Mikio Shinya
Marie-Claire Forgue

NTT Human Interface Laboratories, 1-2356, Take, Yokosuka, Kanagawa 238-03, Japan

Interference Detection through Rasterization

SUMMARY **Interference detection is a useful technique, but it is also generally time-consuming. In this paper, a new type of interference detection algorithm is proposed for real-time interference detection. The algorithm first rasterizes the projection of the target objects and calculates the z-values, just as done by the z-buffer visible surface algorithm. For interference detection, all z-values and pointers to the corresponding faces of objects are saved in a z-list for each pixel. Sorting the z-list against the z-values allows the detection of overlapping objects in the z-direction at each pixel position and, thus, finds interfering faces by referring to the face pointers in the z-list.**

The algorithm is simple and easy to implement. Its computational complexity is directly proportional to the number of polygons, and, in addition, standard graphics hardware can be used to greatly accelerate execution. Another advantage is that the algorithm can be applied to all 'ray-traceable' objects, including algebraic surfaces, and procedurally defined objects; traditionally these were not suitable subjects for interference detection.

The algorithm is implemented on a graphics workstation using a standard graphics library. Interference detection at a practical interaction speed is achieved for complicated objects such as polyhedra with thousands of polygons. The algorithm can be used in two ways: for inexpensive interference detection, and as an efficient culling method for more precise collision/interference detection algorithms.

KEY WORDS: Interference/collision detection Rasterization Z-buffer algorithm Robot simulator Computer animation

INTRODUCTION

Interference detection is an important issue in many fields, e.g. computer animation, robotics, computer-aided geometric design and artificial reality. Many studies have been made to detect interference through geometric calculations, and existing algorithms can be classified into two main approaches: calculating object intersection,[1–4] and calculating the distance between objects.[5,6] However, there are four serious problems with either approach.

1. *Cost:* complex objects are computationally expensive, typically $O(n^2)$, for the number of polygons, n.
2. *Robustness:* because of numeric errors, it is difficult to build a program robust for all conditions, e.g. for a vertex touching another vertex, an edge on another surface, and so on.
3. *Applicable surfaces:* usually only particular kinds of surfaces can be treated, typically polyhedra, and sometimes just convex ones.
4. *Simplicity of implementation:* the algorithms are usually complicated to implement due to the necessity of carefully handling special cases.

Space subdivision techniques such as octree[7] and hierarchical bounding box techniques[8] do save time. However, in applications such as computer animation, the space structures must be rebuilt frame by frame whenever the objects move or are deformed, which is generally too expensive. Moreover, the computation cost still remains $O(n^2)$ in the worst case.

Several studies have recently demonstrated that the rasterization scheme can simplify some geometrical problems, such as path planning,[9] ray tracing CSG objects[10] and contour tracing.[11] In this paper, we apply rasterization techniques to interference detection and propose a new type of interference detection algorithm.

The algorithm first rasterizes the projection of objects and calculates their z-values, just like the z-buffer visible surface algorithm. For interference detection, all z-values and pointers to the corresponding faces of objects are saved in a z-list at each pixel. Sorting the z-values in each z-list detects object overlaps in the z-direction at each pixel position, and thus, interfering faces are found by referring to the face pointers in the z-list.

The advantages of the algorithm are as follows:

(a) it is simple and easy to implement
(b) it has linear time complexity proportional to the total number of faces per object
(c) acceleration is possible with standard graphics hardware
(d) it is applicable to any renderable surface.

A drawback of the algorithm is that its resolution is finite. However, there are many applications which do not require exact solutions. For example, visual realism is sufficient in computer animation. In robotics, as well, resolution greater than the accuracy of robot control is sufficient.

When more accuracy is required, the algorithm can be used as an efficient culling algorithm, i.e. if interference is detected, more precise collision/interference detection programs can be called.

ALGORITHM

Consider the example shown in Figure 1, where A, B and C are the objects being examined. The z direction is the projection direction, and x_i is a pixel on the projection plane. If two objects interfere with each other, there exists a pixel where the z-ranges of the objects overlap. In the example, the z-ranges (z_{00},z_{02}) of object A and (z_{01},z_{03}) of object B overlap at the pixel x_0, and thus A and B interfere. On the other hand, objects A and C do not have overlapping z-ranges at any pixel, so they do not interfere at all.

This scheme is realized in the following way. To store the z-ranges of each object, each pixel contains a list of object identifiers (object-ids) and the z-value pair, which is called a z-list. All of the objects are projected and scan-converted onto the pixels in a similar way to the z-buffer algorithm, and their object-ids and z-values are stored in the z-list. For example, the four pixels of the objects in Figure 1 are stored in the z-list as shown in Figure 2(a), where z_{ij} is a z-value and A, B, C are object-ids.

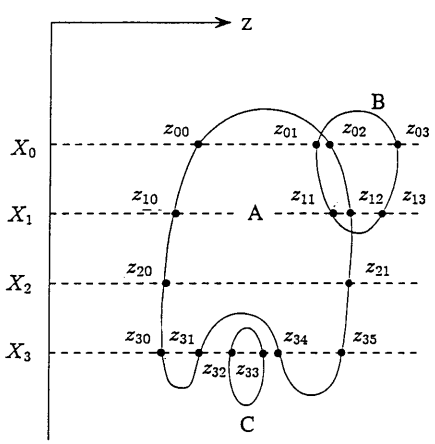


Figure 1. An example of rasterized objects

1049–8907/91/040132–03$05.00

Received February 1991
Revised June 1991

$X_0 \rightarrow (z_{00}, A) \rightarrow (z_{02}, A) \rightarrow (z_{01}, B) \rightarrow (z_{03}, B)$
$X_1 \rightarrow (z_{10}, A) \rightarrow (z_{12}, A) \rightarrow (z_{13}, B) \rightarrow (z_{11}, B)$
$X_2 \rightarrow (z_{21}, A) \rightarrow (z_{20}, A)$
$X_3 \rightarrow (z_{35}, A) \rightarrow (z_{30}, A) \rightarrow (z_{31}, A) \rightarrow (z_{34}, A)$
$\rightarrow (z_{33}, C) \rightarrow (z_{32}, C)$

(a) z-list before sorting

$X_0 \rightarrow [(z_{00}, A) \rightarrow (z_{01}, B)] \rightarrow [(z_{02}, A) \rightarrow (z_{03}, B)]$
$X_1 \rightarrow [(z_{10}, A) \rightarrow (z_{11}, B)] \rightarrow [(z_{12}, A) \rightarrow (z_{13}, B)]$
$X_2 \rightarrow [(z_{20}, A) \rightarrow (z_{21}, A)]$
$X_3 \rightarrow [(z_{30}, A) \rightarrow (z_{31}, A)] \rightarrow [(z_{32}, C) \rightarrow (z_{33}, C)]$
$\rightarrow [(z_{34}, A) \rightarrow (z_{35}, A)]$

(b) z-list after sorting

Figure 2. *Z*-list

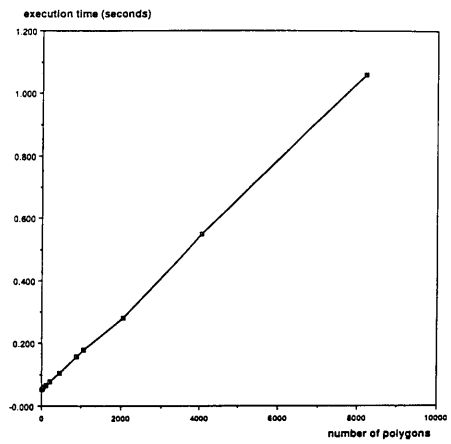


Figure 3. Execution time vs. number of polygons

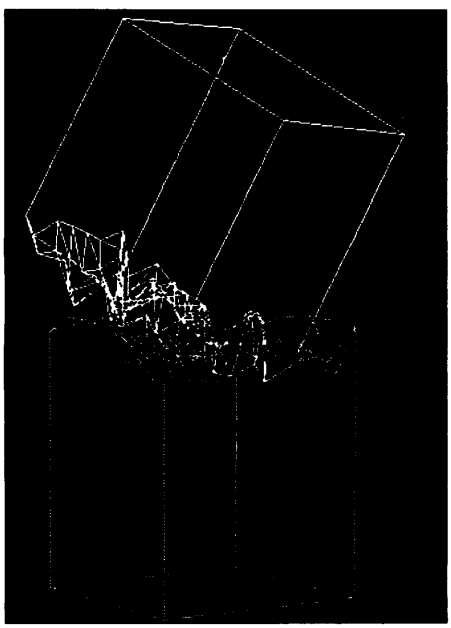


Figure 4. Test objects in the experiment

To detect overlaps in z-ranges, the z-list is sorted against z-values at each pixel. For example, the z-list shown in Figure 2(a) is sorted as shown in Figure 2(b). If a sorted z-list contains pairs of dissimilar object-ids, then interference is detected. In the list in Figure 2(b), pixel x_0 contains the pairs $[(z_{00}, A), (z_{01}, B)]$ and $[(z_{02}, A), (z_{03}, B)]$ (in the Figure, pairs are highlighted by boxes). The two pairs are dissimilar.

In some applications, information on interfering faces is very important, and in this case, the face identifiers (face-ids) can be also saved in the z-list.

One pseudocode of the algorithm is as follows:

```
for all objects {/*make the z-list*/
    draw_id_and_z(object); /*project and
scan-convert*/
    for all pixels drawn {
        append_id_and_z(z_list);
/*append (id,z) pair to the list*/
    }
}
for all pixels drawn {
    sort_z(z_list); /*sort the z-list*/
    if ( overlap(z_list) ) { /*if overlapping*/
        append_id_to_list(); /*make the
list of interfering faces*/
    }
}
```

The complexity of the algorithm is estimated for polygonal objects as follows. Let the area of the projected image be A pixels, n be the total number of polygons, and p be the average number of z-list items per pixel.

The algorithm consists of three main parts; these are object transformation, pixel drawing and z-list sorting. The complexity of each part is detailed as follows:

1. *object transformation:* since all vertices are transformed for the object transformation, the cost is proportional to the total number of vertices, and thus we get $O(n)$, assuming a constant number of vertices per polygon.
2. *pixel drawing:* the total number of drawn pixels is pA, so the cost of pixel drawing is $O(pA)$.
3. *z-list sorting:* sorting a z-list with p items costs $O(p\log p)$. The number of z-lists is A; thus, the total cost is $O(Ap\log p)$.

Finally, when the total number of drawn pixels pA is constant, the complexity of the algorithm is directly proportional to the number of polygons.

For polygonal objects, standard graphics hardware can considerably accelerate the calculation speed. However, since the algorithm uses only sampled z-values, it can deal with any kind of surface for which ray intersection can be calculated (i.e. 'ray-traceable'). In general, ray–surface intersection calculations are much easier than those of surface–surface intersections. Previously untreated surface classes can now be used for interference detection. This is a significant benefit of the algorithm.

IMPLEMENTATION

We implemented the algorithm on a Personal IRIS 4D/25 that contained the GL library. The applicable object shapes are currently either polygons or NURBS surfaces. Only maximum or minimum z-values are stored in the z-buffer. Because of this limitation, the stored z-lists are not assured to be complete in all cases. Fortunately, the z-list is complete for convex objects; thus the program returns correct answers for them.

For general polyhedra, when the program returns no interference the objects do not interfere. When interference is detected, complete z-lists should be built to get the correct answer. This can be performed by, for example, ray tracing all pixels where interference is detected, or by 'software' z-buffering over these pixels.

Implementation was performed in a straightforward way using the GL library. Object-ids and face-ids are stored in the RGB plane and z-values in the z-plane. To obtain maximum and minimum z-values of objects, objects were drawn twice with different logics (less than equal and greater than equal), and stored in the main memory (lrectread).

Figure 3 shows the execution time of the proposed algorithm as a function of the number of polygons n. As expected, the execution time is linear in n. The execution time is almost equal to the display speed, and an interactive speed can be achieved with thousands of polygons (e.g. 0·7 s for 5000 polygons). The pixel resolution used is 128×128. The test object was the randomly displaced mesh shown in Figure 4.

We also implemented a typical interference/collision detection program, based on polygon-edge intersection.[1] The required CPU time was measured under the same condition, and the result is shown in Figure 5. For comparison, we replotted the execution time of the proposed method in this Figure. Note that the complexity of the traditional algoirthm is in $O(n^2)$, and that the algorithm becomes impractical when the number of polygons is greater than one hundred. The difference in the computational complexities of the two methods is clearly demonstrated, and at $n = 1000$, the proposed method is more than 1000 times faster than the conventional one.

Figure 6 shows the execution time of the rasterizing method when pixel resolution changes. As expected, it linearly increases with the number of pixels.

The method was applied to a real-time animation involving dynamic impacts. Figure 7 shows two shots from a sequence of a teapot falling onto a Mandelbrot field. The teapot consists of 552 polygons and the field contains 8192 polygons. The animation was displayed in real time on the IRIS 4D/25,

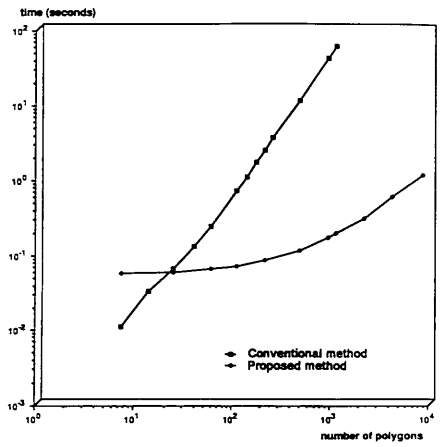


Figure 5. Comparison between conventional and proposed methods

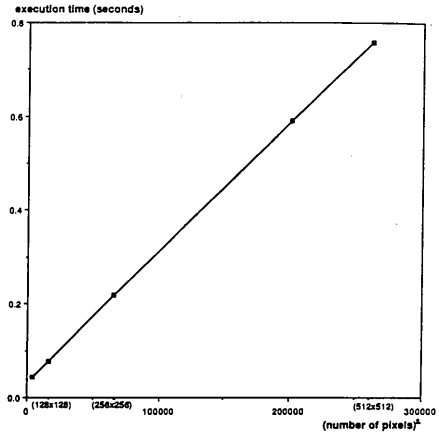


Figure 6. Execution time vs. number of pixels

thus clearly demonstrating the advantage of the proposed method.

CONCLUSION

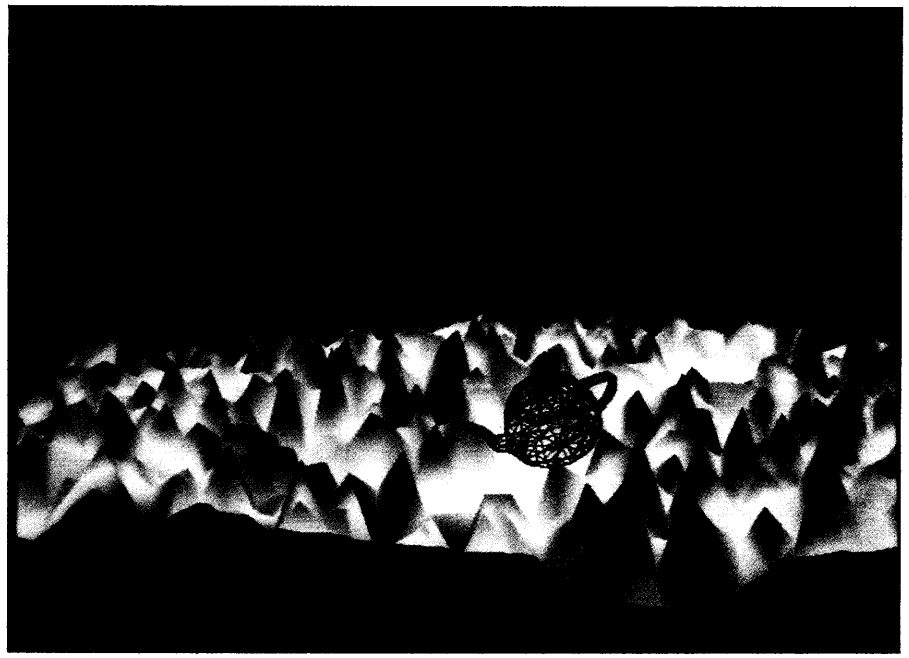
This paper has proposed a new type of interference detection algorithm that uses rasterization techniques.

The advantages of the algorithm are as follows:

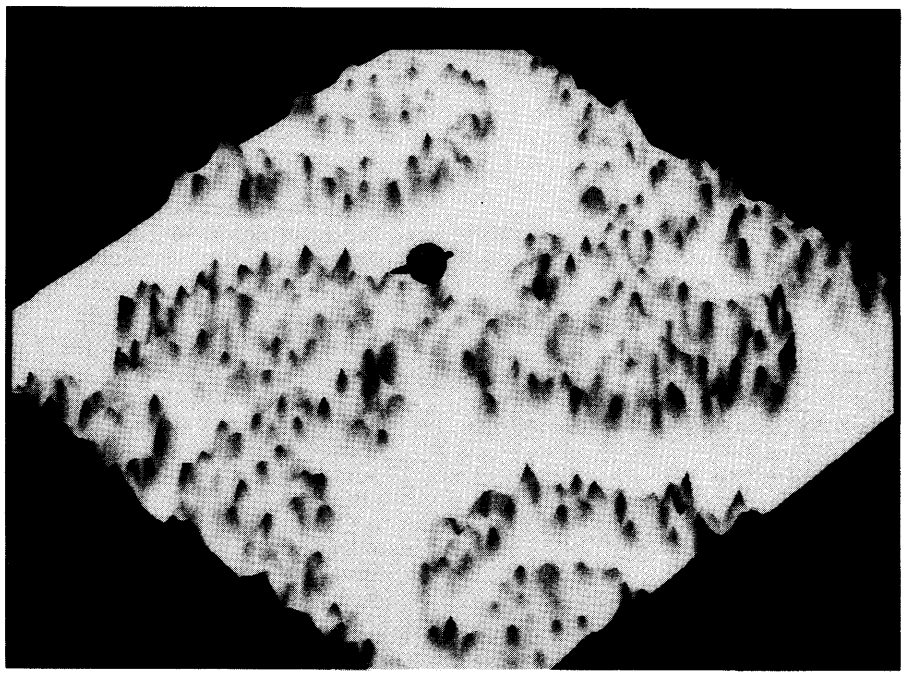
(a) it is simple and easy to implement
(b) it has $O(n)$ complexity for the total number of object faces
(c) acceleration by standard graphics hardware is possible
(d) it is applicable to any renderable surface.

The algorithm has been implemented on a Personal Iris workstation using the GL library. Experiments show that the computation time is comparable to display-calculation time and that, for up to 10^4 polygons, interactive speeds are achieved. The speed is about 10^4 times faster than that of existing algorithms. The present implementation has not been optimized at all, and there is plenty of room for improvement.

The algorithm can be used in two ways: inexpensive interference detection, and an efficient culling method for more precise collision/interference detection algorithms. The algorithm makes real-time interference/collision detection possible.



(a)



(b)

Figure 7. A teapot falling onto a Mandelbrot field

REFERENCES

1. J. W. Boyse, 'Interference detection among solid and surfaces', *Communications of the ACM*, **22**, (1), 3–9 (1979).
2. J. K. Hahn, 'Realistic animation of rigid bodies', *Computer Graphics*, **22**, (4), 299–308 (1988).
3. M. Moore and J. Wilhelms, 'Collision detection and response for computer animation', *Computer Graphics*, **22**, (4), 289–298 (1988).
4. N. M. Aziz, R. Bata and S. Bhat, 'Bezier surface/surface intersection', *IEEE Computer Graphics and Applications*, **10**, 50–58 (1990).
5. E. Gilbert and D. Johnson, 'Distance functions and their application to robot path planning in the presence of obstacles', *IEEE J. Robotics Automation*, **RA-1**, (1), 21–30 (1985).
6. R. O. Buchal, D. B. Cherchas, F. Sassani and J. P. Duncan, 'Simulated off-line programming of welding robots', *The International Journal of Robotics Research*, **8**, (3), 31–43 (1989).
7. H. Samet, 'The quadtree and related hierarchical data structures', *ACM Computing Surveys*, **16**, (2), 187–260 (1984).
8. J. Kajiya, 'Ray tracing complex scenes', *Computer Graphics*, **20**, (4), 269–278 (1986).
9. J. Lengyel, M. Reichert, B. R. Donald and D. P. Greenberg, 'Real-time robot motion planning using rasterizing computer graphics hardware', *Computer Graphics*, **24**, (4), 327–335 (1990).
10. D. Salesin and J. Stolfi, 'Rendering CSG models with a ZZ-buffer', *Computer Graphics*, **24**, (4), 67–76 (1990).
11. T. Saito and T. Takahashi, 'Comprehensible rendering of 3-D shapes', *Computer Graphics*, **24**, (4), 197–206 (1990).