New algorithm for the rendering of CSG scenes

A New Algorithm for the Rendering of CSG Scenes

Andrea Sanna[†] *Paolo Montuschi*[†] *Antonio Fisone*[†] *Bartolomeo Montrucchio*[‡]

[†] Dipartimento di Automatica e Informatica,
Politecnico di Torino, corso Duca degli Abruzzi 24,
10129 Torino (Italy)
[‡] Centro di Servizi Informatici e Telematici,
Politecnico di Torino, corso Duca degli Abruzzi 24,
10129 Torino (Italy)

Keywords: rendering of CSG scenes, bounding box computation, shadow detection, Computer Aided Design.

Abstract

The generation of 3D solid objects, and more generally solid geometric modeling, is very important in Computer Aided Design (CAD). An important role is played by the Constructive Solid Geometry (CSG) representation scheme. In CSG, objects are described by trees of Boolean operations on half-spaces or boundary primitive solids.

The study of techniques to speed up rendering of scenes modeled with CSG scheme is an attractive field of research; in this paper we propose a new algorithm which reduces the computation complexity for ray casting approaches. Our strategy identifies a set of areas on the plane of view where the rays starting from the observer have to be traced; for each zone, only a portion of the entire CSG tree has to be considered for intersection tests instead of the whole database of the primitive objects. A comparison of our algorithm with a ray caster adopting bounding volume hierarchies and with a freeware ray tracer called POV-Ray, shows that, for the considered examples, we may reduce the intersection tests up to one third of the ones performed adopting standard optimizations

1 Introduction

Several computer graphics algorithms heavily relate their performances to the efficiency of the tests of intersection between the rays of light sources and the objects of a synthetic scene. Since Whitted [1] refined the original ray tracing algorithm to implement a global illumination model, which involves the phenomena of reflection, refraction, shadow and specular reflection, ray tracing has become one of the most powerful approach to image synthesis. Whitted found that the rendering time of a scene obtained with a ray tracer is dominated by the intersection calculations, which may require up to 95% of the whole time, thus, many algorithms to speed up ray tracing were proposed.

One way to speed up ray tracing is to attempt to reduce the number of intersection tests. The most primitive approach was suggested by Rubin and Whitted [2]; the basic idea was to surround the objects with bounding volumes of very simple shape. If a ray does not intersect the bounding volume, it is not necessary to check the intersection with the object enclosed. Since most of the rays will not intersect with the bounding volume many expensive tests will be saved. In [2] spheres were used as bounding volumes. The main drawback of this approach is that the complexity depends on the number of the objects belonging to the scene, since all bounding volumes have to be tested to detect the object closest to the observer.

In order to avoid this, two main methodologies have been proposed in the existing literature: hierarchies of bounding volumes ([3],[4]) and spatial subdivision ([5], [6], [7]); these techniques will be reviewed in section 2.

The algorithm proposed in this paper is particularly attractive for ray casting applications of CSG (Constructive Solid Geometry) models. Constructive Solid

Geometry (CSG) describes the objects in terms of a set of primitive solids, (such as prisms, spheres, cylinders, cones and so on) and of operators (*union, intersection* and *difference*). An object is described in terms of a binary tree whose leaves are the primitive solids and the intermediate nodes are the operators [8]. CSG has been found to have a very interesting application environment in the description of mechanical objects and in the modeling of synthetic images. A straightforward method for intersecting rays with a CSG model is to classify each ray against the CSG tree, determining the intervals along the rays which intersect the solid. Roth in [9] described this process as a recursive walk down the tree structure, intersecting a ray with each CSG primitive, and combining the resultant intervals according the operators walking up the tree. This strategy may be accelerated using bounding volumes; in [9] the application of 2D boxes for primary rays and 3D spheres for the other rays was discussed.

With our technique we can compute, before the rendering process begins, a group of areas, which identifies a set of independent rectangles on the screen where the rays have to be traced. Moreover, just a limited CSG sub-tree of the whole scene has to be tested for intersection for each rectangle. We will show with practical examples that the application of our method leads to a reduction of the intersection tests and, hence, to a reduction of the computation times.

The content of the paper is as follows. In section 2 basic works on ray tracing acceleration are reviewed; in section 3 we provide the notation used in this paper, while basic idea and a detailed description of the algorithm are outlined in section 4. Performance comparisons of our method vs. a hierarchical approach of bounding volumes and vs. a freeware ray tracer called POV-Ray are presented in section 5. Finally, several remarks can be found in section 6; mathematical details are reported in the appendix.

2 Previous works

Two main methodologies have been developed to speed up ray tracing process; the former strategy is based on hierarchies of bounding volumes ([3],[4]), while the latter is based on spatial subdivision ([5], [6], [7]). With the first approach, each object is surrounded by a bounding volume. The bounding volumes are recursively combined into a tree, by picking up and surrounding some of them with a larger bounding volume. This process is repeated until a bounding volume encapsulates the whole scene. In order to speed up the ray/volume test, the bounding volumes are chosen among the solids with "simple shape". Spheres and rectangular prisms with the edges parallel to the coordinate axes are often used.

The second approach is called spatial subdivision because it subdivides the synthetic scene into non-intersecting cells (voxels) and, then, it relates each cell to a list of objects. The scene is recursively partitioned until each cell contains less than a fixed number of objects. Bounding volumes may offer good object encapsulations but poor hierarchies; on the other hand, spatial subdivision techniques provide good hierarchical organizations but poor bounds.

The idea to project the objects on the plane of view to speed up the intersection tests for the rays traced from the observer is not new. For instance, Coquillart in [7] proposed a method where the projections on the plane of view are administrated in a graph. With this approach is easy to find the intersection point by travelling through this graph along the ray. Unfortunately, as pointed out by Gervautz in [10], this technique can not be used for CSG models. We must keep memory of the operators of the CSG tree to recursively combine the intersections at the leaf level to obtain the intersection point, if it exists, closest to the observer at the root level. With the approach proposed in [7] this is not possible. Gervautz in [10] proposed a method to build dynamic temporary object trees in order to reduce the part of scene to be considered for intersection with each ray. With this method the primitive CSG objects are projected onto a plane; each projection divides the plane into four half-planes. The projections of all objects produce a matrix of rectangles (raster) on the plane. The objects overlapping with each rectangle have to be found, in this way, for each rectangle may be built a temporary tree according to several simple rules. With this approach, each time a temporary tree has to be built the whole CSG tree must be analyzed and this is a time-consuming step. A survey of the acceleration techniques can be found in a chapter written by Arvo and Kirk in [11].

Our algorithm differs from the others known in the literature for the methodology of construction of the bounding entities at the intermediate nodes and at

the root level of the CSG tree. In particular, a new definition of bounding box allows us to find, in an effective way, the CSG sub-tree to be considered for the intersection with each ray traced. The problem of computing the bounding box of the root of a CSG tree starting from the bounding boxes of its leaves, has already been considered in [12], [13], [14] and [15]. In particular, in [15] the bounding box group theory has been presented. With this approach, the limitation of having just one box at any level of the CSG tree has been relaxed. For each node of the tree, the bounding entity is identified as a group of boxes; in this way, it has been proved that the bound at the root level is as tightest as possible. In this paper we extend and improve this theory, by presenting an algorithm which is based on a new definition of bounding box. Our bounding box is a rectangular prism with the edges parallel to the coordinate axes, as in [15], but we associate with each box a CSG sub-tree to be considered when a ray strikes the box. With this approach we maintain the tightness of the bounds of bounding volume techniques and only the necessary intersections ray/object are performed as in the spatial subdivision methods.

3 Definitions and symbols

In this section definitions and notation used in this paper are provided:

Definition 1: Bounding Box. In this paper we define a bounding box (bb) as a prism whose edges are parallel to the coordinate axes. A bb is described by a pair of two vertices connected by a diagonal and by a CSG sub-tree; for instance: $\{(X_{min}, Y_{min}, Z_{min}); (X_{max}, Y_{max}, Z_{max});$ $(1 \cap 2) \cup 3\}$ (each object at the leaf level of a CSG tree may be identified with a number).

Definition 2: BB group. A *group* of bbs (BBG) is a list of bbs (i.e., $\{bb_1, bb_2, \ldots, bb_n\}$) such that no bb belonging to the list can enclose or intersect another bb of the same list.

Symbols. In this paper we denote the *union* and the *intersection* operators with the symbols $\cup$ and $\cap$, respectively; the symbol $-$ will be used to denote the difference operator. Capital letters are used to represent the BBGs, small letters denote the bbs belonging to a group and the symbol $ST()$ represents the CSG sub-tree associated with each bb; for instance: $A = \{(a_1, ST(a_1)), (a_2, ST(a_2)), \ldots, (a_n, ST(a_n))\}$.

4 The algorithm

4.1 Basic idea

In [15] it has been proved that the BBGs algebra is Boolean; therefore, the BBG of the root is as tightest as possible bounding entity obtained considering the bounding boxes at leaf level and combining them according to the operators of the intermediate nodes. The main limitation of the BBGs approach of [15] is that there is no relationship among the objects (primitive CSG objects) and the bounding boxes at the root level. We overcome this drawback by defining a **new** bounding volume. In this paper, a bounding box is described by two fields:

1. a pair of vertices connected by a diagonal, for instance the lower left and the upper right corners of a prism;

2. a CSG sub-tree of the whole scene, which has to be considered for intersection when a ray strikes the bounding box.

Our approach requires new definitions of union, intersection and difference operators, which are different from those provided in [15]. These definitions, together with the mathematical details, can be found in the appendix.

4.2 Steps of the algorithm

The operators and data structures (see appendix) proposed in this paper have been used to develop a rendering (software) system based on the ray casting approach [16]. By ray casting approach we mean a rendering algorithm which does not consider the secondary rays but only the rays traced from the observer. Ray casting algorithms provide good quality rendering with computational times lower than ray-tracing-based methods; on the other hand, a ray casting approach can not handle optical effects such as reflection and refraction. Ray casting can be used for fast previews of complex scenes, in volume rendering algorithms and in animation rendering.
With the BBG method we can compute, before the beginning of the rendering process, all the areas on the screen where the rays have to be traced. The algorithm can be summarized by the following steps:

1. the BBG at the root level of the CSG tree is computed applying the rules shown in the

appendix. We consider a ray casting algorithm, hence, only the projection of the bounding boxes on the plane of view must not overlap; therefore, during this phase the intersections among boxes in three dimensions are not eliminated. If a ray tracing algorithm were employed, the intersections among 3D boxes should be resolved.

2. Each bb is projected on the screen (2D space) using the observer as center of projection. For each vertex, a line linking the view point with the vertex itself is computed, and the point of intersection of this line with the screen is obtained. For each bounding box a rectangular area is found, which encapsulates the projected vertices in the tightest way. The CSG sub-tree of the projected 3D box is associated with the corresponding rectangle.

3. The intersections among the rectangles are eliminated (Fig. 1 and Fig. 2) and a list of areas is obtained (observer's group); a CSG sub-tree is associated with each area. Only the CSG sub-tree of a rectangle is considered for intersection when a ray is traced into the rectangle itself instead of the whole CSG tree. In order to discard the new rectangles which do not contain any CSG primitive a test to detect the "empty" areas could be executed. In this way, the number of rectangles might be reduced as well as the complexity of this step.

4. For each light it is determined a plane where 3D boxes can be projected by using the light itself as center of projection.

5. Steps 2 and 3 are executed for each light source by computing a list of rectangles for each light (light groups).

6. For each pixel enclosed by an observer's rectangle a ray has to be traced. If the ray does not strike any object of the rectangle CSG sub-tree, the background color is assigned to the pixel, otherwise, it must be determined if the point hit is either illuminated or in shadow.

7. A line which joins the point and the first light source is computed. The rectangle (of the light group) enclosing the intersection point between the plane of the light and the line has to be found. The objects associated with this rectangle are tested for intersection with the line itself. If no intersection is detected the light under test is not occluded; otherwise, its contribution must not be considered. This step has to be repeated for each light.

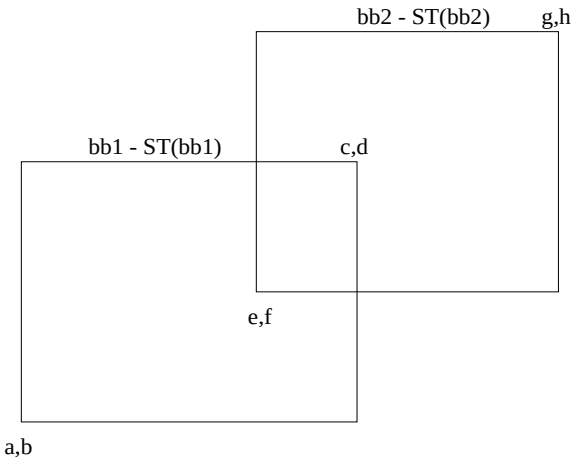

Figure 1: Two overlapping bbs.

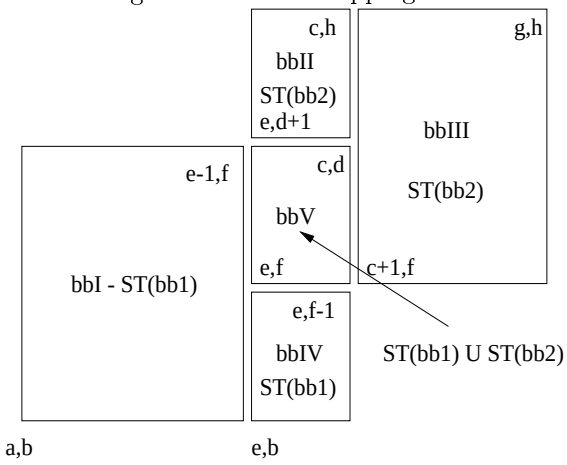

Figure 2: Five new bbs have been obtained.

A C-like pseudo code of the algorithm is shown in Fig. 3. The *read_scene_descr*() procedure reads the scene description from file $file\_des$, while $build\_3D\_BBG()$ builds the three dimensional BBG at the root level. The 3D BBG is projected with the $project\_3DBBG$ procedure and the overlapping rectangles are decomposed by *decompose_overl_rect*. For each light a plane of projection ($find\_project\_plane$) is determined and a list of non overlapping areas is obtained as for the observer. Then, each rectangle of the observer is considered and a ray is traced for each pixel ($trace\_ray$); if no intersection is detected the pixel is set to background color, otherwise, the contribution of each light is computed. The $find\_line$ procedure computes a line joining the current light source to the

struck point; then, the rectangle of the light enclosing the intersection between line and the plane of projection is determined by *find_rect_light* procedure. Finally, if the light is not occluded its contribution is added by *add* procedure. When all light sources have been considered the pixel is rendered.

```
struct {
        int xmin, xmax, ymin, ymax;
        CSG_tree *CSG_head;
} rectangle;
render()
{
        int light, num_light;
        rectangle *obs_list;
        rectangle *vett_light[num_light];

        read_scene_descr(file_des);
        build_3D_BBG();
        obs_list=project_3DBBG(observer,plane_view);
        decompose_overl_rect(obs_list);
        for(light=1;light<=num_light;light++)
        {
            new_plane=find_project_plane(light);
            vett_light[light]=project_3DBBG(light,new_plane);
            decompose_overl_rect(vett[light]);
        }
        for(rect_oss=1;rect_oss<=num_rect;rect_oss++)
        {
            for(y=ymin;y<=ymax;y++)
                for(x=xmin;x<=xmax;x++)
                    intersection=trace_ray(x,y,CSG_head);
                    if(intersection==NULL)
                        set_pixel(x,y,background);
                    else
                    {
                        for(light=1;light<=num_light;light++)
                        {
                            line=find_line(intersection,light);
                            find_rect_light(line,light);
                            shadow=searc_intersection(line,CSG_head);
                            if(shadow==NULL)
                                contrib=add(light);
                        }
                        set_pixel(x,y,contrib);
                    }
        }
        return;
}
```

Figure 3: Pseudo code of the algorithm.

5 Performance comparison

In this section a performance comparison of the proposed method (Algo 1) vs. a hierarchical approach based on bounding volumes (Algo 2) and vs. a freeware ray tracer called POV-Ray (Persistence of Vision Ray Tracer) [17] is presented. For our tests we used POV-Ray version 3.01 for Windows 95. Computational times have been obtained by using a PC with a Pentium Pro processor at 133 MHz; each picture has been rendered with a resolution of 640x480 pixels in true color.

Algo 2 implements an acceleration technique based on hierarchies of bounding volumes. Each primitive CSG solid is surrounded by a bounding box, then the bounding boxes are combined according to the CSG operators; in this way, a bounding box is computed for each node of CSG tree. The box at the root level encapsulates the whole scene. The rules to combine the boxes are very easy; when an union operation between two boxes A and B has to be performed, the bounding box surrounding in the tightest way A and B is computed. On the other hand, when an intersection is encountered only the volume shared, if it is not empty, by the two boxes is taken. For the difference between two boxes A and B, the box A is chosen as result. When a ray strikes the box of a node the sibling nodes are considered; if the bounding box of a sibling node is not hit by the ray, its sub-tree CSG can be pruned without other tests, otherwise, the hierarchy has to be recursively descended until a leaf node is reached. The intersection points, if they exist, with primitive CSG solids have to be combined according to the CSG operators to obtain the closest intersection point to the observer.

In order to be fair, it has been used the same ray casting software both for bounding volumes (Algo 2) and for the proposed method (Algo 1), that is, both implementations use the same light model and the same procedure to test the intersections with primitive CSG objects.

5.1 Image rendering

Four examples have been considered:

- example 1: a simple molecule, modeled with 20 objects (Fig. 4);
- example 2: a wheel of cart, modeled with 12 objects (Fig. 5);
- example 3: a ball bearing, modeled with 61 objects (Fig. 6);
- example 4: three gears, modeled with 93 objects (Fig. 7).

The first example has been built as union of spheres and cylinders and it represents a very simple CSG model; the second example has been built as union of cylinders of which two of them were achieved with difference operations. The third and the fourth example represent mechanical pieces. In Table 1 we provide the number of intersection tests for Algo 1, Algo 2. Scenes as similar as possible to the considered

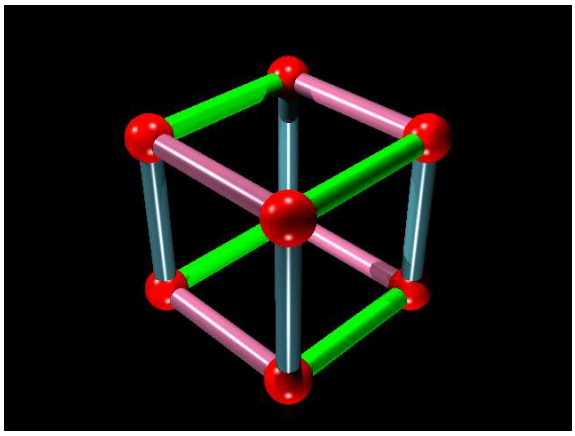

Figure 4: Example 1: a simple molecule.

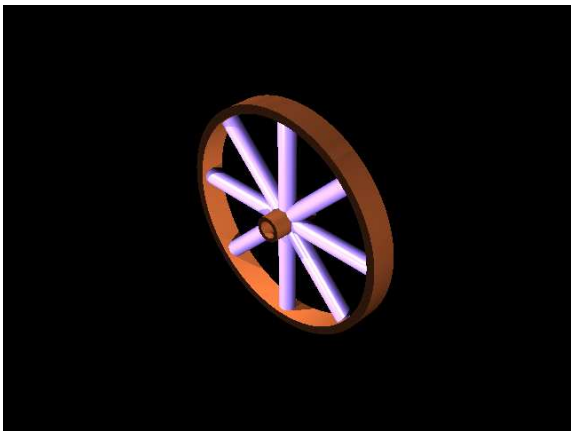

Figure 5: Example 2: a wheel of a cart.

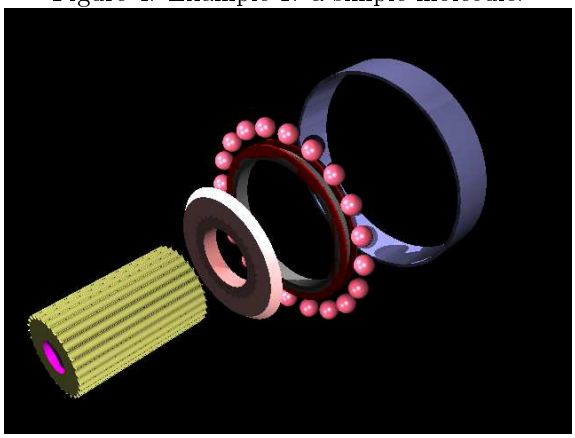

Figure 6: Example 3: a ball bearing.

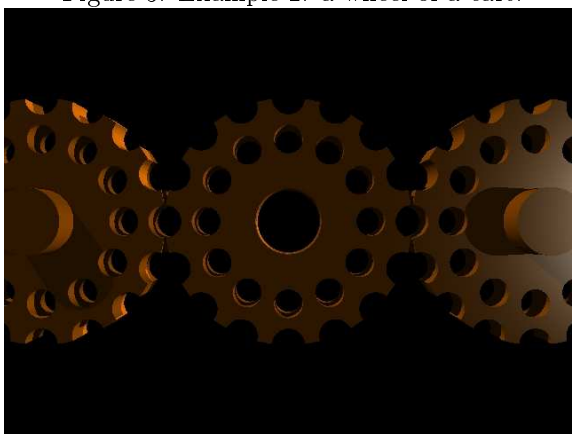

Figure 7: Example 4: gears.

examples have been built also for POV-Ray. In the third and fourth column are listed the intersection tests with primitive CSG objects and 3D bounding box, respectively. The last two columns have to be considered only for POV-Ray; POV-Ray uses a variety of systems to speed up ray-object intersection tests. The primary system uses a hierarchy of nested bounding boxes. Additionally, POV-Ray adopts systems known as *Vista Buffer* and *Light Buffer* to further speed the rendering process up. The vista buffer is created by projecting the bounding box hierarchy onto the plane of view and determining the rectangular areas that are covered by each element in the hierarchy. Only those objects whose rectangles enclose a given pixel are tested by the primary viewing ray. The light buffer is created by enclosing each light into an imaginary box and projecting the bounding box hierarchy onto each of its six sides. The tests concerning POV-Ray were made with all optimizations turned on.

In Table 2 computation times of Algo 1 and Algo 2 are reported. We do not list computation times obtained with POV-Ray because it uses different and optimized procedures to test the intersections. We use just one routine to test the intersection between a ray and a generic quadric; this has the advantage that our code is very short but may be inefficient at run time. On the other hand, POV-Ray uses specific intersection routines for each primitive object. The number of primitive CSG objects is listed in the second column of Table 2, while in the next column the number of lights can be found. The number of rectangles for the observer and for the lights is listed in the next columns. The last two columns report the computational times for Algo1 and Algo 2. Other statistics concerning memory requirements and traced points can be found in Table 3. Memory requirements for data structures are listed in the second, third and fourth column, respectively; traced points during the rendering process are reported in the last three columns.

Table 1: Intersection tests.

Ex.	Algorithm	Int. Ob.	Int. BB	Int. Light Buf.	Int. Vista Buf.
molecule	Algo 1	611,708	-	-	-
	Algo 2	4,106,860	7,745,915	-	-
	POV-Ray	321,880	469,990	1,578,577	1,489,570
wheel	Algo 1	591,228	-	-	-
	Algo 2	2,273,803	4,143,395	-	-
	POV-Ray	553,859	382,675	904,228	746,410
ball bearing	Algo 1	4,956,484	-	-	-
	Algo 2	9,853,671	19,226,892	-	-
	POV-Ray	15,353,438	541,098	699,265	776,538
gears	Algo 1	18,000,703	-	-	-
	Algo 2	22,742,628	44,825,967	-	-
	POV-Ray	35,743,511	466,283	653,340	589,922

Table 2: Computational times.

Ex. #	*CSG* objects	lights	rect. observer	rect. lights L1	L2	L3	Algo 1 [min:sec]	Algo 2 [min:sec]
1	20	3	251	261	236	265	0':37"	16':13"
2	12	3	114	92	160	140	0':35"	8':52"
3	61	3	453	431	229	434	3':47"	42':28"
4	93	3	11	19	22	22	19':47"	93':45"

6 Remarks

The statistics reported in section 5 show our method may reduce the number of intersection tests and, hence, can improve the rendering process. Let us consider only the intersection tests with primitive CSG solids. For the first example, our algorithm executes about the double of the intersection tests performed by POV-Ray; on the other hand, POV-Ray executes a very large number of vista and light buffer tests. These tests are faster than ones necessary to check a ray/object intersection but it has to be considered that their number is, for the example 1, of an order of magnitude larger. For the second example the performance of Algo 2 and POV-Ray are quite similar, while in the last two examples Algo 2 reduces up to one third the intersection tests. In these two examples, where the difference operation is widely employed, the vista and light buffer techniques do not speed up the ray tracing as for the first two examples where the most used operation is the union.

If compare Algo 1 with Algo 2, which uses only bounding box hierarchies, we speed up the rendering process in the best case of 26 times (example 1) and in the worst case of 3 times (example 4). It has to be pointed out that the scenes have been built manually and no post-processing step has been done to arrange the CSG trees in an optimized way. The performance gain is achieved by considering only a little portion of the entire database for intersection tests, instead of the whole CSG tree. In this way, we may save the unnecessary and computational intensive tests (in particular, the difference operation requires a high computational time); moreover, an automatic reorganization of the CSG sub-trees is executed, since primitive objects of a CSG sub-tree have a strong spatial locality. This overcomes the problem of obtaining good hierarchical organizations of the bounding volume approach. The organization of a CSG tree heavily affects the performance of ray tracers adopting standard optimizations. In Table 4 are listed the intersection tests executed with POV-Ray arranging the CSG tree of example 4 in two alternative ways. With the first arrangement of the tree the performance of POV-Ray are quite similar to our algorithm, while in the second case POV-Ray **automatically** disables all optimizations (i.e. bounding box hierarchies and light and vista buffer); in this case Algo 1 reduces the intersection tests to about one fifth. The performance of our algorithm is in-

Table 3: Memory requirements and traced points.

Ex. #	Algo 1 kbytes	Algo 2 kbytes	POV-Ray kbytes	Algo 1 points	Algo 2 points	POV-Ray points
1	347	35	142	134362	307200	307200
2	162	21	138	86844	307200	307200
3	478	108	295	145354	307200	307200
4	763	163	320	178839	307200	307200

Table 4: Intersection tests for two alternative organizations of the example 4.

Ex.	Int. Ob.	Int. BB	Int. Light Buf.	Int. Vista Buf.
gears	22,205,461	517,238	842,484	856,596
	94,945,524	-	-	-

dependent of tree organization and we obtain always the same results (already reported in Table 1).
Furthermore, we trace rays only into zones where certainly there are some objects. The statistics of traced pixels for different examples are given in Table 3; we can see that in the best case (example 2) only one third of the rays considered with a standard ray caster has to be traced. This may be an advantage for scenes where the objects do not affect a wide part of the plane of view.
On the other hand, our technique presents three drawbacks:

1. Our implementation requires more memory than a standard ray caster. In the worst case (example 4) our algorithm needs 763 kbytes against 163 kbytes used by a bounding volume approach. Anyway, memory requirement of our algorithm can not be considered critical for a graphic system, and only scenes with several thousands objects could be a problem.

2. With our strategy we need planes where to project 3D bounding boxes by using light sources as centers of projection. This constrains to place the lights outside the scenes, since only in this way we can be sure to find a projection plane for each light. This drawback might be overcome using the light buffer technique [18]; when a plane of projection can not be found for a given light source, the same technique used by POV-Ray can be employed.

3. Our algorithm is more complex than a standard strategy.

Furthermore, our algorithm might be concurrently used together with other acceleration techniques. For instance, the zz-buffer algorithm [19], or similar, could be employed to sort along the Z axis the primitive CSG objects of each sub-tree. Our methodology allows us to identify for each primary and shadow ray the CSG sub-tree to be checked for intersection, then, each other optimization should be applied only for the CSG sub-tree and for the area of the projection plane associated with it, instead of considering the whole CSG model and the entire screen.
Moreover, it is important to point out that also the rectangles of the observer could be rendered in parallel and each processor should load into memory just the portion of the database related to the rectangle being processed.

6.1 Evaluation of complexity

To evaluate the complexity of our algorithm we have to consider three phases:

- determination of the 3D bounding boxes at the root level;
- 2D projection of the 3D boxes at the root level;
- decomposition of the projections into non overlapping rectangles.

Let us denote with the capital letters G, N and P the number of primitive CSG objects, the number of 3D boxes at the root level, and the number of projections, respectively. For sake of simplicity, let us assume to consider a CSG with only union operators; in the worst case, the problem to find the number

of 3D boxes at the root level is linear with the number of objects $N = O(G)$, since each union operation produces a number of resulting boxes equal to the sum of the boxes of the operands. The problem of computing the 2D projections is linear with the number of 3D boxes at the root level, (i.e. $P = O(N)$) since the 2D projections can be obtained by analyzing, in sequence, the N bounding boxes. Finally, the problem of decomposition of the projections in non overlapping rectangles is quadratic $O(P^2)$, since each projection has to be checked for intersection with all the other ones. From these results, it follows that in the worst case the complexity is $O(G^2)$.

A general, accurate and tight evaluation of the complexity could be extremely difficult in the event of considering also intersection and difference operators and this is mainly due to the evaluation of the first phase (determination of 3D boxes). In fact, it can be observed that the decomposition in non overlapping rectangles is always a problem of quadratic complexity $O(P^2)$ and the projection always depends in linear way with the number of 3D boxes at the root level $P = O(N)$. On the other hand, spatial coherence among the objects strongly affects in an unpredictable way, even in the worst case, the dependency between primitive CSG objects and the 3D boxes at the root level. Therefore, the evaluation of the complexity of the first phase is strongly related to the typology of the scene (i.e. how the objects are placed in the 3D space) and it could be extremely difficult to provide a general and tight bound.

However, the examples we have considered in section 5 show that the number of rectangles grows, in the worst case, (i.e. example 2) about as the square of the object number, i.e. to the order of complexity of a tree with only union operators. For the considered examples we observe that the time spent to obtain the non overlapping rectangles is negligible when compared with the rendering time, since in the worst case (example 4) the rectangle computation requires 75 seconds which amount to about the 6% of the entire rendering time. However, in scenes consisting of several thousands objects, this step might require a computational time larger than the rendering time itself. Anyway, with the CSG representation, very complex scenes can be often obtained with several hundreds objects and, in these cases, it could be reasonable to expect that the rectangle computation should not have a considerable impact on the computational time.

7 Conclusion

In this paper a new method of spatial subdivision for fast rendering of CSG scenes has been presented. We provide a new definition of bounding box; in this way, we can consider bounding boxes containing the CSG sub-tree to be tested when a ray strikes the box. The considered examples show that we may reduce the intersection tests up to one third of the ones performed by a ray caster adopting standard optimizations.
Our strategy uses bounding volumes to surround the objects, in this way, tight bounds can be obtained; moreover, we split the screen into a set of non overlapping rectangles, associating a CSG sub-tree with each of them. With this approach just a portion of the whole database is considered for intersection tests. Bounding box group technique offers an automatic reorganization of the CSG hierarchy, in fact, the objects belonging to the CSG sub-trees have a strong spatial locality. In this way, the performance of our algorithm are not affected by the CSG tree organization, on the other hand, standard optimization may be heavily affected by the tree organization.

Acknowledgments

We are grateful to Ing. Enrico Porta for helping us in developing our software and to the anonymous reviewers for their helpful comments.

References

[1] Whitted, T. (1980) An Improved Illumination Model for Shaded Displays. CACM, **23**, 343-349.

[2] Rubin, S. M. and Whitted, T. (1980) A 3-Dimensional Representation for Fast Rendering of Complex Scenes. Proc. Siggraph, ACM Computer Graphics, 110-116.

[3] Kay, T. L. and Kajiya, J. T. (1986) Ray Tracing Complex Scenes. Proc. Siggraph, ACM Computer Graphics, 269-277.

[4] Goldsmith, J. and Salomon, J. (1987) Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE* Computer Graphics & Applications, **7**, 14-20.

[5] Glassner, A. S. (1984) Space Subdivision for Fast Ray Tracing. *IEEE* Computer Graphics & Applications, **4**, 15-22.

[6] Wyvill, G. Kunii, T. L. and Shirai, Y. (1986) Space Division for Ray Tracing in CSG. *IEEE* Computer Graphics & Applications, **6**, 28-34.

[7] Coquillart, S. (1985) An Improvement of the Ray-Tracing Algorithm. Proc. Eurographics, Elsevier, 77-88.

[8] Requicha, A. A. G. and Voelcker, H. B. (1982) Solid Modeling: a historical summary and contemporary assessment. *IEEE* Computer Graphics & Applications, **2**, 9-22.

[9] Roth, S. D. (1982) Ray Casting for Modeling Solids. Computer Graphics and Image Processing, **18**, 109-144.

[10] Gervautz, M. (1986) Three Improvements of the Ray Tracing Algorithm for CSG Trees. Computer and Graphics, **10**, 333-339.

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

[12] Cameron, S. (1991) Efficient Bounds In Constructive Solid Geometry. *IEEE* Computer Graphics & Applications, **11**, 68-74.

[13] Cameron, S. and Yap, C. K. (1992) Refinement Methods for Geometric Bounds in Constructive Solid Geometry. ACM Transaction On Graphics, **11**, 12-39.

[14] Mazzetti, M. and Ciminiera, L. (1994) Computing CSG tree boundaries as algebraic expressions. Computer-Aided Design, **26**, 417-425.

[15] Sanna, A. and Montuschi, P. (1995) On the Computation of Groups of Bounding Boxes For Test of Objects Intersection. International Phoenix Conference on Computers and Communications, 684-690.

[16] Magnenat-Thalmann, N. and Thalmann, D. (1987) *IMAGE SYNTHESIS: Theory and Practice*, Springer-Verlag Tokyo.

[17] POV-Ray is a freeware ray tracer. Source code and documentation can be downloaded from the web site http://www.povray.org.

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

[19] Salesin, D. and Stolfi, J. (1990) Rendering CSG Models with a ZZ-Buffer. Proc. Siggraph, ACM Computer Graphics, 67-76.

[20] Sanna, A. and Montuschi, P. (1995) Spatial Bounding of Complex CSG Objects. IEE Proc.-Comput. Digit. Tech., **142**, 431-439.

Appendix: basic operators

In this section we provide the definition of the new operators on BBGs (which are different from those given in [15]). Given two BBGs $A = \{(a_1, ST(a_1)), (a_2, ST(a_2)), \ldots, (a_n, ST(a_n))\}$ and $B = \{(b_1, ST(b_1)), (b_2, ST(b_2)), \ldots, (b_m, ST(b_m))\}$, the operators are defined as follows:

Intersection:

$$\begin{aligned} D &= A \cap B = \{(a_1 \cap b_1, ST(a_1) \cap ST(b_1)), \\ &\quad (a_2 \cap b_1, ST(a_2) \cap ST(b_1)), \ldots, \\ &\quad (a_n \cap b_1, ST(a_n) \cap ST(b_1)), \ldots, (a_n \cap b_m, \\ &\quad ST(a_n) \cap ST(b_m))\} \end{aligned} \quad (1)$$

With respect to the intersection operator defined in [15], a CSG sub-tree is associated with each bounding box belonging to the resulting group.

Union: the union operation between two BBGs can be divided into two steps. The first step is the union operation similar to the one defined in [15]:

$$\begin{aligned} D &= A \cup B = \{(a_1, ST(a_1)), \ldots, (a_n, ST(a_n)), \\ &\quad (b_1, ST(b_1)), \ldots, (b_m, ST(b_m))\} \end{aligned} \quad (2)$$

In the second step, the intersections among the bbs have to be detected and eliminated. In three dimensions, it can be seen that, in general, when two bbs overlap they can be decomposed into seven new bbs: one intersection bb plus "other" six bbs. On the other hand, in two dimensions an intersection leads to one intersection bb plus four "other" bbs. The CSG sub-tree related to the intersection bb is obtained as a union of the CSG sub-trees of the two intersecting bbs, while the other four bbs have a sub-tree equal to the one of the bb from which they have been obtained. An example in two dimensions is presented in Fig. 2.

A similar process has to be carried out if a bounding box encloses another bb. The result of the union operation is a BBG where the elements do not have reciprocal intersections and, in general, $k > m + n$:

$$D' = \{(d_1, ST(d_1)), (d_2, ST(d_2)), \ldots, (d_k, ST(d_k))\} \quad (3)$$

The intersections among boxes may be all detected at the root level of the CSG trees, instead of testing for the intersection after having performed each union operation. In this way, the union operation is reduced to the step 1, but a supplementary step at the root level has to be executed.

7.1 Difference operator

Given two objects A and B, the difference may be obtained as: $D = A \cap \overline{B}$, where the symbol $\overline{B}$ denotes the complement of the object B, i.e., the set of the points of the space not belonging to B. The difference between the bounding entities of A and B must be examined carefully, since the relationship $bb(D) = bb(A) \cap \overline{bb(B)}$ does not hold. We have to consider the following problem: the complement of a bounding box is not, in general, a bounding box [20]. A trivial but inefficient way of overcoming this problem, that is known in literature, is simply based on the assumption that in the difference $D = A - B$, the object B is smaller than A, therefore, the bounding box of D can be defined as being equal to the bounding box of the larger object, i.e., A. This technique has to be used in each algorithm that considers just one bounding box for each node of a CSG tree ([12],[14]). A more efficient method to solve the difference problem has been proposed in [20]. The concept of *inner* bounding box group is introduced in that paper. The inner bounding boxes do not encapsulate the objects like the *external* boxes, but they enclose only points which certainly belong to the objects. In this way, a new bounding entity can be defined for each node of a CSG tree. The new bounding entity is made up of two groups: an external group and an inner group $\{EBBG; IBBG\}$. In [20], the difference between two BBGs A and B can be computed as the "intersection" between the external group of A and the complement of the inner groups of B: $D = EBBG_A * \overline{IBBG_B}$. We can not use the intersection operator $\cap$ above defined since the CSG sub-trees associated with the inner boxes have to be subtracted.

In this section we provide a new definition of the difference operation, such that it can be used with the boxes having associated a CSG sub-tree. In the following we use the subscript $_{in}$ to distinguish the inner from the external boxes. A new definition of complement is provided:

Complement:
If $B = \{(b_{in1}, ST(b_{in1})), \ldots, (b_{inm}, ST(b_{inm}))\}$ is an IBBG, then $\overline{B}$ can be computed as:

$$\overline{B} = \{(\overline{b_{in1}}, ST(b_{in1})) \cap (\overline{b_{in2}}, ST(b_{in2})) \cap \ldots \cap (\overline{b_{inm}}, ST(b_{inm}))\} \quad (4)$$

Observe that the complement of a single bounding box is, in general, a group of boxes. Now we can define the difference operator between two bounding box groups:

Difference: The difference $(A - B)$ between $A = \{(a_1, ST(a_1)), \ldots, (a_n, ST(a_n)); (a_{in1}, ST(a_{in1})), \ldots, (a_{inm}, ST(a_{inm}))\}$ and $B = \{(b_1, ST(b_1)), \ldots, (b_k, ST(b_k)); (b_{in1}, ST(b_{in1})), \ldots, (b_{inz}, ST(b_{inz}))\}$ is computed as:

$$\begin{aligned}
D = {} & A - B = \{(a_1, ST(a_1)), \ldots, (a_n, ST(a_n)); \\
& (a_{in1}, ST(a_{in1})), \ldots, (a_{inm}, ST(a_{inm}))\} - \\
& \{(b_1, ST(b_1)), \ldots, (b_k, ST(b_k)); (b_{in1}, \\
& ST(b_{in1})), \ldots, (b_{inz}, ST(b_{inz}))\} = \\
& \{(a_1, ST(a_1)), \ldots, (a_n, ST(a_n)); (a_{in1}, \\
& ST(a_{in1})), \ldots, (a_{inm}, ST(a_{inm}))\} - \\
& \{(b_{in1}, ST(b_{in1})), \ldots, (b_{inz}, ST(b_{inz}))\} = \\
& = \{(a_1, ST(a_1)), \ldots, (a_n, ST(a_n)); (a_{in1}, \\
& ST(a_{in1})), \ldots, (a_{inm}, ST(a_{inm}))\} * \\
& *\{\overline{(b_{in1}, ST(b_{in1})), \ldots, (b_{inz}, ST(b_{inz}))}\} = \\
& = \{\{(a_1, ST(a_1)), \ldots, (a_n, ST(a_n))\} * \\
& *\{(\overline{b_{in1}}, ST(b_{in1})), \cap \ldots \cap (\overline{b_{inz}}, ST(b_{inz}) \\
&)\}; \{(a_{in1}, ST(a_{in1})), \ldots, (a_{inm}, \\
& ST(a_{inm}))\} * \{(\overline{b_{in1}}, ST(b_{in1})) \cap \ldots \cap \\
& (\overline{b_{inz}}, ST(b_{inz}))\}\} = \{\{(a_1 \cap \overline{b_{in1}}, ST(a_1) \\
& -ST(b_{in1})), \ldots, (a_1 \cap \overline{b_{inz}}, ST(a_1) - \\
& ST(b_{inz})), \ldots, (a_n \cap \overline{b_{inz}}, ST(a_n) - \\
& ST(b_{inz}))\}; \{(a_{in1} \cap \overline{b_{in1}}, ST(a_{in1}) - \\
& ST(b_{in1})), \ldots, (a_{inm} \cap \overline{b_{in1}}, ST(a_{inm}) - \\
& ST(b_{in1})), \ldots, (a_{inm} \cap \overline{b_{inz}}, ST(a_{inm}) - \\
& ST(b_{inz}))\}\} \quad (5)
\end{aligned}$$

An example of difference in 2D is shown in Fig. 8. The first picture shows the two objects of whose the

difference has to be computed, while in the second picture external boxes and inner box of the object B are depicted. The next two pictures show the BBG complement of the inner box of B and, finally, the BBG result is presented.

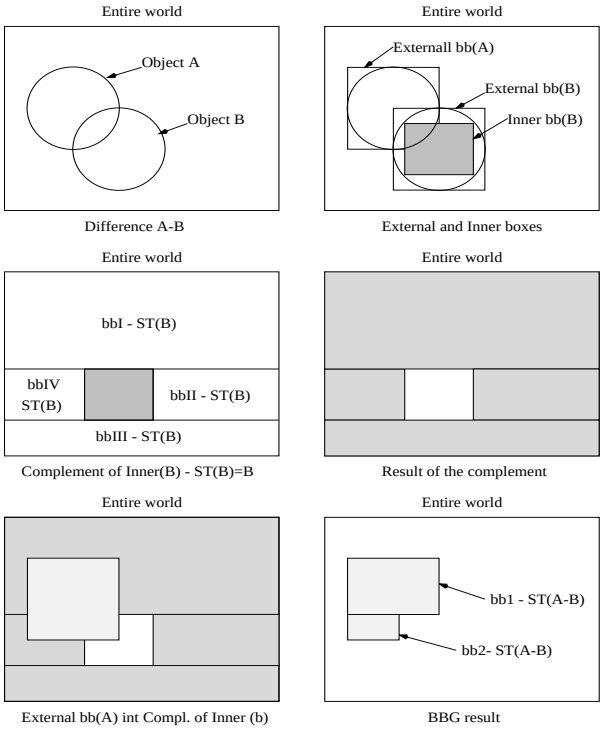

Figure 8: Example of difference in 2D.