Exploiting Shadow Coherence in Ray Tracing

Andrew Pearce

Alias Division, Alias Research Inc.
110 Richmond Street East
Toronto, Ontario
M5C 1P1

David Jevans

Apple Computer Inc.
20705 Valley Green Dr.
Cupertino, California
95014

1. Abstract

We present a method, based on Haines' *shadow caches*, for accelerating shadow ray calculations for ray tracing processes which use spatial subdivision and surface tessellation. Trees of *shadow voxel caches*, containing references to both objects and voxels, are used to quickly determine whether or not a surface lies in shadow. This method does not require an additional preprocessing stage before rendering, and its memory requirements are small.

2. Résumé

Nous présentons une méthode basée sur les *ombres en mémoire cache* de Haines, qui utilise la subdivision spatiale et la tessellation de surfaces afin d'accélerer le calcul de rayons d'ombre dans un programme de lancé de rayon. Des arbres de *voxels d'ombre en mémoire cache*, contenant des références aux objets ainsi qu'aux voxels, sont utilisés pour déterminer rapidement si une surface se trouve dans une région d'ombre. Cette méthode ne requiert pas d'étape additionelle de prétraitement avant le rendu de l'image, et ses besoins en mémoire sont faibles.

Keywords: Ray Tracing, Shadow Testing, Shadow Caching, Spatial Subdivision

3. Introduction

Shadow calculation is essential to the synthesis of realistic computer generated images. Ray tracing and radiosity are two commonly used rendering techniques that produce shadows as a natural part of the rendering process. Despite the recent advances in radiosity techniques [Cohe88], ray tracing is still the most practical solution for rendering scenes which contain specularly transmissive surfaces, and can be used in the calculation of radiosity form factors [Wall87, Wall89]. It is with this in mind that we examine the ray tracing shadow testing calculation.

Ray tracing [Whit80] is one of the simplest and most elegant rendering algorithms in existence. The ray tracing algorithm traces rays of light backwards from the eye, through the pixel grid of the screen, and into the scene where they reflect off or refract through surfaces and finally hit a diffuse surface or exit the bounds of the scene. Shadows are calculated by tracing a ray from each ray/surface intersection point toward each light source. If a shadow ray strikes an opaque surface before reaching the light source, then the intersection point is in shadow.

When testing a ray/surface intersection point to see if it lies in shadow, it is sufficient to know that some opaque object lies between it and the light source. This is a simpler problem than visible surface determination, where the first object that is intersected by the ray must be found, and a number of acceleration techniques have been developed.

The scenes we are interested in ray tracing contain thousands of parametric surface patches that are tesselated into meshes of triangles to facilitate and accelerate ray/patch intersection calculations [Swee 86, Snyd87]. Since spatial subdivision techniques [Glas84, Fuji86, Avro87, Snyd87, Jeva89] are widely used to accelerate the ray tracing process, it is desireable to take advantage of these spatial subdivision structures in a shadow testing acceleration algorithm. This paper presents such an algorithm.

4. Previous Work

Haines and Greenberg [Hain86] introduced the *light buffer* to quickly identify and reduce the number of potential shadowing objects. A light buffer is a cube surrounding a

light source, where each side of the cube is subdivided into regions which contain a list of the objects that could potentially block light travelling from the light source in the region's direction. When testing a ray/surface intersection point for shadowing, the region of each light buffer which contains the projected intersection point is queried. In many cases, trivial shadowing or nonshadowing can be determined without casting a shadow ray, substantially improving performance. The main drawback of the light buffer method is that it requires the creation of large data structures which are used only for shadow acceleration. The creation of these data structures entails preprocessing every object 6 times for each light source in the scene. This takes $N \times L \times 6$ operations, where N is the number of objects and L is the number of lights. As N becomes large, this approach becomes less desirable.

Eo and Kyung [Eo89] introduced a hybrid shadow testing scheme for ray tracing which combines *shadow polygons* [Crow77] with traditional ray tracing shadow testing techniques. The algorithm calculates shadow polygons, which define shadow volumes, for each light source and places them into the scene. When a ray is traced through the scene, a count is made of the number of shadow polygons that it intersects before hitting a surface. When a ray/surface intersection is found, the shadow polygon intersection count indicates whether or not the point lies within a shadow volume. At points where shadow determination is too difficult to determine via shadow polygons, the algorithm resorts to the traditional shadow testing method, and traces a shadow ray toward each light source. Creating shadow polygons requires knowledge of object silhouettes and requires special case shadow polygons to be stored in the scene, leading to extra intersection calculations for every ray, and the added costs of preprocessing and increased memory use. The complexity of creating shadow polygons for even simple patches is a limiting factor of this hybrid approach.

Woo and Amanatides [Woo90] introduced *voxel occlusion testing*, a shadow testing acceleration technique that makes use of uniform space subdivision grid structures. In a preprocessing step, each voxel of the uniform space subdivision grid is marked as **full, null** or **complicated occlusion** for each light source in the scene. The occlusion status of a voxel is calculated by projecting silhouette edges of objects from the point of view of a light source onto the voxel grid structure. Voxels that lie entirely inside or outside these silhouette edges are marked as **full** or **null occlusion**, and voxels that lie on a silhouette edge are marked as **complicated occlusion**. If a ray/surface intersection point lies in a voxel which is marked as **full occlusion**, then it is in shadow. If the intersection point lies in a voxel marked as **null occlusion**, it is not in shadow. When a ray/surface intersection point lies in a voxel that is marked as **complicated occlusion**, a shadow ray is traced toward the light source. If the shadow ray encounters a voxel that is marked as **full** or **null occlusion** as it travels toward the light source, then traversal can halt. As with the previous method, this technique requires the calculation of object silhouettes from the point of view of each light source. Further, the marking of voxels becomes complicated if the spatial subdivision structure is other than a uniform voxel grid, more so if lazy voxel subdivision is involved.

5. Shadow Caching

Haines [Hain86] proposed the *shadow cache* method to accelerate shadow testing without preprocessing or knowledge of object silhouettes. Each light source maintains a shadow cache, which stores a reference to the opaque object that last cast a shadow from the light source, or a **null** reference if the last shadow ray fired toward the light source did not encounter a shadowing object. Before a shadow ray is traced toward a light source, the object in the light source's shadow cache is tested for intersection. If the shadow ray intersects the object, then the object occludes the light source, and further processing is not required. If the shadow ray does not intersect the object, then it must be traced toward the light source until it hits the light source or an occluding object.

The shadow cache method takes advantage of the spatial coherence of a scene by assuming that successive shadow rays fired toward a light source will emanate from intersection points adjacent to previous intersection points, an assumption that is valid for scenes consisting largely of diffuse surfaces. This assumption may not be valid, however, if the scene contains specularly reflective or transparent objects, because successive shadow rays will generally not emanate from adjacent intersection points.

6. Shadow Voxel Caching

Our method is a two-part extension of Haines' shadow cache method. Since we are dealing with surface tessellations, the expected amount of object coherence is smaller than if we were dealing with polygonal or implicit surfaces which tend to be larger than triangles and cover more of the screen. To improve upon the small amount of object coherence, shadow caches are modified to additionally store a reference to the voxel which contains the cached occluding object. This *shadow voxel cache* technique is based on the assumption that if the cached object does not occlude the light source, it is likely that one of the objects in the same voxel does.

7. Shadow Voxel Cache Trees

To avoid incorrect shadow caches when shadow testing ray/surface intersection points of reflected and refracted rays, each light source maintains a binary tree of shadow voxel caches, where each node has a child for reflected rays and a child for refracted rays. When testing a point for shadowing, the tree level of the ray which spawned the shadow ray is used to index into the shadow voxel cache tree of the light source.

8. Drawbacks

The drawback of shadow cache methods is that the cached objects are always tested for intersection with a shadow ray, even if they do not lie near the path of the ray. This can happen when the spawning location, the intersection point that is being shaded, changes drastically from ray to ray. The

spawning location changes drastically when adjacent rays intersect objects at different locations in the scene; when the tracing of a new scanline begins; and when a highly curved surface causes reflected or refracted rays to scatter widely.

Part of the problem can be avoided by setting the shadow voxel caches to **null** at the begining of each new scanline, or by tracing scanlines in alternating directions, where even numbered scanlines sweep left to right across the screen and odd numbered scanlines sweep right to left.

9. Avoiding Multiple Checks of the Shadow Voxel Cache Objects

If a shadow ray does not intersect the cached object or any of the objects in the cached voxel, it is traced through the scene toward the light source. A potential waste of computation will occur if the shadow ray enters the voxel that is referenced by the light source's shadow voxel cache, since the ray will be retested for intersection with all of the objects in the voxel. A simple technique to avoid this duplication of computation [Arna87,Pear87,Aman87] is to assign a unique identification number to each ray so that objects will be tested for intersection with any given ray only once. Once an object is tested for intersection, the ray's number is stored in a *last ray* field in that object. Before testing an object for intersection, the ray's number and the *last ray* field of the object are compared. If the numbers match, then the object has already been encountered and tested for intersection with the ray and has been eliminated from the set of possible intersecting objects, and should not be retested.

To avoid testing the spawing object for intersection with a shadow ray, a common technique is to add a small epsilon value to the origin of the shadow ray along its direction of travel. An alternate technique for planar objects is to set the object's *last ray* field to the shadow ray's number before starting the shadow test.

Field	Size	Purpose
last_object	4 bytes	-> last object intersected at this level
last_voxel	4 bytes	-> the voxel which the last object was in
refraction_ray	4 bytes	-> another of these structures
reflection_ray	4 bytes	-> another of these structures

Table 1. *Shadow_Tree* Structure

10. Shadow Voxel Cache Tree Data Structure

Each light source must have a tree of shadow voxel cache structures. The shadow voxel cache data structure is small in size, and its fields are outlined in Table 1. If a reasonably deep shadow tree is allowed, for example a maximum of 10 bounces, the storage requirements of a shadow voxel cache tree on a 4 byte integer machine is 32 kilobytes. For most scenes a more reasonable maximum of 5 bounces requires less than 1 kilobyte per shadow voxel cache tree. We shall refer to a shadow voxel cache structure as a *shadow_tree* structure for the purposes of our example algorithm.

11. The Shadow Voxel Cache Tree Algorithm

Figure 1 gives an outline of the shadow voxel cache tree method in C pseudo. The shadow ray has been generated before the *check_shadowing* routine is called, but the ray/voxel traversal initialization does not occur until the *traverse_voxels_for_shadows* routine is called. Note that only opaque occluding objects are stored in shadow caches.

The routine **test_objs_in_voxel_for_shadows()** returns hit = TRUE on first affirmed intersection with an opaque object, and ignores transparent objects.

The routine **traverse_voxels_for_shadows()** intersects transparent and opaque objects and sorts the intersections for proper attenuation of the light intensity. If multiple objects are hit, then all of the objects intersected must be transparent, and the object returned is arbitrarily the first one. Tracing of the shadow ray halts once the light source has been reached.

```
float check_shadowing(ray, light, path, Spawning_ray_level)
RAY_REC    *ray;    /* Ray from shading point */
                    /*       to light source */
LIGHT_REC *light;   /* The light source we are */
                    /*         interested in. */
int        path;    /* Bit table describing current */
                    /* position in the vision ray tree. */
int        Spawning_ray_level; /* Level of the ray */
                               /* spawning the shadow ray. */
{
    unsigned int  Mask;
    shadow_tree   *cache;

    cache = light->cache_tree;
    Mask = 0x01;
    /* If the spawning ray's level is 0 (primary ray), */
    /* then we use the head of the cache_tree. */
    for (i = 0; i < Spawning_ray_level; ++i) {
        if (Mask & path) cache = cache->refraction_ray;
        else             cache = cache->reflection_ray;
        Mask = Mask << 1; /* Shift mask left 1 bit */
    }

    if (cache->last_object != NULL) {
        /* intersect_object() marks object as having */
        /* been intersected by this ray. */
        hit = intersect_object( ray, cache->last_object,
                                &object);

        if (hit) {
            return(1.0); /* full shadowing */
        }
        cache->last_object = NULL; /* object was not hit */

        if (cache->last_voxel != NULL) { /* implied !hit */
            if ( hit = test_objs_in_voxel_for_shadows
                    ( ray, cache->last_voxel, &object) ) {
                if ( hit_not_in_voxel( ray, object ) )
                    cache->last_voxel = NULL;
                cache->last_object = object;
                return(1.0);
            }
            /* voxel did not supply a hit */
            cache->last_voxel = NULL;
        }
    }
```

```
    if ( !hit = traverse_voxels_for_shadows(ray, &object,
                              &voxel, &shadow_percent)
        || (object->transparency_value > 0.0)) {
        cache->last_object = NULL;
        cache->last_voxel  = NULL;
    }
    else {
        /* The object was NOT transparent, */
        /* cache the info. */
        cache->last_object = object;
        cache->last_voxel  = voxel;
    }
    return ( shadow_percent );
}
```

Figure 1. Shadow Voxel Cache Algorithm

12. Creation of the Ray Path Bit Table

Indexing into the shadow voxel cache tree is aided by a bit table, the ***path*** variable in Figure 1, which represents the path that the spawning ray took in terms of reflection and refraction bounces. A set bit in the bit table indicates that the ray refracted at that level, while an unset bit indicates a reflection. Each new ray that is spawned updates the bit table to mark its path, as shown in Figure 2.

Refraction Ray:

```
Mask = 0x01 << Spawning_ray_level;
/* Turn on correct bit. */
path = path | Mask;
trace( /* refraction ray */ );
path = path & ~Mask;
```

Reflection Ray:

```
Mask = 0x01 << Spawning_ray_level;
path = path & ~Mask;
trace( /* reflection ray */ );

/* NOTE: */
/*  << is a bitwise left shift of the        */
/*     left operator by right operator bits */
/*   | is a bitwise logical OR operation     */
/*   & is a bitwise logical AND operation    */
/*   ~ is a bitwise negation operation       */
```

Figure 2. Creating the *Path* Bittable for Secondary Rays

13. Results

All of the test scenes were rendered on an unloaded SGI VGX, a 33MHz MIPS processor based machine with 64Mb of real memory. A vendor specific profiler named *pixie* was used to count machine cycles in various routines. The cycle count is accurate and does not vary due to processor load. All times and speedups in Table 2 are in terms of machine cycles. The speedups reported are with respect to total number of cycles for the entire ray tracing operation including reading the scene description files, and tessellation of objects. All images except Spanky were calculated at 640 by 486 pixels with 9 samples per pixel (some samples shared between adjacent pixels). Spanky was calculated at 512 by 512 pixels.

In Table 2, the *Number of Rays* represents the total number of rays, including shadow rays.

The *Shadow Cache Success Rate* reports the percentage of times a shadow cache object provided positive shadowing when tested for intersection.

The *Voxel Cache Success Rate* shows the percentage of times the cached voxel provided positive shadowing when the shadow cache failed.

The *Voxel Cache Speed Up Over Shadow Cache Alone* describes the percentage fewer cycles which the voxel cache method used, and *Voxel Cache Tree Speed Up Over Voxel Cache Alone* shows the percentage cycles saved when a tree of voxel caches is employed (a negative value indicates a slow down). Note that these speedups are with regard to the entire ray tracing process, not just with regards to shadow testing.

Scene Description	64,000 jittered polygons (0.2) (sparse)	64,000 jittered polygons (1.0) (dense)	Spanky the Drum	Chair Room
Number of Triangles	551,408	551,408	59,692	40,937
Number of Shadow Casting Lights	15	15	6	4
Number of Rays	11,324,318	8,427,904	2,230,193	19,269,028
Shadow Cache Success Rate (Haines)	50.7%	90.9%	47.8%	2.5%
Voxel Cache Success Rate	23.4%	39.3%	76.9%	2.76%
Voxel Cache Speed Up Over Shadow Cache Alone (entire process)	1.04%	3.6%	5.53%	3.62%
Voxel Cache Tree Speed Up Over Voxel Cache Alone (entire process)	-0.003%	-0.00002%	2.7%	7.27%

Table 2. Timing Results

Four test scenes were used, as shown in Figures 3, 4, 5 and 6. Two of the scenes were jittered distributions of polygons within a closed area, while the other two were more typical of scenes used in computer animation and product evaluation. The jittered distribution scenes contained a 40x40x40 array of polygons. Each polygon is approximately a unit square and is tessellated into 8 triangles. The polygons

are initially placed at unit distances from each other. A Gaussian pseudo-random number generator is then called to jitter the location of the center of a polygon in each of the three orthogonal axes by a value in the range of −1.0 to 1.0, and the orientation by values of −90 degrees to 90 degrees. In the sparse jittered scene, the polygons are scaled by 0.2, and in the dense scene they are left at their original size.

For all four of the test scenes, the addition of a voxel cache to the shadow cache improves shadow testing performance. The jittered scenes contain no reflective or refractive objects, therefore maintaining a tree of shadow voxel caches does not improve performance over a single shadow voxel cache, in fact the performance is slightly worse in the sparse scene due to the overhead of maintaining and accessing a tree of caches instead of a single cache at each light source

When rendering the snare drum scene (*Spanky*), the use of a tree of shadow voxel caches results in somewhat better performance than the use of a single shadow voxel cache at each light source, due to the small number of reflective surfaces in the scene. Because the chair room scene is highly specular, however, maintaining a tree of shadow voxel caches yields significant performance gains over maintaining a single shadow voxel cache with each light source.

14. Future Work

If the ray/surface intersection point and a shadow ray direction vector are stored at each level of the voxel cache tree, then a tolerance check could be performed to avoid testing object and voxel caches which are not along the path of the current shadow ray.

A further improvement to the algorithm, suggested by Andrew Woo, is to only maintain a voxel pointer in each cache, and to move the shadowing object to the front of a voxel's object list. In addition to simplifying the shadow cache data structure, this process ensures that the most recently intersected shadowing objects are at the head of a voxel's object list. This approach will be examined in a future publication.

15. Conclusion

A method has been presented to accelerate shadow testing for ray tracing. The technique performs well when the scene contains specular surfaces, and can utilize an existing spatial subdivision structure to increase performance when rendering tessellated surfaces. It is easy to implement, and can be used in conjunction with other shadow testing acceleration techniques.

16. Acknowledgements

We would like to thank Andrew Woo for his encouragement and helpful suggestions, Stanley Liu and Gary Mundell for modeling the room and chairs in the Chair Room image, and Peter Schoeler and Marie France Roy for providing the french translation of the abstract. Silicon Graphics is a registered trademark of Silicon Graphics, Inc.

17. References

[Aman87] Amanatides, J. and A. Woo, "A Fast Voxel Traversal Algorithm for Ray Tracing," EuroGraphics '87, pp. 1-10, 1987.

[Arna87] Arnaldi, B. T. Priol, and K. Bouatouch, "A New Space Subdivision Method for Ray Tracing CSG Modelled Scenes," The Visual Computer, 3(2), pp. 98-108, 1987.

[Avro87] Avro, J., and D. Kirk, "Fast Ray Tracing by Ray Classification," Computer Graphics (SIGGRAPH '87), 21(4), pp. 55-64, 1987.

[Cohe88] Cohen, M., S. E. Chen, J. Wallace, and D. Greenberg, "A Progressive Refinement Approach to Fast Radiosity Image Generation," Computer Graphics (SIGGRAPH '88), 22(4), pp. 75-84, 1988.

[Crow77] Crow, F., "Shadow Algorithms for Computer Graphics," Computer Graphics (SIGGRAPH '77), 11(2), pp. 242-248, 1977.

[Eo89] Eo, D., and C. Kyung, "Hybrid Shadow Testing Scheme for Ray Tracing," Computer Aided Design, 21(1), pp. 38-48, January 1989.

[Fuji86] Fujimoto, A., T. Tanaka, and K. Iwata, "ARTS: Accelerated Ray-Tracing System," IEEE Computer Graphics and Applications, 6(4), pp. 16-26, April 1986.

[Glas84] Glassner, A., "Space Subdivision for Fast Ray Tracing," IEEE Computer Graphics and Applications, 4(10), pp. 15-22, October 1984.

[Jeva89] Jevans, D. and B. Wyvill, "Adaptive Voxel Subdivision for Ray Tracing," Proc. Graphics Interface '89, pp. 164-172, 1989.

[Hain86] Haines, E. A., and D. P. Greenberg, "The Light Buffer: A Ray Tracer Shadow Testing Accelerator," IEEE Computer Graphics and Applications, 6(9), pp. 6-16, September 1986.

[Pear87] Pearce, A., "An Implementation of Ray Tracing Using Multiprocessing and Spatial Subdivision," Master's Thesis, University of Calgary, Dept. of Computer Science, 1987.

[Snyd87] Snyder, J., and A. Barr, "Ray Tracing Complex Models Containing Surface Tessellations," Computer Graphics (SIGGRAPH '87), 21(4), pp. 119-128, 1987.

[Swee86] Sweeney, M., and R. Bartels, "Ray Tracing Free-Form B-Spline Surfaces," IEEE Computer Graphics and Applications, 6(2), pp. 41-49, February 1986.

[Wall87] Wallace, J., M. Cohen, and D. Greenberg, "A Two-Pass Solution to the Rendering Equation: A Synthesis of Ray Tracing and Radiosity Methods," Computer Graphics (SIGGRAPH '87), 21(4), pp. 311-320, 1987.

[Wall89] Wallace, J., K. Elmquist, E. Haines, "A Ray Tracing Algorithm for Progressive Radiosity," Computer Graphics (SIGGRAPH '89), 23(3), pp. 315-324, 1989.

[Whit80] Whitted, T., "An Improved Illumination Model for Shaded Display," Communications of the ACM, 23(6), pp. 343-349, June 1980.

[Woo90] Woo, A., and J. Amanatides, "Voxel Occlusion Testing: A Shadow Determination Accelerator for Ray Tracing," Proc. Graphics Interface '90, pp. 213-220, 1990.

Figure 3. Sparse jittered polygons.

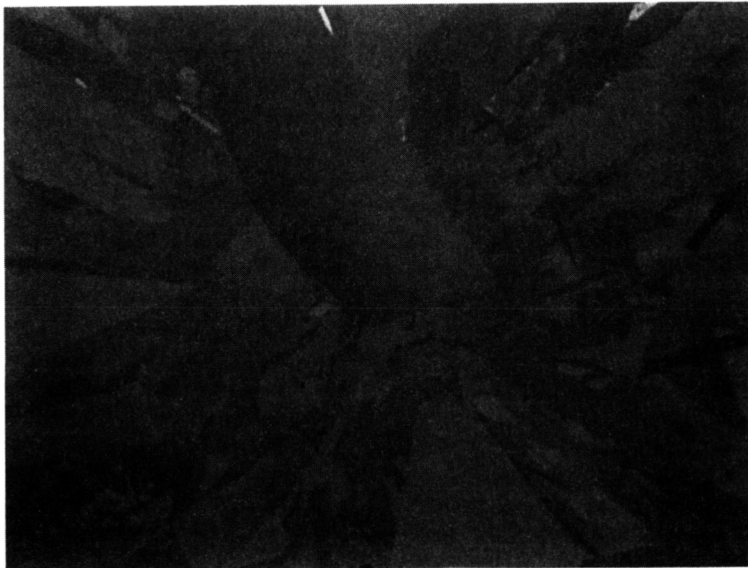

Figure 4. Dense jittered polygons.

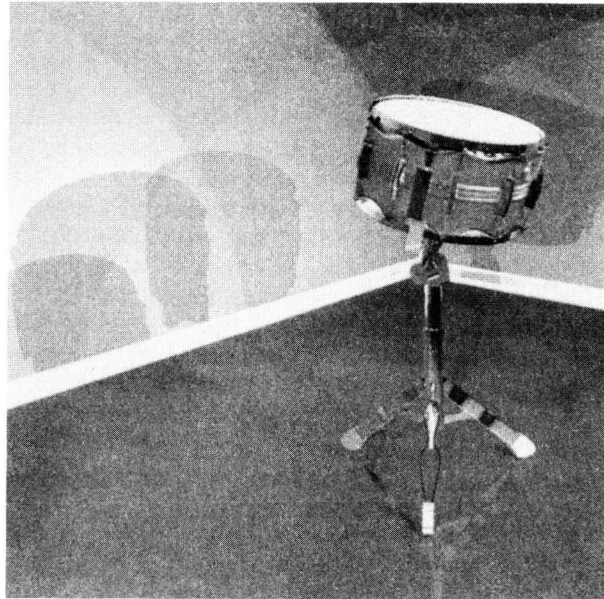

Figure 5. Spanky the Snare Drum

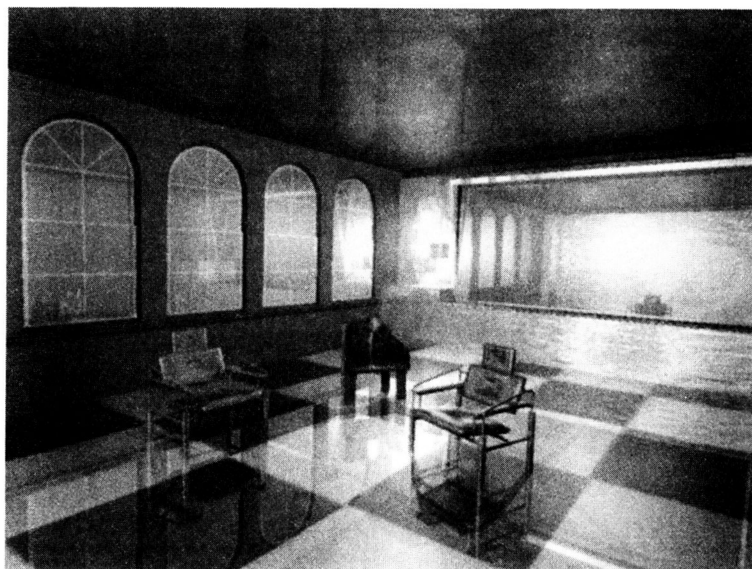

Figure 6. Chair Room