The Visual Computer

Fast collision detection between complex solids using rasterizing graphics hardware

Karol Myszkowski,
Oleg G. Okunev, Tosiyasu L. Kunii

Department of Computer Software, The University of Aizu, Aizu-Wakamatsu, 965-80 Japan

We present an efficient method for detecting collisions between complex solid objects. The method features a stable processing time and low sensitivity to the complexity of contact between objects. The algorithm handles both concave and convex objects; however, the best performance is achieved when at least one object is convex in the proximity of the collision zone (our techniques check the required convexity property as a byproduct of the calculations). The method achieves real-time performance when calculations are supported by the standard functionality of graphics hardware available on high-end workstations.

Key words: Computer animation – Virtual environment – Collision detection

Correspondence to: K. Myszkowski
e-mail: k-myszk, o-okunev, kunii@u-aizu.ac.jp

1 Introduction

Detecting collisions is an important component of computer systems involved in realistic animation, robot motion planning and simulation, and virtual reality. Detecting collisions between moving objects is a computationally intensive task dominated by geometric calculations. In practical applications, the object models are composed of multiple elements such as polygonal faces or patches defined by parametric or implicit functions. The naive collision detection algorithm has a complexity $O(n^2)$ for n elements. When the contact between the colliding objects is simple (e.g., an isolated point), the majority of elements can be discarded with inexpensive tests like one for the interference of bounding volumes. In such a case, only the elements located in the proximity of the collision zone require more elaborate processing. However, in many applications, complex contacts involving multiple faces or patches are possible, and simple discarding elements does not solve the problem.

The motivation for this work is detection of collisions in a computer simulation of human jaw articulation. The model of the jaws derived from scanning the teeth is complex (over 100 000 polygons). The dentist interactively manipulates the mandibular position, inspecting occlusion of the jaws and determining the regions of contact between the teeth.

The goal of this work is to develop a general purpose collision detection algorithm that has:

- The ability to handle complex colliding objects. The algorithm is expected to perform best for a few extremely complex objects rather than for many simple objects.
- A negligible penalty for the complexity of the contacts.
- Interactive time rates for the interference checking.
- Similar performance for single interference tests and equivalent tests done in a sequence of tests. This means in particular that the strength of the algorithm cannot rely on any form of space-time coherence with the previous interference tests.
- Insensitivity of performance to interactive changes of positions of the object. The algorithm does not exploit a priori given paths of motion of the objects.

The Visual Computer (1995) 11:497–511

- The ability to run on standard workstations equipped with graphics hardware.

The requirements imposed on the complexity of objects and latency of response preclude the use of general purpose computers. A natural choice is graphics workstations, granting high polygon throughput of hardware-assisted rendering. A reformulation of the collision detection algorithm is needed for efficient use of the computational power of the graphics hardware. The distance or intersection checking between objects used by traditional algorithms should be replaced by depth calculations, which are directly supported by graphics accelerators. This strategy seems to be justified by the rapid progress and development of rendering techniques implemented by means of specialized hardware. At the same time, it is difficult to expect hardware support for the aforementioned traditional approaches. The algorithm presented here refers to the technologies becoming an industrial standard, such as Z-buffer, color alpha blending, pixel masks (also called *stencil buffer*; Neider et al. 1993), which are available on high-end graphics workstations. We also postulate some simple extensions of the graphics libraries (granting access to the hardware resources), which could even further improve the efficiency of the collision detection.

The next section discusses the existing methods of collision detection. Sections 3–7 describe our hardware-assisted techniques for detecting collisions which handle both convex and concave objects. Experimental results showing the performance and accuracy of the algorithm are presented in Sect. 8. Finally, Sect. 9 contains our conclusions and suggests some areas for future research.

2 Previous work

Conservative collision determination requires an interference test between the first object and the volume swept by the second object moving relatively to the first object within time δt. However, this approach is computationally expensive, and the general solution for complex objects and motion paths is difficult to find (Menon et al. 1994). To avoid these problems, the object motion is usually discretely sampled as a function of time, and collisions are tested for frozen positions of the objects only. This is also the case when discrete sets of positions are experimentally measured in a real environment or are registered for objects interactively (Kunii et al. 1994). The discrete approach produces acceptable results for many practical applications. However, some intersections can be missed, especially for the objects exhibiting spikelike surfaces (von Herzen et al. 1990). One can adaptively adjust the sampling density of the object position, taking into account the shape of the object, its velocity, curvature of the motion path, and position with respect to other objects.

The intersection test for nonpolygonal surfaces (Hanna et al. 1983; Phillips and Odell 1984; Aziz et al. 1990; von Herzen et al. 1990) requires a substantial computational effort, which makes this approach impractical for real-time applications. The majority of the existing collision-detecting algorithms assumes a polygonal representation of the surfaces (Hahn 1988; Moore and Wilhelms 1988; Yang and Thalmann 1993; Garcia-Alonso et al. 1994). The tessellation of the surface into polygons can be a source of collision-determination errors traded off with a larger number of polygons and longer processing times (Boyse 1979). The main strategy to speed up the interference test is to use inexpensive tests to exclude as many noncolliding polygons as possible. Usually a hierarchical structure is imposed on the objects and corresponding polygons. The collision interest matrix (Garcia-Alonso et al. 1994) can be built on the top level, which determines whether the interference test is necessary for each pair of objects. If a pair of objects can collide, the bounding volumes embedded in hierarchical data structures, such as the octree (Moore and Wilhelms 1988; Yang and Thalmann 1993), voxel grid (Garcia-Alonso et al. 1994), and bounding volumes hierarchy (Hahn 1988), are used to exclude the polygons located in the space regions occupied by one object only. The response time for collision-detecting algorithms based on bounding volumes is very unstable. When complex contacts between objects are possible, trivial polygon rejection fails completely, and expensive intersection tests on polygons must be used.

In robotics applications, the distances between the nearest points (edges, faces) in two convex polyhedra are often alternately examined

(Cameron and Culley 1986; Culley and Kempf 1986). However, when the objects move, the nearest points should be updated. For complex objects, this is a time-consuming task, and concave objects must be decomposed into convex pieces. The best performance of this algorithm seems to be achieved when a collision of many very simple objects is to be checked. The algorithm also takes advantage of the collision paths known in advance to decrease the frequency of the interference tests. As we have stated in the introductory section, the objectives of our algorithm are different.

An alternative approach to collision detection is based on the elimination of analytic intersection or distance checking by inexpensive, discrete tests at selected sample points. Uchiki et al. (1983) represent objects as a grid of voxels and detect the voxels assigned to more than one object. The accuracy of such an interference test is rather poor, and memory requirements are very high. Menon et al. (1994) proposed the ray representation of solids computed by "clipping" the grid of regularly spaced parallel lines against the solid. The method was developed for efficient handling of boolean operations for constructive solid geometry (CSG), including intersection operation, which can be used for detecting collisions. However, a specialized hardware architecture called a RayCasting Engine is required to classify grids of parallel lines in interactive rates.

Detecting interference through rasterization with a Z-buffer or ray tracing was proposed by Shinya and Forgue (1991). For every pixel, the list of overlapping objects and corresponding depth values is calculated. The lists are sorted by z values, and the object identifiers are grouped into pairs. The existence of a pair formed by two objects is equivalent to a collision. Shinya and Forgue show that the complexity of the algorithm is directly proportional to the number of polygons and the number of pixels. The algorithm performs well when the hardware accelerator can be used, but otherwise it becomes too slow to be practical in real-time applications. Shinya and Forgue manage to use a hardware-supported Z-buffer when all objects are known in advance to be convex. In this case, two depth maps (Z_{min} and Z_{max} values) must be stored for every object. A huge stream of bytes is copied from the depth buffer into the main memory. The application software sorts depth maps and groups object identifiers in pairs, which is slower than the hardware-supported process. Also, maintenance of the lists is memory consuming. When concave objects are considered, the graphics hardware only tests whether Z_{max} of the first object is smaller than Z_{min} of the second. When this test fails, further processing by ray tracing or software Z-buffer is required.

Rossignac et al. (1992) propose a method for checking the interference between slices of solid objects that are cut by two parallel cross-section planes. The hardware-supported rendering uses pixel masks and clipping planes to determine whether the slices intersect, which is equivalent to collision. We test for collision on the front and back cross-section planes, and this information is extrapolated to the interior of the slice. This approach is robust when the intersection between slices of solids is detected. Otherwise, collision cannot be excluded, and thinner slices should be considered. The slices are recursively subdivided until a collision is detected or the maximum subdivision limit is reached. The algorithm usually requires many scan conversions of the solids and does not achieve real-time performance. The processing time is sensitive to the initial guess of the collision zone. The calculation cost grows rapidly when many collision-free slices are recursively subdivided.

The techniques for detecting collisions with rasterizing graphics hardware rely on the assumption that every straight line crosses the boundaries of the objects at an even number of points (Rossignac et al. 1992). This condition holds in particular for solid objects. The proper treatment of tangent rays by scan-conversion hardware is discussed in Rossignac (1992).

3 Overview of our approach

The algorithms proposed in this paper also use rasterizing graphics hardware and are applicable to solid objects (i.e., bounded, closed, regular point sets, with polyhedral surfaces, or other types of surfaces supported by the functionality of graphics hardware, e.g., nonuniform rational B-splines [NURBS]). However, a smaller number of rendering passes than used by the algorithm

proposed by Rossignac et al. is needed to reconstruct the collision areas. Also, more general classes of objects can be completely processed with hardware assistance than with the Shinya and Forgue algorithm. The proposed techniques exhibit the following features improving performance and extending functionality of the algorithms discussed:

- The Shinya and Forgue approach assumes that the objects are convex for effective use of the hardware Z-buffer. If this assumption holds, our basic algorithm requires much less depth map processing by the application software than the Shinya and Forgue techniques. We use the functionality of graphics hardware for depth sorting and detecting the interference regions. In many practical cases, only one rendering pass for each object is needed to detect some collision and noncollision positions of the objects. (The Shinya and Forgue algorithm can detect only some noncollision positions of the objects after a single rendering pass.) Two rendering passes are needed in more complex cases only.
- No *a priori* assumption about the convexity of the objects is required by our extended algorithm. For every pixel and for the particular viewing projection used by the scan-conversion algorithm, our algorithm checks whether the object should be treated as convex or concave. This check is obtained as a byproduct of depth calculations that use the standard capabilities of the graphics hardware. The appropriate algorithm for handling convex or concave objects is chosen for the groups of pixels that are overlapped by the colliding objects in the same way. We show that, in many cases, concave objects can be processed by the cheaper algorithm for convex objects.
- We propose hardware-assisted techniques for concave object processing that can handle many layers of overlapping polygons per pixel. The maximum number of layers is equal to the number of the bitplanes in the frame buffer.
- The accuracy of the interference test is improved.

Each of these items is described in more detail later. Our presentation develops in a step-by-step manner, starting from the simplest case of collision detection with the convexity of the objects known in advance. The convexity assumption is not needed for an extended version of the basic algorithm, which checks the required convexity property. Finally, a general algorithm that handles concave objects is proposed.

4 Convex objects

In this section we present the hardware-assisted graphics algorithm for detecting a collision of a pair of convex objects A and B. The algorithm can easily be implemented on commercially available high-end graphics workstations. We adjust particular operations of the algorithm to the functions supported by graphics libraries. At the end of the section, we postulate some extensions for these functions, which would improve the efficiency of our techniques.

When graphics techniques such as ray casting or depth buffer are applied to collision detection, the problem is reduced to one dimension, and for two convex objects six trivial cases are possible (Fig. 1). Usually it is not important which particular case occurs, and it is enough to detect any kind of collision. The criterion for interference may be stated in the following way: *There is no collision between the objects A and B if for every pixel overlapped by these objects, only the cases a or f occur.* (Of course, this statement is correct within the accuracy of the rasterization techniques, which, for the particular viewing projection, is determined by the frame and depth buffers resolutions.) This means that the exact sorting of intersection points is not necessary. Case a is equivalent to a successful test

$$Z^A_{max} < Z^B_{min} \tag{1}$$

where Z^A_{max} and Z^B_{min} stand for the maximum and minimum values of depth for the objects A and B, respectively. Test (1) can be performed during a single rendering of A and B with the depth comparison enabled. However, the second rendering pass is needed when the test fails for some pixels, because the cases f (no collision) and b–e (Fig. 1) cannot be distinguished at the first pass. We propose an algorithm that detects conservatively, not only the case a, but also the cases b and c in the first rendering pass, thus avoiding the

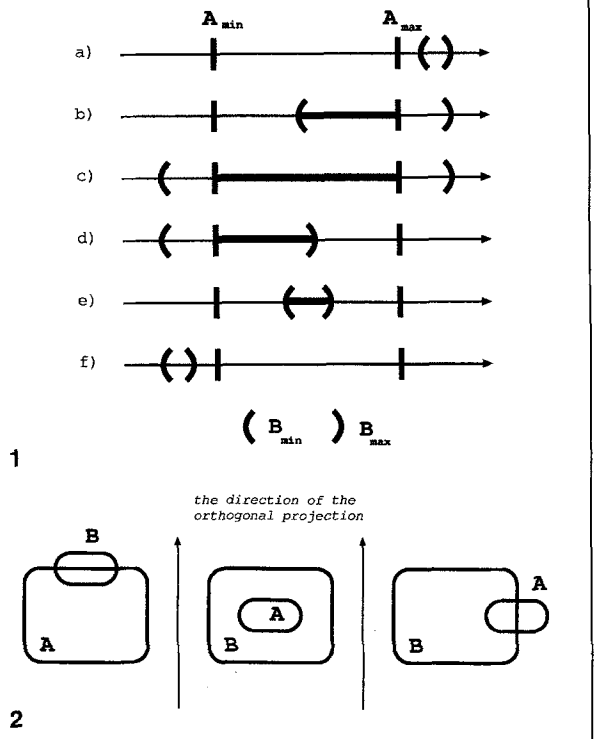


Fig. 1. Interference of two convex objects in 1D space

Fig. 2. Locations of the objects A and B for which the second rendering pass is not needed

second rendering for some locations of A and B in space (Fig. 2).

In our formulation of the depth test we count the number of successful passes for the test

$$Z^A_{max} \geq Z^B_{arbitrary} \tag{2}$$

where $Z^B_{arbitrary}$ is the unsorted depth value for B. Only the pixels overlapped by A are tested.

The interpretation of the test results is straightforward. When failures only are scored, no collision is possible, because there is a clearance between A and B, or B does not overlap this pixel. The same result for all pixels means no collision, and the procedure is completed.

A collision is detected when one failure and one pass are scored (case b or c in Fig. 1, see also Fig. 2). If the only goal of the interference test is a binary decision, *collision* or *no collision*, this result completes the procedure. However, if the exact reconstruction of the whole collision area is required, then we render a second time for the pixels with two recorded passes.

After two passes of test (2), collision cannot be excluded; the cases d, e, or f are still possible (Fig. 1). A second rendering is needed to classify these cases. The depth test during the second rendering is

$$Z^A_{min} > Z^B_{arbitrary} \tag{3}$$

Taking into account the results of the first rendering, two passes mean no collision (case f in Fig. 1); otherwise we have a collision (case d or e in Fig. 1). It should be noted that test (3) is equivalent to $Z^A_{min} \leq Z^B_{arbitrary}$ where failures and passes are interpreted in the opposite way. Test (3) was chosen for consistency of interpretation of failures and passes in the following description of algorithms for concave and convex objects.

The outlined algorithm can easily be implemented with Z-buffer rendering and pixel masks. Two operations are performed on the pixel masks: (1) setting the pixel mask SA when the object A overlaps the pixel, and (2) counting the number of polygons overlapping the pixel. The counter SC (implemented as two pixel masks) is incremented when the depth test is successfully passed for scan-converted B.

The basic algorithm for the collision detection between convex objects A and B

```
01 For all pixels set Z=0 and SA=0
02 Disable writing into the frame buffer
03 Set the Z comparison function for finding Z_max
04 Render all faces of A setting SA (search for
   Z_max)
05 Change the Z comparison for finding Z_min (test
   (2))
06 RenderObjectB( )
07 second_rendering=FALSE
08 For all pixels where SA=1
09   If (SC==1) return COLLISION
10   If (SC==2) second_rendering=TRUE
11 If (second_rendering==FALSE)
     return NO_COLLISION
12 For all pixels
     If (SC==2) SA=1 Else SA=0
13 Enable writing into the Z-buffer
14 For all pixels set Z=INFINITY
```

```
15 Render all faces of A for pixels where SA = 1
   (search for Z_min)
16 Change the Z comparison for finding Z_min (test
   (3))
17 RenderObjectB ( )
18 For all pixels where SA=1
19   If (SC==0 or SC==1)
        return COLLISION
20 return NO_COLLISION

RenderObjectB( )
01 Disable writing into the Z-buffer
02 For all pixels set SC=0
03 Render all faces of B for pixels where SA=1 and
   increment SC when the depth test passes
```

When object A is rendered, the pixel mask SA is set for every pixel overlapped by A (line 04). Object B is only rendered for pixels where SA is set (line 03 in the procedure RenderObjectB). Pixels overlapped by A and not overlapped by B have counter SC equal to zero when the first rendering pass is completed (the same value corresponds to case a in Fig. 1). These pixels are ignored in the second rendering pass (line 12).

The whole algorithm except pixel masks examination (lines 09, 10, and 19) was implemented with standard functions provided by the graphics libraries. This means that, on high-end workstations, most of these calculations are hardware supported. Unfortunately, we were forced to examine final values of the counter SC for every pixel marked by SA with software. We hope that, in future releases of graphics libraries, services such as checking the settings of the chosen pixel masks for the whole frame will also be hardware assisted; similar extensions of functionality have also been postulated by Rossignac et al. (1992).

In the pseudocode, the frame buffer is not used explicitly. However, in practical implementation, access to pixel masks is granted by mapping their status into the frame buffer. As the depth of the frame buffer on a high-end workstation is at least 24 bits, the status of the pixel masks acquired during two rendering passes can easily be stored there.

In the pseudocode, we assume that the goal of the algorithm is to detect any collision between objects. In fact, the algorithm calculates the complete map of collision areas, which are directly available and processed by the application software (lines 09, 10, and 19). The processing can easily be adjusted to the application needs. It is worth noting that the algorithm distinguishes all possible collision cases (Fig. 1) except cases b and c, which are encoded in the same way in the collision map (line 09).

To reduce the number of rendering passes and computation done by the application software, the following measures can be taken:

- As the initial guess, consider the object located farther along the direction of the orthogonal projection as B.
- When two rendering passes are required for the particular locations of objects and two successful passes of test (2) are recorded for many pixels, the order of the objects considered can be reversed at the next animation step (the current A becomes B and vice versa).
- When the objects are simple (can be rendered very fast), and high-resolution pixel masks are used (because of the required accuracy of the collision detection), the examination of SC by the application software becomes the bottleneck of the calculations. In such a case, a speed-up can be achieved by examining the SCs only once after completion of two rendering passes. As has already been mentioned, the frame buffer can be used to store the SC count from the first rendering pass.

It should be noted that when the collision detection is completed after the first rendering pass, and the orthogonal projection parameters are not changed for the next animation step, then the rendering of A should not be repeated (lines 01–05 can be skipped). (When A is mobile, the depth maps can be re-used if the relative motion of B with respect to A is considered.) The rendering of B does not overwrite the depth values calculated for A (line 01 of the procedure RenderObjectB). Of course, the second rendering destroys the depth maps calculated during the first rendering. If the second rendering is required often, and A is complex, it may be worthwhile to store the depth maps of A in the main memory and copy them into the depth buffer instead of repeating the rendering. Direct writing into the depth buffer is supported by graphics libraries

and can be faster than rendering the complex object.

5 Convexity test

The algorithm proposed in the previous section works efficiently for convex objects; however, in practical applications we are very often dealing with nonconvex objects. Usually it is not known in advance whether the object is convex; for example, when the description of the shape is derived experimentally through 3D scanning.

We may notice that the presented algorithm does not require *global convexity* of the colliding objects. Let P be a point on the surface of an object. We say the object is *convex at P in a direction π* if the intersection of the line through P in the direction π with the object is connected (that is, if it is a single segment). From the point of view of the representation of the object, convexity in a direction π at a pixel p means that there are exactly two polygons in the polygonal representation of the boundary of the object whose projections in the direction π contain p (for brevity, we say that the object is *π-convex* at the pixel p). We assume here that pixels in the projections of edges are treated as belonging to exactly one of neighboring polygons; this assumption is commonly implemented in graphics hardware (Graphics Library Programming Guide 1992; Neider et al. 1993). If A and B are π-convex in a given projection direction for all pixels overlapped by A and B, then the algorithm presented in the previous section works properly, and we need not worry about global convexity.

In this section, the basic collision-detecting algorithm for convex objects (Sect. 4) is extended to determine the π-convexity of A and B in a given projection direction and to handle the simple cases of interference between concave objects. The π-convexity is checked for every pixel by counting the number of the overlapping polygons. When the π-convexity holds, the old approach can be used. Otherwise, our strategy is to solve the collision problem for as many pixels as possible, minimizing the number of pixels for which a more expensive algorithm for concave objects must be used. The extended algorithm completely processes a collision between two objects, when one of them is π-convex. The proposed extension of functionality requires only minor changes in the basic algorithm, which can be handled by standard graphics libraries, except inspecting the pixel masks settings.

The first rendering pass for objects A and B is the same as in the basic algorithm. The counter SC is also incremented when a polygon of B passes depth test (2). However, more pixel masks are needed to count all overlapping polygons and avoid counter overflow. The number of pixel masks for the SC depends on the expected complexity of B. For example, three pixel masks can safely handle six layers of polygons overlapping a pixel, but for eight layers, the counter clamping to its maximal value is guaranteed (Neider et al. 1993). In such a case the counter overflow is easy to detect, because the value of SC is seven, which is not allowed for solid objects. The interpretation of SC (lines 08–12 and 15 in the basic algorithm for convex objects) is extended in the following way:

```
Extension #1 of the basic algorithm for concave objects
080 For all pixels where SA = 1
090    If (SC odd) return COLLISION
100    Else If (SC > 0)
          second_rendering = TRUE
110 If (second_rendering == FALSE)
       return NO_COLLISION
120 For all pixels
       If (SC > 0) SA = 1 Else SA = 0
121 For all pixels set SC = 0 (used for the convexity
    test of A)
...
150 Render all faces of A for pixels where SA = 1
    and always increment SC (ignore results of
    Z comparison)
151 For all pixels If (SA == 1 and SC == 2) SCA = 1
    Else SCA = 0
```

When SC = 0 for all pixels, collision is impossible. A second rendering pass is required for pixels for which collision was not detected (line 090) and the number of the overlapping polygons is even and nonzero (line 100). The second rendering pass is performed only for the pixels for which SA = 1 (line 120).

When A is rendered the second time, the SC counts the number of polygons overlapping the pixels where SA = 1 (line 150). After completion

of the rendering, one pixel mask SCA is allocated and set to 1 for pixels overlapped by two polygons (SC = 2), i.e., exhibiting the property of π-convexity for the object A (line 151).
The second rendering of the object B and the use of the SC are the same as in the first pass. The interpretation of the SC is more complex than in the basic algorithm (lines 18–20, Sect. 4) and requires the comparison of the SC values calculated for the first and second rendering passes (hereafter referred to as SC1 and SC2, respectively).

Extension #2 of the basic algorithm for concave objects

```
180 For all pixels where SA=1
190   If (SC2 odd) return COLLISION
191   Else If (SC1==SC2) continue
192   Else If (SCA==1) return COLLISION
193   Else mark this pixel to be processed by the
      algorithm for concave objects
200 If (Line 193 is never reached)
201   return NO_COLLISION
202 Else
203   Process pixels marked at line 193
```

Fast operations on stencil planes uniquely mark the pixels in the frame buffer that pass the tests in lines 190–193. However, the final interpretation of the frame buffer is done by the application software. Line 191 requires the copies SC1 and SC2 of the SC obtained at the first and second rendering passes. If the same even number of polygons overlapping the pixel passed the depth test in the first and second rendering of B, then collision is not possible. Line 192 examines whether the object A is π-convex at the current pixel. If this is the case, then for an even value of SC2, and an SC2 different from SC1, the collision is detected, completing the procedure for the considered pixel. However, when A is not π-convex, further processing, described in the next section, is needed.
If two objects are concave, then the more complex object should be chosen as A to avoid SC overflow. If one object is convex, then it should be rendered as A, and the complex algorithm for concave objects is not used (line 193 is never reached).
The critical issue is the number of pixel masks supported by the graphics workstation (e.g., up to eight stencil planes on SGI's RealityEngine2). When the second rendering pass is required, the copies of SC (SC1 and SC2) for two rendering passes are needed, plus two single pixel masks SA and SCA. If B is not complex (up to six polygon layers per pixel), then eight pixel masks are sufficient to store all informations. Otherwise, SC1 should be stored in the frame buffer before the second rendering pass is started. This is also the case when only four pixel masks are available, as for example, in SGI's Indigo2. In such a case, operations on the pixel masks that could be supported by graphics hardware are performed by the application software.
The extended algorithm minimizes the number of pixels that must be handled by the time-consuming collision detection for concave objects. However, this can require many pixel masks to process very complex objects. The simpler version of this algorithm can be implemented with only three pixel mask planes (one for SA and two for SO and SP, which replace SC). In the first rendering pass, B should be scan-converted twice: (1) for the SO setting, for pixels overlapped by B, and (2) for SP toggling, when the depth test is passed successfully. The interpretation of SO and SP is the following:

Extension #1′ of the basic algorithm for concave objects (simplified version)

```
080 For all pixels where SA=1
090   If (SO==1 and SP==1)
        return COLLISION
100   If (SO==1 and SP==0)
        second_rendering=TRUE
110 If (second_rendering==FALSE)
      return NO_COLLISION
120 For all pixels If (SO==1 and SP==0) SA=1
    Else SA=0
```

In the second rendering pass, B should be scan-converted only once, and SP is toggled for every successful pass of the depth test. The final evaluation of SP is:

Extension #2′ of the basic algorithm for concave objects (simplified version)

```
180 For all pixels where SA=1
190   If (SP==1) return COLLISION
```

```
191  Else mark this pixel to be processed by the
     algorithm for concave objects
200 Process pixels marked at line 191
```

Because the π-convexity is not checked, the simplified algorithm requires further processing for some pixels that could be fully processed by the extended version of the collision-detecting algorithm. One extra rendering of B is also required. When the objects are concave, Shinya and Forgue only test whether $Z^A_{max} < Z^B_{min}$, and for all remaining pixels apply ray tracing or software Z-buffer. This means that the software-implemented techniques must process many pixels that can be completely processed by graphics hardware when our algorithm is used.

6 Concave objects

The unsolved cases for a pair of concave objects that cannot be handled by the algorithm presented in the previous section require polygon sorting along the direction of the orthogonal projection. A similar requirement is imposed by rendering CSG or transparent objects. The traditional methods used to solve these problems, like ray casting (Roth 1982), A-buffer (Carpenter 1984) or ZZ-buffer (Salesin and Stolfi 1990) are too slow, and a significant speed-up can only be expected when hardware-supported sorting of polygons is provided. The most popular approach is polygon sorting during the multi-pass rendering. Mammen (1989) solves the problem of transparency with duplicate depth and frame buffers. The first buffer stores the reference depth, and the other one is used to find the nearest object located in front of this reference. The object just found becomes the reference in the next rendering pass when sorting is continued for the next layer of polygons. The implementation of this algorithm requires flexible configuration of the frame and depth buffers offered by the Stellar graphics accelerator. Recently, a low cost, single-ASIC (Application-Specific Integrated Circuits) graphics accelerator was produced for Power Macintosh (Kelley et al. 1994), which stores and sorts up to four depth values for a 16-pixel-long segment of the scan line. Unfortunately, similar capabilities are not yet very common on commercially available workstations. However, the number of bits per pixel is growing for the subsequent generations of graphics hardware. Also, graphics libraries offer more and more freedom in tailoring these resources to the requirements imposed by applications. Observing these trends, we believe that hardware assisted polygon sorting will be easy to implement on many hardware platforms soon. As was shown, some platforms offer such functionality even now.

In this section we propose an algorithm that detects collisions of concave objects. It is assumed that all polygons are sorted along the direction of the orthogonal projection. Real-time performance can only be expected from this algorithm when hardware-supported object sorting is available. (In our current implementation the application software does the sorting.) The layers of polygons can be rendered in front-to-back or back-to-front order. Alpha blending is used to accumulate information about the order of the polygon layers belonging to A and B. The value $\alpha = 0.5$ is used, and the blending model is described by the formula

$$C_{acc} = C_{object} + \alpha C_{acc} \tag{4}$$

where C_{acc} stands for the accumulated value of the red, green, blue (RGB) color components stored in the frame buffer, and C_{object} is the corresponding component of the object color. The color of the first object is set to zero for all components. The color of the second object is set to 2^{k-1}, where k is the number of bitplanes for a single color component. The proposed blending function shifts the contents of C_{acc} right and sets the most significant bit to one or zero, depending on whether A or B is rendered. As a result, zeros and ones in C_{acc} correspond to the objects A and B respectively. The collision can be detected when an odd number of the same object identifiers is found in the sequence in C_{acc}, e.g., *AABBABBA* (00110110), *BBBAABBB* (11100111).

One may notice that an 8-bit C_{acc} encodes up to 8 layers of the overlapping polygons. However, C_{acc} consists of three RGB buffers, which can be used to encode up to 24 layers (or 32 layers when alpha bitplanes are available, e.g., RealityEngine2). The technical problem to be solved is to prevent simultaneous blending for all RGB buffers. This can easily be done by masking two components and allowing blending for the third. Every 8

rendering passes (corresponding to 8 layers processed), writing to another color component is enabled while the component already processed is masked. Another technical problem is related to the numerical accuracy of alpha blending. We did not encounter such problems on SGI's Extreme graphics hardware when $\alpha = 128/255$ was assigned. On SGI's RealityEngine2, floating point α is supported, and our layer encoding also works well. However, when numerical problems are expected, the least significant bit should not be used, and the next color component should be switched every seven rendering passes.

Multipass rendering for complex multilayered objects is time consuming. However, the calculations are only required for pixels that cannot be processed by simple depth comparison and counting of parity of layers proposed in the previous section. If these pixels are concentrated into small and compact image areas, then viewport parameters can be changed by focusing multipass rendering on such areas. Many polygons outside the viewport will be discarded immediately at the clipping stage, avoiding scan conversion. However, when only a small number of dispersed pixels requires extended multipass rendering, then ray tracing is the best choice. The adaptive density of traced rays can be applied to surfaces exhibiting high variation of depth for neighboring samples.

7 Accuracy issues

An important problem is the choice of the orthogonal projection that provides a high accuracy of collision detection. The projection should maximize the number of pixels falling into the collision zone. Pixels that do not overlap objects A and B do not provide any useful information. The best depth-measurement accuracy in the proximity of the collision can be expected when the projection plane is tangent to the surfaces of A and B at the point of collision. When the angle between the projection plane and polygon increases, the accuracy of the depth calculations drops abruptly. When the contact between colliding surfaces is complex, these difficulties cannot be avoided if only one projection is considered. We propose an inexpensive algorithm for the choice of the initial guess of the projection plane that takes into account the approximate collision zone calculated as the intersection of bounding volumes. Then we propose a more exact algorithm that uses the information acquired in the previous step of the collision detection.

The bounding volume intersection test is done to eliminate quickly the cases for which collision is impossible. First, the bounding volume of the object A transformed to the local coordinates of B is tested against the bounding volume of B. If an intersection is detected, a similar test is performed in the local coordinates of A. As a result, two perpendicular parallelepiped intersection volumes are obtained. For each intersection volume, the largest wall is found. The average of the normal vectors to these two walls is the initial guess of the orientation of the projection plane used by our collision-detecting techniques. The orthogonal projection of the intersection volumes on this plane establishes the viewport parameters used during the scan conversion.

The accuracy of the collision detection may be improved if the projection direction is calculated as the average of the normal vectors of all polygons in the collision zone. The surface areas of the polygons can be used as weights for such averaging. The polygons in the collision zone are identified with the *item buffer* (Weghorst et al. 1984), where each polygon is drawn to the frame buffer with a unique color. In such a case, drawing the polygons into the chosen bitplanes of the frame buffer should be enabled (line 02 of the basic collision-detecting algorithm). Other bitplanes remain reserved for storing copies of the stencil planes. The application of the item buffer is just a special way of rendering and does not involve any additional cost. Of course, processing of the item buffer and calculating the average normal vectors requires some extra time and should not be repeated for every step of the animation of the colliding objects. When changes of the object position are small, the same projection plane can be used for subsequent collision tests.

In many practical applications, the expected collision zone can be guessed in advance, and then the best orthogonal projection plane can be found once and for all at the preprocessing stage.

When high accuracy of collision detection is critical, and the variation of the normal vectors of polygons in the collision zone is high, then multiple projection planes can be defined. Of course,

this requires multiple rendering passes. Another solution, originally proposed by Shinya and Forgue (1991) is to have the application software calculate exactly the collision for the polygons identified with hardware assistance. This solution can be extended to process spike-shaped surfaces, which can be missed easily by rasterizing graphics hardware. During scan conversion, special bounding volumes can be rendered instead of original spike-like surfaces. When the item buffer contains the identifier of such a bounding volume, then the original surface can be processed by the application software.

8 Experimental results

The algorithms presented in this paper were implemented on Onyx and Indigo Silicon Graphics workstations. All timings given in this section were measured for the RealityEngine2 and Extreme graphics hardware (the results for Extreme are in brackets).

To get easy-to-validate results, we evaluated the efficiency of our collision-detecting algorithm for multiple simple polyhedra. We simulated variable complexities of the objects by tessellating their faces into multiple triangles. Table 1 presents timings of collision detection for scenes in Fig. 3, assuming that each scene is built of 160 000 3D polygons and the resolution of the stencil and frame buffers is 512×512 pixels. The scene in Fig. 3a, which is composed of one π-convex and one concave object (up to six layers of polygons per pixel), was processed by the algorithm described in Sect. 5. The basic algorithm presented in Sect. 4 was tested with the scene in Fig. 3b composed of two π-convex objects. A general algorithm for concave objects (Sect. 6) was used to check the collision between the objects in Fig. 3c. The total calculation time (Table 1) was measured for the worst case scenario, i.e., when two or three (for two concave objects in Fig. 3c) rendering passes were performed. The column *Graphics hardware* presents timings of the single rendering of the objects A and B and the corresponding pixel masks processing (in practice, no difference between the first and second rendering passes was registered). The cost of examining the pixel masks by application software, including the image transfer from the frame buffer to the main memory, is given separately for every rendering phase in the column *Application software*. The subcolumn "*phase* 3" shows the costs of the analysis of the order of polygons encoded by colors in the frame buffer. These calculations are relevant only for the algorithm processing two concave objects, which requires one extra rendering pass and encoding the order of polygons by alpha blending (Sect. 6). We assumed that the polygons are depth sorted, and the cost of sorting (at present done by the application software) was not included in Table 1. The algorithm for concave objects exhibits reasonable performance under the condition that depth-sorted polygons are available, i.e., hardware support for the sorting is provided. Also, the comparison of the algorithm for convex objects (Sect. 4) and the more general algorithm for one π-convex object and an arbitrary object (Sect. 5) shows that their performance is similar; however, the latter algorithm requires more pixel masks.

We noticed a linear growth of the calculation costs with respect to the number of polygons for all presented algorithms. This is not a surprise, because Shinya and Forgue (1991) also reported

Table 1. Timings of collision detection for scenes in Fig. 3 measured for SGI's RealityEngine2 and Extreme graphics hardware.

	Graphics hardware (s)	Application software			
		Phase 1 (s)	Phase 2 (s)	Phase 3 (s)	Total (s)
Fig. 3a/RE2	0.27	0.13	0.17	–	0.84
Fig. 3b/RE2	0.27	0.13	0.10	–	0.77
Fig. 3c/RE2	0.27	0.13	0.17	0.28	1.39
Fig. 3a/Extreme	0.52	0.18	0.25	–	1.47
Fig. 3b/Extreme	0.52	0.18	0.14	–	1.36
Fig. 3c/Extreme	0.52	0.18	0.25	0.35	2.34

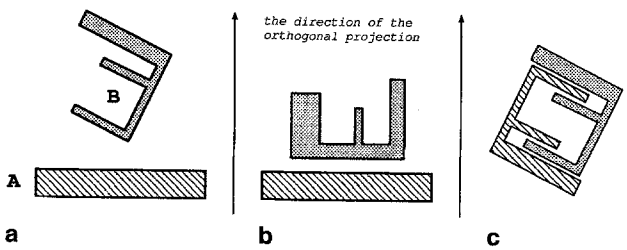
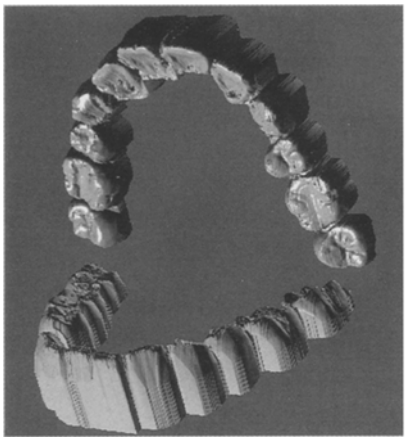
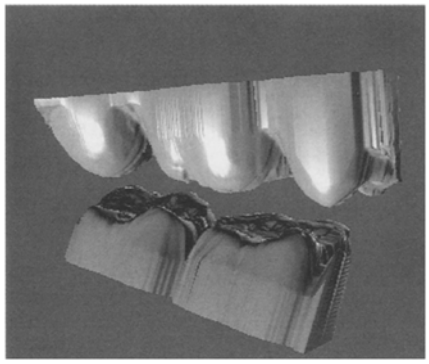


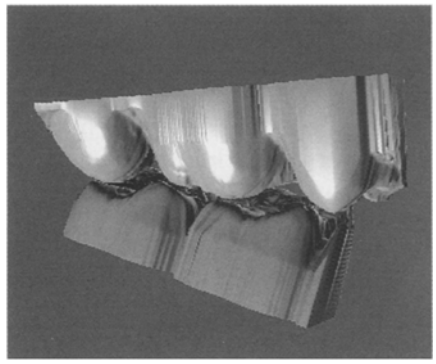
Fig. 3. Examples of test objects: **a** one π-convex object; **b** two π-convex objects; **c** a general case



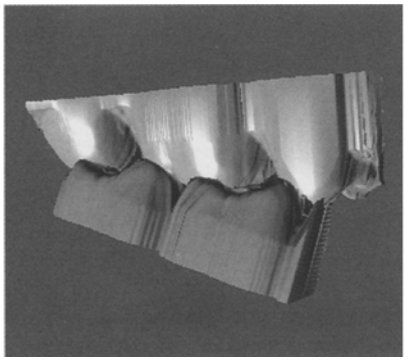
4



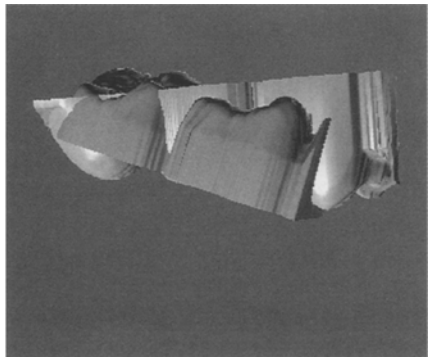
5a



5b



5c



5d

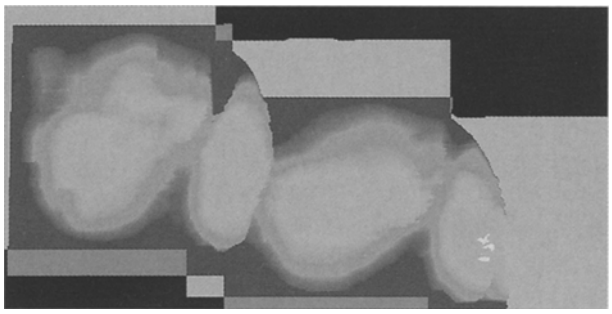
Fig. 4. General view of the computer model of human jaws

Fig. 5a–d. Four stages of an abstract motion of the lower jaw (first and second molar teeth)

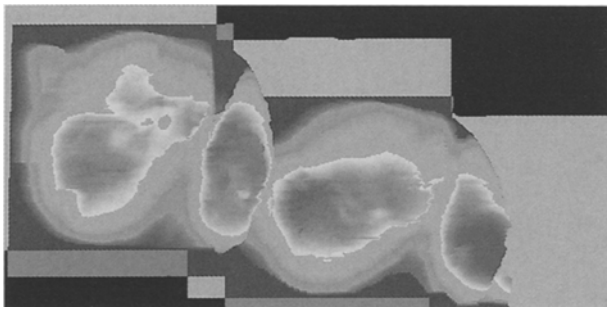
the linear complexity of their algorithm. However, this means that hardware-assisted operations on pixel masks and alpha blending did not affect the overall algorithm complexity.

We also tested the cost of our algorithm's dependency on the resolution of the frame buffer. The rendering time was quite stable for resolutions in the range of 128×128–512×512 pixels (these results indicate an overload of geometry engines and an underload of raster operations in our application), while timings of the processing of pixel masks by the application software exhibited linear growth with respect to the number of pixels.

We applied rasterization-based collision detection in a simulation of human jaw articulation (Kunii et al. 1994). At present, we only have access to the description of the shape of teeth (Fig. 4) as the height of field data measured experimentally with a mechanical scanner. In this case, collision detection can be reduced to the trivial test (1) when the direction of the orthogonal projection corresponds to the depth axis of the scanner used for the measurement of the upper or lower jaw. However, mechanical scanning requires a plaster mold of the jaws, and the measurement takes a long time. In practical applications, optical scanning for multiple viewing directions directly inside the mouth of the patient is used (Duret et al. 1988). This means that the general algorithm for detecting collisions of concave objects is needed. Having only simplified measurement data, we simulate such a situation by arbitrary positioning of the teeth with respect to the orthogonal projection direction. Additional polygons are inserted at the bottom of every tooth to make the surface closed. Figure 4 shows the general view of the human jaws exposing the complexity of possible contacts between teeth. We focus our analysis on five teeth built of more than 110 000 polygons (Fig. 5; teeth belonging to the upper jaw are brighter). An important issue is the sensitivity of the algorithm to the contact complexity. The positions of the jaws

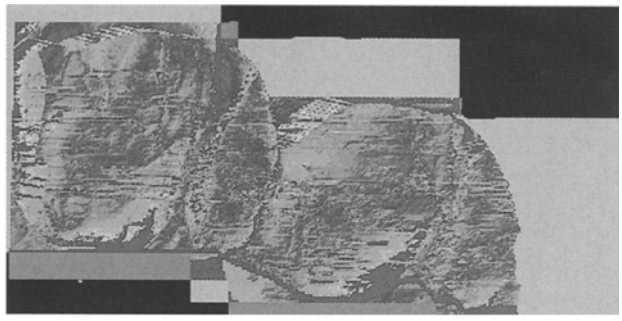


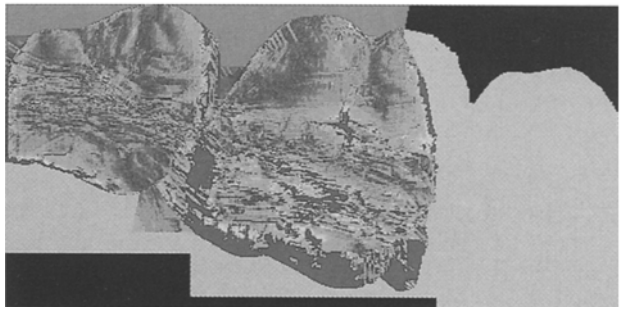
6a



6b

Fig. 6. The map of the distance between the upper and lower jaws: **a** the collision is just detected (the yellow spots); **b** the collision corresponding to the position of jaws in Fig. 5c





7

Fig. 7. Distribution of distance measurement errors for the occlusal surface of teeth for various choices of the orthogonal projection

shown in Fig. 5a (no collision) and b (collision) required only a single rendering pass, and the collision detection timings were the same, 0.30s (0.49s). Two rendering passes and a single alpha-blending pass were needed for the positions shown in Fig. 5c and d. Detecting a collision took 1.11s (1.78s) in both cases, excluding the time for sorting polygons. Of course, when the sorting is based on multipass rendering, this time will be greater and proportional to the number of layers of polygons overlapping pixels (up to eight layers in this case). Figure 6a and b presents the maps of distance between teeth corresponding to Fig. 5b and c. Gray and pink colors depict pixels only covered by the upper and lower jaws, respectively. Red and yellow colors indicate interference, while the border between yellow and green shows the contact area. In Fig. 6a the collision is just detected during free motion of the lower jaw, and the contact zone is very small (isolated pixels).

We used ray tracing as a reference method for validating the depth results provided by the rasterizing graphics hardware. We used 16 rays per pixel, and the average distance between the jaws was calculated. This distance was compared with the corresponding distance calculated with a Z-buffer (24 bits were used to encode the depth). Figure 7 (upper) presents the distribution of errors over the occlusal surface of the teeth in Fig. 5c. In general, the error is kept at a low level (RMS Error = 0.0009 mm). However, in the proximity of the edges of the occlusal surface, it grows significantly (red color in Fig. 7; $\delta_{max} = 0.06$ mm). The higher error is related to the orientation of polygons with respect to the orthogonal projection plane. The projection plane was adjusted to the orientation of the polygons with the techniques based on the item buffer (Sect. 7). Otherwise, the error can be even more significant (Fig. 7, lower).

9 Conclusions

In this paper, efficient techniques that exhibit real-time performance for detecting collisions between

complex solids have been presented. The sensitivity to the complexity of the contact between objects is very low. The method handles both convex and concave solids. The algorithm checks the π-convexity of the objects as a byproduct of scan conversion, and in many cases discussed in the paper, the concave objects can be processed completely within one or two rendering passes. The algorithm refers to the typical functionality of graphics hardware and can be implemented easily on high-end workstations. Hardware-assisted calculations constitute the main strength of the method. However, at present, a functionality missing in graphics libraries forced us to inspect the values of stencil planes and the depth sorting of polygons with application software. Since similar extensions of functionality are required by other important applications (e.g., rendering of semitransparent or CSG objects), we believe that such hardware support will soon be commonly available, further improving the efficiency of our method.

The main drawback of the method, inherent in all rasterizing graphics techniques, is the possibility of missing collisions that occur between sample points (pixels). The resolution of the frame buffer is limited, and in many cases cannot be adjusted to the requirements of the collision-detection accuracy. A hierarchical, multiresolution structure of collision maps can be built, which contains more detailed information on the regions of special interest, e.g., exhibiting a complex geometry. However, further research is needed to find out how to construct such a structure to reduce errors in detecting collisions.

Another important topic for future research is the choice of viewing parameters for scan conversion. When the variation of the orientation of the surface polygons is high, multiple directions of orthogonal projection should be used to secure high accuracy of the collision detection. An open question is the number and parameters of such multiple projections.

Acknowledgements. The authors thank Dr. M. Ibusuki for educating us about human jaw articulation. Thanks also go to Y. Shinagawa for providing the measurement data of teeth, and Y. Kesen and K. Yamauchi for their contributions to this work. Special thanks for proofreading the manuscript go to A. Pasko and C. Lecerf. We thank the Fukushima Prefectural Foundation for the Advancement of Science and Education and the Multimedia Open Network & Virtual Reality for New Education Consortium (*MOVE*) for support of the Intelligent Dental Care System project.

References

Aziz N, Bata R, Bhat S (1990) Bézier surface/surface intersection. IEEE Comput Graph Appl 10(1):50–58

Boyse JW (1979) Interference detection among solids and surfaces. Commun ACM 22:3–9

Cameron SA, Culley RK (1986) Determining the minimum translational distance between two convex polyhedra. Proceedings of the International Conference on Robotics and Automation 2:591–596

Carpenter L (1984) The *A*-buffer, an antialiased hidden surface method. ACM Comput Graph Proc SIGGRAPH '84 18:103–108

Culley RK, Kempf KG (1986) A collision detection algorithm based on velocity and distance bounds. Proceedings of the International Conference on Robotics and Automation 2:1064–1069

Duret F, Blouin JL, Duret B (1988) CAD/CAM in dentistry. J Am Dent Assoc 117:715–720

Garcia-Alonso A, Serrano N, Flaquer J (1994) Solving the collision detection problem. IEEE Comput Graph Appl 14(3):36–43

Graphics Library Programming Guide (1992) Document Number 007-1680-010. Silicon Graphics

Hahn JK (1988) Realistic animation of rigid bodies. ACM Comput Graph Proc SIGGRAPH '88 22:299–308

Hanna S, Abel J, Greenberg DP (1983) Intersection of parametric surfaces by means of lookup tables. IEEE Comput Graph Appl 3(7):39–48

von Herzen B, Barr AH, Zatz HR (1990) Geometric collisions for time-dependent parametric surfaces. ACM Comput Graph Proc SIGGRAPH '90 24:39–48

Kelley M, Gould K, Pease B, Winner S, Yen A (1994) Hardware accelerated rendering of CSG and transparency. ACM Comput Graph Proc SIGGRAPH '94 28:177–184

Kunii TL, Herder J, Myszkowski K, Okunev O, Okuneva GG, Ibusuki M (1994) Articulation simulation for an Intelligent Dental Care System. Displays 15: 181–188

Mammen A (1989) Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. IEEE Comput Graph Appl 9(4):43–55

Menon J, Marisa RJ, Zagajac J (1994) More powerful solid modeling through ray representations. IEEE Comput Graph Appl 14(3):22–35

Moore M, Wilhelms J (1988) Collision detection and response for computer animation. ACM Comput Graph Proc SIGGRAPH '88 22:289–298

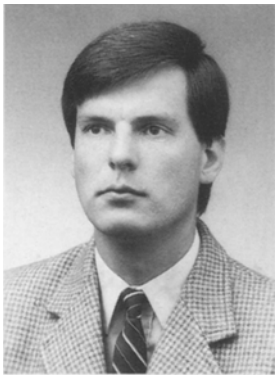
Neider J, Davis T, Woo M (1993) OpenGL programming guide. Addison-Wesley, New York

Phillips MB, Odell GM (1984) An algorithm for locating and displaying the intersection of two arbitrary surfaces. IEEE Comput Graph Appl 4(9):48–58

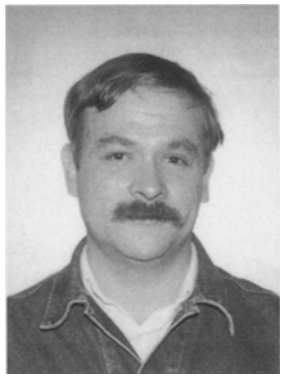
Rossignac J (1992) Accurate scan conversion of triangulated surfaces. In: Kaufman A (ed) Advances in computer graphics hardware VI. Springer, Berlin Heidelberg New York

Rossignac J, Megahed A, Schneider B-O (1992) Interactive inspection of solids: cross-sections and interferences. ACM Comput Graph Proc SIGGRAPH '92 26:353–360

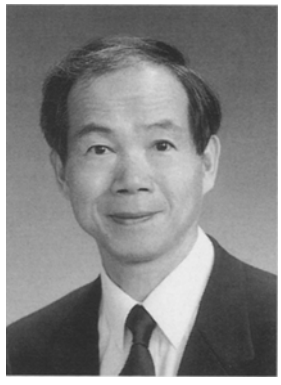
Roth SD (1982) Ray casting for modelling solids. Comput Graph Image Process 18:109–144
Salesin D, Stolfi J (1990) Rendering CSG models with a ZZ-buffer. ACM Comput Graph Proc SIGGRAPH '90 24:67–76
Shinya M, Forgue M-C (1991) Interference detection through rasterization. J Visualization Comput Animation 2:131–134
Uchiki T, Ohashi T, Tokoro M (1983) Collision detection in motion simulation. Comput Graph 7:285–293
Weghorst H, Hooper G, Greenberg DP (1984) Improved computational methods for ray tracing. ACM Trans Comput Graph 3(1):52–69
Yang Y, Thalmann NM (1993) An improved algorithm for collision detection in cloth animation with human body. Pacific Graphics '93 1:237–251



KAROL MYSZKOWSKI is an Associate Professor at the Computer Science and Engineering Laboratory, the University of Aizu, Japan. His research interests include lighting simulation, rendering, and animation. He received his PhD in Computer Science from Warsaw University of Technology, Warsaw, Poland in 1991.



OLEG OKUNEV is an Associate Professor at the Department of Education, Tver State University, Tver, Russia, and currently a visiting researcher at the Computer Science and Engineering Laboratory, the University of Aizu, Japan. His reasearch interests are in topology and its applications, in particular, in computer science. He received his PhD in Mathematics from Moscow State University in 1985.



TOSIYASU L. KUNII is currently President and Professor at the University of Aizu. His research interests include synthetic worlds, computer graphics, database systems, and software engineering. Dr. Kunii is the former President and Founder of the Computer Graphics Society, Fellow of IEEE, Editor-in-Chief of *The Visual Computer* (Springer) and the *International Journal of Shape Modeling* (World Scientific), and Associate Editor-in-Chief of *The Journal of Visualization and Computer Animation* (John Wiley & Sons). He received his BSc, MSc, and DSc in Chemistry from the University of Tokyo in 1962, 1964, and 1967, respectively.