Ray Path Categorization

DIEGO NEHAB MARCELO GATTASS

TeCGraf: Grupo de Tecnologia em Computação Gráfica, PUC-Rio,
R. Marquês de S. Vicente 225, Gávea, Rio de Janeiro, 22453-900, RJ, Brasil
{diego,gattass}@tecgraf.puc-rio.br

Abstract. Edge detection and image segmentation algorithms usually operate on an image to extract geometrical information based on pixel colors. For ray-traced images, the presence of geometrical information on the scene from which the image was rendered allows for a completely different approach. We present an algorithm that divides rays into equivalence classes, or categories. The category information is generated during the rendering process and used to determine edges in the resulting image. Detected edges can later be used to help determine areas subject to aliasing. Little effort is needed to implement the described algorithms over an existing ray-tracer. Furthermore, the extra computational and memory requirements are modest.

1 Introduction

Segmentation algorithms are used to generate a set of regions representing parts or objects in an image. Edge detection algorithms try to determine the pixels that fall on the boundaries between two regions. Segmentation and edge detection are usually essential preliminary steps in a vast number of image processing applications, such as pattern-recognition and scene-analysis.

Most edge detection algorithms apply some kind of local derivative operator over pixel intensities. Image segmentation methods usually threshold pixel intensities to divide them into similar groups. These tasks are difficult because the only information available for a given image is a matrix of pixels colors. Several approaches to solve both problems are discussed by Gonzalez [9].

A ray-tracer scene consists of the geometrical description of a set of objects. Object properties such as shape, material, texture, and position relative to some reference frame are given in this description. Finally, a camera facing the scene is defined, and the image as seen by this camera is generated [8].

Traditionally, rendering is done by the casting of one ray into the scene for each pixel of the image. Each ray starts on the camera and passes through its associated pixel on the camera's projection plane. The color of a pixel is determined by the path the ray follows as it intersects objects in the scene, recursively spawning reflection and refraction rays [15, 8].

In order to segment or detect edges in ray-traced images, we can use the geometrical information given by the scene description. In Section 2 we present four variations of a method that uses this information. Section 3 introduces the edge detection algorithm based on ray path categorization. As an application, in Section 4 we apply the idea in adaptive antialiasing of ray-traced images. Section 6 discusses further applications for ray path categorization.

2 Ray categories

A ray-tracer uses an illumination equation to combine scene information and determine the color of a pixel. Consider a modified version of the illumination equation presented by Whitted [15]:

$$\begin{aligned} I = O_e &+ O_a S_a \\ &+ \sum_i [O_d L_{d_i} (\hat{N} \cdot \hat{L}_i) + O_s L_{s_i} (\hat{N} \cdot \hat{H}_i)^n] \\ &+ k_s I_r + k_t I_t \end{aligned} \tag{1}$$

In this equation, the original material constants semantics have been changed to those present in the OpenGL model [16]. There, O_e, O_a, O_d and O_s represent the object's emitting, ambient, diffuse and specular colors respectively evaluated at the intersection. The summation is taken over every light source in the scene that is visible from the intersection point. L_{d_i} and L_{s_i} are the diffuse and specular colors of each light source. I_r is the color of the reflected ray and I_t is the color of the refracted ray, both being calculated by the recursive spawning of child rays.

By examining Equation 1, we see that the path followed by a ray can be described by the objects the ray hits, by the light sources that are visible at the intersection points and by the recursively generated reflection and refraction rays. Some examples of different ray paths are shown for the pixels a, b, and c of Figure 1.

We look for a way to partition these different paths into different equivalence classes. To avoid confusion with the concept introduced by Arvo [1], who classifies rays to reduce the number of intersection calculations, we will use the term *category* to refer to each equivalence class. The division of pixels into groups that have the same category is an image segmentation problem.

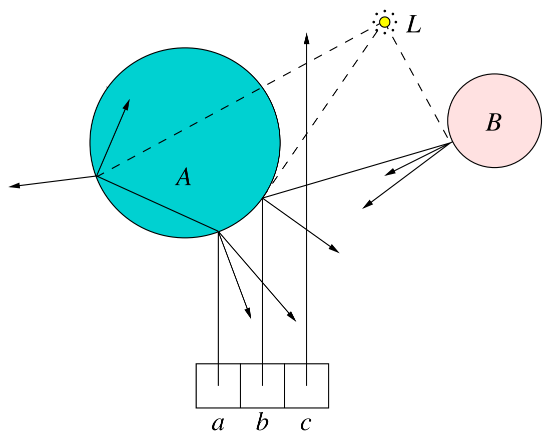


Figure 1: Paths followed by three adjacent rays *a*, *b*, and *c*, up to recursion depth 2 of their paths. Ray *a* hits sphere *A* where it reflects and refracts. Ray *b* also hits sphere *A*, but the angle does not permit refraction at that point. Its reflection, though, hits sphere *B*. Finally, ray *c* does not hit any object in the scene.

2.1 Describing paths

The ray associated to each pixel defines a tree while intersecting objects and spawning reflection and refraction child rays in the scene. Tree nodes represent intersections with objects. Left and right branches represent refraction and reflection child rays respectively. Figure 2 shows some examples of such trees. We say that two rays have the same category if the trees induced by them are equal.

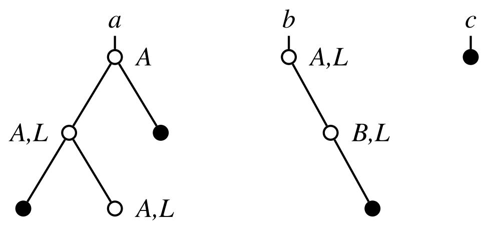


Figure 2: The trees induced by the rays *a*, *b*, and *c* of Figure 1. Each node is labelled by the id of the object hit and by the ids of the light sources that were visible at the intersection.

We assume that objects and each light sources in the scene have unique identification symbols, henceforth called *object ids*, or *ids* for short, just like the labels given for the spheres and the light source in Figure 1. Usually, all ids will be numbers, but the only requirement is that they be unique and that we can compare them for equality.

With this information available during the rendering process, there are several ways to generate a description of the trees seen in Figure 2. We propose and analyze four different approaches: the string method, the primes product method, the Gödel method and the binary heap method.

2.1.1 String method

The first idea that comes to mind is to concatenate all symbols in the tree, in depth-first search order, creating a string of ids. The implementation is simple because this is the natural order the intersections are found during the recursive ray-tracing process. The method would yield strings *ALALA*, *BLAL*, and λ (the empty string) for rays *a*, *b*, and *c* of Figure 2, respectively.

Since the recursion depth for a regular scene is usually small, tree sizes remain treatable. Nevertheless, we would gain in performance by working with more concise descriptions of trees: hash values would compare faster and take less storage space. The primes product and and Gödel methods follow this idea.

2.1.2 Primes product

Suppose the ids in Figure 1 are replaced by different numbers: say *A* is 2, *L* is 3, and *B* is 4. One could describe the trees in Figure 2 by multiplying all the ids in each tree. This would give values $2\cdot3\cdot2\cdot3\cdot2 = 72$ for tree *a*; $4\cdot3\cdot2\cdot3 = 72$ for tree *b*; and 1 for tree *c* (1 being defined as the value for the empty tree).

The results are not satisfactory because the categories for trees *a* and *b* collided even though the trees are different. The problem is that the numbers chosen for spheres *A* and *B* were not coprime: in fact, $A \cdot A = B$. The Fundamental Theorem of Arithmetic [2] states that every integer has a unique prime factorization. Therefore, we can solve this problem by restricting ids to the set of prime numbers. We will not run out of prime numbers because, by the Prime Number Theorem [2], there are more than 200,000,000 prime numbers less than 2^{32}. The primes can be precomputed or generated incrementally while the scene is being loaded by the ray-tracer. Generating primes by the Eratosthenes Sieve [2] should be fast enough for any ray-tracer application.

The new values for trees *a* and *b*, considering $A = 2$, $B = 3$, and $L = 5$ are 200 and 150, respectively. Since there will be usually one or more light sources visible from most object-ray intersections, light sources should have the lowest ids to avoid overflow problems.

Although the results obtained with the prime number method are quite satisfactory, it is easy to construct a scene for which the method will fail. Consider, for example, Figure 3. Since the product operation is commutative, different strings may map to the same number. That is, the primes product map from strings to natural numbers is not one-to-oner. Indeed, for the example in Figure 8, rendered as a 512×512 image, the primes product method fails for 26 pixels due to improper collisions between categories while detecting edges by the method explained in Section 3.

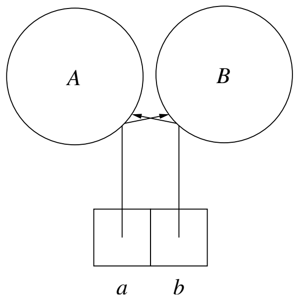


Figure 3: The two adjacent pixels a and b have string categories of *BA* and *AB* respectively, but the primes product categories are the same. The classification fails for this case.

2.1.3 Gödel numbering

A subtle change in representation leads to a new method that succeeds for Figure 3. Assume object ids are sequential natural numbers. A device called *Gödel numbering* [12] establishes a one-to-one relation between strings and natural numbers. Gödel used this mapping to prove his Incompleteness Theorem.

Let $a_1a_2a_3 \ldots a_n$ be the string of integer ids that we want to convert to a single number. The corresponding Gödel number is $p_1^{a_1} \cdot p_2^{a_2} \cdot p_3^{a_3} \cdots p_n^{a_n}$, where p_i is the i^{th} prime number. It is easy to see, again by the Fundamental Theorem of Arithmetic, that the mapping is indeed a bijection between the set of integer strings and the set of natural numbers. For the example of Figure 3, tree a would map to $2^B \cdot 3^A$, whereas tree b would be represented as $2^A \cdot 3^B$.

Although Gödel numbering succeeds on the 13 pixels for which primes product fails in Figure 8, it has a serious limitation: Gödel numbers tend to grow much faster than primes products. Consider a scene in which a ray spawns a full tree up to depth 4. The corresponding string would have more than 16 nodes if there are any visible light sources. If we take the first 16 prime numbers and multiply them together, the result is 32589158477190044730, which is a 65-bit integer. If we raise the prime numbers to their appropriate powers—the object ids—the result can easily overflow a 80-bit double precision floating-point number.

Just as an example, consider Figure 2. Recall that the computed primes product numbers were 200 and 150 for trees a and b, respectively. If we take $A = 1$, $B = 2$, and $L = 3$, the corresponding Gödel numbers are $2 \cdot 3^3 \cdot 5 \cdot 7^3 \cdot 11 = 1018710$ and $2^2 \cdot 3^3 \cdot 5 \cdot 7^3 = 185220$, respectively.

2.1.4 Binary heap

For scenes with no refraction, both the string and Gödel methods define one-to-one mappings from ray paths to categories. Unfortunately, when refraction is present, these methods can fail. Since not all nodes are present on the tree, a depth-first listing of the nodes is not enough to uniquely determine a tree. Again, improper collisions between categories of different trees can happen. This fact is better illustrated by the example in Figure 4.

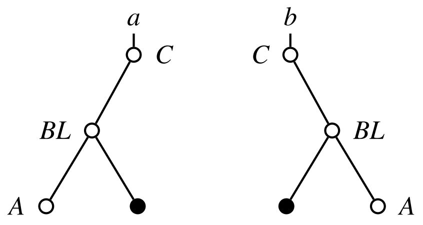


Figure 4: Trees a and b are different, but have the same depth-first order listing, *ABLC*, of their nodes.

We can solve this problem by assigning a unique number to each node position in a tree and storing this number, along with the collision information, in the string. To do so, we imagine that the tree is a complete binary tree and number the nodes with their binary heap ordering [4]. Figure 5 explains this idea.

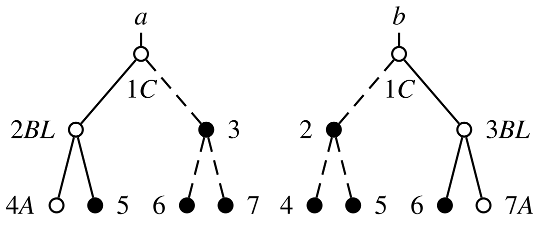


Figure 5: With the numbering of the node positions, trees a and b receive strings 4*A*2*BL*1*C* and 7*A*3*BL*1*C*, respectively. The new coding is a perfect hash.

Although the method is general as stated, we could save some memory coding each node as a 32-bit integer if we impose some restrictions. Usually, a ray-tracer will not go deeper than eight recursion levels so that the node number can be stored in 8 bits. Following the OpenGL illumination model, we can limit the maximum number of light sources to eight, so that we can also pack all light source visibility information into 8 bits. This leaves a 16-bit word to identify the object intersected, leading to a maximum of 65536 different objects in the scene. The new coding takes no more memory than the unrestricted version of the string method.

2.2 Method comparison

The primes product method is the simplest but is also the least precise. The results are very good for simple scenes. For complex scenes, however, the method can fail unpredictably. Nevertheless, the simplicity of its implementation and the reduced performance overhead may justify its use.

The Gödel numbering method is more of academic interest than of practical use. Even though for scenes with few objects the method can achieve the same results as the string method, categories are likely to overflow any machine size number for scenes with many objects. Any advantages over the primes product method will be lost due to overflow problems. The method is also somewhat harder to implement than the primes product, because child ray categories can not be directly merged (i.e. multiplied) to generate the parent's ray category.

For scenes not involving refraction, the string method can be used safely. In fact, for those scenes, all nodes in the recursion tree have degree one. Therefore, a depth-first order listing of the tree nodes is perfect at describing ray paths. For some scenes involving refraction, the string method can fail for a few points. It is hard to design a scene specifically to force the method to fail. To detect failures, the scene of Figure 9 was rendered from several different viewing angles and then compared with the results obtained by the binary heap method. For some of the viewing angles, failures were spotted.

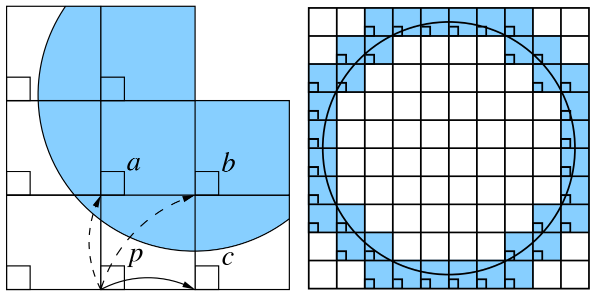
Table 1 shows that, besides being the only method to obtain correct results for all scenes, the performance of the binary heap method compares favorably with that of the other methods. Binary heap can be easily implemented as a variation of the string method, where the node number is added to the string whenever an intersection is detected. In the worst case, string lengths will only double. The numbering of the nodes can be generated and made available on-the-fly as an extra argument to the recursive ray casting function. For these reasons, the binary heap method should be preferred whenever precise results are required.

Method	Pixels selected		Rendering time (s)	
Figure	9	8	9	8
No categories	0	0	46.3	9.0
Primes	43280	23630	48.2	9.3
Gödel	43330	23656	97.5	48.7
Strings	43330	23656	50.6	11.4
Heap	43340	23656	51.7	11.9

Table 1: Comparison of the categorization methods. Time results for the Gödel method are for an unoptimized implementation over a modified version of the string method. Hence the bad performance.

3 Detecting edges

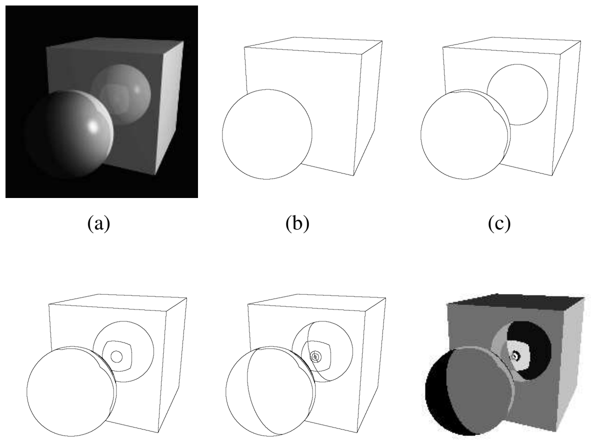
Ray path categories can be used to detect edges in a ray-traced image. Our edge detection algorithm uses some of the ideas that Whitted [15] developed for his adaptive antialiasing approach. We consider a pixel as representing the ray cast through its lower left corner. To determine the pixels that spawn edges, each pixel is compared against its three 4-neighbors (a, b, c for pixel p in Figure 6(a)). If any differences are detected, the pixel is selected as an edge pixel. If further precision is required, each pixel can be broken into four subpixels and the comparison process can continue recursively at subpixel level. Using this idea, the colored pixels shown in Figure 6(b) are exactly those over the borders of the object.



(a) (b)

Figure 6: (a) Edges are detected when a pixel's category is different from that of one of its neighbors. (b) Pixels selected as edges for a simple scene.

By choosing the information that gets registered in a category and the recursion depth up to which this information is registered, it is possible to choose the level of detail of the edge detection algorithm. Figure 7 shows a simple scene for which edge detection has been performed in several levels of detail.



(a) (b) (c)

(d) (e) (f)

Figure 7: Different levels of detail for edge detection. (a) Original; (b) Depth 1; (c) Depth 2; (d) Depth 4; (e) Depth 6 with light visibility; (f) Segmented at depth 6 with light visibility.

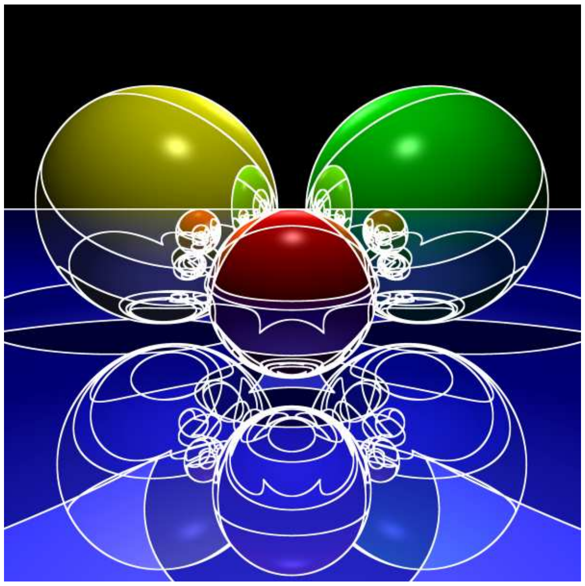


Figure 8: A highly reflective scene. Chosen edges have been enhanced over the rendered image

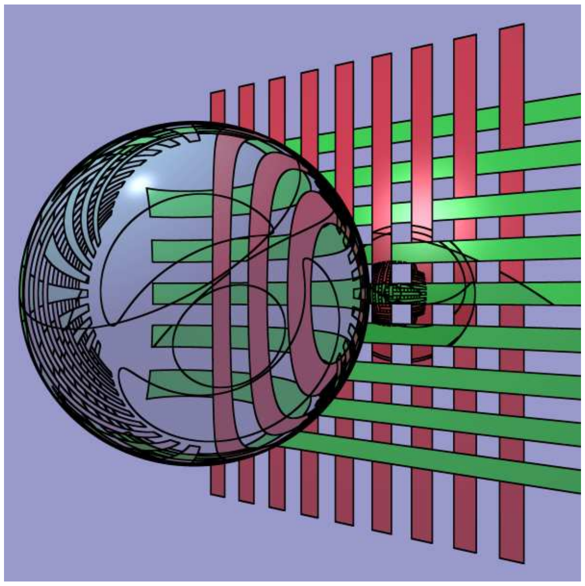


Figure 9: Refraction example. Chosen edges have been painted black over the rendered image

Edges have been marked over two test scenes. The binary heap method was used to generate category information for both images. Figure 8 shows a scene where highly reflective objects are present. The algorithm detects all edges within reflections and shadows. Figure 9 shows the results of the algorithm on a scene where a transparent sphere is present. The algorithm reveals structures created by the refraction and internal reflection inside the sphere.

3.1 Composite objects

Composite objects are objects containing more than one primitive surface. Polyhedra, cones, and cylinders are examples of composite objects. Constructive Solid Geometry (CSG) objects are composite objects defined by set operations over other objects.

Consider an intersection with a cube. If there is no distinction between cube faces, all that gets registered in a category is the cube's id and all methods will fail to detect edges between cube faces.

If, on the other hand, the ids reported at intersections are those of the primitive objects containing the surfaces hit, the methods presented here work for composite objects. Figure 10 shows the results obtained while detecting edges for the basic CSG operations.

4 Application to antialiasing

Aliasing arises because ray-tracing is a sampling process. The sampled image can only represent information up to a bounded frequency. Higher frequencies present in the scene appear falsely as low frequencies when regular sampling is used, creating annoying jagged edges and Moiré patterns.

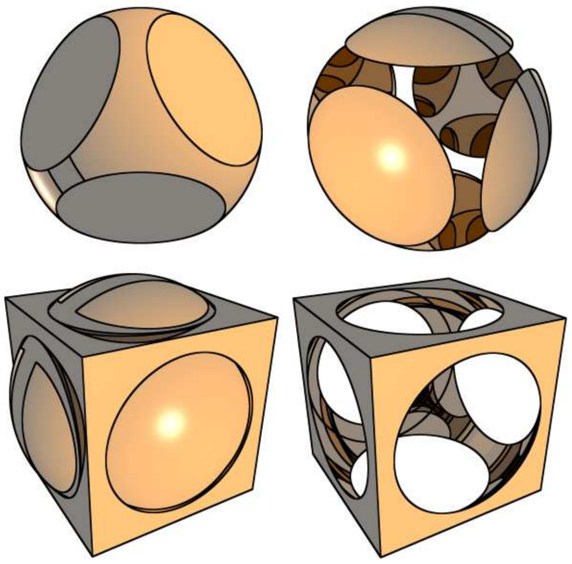


Figure 10: The set operations of union, intersection and subtraction over primitive objects sphere and box. Detected edges have been painted black over the rendered image.

The simplest way to improve image quality is to oversample the image. Figure 11 shows the result of the technique for a simple scene. Oversampling the whole scene is too expensive, so that several alternatives have been studied to generate high-quality images at low average sampling

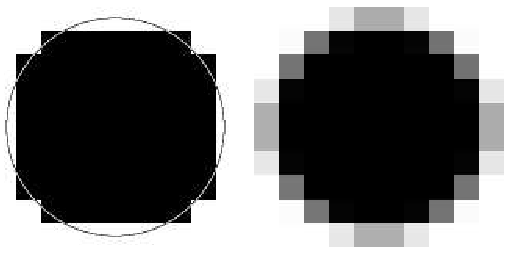


Figure 11: (a) A simple sphere, rendered at a low sampling rate. The circle shows a better approximation for the object's borders. (b) A 16×16 uniformly supersampled version of the same scene.

rates.

For some classes of models, prefiltering [5] can ensure that high frequencies in the scene do not exceed half the sampling rate, thus eliminating aliasing. It is not, however, a viable option for general scenes. Stochastic sampling methods [3, 7, 11] use nonuniform sampling to convert aliasing patterns into less perceptible noise patterns. Adaptive sampling methods [15] improve image quality by oversampling close to edges in the image. Some hybrid algorithms that use both adaptive and stochastic sampling attain even better results [14].

Adaptive algorithms are likely to differ in the way they detect edges. Most of them base this decision on a heuristic measure of color variation on a pixel neighborhood. Instead, we can expect that high frequencies in the scene will lead to different categories between adjacent pixels in the image. That is, we can choose to oversample only the pixels found to lie over detected edges.

4.1 Scalability

One of the advantages of using edge detection for antialiasing is that the larger the image dimensions, the better the algorithm performs, proportionally. This happens because, for a scene rendered as a $n \times n$ image, the number of pixels on the edges of objects and shadows is $O(n)$. Thus, the percentage of pixels that will be chosen for oversampling is $O(\frac{n}{n^2}) = O(\frac{1}{n})$. The graph in Figure 12 shows the ratio between the number of pixels selected and the total number of pixels for the scene in Figure 9 as a function of the image resolution.

4.2 Textures

Our implementation of the methods presented here do not detect aliasing caused exclusively by texture sampling, since they consider objects as being uniform. Fortunately, it is easy to detect and mark trees containing intersections with objects on which textures have been applied. For example,

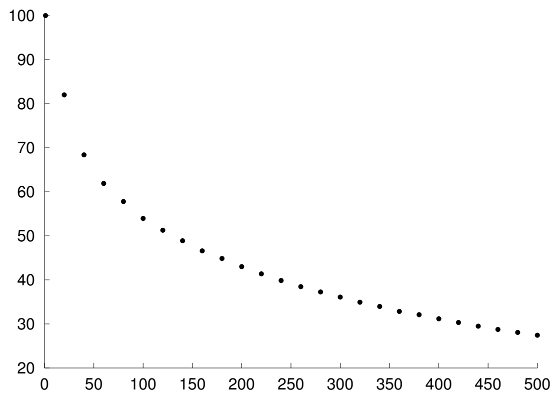


Figure 12: The hyperbolic decrease of the percentage of pixels to be oversampled with the linear growth of the image dimensions.

if we define as zero the ids of all texturized objects when using the primes product method, the category of any tree hitting one of those objects will be zero. While checking for pixels to be oversampled, the renderer can check if its category is zero and select it regardless of adjacent categories. Gödel numbering can also be marked with the number zero. String and binary heap methods can, for example, have categories marked with an invalid first element.

There are many algorithms that perform antialiasing on texture-mapped pixels [10]. By detecting the pixels affected by textures, we can later apply one of these methods only where needed.

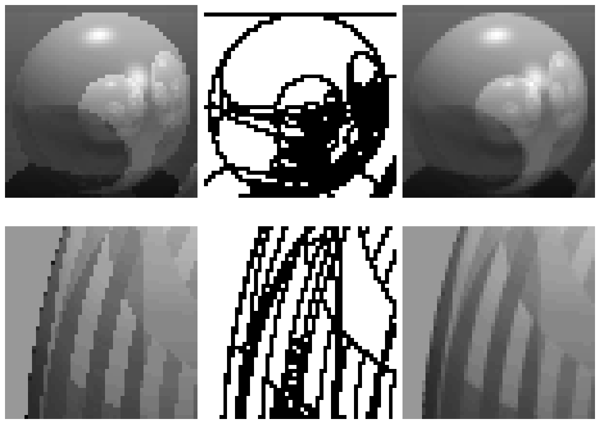


Figure 13: Antialiasing process. (a) Original image for which category information is generated; (b) Pixels chosen for antialiasing by the edge detection method; (c) Result of oversampling chosen pixels.

4.3 Examples

Figure 13 shows the steps of the antialiasing process. With category information available during the rendering process, pixels selected by the edge detection method are oversampled. Oversampling the edges is enough to improve image quality. In fact, some of the edges detected by category comparison do not correspond to discontinuities in the image. For this reason, the method may oversample even more pixels than needed for visual effects.

5 Implementation

The results presented so far have all been obtained with a ray-tracer specially constructed for this article. The program was designed to simplify all implementation aspects of the different categorization algorithms. In order to assess the amount of work involved in adapting an existing ray-tracer to benefit from the methods described in this article, we chose to extend POV-Ray [13], a well-known free-software ray-tracer.

POV-Ray implements two antialiasing methods. Both methods are adaptive to some extent. The default method is a simple threshold method, where a pixel is oversampled at a fixed rate if its intensity is found to differ too much from that of its neighbors. The second method is very similar to that presented by Whitted [15].

With the addition of only 160 lines of code, within a period of three days, we were able to modify POV-Ray to perform edge detection and antialiasing by the primes product algorithm. Primes product was chosen for its simplicity, and was used instead of the comparison function used by the simple threshold method to create a third antialiasing method. A member of the POV Development Team would certainly have saved the time we spent analyzing the source code and thus finish the implementation much faster, given the simplicity of the new algorithm.

Table 2 shows the average number of samples per pixel obtained with POV-Ray and the different antialiasing methods. Although it is hard to define final image quality, Whitted's method produced considerably superior results, in part due to higher sampling rates. Primes product and the simple threshold methods produced similar results regarding quality, although simple threshold generated less samples per pixel.

Method	160×120	320×240	640×480
Thresholding	2.10	1.56	1.29
Whitted	3.33	2.15	1.52
Primes	2.27	1.75	1.52

Table 2: Average number of samples per pixel for different antialiasing methods. Uniform supersampling on every pixel would generate 4 samples per pixel.

Even though the new method oversamples more pixels to obtain results similar to that of existing methods, it is still much better than uniform supersampling. The next section discusses further applications for edge detection based on ray path categorization.

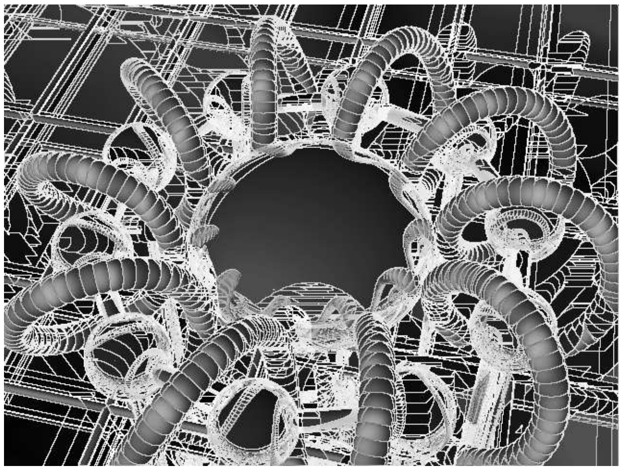


Figure 14: Edge detection by the primes product algorithm running on a POV-Ray sample image due to Truman Brown. Chosen edges have been enhanced over the rendered image.

6 Further applications

Edge detection can be used to generate line-arts of ray-traced scenes. Starting with a blank image, we paint the selected pixels black. The result is a sketch of the rendered image, with no shading information. Examples are seen in Figure 15.

Line-art scenes have more than aesthetic value. They give some insight into the behavior of the ray-tracing algorithm. Figuring out the meaning of the edges, one can learn about the paths the rays belonging to each category followed during the rendering process. Alternatively, generating a sequence of images like that of Figure 7 can reveal what information is gained by the increase of the maximum tree depth. Observing a set of images like Figure 15, one can better understand the internal reflection and refraction in transparent objects.

7 Conclusions

Although the application to antialiasing did not improve over existing methods, the proposed algorithms for edge detection generate interesting results, as shown by the line-art images. We believe that other uses may be found for these results in the near future, perhaps in areas not directly related to computer graphics that use ray-tracing, such as telecommunications or seismic waves analysis.

Future works in the area include the development of other forms of visualization for the category information

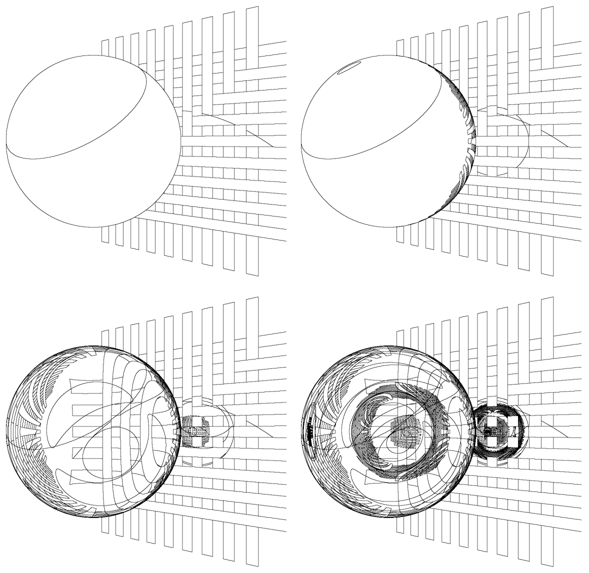


Figure 15: Line-arts of Figure 9 for several depths of recursion.

and the investigation of further applications for the ray path categorization algorithms.

8 Acknowledgements

We would like to thank Vinicius da Silva Almendra, who is coauthor of the first program implementing the ideas presented here. Many thanks to Luiz Henrique de Figueiredo, Antonio Scuri, Alexandre Sieira Vilar, and Ana Elisa Schmidt for the help fine tuning the text. Finally, we would like to thank the SIBGRAPI reviewers, who presented several new ideas for improvements and for the continuation of this work. This work has been developed in TeCGraf/PUC-Rio, and was partially funded by CNPq. TeCGraf is mainly funded by PETROBRAS.

References

[1] J. Arvo and D. Kirk. Fast ray tracing by ray classification. In *Computer Graphics*, Annual Conference Proceedings, pages 55–64. ACM SIGGRAPH, 1987.

[2] Lindsay N. Childs. *A Concrete Introduction to Higher Algebra*. Springer, 1995.

[3] Robert L. Cook. Stochastic sampling on computer graphics. *ACM Transactions on Graphics*, 5(1):51–72, January 1986.

[4] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw Hill Book Co., 1989.

[5] F. C. Crow. The aliasing problem in computer-generated shaded images. *Communications of the ACM*, 20(11), November 1977.

[6] F. C. Crow. A comparison of antialiasing techniques. *IEEE Computer Graphics and Applications*, 1(1):40–48, January 1981.

[7] Mark A. Z. Dippé. Antialiasing through stochastic sampling. In *Computer Graphics*, Annual Conference Proceedings, pages 69–78. ACM SIGGRAPH, July 1985.

[8] Andrew S. Glassner. *An Introduction to Ray Tracing*. Academic Press, 1989.

[9] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Addison-Wesley Publishing Co., 1992.

[10] Homan Igehy. Tracing ray differentials. In *Computer Graphics*, Annual Conference Proceedings, pages 179–186. ACM SIGGRAPH, August 1999.

[11] Don P. Mitchell. Generating antialiased images at low sampling densities. In *Computer Graphics*, Annual Conference Proceedings, pages 65–72. ACM SIGGRAPH, 1987.

[12] R. J. Nelson. *Introduction to Automata*. John Wiley & Sons, Inc., 1968.

[13] Persistence of Vision Development Team. POV-Ray. `http://www.povray.org`.

[14] James Painter and Kenneth Sloan. Antialiased ray tracing by adaptive progressive refinement. In *Computer Graphics*, Annual Conference Proceedings, pages 281–288. ACM SIGGRAPH, 1989.

[15] T. Whitted. An improved illumination model for shaded displays. *Communications of the ACM*, 23(6):343–349, June 1980.

[16] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide*. Addison Wesley, 1999.