RayFilling: a new method to accelerate ray casting

V.Belloli, S. Callegari, M. Della Monica, C. Gatti, D. Marini

Università degli Studi di Milano, Dipartimento di Scienze dell'Informazione Laboratorio di Eidomatica
Via Comelico 39, 20135 MILANO, Italy
tel. +392 55006 358
fax + 392 55006 334
e_mail: marini @ imiucca.csi.unimi.it

Abstract

This paper presents a new method for ray casting solid models described by CSG trees. This new approach accelerates image generation by a reduction of the number of rays analysed in search of an intersection with the objects of a scene. Coherence properties of the projected model, an image filling method and an efficient search of the intersecting rays are the basic assumptions of the approach. The method can be further improved by a progressive refinement technique, so that RayFilling can be used to implement interactive CSG modelling systems. RayFilling alows to visualise a wide variety of primitive solids. This new approach to ray casting solid models can be integrated into existing recursive ray tracing algorithms and reduces therefore the gap between modelling and high quality rendering. *

Keywords: ray casting, ray tracing, progressive refinement, interactive solid modelling, rendering, modelling coherence, seed-fill

---

* Present work is partially funded by CNR, Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo, contract n. 91.00915.PF69

RayFilling: a new method to accelerate ray casting

1. Introduction.

The ray tracing algorithm is a powerful method for generating images of scenes having a very high degree of photo-realism; it is particularly interesting for the simulation of reflective and refracting light properties. Its conceptual simplicity makes it a suitable tool for displaying CSG objects created using some sort of geometric modeller. Its fundamental drawback lies in the computational cost: not only of its recursive version, which generates, at each intersection, up to three secondary rays: a shadow ray, a reflected ray and a refracted ray, but also of the non recursive version, called ray casting.

The basic algorithm can be described by a sequential process, which computes all the intersections for each pixel of the frame buffer along any scan line. This rule is very expensive because for each ray the whole set of geometric objects in the scene must be checked to verify the existence of an intersection. An analysis of the time complexity of such a simple algorithm shows that most of the time (up to 95%) is dedicated to the computation of intersections; this limit has been observed even by the inventor of the ray tracing method [Whitted, 1980].

1.1. Accelerating ray tracing and ray casting

Many solutions have been devised to avoid the computation of object-ray intersection for non-intersecting rays. In an extensive survey Arvo and Kirk [1989] suggested a classification of the most important acceleration techniques of ray tracing into three main areas: faster ray intersection, fewer rays and generalised rays. Typical algorithms that speed up the computation of ray intersection are based on numerical methods that efficiently compute the solution for different kinds of geometric entities [Sederberg and Anderson 1984], [Levner et alt., 1987], or that perform efficient minimax tests on bounding volumes [Rubin and Whitted, 1980], which, in some cases, are organised into hierarchies that reflect the CSG composition [Marini and Tampieri, 1986]. Another typical acceleration technique is the parallel computation of groups of rays, on distributed architecture machines. Many solutions have been devised distributing pixels among processors [Packer 1985] or distributing geometric entities [Glassner 1984] and considering rays as messages exchanged between processors [Dippe and Swensen 1984]. A spatial redistribution, uniform or irregular, of geometric entities can improve intersection acceleration by allowing the adoption of integer arithmetic [Fujimoto et alt. 1986] [Marini and Rossi 1990]. Special purpose hardware has been designed for parallel implementation of these algorithms (e.g. Pixel Machine by AT&T, or LINKS). 'Fewer rays' oriented methods are, for example, those based on statistical methods; they generate at higher resolutions only rays that contribute to a reduction of aliasing phenomena. These algorithms are of major interest for producing realistic effects (e.g. motion blur and depth of field) rather then for speeding up the rendering process [Cook et alt. 1984]. 'Generalised rays' method adopts beam [Heckbert and Hanrahn 1984], cones [Amantides 183] or pencil tracing [Shinya et alt. 1987], all exploiting local coherence of a geometry, nevertheless, many of these methods are typically limited to polyhedral geometric models.

Most of these approaches consider ray tracing as a batch process, and devise solutions that reduce the computation of a scene as a whole. This viewpoint is not suited for interactive modelling application, where the main requirements are, at the same time, efficient interaction and possibility of improving the quality of rendering.

The importance of solid modelling for improved modellisation of complex shapes has been extensively underlined. Interactive solid modelling systems normally adopt a boundary representation, which is very well suited for specialised graphical architecture (like geometry engine) that allows to get real time interaction with displayed objects. This approach has some drawbacks: curved geometry is approximated by plane faces, so that both rendering and computations of volume properties are approximated, and to avoid or limit approximation a secondary CSG representation is frequently adopted. So a gap arises between two different internal representations of the same geometry. Such double representation can be tolerated and accepted if high quality visual representation plays a limited role in design, or if numerical approximation stays below some given threshold. As increasingly sophisticated CAD applications are being developed in new industrial context, where visual design plays a relevant role, it becomes more important to have an integrated environment where modelling and visualisation are simple and interchangeable ways of representing a design object. Technical and implementation problems now arise: to maintain two data structures a larger memory is required and more complex code is necessary to guarantee the coherence between the two representations. If new modelling systems that can take advantage of more up to date methodological approach to programming (e.g. object orientation) are to be devised, a cleaner conception suggests finding a solution that can be based on the more natural internal representation, which in many cases can be CSG. A uniform representation of solid models can also allow to improve user interaction. Moreover research in the field of displaying CSG geometry can open fruitful ideas to the design of new graphical architecture more suited for this kind of geometry then present day ones. In summary we think that ray casting can be seen as a "scan conversion" method for solid objects, while scan conversion of polyhedra is satisfied by established geometry engines.

1.2. Overview of our method

We consider ray casting as the best candidate method displaying a solid model interactively during its construction. In interactive modelling we have some general requirements that must be fulfilled by rendering algorithms:

- quick feedback to highlight a selected entity
- quick feedback in moving, dragging and resizing an entity (or a sub hierarchy)
- quick feedback in checking obstructions and collisions among entities,

We remember that photo-realistic effects are required for visual appreciation of the resulting model, and can be produced only from time to time.

During modelling a user is mainly interested in a detailed display of model regions where his attention is oriented We call this region the focus. A transformation on a geometric model is normally applied to part of a model, and the transformation involves only subparts of this part. We can identify a coherence property that binds all modelling parts that are influenced by the focus chosen by the user. In the case of CSG model, this coherence is linked to the hierarchical structure of the modelling tree; we call it modelling coherence. On the other hand this coherence, from the visual viewpoint, is an image coherence: in other words, the transformation on a model part affects the image projected on the screen, so that modelling coherence and image coherence are strictly linked. Methods for exploiting image coherence are well known in rendering, we mention here, for the sake of completeness only the work of Broonswort, [Broonswort et alt. 1984], who adopts scan line coherence and progressive refinement to speed up ray casting solid objects.

Classical formulation of the ray casting problem can be considered essentially a search problem. Indeed it is necessary to find out which primitives, if any, intersect a given ray. We can reverse the problem and simplify the search step if we know "a priori" that a given ray intersects a given object. We can imagine that any object has a knowledge of an intersection with a given ray, which is the intersection between its surface and the line going from its centre (or centroid) to the viewpoint. We call this the first intersection; once the first intersection has been found, the ray casting problem is reduced to the computation of the pixels that are the projection of the object onto the screen. This can be executed by adopting an 'image filling' algorithm, such as the classical 'seed-filling'.

During interactive modelling the focus of attention suggests a generalisation to find the first intersection of a newly created or transformed object: the user points to objects on the screen, selection is done by identifying a pixel and the object to which it belongs. A rendering process can use this information to identify immediately the intersecting ray, which is the ray passing from the viewpoint to the selected pixel of the selected object.

1.3. Previous works

We postpone a detailed description of our method, and we limit the rest of this introduction to some general considerations.

The relationship between ray tracing and search problem has been underlined by Kaplan [1985], who bases his method on a pre-processing of the geometric model so that the search problem is solved using a BSP-tree. Hanrahan suggests an analogy between ray tracing and region filling [Hanrahan 1986]. The Kaplan approach is elegant and efficient but still far from real time; the Hanrahan approach is mainly an optimisation of beam tracing, and implements an efficient search solution by adopting a breadth first technique; his method is limited to rendering simple primitives and does not support CSG models. The problem of rendering CSG geometry during modelling has been considered by Rossignac and Requicha [1986], who propose a z-buffering method to render B-representations. The relevant efficiency of this method is penalised by the fact that high quality photo-realistic images still require the adoption of a batch oriented ray tracing method. Also Weghorst et alt. [1984] adopt z_buffer techniques, using the so called item-buffer, but when applied to solid CSG models this approach requires a significant memory overhead. Encarnaçao and Requicha [1993] have also addressed the importance of user interaction in modelling CSG solids, and stress the relvance of an efficient ray casting solution to avoid double internal representation of a solid model.

Our seed-filling approach has a first important advantage with respect to other methods: no search is required to find which object is intersected by a given ray; as a consequence fewer ray/object intersections are effectively computed, depending in general on the structure of the scene. If the scene has a prevalence of background, the computed rays are a small fraction of the total. In this paper we shall discuss and compare the efficiency of the algorithm depending on the type of the scene. Another feature of the method is that it takes advantage of the image coherence: the projection of the scene onto the image plane is generated by the filling technique. Antialiasing can be easily implemented, and progressive refinements methods can improve its global efficiency. Moreover the method exploits what we have called modelling coherence: the part of a geometric model that has been modified recently is updated first, and the rest of the scene is computed concurrently either during modelling operations or while the user is idle.

We shall illustrate the RayFilling algorithm by exposing its details progressively. First we shall describe how solid primitives are traced, then we shall consider how the computation of initial intersection can be performed Eventually we shall present the general algorithm to display complex CSG models and scenes. Moreover we shall discuss other features that can be integrated into the method, such as recursive ray tracing and progressive refinements; and we shall discuss performances and compare the method with the standard one.

2. RayFilling.

2.1. RayFilling solid primitives

Typical hidden surface algorithms have a computational complexity that is maximised by a *sort* method: in fact the problem is to decide upon the visibility of a series of geometric elements with respect to a given direction, and visibility depends on the relative position of the geometric elements with respect to the view direction. In the case of ray casting, the maximising problem can be considered a *search* problem: to find out which geometric element is intersected by a given ray. Moreover the search must be repeated for each pixel of the screen. Our approach reduces the role of search so that the number of rays to test against each geometric element is minimised. More important, we avoid testing of rays that do not intersect any element but are lost in the background. Standard ray casting adopts an exhaustive search method, bounding volumes reduce test cost, while space reorganization allows us to adopt binary search methods (tree search).

The first idea is to reverse the way ray casting is considered: instead of looking for a geometric primitive intersecting a given ray, we try to generate a ray that unquestionably intersects a primitive. Once such a ray is known (we call it *first ray*) the intersection can be easily computed and its value can be projected onto the projection plane; from this point on the rendering of the geometric primitive can proceed in the same way as a seed-filling method: all subsequent intersections are computed starting from the first one (*first intersection*).

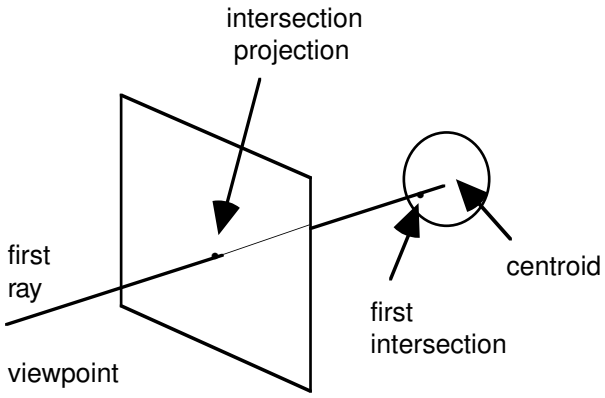


Fig. 1 - Determination of the first intersection

The seed-filling method that has been adopted generates subsequent potential intersections evaluating the 4-neighbours of the first intersection. The filling proceeds recursively by stacking the 4-neighbours until a ray does not intersect the element any more (we call it *missing ray*) or a border pixel has been reached. To avoid recalculation

of previously computed intersections, a bit map (we call it *fill_buffer*) is maintained that records this information with a Boolean value.

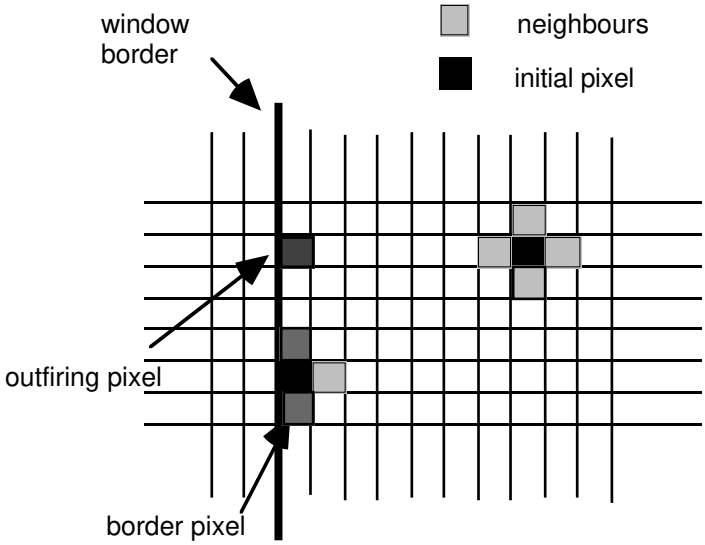


Fig. 2 - Seed filling and clipping

The determination of the colour of an existing intersection is computed by applying an illumination model, which gives the final colour to be associated to each pixel. Ray casting a given geometric element with the above schematised method is similar to scan converting its projection onto the projection plane.

Let us analyse the pseudo-code of the algorithm shown in fig. 3, which computes the rendering of a single primitive. First of all a fill_buffer is initialised and set to 0: it will be used to check whether anyone among the 4-neighbours of a pixel has been previously converted; the fill_buffer is a bit map one-to-one with the image buffer. Then a first ray of a geometric element is chosen and checked with respect to the view pyramid border. In the case it is internal it is immediately pushed into a stack. Otherwise the initial intersection of the primitive is outside the view pyramid and the geometric element must be clipped. To this aim all border rays are pushed into the stack. The main filling loop consists of progressively popping a ray from the stack, checking whether it has been previously analysed. If not, its intersection with the geometric element is computed and its 4-neighbours are pushed into the stack. The loop continues until the stack is empty.

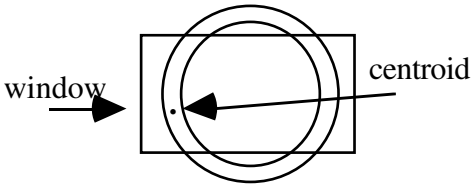
This seed-filling implementation has a memory overhead limited to the fill_buffer (whose size can be about 128 k for a 1024x1024 image resolution) and to the stack for storing intermediate rays.

```
/* RayFilling a solid primitive */
init(fill_buffer);                  /*set fill_buffer to 0*/
if first_ray() != OUTSCREEN
 push_in_stack(first_ray());
else                                /* Outfiring */
 for border_rays() ++               /*push in stack all border rays*/
   push_in_stack(border_ray);
while stack!=EMPTY                  /* seed-fill loop */
```

```
{
 pop_from_stack();
 if fill_buffer[ray] == 0          /* the ray was not previously analysed */
   {
   fill_buffer[ray]=1;
   if intersect_ray() == TRUE              /* the ray hits the primitive */
        push_in_stack(4_neighbours_rays);
   }
}
```

Fig. 3 - RayFilling a single primitive

The above algorithm is usually executed during interactive modelling, when a new primitive is created. In this case the first ray is almost certainly internal to the screen, so the outfiring is usually not executed. A special case arises when a torus has been created as in figure:



If the filling starts from the centroid only the left part would be filled, because no border rays have been pushed in the stack. To solve this case all border rays are pushed in the stack if the primitive is a torus and, if during the filling, a border pixel has been reached.

The function *intersect_ray()* calls the illumination model function that computes the pixel's colour.

2.1.1. The determination of the initial intersection with a primitive

The fundamental step of the algorithm is the selection of the first ray and of the first intersection of the ray with the primitive. In the case of single primitives the solution is quite easy: a ray from the projection point to a point internal to the primitive unquestionably hits the primitive. In our system the typical primitives of CSG modelling (non including volumes limited by spline surfaces) are defined in an absolute co-ordinate system located in the centroid of the primitive; during the modelling stage all primitives are roto-translated and scaled in their final position. Thus the determination of the centroid of the primitive is straightforward. As shown in the above diagram torus has a centroid in the centre of a section circle; polyhedra centroids are defined not in terms of volume interior but for all its faces (since faces can be non convex we choose a vertex).

As previously mentioned, in the case the centroid is external to the view plane window an "outfiring" computation is activated that pushes all the border rays. in the stack. These will be tested during stack pop; in the worst case, this requires the algorithm to test a number of rays 2*(n-1+m-1), for an n x m resolution screen.

A second policy for choosing the first ray and computing the first intersection during interactive modelling is to select the pixel pointed by the mouse.

2.2. RayFilling CSG solids

CSG solids can be rendered following two fundamental approaches, depending on the strategy adopted during the filling stage. The first one, called *by-primitive,* traces the whole solid by 'flood' filling the screen projection of each primitive. The second one, called *4-connected,* traces the whole solid by 'boundary' filling the screen projection of each 4-connected subset of the projection of the whole solid [1].

By flood filling we mean the progressive filling of pixels while the condition of a given primitive intersection is true. By boundary filling we mean the progressive filling while the condition of background hit is false. The flood filling approach is necessary during interactive modelling, when a user inserts, deletes or modifies the model acting on a single primitive in turn. The boundary filling method is better suited for rendering the final view of a complex model or a scene, or for updating the view when the CSG tree is updated, a global transformation is applied to it, or a viewpoint change has been requested.

To evaluate the Boolean operators, a structure is created during CSG tree traversal: the *primitive-list.* An attribute for each 'right' primitive of *intersection* and *subtraction* operators is set to 'not-to-be-filled'. The Boolean operators are therefore evaluated using the Roth diagram applied to 'in' and 'out' points of the ray through each 'to-be-filled' primitive.

2.2.1. Rendering by primitives

The first generalisation of the method is to render a set of primitives in a CSG tree that model a scene by application of Boolean operators. In fig. 4 the pseudo-code has been slightly modified in order to allow the rendering of a CSG tree by 'scan converting' each primitive.

Once the *primitive-list* has been built, the method continues in a similar way as the rendering of a single primitive, until the primitive list has been scanned through. To solve the visibility test we have adopted a standard ray casting classifier: all the intersections of the given ray with the other primitives of the scene are computed. The function *cast_ray* performs this computation using the Roth method. This function can be easily modified to extend the renderer, including computation of shadows, reflections and refraction properties: it is sufficient to substitute *cast_ray* with a *ray_trace* function that invokes a classical recursive ray tracer.

```
/* RayFilling CSG solids by primitives */
make(primitive_list);
for a_primitive in primitive_list[]
  {
 if primitive_list[a_primitive].to_be_filled  /* the primitive must be
                                                 filled */
        {
        init(fill_buffer);                  /*set fill_buffer to 0*/
        if first_ray(primitive_list[a_primitive].name) != OUTSCREEN
               push_in_stack(first_ray(primitive_list[a_primitive].name));
        else                                /* Outfiring */
               for border_rays() ++         /*push in stack all border rays*/
                  push_in_stack(border_ray);
        while stack!=EMPTY                  /* seed-fill loop */
               {
               pop_from_stack();
               if fill_buffer[ray] == 0     /* the ray has not previously analysed */
                  {
```

[1] We follow Foley et alt. terminology

```
                    fill_buffer[ray]=1;
                    if cast_ray(primitive_list[a_primitive].name) == TRUE
                                        /* the ray hits the primitive */
                       push_in_stack (4_neighbours_rays);
                    }
                 }
        }
  }
```

Fig. 4 - RayFilling CSG solids by primitives

The fill_buffer is zeroed before a new primitive is filled This implies that overlapping regions of two primitives are traced twice, but this is only necessary during interactive modelling.

The cast_ray() function calls intersect_ray() mentioned in former algorithm.

2.2.2. Rendering by 4-connected seed-fill

The process of rendering primitive by primitive on a complete scene has the effect that each primitive is completely traced and filled by the RayFilling algorithm before a new one is rendered.

A difference with respect to the former technique is due to the use of the fill_buffer structure. In the first method the role of the fill_buffer is to record that a pixel of the primitive has been previously computed. To this aim the first method initialises the fill_buffer as a new primitive is rendered. On the contrary, the second approach initialises the fill_buffer only at the onset of the rendering process. So if during the filling the fill_buffer location is set, the corresponding pixel has been computed and must not be computed again, because the visibility and classification for the corresponding ray have been completely checked. Another difference is that the filling strategy is based on a boundary condition against the background, while in the flood strategy the test is based on the current primitive.

The visual effect in this case is that a set of primitives that are 4-connected on the projection plane is completely rendered before another part of the CSG scene is rendered.

```
/* RayFilling CSG solids 4-connected */
make(primitive_list);
init(fill_buffer);                  /*set fill_buffer to 0*/
for a_primitive in primitive_list[]
  {
 if primitive_list[a_primitive].to_be_filled  /* the primitive must be
                                                 filled */
        {
        if first_ray(primitive_list[a_primitive].name) != OUTSCREEN
               push_in_stack(first_ray(primitive_list[a_primitive].name));
        else                                /* Outfiring */
               if OUTFIRING ==0     /* border rays not yet pushed in the stack */
                  {
                   OUTFIRING =1;
                   for border_rays() ++           /*push in stack all border rays*/
                     push_in_stack(border_ray);
                  }
        while stack!=EMPTY                  /* seed-fill loop */
               {
               pop_from_stack();
               if fill_buffer[ray] == 0    /* the ray has not previously analysed */
```

```
                    {
                     fill_buffer[ray]=1;
                     if cast_ray() != NO_INTERSECTION
                                              /* the ray hits some primitive */
                        push_in_stack(4_neighbours_rays);
                    }
                }
            }
        }
```

Fig. 5 - RayFilling CSG solids by 4-connected seed-fill

As the RayFilling proceeds the seed pixel for the primitive under test in the primitive_list is checked against the fill_buffer: if it is 0 the seed-fill procedure for 4-connected sets is activated; if it is 1, the corresponding ray has been traced before, and the primitive has been previously rendered in that pixel because it is 4-connected with some other pixel of the primitive or of another primitive that comes first in the primitive_list.

A third difference with former algorithm is the treatment of outfiring: since we want to test the border pixel only once, we use a flag that is set at the first outfiring occurrence. Since the fill_buffer is zeroed only at the onset, this guarantees that no changes occur to the border pixel status.

The 4-connected RayFilling is particularly interesting because less stack usage is required, fewer tests and assignments on the fill_buffer are computed and fewer rays are cast. These advantages are lost when the whole background of the scene is covered by some primitive. In the evaluation paragraph we shall present some comparison results.

2.2.3. The determination of the initial intersection in CSG solids

CSG is a very flexible and powerful modelling method Nevertheless all non trivial information on the model must be explicitly computed: this is particularly true for the determination of a centroid. What we are interested in is not the effective centroid of a solid CSG model, but an internal point. Since a CSG model is composed of solid primitives, a point that is internal to a primitive is also internal to the CSG solid. Therefore we are reconnected to the same problem solved in the case o single primitives casting. It is important to underline that not all the primitives of CSG solid are effectively included in the solid: primitives that are linked to the solid by a *union* operator are completely included, while primitives that are the first argument of an *intersection* or *subtraction* operator are partially included. It may happen that the *intersection* or *subtraction* operator eliminates from the primitive the region that includes the centroid, and the effect is a limited overhead for tracing subtracted or non visible pixels. We can avoid any problem by tracing completely only the first argument primitives, as described in § 2.3. It may happen that excess rays are traced on the region of the first primitive that does not belong to the intersection region But this is in general less expensive then the explicit calculation of an internal point to an intersection region.

3. RayFilling by Progressive Refinements

The RayFilling technique allows us to implement a progressive refinement method, which is particularly useful for improving feedback effects during interactive modelling.

The progressive refinement method that we have implemented allows us to display almost immediately at a lower resolution an image of a solid model so that, after any interactive operation, an evaluation of its effect can be performed without waiting for the completion of the rendering. We have adopted a three stage refinement process, based on two distribution arrays of pixels.

3.1. Pixel cells 9x9 and 3x3

The first stage of the progressive refinement is based on a limited resolution of a linear factor of 9. The rays are searched so that the pixel size is 9x9 times the effective image resolution. We call this a cell. The ray intersection is computed considering the centre pixel of the cell, and the computed value is used to evaluate the illumination model and to colour all the pixels of the 9x9 cell.

On average the 9x9 cell is computed tracing 1.23% of the total rays, and the corresponding computation time is about 1/81 of the total time of the standard technique. When this stage is completed the user obtains a rough display of the model suitable for a preliminary evaluation.

The process of refinement continues while the user is idle, or concurrently with other modelling operations; the 9x9 cell is subdivided into 9 smaller 3x3 cells. Only the 8 external rays are computed, because the central one has been computed during the first stage; the colour attributed to the 8 3x3 cells is relative to their central pixel. At this time 11.11% of the total number of rays has been traced and the computation time is about 1/9 of the total time, with a linear resolution of 1/3 of the effective one.

10	46	19	14	50	23	11	47	20
64	**1**	73	68	**5**	77	65	**2**	74
28	55	37	32	59	41	29	56	38
16	52	25	9	45	18	17	53	26
70	**7**	79	63	**0**	72	71	**8**	80
34	61	43	27	54	36	35	62	44
12	48	21	15	51	24	13	49	22
66	**3**	75	69	**6**	78	67	**4**	76
30	57	39	33	60	42	31	58	40

Table 1 - Distribution of the priority of computation in the pixel array

The third stage refines the rendering by computing the 72 remaining pixels It introduces a limited overhead due to the necessity of tracing all the remaining pixels, even those that are beyond the border of the geometric primitive, in order to reach the desired resolution.

The efficacy of the refinement can be improved by taking advantage of a visual effect: if the illumination model and colour computation are applied only to the centre pixel of the 9x9 and 3x3 cells, the image appears on the screen as "emerging from the fog", consequently improving the perception of the shape by the user. In this case there is no overhead to reset the background pixels. The user is allowed to select the starting level of resolution for the progressive refinement method, so that when working on lower resolution systems the 9x9 stage can be avoided. The order of computation of the pixel in the arrays has been chosen so that there are no privileged areas, as shown in Table 1. Colour image 5 exemplifies the visual effect of progressive refinement.

The progressive refinement method can also be used to generate antialiased images; antialiasing is aided by information contained in the fill_buffer. Normally antialiasing is required for final high quality rendering, or for detailed inspection of a critical area. In these cases a delay in image generation can be accepted; therefore antialising is available only for the 4-connected RayFilling method.

4. Performance evaluation and comparison

We give a synthetic outline of the computational complexity of RayFilling, comparing it to traditional ray casting. Let us consider first ray casting a single primitive. Let k be the average amount of pixels covered by the primitive, out(k) the average number of primitive's border pixels; let I be the computationl cost of the intersection computation with a primitive Then we have a complexity C_{RF}:

$$C_{RF} \approx [k + out(k)]* I$$

We do not consider the overhead due to stack and fill_buffer manipulation, which is much faster then intersection computation. We can compare this cost with the cost of traditional ray casting:

$$C_{RC} \approx n*m*I$$

where n*m is the size of the screen. If we consider p primitives, RayFilling gives a cost:

$$C'_{RF} \approx q * (q_{in}\, kq_{in} + q_{out}\, kq_{out}\ 2 * (n+m)) * I$$

where I is the average cost of intersecting a primitive, q (< p) is the number of primitives to be filled, $k_{q_{in}}$ is the average number of pixels covered by primitive q_{in} that is totally inside the window, including border pixels, q_{out} is the number of primitives (partially) outside the window, $k_{q_{out}}$ is their average number of pixels. Traditional ray casting has a cost:

$$C'_{RC} \approx n*m*p*I$$

It is evident from above equations that the advantage of RayFilling is relevant in the case that the percentage of screen coverage of primitives is small. A comparison of the RayFilling algorithm to other approaches to ray casting is meaningful only for the 4-connected approach. On the contrary the primitive-by-primitive rendering is, in our approach, the only possibility for interactive modelling of CSG solids. The major advantage of RayFilling is in scenes having a wide background. And its performances increase with respect to other methods as the resolution increases.

We have integrated the RayFilling algorithm in a complete ray tracing system, implemented by our research group. The bounding volume test is based on a spherical shape centred around any primitive; the bounding volumes are organised in a hierarchy that can speed up bounding volume test. The computer system is a Silicon Graphics Indigo SX, with 48 M Byte main memory, Times are expressed as total time, including I/O, and since the tests run on the same program with different options I/O can be considered as constant. The final images of the Spheres, Snow-flake and Ambient tests are 512x512. In table 2 the results are shown.

		RAY TRACING with Bounding Vol		RAY TRACING No Bounding Vol		RAY CASTING with Bounding Vol		RAY CASTING No Bounding Vol	
		Standard	Filling	Standard	Filling	Standard	Filling	Standard	Filling
1000 reg spheres	time	3:17:35	2:26:54	2:52:42	1:54:51	1:40:46	0:47:44	1:28:40	0:39:11
	# rays	415.040	281.369	415.040	281.369	262.144	128.473	262.144	128.473
	# rays in vol	167.889	167.889			114.027	114.027		
1000 rand spheres	time	6:17:47	6:07:14	5:25:38	4:54:42	1:40:55	1:38:15	1:27:23	1:16:27
	# rays	661.146	656.873	661.146	656.873	262.144	257.871	262.144	257.871
	# rays in vol	493.674	493.674			256.769	256.769		
Snow flake	time	3:08:18	2:13:12	2:43:20	1:44:50	1:20:10	0:24:12	1:10:35	0:19:26
	# rays	361.998	176.552	361.998	176.522	262.144	76.668	262.144	76.668
	# rays in vol	112.094	112.094			74.064	74.064		
Ambient	time	0:09:47	0:08:25	0:18:47	0:16:43	0:03:28	0:02:46	0:07:12	0:05:00
	# rays	350.166	260.741	350.166	240.741	262.144	172.719	262.144	172.719
	# rays in vol	240.779	240.777			171.266	171.266		

Table 2 - Time comparison for different ray tracers

First of all we note that the number of rays cast can vary from 34% to 70% less then those cast by a traditional ray tracer, except in a worst case, the 1000 random spheres, where the screen is almost completely covered by the objects. The '# rays in vol' gives the number of rays that hit a bounding volume The number is less then the number of rays cast or traced by the RayFilling algorithm, because we also test pixels around an object border. We note the increase in total time when the bounding volume test is activated for the models 'Spheres' and 'Snowflake': this is because the bounding volume test has a higher cost than a simple intersection test on a primitive sphere (recall that we adopt spherical BV). In the test case 'Ambient' we have a slightly better time in favour of ray filling without bounding volume, because the scene has different kinds of primitives, and the BV test is more efficient for non spherical shapes.

We have implemented the RayFilling algorithm in an experimental interactive solid modelling system, and we have verified that the creation and update of a simple model (as shown in image 6) composed of 600 primitives requires less than one minute. The rendering process runs concurrently to modelling and takes advantage of idle states of the user Rendering is always done with progressive refinement, and in a few seconds a user has a reliable perception of the result of the operation applied and can continue the modelling process in concurrence with progressive rendering.

5. Conclusions

We have devised and implemented a new method for ray casting CSG solid models. The method is based on two ideas: to minimise the search for a first intersection, and to take advantage of local coherence of the model and of its projected image. The first idea leads to the selection of a ray that unquestionably hits a given primitive by selecting a ray going from the viewpoint to the centroid of the primitive. The second idea is implemented using a seed-fill algorithm to compute all other pixels of the projected primitive, implementing a sort of solid-to-plane scan conversion. The method can proceed primitive by primitive or it can ray cast a whole scene by filling 4-connected regions of the projected scene. Both techniques can be realised by progressive refinement; the 4-connected seed-fill can be extended to reduce aliasing effects.

Our RayFilling approach can be easily integrated in any existing standard ray tracer. It allows us to display a complex scene quickly, therefore it is suited for the development of an interactive CSG modeller integrated to a photo realistic image

renderer. To verify this we have also implemented an experimental interactive modeller, which confirms our assumptions.

It is worthwhile to comment briefly on the possible adoption of a z_buffer technique to further improve alorithm speed-up. We have considered recurring to this data structure to support depth test, but it would require a very large memory in order to maintain information about any intersection of a given ray with all the primitives. It is indeed important to recall that the method is applied to solids described by booleans operator and a given ray can collect hundreds of intersections.

Some modifications could be devised to further improve performances: the outfiring phase (to clip a primitive partially outside the window) could be accelerated by improving the border pixel's intersection test; moreover the tracing of *intersection* or *subtraction* CSG operator could start from a seed that is visible, by identifying the projected primitive with smallest area. Moreover if a geometric element can be described with parametric equations, the determination of each new intersecting ray during the seed-fill process can be solved by incremental methods. This can be adopted to RayFill spline delimited volumes.

6. References.

Amantides J. - *Ray tracing with cones* - Computer Graphics, 18(3), 129-135, July 1984

Arvo J. & Kirk D. - *A survey of Ray tracing Acceleration Techniques* - in: Glassner A.S. Ed. - *An introduction to ray tracing* - Academic Press, 201-262, 1989

Atherton, P.R. - *A Scan-Line Hidden Surface Removal Procedure for Constructive Solid Geometry* - Computer Graphics, 17(3), 73-82, July 1983

Bronsvort A.F, van Wijk J.J. and Jansen F.W. - *Two methods fro improving the efficiency of ray casting in solid modelling* - CAD, 16(17), 51-55 (1984)

Cook R.L., Porter T. and Carpenter L. - *Distributed ray tracing* - Computer Graphics, 18(3), 137-145, July 1984

Dippé M., Swensen J .- *An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis* - Computer Graphics, 18(3), 149-158, July 1984

Encarnaçao L.M. and Requicha A.G.A. - *Direct Graphic User Interaction with Modelers Based on Constructive Solid Geometry* - in: Goebel M., Teixeira J.C. Eds. *Graphics Modeling and Visualization in Science and Technology*, Springer Verlag, Berlin, 1993

Foley J. D., van Dam, A., Feiner S. K., Hughes J. F.- *Computer Graphics Principles and Practice* - Addison Wesley Publisjing Co., 1990

Fujimoto A., Tanaka T., Iwata K. - *ARTS: Accelerated Ray Tracing System* - IEEE Computer Graphics & Applications, 6(4), 16-26, April 1986

Glassner A.S, Ed. - *An Introduction to Ray Tracing* - Academic Press, 1989

Glassner A.S. - *Space Subdivision for Fast Ray Tracing* - IEEE CG&A- 4(10), 15-22, October 1984

Hanrahan P. - *Using caching and Breadth-First Search to speed up Ray-Tracing* - Proc. of Graphics Interface, 56-61, Vancouver, 1986

Heckbert P.S., Hanrahn P. - *Beam tracing polygonal objects* - Computer Graphics 18(3), 119-127, July 1984

Kaplan M.R. - *Space-tracing, a constant time ray tracer* - Siggraph '85 State of the Art in Image Synthesis seminar notes, July 1985

Levner G., Marini D., Tassinari P. - *A simple general Method for Ray Tracing Bicubic Surfaces* - in: Earnshow R.A. Ed. *Theoretical Foundations of Computer Graphics and CAD*, NATO Asi Series, Berlin 1988; also in: T.L. Kunii Ed., 285-302, Computer Graphics 87, Springer, Tokio, 1987

Marini D., Rossi M., *Optimization of ray tracing algorithm*, (in Italian), PIXEL anno 11, n. 9, 1990

Marini D., Tampieri F. - *A Ray Tracing System* - proc. II Colloque Image, Nice 1986

Packer J. -*Exploiting Concurrency; a ray tracing example* - Inmos Technical note 7, 1985

Rossignac J.R., Requicha A.A.G. - *Depth-Buffering Display Techniques for Constructive Solid Geometry* - CG & A, 6(9), 29-39, September 1986

Roth S. - *Ray Casting for Modeling Solids* - CGIP, 18(2), 109-144, February 1982

Rubin S, Whitted T. - *A three dimensional representation for fast rendering of complex scenes* - Computer Graphics, 14 (3), 110-116, July 1980

Sederberg T.W., Andersoon D.C - *Ray Tracing of Steiner Patches* - Computer Graphics, 18(3), 159-164, July 1984

Shinya M., Takahashi T., Naito S. - *Principles and applications of Pencil Tracing* - Computer Graphics, 21(4)), 45-54, July 1987

Weghorst H., Hooper G., Greenberg D.P. - *Improved computational methods for ray tracing* - ACM Trans. on Graphics, 3(1), 52-69, Jan. 1984

Whitted T. - *An improved illumination model for shaded display* - CACM, 23(6), 343-349, June 1980

Acknowledgements

We want to express our thanks to Filippo Tampieri, who realised the first version of the standard ray tracer; Maurizio Rossi for the implementation of ARTS version and for providing comparison data; Andrea Colombo who implemented the parallel version of the standard ray tracer and extended the treatment to depth of field and motion blur effects; and all the students and collaborators of the Laboratorio di Eidomatica, who have been patient throughout our experiments and workstation overload. Models are due to: Maurizio Rossi, Gianluca Pensavalle, Marilena Mingoia, Vincenzo Belloli

Image Captions

Image 1: 1000 spheres uniformly distributed

Image 2: Snowflake composed of 740 spheres

Image 3: 1000 spheres distributed at random

Image 4: Another scene composed of about 80 primitives

Image 5: Progressive refinement of a model; top: by lower resolution; bottom: by center pixel computation. The model is texture mapped.

Image 6: A CSG design model composed of 600 primitives, realised using the experimental interactive modeller.