CInDeR

Collision and Interference Detection in Real Time Using Graphics Hardware

by

David Knott

B.Sc., Simon Fraser University, 1998

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

The Faculty of Graduate Studies

(Department of Computer Science)

We accept this thesis as conforming
to the required standard

. . . . . .

. . . . . .

THE UNIVERSITY OF BRITISH COLUMBIA

October 13, 2003

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

(Signature) _____

Department of Computer Science

The University Of British Columbia
Vancouver, Canada

Date _____

Abstract

Collision detection is a vital task in almost all forms of computer animation and physical simulation. It is also one of the most computationally expensive and therefore a frequent impediment to efficient implementation of real-time graphics applications.

We describe how graphics hardware can be used as a geometric co-processor to carry out the bulk of the computation involved with collision detection. Methods for performing out this task are described in the context of two different forms of collision detection and using two separate portions of the hardware graphics pipeline.

We first demonstrate how a programmable vertex engine can be used to perform all of the computation required for a closed-form particle simulation in which the particles may impact with a variety of surfaces. The technique is used for both visual simulation and to report collision data back to an application running on the computer's CPU.

The second form of collision detection involves using frame buffer operations to implement a ray-casting algorithm which detects static interference between solid polygonal objects. The algorithm is linear in both the number of objects and number of polygons and requires no preprocessing or special data structures.

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgements

The journey toward the completion of this thesis has been a long and by no means easy one. There were many false starts, delays, and detours. I have been very fortunate, in that I have not travelled alone. My companions and fellow travellers have been constant sources of friendship and support. I could not have made it without them.

Most graduate students are lucky to have a decent supervisor for their thesis research. I somehow managed to outlast not one, not two, but *three* supervisors, each of whom was exceptional.

- First and foremost, my primary supervisor Dinesh Pai. His guidance and insight was invaluable.
- Dave Forsey was a mentor and friend during my industrial internship.
- The late Alain Fournier was inspirational in the early stages of my degree, and I dearly wish I could have known him better.

Most of my research was performed as a member of the Imager computer graphics laboratory. It was a stimulating research environment. The various students and faculty were insightful colleagues and wonderful friends.

A large portion of the work associated with this thesis was performed while I was working as an intern at Radical Entertainment. My experiences there were enlightening, and it was an amazing atmosphere in which to work. I am indebted to all of my friends and co-workers there.

For two years, my studies were largely funded by a GREAT scholarship from the Science Council of British Columbia.

The various research groups of which I was a member were funded in part by grants from the Institute for Robotics and Intelligent Systems (IRIS) and the Natural Sciences and Engineering Research Council of Canada (NSERC).

Dedication

This thesis is dedicated to my parents, who have never stopped believing.

Chapter 1

Introduction

1.1 Problem and Motivation

A vital task in almost all forms of computer animation or physical simulation is the act of *collision detection*. When solid objects are being animated, it is critical to determine if and when they are about to, or have already, come into contact with each other. Example problem domains where collision detection is almost ubiquitous include rigid and deformable body simulation, computer games, virtual reality, surgical simulation, robotics, path planning, and computer-aided design and manufacturing (CAD/CAM).

Collision detection is also usually one of the most computationally expensive operations in animation or simulation. Collision detection traditionally does not take temporal coherence into account and therefore must be performed at least once for each frame of the animation or time step of the simulation.

The simplest test for collision between two individual polygonal objects is a function of the number of polygons in the two models. For two objects with m and n polygons, respectively, the algorithm is expected to run in $\mathcal{O}(mn)$ time. Similarly, for systems involving large numbers of potentially colliding objects, the computational expense involved is particularly large. This is true regardless of the representation used for the object's geometry. For a group of n objects, the naïve approach requires $\mathcal{O}(n^2)$ pair-wise tests for collision. This is hardly efficient.

Many algorithms have been devised which improve the asymptotic running time of collision detection in both the polygonal object case and the multi-body case. These can be very efficient, but they traditionally involve specialized data constructs in addition to the geometric definitions of the objects.

This problem is further exacerbated in real-time graphics applications, where additional constraints associated with real-time computation must be taken into account. Such applications must execute at a minimum frame rate and, as a result, the resources allocated to collision detection are much more limited.

It is reasonable, then, to ask whether the computation involved with collision detection can be offloaded from the computer's primary processor and memory, and be performed somewhere else. To do so frees up CPU power for other tasks and enables the application to make use of a computational resource that might otherwise be underutilized.

In recent years, the computational power of graphics hardware has made enormous leaps, not only in speed but also in functionality. The advent of programmable graphics pipelines in particular has caused a flurry of research activity into the possibility of using graphics hardware for other forms of computation. Furthermore, graphics hardware is optimized for particular types of computation, including three-dimensional vector mathematics and image-based operations. This specialization makes collision detection algorithms good candidates for implementation on graphics hardware.

1.2 Research Contributions

This thesis shows how graphics hardware may be used to perform the bulk of the computational work involved with collision detection.

We outline how the graphics hardware can be viewed as a more general-purpose computing device. In support of this, we describe the implementation of two separate collision detection algorithms.

The first form of collision detection involves using a programmable vertex engine to perform closed-form dynamical simulation of particles whose motion paths may intersect a variety of analytical surfaces. When the main application requires no feedback about collision, the algorithm executes entirely on the computer's GPU. We also describe how, using a modification of the technique, information about the collisions can be transmitted back to the main application.

The second algorithm uses a commodity 3D graphics accelerator to realize an image-space algorithm for detecting interference between an arbitrary number of polygonal solid bodies. The algorithm handles both convex and non-convex objects and has an expected running time that is linear in both the number of objects and number of polygons involved. It also has the advantage of requiring no preprocessing, and no additional data constructs beyond the polygonal meshes that make up the objects.

1.3 Thesis Organization

This thesis is divided into five chapters.

This chapter introduced the problem that is addressed by the thesis research.

In Chapter 2, we give an overview of the general problem of collision detection and a brief outline of prior research conducted in that area. We also describe the current state of the art in programmable graphics hardware and contrast it with the traditional hardware-based graphics pipeline. The concept of using graphics hardware as a more general processing device is introduced and we outline some of the hurdles that must be overcome for it to become a reality.

Chapters 3 and 4 provide detailed case studies of the two hardware-assisted forms of collision detection.

Chapter 3 describes the closed-form particle simulation. We provide details of how the collision detection can affect the visual simulation of the particles, and can also be used to provide feedback about the state of the particle simulation to the non-GPU portion of the simulator.

Chapter 4 provides details of the image-space interference detection algorithm. The underlying methodology, a class of hardware-assisted ray-casting algorithms, is also described.

In the final chapter, we state our conclusions. We also outline some potential directions for future research.

1.4 Terminology and Typographic Conventions

In much of the literature, the term *interference detection* is used in place of collision detection. For the purposes of this thesis, we make the distinction that interference detection is used only to denote the determination of whether or not two objects are currently in contact. Collision detection, on the other hand, encompasses both interference detection and the determination of whether or not

two object are about to come into contact. In other words, we make a distinction between collision detection in static and dynamic settings. We note, however, that collision detection most often involves repeated iteration of interference detection.

We use the acronym *CPU* to denote the central processing unit or primary computational component of a computer. We also use the less common acronym *GPU* to denote the graphics processing unit, or primary computational component of a computer's graphics hardware.

Chapter 2

Background and Related Work

2.1 Collision and Interference Detection

For a comprehensive overview of collision detection techniques, we refer the reader to the surveys by Lin and Gottschalk [LG98] and by Jiménez *et al.* [JTT01].

There have been previous attempts to use graphics hardware to aid in collision detection. Almost all of these techniques have been used to find interferences between solid polygonal models, which is the subject of our work described in Chapter 4. We therefore defer an overview of hardware-assisted interference detection techniques to Section 4.1.1.

Collision detection algorithms are often classified by the types of models that they operate on. One such taxonomy is presented in [LG98] and is the one that we adopt here. It is shown in Figure 2.1. The types of models that we are most interested in are highlighted.

For the particle system collision detection in Chapter 3, we intersect the motion paths of particles with analytical surfaces such as planar sections and quadrics. In general, the geometry is most closely related to implicit surfaces.

The ray-casting algorithm of Chapter 4 detects whether the edges of rasterized polygons intersect solid polyhedral volumes. It is therefore concerned primarily with structured polygonal volumes.

2.1.1 Polygonal Models

If polygon-polygon intersection tests are used to determine whether two polyhedral objects are overlapping, then the simplest algorithm requires $\mathcal{O}(n^2)$ such tests, where n is the number of polygons. For real-time applications with models that have thousands of polygons, that many intersection tests are simply not feasible.

For convex models, there are many algorithms that are known to be quite efficient. Indeed, with hierarchical mesh representations, algorithms are known which run in sub-linear time [DK90]. These techniques all require special data structures and quite often need to subject the models to a significant amount of preprocessing.

Non-convex models are more difficult to deal with. So much so that the most typical method is to decompose the non-convex object into convex parts. Some approaches use volume decomposition [Cha84], while others use surface decomposition [EL01]. Such decompositions are typically very expensive and models therefore need to be preprocessed. This makes decomposition algorithms suitable for rigid bodies but much less so for deformable objects.

It is also quite common to perform collision detection using the bounding volumes of objects instead of using the full polygonal models. Doing so allows trivially non-colliding objects to be rejected quickly through calculations that are optimized for simple volumes. Axis-aligned boxes and spheres are particularly fast, as are implicit volumes such as cones and spheres. Oriented bounding boxes are another popular data structure [GLM96]. In general, the trend has been to attempt

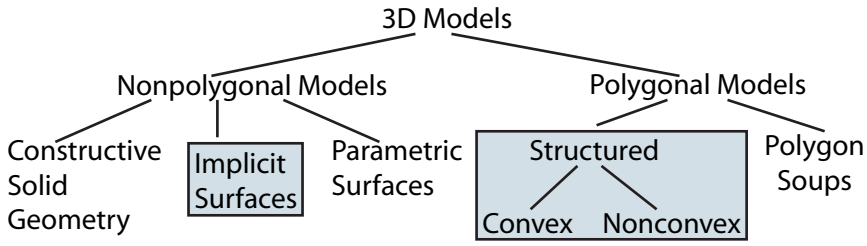


Figure 2.1: A taxonomy of 3D model representations

to find bounding volumes that approximate the underlying geometry as closely as possible. In addition, hierarchies of these volumes are often constructed in order to allow progressive refinement of intersection queries.

Hierarchical partitioning of the space in which models are embedded using structures such as binary space partitions is also a common technique.

2.1.2 Multiple Object Collision Detection

Finding collisions in large collections of objects adds another layer of computational complexity. Not only do the individual objects have to be tested against each other, but in a collection of n objects, there are $\mathcal{O}(n^2)$ potential collisions. Performing interference tests between all pairs of objects is therefore highly inefficient. This is sometimes called the "N-body processing problem".

Various techniques have been developed to aid in reducing the number of required pairwise interference tests.

For dynamical systems, many N-body techniques attempt to make use of spatiotemporal coherence. By estimating the maximum velocities and accelerations of all objects, bounds can be placed on when collisions between objects are expected to occur.

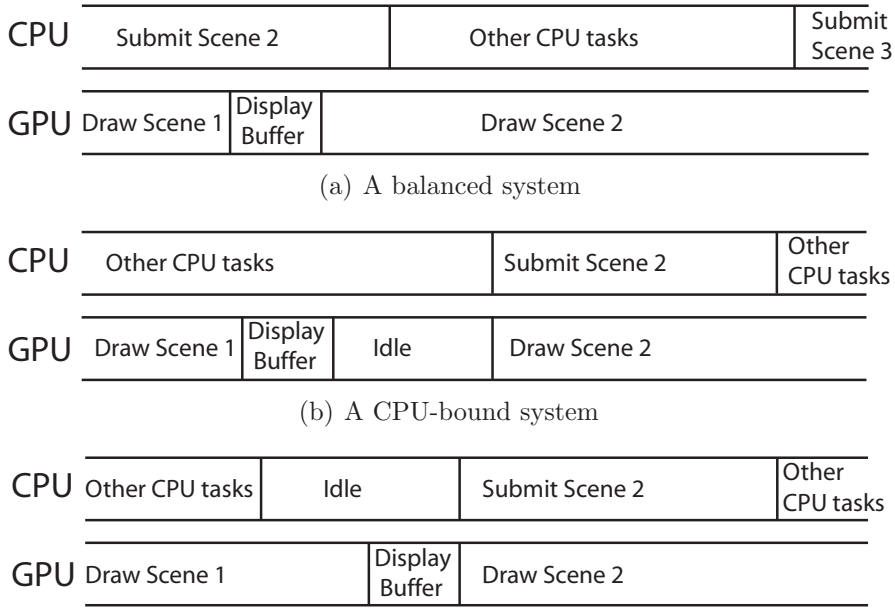
For interference queries in static environments, a common method is to perform some type of spatial subdivision. The space occupied by the objects is divided into cells and a data structure is constructed which enumerates the objects occupying each cell. Interference detection is only performed when two or more objects are found to occupy the same cell.

It is also common to perform pairwise tests on object bounding boxes in order to avoid performing full interference detection on objects that are distant from each other. When this is combined with a technique for spatially sorting the bounding boxes, N-body processing becomes more tractable [CLMP95].

2.2 Balancing the CPU and GPU loads

All computing systems have a limited amount of computational resources. Most real-time graphics applications will try to use as much of those resources as possible, and for complex simulations or large virtual worlds there is never a lack of ways to expend them. It is no easy task decide how these resources are allocated. This is especially true of applications involving user interactivity, where there are strict limits on the frame rate of the system.

In most current systems, the two major computational units are the CPU and GPU. When geometry data is submitted to the GPU, it is not necessarily immediately processed. The GPU can



(a) A balanced system

(b) A CPU-bound system

(c) A GPU-bound system

Figure 2.2: CPU and GPU load balancing

handle a large volume of data, but the CPU can submit it even more quickly than it is processed. Therefore, geometry data is usually passed from the CPU into a "staging buffer" where it waits for the GPU to process it. What this means is that the CPU can quickly submit large volumes of data and go on to perform other tasks while waiting for the GPU to use it up. Figure 2.2(a) shows a system in which the load is balanced between the CPU and CPU, and both are used to maximum capacity.

Maintaining a perfectly balanced load between the CPU and GPU is extremely difficult. It is usually the case that one of them is used to maximum capacity, and while it finishes processing, the other sits idle for a time. If a system uses the CPU to maximum capacity, we say that it is *CPU-bound*. Similarly, if the GPU is used to the maximum, we say that the system is *GPU-bound*. Figures 2.2(b) and 2.2(c) show CPU-bound and GPU-bound systems, respectively.

In the past, it was possible for many systems to be GPU-bound. However, recent rapid advances in graphics hardware have outstripped advances in CPU technology, meaning that more and more applications are now CPU-bound. Therefore, the motivation becomes even stronger to offload computation to the graphics hardware.

2.3 Graphics Hardware

The hardware real-time graphics pipeline is a complicated mechanism and an exhaustive description of it is beyond the scope of this thesis. For an excellent overview, we refer the reader to a recent book by Akenine-Möller and Haines [AMH02].

A high-level overview is shown in Figure 2.3. When we are using the graphics hardware to perform calculations, we make a distinction between the *geometry pipeline* and the *pixel pipeline*. Although both portions of the pipeline are user-controllable and at least partially programmable,

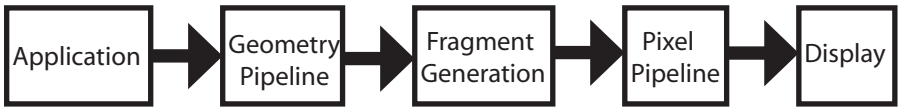


Figure 2.3: The graphics pipeline

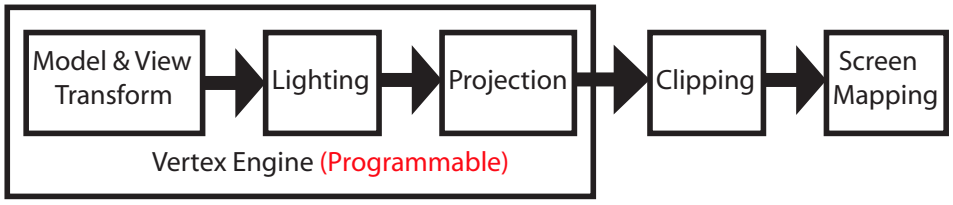


Figure 2.4: The geometry pipeline

they operate on fundamentally different data.

The geometry pipeline is concerned with manipulating vertex data and transforming it into a format that is ready for fragment generation. The pixel pipeline, on the other hand, is used to manipulate the per-pixel elements of fragments that are going to be transferred to the display.

2.3.1 The Geometry Pipeline

The operations encompassed by the geometry pipeline are shown in Figure 2.4. The geometry pipeline is primarily concerned with transforming, lighting, and providing texture coordinates for vertices. It also takes care of clipping geometry and mapping it to screen coordinates.

We use the geometry pipeline to perform calculations that pertain to data that is specified in object space or world space. In particular, point or vector data is particularly well suited to processing at this stage. We also use the geometry pipeline to set up data for further calculations that take place in the pixel pipeline.

Programmable Vertex Pipelines

One of the most significant recent developments in graphics hardware has been the introduction of *programmable vertex pipelines*.

Up until recently, the typical geometry pipeline was characterized by a *fixed-function* model. That is, geometry submitted to the GPU was subjected to a fixed set of transformation, lighting, texturing and rasterization operations. These operations were hardwired into the graphics hardware and controlled by a very small set of input variables. The geometry pipeline is now characterized by a *programmable* model, in large part due to the work described by Lindholm *et al.* [LKM01]. The programmable portions of the vertex pipeline are highlighted in Figure 2.4.

The various stages may be programmed to perform a variety of operations, vastly increasing the types of visual effects that may be constructed. More importantly, the range of computational capability is good enough to support non-visual tasks that have previously been difficult or impossible to perform on graphics hardware.

Programmable vertex pipelines are now supported as part of the standard DirectX [Mic02] API as *vertex shaders* and in the OpenGL [SA02] API as *vertex programs*.

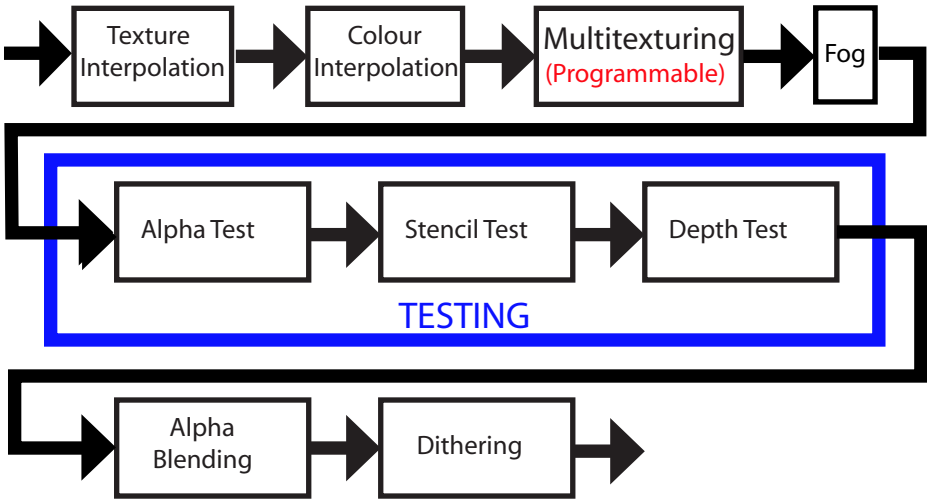


Figure 2.5: The pixel pipeline

Vertex Programming Model

We can view the vertex engine as Single Instruction Multiple Data (SIMD) machine, with vertex programs as the programming API for this machine. The instruction set for this machine is the set of vertex program instructions and the input data is characterized by the per-vertex data sent from the primary application. This is reinforced by one of the fundamental assumptions of the programmable vertex engine, which is that vertex input data has no *a priori* meaning attached to it, other than that the vertex engine will process it in some manner to produce the homogeneous screen coordinates, colours, and texture coordinates of the vertex.

2.3.2 The Pixel Pipeline

The other major user-controllable portion of the graphics pipeline is the pixel pipeline. Figure 2.5 shows its various stages. Currently, the only programmable portion is the multitexturing portion of the pixel pipeline.

We use the pixel pipeline to perform calculations on image space data. Because calculation takes place primarily on frame buffer data, the pixel processing stages are inherently two dimensional (or, at best, 2.5D if the depth buffer is considered).

For the purposes of the collision detection algorithm presented in Chapter 4, the most relevant portion of the pixel pipeline is the stencil and depth buffer operations.

2.3.3 A Streaming Model of Computation

We take a cue from the work of Purcell *et al.* [PBMH02] and view the graphics hardware as a *streaming processor* for both vertex and pixel data. An example of a system implemented for streaming computation and an overview of the concepts involved can be found in [KDR+01].

The basic idea is that data is processed as a sequential stream of independent elements. Each data element is processed in a near-identical manner. This is done by creating a program (otherwise

known as a *kernel*) that is executed on each member of the input stream. When computation is completed, results get placed into the output stream.

We can therefore view portions of both the geometry pipeline and the pixel pipeline as streaming models of computation. In the geometry pipeline data elements and kernels are represented, respectively, by streams of vertex data and vertex programs. In the pixel pipeline, they are represented by fragment data and pixel shaders (fragment programs). As an example, we can characterize the vertex programs used in Section 3.2 as kernels for a streaming implementation of closed-form particle simulation. Similarly, the pixel operations of each rendering pass in Section 4.4 are kernels contributing to an interference algorithm.

The primary advantage of a streaming model of computation is that members of the input stream can each be operated on in parallel. This is so because each element of data is independent and not reliant on any other element for processing. If performance needs to be improved, more pipelines can be added to the system. Most graphics hardware already implements the pixel pipeline in parallel. Parallel geometry pipelines are implemented on the Xbox GPU, as well as current high-end chip sets from NVIDIA and ATI.

The characterization of the pixel pipeline as a streaming processor follows in the footsteps of Fournier and Fussell [FF88], who provide a formal description of the frame buffer as a computational engine. In their model, they consider each pixel of the frame buffer to be a finite automaton. The input for the automaton consists of incoming pixel data. Register memory for the automaton consists of the various buffers (colour, depth, stencil, etc). The automaton itself is viewed as a finite-state machine, where the operations that it may perform are both functions (e.g. colour modulation) and boolean predicates (e.g. depth test) that the frame buffer hardware can perform.

2.3.4 Retrieving Computational Results from Graphics Hardware

One of the primary problems encountered in using graphics hardware to perform non-graphics computation is that the results must almost always be accessed for further processing.

The difficulty stems from the fact that graphics hardware generally has only one destination for the results of its computation, and that is the frame buffer. Therefore, the efficiency of using the GPU as an auxiliary computational device is largely dependent on how easily frame buffer memory can be accessed.

This is not so much of a problem if the results are used for subsequent card-local computations. If such is the case, then rendering to texture memory or an off-screen rendering surface (Section 2.3.5) ensures that the GPU can quickly access the data in future rendering passes. However, in many applications, the host CPU will require the data, which means that the data needs to be transferred from frame buffer memory to system memory (Figure 2.6).

Unfortunately, in most computers, the memory architecture and/or system bus is designed for maximum throughout from CPU to graphics hardware, and not for transfer in the other direction [AMH02].

For instance, in a PC, the frame buffer memory is located on the graphics accelerator. The system CPU is connected to the graphics accelerator by a bus such as the Accelerated Graphics Port (AGP) interface. However, for transferring data in the other direction, the much slower PCI interface is used, making CPU access to the frame buffer inefficient. As an example, with a such an architecture, pixels can be read from a 32-bit frame buffer at a maximum rate of 66MHz [Mau01].

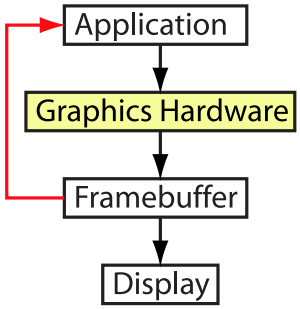


Figure 2.6: Reading the frame buffer

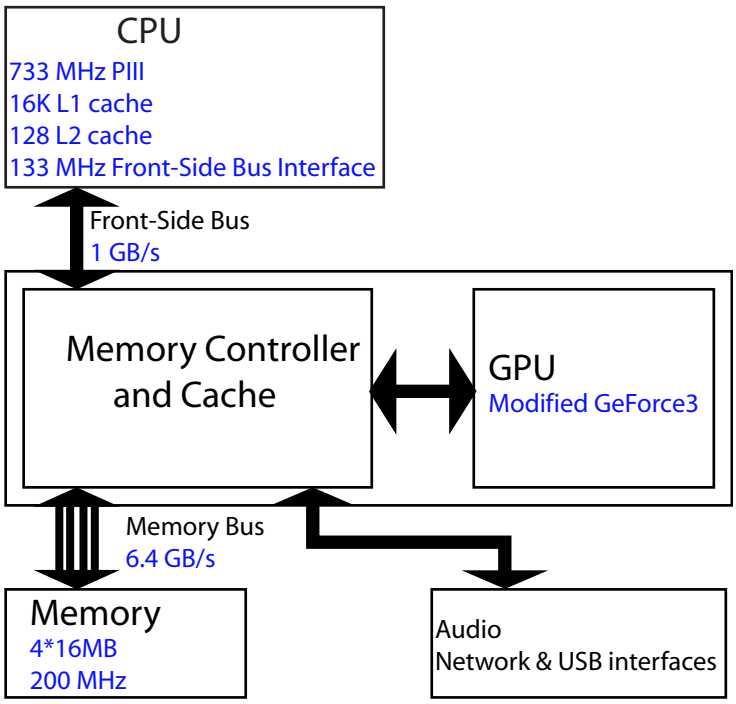


Figure 2.7: The UMA architecture of the Xbox (from [Abr00])

The result is reading a 256 by 256 pixel area of the frame buffer from a graphics accelerator will take approximately 1 msec[1].

We hasten to add that it is not always necessary to access all of the colour channels of the frame buffer. For instance, often the computational results require less than the full 32 bits of precision available in most colour buffers. If such is the case, then frame buffer access time can be vastly improved by retrieving only those data channels that store significant bits of data. Similarly, other buffers such as the stencil buffer have fewer bits of precision than the colour buffer, and access to their values is therefore faster.

Access to frame buffer memory is less of a problem in systems that use a *Unified Memory Architecture* (UMA). In such systems there is only one pool of memory which is shared by all of the components of the computer including the CPU, GPU, sound processor and other subsystems.

[1]Our experiments verify this and show roughly the same performance.

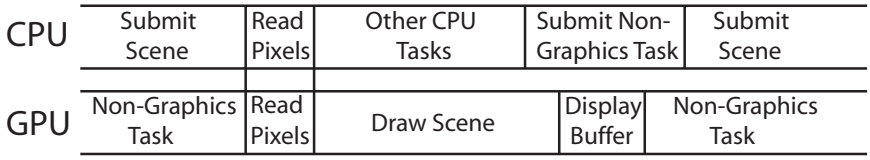


Figure 2.8: Coordinating the CPU and GPU during readback

Examples of systems using UMA are the SGI O2 and the Microsoft Xbox. An example is shown in Figure 2.7.

Our expectation is that CPU access to graphics accelerator memory will become much more efficient in the near future. The amount of interest being shown in using the GPU as a secondary computing device will likely motivate hardware manufacturers to optimize data transfer from graphics hardware to system memory.

CPU-GPU Coordination

Another difficulty with retrieving data from the graphics hardware is that there must be more coordination than usual between the CPU and the GPU. Consider the computational scheduling posed by Figure 2.8. The CPU must interrupt whatever it is doing and grab the pixels as soon as the non-graphics GPU task is complete. The reason for this is that the GPU cannot do any further work until the pixels have been retrieved, because subsequent drawing actions will corrupt the data. When the GPU is waiting for the CPU, it is in a stall state.

This problem can be at least partially alleviated by using more than one rendering surface (Section 2.3.5), if such is available.

2.3.5 Off-Screen Rendering Surfaces

Most graphics hardware does not usually associate the whole of its frame buffer memory with a single rendering context. This allows each application that makes use of the frame buffer to control and impose its own state on a virtual version of the graphics hardware.

Current graphics hardware also allows a rendering context to be associated with an *off-screen rendering surface* (sometimes called a *pbuffer*). An off-screen rendering surface is a portion of frame buffer memory that is not associated with any form of visual display. Off-screen surfaces are commonly used for real-time graphics techniques such as rendering directly to a texture map. The advantage of this is that all operations associated with off-screen surfaces are card-local. In addition, the overhead associated with visual display is eliminated.

For the purposes of our collision detection algorithms, off-screen rendering surfaces are ideal. Firstly, and mostly importantly, we do not wish to visually display the pixel data that is rendered during collision detection. Off-screen rendering surfaces allow us to access the results of card-local computation without displaying that data in a visible context.

The alternative to rendering to an off-screen surface is to render to the back buffer of a visible surface. Once the collision algorithm is through using the surface, then surface is cleared and the visible data is rendered before swapping the buffer to the front for display. The problem with this technique is that reading pixel data from the surface causes the graphics hardware to cease rendering to that surface until the pixel read is complete. In addition, reading pixels is a slow

operation. This means that there is a long period of time during which the visible surface is unavailable for rendering. For many applications, such as video games, where there is only one visible surface, the stall engendered by this is unacceptable. By rendering to an off-screen surface, we at least partially alleviate this stall in the graphics pipeline.

We emphasize, however, that rendering to an off-screen surface does not eliminate or significantly reduce the computational overhead associated with transferring information from card-local frame buffer memory to the computer's main memory.

2.4 Geometric Computations with Graphics Hardware

The rapid increase in the capabilities of graphics hardware have spurred numerous researchers to investigate using it for non-graphics purposes. One of the more active threads of research in this area has been using graphics hardware to perform geometric computations. Lin and Manocha [LM02] provide a good overview of recent results.

Geometric problems are particularly well suited to this form computation assistance. Hardware-based transformation engines are optimized for calculations on geometric data. For instance, programmable vertex pipelines are optimized for vector-based mathematics and are able to perform calculations such as vector dot-products in a single GPU cycle, or a matrix-vector multiply in as little as three GPU cycles. Fixed-function vertex pipelines are even more efficient at transforming vertex data. In addition, many graphics accelerators have several geometry engines, making graphics hardware even more useful for geometric calculations on streaming data or independent sets of geometric data.

Graphics hardware is also very good at performing image-space geometric computation by using the advanced capabilities of modern frame buffers. Pixel pipelines and fragment programs allow almost arbitrarily complicated calculations to be performed on frame buffer pixel data.

Chapter 3

Particle System Collision Detection

This chapter develops a technique for performing collision detection between dynamically simulated particle primitives and groups of implicitly defined objects. We describe how this collision detection can be performed using a programmable vertex engine.

We use the particle collision detection in the context of visual simulation, which requires no CPU intervention. We also provide a method for relaying collision information back to the CPU for further processing, using a construct called the *impact map*.

3.1 Background

3.1.1 Particle Systems

Particle systems as modelling primitives were first introduced to the computer graphics literature by Reeves [Ree83, RB85].

For the vast majority of applications, a particle is viewed geometrically in one of two different ways: either as representative point in some continuum, or as a description of the dynamic state of some larger solid.

In the context of computational modelling of continuous dynamical systems, particles are typically used to describe the state of individual points within the continuum. In other words, a collection of particles describes a discretization of some continuous system. This is necessary because of the limitations inherent in modelling a continuous system on a non-continuous device (i.e., the computer). Examples of such applications include simulation of deformable objects [TF88], liquids and gases [Sta99], and other natural phenomena. There has already been research into using graphics hardware to aid in such computations, as evidenced by the recent work of James and Pai [JP02]. Particles are also commonly used in geometric modelling for tasks such as sampling implicit surfaces and volumes [WH94].

Particles can also be used to describe dynamic states of large collections of individual objects. For example, particles are commonly used to model snow, rain, fireworks, or other phenomena that consist of large numbers of individually replicated geometric objects [Ree83].

Our particle collision detection is used in the context of simulations of the second type. A sample application included detecting collisions of hailstones against other objects. However, there is nothing to preclude us from modelling simulations of the first type. For instance, we can easily envisage performing collision detection for surface elements of a soft object that is undergoing some procedural deformation.

In many applications, particles can self-replicate, or spawn other particles. We do not model particles of this type. This is because particle state is computed using a programmable vertex pipeline. This pipeline has the restriction of allowing the state of only a single vertex to be computed for each set of data passed to the hardware.

It is important to note that if we are using particles to represent the states of individual objects, then each particle is used to represent an object that has its own geometric structure. For "small" particles, it suffices for us to compute the equations of motion using the centre of mass of the particle as the particle's position. Each vertex of the object is displaced from the computed position of the centre of mass by an amount equal to the vertex's object space position. Given additional information, such as the angular velocity of the particle's coordinate frame, we could give the particle a more complex motion at the cost of more computational expense for the GPU.

Particles can also contain other state information besides data representing geometry or motion. For instance, if colours or texture coordinates need to be animated, these features can be set as part of the particle's state.

3.2 Particle Simulation Using Graphics Hardware

3.2.1 Formulation of Particle Dynamics

The motion of particles is completely determined by initial conditions, with the exception of the calculation of impacts with (and possible response to) interfering bodies. We therefore let the instantaneous three-dimensional position, $\mathbf{p}$, of a particle be given by a parametric equation,

$$\mathbf{p} = \boldsymbol{g}(t), \tag{3.1}$$

where t is the global simulation time.

The initial attributes of each particle, such as position and velocity, are passed as data to the programmable vertex engine. In addition, particles are "regenerative" in the sense that they are created with an initial state, simulated for a finite period of time and then expire, at which time they are "reborn". When a particle is reborn, it may be recreated with altered starting conditions. The key to altering starting conditions is the observation that since a particle's life span is periodic in nature, then we can count how many generations a particle has undergone. This generation count can be used as input to a method for procedurally altering the particle's starting conditions.

The advantage to using closed-form equations of motion is that a particle's defining parameters are specified once and never change. This allows us to create graphical simulations in which the computer's CPU need never participate, except to pass data to the graphics accelerator at each frame of the animation.

3.2.2 Formulation of Collider Objects

We allow particles to impact with *colliders*, objects whose surfaces we implicitly define by some equation

$$f(\mathbf{p}) = 0, \tag{3.2}$$

where $\mathbf{p}$ is a point on the surface.

Configuration of Colliders

It is worth noting that the configuration of multiple colliders with respect to the motion of a particle is largely irrelevant. By this we mean that if a particle's motion path intersects more than

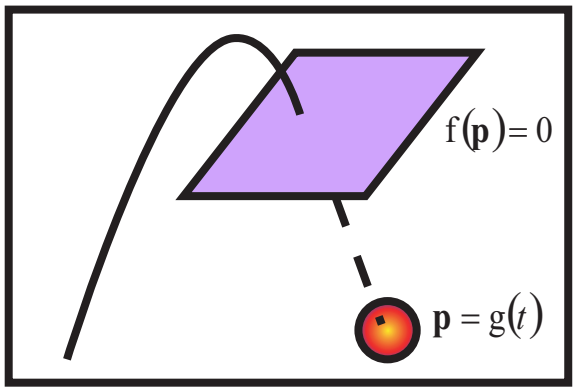


Figure 3.1: A particle whose motion intersects a planar collider

one collider, then it does not matter which collider is impacted first. When a particle is capable of colliding with multiple objects, then we compute the time of collision with each object and use only the results of the earliest collision.

3.2.3 Detecting Collisions

Combining the two previous equations, the intersection of a particle's motion with the surface is then described by

$$h(t) = f(\boldsymbol{g}(t)) = 0. \tag{3.3}$$

This formula implicitly gives us the global simulation time, t, at which a particle impacts a collider.

We detect a collision by comparing the computed collision time against the global simulation time. If the computed collision time is the lesser of the two, then we know that a collision has already occurred.

We can also process collisions even further, in order to detect collisions with more complex objects. This is done by associating with each collider one or more functions $c(\mathbf{p}) > 0$. These are used to test whether or not the computed collision point matches some constraint. This is useful in detecting collisions with bounded planar sections, or objects with holes, for example.

3.2.4 Examples of Particle Dynamics

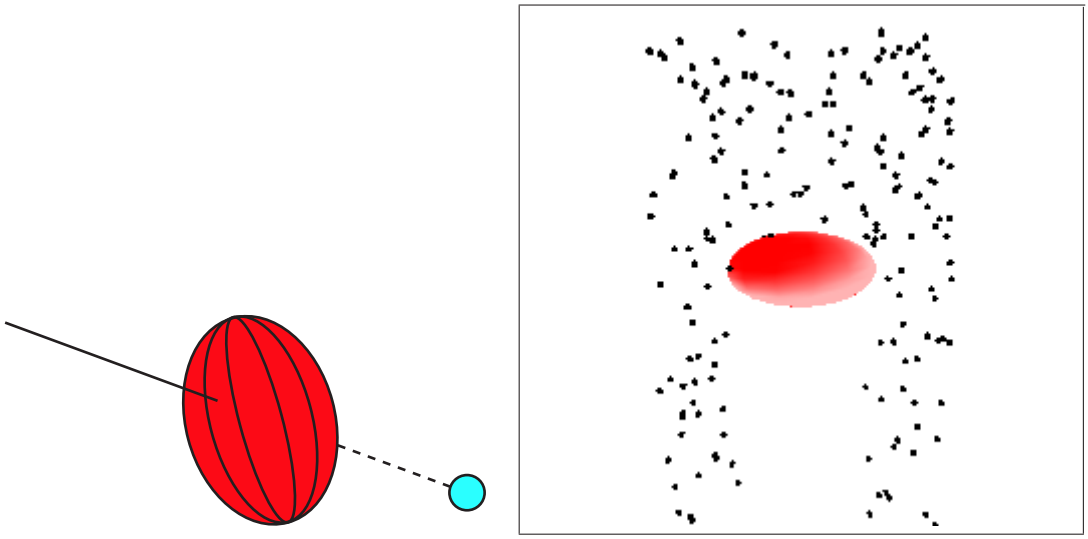
We have implemented simulations of particles that have both first and second order dynamics.

Particles with Second Order Dynamics

We simulate particles that obey second order dynamics.

For a particle with initial position x_0 and initial velocity v_0, subject to constant acceleration a, the instantaneous position of the particle is given by:

$$\boldsymbol{g}(t) = (\frac{1}{2}\mathbf{a}t^2) + \mathbf{v_0}t + \mathbf{x_0} \tag{3.4}$$



(a) (b)

Figure 3.2: Particles whose motion intersects a quadric collider

We detect collisions between second order particles and colliders that are bounded planar sections. The unbounded surface of a collider is then given by

$$f(\mathbf{p}) = \mathbf{N} \cdot (\mathbf{p} - \mathbf{p_0}) = 0 \tag{3.5}$$

where $\mathbf{N}$ is the surface's normal, and p_0 is any point on the surface.

The collision time is then given implicitly by

$$h(t) = f(\boldsymbol{g}(t)) = \mathbf{N} \cdot ((\frac{1}{2}\mathbf{a}t^2) + \mathbf{v_0}t + \mathbf{x_0} - \mathbf{p_0}) = 0 \tag{3.6}$$

which is a quadratic equation in t.

Solving a quadratic equation using vertex programs is easy. It is also a scalar operation, meaning that using vector operations we can solve four such equations simultaneously.

We bound the planar sections that particles may collide with by using the additional constraint, $c(\mathbf{p})$. For example, intersections with circular discs require $c(\mathbf{p}) = r - |\boldsymbol{g}(t) - \mathbf{q}| > 0$, where r and $\mathbf{q}$ are, respectively, the radius and centre point of the disc.

Particles with First Order Dynamics

We can also simulate particles that obey first order dynamics.

For a particle with initial position $\mathbf{x}_0$ and initial velocity $\mathbf{v}_0$, the instantaneous position of the particle is given by

$$\boldsymbol{g}(t) = \mathbf{v}_0 t + \mathbf{x}_0 \tag{3.7}$$

We can detect collisions between first order particles and colliders that are described by second order surfaces such as natural quadrics. These colliders are objects such as ellipses, paraboloids, hyperboloids, and elliptical cylinders and cones. This is illustrated in Figure 3.2(a).

Given $\mathbf{p} = \begin{bmatrix} x & y & z & 1 \end{bmatrix}^T$, the general form of a quadric surface is given by

$$f(\mathbf{p}) = Ax^2 + By^2 + Cz^2 + 2Dxy + 2Exz + 2Fyz + 2Gx + 2Hy + 2Iz + J = 0$$

which can be expressed in matrix form as [Han89]:

$$f(\mathbf{p}) = \mathbf{p}^T \mathbf{Q} \mathbf{p} = 0 \tag{3.8}$$

where

$$\mathbf{Q} = \begin{bmatrix} A & D & E & G \\ D & B & F & H \\ E & F & C & I \\ G & H & I & J \end{bmatrix}$$

Collision time is then given implicitly by:

$$h(t) = f(\mathbf{g}(t)) = \mathbf{g}(t)^T \mathbf{Q} \mathbf{g}(t) = (\mathbf{v}_0 t + \mathbf{x}_0)^T \mathbf{Q} (\mathbf{v}_0 t + \mathbf{x}_0) = 0 \tag{3.9}$$

This is again a quadratic equation in t and easily solved using vertex programs.

For example, for an ellipsoidal collider with axis lengths $\mathbf{r}_x$, $\mathbf{r}_y$, and $\mathbf{r}_z$, the collider's surface is given by:

$$f(\mathbf{p}) = \frac{\mathbf{p}_x^2}{\mathbf{r}_x^2} + \frac{\mathbf{p}_y^2}{\mathbf{r}_y^2} + \frac{\mathbf{p}_z^2}{\mathbf{r}_z^2} - 1 = 0$$

or

$$\mathbf{Q} = \begin{bmatrix} 1/\mathbf{r}_x^2 & 0 & 0 & 0 \\ 0 & 1/\mathbf{r}_y^2 & 0 & 0 \\ 0 & 0 & 1/\mathbf{r}_z^2 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

A sample scene with particles impacting a quadric collider is shown in Figure 3.2(b).

3.3 Visual Simulation

For visual simulation, it is not enough to simply calculate whether or not the particles have impacted another object. The particles should actually *do* when a collision is determined to have occurred. We currently have two ways of letting particles react to collisions: they are either eliminated or undergo a limited number of bounces.

3.3.1 Post-Collision Particle Elimination

The collision detection calculation lets us know if any collision has occurred at some previous point in a particle's current iteration. If a collision has occurred then we can eliminate the particle from

Figure 3.3: Particles disappear after collisions

visual display in some manner. This is usually accomplished by assigning the particle some reference alpha colour and using alpha testing to direct the hardware not to display any fragment with that alpha value. Alternatively, we can translate the particle to some point that is known to not be visible in the scene.

Figure 3.3 shows a scene with a variety of colliders that are all blocking particles.

3.3.2 Collision Response

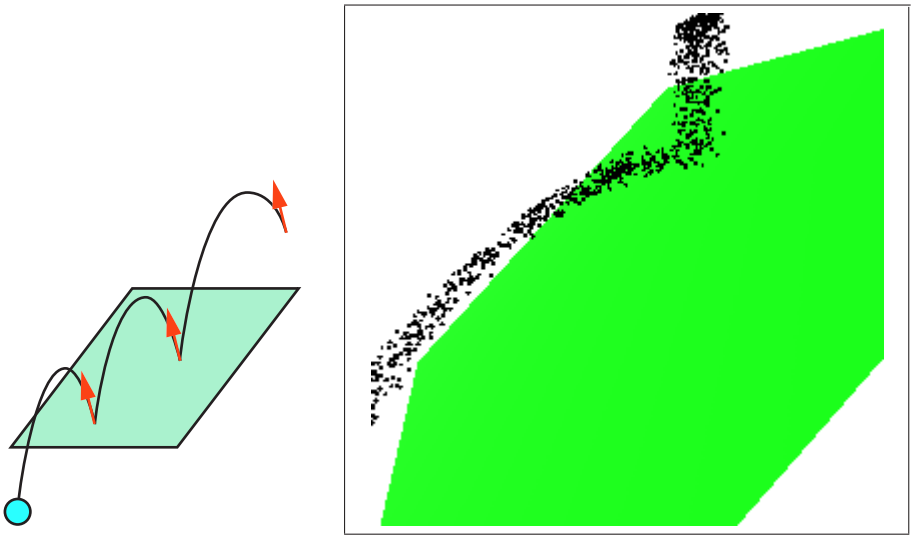
It is possible to simulate particles with a limited amount of collision response.

The primary difficulty in modelling collision response is that when a collision is detected in a vertex program, it is not possible to use the particle's attributes to record the collision. This is because vertex programs treat per-vertex data as read-only information.

It is therefore not possible to permanently modify a particle's position or velocity in response to a detected collision.

Therefore, when modelling collisions with response, we check if collision has occurred at any time in the past. If a collision was found, then temporary hardware registers are used to record the time, position, and velocity of collision restitution. The particle's motion is then recomputed using these updated simulation parameters (Figure 3.4(a)). The collision detection is also recomputed with the new parameters.

This process can be repeated several times. Since the hardware that we used does not support looping in vertex programs, our implementation allowed only a limited number of collision responses. Using a DirectX 8 vertex shader or OpenGL 1.4 vertex program implementation, there are enough instructions slots available to compute three bounces. This limitation could be partially overcome by using hardware that supports the newer release of DirectX 9, which supports limited looping in



(a) (b)

Figure 3.4: Particles exhibiting collision response

vertex computation.

3.4 Collision Feedback

3.4.1 Retrieving Collision Information

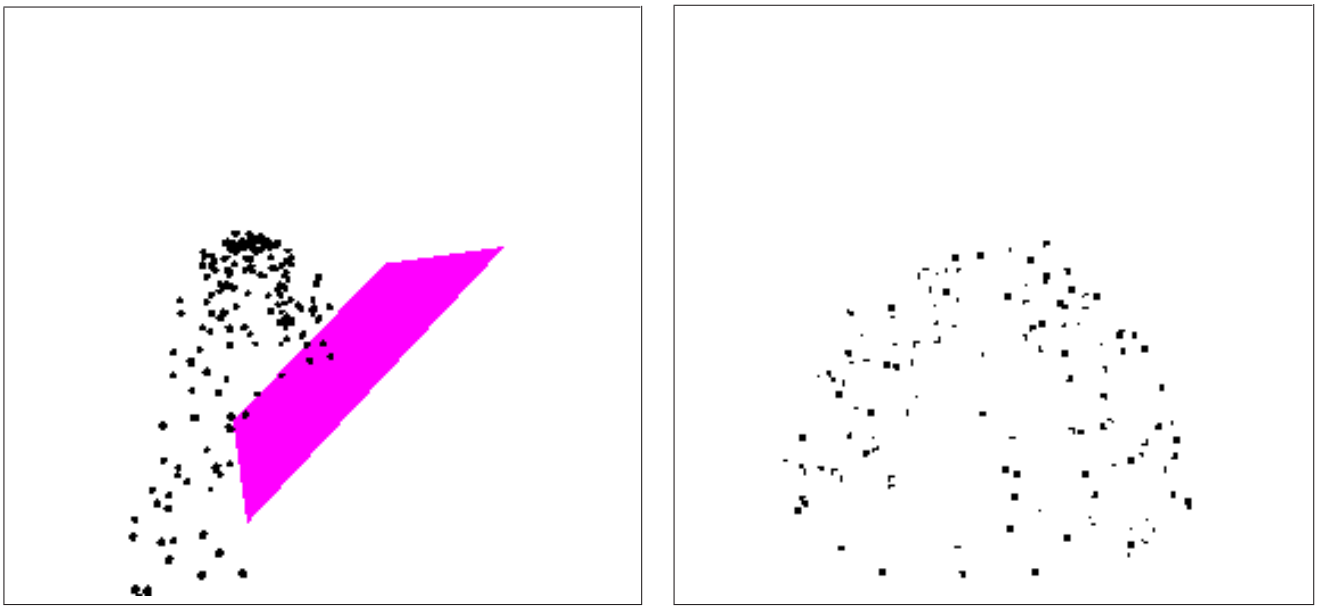
Using the graphics hardware to report collisions to an application running on the computer's main CPU is considerably harder than using it to simulate particles for purely graphical purposes. To do so, we must find some way to transmit information back from the graphics hardware. As we described in Section 2.3.4, the only way to access such information is through the frame buffer.

Therefore, to retrieve collision events from the frame buffer, we introduce a device that we call the *Impact Map*.

3.4.2 The Impact Map

When we are using vertex programs to calculate impact events, we do not submit to the hardware the same particle geometry as we use for visual simulation. Instead, we direct the GPU to render a single point primitive with the same simulation parameters as the particle. Furthermore, we do not render this point primitive as part of the graphical scene. Instead, we redirect rendering to another graphics context that contains a logical device that we will henceforth refer as the impact map. An example of a context for the impact map might be an off-screen rendering surface, such as is often used for purposes like rendering to a texture map.

The impact map is basically a two-dimensional representation of the locations on colliders at which particles have impacted. At each simulation step, the GPU vertex program computes whether or not each particle has impacted a collider at some time between the previous and current frames of the animation. If a collision is detected, then a two-dimensional representation of the exact impact



(a) (b)

Figure 3.5: A collider and its associated impact map. The impact map colours are enhanced for explanatory clarity.

location is calculated. The 2D representation is then renormalized to fit the dimensions of the viewport containing the impact map and the point primitive is drawn at that location. If no impact occurred, then the point primitive is transformed to some location outside of the impact map's viewport. This causes the graphics hardware to cull the impact point during fragment generation.

The impact map for a single rectangular collider is shown in Figure 3.5.

3.4.3 Encoding Impact Information

When we detect a collision and render it into the impact map, we can also attach some auxiliary data about the collision. This data is stored in the colour channels of the pixel that the collision is mapped into.

For instance, given three colour channels, we may choose to encode the identity of the collider that was impacted, the exact collision time, and the magnitude of the velocity with which the particle hit the collider.

Multiple Impacts

Note that it is possible for more than one collision to map into the same pixel of the impact map. For many applications, such as rendering "paintball" or "splat" textures (see Figure 3.6) at the impact points on colliders, this is perfectly acceptable. In many cases the results of one collision may mask the results of another.

Alternatively, multiple collocated collisions could be naïvely combined in the impact map by using an additive blend of one or more colour channels. This blend is easily accomplished using the

Figure 3.6: A table with "splat" textures to indicate collisions

pixel shading stage of the graphics pipeline[1].

3.5 Key Issues

3.5.1 Spatiotemporal Resolution of Collision Feedback

The precision with which collisions are reported by the graphics hardware is controlled by two factors:

- The pixel resolution of the viewport into which the impact map is rendered. This impacts the spatial resolution of collision feedback
- The maximum frame rate at which the impact map can be rendered. This impacts the temporal resolution of collision feedback.

Spatial Resolution

All collisions are reported back from the graphics hardware via the impact map. Since the impact map is rendered into a viewport of fixed pixel resolution, then the impact map itself will be constrained to that same resolution. Since the impact map is simply a two-dimensional parameterization of the surface of a collider, then the collider-space resolution of collision feedback is similarly constrained.

[1] If the pixel shading stage is programmable, then collisions can be combined in even more interesting ways.

As an example, consider a rectangular planar collider, of dimensions M by N. Such a collider maps into the impact map via a simple scale transformation. If the impact map has a resolution of p pixels by q pixels, then the spatial resolution of collision feedback is limited to $\frac{M}{p}$ by $\frac{N}{q}$.

A serious implication of decreased spatial resolution is that the algorithm is more prone to mapping multiple collisions to a single pixel. However, as explained in Section 3.4.3, this may not be a significant problem in all applications.

Also, if the relationship between impact map coordinates and collider surface coordinates is non-linear, then the resolution of collision feedback will be non-constant across the surface of the collider. For most applications, this type of artifact would be unacceptable, and we therefore attempt to maintain a linear relationship, if at all possible.

Increased spatial resolution is gained by enlarging the impact map. Rendering into a larger impact map is not appreciably slower than rendering into a smaller impact map. This is because every particle still gets "rendered" only once and when a collision occurs is still rendered as only a single pixel. Therefore, the vertex transformation, rasterization, and pixel fill rates of the algorithm are the same regardless of the size of the impact map. However, when the impact map size is increased, more pixel data must be transferred back from the graphics hardware over the system bus, and subsequently inspected by the CPU. Therefore, the spatial resolution of collision feedback is increased at the expense of extra temporal latency.

Temporal Resolution

The graphics hardware performs collision detection at discrete intervals, governed by how fast it can render and retrieve an impact map. Therefore, at a coarse level, the resolution of collision detection is equal to impact map's maximum sustainable frame rate.

We can, however, make use of the fact that the simulation and collision detection is computed in closed-form. This means that not only can we check whether or not collision has occurred between the previous and current frames, but we can also calculate exactly (to within 8 bits of precision) when the collision happened. We calculate this value as an offset between the two frames and report it through one of the colour channels of the impact map.

Latency in Collision Feedback

Since the impact map's frame rate determines how quickly collision detection can be done, it is also the prime source of latency. Even though the graphics hardware can calculate collision times with relatively high precision, latency results from the rate at which the results of those calculations can be reported. Since those results are reported by reading the frame buffer, latency can be viewed as the maximum amount of time that it takes to render and read an impact map. For scenes with very large numbers of particles or a large impact map, this latency can be fairly high.

However, we can again make use of the fact that the calculations are closed-form. Since the behaviour of particles is entirely predetermined, we can "look ahead" into the future when performing collision detection. If the global time for collision detection is offset by an amount equal to the maximum expected latency, then collision events will be reported in a more timely fashion.

3.5.2 An Alternative to the Impact Map

An alternative to using the impact map would be to assign every particle one or more pixels in the viewport. The pixels assigned to a particle would be reserved exclusively for the results of that particle's collision detection computations. In this case, a particle's impact location would not be inferred the frame buffer position of a pixel. Instead, impact location could be encoded in the colour channels of one of the particle's pixels.

This approach has several advantages. Since the number of particles is typically much fewer than the number of pixels in the impact map, the amount of frame buffer memory that needs to be transferred to the CPU could potentially be minimized.

Assigning each particle its own pixels could also overcome many difficulties related to spatial resolution (Section 3.5.1). However, in order to improve resolution, more than one pixel is needed to encode position. This is because the naïve mapping of $\{X, Y, Z\} \Rightarrow \{R, G, B\}$ allows only 8 bits of resolution for each of X, Y, and Z. This is equivalent to the resolution afforded by a relatively small 256 by 256 pixel viewport. The use of floating point frame buffers may also aid in overcoming this limitation.

In addition, if multiple pixels are assigned to a particle, then rendering time will be vastly increased. Since vertex programs can generate only one set of vertex data, then rendering to multiple pixels will effectively require each particle's collision detection to be computed multiple times.

3.6 Results

3.6.1 Implementation

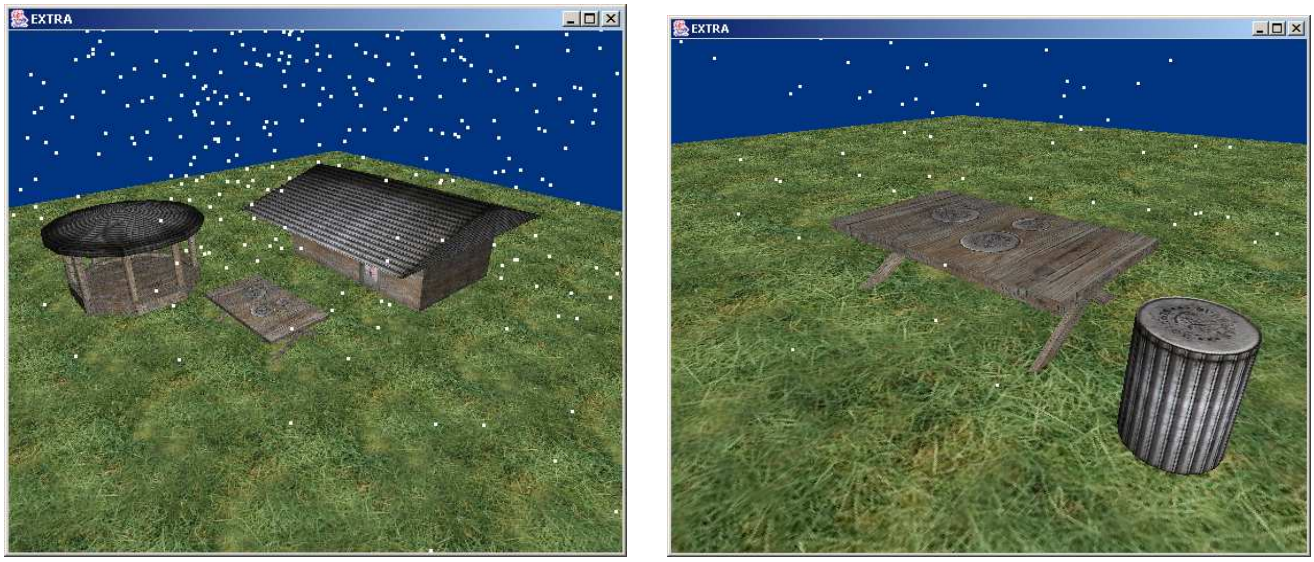
We implemented the particle system collision detection algorithm in the Java programming language, using OpenGL [OS99] as the real-time rendering API. The interface between Java and OpenGL was provided by the *OpenGL for Java* [GL4] third-party interface layer.

The initial development and testing was done using a computer with an 800 MHz Pentium III CPU, and a graphics accelerator that used the NVIDIA GeForce3 chip set.

We implemented the collision detection in such a manner that the test application consisted of three separate threads, each performing a distinct function:

- Rendering the graphical scene
- Rendering and retrieving the collision information through the impact map
- Processing and making use of the collision information (as, for example, in sound simulation)

Our justification for this is that the particle simulation is computed in closed-form, and therefore requires minimal intervention from the main application. The advantage, of course, is that the thread that uses the collision information can take full control of the computer's main CPU while the graphics hardware computes the data. In a single-threaded application, the CPU would be sitting idle while waiting for GPU computation to complete. In addition, multi-threading allows the application to take full advantage of computers that have more than one CPU.



(a) (b)

Figure 3.7: Hailstone particles colliding with outdoor objects

3.6.2 Simulation of Hail and Rain

We have implemented an application that uses hardware-assisted collision detection for both visual simulation and feedback to an application running on the computer's main CPU. The application is an audio-visual simulation of hail falling on an outdoor scene.

The hail is simulated as particles with motion defined by first-order dynamics. We perform collision detection between each hail particle and a variety of objects. For instance, Figure 3.7(a) shows a scene where the hail collides with a house and gazebo with metal roofs and a picnic table. Figure 3.7(b) shows a scene where the hail collides with a garbage can and a picnic table.

For visual simulation, the hailstones are dispensed with when they impact with collider objects in the scene. In Figure 3.7 colliders include building roofs and the tops of the table and can.

We use the collision feedback to drive an audio simulation that uses modal synthesis to generate sound whenever a collider is struck by a particle. The audio synthesis is based on an extension to the work of van den Doel *et al.* [vdDKP01].

Chapter 4

Interference Detection through Ray-Casting

In many instances of animation and simulation, the types of objects being modelled are primarily polygonal solids. This is especially true in fields such as CAD/CAM, rigid body simulation, and character animation.

This chapter develops an image-space method for detecting interference between solid polygonal bodies. The algorithm makes use of virtual ray casting to determine which portions of the edges of the solids lie within the volumes enclosed by other solids.

The technique exhibits a number of features which, to our knowledge, no other interference detection algorithm has successfully combined:

- Processing is done with the aid of commodity-level graphics hardware.
- Convex and non-convex geometry can be handled.
- An arbitrary number of objects can be handled and identified.
- Intersection tests are performed on the geometry itself, not on an approximation to the surface.
- No special data structures are required.
- No preprocessing is required.
- The algorithm's expected asymptotic running time is linear in both the number of objects being tested and the number of polygons comprising the objects.

In the context of the following discussion, a polygonal solid is deemed to be a closed manifold, enclosing a finite volume. Unless otherwise noted, the solid may be non-convex and may contain hollow regions.

4.1 Background and Related Work

4.1.1 Interference Detection

There have been a number of previous attempts at using graphics hardware to aid in interference detection.

Perhaps the best known example of using graphics hardware to detect interference regions is the work of Rossignac *et al.* [RMS92]. They use the depth and stencil buffer capable hardware to aid in the inspection of cross-sections of computer-modelled mechanical assemblies. Clipping planes are

moved through volumes occupied by the solid assemblies. Rays are cast toward points on a clipping plane. A point is known to be within the solid if a ray passes through an odd number of polygonal faces before reaching the point.

Shinya and Forgue [SF91] reported some early results of using a hardware depth buffer to support interference detection. They start with the assumption that all objects are convex. For each pixel, a list of the maximum and minimum depth values of each object is stored. These lists are then sorted. If any object's z_{max} and z_{min} values are not adjacent in the sorted lists, then two objects are interfering. The hardware is used to calculate the z_{min} and z_{max} depth maps of each object. The main drawback of this approach is the huge overhead of repeatedly copying the depth buffer and then sorting the depths for each pixel. Storing many depth maps also requires huge amounts of memory.

Myszkowski *et al.* [MOK95] describe using the depth and stencil buffers in conjunction to detect inference. Of all the previously reported results, their work most closely resembles our own. As with our algorithm, they use the stencil buffer to store a running count of how many solid objects a ray enters and leaves before reaching a surface point of another object of interest. As in [SF91], their method is applicable only to objects that are convex in the direction of the rays being cast. In addition, their algorithm does not work for more than two objects and must be repeated for every object pair that requires interference detection.

The work of [MOK95] was subsequently expanded on by Baciu and Wong [BW97, BWS98]. Their primary contribution was to extend the techniques developed in [MOK95] to compute the area of the region of overlap between two interfering solids. Like the previous work, their algorithm is applicable only to individual pairs of convex objects. In order to minimize the number of object pairs being tested, they use axis-aligned or oriented bounding box tests [GLM96] to winnow down the number of objects that may be interfering with each other. One significant benefit of performing only pair-wise interference tests is that the ray-casting viewport's size and position are optimal for the objects being tested. Even so, using such a technique to compute the areas of interference is of limited utility, since such a calculation is limited to the precision imposed by frame buffer resolution.

Vassilev *et al.* [VSC01] have used an image-space depth and colour buffer technique for detecting collisions in cloth animation for computer-generated characters. They first render two depth maps, from the front and back, associated with the character. For the clothing, the coordinates of the cloth vertices are transformed into the image space of the depth maps. The x and y image-space coordinates of the cloth can then be used to index into two depth maps. If, for either depth map, the z value of the cloth vertex is less than the depth map value, then they conclude that the cloth has intersected the character's body. This method works in practise, but has the drawback that if a body part occludes another body part in the depth map, then collision between the cloth and the occluded body part can be missed by the algorithm.

Interestingly, the work of [VSC01] has also produced a method for using the graphics hardware to aid in collision response calculations. They do so by rendering normal maps and velocity maps for the characters' bodies, along with the depth maps. The normal maps are produced by specifying, for each body vertex, that its colour is equal to its normal. By smooth shading the character, this ensures that, in the rendered image, the colour of any body point is that point's interpolated normal. Similarly, by mapping a vertex's velocity vector to its colour, a velocity map of the body can be rendered. When a cloth/body collision is detected in the depth map, the corresponding body normal and velocity can be extracted from their respective maps for use in collision response calculations.

More recently, Wagner *et al.* [WSM02] have used an image-based approach for collision detection within the context of a physical simulator used for eye surgery. With the assumption that a retinal membrane is convex in directions perpendicular to the movement of a surgical instrument, they compare the rendered depth buffers of the membrane and instrument. They also use a colour buffer-based feature identification method similar to our own to find the facet of the membrane which was penetrated by the instrument. In addition, they use the difference in the depth buffers to give a crude approximation of the displacement vectors of membrane features.

Another approach to using graphics hardware to aid in collision detection in the context of surgery is presented by Lombardo *et al.* [LCN99]. Their method is distinguished by the use the picking and feedback modes of OpenGL to determine which features of a simulated organ are in contact with a model of a laparoscopic surgery tool. However, these features are usually not fully hardware-accelerated, which may cause performance to not be as good as expected.

Interference detection is a subset of a more general class of computation which is sometimes called proximity queries. Hoff *et al* [HZLM01] have demonstrated the use of graphics hardware to generate proximity information for two-dimensional objects. They perform image-space computations for collision detection, separation distance, penetration depth, and contact points and normals. Their method is applicable only for individual pairs of objects, and makes use of a distance field computation that was originally used in the context of generating Voronoi diagrams using graphics hardware [HCK+99]. The technique has also recently been expanded to proximity queries in three dimensions [HZLM02].

Graphics hardware has also been used by Kim *et al* for the computation of penetration depth between pairs of 3D polyhedral models in the context of physics-based simulation of solid bodies [KOLM02]. They make use of the depth buffer to aid in the computation of Minkowski sums in order to find the minimum translational vector needed to separate two interfering bodies.

Most recently, Govindaraju *et al* have used graphics hardware to find sets of potentially interfering objects in complex scenes [GRLM03]. As with our algorithm, they use hardware-based occlusion queries (Section 4.5.2) to aid in their interference detection. Their algorithm requires the use of traditional collision detection methods to determine whether or not potentially interfering objects are indeed engaged in interference.

4.1.2 The Depth and Stencil Buffers

Our multi-body collision detection algorithm makes heavy use of two specific features of graphics hardware. These are the depth buffer and the stencil buffer. Both buffers are standard features of the vast majority of commodity-level 3D graphics accelerators.

The *depth buffer* [Cat75] (or *z-buffer*) stores the screen-space depth, or z-value of each pixel of the frame buffer. It is most commonly used for visible-surface determination during rasterization, but may be used for other purposes such as generation of shadow maps [Wil78, SKvW+92]. In recent years, the depth buffer has been used by numerous researchers to aid in other forms of geometric computation. A survey of some of these alternate uses may be found in [TPK01]. For the purposes of our algorithm, the depth buffer is used to store the depth values of all polygon edges. It is subsequently used to test the depth values of polygons against those edge depths.

The *stencil buffer* [OWN+99] (also known as *pixel masks*) is a special buffer, used to associate an extra integer value with each pixel. This value may be altered during a rendering pass, or used in a test to determine whether or not a pixel is drawn. It is typically used for purposes such as

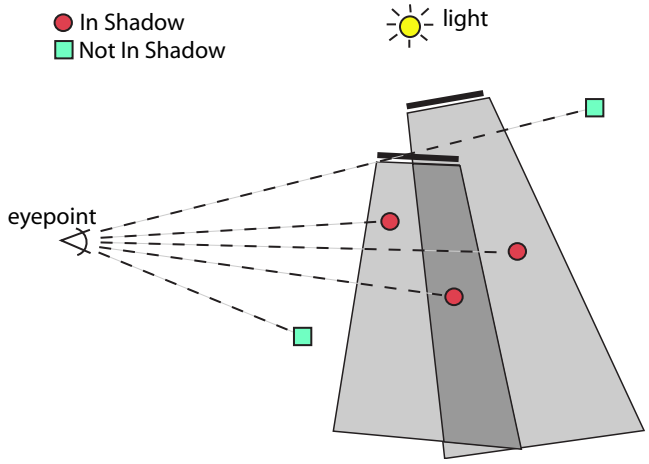


Figure 4.1: Shadow volumes

masking off which areas of the frame buffer may be rendered to. For the purposes of our algorithm, it is used to mark pixels that correspond to rays that intersect polygon edges that intersect other solids.

4.1.3 Relationship to Shadow Algorithms

The initial inspiration for this algorithm came from the one of the most common shadowing techniques in real-time rendering, the *shadow volume* algorithm [Cro77, Ber86]. Using the shadow volume technique, a polygonal mesh is created that represents the volume of space that lies in the shadow cast by an object. Determining whether or not a point lies in shadow involves casting a ray from the viewer toward the point. The point is in shadow if the ray enters more shadow volumes than it exits. This is illustrated in Figure 4.1. The test can be performed in hardware by using the stencil buffer to count the difference in the number of front-facing and back-facing polygons that lie between a point and the viewer [Hei91]. As will be seen shortly, our algorithm employs this same basic ray casting technique to determine if a point lies within the volume enclosed by some solid object.

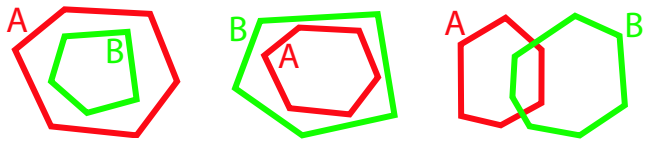
4.2 Interference

Before describing our interference detection algorithm, we will give a brief description of what exactly we mean by interference.

We let $p \in P$ denote that point p is contained within solid polygonal object P. Also let ∂P be the set of edges of P. We denote intersection of two solid objects by $A \cap B$. We define intersection as follows: $\exists a \in A, b \in B | a = b$. When two objects intersect each other, we say that they are in *interference*.

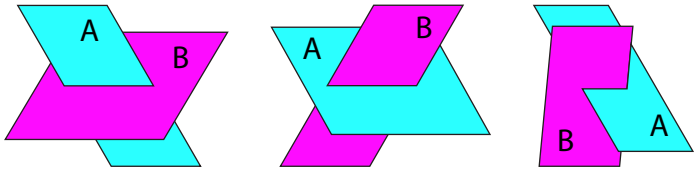
Our interference detection algorithm is predicated on the following property: Two polygonal objects are interfering with each other if and only if an edge of one object intersects the volume occupied by the other. This is expressed by the following theorem [Can87]:

Theorem 4.1 $A \cap B \neq \emptyset$ *iff* $\{\exists a \in \partial A, b \in B | a = b\}$ *or* $\{\exists a \in A, b \in \partial B | a = b\}$



(i) $B \subset A$ (ii) $A \subset B$ (iii) $A \cap B \neq \emptyset$

Figure 4.2: Objects in interference



(a) (b) (c)

Figure 4.3: Ways in which two faces may intersect

Proof If $p \in \partial P$, then $p \in P$. Therefore, if $a \in \partial A$ and $a \in B$, then $a \in A$ and $a \in B$. Similarly, if $b \in \partial B$, and $b \in A$, then $b \in B$ and $b \in A$.

Conversely, if $A \cap B \neq \emptyset$, then either (i) A wholly contains B (i.e., $B \subset A$), or (ii) B wholly contains A (i.e., $A \subset B$), or (iii) the boundaries of A and B intersect (Figure 4.2). Case (i) implies $\forall b \in \partial B, \exists a \in A | b = a$, while case (ii) implies $\forall a \in \partial A, \exists b \in B | a = b$. Case (iii) implies that some face of A intersects some face of B (Figure 4.3). Either the intersection is bounded by an edge of A, in which case this edge intersects the face of B (i.e., $\exists a \in \partial A, b \in B | a = b$), or the intersection is bounded by an edge of B, in which case this edge intersects the face of A (i.e., $\exists b \in \partial B, a \in A | a = b$).

We also note that this property is invariant under affine and projective transformations [TT02], meaning that an intersection test based on this property may be applied at any point in the graphics pipeline.

Detecting interference by querying for intersection between edges and polyhedral volumes has been used in the past. However, most such algorithms are very slow, since they typically require exhaustive tests for edge-face intersections between objects [Boy79, Can86].

4.3 Ray Casting in Graphics Hardware

4.3.1 Casting Rays through the Viewport

When we are rendering a scene, we specify a rectangular region of the frame buffer that pixels may be rendered into. This region is termed the viewport.

We think of a pixel in the frame buffer as the point at which a ray cast from the hypothetical viewer's eye intersects the viewport. The various buffers are used to store relevant information about the ray. For instance, the depth buffer might hold the screen-space depth value of the first object (or, in our case, edge of an object) that the ray intersects. The stencil buffer can hold a value

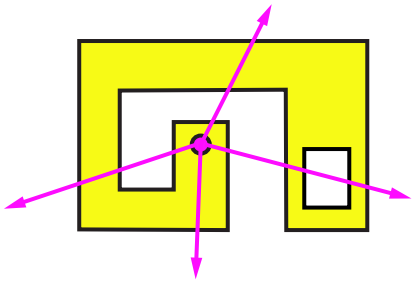


Figure 4.4: Casting rays from a point in a polygon

(typically 8 bits), which is ancillary information about the ray, such as whether or not it intersects an object that is in interference.

The depth value of a pixel is calculated as an interpolation of the depth of the polygon that covers the pixel [FvDFH90]. Since our ray casting algorithm largely relies on depth values for determining interference, we consider the buffer values of the pixel to be a reasonable approximation to a ray cast through the "centre" of the pixel. In effect, when rays are cast at objects, we are sampling the depth values of those objects at discrete intervals.

In general, therefore, we regard the viewport as a framework for point sampling a graphical scene on a regular grid [Smi95].

4.3.2 Counting Boundary Crossings

A central feature of our algorithm is the concept of locating a point relative to a solid by casting a semi-infinite ray from the point. The number of polygons of the solid's boundary that the ray passes through are counted. The two-dimensional version of this technique was first introduced in 1962 by Shimrat [Shi62] and corrected by Hacker [Hac62]. It is commonly used to solve the point-in-polygon problem, and was first used in the context of interference detection between solid models by Boyse [Boy79].

It is a theorem of computational geometry that a semi-infinite ray originating within a closed solid will intersect the boundary of the solid an odd number of times [O'R98]. This is a three-dimensional version of the *Jordan Curve Theorem*, first formulated in two dimensions by Camille Jordan in 1893 [Jor93], which states that any simple closed curve divides the plane into two regions.

In addition, for a directed ray, we can specify whether or not an intersection of the ray with a solid corresponds to the ray *entering* or *leaving* the solid's volume. We can make use of the fact that "enters" and "leaves" alternate and that there may not be two consecutive instances of an "enter" or a "leave". This means that the difference between the number of "enters" and "leaves" is at most one.

We therefore restate the theorem as follows:

Theorem 4.2 *Let all intersections of a ray with the boundary of a solid be classified as either "entering" or "leaving" the volume enclosed by the solid. A semi-infinite ray cast from the interior of a solid will "leave" the solid one more time than it "enters" the solid (Figure 4.4).*

This has an interesting implication for points that are possibly interior to more than one solid. Suppose that we count the difference between the number of times a semi-infinite ray leaves or enters

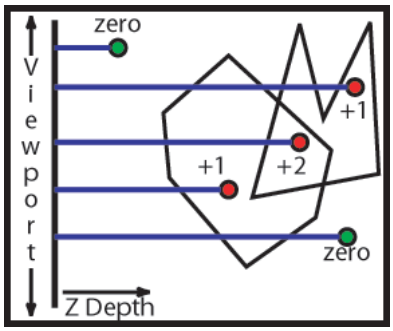


Figure 4.5: Counting boundary crossings

all solids. The origin of the ray lies within some solid if the count is positive. Furthermore, the magnitude of the count indicates how many solids the ray's origin lies within. Figure 4.5 illustrates several rays and the counts that they produce.

We recognize that there are several degenerate cases when rays are cast through solids in this manner. These are described in the context of our algorithm in Section 4.5.7.

4.3.3 Geometry Format and Requirements

Since the algorithm uses rasterizing graphics hardware, object geometry must necessarily be specified as a collection of polygons. For maximum polygon throughput, optimized geometric data structures such triangle strips [ESV96] are naturally preferable. Our algorithm is not dependent on such structures, however. In addition, we render the geometry in two different formats: wireframe and filled polygons. Therefore, an optimized version of the wireframe geometry, with duplicate edges removed, is also preferable.

Because the algorithm detects interference between solid objects, the polygonal meshes that represent them must be closed. That is, there should be no cracks or holes in the meshes that rays can slip through[1].

We also require that the mesh representing an object be simple, in the sense that it is not self-intersecting. The reason for this is that the Jordan Curve Theorem (Section 4.3.2) does not hold for self-intersecting objects.

An object is therefore required to have a well-defined inside and outside - in other words, it should be a closed manifold.

Offset Edges

It is also necessary for surface normals to be supplied with polygons, hopefully on a per-vertex basis. When we draw the wireframe version of the geometry, it is offset slightly outward from the polygonal version. The reason for this is that the graphics hardware does not rasterize lines in exactly the same way that it rasterizes polygons. In particular, an edge of a polygon, when rasterized as a line, is not guaranteed to have exactly the same depth values as the "edge" of the rasterized polygon. Therefore, it is possible for the rasterized edges of a solid to lie slightly inside or outside its polygonal boundary. If an edge lies inside the boundary, then the algorithm will

[1]The Newell teapot [Cro87], for example, does not meet this criterion.

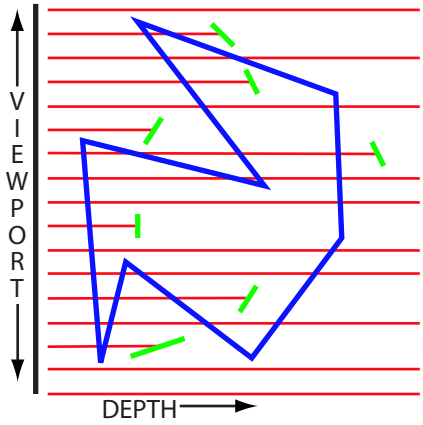


Figure 4.6: Rays cast at the edges of an object of interest. The edge points are disjoint. The polygons of another object enclose some of the edge points.

(incorrectly) determine that the edge is in interference with its own solid. We therefore offset all edges by a small amount in the direction of their normals, ensuring that such a situation cannot occur.

If the graphics hardware has a programmable vertex pipeline (Section 2.3.1), then we do not need to precompute vertex offsets or waste memory storing an offset version of the geometry. Instead, a vertex program is used to automatically offset edges at the time that they are rendered. This also allows us to generate view-dependent offsets, if necessary.

4.4 The Algorithm

Interference detection is performed by point-sampling the scene and looking for object edges that are interior to other solids. This is done using hardware ray-casting. Rays are cast through the pixels of the viewport at objects of interest. Rays that strike those objects' edges are of particular interest. Figure 4.6 shows this in two dimensions. Note that the edge points appear disjoint. In a one-dimensional slice of the viewport, the edge points will only be connected if the edge lines up exactly with the slice.

When a ray strikes an edge, then we count the difference in the number of back-facing and front-facing polygons lying between the edge point and the ray's origin at the viewport. If the difference is not equal to zero, then we know that the edge point lies within the volume of space occupied by another object.

More formally, suppose that a ray with origin point $\mathbf{p}_v$ is cast toward an edge point $\mathbf{p}_e$. Let $\mathbf{n}_0, \ldots, \mathbf{n}_n$ be the normals of the polygons that lie between $\mathbf{p}_v$ and $\mathbf{p}_e$ (See Figure 4.7). Now define $\text{sgn}(x)$ as:

$$\text{sgn}(x) = \begin{cases} 1 & x > 0 \\ -1 & x < 0 \\ 0 & x = 0 \end{cases} \tag{4.1}$$

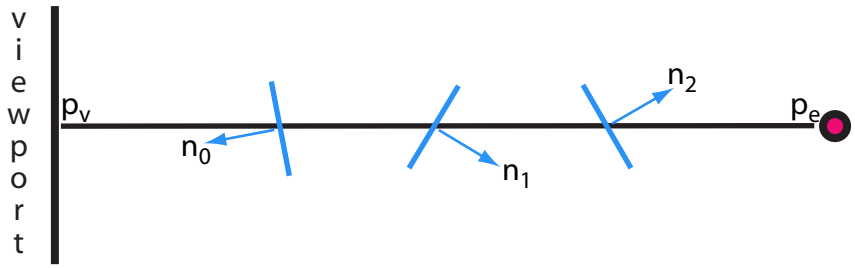


Figure 4.7: Polygons lying between the ray source and an edge point

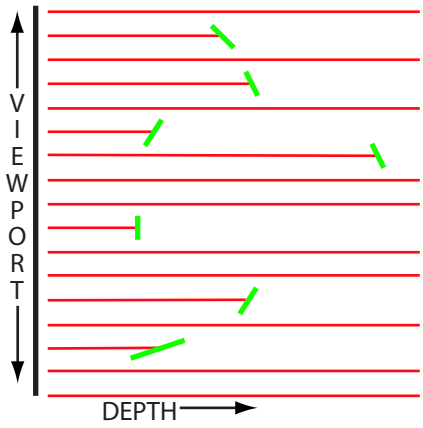


Figure 4.8: Initialize the depth buffer

In effect, the following equation is evaluated:

$$f(\mathbf{p}_e, \mathbf{p}_v) = \sum_{i=1}^{n} \text{sgn}(\mathbf{n}_i \cdot (\mathbf{p}_e - \mathbf{p}_v)) \tag{4.2}$$

Now if $f(\mathbf{p}_e, \mathbf{p}_v) \neq 0$, then the edge point $\mathbf{p}_e$ is interior to some other object. Furthermore, the number of objects that $\mathbf{p}_e$ is interior to is equal to $|f(\mathbf{p}_e, \mathbf{p}_v)|$.

4.4.1 The Rendering Passes

The primary interference detection algorithm consists of the equivalent of three rendering passes. This is shown in pseudocode in Algorithm 1.

We also illustrate the process with a sequence of explanatory images showing a two-dimensional version of two objects (from Figure 4.6) in interference. These figures show how we test whether the edge points of one object lie within the volume enclosed by the polygons of another object.

In the first rendering pass (Lines 4-10), we render all of the edges that we wish to check for interference, and initialize the depth buffer with their depth values (Figure 4.8). This ensures that all rays cast through pixels will be targeted at polygon edges. The first pass is the only one in which the depth buffer is altered. All subsequent rendering passes perform depth tests relative to these depth values. What this means is that all rays cast through pixels will either intersect an edge or

Algorithm 1 Detect Interference

```
1: for all pixels do {clear depth and stencil buffers}
2:    Z = 0, stencil = 0
3: end for
4: depth test = none
5: Enable depth update
6: stencil function = none
7: Disable colour update
8: for all objects do {draw the edge depths}
9:    Draw edges blue{Pass #1}
10: end for
11: Disable depth update
12: depth test = '<'
13: for all objects do
14:    cull mode = back-face
15:    stencil function = increment
16:    Draw polygons blue{Pass #2: add front-facing polygons}
17:    cull mode = front-face
18:    stencil function = decrement
19:    Draw polygons blue{Pass #3: subtract back-facing polygons}
20: end for
21: for all pixels do {check for interference}
22:    if stencil > 0 then
23:       RETURN( interference=true )
24:    end if
25: end for
26: RETURN( interference=false )
```

go to infinity[2].

In the second and third rendering passes, we do not alter the depth buffer or the colour buffer. We do use depth testing, and reject all pixels that fail the depth test. This allows us to count only those pixels of polygons that lie in front of edges. During these two rendering passes, we count the number of polygons that lie between the edges and the origin of the rays.

In the second rendering pass (Lines 14-16), we draw only those polygons whose normals face toward the ray's origin (Figure 4.9). That is, we reject all polygons for which the dot product of the normal with the ray direction is positive. In the graphics hardware this corresponds to a back-face cull. We increment the stencil buffer for each pixel that passes the depth test.

In the third rendering pass (Lines 17-19), we draw only those polygons whose normals face away from the ray's origin (Figure 4.10). That is, we reject all polygons for which the dot product of the normal with the ray direction is negative. In the graphics hardware this corresponds to a front-face cull. We decrement the stencil buffer for each pixel that passes the depth test.

Passes two and three have the effect of using the stencil buffer to count the difference between the number of front-facing and back-facing polygons that lie between the ray's origin and the edge point in question.

After the third pass, the stencil buffer value at each pixel gives the results of the collision detection:

[2]Actually the far clipping plane, which is infinity for our purposes.

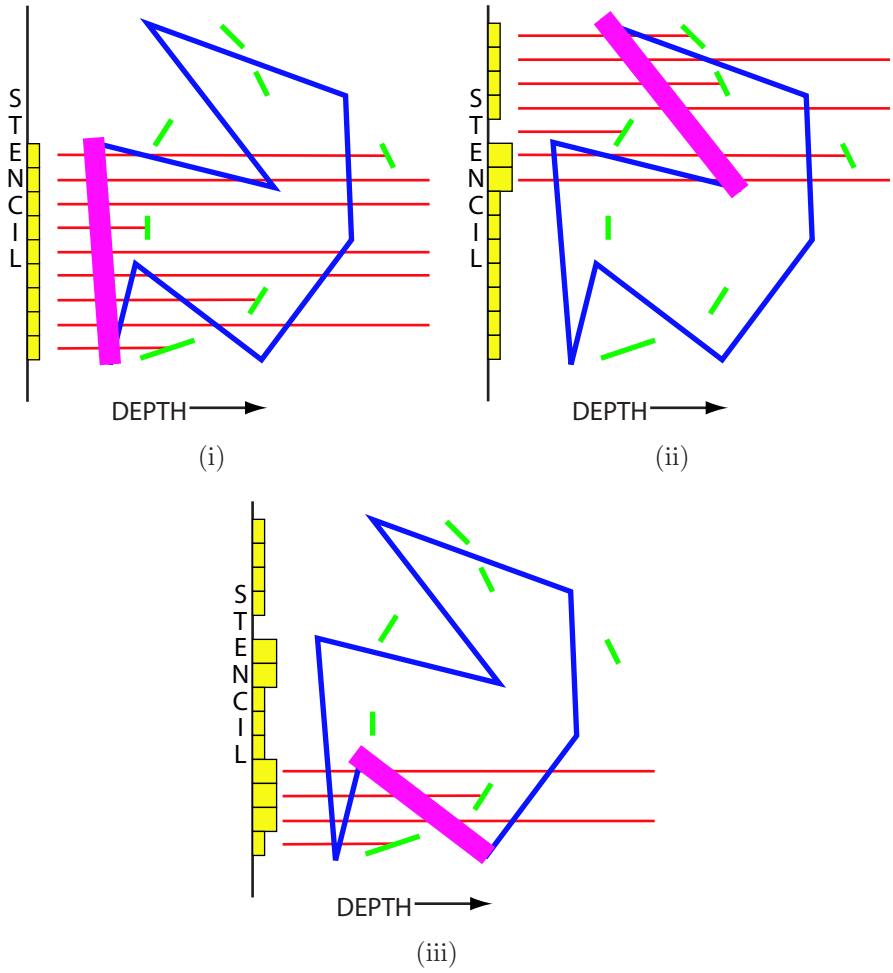


Figure 4.9: Render all front-facing polygons of possibly enclosing objects. The polygon being rendered is highlighted. The value of the stencil buffer after rendering each polygon is shown at left.

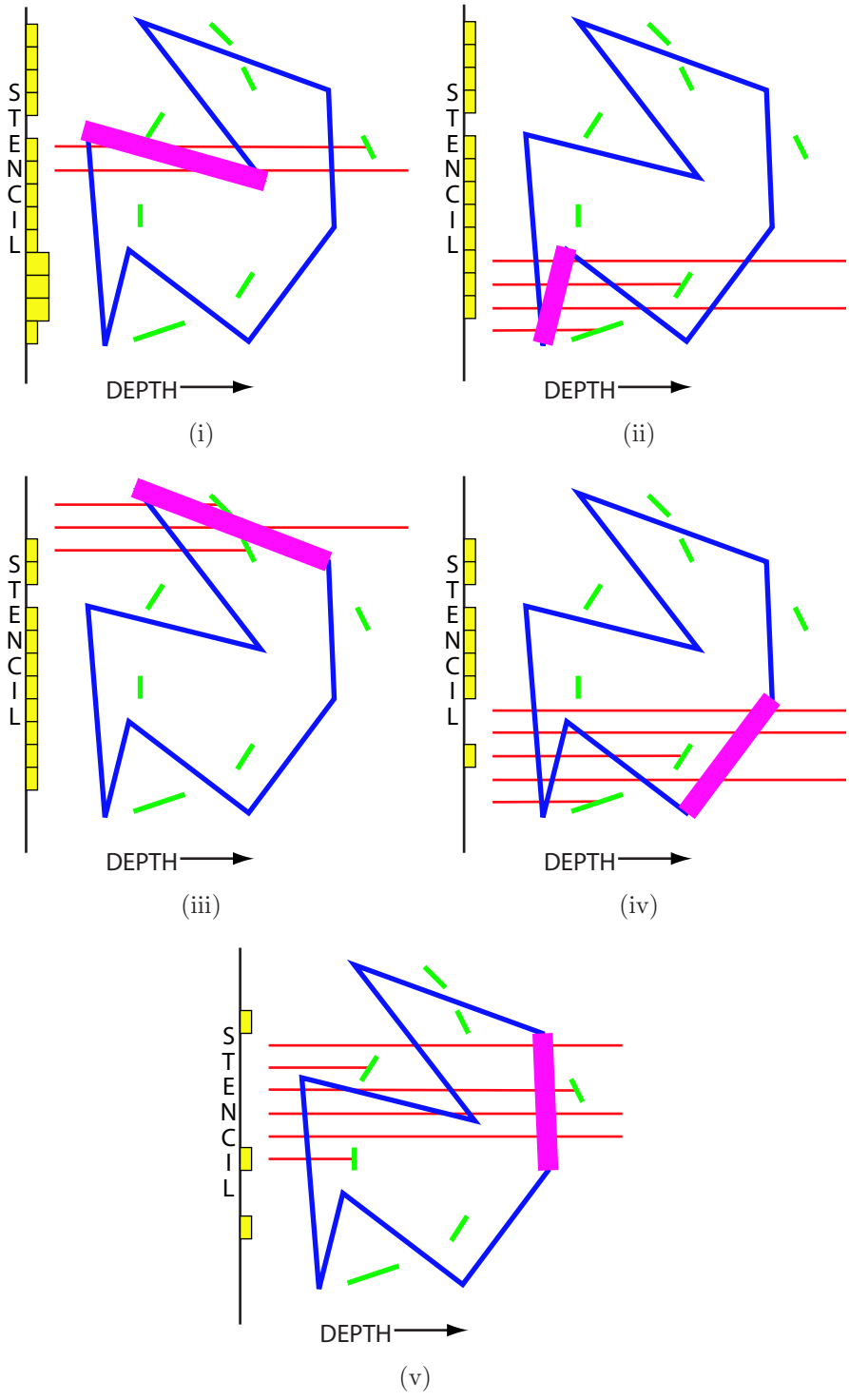


Figure 4.10: Render all back-facing polygons of possibly enclosing objects. The polygon being rendered is highlighted. The value of the stencil buffer after rendering each polygon is shown at left.

- A positive stencil buffer value at a pixel represents a ray cast toward an interfering edge point. The magnitude of the value indicates how many objects the edge point is interfering with.
- A stencil buffer value of zero indicates that the ray represented by the pixel either (a) did not intersect an edge point, or (b) intersected an edge point that was not interfering with another object. To distinguish between these two cases, a depth value of less than infinity indicates that an edge was intersected.
- A negative stencil buffer value means that the ray intersected more back-facing than front facing polygons. This indicates that at least one object is not a closed manifold.

To check for interference, the values in the stencil buffer must be scanned. This requires the stencil values to be passed from the graphics hardware to the CPU.

This version of the algorithm does not identify which objects are interfering with each other. It only informs us whether or not an edge point of any object has intersected the volume of another object. Note that at no point do we modify the colour buffer. The colour buffer is used in an extended version of the algorithm to identify exactly which objects are involved in the interference. This process is described in Section 4.5.1.

Also note that passes two and three could be collapsed into a single rendering pass if the system is capable of implementing different stencil buffer operations based on whether polygons are front-facing or back-facing. This functionality has recently been introduced independently by both NVIDIA and ATI corporations. They have exposed two-sided stencil operations in OpenGL through, respectively, the `EXT_stencil_two_side` and `ATI_separate_stencil` extensions.

4.5 Key Issues

4.5.1 Object Identification

The interference detection algorithm can efficiently discover whether or not any objects are interfering with each other. However, it is more problematic to *identify* the objects that are participating in the interference. Identifying one of the objects is easy, but identifying both is considerably more difficult.

Identifying One Object

To identify only one of the interfering objects, it suffices to assign each object an unique colour. When the edges of objects are drawn in the first rendering pass, they are drawn with the object's colour. Then when the stencil buffer is parsed, if a pixel is found with a non-zero stencil value, that pixel's colour will uniquely identify one of the interfering objects. Furthermore, we know that the object is the one whose edges are partially enclosed by the other object. Algorithm 2 illustrates this.

Note that the identification of the object requires pixels to be retrieved from both the stencil and colour buffers. If reading the stencil buffer is impossible or too expensive, we can use a modified version of the algorithm that requires reading only from the colour buffer. It does so at the expense of a fourth rendering pass at the end of the algorithm. This is shown in Algorithm 3. The extra

Algorithm 2 Identify one interfering object

```
1: for all pixels do {clear colour, depth and stencil buffers}
2:    Z=0, stencil=0, colour=(0, 0, 0, 0)
3: end for
4: depth test = none
5: Enable depth update
6: stencil function = none
7: Enable colour update
8: for all objects do {draw edges with colour enabled}
9:    Draw edges blue{Pass #1}
10: end for
11: Disable colour update
12: Disable depth update
13: depth test = '<'
14: for all objects do
15:    cull mode = back-face
16:    stencil function = increment
17:    Draw polygons blue{Pass #2: add front-facing polygons}
18:    cull mode = front-face
19:    stencil function = decrement
20:    Draw polygons blue{Pass #3: subtract back-facing polygons}
21: end for
22: for all pixels do {check for interference}
23:    if stencil > 0 then
24:       add colour to list of interfering objects
25:    end if
26: end for
```

rendering pass redraws the edges of the objects and updates the colour buffer at only those locations where the stencil buffer is positive. Note also that the colour buffer is no longer updated in the first rendering pass.

Identifying Both Objects

In our algorithm, an interference point corresponds to an object edge that penetrates the volume contained by another object. For each interfering edge point we can therefore make a distinction between the *penetrating* object and the *penetrated* object. The identification algorithm described in the previous section will successfully identify the penetrating object. The difficulty lies in attempting to discover the identity of the penetrated object.

We have designed a modification to the algorithm that will find the identity of the penetrated object, at the cost of the equivalent of another three rendering passes. This is shown in Algorithm 4.

Before identifying the penetrated object, we clear the stencil buffer in order to remove the effects of identifying the penetrating object. This can actually be accomplished in the same rendering pass that we use to write the penetrating object's identity.

The basic idea is to repeat the counting process for each individual object. If, after repeating the counting for a single object, the stencil buffer was incremented (i.e. stencil=1), then we know that object has an edge penetrating it. We write the object's identity into the frame buffer by redrawing the object's polygons and updating the buffer only where the stencil equals 1. In the same pass we also reset the stencil value to 0 whenever it equals 1, in order to restore the stencil

Algorithm 3 Identify one interfering object without stencil read

```
1: for all pixels do {clear colour, depth and stencil buffers}
2:    Z=0, stencil=0, colour=(0, 0, 0, 0)
3: end for
4: depth test = none
5: Enable depth update
6: stencil function = none
7: Disable colour update
8: for all objects do
9:    Draw edges blue{Pass #1}
10: end for
11: Disable depth update
12: depth test = '<'
13: for all objects do
14:    cull mode = back-face
15:    stencil function = increment
16:    Draw polygons blue{Pass #2: add front-facing polygons}
17:    cull mode = front-face
18:    stencil function = decrement
19:    Draw polygons blue{Pass #3: subtract back-facing polygons}
20: end for
21: depth test = '='
22: stencil function = none
23: Enable colour update
24: stencil test = '> 0'
25: for all objects do
26:    Draw edges blue{Pass #4: identify objects}
27: end for
28: for all pixels do {check for interference using colour buffer}
29:    if colour > 0 then
30:       add colour to list of interfering objects
31:    end if
32: end for
```

buffer's original state.

This algorithm requires that the two objects draw their identifiers into separate channels of the colour buffer. For instance, we might reserve the red and green channels for the first object, and the blue and alpha channels for the second object. With a typical frame buffer, this affords us between 12 and 16 bits per identifier, allowing us to distinguish up to 65,000 objects. For most applications, this is enough identifiers to uniquely identify every polygon and every edge.

As with the single object identification algorithm, if the stencil buffer cannot be accessed directly, the colour buffer alone will suffice, at the cost of an extra rendering pass.

Note that the modified algorithm relies on the assumption that an edge is interfering with at most one object. We reserve only one set of colour channels for storing the identity of the penetrated object. If a single edge point penetrates multiple objects, then only one of then is reported[3].

[3]The penetrated object to be reported will be the last one in the rendering order.

Algorithm 4 Identify both interfering objects

```
1: for all pixels do {clear colour, depth and stencil buffers}
2:    Z=0, stencil=0, colour=(0, 0, 0, 0)
3: end for
4: depth test = none
5: Enable depth update
6: stencil function = none
7: Enable colour update
8: for all objects do
9:    Draw edges blue{Pass #1}
10: end for
11: Disable colour update
12: Disable depth update
13: depth test = '<'
14: for all objects do
15:    cull mode = back-face
16:    stencil function = increment
17:    Draw polygons blue{Pass #2: add front-facing polygons}
18:    cull mode = front-face
19:    stencil function = decrement
20:    Draw polygons blue{Pass #3: subtract back-facing polygons}
21: end for
22: depth test = '='
23: Enable colour update
24: stencil test = '> 0'
25: stencil function = 'replace with 0'
26: for all objects do
27:    Draw edges blue{Pass #4: identify objects & reset stencil}
28: end for
29: stencil function = none
30: depth test = '<'
31: for all objects do
32:    stencil test = none
33:    Disable colour update
34:    cull mode = front-face
35:    stencil function = decrement
36:    Draw polygons blue{Pass #5: add front-facing polygons}
37:    cull mode = back-face
38:    stencil function = increment
39:    Draw polygons blue{Pass #6: subtract back-facing polygons}
40:    Enable colour update
41:    stencil test = '≥ 1'
42:    stencil function = 'replace with 0'
43:    Draw polygons blue{Pass #7 - identify object & reset stencil}
44: end for
45: for all pixels do {check for interference}
46:    if colour <> 0 then
47:       add colour channel 1 to list of interfering objects
48:       add colour channel 2 to list of interfering objects
49:    end if
50: end for
```

Optimizing Retrieval of Interference Data

An important consideration here is the speed of access to the colour and stencil buffers. The colour buffer typically has 32 bits of precision per pixel, while the stencil buffer has 8 bits per pixel. This means that reading the full colour buffer typically takes longer than reading the stencil buffer. Therefore, if colliding objects do not need to be identified, then the data should be retrieved from from the stencil buffer.

Similarly, if objects need to be identified, it may not be necessary to allocate all of the colour channels to store object identifiers. A 32-bit frame buffer allows 16 bits of precision for each of the two identifiers. If there are less than 256 objects, then only 8 bits of precision are needed for object identifiers and therefore only two colour channels are required. By retrieving only the necessary channels from the frame buffer, it may be possible to improve access to pixel data on some architectures.

4.5.2 Avoiding Frame Buffer Reads

We have found that, in practise, one of the most time consuming parts of the algorithm is the act of retrieving the frame buffer to main memory and parsing through it one pixel at a time. It would therefore be useful if there were some method of avoiding some or all of this part of the process. As it turns out, for some applications, there are several methods for doing this.

Testing for Pixel Writes using Occlusion Queries

In our algorithm, interference is indicated if the stencil value for any pixel is greater than zero after the third rendering pass. The fourth rendering pass re-renders the wireframe version of the geometry and updates the colour buffer entry of any pixel for which the depth value is equal to the existing value and the stencil value is greater than zero. Therefore, testing for interference amounts to checking whether or not any pixel passes both the depth test and the stencil test during the fourth rendering pass.

Fortuitously, this test is supported in commodity-level hardware via hardware-based occlusion queries. Hardware occlusion queries were first introduced on commodity hardware by Hewlett-Packard in their *Visualize fx* graphics hardware [SOG98] and are also available in the latest graphics accelerators from the NVIDIA and ATI corporations[4]. This functionality is exposed in OpenGL through the *HP_occlusion_test* and *NV_occlusion_query* extensions. These extensions require almost no extra CPU or GPU overhead and do not require an extra rendering pass.

The NV_occlusion_query extension is slightly more robust, as it returns a count of how many pixels pass the test and, more importantly, several queries may be simultaneously extant. This means that in the fourth rendering pass, we can issue a separate occlusion query for every object. If the occlusion query for an object returns a positive value, then we know that the object is involved in some interference. When we want to identify both objects, the same process can be repeated in the final rendering pass. This will successfully identify all interfering objects. Pseudocode is shown in Algorithm 5.

[4]The technology was first implemented on the *Denali GB* graphics hardware of the *Kubota Pacific Titan 3000* workstation [GKM93].

Algorithm 5 Identify objects using occlusion queries

```
1: for all pixels do {clear depth and stencil buffers}
2:    Z=0, stencil=0
3: end for
4: depth test = none
5: Enable depth update
6: stencil function = none
7: Disable colour update
8: for all objects do
9:    Draw edges blue{Pass #1}
10: end for
11: Disable depth update
12: depth test = '<'
13: for all objects do
14:    cull mode = back-face
15:    stencil function = increment
16:    Draw polygons blue{Pass #2: add front-facing polygons}
17:    cull mode = front-face
18:    stencil function = decrement
19:    Draw polygons blue{Pass #3: subtract back-facing polygons}
20: end for
21: depth test = '='
22: stencil function = none
23: stencil test = '> 0'
24: for all objects do
25:    Begin occlusion query for object
26:    Draw edges blue{Pass #4: identify objects}
27:    End occlusion query for object
28: end for
29: depth test = '<'
30: for all objects do
31:    cull mode = front-face
32:    stencil function = decrement
33:    Draw polygons blue{Pass #5: add front-facing polygons}
34:    cull mode = back-face
35:    stencil function = increment
36:    Draw polygons blue{Pass #6: subtract back-facing polygons}
37:    stencil test = '≥ 1'
38:    stencil function = 'replace with 1'
39:    Begin occlusion query for object
40:    Draw polygons blue{Pass #7 - identify objects & reset stencil}
41:    End occlusion query for object
42: end for
43: for all objects do {check for interference}
44:    if occlusion count > 0 then
45:       add object to list of interfering objects
46:    end if
47: end for
```

What is more, if objects are identified using occlusion queries, then the colour buffer does not need to be touched, since it is sufficient only to know that some pixel passed both the depth and stencil tests. However, it is important to note that when multiple pairs of objects are interfering, we cannot determine which exact objects make up those pairs. We must still read the colour buffer to do this.

Although occlusion queries require no extra work on the part of the CPU or GPU, they still require the geometry being tested to finish rasterizing. Since the CPU passes information to the GPU faster than it can be processed, this means that the CPU may have to wait for a while before occlusion information becomes available. However, no stall is engendered on the GPU, and the CPU can continue performing other tasks until the occlusion query data is available. It is also worth noting that in our application we have to wait for the geometry to finish rasterizing anyway, since we cannot read the frame buffer until rendering is completed.

Querying for Non-Zero Colour

If there is no interference, then the colour buffer is never touched. Therefore, early non-interference detection could be implemented by querying the graphics hardware for the maximum colour value of any pixel processed during the final rendering pass. If the maximum colour is zero, then no interference is indicated, and there is no need to parse the pixels of the frame buffer.

Some graphics hardware supports a minimum/maximum pixel query during pixel transfer operations[5]. In fact, this functionality is exposed in the standard OpenGL API through `glHistogram` and `glMinmax`. The fastest form of pixel transfer is a card-local copy of frame buffer memory. Such a copy does not require data to be transfered from the graphics acclerator to the computer's main memory, and therefore engenders a briefer stall on the graphics pipeline.

The graphics accelerator[6] on which we tested the algorithm unfortunately does not support hardware-accelerated min/max operations, and early non-interference detection of this type was therefore highly inefficient[7]. However, in the presence of hardware that supported this functionality, efficient early non-interference detection would be very simple.

4.5.3 Image Space vs. Object Space

The ray-casting algorithm is an image space algorithm. As such, there are a number of issues specific to image space techniques that must be dealt with.

When interference detection is being conducted, we must ensure that every object of interest is visible when projected onto the viewport. Put another way, all points where collision may occur must be visible by at least one of the rays.

The effectiveness of the algorithm is greatly dependent on the relative distance between objects. If objects are on average separated by distances much greater than their average size, then interference detection will not be very precise. The relationship between object separation and precision of interference detection is largely a function of viewport resolution and depth buffer resolution.

[5] But not during rasterization, unfortunately.

[6] A graphics card using the NVIDIA GeForce 4 chip set.

[7] So inefficient, in fact, that interference detection as a whole slowed down by over two orders of magnitude!

Viewport Resolution

The resolution of the viewport through which rays are being cast is of paramount importance. It directly affects the precision of the interference detection.

Suppose that we are using an orthographic projection. Let x_v, y_v be the image-space dimensions of the viewport, and x_w, y_w, z_w be the world-space dimensions of the view frustum. World-space precision of interference detection in the plane parallel to the viewport is then limited to $\frac{x_w}{x_v}$ by $\frac{y_w}{y_v}$.

As an example, suppose that the viewport is mapped to a spatial volume that is 1 meter to a side. Further suppose that the screen-space resolution of the viewport is a 100 by 100 pixel box. This means that each pixel, and hence each ray, corresponds to a 1 centimetre square box. Interferences can be detected at no better than 1 centimetre precision in any plane parallel to the viewport.

Depth Buffer Resolution

The depth buffer resolution has a similar effect on the precision of interference detection in the direction perpendicular to the plane of the viewport. Additionally, the precision of depth values is affected by the type of projective transform imposed on the scene for the interference detection process. In particular, perspective projection results in distant objects receiving screen-space depth values with less precision than close objects [AMH02].

For many situations, this is actually preferable. In primarily visual applications, such as video games or virtual reality, it is better to give more attention to objects that are close to the user's viewpoint [OD01].

We note, however, that using perspective projection increases greatly the likelihood of self-intersection being reported, as described in Section 4.5.8.

On the other hand, we may wish to detect collisions with uniform precision over the entire view volume. In such is the case, then the best solution is to use an orthographic projection, which requires no perspective division.

We also ensure that the view frustum is as tight as possible around the bounding box of all the objects being tested for interference. This allows maximum precision in the screen-space z direction when objects are rendered.

4.5.4 Multiple Objects and Non-Convex Geometry

Our ray casting algorithm, unlike previous efforts in hardware-assisted interference detection, can handle both non-convex geometry and large sets of potentially interfering objects.

The reason for this stems from the previous observation that if two objects are interfering with each other, then an edge of one of them must intersect the volume of the other. The only way for an intersection to miss detection by the algorithm is if no ray can see an interfering edge point. Recall that the depth buffer stores the depth values of the closest edges to the eyepoint. For an edge to be obscured would require every pixel of every interfering edge to be occluded by other edges (see Figure 4.11).

Such a situation might occur for one of three reasons:

- The configuration of the objects is highly degenerate (see Section 4.5.7).

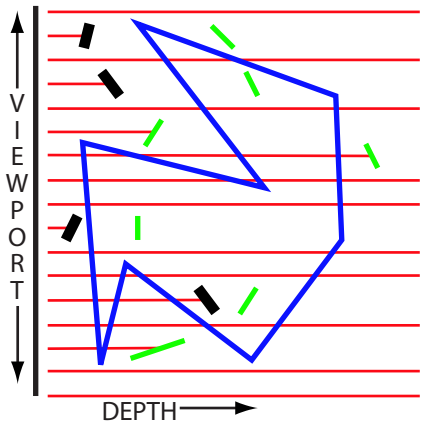


Figure 4.11: An undetectable interference. Interfering edges of one object are blocked by the edges of another object (in purple).

- There is a locally dense cluster of edges in the projection of the scene onto the viewport.
- The viewport resolution not high enough.

The most common of these problems is the second one. Dense clusters of edges indicate that either the objects have very large edge counts, or the projection of the objects is so small that they are taking up a small viewport area. This can be solved by increasing viewport resolution, but in practise doing so is not very practical beyond a small amount. A better solution would be to perform some precomputation that minimizes the world-space area that rays are cast into.

4.5.5 Interference Localization

We have already demonstrated that it is possible to use the graphics hardware to detect object interference. Furthermore, an extension to the algorithm made it possible to detect which pairs of objects are interfering with each other. What remains now is to determine the location of the interference.

For many applications, it does not suffice to know only that two objects are interfering with each other. Rigid body simulation, for instance, requires knowledge of the surface points at which objects are interpenetrating, in order to correctly apply forces or impulses to separate contacting bodies.

There are several problems with attempting to localize collision points using our algorithm. Firstly, the size of the viewport that we are rendering into limits the spatial precision of contact point location. Secondly, there are a potentially large number of points reported for each interference. This is especially true for objects created from dense meshes, which have a large number of edge points that may lie within the volume of another object.

Using a variation of the object identification scheme, it is possible to determine which edges are involved in the interference. Recall that we identified interfering objects by assigning unique colours to them. We can similarly assign colours to edges. During interference detection we can render each edge with an unique colour, which is used to identify the edge when we parse through

the frame buffer[8]. We therefore have a list of edges that intersect the volumes of other objects. It is also the case that, unless one object entirely encloses another, if two objects are interfering, then an edge of one must penetrate the surface of the other. This means that our list of interfering edges must include those edges that penetrate the surfaces of other objects, which makes the search for surface contact points much easier.

We also observe that every pixel in the frame buffer has an associated depth value in the depth buffer. The pixel location and depth combine to give the screen-space location of an interfering point located by the ray cast through the pixel. The screen-space position can be subjected to a reverse transformation to find the world-space (or object-space) location of the associated interference point.

We note, however, that such an operation does not come for free. A typical frame buffer will have 24 bits of depth precision, so reading all of the depth values will take about 3/4 of the time required to read the colour buffer. Of course, the depth values only need to be retrieved for those pixels where interference is detected, but the time required is still non-negligible.

4.5.6 Collision Response

For animation applications, interference between two solids indicates a collision between two moving objects. If a collision is detected, then we are typically interested in also computing the responses exhibited by the colliding bodies.

As presented, our image-space method does not provide enough information to compute collision responses. To do so, we require better interference localization. Specifically, we want to know the object-space positions at which object edges are penetrating the surfaces of other objects. In addition, we require knowledge of the surface normals and object velocities at those points.

Vassilev *et al* [VSC01] make an attempt at using the hardware to aid in computation of collision response in the context of cloth simulation. However, their methods are not completely applicable to our algorithm. Specifically, the concept of producing a normal map and velocity map for the objects is not sufficient for our purposes. This is because the normal and velocity maps will be available only for the edges of objects. This is only half the required information, as the normal and velocity of the surface points of the penetrated object must also be known.

We therefore currently make no attempt to use graphics hardware to aid in response computation. Any such calculations must be performed in software. This is largely sufficient for rigid body simulation, for example, where computation of object-space velocities is a relatively simple matter.

4.5.7 Degenerate Cases

There are several cases where the geometry being tested for interference may lead to degeneracies during the ray-casting process.

Many of these are the result of the geometry format that we use. In particular, polygonal geometry results in degenerate situations when a ray is coincident with an edge or a vertex. Figure 4.12 illustrates these degeneracies in a two-dimensional setting.

Degeneracy #1 illustrates a ray passing through a vertex on the silhouette edge of the polygons. It does not present a problem to our algorithm. The vertex is drawn and counted once for each

[8]For very small numbers of edges, we can assign an occlusion query (Section 4.5.2) to each edge, and thus avoid the frame buffer read.

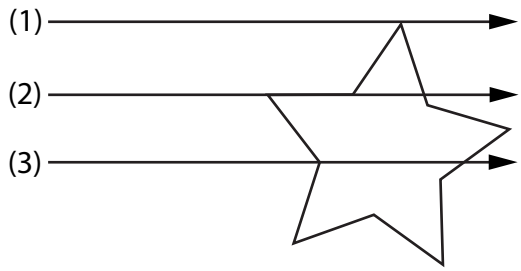


Figure 4.12: Degenerate ray/polygon intersections

polygon, both front-facing and back-facing, that contains it. This means that the stencil buffer count at that pixel is both incremented and decremented.

Degeneracy #2 illustrates a ray which is exactly collinear with an edge. It does not present a serious problem to our algorithm either. This is because the graphics hardware does not render edges (or polygons) that are completely perpendicular to the viewport after projection. Therefore, edges and polygons collinear with rays will not show up in the count.

Degeneracy #3 would seem to be the most likely to cause problems, as both faces touching the vertex would be counted by the ray, when only one should be. However, graphics hardware is designed in such a manner that this situation cannot occur. When two polygons share an edge, the hardware will still scan-convert each pixel only once. This feature was originally designed to avoid problems arising from duplicate renderings of pixels corresponding to polygons with features such as transparency. However, it works equally well in our algorithm by avoiding counting degeneracies arising from duplicate pixels.

Therefore, in general, all three forms of deneneracy cannot occur. However, even if they did, our algorithm is designed in such a way that they are unlikely to arise. The usual method for dealing with degeneracies in geometric computations is to subject the geometry to some form of perturbation [EM90]. We already perform perturbation on the geometry in order to deal with problems that crop up due to limited precision in hardware rasterization (see Sections 4.3.3 and 4.5.3). It is worth noting, however, that this perturbation also helps to minimize the problems arising from degenerate ray-polygon intersections.

4.5.8 Self-Intersection

It is possible for the algorithm to incorrectly report that an object has intersected itself. One of the base assumptions of our algorithm is that we are dealing with simple, closed polygonal meshes. That is, objects that do not intersect themselves. When self-intersection is reported, it is almost always the result of one of two situations occurring.

The first form of self-intersection is the result of the limited precision of both the viewport and the depth buffer. Buffer precision is discussed in detail in Section 4.5.3. We note here that the precision of the viewport is almost always the determining factor. The depth buffer typically has up to 24 bits of resolution, whereas the viewport has the equivalent of between 7 and 10 bits of resolution.

The second form of self-intersection occurs because of rasterization differences between polygons and edges. In particular, an edge may not have the same depth values when rendered in wireframe as when it is rendered as part of a polygon. This problem, and its solution, is described in Section 4.3.3.

Depending on the application context, self-intersection can be a very common problem. Indeed, in some situations it may be unavoidable. For instance, in character animation it is often the case that the skin of one part of the body does not blend smoothly with the skin of an adjacent body part. This is especially true if each body part is modelled separately and the skin mesh as a whole is assembled later.

For cases where self-intersection is inevitable, we have to accept that spurious intersections will be rendered into the frame buffer. Of course, these can be trivially rejected when their colour is inspected. Unfortunately, it also means that early non-interference detection, such as that described in Section 4.5.2 may not always be completely effective.

4.6 Complexity Analysis

It is reasonable to ask what the asymptotic running time of our algorithm is. There are three primary variables which affect the performance of the algorithm.

For collision detection involving multiple objects, the running time is a function of how many objects are involved. The naïve algorithm for N objects involves $\mathcal{O}(N^2)$ pair-wise tests. Our algorithm draws each object a constant number of times and is therefore $\mathcal{O}(N)$ in the number of objects involved. This is, of course, assuming that each object has roughly the same number of polygons. If this is not the case, then performance is better measured as a function of the total number of polygons in the objects being tested.

For collision detection involving two polygonal objects, the running time is usually a function of the number of polygons. If the two objects are constructed of P_i and P_j polygons, respectively, then the naïve algorithm involves $\mathcal{O}(P_iP_j)$ polygon-polygon intersection tests. Our algorithm renders each edge or polygon a constant number of times and is therefore $\mathcal{O}(P)$ in the number of polygons, where P is the total number of polygons of all objects being tested. It is important to note that our algorithm remains $\mathcal{O}(P)$ in the number of polygons regardless of how many individual objects are being tested for interference.

The third determining factor is how many pixels must be scanned in order to determine whether or not interference has occurred. Each pixel is scanned only once for each iteration of interference detection, so if there are R pixels, our algorithm is $\mathcal{O}(R)$ in the number of pixels.

4.7 Results

4.7.1 Implementation

Base Implementation

We implemented the interference detection algorithm in the C++ programming language, using OpenGL [OS99] as the real-time rendering API. OpenGL is standardized across multiple computing architectures and stencil buffers are required by the standard, making it ideal for our purposes.

Our timing tests were performed on a computer with dual 1.8GHz Pentium IV CPUs and a graphics accelerator that used the NVIDIA GeForce4 chip set.

Game Console Implementation

We also implemented the interference detection algorithm using a proprietary API called *Pure3D* [Rad]. Pure3D is a rendering abstraction layer that allows rendering to be performed on a variety of computer platforms via a common API.

On the PC platform, Pure3D can use either DirectX 8 or OpenGL 1.3 to perform rendering. On the Microsoft XBox [Mic01, Abr00] platform, Pure3D uses a specialized version of DirectX 8.

The XBox's main CPU is a 733MHz Pentium III, and it utilizes variant of the NVIDIA GeForce3 chip set. For the purposes of testing our algorithm, the XBox differs from a PC in several ways. The most significant of these is the XBox's memory system, which uses a Unified Memory Architecture (UMA) design (Section 2.3.4). For instance, our tests indicate that frame buffer access is significantly faster on the XBox than on a comparable PC.

4.7.2 Examples

We have tested the algorithm with a wide variety of solid objects.

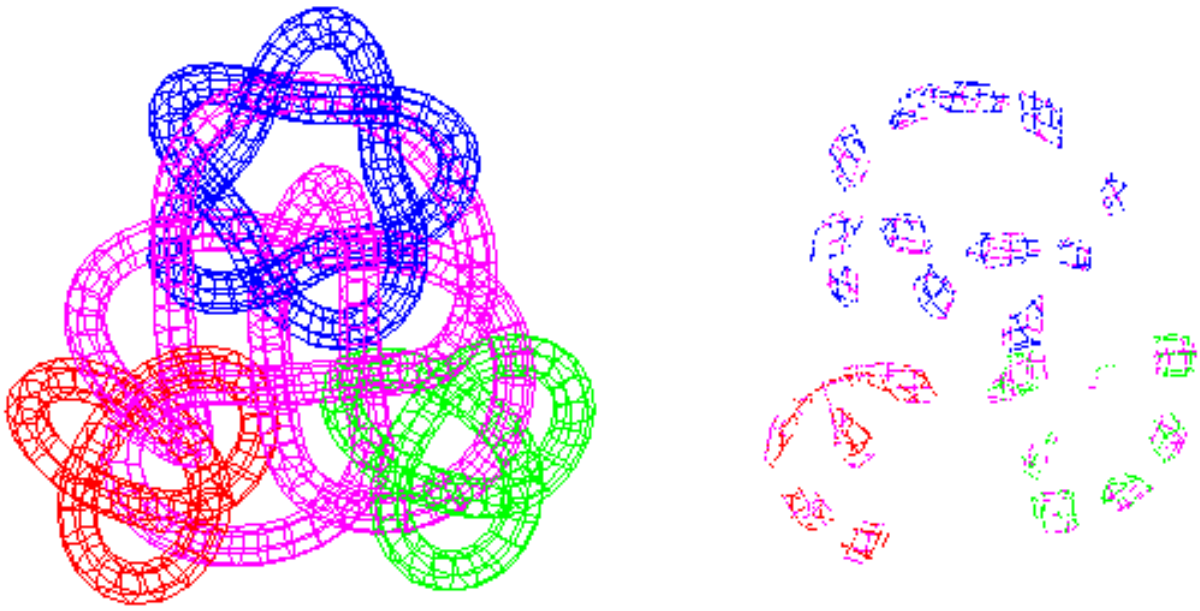
An example is given in Figure 4.13, which shows a scene involving several highly non-convex objects[9] that are entangled and interfering with each other. Figure 4.13(a) shows the edges of the objects being tested. Figure 4.13(b) shows an enhanced version of the interferences reported in the stencil buffer. The colours in the enhanced stencil buffer correspond to the colours of the objects whose edges are in intereference.

Note in particular that the underlying structures of the interfering portions of the meshes are clearly discernible.

Another example is given in Figure 4.14, which shows a large number of objects, many of which are interfering. Objects that are engaged in interference are highlighted in red.

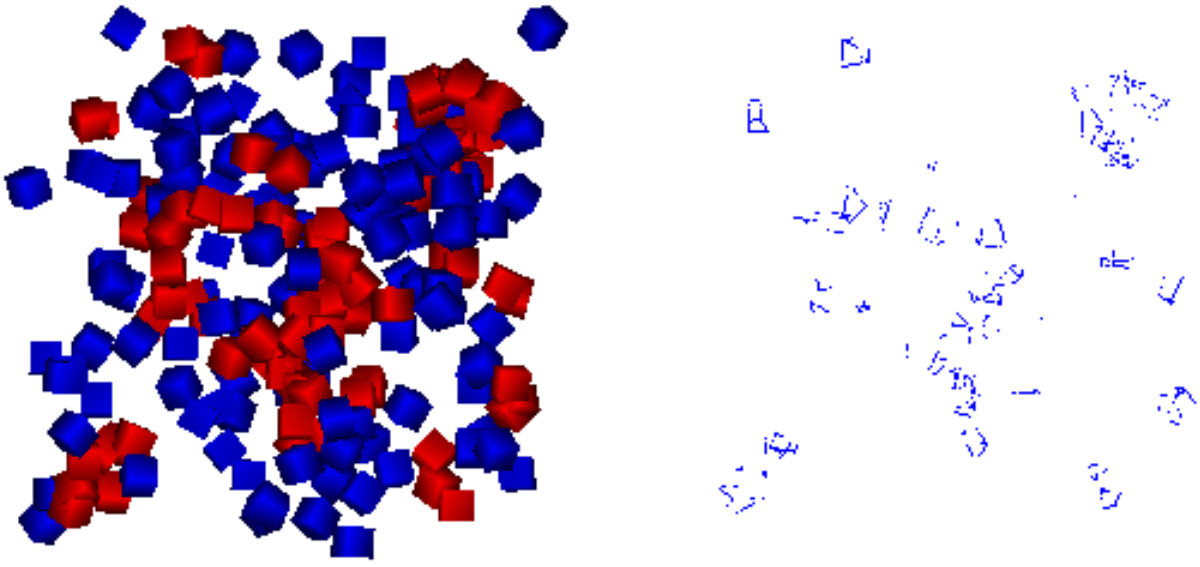
A significant effort was made to test the algorithm with non-convex objects and scenes involving many objects. We also tested the algorithm with objects of very high polygon count. For low polygon count objects such as boxes, we were able to simultaneously detect interference between several hundreds of objects. With small numbers of objects, such as a single pair, we were able to use models with polygon counts of over five thousand before mesh density became too high for the algorithm to function correctly.

[9] Courtesy of [Sch98].



(a) Scene (b) Stencil Buffer

Figure 4.13: Non-convex objects in interference



(a) Scene (b) Stencil Buffer

Figure 4.14: Multiple objects in interference

4.7.3 Timings

When measuring the performance of the algorithm, we start the timing with the first command issued to the graphics card. Timing is concluded when either pixel parsing is finished or the last occlusion query result becomes available. This gives us the total time required by both the CPU and GPU. Note, however, that it is possible for either computational unit to be idle during some of the computation. For example, once the frame buffer is retrived, the GPU is not needed for scanning pixels. Similarly, the CPU may have free cycles while waiting for geometry to be processed.

Timing values were taken as the mean over 100 trials. Vertical bars at data points show the standard deviation. The spatial configuration of the objects was randomized for each trial.

Figure 4.15 shows interference detection time as a function of the number of objects being tested. The objects were all boxes constructed as strips of twelve triangles. The relationship is clearly linear.

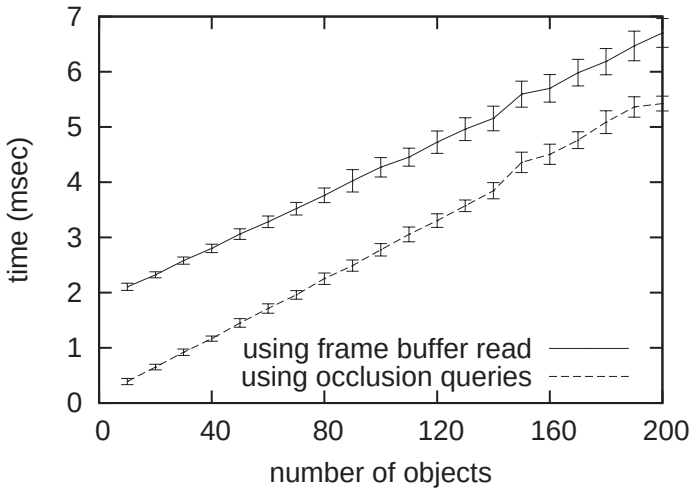


Figure 4.15: Timing data as a function of object count

Figure 4.16 shows interference detection time as a function of the number of polygons in the objects being tested. For this example, we used objects with the same basic shape, but constructed at several levels of detail with different numbers of polygons. The relationship here is also linear.

Note in particular that reading the frame buffer rather than using occlusion queries to identify objects can be fairly costly. In these examples we rendered to a 256 by 256 pixel off-screen rendering surface.

The Cost of Scanning Pixels

We have found that, in practise, for our interference detection algorithm, by far the most expensive operation was the act of reading the pixels from the graphics hardware and parsing through them looking for interferences.

Table 4.1 shows the relative time spent rendering objects and scanning pixels using two different viewport resolutions. Timing data is showed for detecting interference using both the “read pixels” and “occlusion query” methods. With over three thousand polygons, this represents a moderately complex set of objects.

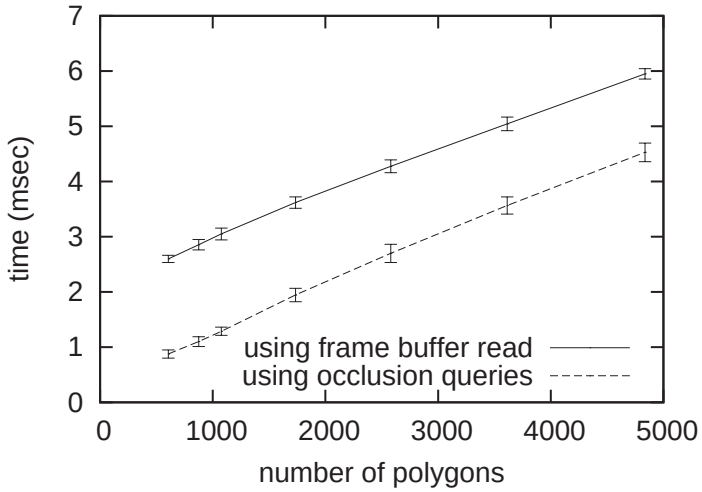


Figure 4.16: Timing data as a function of polygon count

	read pixels		**occlusion query**	
# Polygons	3614	3614	3614	3614
Resolution	640x480	256x256	640x480	256x256
# Pixels	307,200	65,536	307,200	65,536
Pixel Read Time	6.7 msec	1.5 msec	0.0 msec	0.0 msec
Pixel Parse Time	1.2 msec	0.2 msec	0.0 msec	0.0 msec
GPU Render Time	4.0 msec	3.9 msec	4.0 msec	3.9 msec

Table 4.1: Timings for pixel reads and occlusion queries

First note that the time spent in rendering is approximately the same regardless of viewport resolution. However, the time spent in reading and scanning pixels increases rapidly with viewport resolution. As can be seen, if we use occlusion queries to identify objects, then we no longer need to spend time scanning pixels. Any degradation in rendering speed is primarily due to the graphics hardware stalling while waiting for the results of the first set of occlusion queries to return. The occlusion query method allows us to increase the size of the interference detection viewport, without incurring a significant performance penalty. This means that if the hardware has extra frame buffer memory, we can increase the precision of interference detection at almost no cost.

Detecting Non-Interference

Using occlusion queries, we can perform an early test for non-interference, as described in Section 4.5.2. Table 4.2 shows timings for our algorithm using non-interference detection on models when interference is present and when it is not present. When interference is not present, the non-interference technique consistently saves approximately thirty percent of the GPU rendering time.

We also note that early non-interference can be employed even if the frame buffer needs to

# polygons	606	1076	1734	2580
Interference	0.92	1.35	2.04	2.94
No interference	0.63	0.98	1.48	2.11

Table 4.2: Rendering timings using early non-interference detection (in msec)

ultimately be scanned[10]. In this case, the time savings will be even more pronounced, since the expensive pixel read operation can be avoided when non-interference is detected.

[10]To identify the exact pairs of interfering objects, for instance.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

We have presented two collision detection algorithms that make use of graphics hardware to assist in computation.

The first algorithm uses a programmable geometry engine to perform closed-form simulation of particles. The motion paths of the particles are intersected with analytical surfaces in order to determine whether or not an impact with the surface has occurred. Information about collisions is rendered into a two-dimensional reparameterization of the impacted surface and transmitted back to the computer's main processor for further processing.

The second algorithm uses a ray-casting technique to point-sample closed polyhedral objects and locate edge points that are interior to other objects. The algorithm requires no preprocessing or custom data structures and its running time is linear in both the number of objects and number of polygons comprising the objects.

The main drawback to both algorithms is their reliance on transferring data from the frame buffer to the main CPU, which may be a slow process. This can be at least partially overcome by applying other techniques such as the occlusion queries used in the ray-casting interference detection.

5.2 Future Work

We have identified a number of potential directions of research for both collision detection techniques.

5.2.1 Particle System Collision Detection

A wider variety of motions for particles is a natural progression of the current work. In particular, dynamic motions that are not closed-form in nature would be desirable. This could be likely be accomplished by rendering dynamic properties of particles to texture memory. Texture memory is not yet accessible to vertex programs, but is accessible to the fragment programs of current graphics hardware. A more sophisticated vertex or fragment programming model could make hardware-based particle dynamics a reality.

We envisage using a much wider variety of objects as colliders. For instance, implicit surface constructs [Blo97] such as skeletal convolutions, blobby objects, or implicit surface patches have analytical surface properties that make them amenable to fast calculation of trajectory intersections. In addition to new forms of colliders, the algorithm should be extended to handle much larger numbers of collider objects.

A difficulty with the impact map concept is that it is not always clear how an analytical surface may be reparameterized in two dimensions. Indeed, for many such objects, no global reparameterization may exist. Several different researchers have demonstrated results that may aid us in this area. Alonso *et al.* [ACJ+01] demonstrate the *virtual mesh*, a Jacobian mapping of curved patches to the plane. The *geometry images* work of Gu *et al.* [GGH02] shows how arbitrary surfaces can be remeshed onto a completely regular 2D structure. Both of these approaches are far too computationally expensive to be feasible with the current generation of programmable geometry engines. However, our expectation is that in the future complex computations such as these will be possible.

Many forms of impact information may be static in the sense that the data remains constant for any given location on the surface of a collider. Such data may be precomputed and then queried when collision occurs. The most logical location in which to store such data on graphics hardware is in texture memory. The impact map could be used to report such data via per-pixel texturing operations.

5.2.2 Interference Detection through Ray-Casting

The precision of our hardware-assisted ray-casting is currently constrained by the dimensions of the viewport that we are rendering the objects into. Overcoming the limitations imposed by viewport resolution is a natural area for future work.

Our method could be extended to provide better interference localization. For applications such as rigid body simulations, this would entail identifying the point at which an edge of one objects intersect the surface of the other object. The proximity queries described by Hoff *et al* [HZLM01, HZLM02] provide a good starting point for this extension to our work.

We believe there is some merit in combining our algorithm with level-of-detail techniques. This would allow us to perform fast rejection tests on coarse approximations to the polygonal models, and only use the full model when higher accuracy is needed. Similarly, the algorithm could also easily be extended to handle interference detection with bounding volume hierarchies such as those of Gottschalk *et al.* [GLM96]. In such a situation, only the portions of the model in the leaves of the hierarchy would be used for full-precision interference detection.

Bibliography

[Abr00] Michael Abrash. Inside Xbox graphics. *Dr. Dobb's Journal*, pages 21–26, August 2000. [cited on p. 11, 51]

[ACJ+01] L. Alonso, F. Cuny, S. Petit Jean, J.-C. Paul, S. Lazard, and E. Wies. The virtual mesh: a geometric abstraction for efficiently computing radiosity. *ACM Transactions on Graphics*, 20(3):169–201, July 2001. ISSN 0730-0301. [cited on p. 58]

[AMH02] Tomas Akenine-Möller and Eric Haines. *Real-Time Rendering*. A K Peters, Ltd., 2nd edition, 2002. ISBN 1-56881-182-9. [cited on p. 7, 10, 46]

[Ber86] Philippe Bergeron. A general version of Crow's shadow volumes. *IEEE Computer Graphics & Applications*, 6(9):17–28, 1986. [cited on p. 30]

[Blo97] Jules Bloomenthal, editor. *Introduction to Implicit Surfaces*. Morgan Kaufmann Publishers, Inc., 1997. ISBN 1-55860-233-X. [cited on p. 57]

[Boy79] John W. Boyse. Interference detection among solids and surfaces. *Communications of the ACM*, 2(1):3–9, January 1979. [cited on p. 31, 32]

[BW97] George Baciu and Wingo Sai-Keung Wong. Rendering in object interference detection on conventional graphics workstations. In *Pacific Graphics '97*, Seoul, Korea, October 1997. [cited on p. 28]

[BWS98] George Baciu, Wingo Sai-Keung Wong, and Hanqiu Sun. RECODE: An image-based collision detection algorithm. In *Pacific Graphics '98*, Singapore, October 1998. [cited on p. 28]

[Can86] John Canny. Collision detection for moving polyhedra. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(2):200–209, March 1986. [cited on p. 31]

[Can87] John F. Canny. *The Complexity of Robot Motion Planning*. PhD thesis, Massachusetts Institute of Technology, 1987. [cited on p. 30]

[Cat75] Edwin Catmull. Computer display of curved surfaces. In *Proceedings of the IEEE Conference on Computer Graphics, Pattern Recognition and Data Structures*, pages 11–17, May 1975. [cited on p. 29]

[Cha84] Bernard Chazelle. Convex partitions of polyhedra: a lower bound and worst-case optimal algorithm. *SIAM Journal on Computing*, 13(3):488–507, August 1984. [cited on p. 5]

[CLMP95] Jonathan D. Cohen, Ming C. Lin, Dinesh Manocha, and Madhav K. Ponamgi. I-COLLIDE: An interactive and exact collision detection system for large-scale environments. In *1995 Symposium on Interactive 3D Graphics*, pages 189–196. ACM SIGGRAPH, April 1995. ISBN 0-89791-736-7. [cited on p. 6]

[Cro77] Franklin C. Crow. Shadow algorithms for computer graphics. In *Computer Graphics (Proceedings of SIGGRAPH 77)*, volume 11, pages 242–248, San Jose, California, July 1977. [cited on p. 30]

[Cro87] Franklin C. Crow. The origins of the teapot. *IEEE Computer Graphics & Applications*, 7(1):8–19, January 1987. [cited on p. 33]

[DeL01] Mark A. DeLoura, editor. *Game Programming Gems 2.* Charles River Media, Inc, 2001. ISBN 1-58450-054-9. [cited on p. 62]

[DK90] David P. Dobkin and David G. Kirkpatrick. Determining the separation of preprocessed polyhedra: a unified approach. In *Proceedings of the seventeenth international colloquium on Automata, languages and programming*, pages 400–413, Warwick University, England, 1990. Springer-Verlag New York, Inc. (Lecture Notes in Computer Science vol. 443) ISBN 0-387-52826-1. [cited on p. 5]

[EL01] Stephan A. Ehmann and Ming C. Lin. Accurate and fast proximity queries between polyhedra using convex surface decomposition. *Computer Graphics Forum*, 20(3):500–510, 2001. ISSN 1067-7055. [cited on p. 5]

[EM90] Herbert Edelsbrunner and Ernst Peter Mücke. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics*, 9(1):66–104, January 1990. ISSN 0730-0301. [cited on p. 49]

[ESV96] Francine Evans, Steven S. Skiena, and Amitabh Varshney. Optimizing triangle strips for fast rendering. In *IEEE Visualization '96*, pages 319–326. IEEE, October 1996. ISBN 0-89791-864-9. [cited on p. 33]

[FF88] Alain Fournier and Donald Fussell. On the power of the frame buffer. *ACM Transactions on Graphics*, 7(2):103–128, 1988. [cited on p. 10]

[FvDFH90] James D. Foley, Andries. van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practise.* Systems Programming Series. Addison–Wesley Publishing Company, 2nd edition, 1990. ISBN 0-201-12110-7. [cited on p. 32]

[GGH02] Xianfeng Gu, Steven J. Gortler, and Hughes Hoppe. Geometry images. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2002)*, 21(3):355–361, July 2002. [cited on p. 58]

[GKM93] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical z-buffer visibility. In *Proceedings of SIGGRAPH 93*, Computer Graphics Proceedings, Annual Conference Series, pages 231–240, Anaheim, California, 1993. ISBN 0-201-58889-7. [cited on p. 43]

[GL4] OpenGL for Java. http://www.jausoft.com. [cited on p. 25]

[Gla89] Andrew S. Glassner, editor. *An Introduction to Ray Tracing*. Academic Press Limited, 1989. ISBN 0-12-286160-4. [cited on p. 61]

[GLM96] Stefan Gottschalk, Ming Lin, and Dinesh Manocha. OBB-Tree: A hierarchical structure for rapid interference detection. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 171–180, New Orleans, Louisiana, August 1996. ACM SIGGRAPH / Addison Wesley. ISBN 0-201-94800-1. [cited on p. 5, 28, 58]

[GRLM03] Naga K. Govindaraju, Stephane Redon, Ming C. Lin, and Dinesh Manocha. CULLIDE: Interactive collision detection between complex models in large environments using graphics hardware. In *Graphics Hardware 2003*, pages 25–32, July 2003. [cited on p. 29]

[Hac62] Richard Hacker. Certification of Algorithm 112: Position of point relative to polygon. *Communications of the ACM*, 5(12):606, December 1962. [cited on p. 32]

[Han89] Pat Hanrahan. *A Survey of Ray-Surface Intersection Algorithms*, pages 79–120. In Glassner [Gla89], 1989. ISBN 0-12-286160-4. [cited on p. 19]

[HCK+99] Kenneth Hoff III, Tim Culver, John Keyser, Ming Lin, and Dinesh Manocha. Fast computation of generalized Voronoi diagrams using graphics hardware. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 277–286, Los Angeles, California, August 1999. ACM SIGGRAPH / Addison Wesley Longman. ISBN 0-20148-560-5. [cited on p. 29]

[Hei91] Tim Heidmann. Real shadows real time. *IRIS Universe*, (18):28–31, 1991. [cited on p. 30]

[HZLM01] Kenneth E. Hoff III, Andrew Zaferakis, Ming C. Lin, and Dinesh Manocha. Fast and simple 2D geometric proximity queries using graphics hardware. In *2001 ACM Symposium on Interactive 3D Graphics*, pages 145–148, March 2001. ISBN 1-58113-292-1. [cited on p. 29, 58]

[HZLM02] Kenneth E. Hoff III, Andrew Zaferakis, Ming Lin, and Dinesh Manocha. Fast 3D geometric proximity queries between rigid & deformable models using graphics hardware acceleration. Technical Report TR02-004, Department of Computer Science, University of North Carolina at Chapel Hill, 2002. [cited on p. 29, 58]

[Jor93] Camille Jordan. *Cours d'analyse de l'Ecole Polytechnique*, volume 1. 2nd edition, 1893. [cited on p. 32]

[JP02] Doug L. James and Dinesh K. Pai. DyRT: Dynamic response textures for real time deformation simulation with graphics hardware. *ACM Transactions on Graphics*, 21(3):582–585, July 2002. [cited on p. 15]

[JTT01] Pablo Jiménez, Federico Thomas, and Carme Torras. 3D collision detection: a survey. *Computers & Graphics*, 25(2):269–285, April 2001. ISSN 0097-8493. [cited on p. 5]

[KDR+01] Brucek Khailany, William J. Dally, Scott Rixner, Ujval J. Kapasi, Peter Mattson, Jin Namkoong, John D. Owens, Brian Towles, and Andrew Chang. Imagine: Media processing with streams. *IEEE Micro*, 21(2):35–46, March/April 2001. [cited on p. 9]

[KOLM02] Young J. Kim, Miguel A. Otaduy, Ming C. Lin, and Dinesh Manocha. Fast penetration depth computation for physically-based animation. In *2002 ACM SIGGRAPH Symposium on Computer Animation*, pages 23–31,187, July 2002. ISBN 1-58113-573-4. [cited on p. 29]

[LCN99] Jean-Christophe Lombardo, Marie-Paule Cani, and Fabrice Neyret. Real-time collision detection for virtual surgery. In *Computer Animation '99*, Geneva, Switzerland, May 1999. IEEE Computer Society. [cited on p. 29]

[LG98] Ming C. Lin and Stefan Gottschalk. Collision detection between geometric models: A survey. In *Proceedings of IMA Conference on Mathematics of Surfaces*, pages 37–56, Birmingham, England, September 1998. [cited on p. 5]

[LKM01] Erik Lindholm, Mark J. Kilgard, and Henry Moreton. A user-programmable vertex engine. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 149–158. ACM Press / ACM SIGGRAPH, August 2001. ISBN 1-58113-292-1. [cited on p. 8]

[LM02] Ming C. Lin and Dinesh Manocha. Interactive geometric computations using graphics hardware. In *SIGGRAPH 2002 Course Notes #31*, July 2002. [cited on p. 13]

[Mau01] Chris Maughan. *Texture Masking for Faster Lens Flare*, pages 474–480. In DeLoura [DeL01], 2001. ISBN 1-58450-054-9. [cited on p. 10]

[Mic01] Microsoft. *Microsoft XBox Developer Documentation*, 2001. [cited on p. 51]

[Mic02] Microsoft. *DirectX Developer Documentation*, 2002. [cited on p. 8]

[MOK95] Karol Myszkowski, Oleg G. Okunev, and Tosiyasu L. Kunii. Fast collision detection between complex solids using rasterizing graphics hardware. *The Visual Computer*, 11(9):497–512, 1995. ISSN 0178-2789. [cited on p. 28]

[OD01] Carol O'Sullivan and John Dingliana. Collisions and perception. *ACM Transactions on Graphics*, 20(3):151–168, July 2001. [cited on p. 46]

[O'R98] Joseph O'Rourke. *Computational Geometry In C.* Cambridge University Press, 2nd edition, 1998. ISBN 0521649765. [cited on p. 32]

[OS99] OpenGL ARB and Dave Shreiner. *OpenGL Reference Manual.* Addison Wesley, 3rd edition, 1999. ISBN 0-201-65765-1. [cited on p. 25, 50]

[OWN+99] OpenGL ARB, Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide.* Addison Wesley, 3rd edition, 1999. ISBN 0-201-60458-2. [cited on p. 29]

[PBMH02] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2002)*, 21(3):703–712, July 2002. [cited on p. 9]

[Rad] Radical Entertainment. Pure3D. [cited on p. 51]

[RB85] William T. Reeves and Ricki Blau. Approximate and probabilistic algorithms for shading and rendering structured particle systems. In *Computer Graphics (Proceedings of SIGGRAPH 85)*, volume 19, pages 313–322, San Francisco, California, July 1985. [cited on p. 15]

[Ree83] William T. Reeves. Particle systems - a technique for modeling a class of fuzzy objects. In *Computer Graphics (Proceedings of SIGGRAPH 83)*, volume 17, pages 359–376, Detroit, Michigan, July 1983. [cited on p. 15]

[RMS92] Jarek Rossignac, Abe Megahed, and Bengt-Olaf Schneider. Interactive inspection of solids: Cross-sections and interferences. In *Computer Graphics (Proceedings of SIGGRAPH 92)*, volume 26, pages 353–360, Chicago, Illinois, July 1992. ISBN 0-201-51585-7. [cited on p. 27]

[SA02] Mark Segal and Kurt Akeley. The OpenGL graphics system: A specification (version 1.4). 2002. [cited on p. 8]

[Sch98] Robert G. Scharein. *Interactive Topological Drawing*. PhD thesis, University of British Columbia, 1998. [cited on p. 51]

[SF91] Mikio Shinya and Marie-Claire Forgue. Interference detection through rasterization. *The Journal of Visualization and Computer Animation*, 2(4):131–134, 1991. [cited on p. 28]

[Shi62] M. Shimrat. Algorithm 112: Position of point relative to polygon. *Communications of the ACM*, 5(8):434, August 1962. [cited on p. 32]

[SKvW+92] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul E. Haeberli. Fast shadows and lighting effects using texture mapping. In *Computer Graphics (Proceedings of SIGGRAPH 92)*, volume 26, pages 249–252, Chicago, Illinois, July 1992. ISBN 0-201-51585-7. [cited on p. 29]

[Smi95] Alvy Ray Smith. A pixel is not a little square, a pixel is not a little square, a pixel is not a little square (and a voxel is not a little cube). Technical Memo 6, Microsoft Corporation, 1995. [cited on p. 32]

[SOG98] Noel D. Scott, Daniel M. Olsen, and Ethan W. Gannett. An overview of the VISUALIZE fx graphics accelerator hardware. *The Hewlett-Packard Journal*, pages 28–24, May 1998. [cited on p. 43]

[Sta99] Jos Stam. Stable fluids. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 121–128, August 1999. [cited on p. 15]

[TF88] Demetri Terzopoulos and Kurt Fleischer. Deformable models. *The Visual Computer*, 4(6):306–331, December 1988. [cited on p. 15]

[TPK01] Theoharis Theoharis, Georgios Papaioannou, and Evaggelia-Aggeliki Karabassi. The magic of the z-buffer: A survey. In *Proceedings of the 9th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG '2001)*, pages 379–386, February 2001. [cited on p. 29]

[TT02] Federico Thomas and Carme Torras. A projectively invariant intersection test for polyhedra. *The Visual Computer*, 18(7):405–414, 2002. ISSN 0178-2789. [cited on p. 31]

[vdDKP01] Kees van den Doel, Paul G. Kry, and Dinesh K. Pai. FoleyAutomatic: Physically-based sound effects for interactive simulation and animation. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 537–544. ACM Press / ACM SIGGRAPH, August 2001. ISBN 1-58113-292-1. [cited on p. 26]

[VSC01] Tzvetomir Vassilev, Bernhard Spanlang, and Yiorgos Chrysanthou. Fast cloth animation on walking avatars. *Computer Graphics Forum (Proceedings of Eurographics 2001)*, 20(3):260–267, 2001. ISSN 1067-7055. [cited on p. 28, 48]

[WH94] Andrew P. Witkin and Paul S. Heckbert. Using particles to sample and control implicit surfaces. In *Proceedings of SIGGRAPH 94*, Computer Graphics Proceedings, Annual Conference Series, pages 269–278, July 1994. [cited on p. 15]

[Wil78] Lance Williams. Casting curved shadows on curved surfaces. In *Computer Graphics (Proceedings of SIGGRAPH 78)*, volume 12, pages 270–274, Atlanta, Georgia, August 1978. [cited on p. 29]

[WSM02] Clemens Wagner, Markus A. Schill, and Reinhard Männer. Collision detection and tissue modeling in a VR-simulator for eye surgery. In *Proceedings of the Eurographics Workshop on Virtual Environments 2002*, pages 27–36, Barcelona, Spain, May 2002. Eurographics Association. ISBN 1-58113-535-1. [cited on p. 29]