1}$$

$$s_{max} = \frac{0.1 + k * 0.9}{0.1 + (1+\alpha) * 0.8 + ((k-1)(1+\beta) + 1 + \gamma) * 0.1}$$

If we assume a 10% overhead, i.e. $\alpha = \beta = \gamma = 0.1$, then we get the diagram in figure 4, which shows the minimal and maximal speedup. Even if we choose $k = 4$ relatively small, i.e. rendering of three out of four frames is accelerated, a maximal speedup of 2.5 can be achieved.

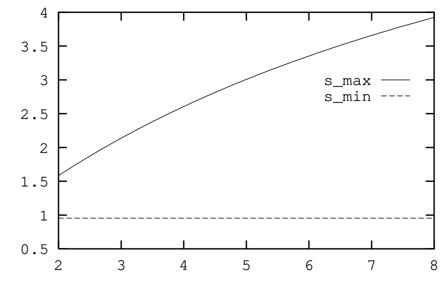

Figure 4: Minimal and maximal speedup depending on the length k of a sequence

If the overhead is large, e. g. $\alpha = \beta = \gamma = 1$, on scenes with no geometrical change after all a speedup of 1.5 is possible (again with $k = 4$). In the case that T_{pre} is very small, e. g. $T_{pre} = 0.01 * T_{frame}$, a maximal speedup of about 4.2 can be expected for $k = 8$ (with $T_{post} = 0.1 * T_{frame}$). The worst case depends mostly on α, which means that the speedup (better "slow down") can be approximated with $s_{min} = 1 - \alpha$. Note, that this definition of speedup is based on infinite long films, i. e. n should be large, because we do not count the time for the last frame. Practically, the speedup will depend on the length of the film and it will be a little bit smaller as it has been stated above.

The algorithm has been implemented in a version of CGR, a fast ray tracer with automatical object oriented space subdivision [9] and [7]. We now compare the standard ray tracing time with our new approach. In order to compare two trees we store the following information in the nodes: a pointer to the basic object, a pointer to the material and a pointer to the transformation matrix, because the objects are embedded in a hierarchy and with the matrix we can distinguish different "incarnations" of the same basic object. Additionally we store a list of light sources at the nodes. In this list we hold for each light source the shadow casting object. This makes tracing of shadow rays much faster. If it happens that during the evaluation of the ray history a proposed object is not hit, we continue at this point with ray tracing rather than starting the whole procedure from the primary ray anew.

As render platform we used an IRIS Indigo with 48 MByte memory and MIPS R3000 micro processor (33 MHz). The program is written in C with standard compiler, double precision floating point arithmetic and best optimization level.

As examples we will give results for four films, in which the camera is moving. So every object may change its position in the image plane. All frames are rendered with a resolution of 512×576 pixels. Figures 5, 6 and 7 show some frames of the films. The first one is a *pyramid* of metallic spheres, the second one is a *building* converted from a CAD–program (refers to building2 in table 1) and the third one is a set of *balls* in front of some mirrors.

Table 1 summarizes some data of the different films with $k = 4$. The number of rays $\#\text{rays}_{std}$ is the amount of rays cast in a fully ray traced sequence (primary rays, reflected rays, refracted rays and shadow rays). The number of rays $\#\text{rays}_{new}$ is the amount of rays really traced in our algorithm. The time T_{std} is the standard ray tracing time to render 4 frames, where preprocessing is counted only once. The time T_{new} is the time we measured with our method for the sequence of 4 frames. One can see that speedups between 1.2 and 2.7 have been achieved. A detailed analysis can be found in [16]. In figure 8 black pixels illustrate which pixels really had to be ray traced in the intermediate 3 frames of the scene *balls*. The white pixels indicate that those parts of the frames could be calculated by evaluating the ray history (fast ray tracing).

In our experiments with oversampling we have found out that the speedup obtained in lower resolution is almost maintained in the higher resolution: oversampling increases the number of pixels mostly in the black regiones (see figure 8), but it seems that ray–history evaluation can be used for the newly introduced pixels with the same ratio as in the previous, non–oversampling case.

6 Limits and Extensions

As mentioned above our algorithm performs undersampling in time. So it is possible that small or fast

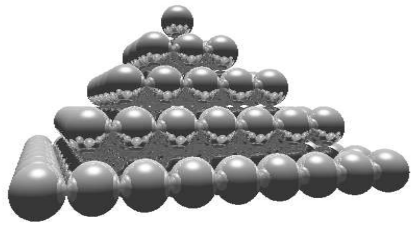

Figure 5: A pyramide of metallic spheres

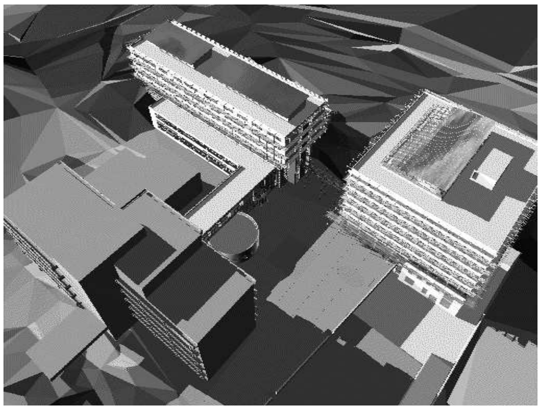

Figure 6: A building from a CAD–program

moving objects are rendered incompletely as shown in figure 9. The ray at the beginning of a sequence does not hit the small object and when the ray is cast at the end of the sequence the object has already passed the pixel.

If we keep k small this effect is not too disturbing; at least if the objects are not very small. But there are other possibilities to reduce this disadvantage of undersampling. The first improvement is using spatial coherence in the image plane within one frame. If we consider not only one pixel at any point of time but its neighbors as well, the probability that we don't hit the object with any ray in the region becomes smaller. Moreover it seems possible to combine the undersampling method of Akimoto ([2]) with our approach. The easiest way is shown in figure 10. We do not trace an entire frame at the beginning of a sequence. The really ray traced pixels are distributed in the time interval of the sequence. In the given example the same number of pixels is traced, but now more ray histories are available to detect a small or fast moving object. It might be possible to reduce the number of pixels, which are ray traced at the beginning, so we would reduce the run time too.

Another method uses the information about the

	pyramide	building1	building2	balls
#objects	165	5444	30621	26
#lights	1	1	2	1
#rays$_{std}$	2123	1472	2823	6648
#rays$_{new}$	1101	469	1646	3633
T_{std}	632	4081	38029	1387
T_{new}	506	1466	22045	1044
speedup	1.25	2.78	1.72	1.32

Table 1: Experimental results (rays in thousands and times in seconds)

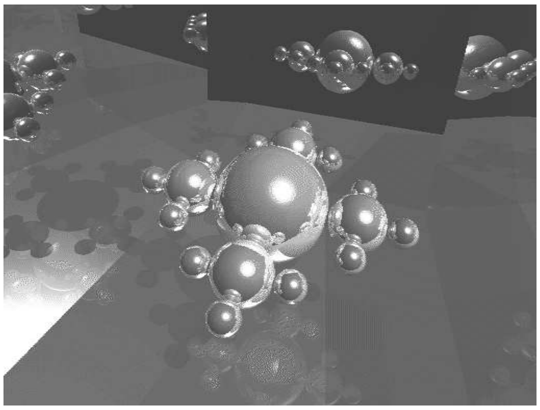

Figure 7: A set of balls in front of mirrors

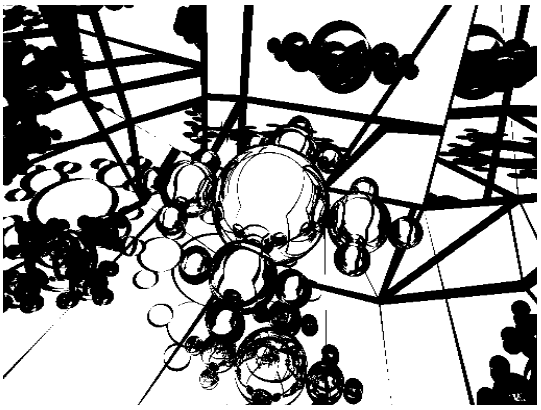

Figure 8: Really traced pixels in intermediate frames ($k = 4$)

movement of an object. If a ray hits an object, the angular velocity of the intersection point in respect to the origin of the ray is calculated. If the velocity exceeds a particular threshold oversampling in that region can be used to reduce aliasing or flickering of objects which have small spatial angles. The flood-fill algorithm introduced by Badt in [5] seems to be appropriate to determine the extent of such a region.

A time comsuming operation of our algorithm is the comparison of two ray histories. A speedup is only gained if the histories are equal. Thus the entire information in the trees must be compared. In order to reduce this time a better encoding might be possible. So far, we used a pointer to the object, a pointer to the material and a transformation matrix, which forms a block of 72 Bytes per node. If the block is compressed or encoded, the compare operation can be made faster, e. g. a hashing scheme as described in [21] is applyable; we would not use the voxel traversal history but a traversal history according to the hierarchy. Errors in the comparison can be tolerated: if the ray history does not lead to an intersection point during fast ray tracing the algorithm switches over to real ray tracing and if two trees are not detected as being equal only run time is wasted.

7 Conclusion

We presented an algorithm, which accelerates rendering of animated scenes. As rendering technique ray tracing was used. The method exploits both temporal and spatial coherence which normally can be found in consecutive frames of a sequence. As in the work of Akimoto et al., who introduced pixel selected ray tracing, it is an adaptive undersampling method with some restrictions in its appliability to general scenes. However, aliasing and flickering can be reduced, even more than in the case of single frame tracing. We obtained a speedup in the examples of about two, although in our test cases camera movements have been used, which is the worst possible scenario for our method. The algorithm is suited to be built in a lot of current ray tracing implementations. With oversampling in space and undersampling in time animated sequences of reasonable quality have been rendered. In scenes with a

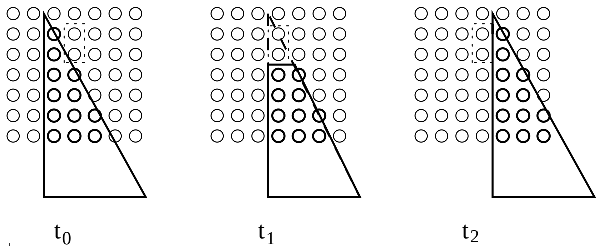

Figure 9: Disadvantage of undersampling: small or fast moving objects are missed

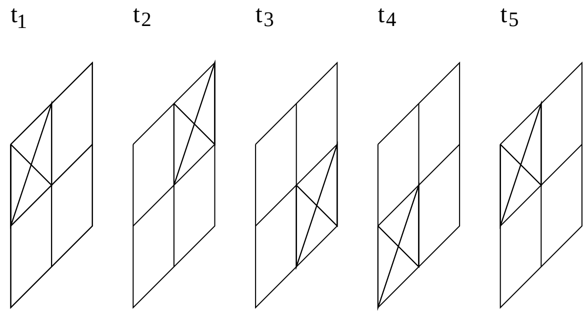

Figure 10: Combining spatial and temporal undersampling

moderate change of the geometrical behavior of the rays, we expect a higher speedup. So the algorithm can be seen as a small step towards ray traced films.

References

[1] ACM. Online Computer Graphics Bibliography, September 1993.

[2] Taka-aki Akimoto, Kenji Mase, Akihiko Hashimoto, and Yasuhito Suenaga. Pixel Selected Ray Tracing. In W. Hansmann, F. R. A. Hopgood, and W. Strasser, editors, *Eurographics '89*, pages 39–50. North-Holland, September 1989.

[3] Arthur Appel. Some Techniques for Shading Machine Renderings of Solids. In *AFIPS 1968 Spring Joint Computer Conf.*, volume 32, pages 37–45, 1968.

[4] James Arvo and David B. Kirk. Fast Ray Tracing by Ray Classification. In Maureen C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 55–64, July 1987.

[5] Sig Badt, Jr. Two Algorithms for Taking Advantage of Temporal Coherence in Ray Tracing. *The Visual Computer*, 4(3):123–132, September 1988.

[6] J. Chapman, T. W. Calvert, and J. Dill. Spatio–Temporal Coherence in Ray Tracing. In *Proceedings of Graphics Interface '91*, pages 101–108, June 1991.

[7] Arno Formella, Christian Gill, and Dirk Owerfeldt. A Hierarchical, Automatically Generated Space Subdivision for Ray Tracing with Fast Bounding Volume Search. In *to be published*, 1994.

[8] Akira Fujimoto and Kansei Iwata. Accelerated Ray Tracing. In Tosiyasu Kunii, editor, *Computer Graphics: Visual Technology and Art (Proceedings of Computer Graphics Tokyo '85)*, pages 41–65, New York, 1985. Springer Verlag.

[9] Christian Gill. Implementierung von parallelem Ray Tracing auf DATIS–P–32. Master's thesis, University of the Saarland, Department of Computer Science, April 1992.

[10] Andrew S. Glassner. Space Subdivision for Fast Ray Tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, October 1984.

[11] Andrew S. Glassner. Spacetime Ray Tracing for Animation. *IEEE Computer Graphics and Applications*, 8(2):60–70, March 1988.

[12] E. Groller and W. Purgathofer. Using Temporal and Spatial Coherence for Accelerating the Calculation of Animation Sequences. In Werner Purgathofer, editor, *Eurographics '91*, pages 103–113. North-Holland, September 1991.

[13] E. A. Haines and D. P. Greenberg. The Light Buffer: a Shadow Testing Accelerator. *IEEE Computer Graphics and Applications*, 6(9):6–16, 1986.

[14] Pat Hanrahan. Using Caching and Breadth–First Search to Speed Up Ray Tracing. In *Proceedings of Graphics Interface '86*, pages 56–61, Toronto, Ontario, May 1986. Canadian Information Processing Society.

[15] Paul S. Heckbert and Pat Hanrahan. Beam Tracing Polygonal Objects. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 119–127, July 1984.

[16] Volker Hofmeyer. Schnelles Ray Tracing von Bild–Sequencen. Master's thesis, University of the Saarland, Department of Computer Science, Februar 1994.

[17] Kenneth I. Joy and Murthy N. Bhetanabhotla. Ray Tracing Parametric Surface Patches Utilizing Numerical Techniques and Ray Coherence. In David C. Evans and Russell J. Athay, editors, *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 279–285, August 1986.

[18] James T. Kajiya. Ray Tracing Parametric Patches. In *Computer Graphics (SIGGRAPH '82 Proceedings)*, volume 16, pages 245–254, July 1982.

[19] Daniel Lischinski and Jakob Gonczarowski. Improved Techniques for Ray Tracing Parametric Surfaces. *The Visual Computer*, 6(3):134–52, June 1990.

[20] Joseph Marks, Robert Walsh, Jon Christensen, and Mark Friedell. Image and Intervisibility Coherence in Rendering. In *Proceedings of Graphics Interface '90*, pages 17–30, May 1990.

[21] Koichi Murakami and Katsuhiko Hirota. Incremental Ray Tracing. In *Eurographics Workshop on Photosimulation, Realism and Physics in Computer Graphics*, pages 15–29, Rennes, France, June 1990.

[22] Andrew Pearce and David Jevans. Exploiting Shadow Coherence in Ray Tracing. In *Proceedings of Graphics Interface '91*, pages 109–116, June 1991.

[23] L. R. Speer, T. D. Derose, and B. A. Barsky. A Theoretical and Empirical Analysis of Coherent Ray Tracing. In M. Wein and E. M. Kidd, editors, *Graphics Interface '85 Proceedings*, pages 1–8. Canadian Inf. Process. Soc., 1985.

[24] L. Richard Speer. An Updated Cross–Indexed Guide to the Ray Tracing Literature. *Computer Graphics*, 26(1):41–72, January 1992.

[25] Turner Whitted. An Improved Illumination Model for Shaded Display. *Communications of the ACM*, 23(6):343–349, June 1980.