Fast Adaptive Previewing by Ray Tracing

Günther Raidl, Wilhelm Barth

Department of Computer Graphics, Vienna University of Technology, Karlsplatz 13/1861, 1040 Vienna, Austria

Abstract. This article shows that the same ray tracing program can be used for performing a fast previewing during scene composition as well as for calculating the final antialiased high quality image. The described algorithm controls a strong undersampling so that a rough approximation can be shown within a few seconds. Then this approximating image will be adaptively refined improving its quality continuously until the final error-free antialiased image is reached. Small changes in the scene will be rendered quite fast because our algorithm additionally exploits similarities to a preceding image. It is based on a recursive image subdivision algorithm which uses a sophisticated heuristic priority calculation to detect nonhomogenous or changed regions and to prefer them in adaptive refinement. Other regions are preliminary treated by interpolation or copied from the preceding image until the priority schedule opens them for more accurate calculation. Our algorithm is very efficient concerning memory and time overhead.

1 Introduction

The normal way for generating realistic images by ray tracing is as follows: First the scene must be built up by a human, and after that ray tracing, which is very time expensive and may need several hours for difficult scenes, can be started to render the image. The process of building a scene consists mainly of arranging objects, defining their material, setting up light sources and defining the camera position. It is desirable to get a sketch of the picture after each step of this process. Usually, for this previewing a separated graphical tool is used which shows the scene from various viewing points in real-time. Because of the demands on speed only a simple rendering technique like displaying only polygon edges or doing flat shading can be used. But in such a sketch various essential properties of the image to be generated are lost: special light sources, mirroring or transparent surfaces, shadows, etc. Often, ray tracing generates therefore an image surprisingly different from the previewed one. Changes in the scene are necessary, and ray tracing has to be done again. For complicated constructions this process has to be repeated several times before the result is satisfactory.

We implemented a method for shortening this time consuming process. The basic idea is to use an algorithm which controls a combination of undersampling of normal ray tracing, interpolation, and copying of less critical parts. This rendering technique runs in background and shows permanently a preliminary image of increasing quality during scene composition.

2 Desired Characteristics

For optimal practical usage, we have defined the following necessary characteristics for our approach:

Table 1. Supported characteristics of previous adaptive ray tracing approaches

Method	Continuous improvement	Antialiasing	Exploitation of sim. prec. images	Overhead (time + memory)
Akimoto et al. [1]	no	no	no	low
Jansen, Wijk [5]	no	no	no	very low
Levoy [8]	no	no	no	very low
Painter, Sloan [9]	yes	yes	no	high
Schlick [11]	no	yes	no	low

a) Continuous improvement up to error-freeness: An approximating image of low quality should be available within seconds. This image should then continuously be refined and quality should be improved, until guaranteed error-freeness compared to traditional ray tracing is reached. Areas with more conspicuous errors should be refined before others.

b) Antialiasing: There should be some mechanism to reduce aliasing artifacts at pixel level even in earlier phases of previewing.

c) Exploitation of similar preceding images: In case of small scene changes the previewing mechanism should not restart from scratch but should recycle presumable unchanged parts of a previously rendered image: *Changing areas* should be determined and calculated anew using interpolation preliminarily, while unchanged parts are copied from the preceding image.

d) Small overhead: The mechanism should cause only a small time and memory overhead compared to normal ray tracing for rendering the final error-free image.

3 Previous Approaches

Rendering with adaptive refinement has been already investigated by other researchers but mainly based on other techniques than normal ray tracing or for very specific tasks (e.g. [3, 6, 7]). But there are also some previous approaches to adaptive refinement based on controlling sampling of ray tracing [1, 5, 8, 9, 11]. They all use a recursive image subdivision algorithm: The whole image is partitioned into several regions. In each region some samples are calculated and the color values of the pixels are chosen by interpolation and filtering over the sampled points utilizing area coherence. The regions will be subdivided recursively – those which contain sharp color contrasts prior to the others. The essential differences of these techniques lie primary in the way of sampling (corner, midpoint or stochastic sampling) and in the decision strategy of selecting regions for further refinement.

Unfortunately none of the known methods fulfills all desired characteristics for our application, see Table 1 and a detailed discussion in [10]. Furthermore it was not possible to extend one of the known techniques in a straight forward way to get a well suited method for our case. The algorithm we developed is also based on recursive region subdivision, but eliminates problems of previous approaches through a fast, heuristic priority calculation including a special mechanism called *neighbour influence* for drastically reducing aliasing problems as they occur in [5, 8]. Our algorithm needs much less memory than [9], and time overhead is also much lower.

Additionally the idea of exploiting similarities to a previous image seems to be new in this context. Eventually it may be compared with the exploitation of temporal coherence in generating frames for an animation sequence, see e.g. [2, 4].

4 Description of Algorithm

4.1 Algorithm

Let us assume the actual image to be calculated measures $x \times y = 2^n \times 2^n$ pixels.[1] The recursive image subdivision algorithm proceeds basically as follows:

Initially the whole image is the *selected region*;
Calculate the colors at the four corner points using normal ray tracing;
repeat
  Divide the selected region into four square subregions of equal size;
  Calculate colors at the new corners;
  Calculate priority values for new subregions and store them in a quadtree;
  Update pixel colors inside the new subregions;
  Search undivided region with highest priority – this is the new selected region;
until optimum quality has been reached **or** scene has been changed;

The crucial part of the algorithm is the selection of the next region for subdivision: Each region possesses a *priority value* which is a heuristical measure of the urgency for calculating pixel colors inside this region more precisely. The selected region is chosen from all those not yet further subdivided as that with the highest priority value.

Our algorithm always calculates an actual image using its predecessor which often is very similar. After finishing an actual image it becomes predecessor for the next step. Its corner color array may be incomplete because of yet undivided regions and has to be completed in some way, e.g. by interpolation or copying from the predecessor (see Sec. 4.2). The first image may be taken as black or background.

4.2 Coloring the pixels

We use the term *changing region* for a region that contains already detected changes against the predecessor, i.e. at least one corner point has a different color compared to the preceding image.

Pixel colors inside a new region are determined in the following way: For a nonchanging region, we assume preliminarily that there is no change in this region. Therefore, all pixel colors are copied from the corresponding part of the preceding image. Pixel colors of a changing region are determined by bilinear interpolation between the four corner colors.

To avoid unnecessary copying or interpolation, the following cases should be checked: If no changes have been detected in both, the actual region and its parent, the pixels of the actual region have got already their designated colors. Similar, if there were changes in the actual region and also in its parent and all corner colors are equal, nothing need to be done.

---

[1] $x \times y = 2^n \times 2^n$ is not a fundamental restriction of our method. An efficient way to render images of any size is discussed in [10].

4.3 Priority calculation

The crucial part of the algorithm is priority calculation. As previously outlined, the priority value should be an approximative measure for the error made probably by copying or interpolation of pixel colors inside a region. Effort has to be invested in finding a convenient priority function because it has substantial influence on the speed of quality improvement of images. There are various features to consider which result in different components of the priority function described below. The size of a region, the color changes of its corner points, and the contrast of color inside have the largest influence on its priority. But also changes in neighbouring regions may cause the urgency for further subdivisions. We define the priority value of a region as the sum of four components according to four goals:

$$P = P_A + P_C + P_D + P_B \tag{1}$$

Large regions first – area priority P_A:

In general, a large region should be subdivided before a smaller one. Therefore, large regions must have higher priority values. We calculate the area priority value as follows (A is the area of the region which measures $2^i \times 2^i$ pixels):

$$P_A = \log_4 A = i \tag{2}$$

Thus, when a selected region is subdivided, the new subregions will get the area priority of the parent decreased by one.

Changing regions first – change priority P_C:

The change priority ensures that changing regions are preferred against nonchanging ones of same size. The intensity of change and the number of changed corner colors do not influence P_C. Because of color interpolation we assume equal urgency for all changing regions. Therefore, P_C is either a constant ch or 0:

$$\textbf{if } \text{changing region } \textbf{then } P_C = ch \textbf{ else } P_C = 0; \tag{3}$$

Changing regions with high contrasts first – contrast priority P_D:

Important for the priority of a changing region is not the amount of change between actual and previous image but the contrast inside the region of the new image because right in the case of large differences between corner colors, interpolation may cause substantial errors. Therefore, we add for all changing regions the contrast priority P_D which is calculated in the following way:

First, we define the contrast of two colors a and b as

$$D_{a,b} \leftarrow \sqrt{\left(\Delta r_{a,b}^2 + \Delta g_{a,b}^2 + \Delta b_{a,b}^2\right)/3} \tag{4}$$

($\Delta r_{a,b}$, $\Delta g_{a,b}$, and $\Delta b_{a,b}$ are the differences of the red, green and blue components of the colors a and b), a value between 0 and 1. Note that this measure of contrast is only a crude approximation to real perceptual contrast, but it can be determined very quickly and seems to be precise enough for our application.

Then, the region contrast D is calculated from the corner colors $C1$ to $C4$:

$$D = \max(D_{C1,C2}, D_{C3,C4}, D_{C1,C3}, D_{C2,C4}) \tag{5}$$

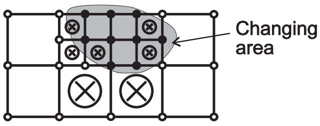

Fig. 1. Obvious border regions

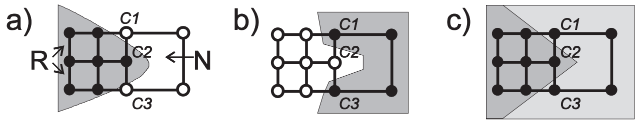

Fig. 2. Three cases where neighbour influence will be activated

Now we get the contrast priority (d_{max} is a constant factor which controls the influence of D):

$$\textbf{if} \text{ changing region } \textbf{then } P_D = D \cdot d_{max} \textbf{ else } P_D = 0; \tag{6}$$

Border regions first – border priority P_B:

The priority function should contain another term which forces finer subdivision of regions at borders of changing areas because they are more critical concerning errors made by interpolation. We define *border regions* as such containing obviously a part of the border of a changing area. They reveal this feature by: at least one corner color is changed and at least another is unchanged (see Fig. 1). These regions get a constant increase cb to priority:

$$\textbf{if} \text{ border region } \textbf{then } P_B = cb \textbf{ else } P_B = 0; \tag{7}$$

Note that not all regions containing a part of a changing area or high color contrast can be discovered by the conditions described above. Therefore we try to find more such critical regions by regarding *neighbour influence*:

Tests with the priority function as explained up to now have shown bad performance because of the following problem: Often, a changing area of a region R enters a neighbour region N over an edge without hitting any of its corners (see top-right region in Fig. 1). In such a case, the region N will get a very low priority value and its further subdivision will be postponed. Even if the changing area is discovered in R, this cannot help because subdivision of all regions works independently. Neighbour influence has to be considered with priority calculation to solve this and some similar problems. When a selected region R is partitioned, we have to examine whether a neighbour region is likely to be influenced.

Therefore, for each edge of a selected region R leading to a neighbour region N of equal or larger size check the following three conditions which may give a hint that the priority value of N got by isolated examination of its corner points is probably too low (see also Fig. 2):

a) No changes in $C1$ and $C3$, yet the color in $C2$ is different from the predecessor color: a changing area of R enters N possibly undetected (Fig. 2a).

b) Changes in $C1$ and $C3$, yet the color in $C2$ is unchanged: N contains an already discovered changing area, but the fact that an unchanged part reaches from R into N possibly disturbing color interpolation has not been detected (Fig. 2b).

c) Changes in $C1$, $C2$ and $C3$ and the mean value of the colors at $C1$ and $C3$ differs substantially from the color at $C2$: pixels around $C2$ cannot be interpolated well, because an undetected part of different color enters N in the actual image (Fig. 2c). This is often the case when the scene contains thin or wedge shaped objects.

If at least one of these conditions is fulfilled, the following actions take place: The region N is repeatedly subdivided until it reaches the size of R. Note that such a recursive subdivision can trigger further recursive subdivisions elsewhere. A neighbour region of equal size, maybe generated by the previous action, gets a new priority which is the maximum of the old one and a value calculated by a similar priority function as described before using only the three corner points $C1$, $C2$ and $C3$. Priority values may therefore be increased but never decreased by considering neighbour influence.

Note that even when using neighbour influence local discontinuities (e.g. specular highlights or very small objects) sometimes cannot be detected from the sampled colors. Therefore such a region may be subdivided later than desired. Including object space information in priority calculation would improve the possibility of detection but also slow down the calculation and increase memory requirements dramatically (compare [1]).

4.4 Antialiasing

We propose to do antialiasing by adaptive supersampling in a less or more traditional way: Region subdivision of our algorithm proceeds until pixel level is reached. When a new region has the size of one pixel, the following will be done: If there was no change detected at its four corners, the color of the pixel is copied from the preceding image. Otherwise the color is determined in one immediate step using traditional adaptive supersampling down to a given maximum depth d. For better performance, the calculated points at the pixel border should be stored in a hash table, so they can be reused when adaptive supersampling takes place for a neighbouring pixel.

Another way to do antialiasing by adaptive supersampling would be to continue our algorithm down under the pixel level similar as it is done in [9]. But we cannot recommend this approach due to much higher memory effort and general performance decrease for rendering final error-free images.

4.5 Choosing the constants

There is no general method to get "good" values for the constants ch, d_{max} and cb of the priority function. The optimal values of these constants depend on type and structure of the image sequence treated. But we will summarize some results got from various tests. We calculated images with less or more details, with many small or only a few large changing areas. A more detailed description of these tests can be found in [10]. We have used the following error measure to rate the quality of an intermediate image A in comparison to an error-free image I calculated with normal ray tracing ($D_{a,b}$ is defined as in Eq. 4):

$$Err = \sum_{j=1}^{y} \sum_{i=1}^{x} D^2_{A[i,j],I[i,j]} \tag{8}$$

Experiences show that the constant ch – it is responsible for preferring changing regions – should have a value between 0.5 and 3. If ch is too large, changing regions once detected will quickly be refined to a high level of detail. But this happens at the expense of larger regions containing changing areas not yet hit because their subdivision will be postponed. An appropriate value for ch depends on the number and size of changing areas: If there are many small changing areas, a lower value will give better results, because most of the changing areas will be detected earlier. Images which contain only a few large changing areas may be approximated faster with larger values. For images with unknown changing area properties, a value of 1 is proposed.

The constant d_{max} for preferring regions of high contrasts is less critical than ch. Our tests have shown that it does not depend very strongly on the properties of changing areas or the complexity of the image. We made good experiences with values between 3.5 and 5 and propose to take 4.

The constant cb for preferring border regions is also not very critical. Values between 0.01 and 0.1 seem to be sufficient for realizing the neighbour influence effect. We propose to take 0.05.

4.6 Data structures

An efficient data structure for storing all priority values is a quadtree. It represents the partitioning of the whole image in a very natural way: A leaf node corresponds with a region not yet subdivided and contains its priority value. Each inner node represents an already subdivided node and stores the highest priority value of its four subnodes (which may be leaf nodes or inner nodes). The priority value of the root node is the maximum of the priority values of all regions and therefore also a heuristical measure of the necessity that the whole image should be further refined. In other words, the root priority estimates the error of the actual image in some way. If it reaches zero, all regions are refined as far as possible and optimum quality has been reached.

Searching the region with highest priority can be done very efficiently by starting from the root node and always going down to the subnode with the highest priority value until a leaf node is reached. When priority values of two or more subnodes are equal, one of them may be chosen at random. After partitioning the selected region, priority values are calculated for the four new subregions. Preceding inner nodes need to be updated to hold again the maximum priority values of their subnodes.

Color values at the region corners are shared by up to four regions and therefore not stored in the quadtree but in an own array of size $(x + 1) \times (y + 1)$. Note that these values are the exact colors at pixel corners and in general are not identical with pixel colors. This distinction is necessary for the described antialising mechanism. To avoid multiple calculation of corner colors, a corresponding bitmap of the same size is used in which the already known colors are marked. We further require the previously calculated image and its corner color array for exploiting similarities.

5 Implementation and Results

The algorithm has been implemented on a 486 PC using the ray tracing system FLIRT, which has been developed at our department and uses state-of-the-art optimization techniques. The scenes of the test sequence shown in Figs. 3–6 (see Appendix) contain only primitive objects (spheres, boxes, cylinders) combined with CSG operators. All

Table 2. Performance statistics of rendering Figs. 3–6 (see Appendix)

Image (Fig.)	Prec. Img. (Fig.)	Total time t (s)	Calcul. points	P_{max}	Err	t_{nr} (s)	t/t_{nr} (%)
3a	—	7	2503	6.26	136.05	291	2.4
3b	—	36	13433	4.50	41.18		12.4
3c	—	150	50803	3.15	4.31		51.5
3d	—	322	118917	0.00	0.00		110.6
4a	3c	2	601	5.45	9.21	283	0.7
4b	3c	11	3528	3.25	4.28		3.9
4c	3c	320	115841	0.00	0.00		113.1
5a	4b	3	1017	5.80	55.09	270	1.1
5b	4b	33	11506	3.83	5.19		12.2
5c	4b	307	113025	0.00	0.00		113.7
6a	5b	2	600	5.60	36.95	294	0.7
6b	5b	11	6237	3.99	6.31		3.7
6c	5b	334	118649	0.00	0.00		113.6

P_{max}, maximal priority value; Err, error of approximation as defined in Eq. 8; t_{nr}, total time for conventional ray tracing

images measure 200 × 250 pixels. Antialiasing is done up to a depth of one. The constants of the priority function are chosen as suggested above ($ch = 1$, $d_{max} = 4$, $cb = 0.05$).

In Fig. 3 you can observe how our algorithm generates approximations to a first image starting with a black predecessor. The four snapshots are taken after 7, 36, 150 and 322 seconds CPU time. Fig. 3d is the final error-free and antialiased image – the same that will be generated by conventional ray tracing after 291 seconds. As you can see, the time overhead for using our algorithm instead of conventional ray tracing for calculating the final image is quite low (10.6%). Note also that already Fig. 3c is visually nearly identical with Fig. 3d. In most cases error-freeness is reached much earlier than in the final image, but it can't be guaranteed.

Fig. 4 shows how the algorithm copes with small changes in the scene – especially with a modification of the top part of the rook. Note that the exploitation of similarities to the preceding image speeds up quality improvement considerable. In the next step the rook changed its material (Fig. 5). At last a pawn was added to the scene (Fig. 6). Detailed information about the rendering can be found in Table 2. Note further, that our implementation required never more than 2 MB of data memory for calculating the images shown in this article.

Contrary to the algorithm described in [9], our sampling is done at the corners of regions. This kind of sampling is surely more vulnerable to aliasing artifacts than stochastic sampling. But corner sampling has substantial advantages in conjunction with our algorithm: First, the sampling position need not to be stored. Second, the corner points are shared by up to four regions. Every region uses therefore four corner points instead of only one sample as in [9]. A much simpler and faster priority calculation is the consequence. The third advantage is the possibility to do efficient bilinear color interpolation. Aliasing artifacts above pixel level, are considerably reduced by the neighbour influence mechanism. Bilinear color interpolation seems to be an adequate technique for our algorithm. It preserves continuity between equally sized regions. More sophisticated techniques, which also maintain continuity between regions of different size or higher order continuity, seem to be too time expensive.

6 Conclusion

The algorithm introduced in this paper proves particularly useful as previewing mechanism during scene construction. Controlled by our algorithm, basic ray tracing can be used to generate a rough approximation for an image within seconds. Image quality is then improved continuously until the final error-free and antialiased image is reached with only a small time overhead compared to conventional ray tracing. Although the algorithm needs some data structures, memory overhead is not very large. The crucial part of our algorithm is the suitable heuristic priority calculation which includes the powerful neighbour influence mechanism and also exploits similarities to preceding images.

Acknowledgments

This work was sponsored by the Austrian *Hochschuljubiläumsstiftung der Stadt Wien.*

References

[1] Akimoto T., Mase K., Suenaga Y. (1991) *Pixel-selected ray tracing.* IEEE Computer Graphics & Applications 11:14–22

[2] Badt S. (1988) *Two algorithms for taking advantage of temporal coherence in ray tracing.* The Visual Computer 4:123–132

[3] Bergman L., Fuchs H., Grant E. (1986) *Image rendering by adaptive refinement.* Computer Graphics 20:29–37

[4] Chapman J., Calvert T.W. (1990) *Exploiting temporal coherence in ray tracing.* Graphics Interface '90 196–204

[5] Jansen F.W., Wijk J.J. (1983) *Fast previewing techniques in raster graphics.* Eurographics '83 195–202

[6] Ke H.R. (1993) *Sample buffer: a progressive refinement ray-casting algorithm for volume rendering.* Computers & Graphics 17:277–283

[7] Laur D., Hanrahan P. (1991) *Hierarchical splatting: a progressive refinement algorithm for volume rendering.* Computer Graphics 25:285–288

[8] Levoy M. (1990) *Volume rendering by adaptive refinement.* The Visual Computer 6:2–7

[9] Painter J., Sloan K. (1989) *Antialiased ray tracing by adaptive progressive refinement.* Computer Graphics 23:281–288

[10] Raidl G. (1994) *Progressives, adaptives Ray-Tracing von Bildern mit hoher zeitlicher Kohärenz.* Ph.D. thesis at the Vienna University of Technology, Department of Computer Graphics

[11] Schlick C. (1991) *An adaptive sampling technique for multidimensional integration by ray-tracing.* Eurographics workshop on rendering 1–10