Chapter 2

Collision Detection Design Issues

Designing an efficient collision detection system is a bit like putting a puzzle together: a lot of pieces must be connected before the big picture starts to appear. In a similar fashion, the majority of this book is concerned with examining the individual pieces that go into different approaches to collision detection. The big picture will become clear over the course of the book. This chapter provides a quick overview of a number of issues that must be considered in selecting among approaches, and how the components of these approaches relate. This chapter also introduces a number of terms, defined and explained further in following chapters. More in-depth coverage of the items touched upon here is provided throughout remaining chapters of the book.

2.1 Collision Algorithm Design Factors

There are several factors affecting the choices made in designing a collision detection system. These factors will be broken down into the following categories:

1. Application domain representation. The geometrical representations used for the scene and its objects have a direct bearing on the algorithms used. With fewer restrictions put on these representations, more general collision detection solutions have to be used, with possible performance repercussions.

2. Different types of queries. Generally, the more detailed query types and results are, the more computational effort required to obtain them. Additional data structures may be required to support certain queries. Not all object representations support all query types.

3. Environment simulation parameters. The simulation itself contains several parameters having a direct impact on a collision detection system. These include how
many objects there are, their relative sizes and positions, if and how they move, if they are allowed to interpenetrate, and whether they are rigid or flexible.

4. *Performance*. Real-time collision detection systems operate under strict time and size restrictions. With time and space always being a trade-off, several features are usually balanced to meet stated performance requirements.

5. *Robustness*. Not all applications require the same level of physical simulation. For example, stacking of bricks on top of each other requires much more sophistication from a collision detection system than does having a basketball bouncing on a basketball court. The ball bouncing slightly too early or at a somewhat larger angle will go unnoticed, but even the slightest errors in computing contact points of stacked bricks is likely to result in their slowly starting to interpenetrate or slide off each other.

6. *Ease of implementation and use*. Most projects are on a time frame. Scheduling features of a collision detection system means nothing if the system cannot be completed and put in use on time. Decisions regarding implementational simplicity therefore play a large role in what approach is taken.

These issues are covered in further detail in the remainder of the chapter.

### 2.2 Application Domain Representation

To select appropriate collision detection algorithms, it is important to consider the types of geometrical representations that will be used for the scene and its objects. This section talks briefly about various object representations, how simplified geometry can be used instead of modeling geometry, and how application-specific knowledge can allow specialized solutions to be used over more generic solutions.

#### 2.2.1 Object Representations

Most current hardware uses triangles as the fundamental rendering primitive. Consequently, a *polygonal representation* is a natural choice for scenes and scene objects, as well as for their corresponding collision geometry. The most generic polygonal representation is the *polygon soup*: an unordered collection of polygons with no connectivity information specifying how one polygon relates to another. With no inherent constraints, the polygon soup is an attractive representation for artists and level designers. Algorithms operating on polygon soups apply to any collection of polygons but tend to be less efficient and less robust than those relying on additional information. For example, a polygon soup contains no information regarding the “inside” of an object, so there is no easy way of finding out if an object has somehow
Figure 2.1 Geometrical models, like the one pictured, are commonly built from a collection of polygon meshes.

erroneously ended up inside another object. The additional information mentioned could include which edges connect to what vertices and what faces connect to a given face, whether the object forms a closed solid, and whether the object is convex or concave.

Polygons may be connected to one another at their edges to form a larger polygonal surface called a polygon mesh. Building objects from a collection of polygon meshes is one of the most common methods for authoring geometrical models (Figure 2.1).

Polygonal objects are defined in terms of their vertices, edges, and faces. When constructed in this way, objects are said to have an explicit representation. Implicit objects refer to spheres, cones, cylinders, ellipsoids, tori, and other geometric primitives that are not explicitly defined in such a manner but implicitly through a mathematical expression. Implicit objects are often described as a function mapping from 3D space to real numbers, \( f : \mathbb{R}^3 \to \mathbb{R} \), where the points given by \( f(x, y, z) < 0 \) constitute the interior, \( f(x, y, z) = 0 \) the boundary, and \( f(x, y, z) > 0 \) the exterior of the object (Figure 2.2). An object boundary defined by an implicit function is called an implicit surface. Implicit objects can be used as rough approximations of scene objects for quick rejection culling. The implicit form may allow for fast intersection tests, especially with lines and rays — a fact utilized in ray tracing applications. Several examples of implicit tests are provided in Chapter 5.

Convex polygonal objects can also be described as the intersection of a number of halfspaces. For example, a cube can be expressed as the intersection of six halfspaces, each halfspace "trimming away" the portion of space that lies outside a face of
Figure 2.2 An implicitly defined sphere (where the sphere is defined as the boundary plus the interior).

Figure 2.3 (a) A cube with a cylindrical hole through it. (b) The CSG construction tree for the left-hand object, where a cylinder is subtracted from the cube.

the cube. Halfspaces and halfspace intersection volumes are described in more detail in Chapter 3.

Geometric primitives such as spheres, boxes, and cylinders are also the building blocks of objects constructed via the constructive solid geometry (CSG) framework. CSG objects are recursively formed through applying set-theoretic operations (such as union, intersection, or difference) on basic geometric shapes or other CSG objects, allowing arbitrarily complex objects to be constructed. Thus, a CSG object is represented as a (binary) tree, with set-theoretic operations given in the internal nodes and geometry primitives in the leaves (Figure 2.3). CSG objects are implicit in that vertices, edges, and faces are not directly available.

A strength of CSG modeling is that the resulting objects are always valid — without cracks and other problems that plague polygonal representations. CSG is also a volume representation, making it easy to determine if, for example, a query point lies inside the CSG object. CSG on polyhedral objects can be implemented through the processes described in, for example, [Laidlaw86] and [Thibault87]. However, it can be difficult to achieve robust implementations due to numerical imprecision in the calculations involved.
2.2.2 Collision Versus Rendering Geometry

Although it is possible to pass rendering geometry directly into a collision system, there are several reasons it is better to have separate geometry with which collision detection is performed.

1. Graphics platforms have advanced to the point where rendering geometry is becoming too complex to be used to perform collision detection or physics. In addition, there is a usually a limit as to how accurate collisions must be. Thus, rather than using the same geometry used for rendering, a simplified proxy geometry can be substituted in its place for collision detection. For games, for example, it is common to rely on simple geometric shapes such as spheres and boxes to represent the game object, regardless of object complexity. If the proxy objects collide, the actual objects are assumed to collide as well. These simple geometric shapes, or bounding volumes, are frequently used to accelerate collision queries regardless of what geometry representation is used. Bounding volumes are typically made to encapsulate the geometry fully. Bounding volumes are discussed in detail in Chapter 4.

2. For modern hardware, geometry tends to be given in very specific formats (such as triangle strips and indexed vertex buffers), which lend themselves to fast rendering but not to collision detection. Rather than decoding these structures on the fly (even though the decoded data can be cached for reuse), it is usually more efficient to provide special collision geometry. In addition, graphics hardware often enforces triangle-only formats. For collision geometry, efficiency sometimes can be had by supporting other, nontriangle, primitives.

3. The required data and data organization of rendering geometry and collision geometry are likely to vary drastically. Whereas static rendering data might be sorted by material, collision data are generally organized spatially. Rendering geometry requires embedded data such as material information, vertex colors, and texture coordinates, whereas collision geometry needs associated surface properties. Separating the two and keeping all collision-relevant information together makes the collision data smaller. Smaller data, in turn, leads to efficiency improvements due to better data cache coherency.

4. Sometimes the collision geometry differs from the rendered geometry by design. For example, the knee-deep powder snow in a snowboarding game can be modeled by a collision surface two feet below the rendered representation of the snow surface. Walking in ankle-deep swaying grass or wading in waist-deep murky water can be handled similarly. Even if rendering geometry is used as collision geometry, there must be provisions for excluding some rendering geometry from (and for including additional nonrendering geometry in) the collision geometry data set.

5. For simulation purposes, collision data must be kept around even when rendering data can be thrown out as not visible. With the collision geometry being smaller
than the corresponding rendering geometry, the permanent memory footprint is therefore reduced.

6. The original geometry might be given as a polygon soup or mesh, whereas the simulation requires a solid-object representation. In this case, it is much easier to compute solid proxy geometry than to attempt to somehow solidify the original geometrical representation.

However, there are some potential drawbacks to using separate collision geometry.

1. Data duplication (primarily of vertices) causes additional memory to be used. This problem may be alleviated by creating some or all of the collision geometry from the rendering geometry on the fly through linearization caching (as described in Section 13.5 and onward).

2. Extra work may be required to produce and maintain two sets of similar geometry. Building the proxy geometry by hand will impair the schedule of the designer creating it. If it is built by a tool, that tool must be written before the collision system becomes usable. In addition, if there is a need to manually modify the tool output, the changes must somehow be communicated back into the tool and the original data set.

3. If built and maintained separately, the rendering and collision geometries may mismatch in places. When the collision geometry does not fill the same volume as the render geometry, objects may partially disappear into or float above the surface of other objects.

4. Versioning and other logistics problems can show up for the two geometries. Was the collision geometry really rebuilt when the rendering geometry changed? If created manually, which comes first: collision geometry or rendering geometry? And how do you update one when the other changes?

For games, using proxy geometry that is close to (but may not exactly match) actual visuals works quite well. Perceptually, humans are not very good at detecting whether exact collisions are taking place. The more objects involved and the faster they move, the less likely the player is to spot any discrepancies. Humans are also bad at predicting what the outcome of a collision should be, which allows liberties to be taken with the collision response as well. In games, collision detection and response can effectively be governed by "if it looks right, it is right." Other applications have stricter accuracy requirements.

2.2.3 Collision Algorithm Specialization

Rather than having one all-encompassing collision detection system, it is often wise to provide specialized collision systems for specific scenarios. An example of where
specialization is relevant is particle collisions. Rather than sending particles one by one through the normal collision system, they are better handled and submitted for collision as groups of particles, where the groups may form and reform based on context. Particles may even be excluded from collision, in cases where the lack of collision is not noticeable.

Another example is the use of separate algorithms for detecting collision between an object and other objects and between the object and the scene. Object-object collisions might even be further specialized so that a player character and fast-moving projectiles are handled differently from other objects. For example, a case where all objects always collide against the player character is better handled as a hard-coded test rather than inserting the player character into the general collision system.

Consider also the simulation of large worlds. For small worlds, collision data can be held in memory at all times. For the large, seamless world, however, collision data must be loaded and unloaded as the world is traversed. In the latter case, having objects separate from the world structure is again an attractive choice, so the objects are not affected by changes to the world structure. A possible drawback of having separate structures for holding, say, objects and world, is that querying now entails traversing two data structures as opposed to just one.

2.3 Types of Queries

The most straightforward collision query is the interference detection or intersection testing problem: answering the Boolean question of whether two (static) objects, \( A \) and \( B \), are overlapping at their given positions and orientations. Boolean intersection queries are both fast and easy to implement and are therefore commonly used. However, sometimes a Boolean result is not enough and the parts intersecting must be found. The problem of intersection finding is a more difficult one, involving finding one or more points of contact.

For some applications, finding any one point in common between the objects might be sufficient. In others, such as in rigid-body simulations, the set of contacting points (the contact manifold) may need to be determined. Robustly computing the contact manifold is a difficult problem. Overall, approximate queries — where the answers are only required to be accurate up to a given tolerance — are much easier to deal with than exact queries. Approximate queries are commonplace in games. Additionally, in games, collision queries are generally required to report specific collision properties assigned to the objects and their boundaries. For example, such properties may include slipperiness of a road surface or climbability of a wall surface.

If objects penetrate, some applications require finding the penetration depth. The penetration depth is usually defined in terms of the minimum translational distance: the length of the shortest movement vector that would separate the objects. Computing this movement vector is a difficult problem, in general. The separation distance between two disjoint objects \( A \) and \( B \) is defined as the minimum of the distances between points
in A and points in B. When the distance is zero, the objects are intersecting. Having a distance measure between two objects is useful in that it allows for prediction of the next time of collision. A more general problem is that of finding the closest points of A and B: a point in A and a point in B giving the separation distance between the objects. Note that the closest points are not necessarily unique; there may be an infinite number of closest points. For dynamic objects, computing the next time of collision is known as the estimated time of arrival (ETA) or time of impact (TOI) computation. The ETA value can be used to, for instance, control the time step in a rigid-body simulation. Type of motion is one of the simulation parameters discussed further in the next section.

2.4 Environment Simulation Parameters

As mentioned earlier in the chapter, several parameters of a simulation directly affect what are appropriate choices for a collision detection system. To illustrate some of the issues they may cause, the following sections look specifically at how the number of objects and how the objects move relate to collision processing.

2.4.1 Number of Objects

Because any one object can potentially collide with any other object, a simulation with \( n \) objects requires \( (n-1) + (n-2) + \cdots + 1 = n(n-1)/2 = O(n^2) \) pairwise tests, worst case. Due to the quadratic time complexity, naively testing every object pair for collision quickly becomes too expensive even for moderate values of \( n \). Reducing the cost associated with the pairwise test will only linearly affect runtime. To really speed up the process, the number of pairs tested must be reduced. This reduction is performed by separating the collision handling of multiple objects into two phases: the broad phase and the narrow phase.

The broad phase identifies smaller groups of objects that may be colliding and quickly excludes those that definitely are not. The narrow phase constitutes the pairwise tests within subgroups. It is responsible for determining the exact collisions, if any. The broad and narrow phases are sometimes called n-body processing and pair processing, respectively.

Figure 2.4 illustrates how broad-phase processing reduces the workload through a divide-and-conquer strategy. For the 11 objects (illustrated by boxes), an all-pairs test would require 55 individual pair tests. After broad-phase processing has produced 5 disjoint subgroups (indicated by the shaded areas), only 10 individual pair tests would have to be performed in the narrow phase. Methods for broad-phase processing are discussed in Chapters 6 through 8. Narrow-phase processing is covered in Chapters 4, 5, and 9.

In addition to the number of objects, the relative size of the objects also affects how many tests have to be performed. With both small and large objects present in a scene,
2.4 Environment Simulation Parameters

Having identified a collision, the next step is prioritization of collision candidates. In the broad phase, we identify contact points between potential colliders, and then, using the currently detected objects, we may be able to deduce the earliest time of impact (TOI) for each collision. This step in a sequence of techniques is discussed further in Chapter 7.

The broad-phase system generally must work harder (or be more sophisticated) to identify groups than it would for a set of homogeneously sized objects. How object size affects broad-phase methods is discussed further in Chapter 7.

2.4.2 Sequential Versus Simultaneous Motion

In real life, objects are moving simultaneously during a given movement time step, with any eventual collisions resolved within the time step. For an accurate computer simulation of the real-life event, the earliest time of contact between any two of the moving objects would somehow have to be determined. The simulation can then be advanced to this point in time, moving all objects to the position they would be in when the first collision occurs. The collision is then resolved, and the process continues determining the next collision, repeating until the entire movement time step has been used up.

Executing a simulation by repeatedly advancing it to the next earliest time of contact becomes quite expensive. For example, as one or more objects come to rest against a surface, the next time of collision follows almost immediately after the current time of collision. The simulation is therefore only advanced by a small fraction, and it can take virtually “forever” to resolve the full movement time step. One solution to this problem is to use the broad phase to identify groups of objects that may interact within the group, but not with objects of other groups during the time step. The simulation of each group can therefore proceed at different rates, helping to alleviate the problem in general.

An alternative option is to move objects simultaneously, but setting a fixed (small) time step for the movement. Simultaneous movement can result in objects interpenetrating, which typically must be dealt with somehow, for example, by backing up the simulation to an earlier state. In both cases, simultaneous updates remain expensive and are therefore often reserved for accurate rigid-body simulations. However, many games, as well as other applications, are not rigid-body simulations and it would be overkill and wasted effort to try to simulate motion with high accuracy. For these, an alternative option is to resolve motion sequentially. That is, objects are moved...
Figure 2.5 (a) Top: If both objects move simultaneously, there is no collision. Bottom: If the circle object moves before the triangle, the objects collide. In (b), again there is no collision for simultaneous movement, but for sequential movement the objects collide. (c) The objects collide under simultaneous movement, but not under sequential movement.

one object at a time and any collisions are detected and resolved before the process continues with the next object.

Clearly, sequential movement is not a physically accurate movement model. Some objects may collide with objects that have not yet moved in this frame but that would have moved out of the way were the two objects moving simultaneously (Figure 2.5a). Other objects may collide with objects that moved before they did and are now in their path (Figure 2.5b). In some cases, where two simultaneously moving objects would have collided halfway through their motion, collisions will now be missed as one object is moved past the other (Figure 2.5c). For games, for example, the problems introduced by a sequential movement model can often be ignored. The high frame rate of games often makes the movement step so small that the overlap is also small and not really noticeable.

One of the benefits of the sequential movement model is that an object nonpenetration invariant is very easy to uphold. If there is a collision during the movement of an object, the movement can simply be undone (for example). Only having to undo the movement of a single object should be contrasted with the simultaneous movement model using a fixed time step, where the movement of all simultaneously moved objects would have to be undone.

2.4.3 Discrete Versus Continuous Motion

Something that can greatly affect both the computational effort needed to determine a collision result and the accuracy of the result itself is if the two objects involved in a pairwise test are moving at the time of testing. Static collision detection
involves detecting intersection between the objects, at discrete points in time, during their motion. At each such point in time the objects are treated as if they were stationary at their current positions with zero velocities. In contrast, *dynamic collision detection* considers the full continuous motion of the objects over the given time interval. Dynamic collision tests can usually report the exact time of collision and the point(s) of first contact. Static tests are (much) cheaper than dynamic tests, but the time steps between tests must be short so that the movement of the objects is less than the spatial extents of the objects. Otherwise, the objects may simply pass each other from one time step to the next without a collision being detected. This phenomenon is referred to as *tunneling*.

The volume covered by an object in continuous motion over a given time interval is called the *swept volume*. If the swept volumes of two moving objects do not intersect, there is no intersection between the objects. Even if the swept volumes intersect, the objects still may not intersect during movement. Thus, intersection of the swept volumes is a necessary, but not sufficient, condition for object collision. For complex motions, the swept volume is both difficult to compute and to work with. Fortunately, perfect accuracy is rarely necessary. Dynamic collision testing of complex tumbling motions can usually be simplified by assuming a piecewise linear motion; that is, a linear translation over the range of movement, with an instantaneous rotation at the end (or start) of the motion. Somewhere between these two alternatives is replacement of the unrestricted motion with a screw motion (that is, a fixed rotational and translational motion).

When working with moving objects it is virtually always preferable to consider the relative motion of the objects by subtracting the motion of the one object off the other object, thus effectively leaving one object static. Assuming linear translational motion for the objects makes this operation a simple vector subtraction. A key benefit of considering only the relative motion is that for testing one moving object against a stationary object a swept volume test is now an exact intersection test. In games, the entire swept volume is sometimes just replaced by a *speedbox*; an elongated box covering the object for its full range of motion (or some similarly simple proxy object, not necessarily a box).

### 2.5 Performance

Taking game consoles as an example, for the best possible visuals games must run at 60 fps (in countries with NTSC TV format; 50 fps in PAL territory). This frame rate leaves 16.7 ms to prepare each game frame. Depending on the type of game, collision detection may account for, say, 10 to 30% of a frame, in turn leaving 2 to 5 ms for collision detection. For an action platform game that may have dozens of collision-dependent objects active at a given time there may be only about 50 to 250 µs available to handle the collision for each object — not a lot of time. Clearly, it is very important to reduce the average running time of collision queries. However, as large sudden drops in frame rate are very noticeable in games (in a bad way) it is
also important to make sure the worst case for the selected algorithms is not taking a magnitude longer than the average case.

A number of things can be done to speed up collision processing, which in large part is what this book is about. Some general ideas of what optimizations are relevant for collision detection are discussed in the next section.

### 2.5.1 Optimization Overview

The first tenet of optimization is that nothing is faster than not having to perform a task in the first place. Thus, some of the more successful speed optimizations revolve around pruning the work as quickly as possible down to the minimum possible. As such, one of the most important optimizations for a collision detection system is the broad-phase processing mentioned in Section 2.4.1: the exploitation of objects' spatial locality. Because objects can only hit things that are close to them, tests against distant objects can be avoided by breaking things up spatially. Tests are then only made against the regions immediately nearby the object, ignoring those that are too far away to intersect the object. There are strong similarities between this spatial partitioning and what is done for view frustum culling to limit the number of graphical objects drawn.

Spatial partitioning can be performed using a flat structure, such as by dividing space into a grid of cells of a uniform size. It also can be implemented in terms of a hierarchy, where space is recursively divided in half until some termination goal is met. Objects are then inserted into the grid or the hierarchy. Grids and hierarchical partitioning are also useful for the pair tests of the narrow phase, especially when the objects have high complexity. Rather than having to test an entire object against another, they allow collision tests to be limited to the parts of two objects nearest each other. Object and spatial partitioning are discussed in Chapters 6 and 7.

Doing inexpensive bounding volume tests before performing more expensive geometric tests is also a good way of reducing the amount of work needed to determine a collision. Say encompassing bounding spheres have been added to all objects, then a simple sphere-sphere intersection test will now show — when the spheres do not overlap — that no further testing of the complex contained geometry is necessary. Bounding volumes are covered in Chapter 4.

The insight that objects tend to take small local steps from frame to frame — if moving at all — leads to a third valuable optimization: to exploit this temporal (or frame-to-frame) coherency. For example, only objects that have moved since the last frame need to be tested; the collision status remains the same for the other objects. Temporal coherency may also allow data and calculations to be cached and reused over one or more future frames, thus speeding up tests. Assumptions based on movement coherency are obviously invalidated if objects are allowed to "teleport" to arbitrary locations. Coherence is further discussed in Chapter 9.

Last, architecture-specific optimizations are also very important. Many platforms support some type of code or data parallelism that when fully exploited can provide
large speedups. Due to big differences between the speed at which CPUs operate and the speeds at which main memory can provide data for it to operate on (with the speed advantage for the CPU), how collision geometry and other data are stored in memory can also have a huge speed impact on a collision system. These issues are covered in detail in Chapter 13.

2.6 Robustness

Collision detection is one of a number of geometrical applications where robustness is very important. In this book, robustness is used simply to refer to a program’s capability of dealing with numerical computations and geometrical configurations that in some way are difficult to handle. When faced with such problematic inputs, a robust program provides the expected results. A nonrobust program may in the same situations crash or get into infinite loops. Robustness problems can be broadly categorized into two classes: those due to lack of numerical robustness and those due to lack of geometrical robustness.

Numerical robustness problems arise from the use of variables of finite precision during computations. For example, when intermediate calculations become larger than can be represented by a floating-point or an integer variable the intermediate result will be invalid. If such problems are not detected, the final result of the computation is also likely to be incorrect. Robust implementations must guarantee such problems cannot happen, or if they do that adjusted valid results are returned in their stead.

Geometrical robustness entails ensuring topological correctness and overall geometrical consistency. Problems often involve impossible or degenerate geometries, which may be the result of a bad numerical calculation. Most algorithms, at some level, expect well-formed inputs. When given bad input geometry, such as triangles degenerating to a point or polygons whose vertices do not all lie in the plane, anything could happen if these cases are not caught and dealt with.

The distinction between numerical and geometrical robustness is sometimes difficult to make, in that one can give rise to the other. To avoid obscure and difficult-to-fix runtime errors, robustness should be considered throughout both design and development of a collision detection system. Chapters 11 and 12 discuss robustness in more depth.

2.7 Ease of Implementation and Use

In the case of a collision detection system implemented from scratch, the issue of expected development time might be as important as the desired feature set. For example, games are often on tight budgets and time frames, and the delay of any
critical component could be costly. In evaluating the ease of implementation it is of interest to look at not just the overall algorithm complexity but how many and what type of special cases are involved, how many tweaking variables are involved (such as numerical tolerances), and other limitations that might affect the development time.

Several additional issues relate to the use of the collision detection system. For example, how general is the system? Can it handle objects of largely varying sizes? Can it also answer range queries? How much time is required in the build process to construct the collision-related data structures? For the latter question, while the time spent in preprocessing is irrelevant for runtime performance it is still important in the design and production phase. Model changes are frequent throughout development, and long preprocessing times both lessen productivity and hinder experimentation. Some of these problems can be alleviated by allowing for a faster, less optimized data structure construction during development and a slower but more optimal construction for non-debug builds.

2.7.1 Debugging a Collision Detection System

Just like all code, collision detection systems are susceptible to errors. Finding these errors can sometimes be both difficult and time consuming. Steps can be taken during development to make this debugging process less painful. Some good ideas include:

- Keep a cyclic buffer of the arguments to the n last collision queries, corresponding to up to a few seconds' worth of data (or more). Then, when something goes visually wrong, the program can be paused and the data can be output for further analysis, such as stepping through the calls with the saved arguments. The logged data may also provide useful information when asserts trigger.

- Provide means to visualize the collision geometry. For example, you might visualize tested faces, their collision attributes, and any hierarchies and groupings of the faces. Additionally, visualize the collision queries themselves, preferably with the history provided by the cyclic buffer mentioned earlier. This visualization provides a context that makes it easy to spot bad collision queries.

- Implement a simple reference algorithm (such as a brute-force algorithm that tests all objects or all polygons against each other) and run the reference algorithm in parallel with the more sophisticated algorithm. If the results differ, there is a problem (in all likelihood with the more advanced algorithm).

- Maintain a test suite of queries, and run the collision code against the test suite when the code changes. Code of geometric nature tends to have many special cases, and having a set of comprehensive tests helps in trapping problems early. Whenever a bug is found, add test cases to detect if it is ever reintroduced.
Of course, all general debugging strategies such as liberal use of \texttt{assert()} calls apply as well. A good discussion of such strategies is found in [McConnell93, Chapter 26].

### 2.8 Summary

This chapter has outlined the many factors that must be considered when designing and implementing a collision detection system. It talked about possible collision geometry representations, and whether to use rendering geometry or specialized geometry to perform collision tests. Collision processing was defined as taking place in two phases, the narrow and broad phase. The broad phase is concerned with coarsely identifying small subgroups of potentially colliding objects, whereas the narrow phase performs the detailed pairwise collision tests between objects. Narrow-phase testing is the primary topic of Chapters 4 and 5, where many different query types on a wide variety of geometrical object representations are discussed. Narrow-phase testing is also discussed in Chapter 9. Broad-phase methods are discussed in Chapters 6 through 8. The importance of robustness was stressed. This book devotes two full chapters to the topic of robustness, Chapters 11 and 12. Because this book is about collision detection for real-time applications, performance was also stressed as an important system consideration. In some respect, the majority of the book is about performance in that efficient algorithms and data structures are discussed throughout. The book concludes with discussion of optimization, in Chapter 13.