Workshop Series: Collision Detection (Broad)

Benjamin Kenwright[1]*

Abstract
We expand upon our collision detection algorithms discussed in the previous practical to make them suitable for real-time simulations. We discuss the two-tiered approach which is typically taken in collision detection (i.e., broad-phase and narrow-phase), while implementing a basic set of broad-phase detection routines.

Keywords
Collision Detection, Intersection, Sphere-Sphere, Box-Plane, Axis Aligned Bounding Box, Broad, Narrow, Physics, Penetration Depth, Contact Normal, Real-Time, Video Games, Interactive, Classical Mechanics, C++

[1] *Workshop Series (www.xbdev.net) - Benjamin Kenwright*

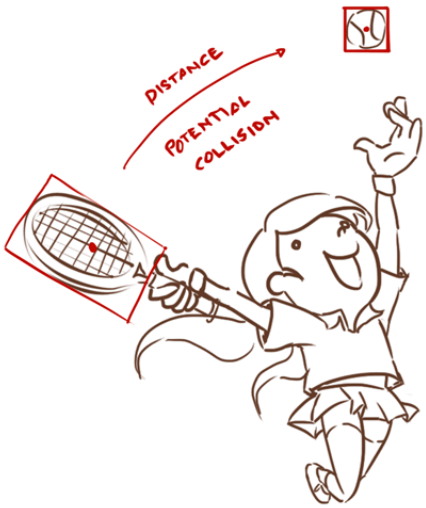

Figure 1. Narrow Phase - In broad phase, the task is to avoid performing expensive computations for bodies that are far away from each other. Simple bounding boxes can be placed around each of the bodies, and simple tests can be performed to avoid costly collision checking unless the boxes overlap.

Contents

Introduction

Overview A physics-based simulation system, typically consists of three major sections:

- Simulate the motion of objects (e.g., particles or rigid bodies) in the world
- Detect when two objects have collided or intersected
- Resolve any collisions to remove inter-penetration

In this practical, we discuss and implemented broad-phased collision detection algorithms. While we are able to move objects around the three-dimensional simulated world in a believable manner and detect precise collisions using a narrow-phase approach, the majority of the time objects do not interact with one another. Hence, we need a computationally efficient system to organise and quickly cull which objects are possibly intersecting.

Essentials The collision detection system is split into two levels. The first level is a broad-phase approach that quickly culls and removes objects that cannot be colliding, so we are left with a reduced list of possibly colliding object. The

second level takes the possible list of objects and performs a more accurate set of checks to determine if objects are colliding and the necessary contact information. We can run a physical simulation using only a set of narrow-phase collision algorithms - however, the simulation will be computationally slow. The broad-phase level offers a number of smarter less accurate speed-up techniques that make the overall solution run at an acceptable frame-rate.

1. Organising Collision Detection (Broad-phase and Narrow-phase)

Principles and Concepts The goal of the collision detection section is to identify which objects in our simulation have intersected, so that we can push them apart. The sledgehammer approach to this would be test every simulated object in the world against every other simulated object. This is a N^2 problem and obviously it quickly becomes impractical for even a relatively small game world.

Structure The collision detection routines will be much more efficient if we can structure the code in a way which ensures that the more complex algorithms are only applied to pairs of objects which are likely to have intersected. Typically a physics engine will divide the collision detection mechanism into two phases:

- Broad-phase. Identify which pairs of objects need to be tested for intersection.
- Narrow-phase. Carry out collision tests for each pair of objects.

We will discuss each of these phases, before moving on to the specifics of the detection algorithms that are commonly employed.

2. Broad-phase

Imagine that the collision detection loop starts with a long list of paired objects, consisting of all N2 possible pairings in the simulated world. The purpose of the Broad-phase is to provide a very quick culling procedure in order to "throw out" as many of these pairings as possible, before moving on to the more complex algorithms during narrow-phase. It is analogous to the frustum culling and other techniques that were introduced during the Graphics course to reduce the number of graphical objects submitted for rendering.

Frustum culling may initially appear to be a good option for use in broad-phase, especially as the work has already been done for the rendering loop. However there are some pretty big disadvantages to this approach, as we would not be testing for collision between any objects out of view - for example, if the player turns around there could well be a big mess of intersecting objects lurking behind him.

A simple but effective approach is to carry out a quick bounding box test between all object pairs. This method culls any object pairs which have no chance of their extremities being in contact. The algorithm and implementation of this approach are discussed in detail in a later section of this practical. While this is a computationally cheap test per object pair, it still must be carried out for every possible pairing (i.e. it is still N2). A far more efficient routine would involve grouping nearby objects together is some way, so that entire groups of object pairs can be culled from the list at once.

2.1 BSP Trees and World Partition

There are various ways in which the objects can be grouped to facilitate this faster culling during broad-phase. In general, the techniques involve dividing the game world into a number of sections, and identifying which section each object belongs to. The assumption is that any object can only be colliding with another object that is in the same section. The bounding box test then just needs to be carried out between objects in the same section. The problem then becomes a series of much smaller N2 tests, where N is now the number of objects in each group rather than the entire world. It should be noted that any particular object can be in more than one world section at the same time; this is perfectly acceptable and the object just needs to be included in the loop for each section that it encroaches on.

The main technique for achieving this goal is Binary Search Partitioning (BSP) which recursively sub-divides the world into convex sections. The Octree and Quadtree are commonly used examples of BSP trees. The octree approach splits the world into progressively smaller cubes (i.e. it divides each cube into eight equally sized smaller cubes - hence the name octree). If a cube contains more than a threshold number of objects, then it is split down into eight further cubes, and so on recursively. The quadtree, as the name suggests, carries out the same process but in two dimensions only, so each square is split into four smaller squares.

2.2 Sort and Sweep

A further step in reducing the number of more expensive object pair tests is to implement a Sort and Sweep algorithm. The bounding volume of each object is projected onto a single axis. If the bounding extents of a particular pair of objects do not overlap then they can not possibly have collided, so that pair is discarded from the list of possible intersecting pairs.

This is achieved by constructing a list of all bounding markers (i.e. two per object, leading to a list of 2n items) along a particular axis. The list is then sorted and traversed. Whenever the start of a bounding volume is found (i.e. an a value in the diagram), that object is added to the active list, when the corresponding end of the bounding value is found (i.e. a b value in the diagram) it is removed from the list. When an object is added to the active list, any other objects on the list are potential collision candidates, so an object pair is created for the new object and each of the currently active objects.

In the example diagram shown, the sort and sweep algorithm creates a possible collision pair for the first and second object, and another for the second and third object, but not for

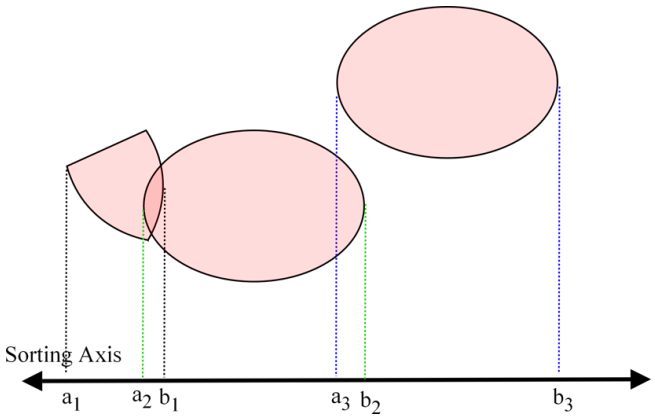

Figure 2. Sort and Sweep - Sorting along each of the primary axes.

the first and third object. So, when the more computationally expensive narrow-phase routines cycle through the possible collisions, only two of the pairs are considered. Obviously expanding this method to a larger area with many more objects can greatly increase the number of potentially intersecting pairs which are culled during broad-phase.

2.3 Narrow-phase

The list of N^2 object pairs that must be checked for intersection should have been greatly reduced by the broad-phase algorithms, leaving a list of object pairs which we believe have a higher chance of having collided. The narrow-phase algorithms test these individual pairs in more detail, so as to identify which objects have actually intersected and need to be moved apart.

The narrow-phase can operate at various different levels of detail, depending on the circumstances. In many cases a test of simplified collision shapes will be ample, but in some cases the algorithms need to get down to the polygonal level. The rest of this practical is focused on describing and implementing the basic algorithms which can be used, either for a simple narrow-phase test, or during the broad-phase. The next practical then describes the more complex algorithms used during narrow-phase for polygonal level collision tests

3. Collision Data

3.1 Collision Shape Data

In the first practical, in this series we discussed the necessity for differentiating between the graphical model which is displayed on screen (the vertex lists, texture data, etc.), and the physical model which is simulated by the physics engine (the size, shape and mass). The prime reason for taking this approach is the efficiency benefits that it brings to the collision detection and response algorithms.

If we need to check for collisions with every single polygon of a complex graphical model, then we are faced with a highly computationally intensive task. Even if this were possible at the kind of real time frame-rate that we strive for, the player is unlikely to notice such extreme attention to detail. Instead, each game object is represented by a simple shape, or a collection of simple shapes, for the purpose of physical simulation. For example, a telegraph pole may be simulated as a tall thin cuboid, while a hand grenade may be simulated as a sphere. The more complex the shape that is used to simulate an object, the more complex the collision algorithms need to be, so care must be taken to use the appropriate simulation shapes for each object in various circumstances during the game.

3.2 Collision Response Data

The aim of the collision detection algorithms is not only to flag when an intersection of two bodies has occurred, but to provide information on how to resolve that intersection. Basically we need to know where to move the bodies so that there is no longer an intersection.

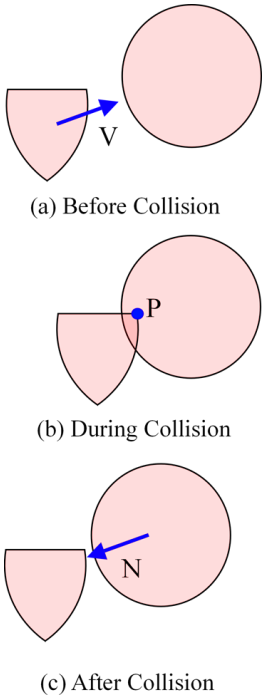

Figure 3. Collision - Resolving interpenetration

The diagram shows a rectangular block approaching a static sphere with velocity V . At the end of the motion update phase of the simulation, the collision detection algorithms detect an intersection which must be resolved (i.e. the corner of the block has penetrated the sphere). The information which the collision detection algorithms must provide, in order to resolve this intersection, is:

- The contact point P where the intersection has been detected.
- The contact normal N which represents the direction vector along which the intersecting object needs to move to resolve the collision.

- The penetration depth, i.e. the minimum distance along the normal that the intersecting object must move.

In the example diagram, the collision response routine has moved the block out along the normal of the sphere so that it is no longer intersecting. The manner in which this is achieved is addressed in the next practical; for now we need to concentrate on how the physics engine recognised that an intersection had occurred, and how it calculated the information required by the collision response algorithms.

4. Collision Detection Algorithms

There are many different algorithms which can be used to detect whether three-dimensional bodies are intersecting. As ever, the more accurate routines are more computationally expensive, so care should be taken when choosing which algorithm to use in which circumstances. This will be discussed further toward the end of this practical. First, let's look at some algorithms in more detail.

For many of these examples, we will use two dimensional drawings to clarify the algorithms. The algorithms themselves can be implemented in the same way in three dimensions, but often discussing them in two dimensions, particularly in relation to diagrams, is more intuitive. For example, the diagram in the previous section is actually of a circle and an oblong, although it purports to be a sphere and a block; hopefully this approach will help clarify the concepts under discussion.

4.1 Sphere-Sphere Collision

The simplest approach to collision detection is to represent each object as a sphere centred on the object's position vector, and to calculate whether the two spheres intersect.

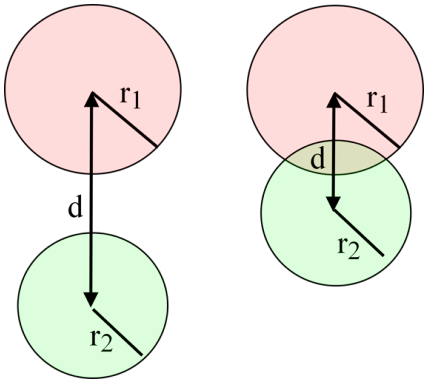

Figure 4. Sphere Sphere Collision - Detecting and gathering contact information for a sphere-sphere collision

The algorithm to detect whether two spheres intersect is very straightforward. If the distance between the centres of the two spheres is less than the sum of the radii of the two spheres, then an intersection has occurred: As the simulation knows the location of the centre of the spheres, we use Pythagoras' theorem to calculate the distance between them, and compare the results to the sum of the radii. So an intersection has occurred if

$$d < r_1 + r_2 \tag{1}$$

where r_1 and r_2 are the sphere radius' and c_1 and c_2 are the sphere positions

$$\begin{aligned} d &= |c_0 - c_1| \\ &= \sqrt{(x2-x1)^2 + (y_2-y_1)^2 + (z_2-z_1)^2} \end{aligned} \tag{2}$$

However, a square root is an expensive thing to compute, so usually the comparison will be between the square of d and the square of the sum of the radii. So an intersection has occurred if:

$$d^2 < (r_1 + r_2)^2 \tag{3}$$

where

$$d = (x2-x1)^2 + (y_2-y_1)^2 + (z_2-z_1)^2 \tag{4}$$

The collision response data is also straightforward to calculate; indeed most of the work has already been done in the detection algorithm. The contact point P is on the vector which connects the centre of the two spheres, which is also the normal vector N, while the penetration distance p is simply the difference between the sum of the radii and the distance between the sphere centres (S1 and S2).

$$\begin{aligned} p &= r_1 + r_2 - d \\ N &= |S_1 - S_2| \\ P &= S_1 = N(r_1 - p) \end{aligned} \tag{5}$$

In C++, the sphere-sphere collision test is written:

Listing 1. Sphere-Sphere Collision Detection

```
1  class Sphere_c
2  {
3  public :
4    Sphere_c ( const Vector3 & p, float r )
5    {
6      m_pos = p;
7      m_radius = r;
8    }
9    Vector3 m_pos ;
10   float m_radius ;
11 };
12 // ↩
                    ----------------------------------------

13 class CollisionData_c
14 {
15 public :
16   Vector3 m_point ;
17   Vector3 m_normal ;
18   float m_penetration ;
19 };
20 // ↩
                    ----------------------------------------

21
22 bool SphereSphereCollision ( const Sphere_c & s0 ,
23 const Sphere_c & s1 ,
```

```
24  CollisionData_c * collisionData = NULL )
25  {
26   const float distSq = LengthSq ( s0. m_pos - s1. m_pos );
27   DBG_ASSERT ( distSq > 0.00001 f );
28
29   const float sumRadius = (s0. m_radius + s1. m_radius );
30
31   if ( distSq < sumRadius * sumRadius )
32   {
33    if ( collisionData )
34    {
35     collisionData -> m_penetration = sumRadius - sqrtf ( ↩
         distSq );
36     collisionData -> m_normal = Normalize ( s0. m_pos -↩
         s1. m_pos );
37     collisionData -> m_point = s0. m_pos - collisionData ↩
         -> m_normal
38     * (s0. m_radius - collisionData -> m_penetration *0.5 ↩
         f );
39    }
40    return true ; // Collision
41   }
42   return false ; // No Collision
43  }
```

In the code the square root is calculated when computing the collision data, even though it was earlier stated that this is expensive computationally. The collision detection algorithm does not use the square root; it compares the squares of the two distances which is much faster. It is only after a collision has been detected that we need to call on the more computationally expensive square root. Most calls to the collision detection routine will result in a false result (i.e. no intersection has occurred); it is only the rare case of a positive result (i.e. an intersection) which then triggers the more expensive collision data calculation.

You may also notice that, in calculating the point of intersection, only half of the penetration distance is used. This is because this algorithm is intended to move both spheres involved in a collision, so each sphere only needs to move half of the overall penetration distance. Remember that the physics engine loops through every object in the simulation, before applying any collision corrections.

4.2 Axis-Aligned Bounding Box

The axis-aligned bounding box (AABB) method is also straightforward. It is typically used as a high level collision test to decide whether it is worthwhile continuing with a more complex test, or to trigger a piece of game logic. Each simulated object is represented as a bounding box aligned with the axes of the world, so each collision object has a position, as well as a height, width and length.

The axes are considered in turn, and if there is an overlap of all three axes then an intersection has occurred. An overlap along a particular axis has happened if the distance between the centres of the two boxes on that axis is less than half the sum of the boxes' lengths along that axis. So an intersection has occurred if all three of these conditions is met:

$$\begin{aligned} |x_2 - x_1| &< 0.5(w_1 + w_2) \\ |y_2 - y_1| &< 0.5(h_1 + h_2) \\ |z_2 - z_1| &< 0.5(l_1 + l_2) \end{aligned} \tag{6}$$

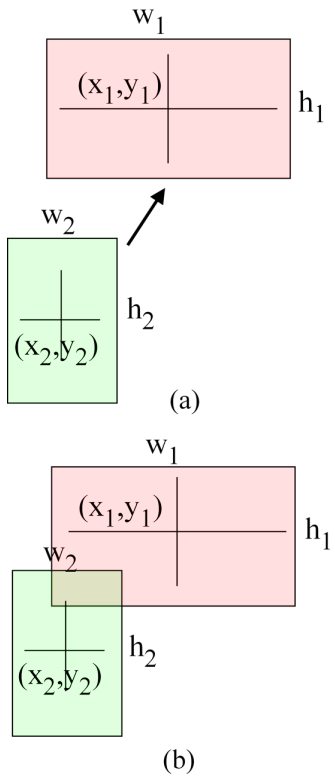

Figure 5. Axis Aligned Bounding Box (AABB) - Detecting and gathering contact information for a AABB collision

Importantly, as soon as one of these checks fails, the algorithm can bail out as there can't possibly be an intersection unless there is overlap in all three axes.

This is a very cheap collision detection algorithm, as the mathematics is very straightforward (only additions and subtractions). However it is very limited - in particular the bounding boxes need to be axis-aligned, so they can not rotate as the object they represent moves around the world. Also the collision response data is not generated by the algorithm. Consequently this algorithm tends to be used only when we need a quick binary decision on whether a collision is likely, before moving on to a more complex collision detection algorithm, or making a high-level game logic decision, such as detecting when the player has entered a new region of the world, or some game logic needs to be triggered by an invisible bounding box.

In C++, the Axis Aligned Bounding Box collision test is written:

```
1  bool AABBCollision ( const Box_c & cube0 ,
2  const Box_c & cube1 )
3  {
4   // Test along the x axis
5   float dist = cube0.pos .x - cube1.pos.x;
6   float sum = ( cube0.halfdims.x + cube1.halfdims.x);
7   // If the dist , is less than the sum , we have an overlap
8   if ( dist <= sum )
9   {
```

```
    dist = cube0.pos.y − cube1.pos.y;
    sum = ( cube0.halfdims.y + cube1.halfdims.y);
    if ( dist <= sum )
    {
      float dist = cube0.pos.z − cube1.pos.z;
      float sum = ( cube0.halfdims.z + cube1.halfdims.z);
      if ( dist <= sum )
      {
        // Overlap in all three axes so there is an intersection
        return TRUE ;
      }
    }
  }
  return FALSE ;
}
```

You may notice that the Box class stores the half dimensions for the height, width and length; this is to avoid multiplying by 0:5 every time the algorithm is used, which is again an efficiency measure.

4.3 Sphere-Plane Collision

Surfaces within the environment are most efficiently simulated as planes in the physics engine. Hence a simple physics simulation of a game would entail representing the game objects as spheres, and the surfaces of the environment (e.g., floors and walls) as planes. We therefore require a method for detecting when a sphere has intersected a plane. You will recall from the practical on frustum culling, that the plane equation can be used to calculate how far a point is from an infinite plane. Obviously if this distance is less than the radius of a sphere, then the sphere intersects the plane.

The plane equation is:

$$Ax + By + Cz + D = 0 \tag{7}$$

where (A,B,C) is the normal to the plane, D is the distance of the plane from the origin, and (x, y, z) is the position of the test point.

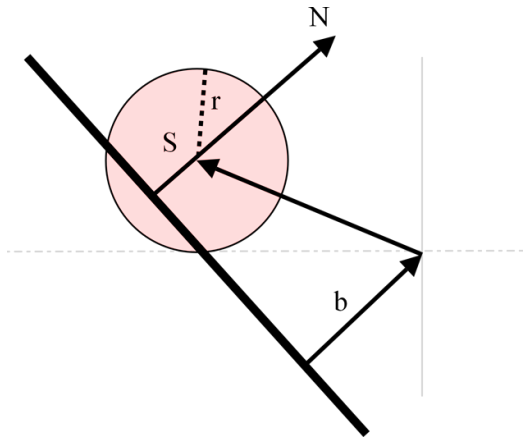

Figure 6. Sphere-Plane - Detecting and gathering contact information for a sphere-plane collision

Consequently, a sphere at position S of radius r, intersects a plane with normal N at distance d from the origin if

$$N \cdot S + d < r \tag{8}$$

where N:S is the dot product of N and S. Note that N is a normal and therefore must be of length 1 (i.e. a unit vector), whereas S is simply the position of the centre of the sphere (ie the vector from the origin to the sphere's centre) and therefore not a unit vector.

The Plane class which you have used during the graphics practical course contains a plane-sphere intersection test. The C code is repeated here:

```
bool Plane :: SphereInPlane ( const Vector3 & position ,
float radius ) const
{
  if ( Vector3::Dot ( position , normal ) + distance <= −↩
       radius )
  {
    return false ;
  }
  return true ;
}
```

Again most of the calculation of the collision response data has already been carried out. The penetration p is simply the difference between the radius and the distance between the sphere centre and the plane. The collision normal is the normal of the plane. The contact point P is calculated by taking the sphere position, and adding a vector along the direction of the normal equal to the distance between the sphere centre and the plane. Mathematically:

$$\begin{aligned} p &= r - (N \cdot S + d) \\ P &= S - N(r - p) \end{aligned} \tag{9}$$

So expanding the SphereInPlane method to return the collision response data:

```
bool Plane::SphereInPlane ( const Vector3 & position ,
          float radius ,
          CollisionData_c * collisionData = NULL ) const
{
  float separation = Vector3::Dot ( position , normal ) + ↩
       distance;
  if ( separation <= −radius )
  {
    return false ;
  }
  if ( collisionData )
  {
    collisionData−>m_penetration = radius − separation ;
    collisionData−>m_normal = normal ;
    collisionData−>m_point = s0.m_pos − normal * ↩
       separation ;
  }
  return true ;
}
```

The plane equation, and the sample code is based on testing for intersection with an infinite plane. Of course, even a simple game environment can't be represented exclusively by infinite planes. The following practical will introduce techniques which can be used to test for collisions between less generic shapes.

5. Implementation

The aim of this practical session is to expand your physics engine to detect intersections between simple geometric shapes - namely spheres and planes. We will construct a set of algorithms which carry out the required tests for sphere-sphere and sphere-plane collisions. To demonstrate that the collision tests are working, we will add functionality to the project which tests for when a launched sphere hits a floor or wall, and halts its motion. We will further test for when a sphere hits another sphere and cause the two spheres to vanish.

6. Summary

The collision detection routines are two-tiered - broad-phase and narrow-phase. In this practical, we have introduced the concept of broad-phase collision detection that works in conjunction with the narrow-phase system. Once a collision is detected, the second step is to respond to that by moving the intersecting objects apart. This practical has dealt with the quick collision detection routines which are typically employed during broad-phase.

Acknowledgements

We would like to thank all the reviewers for taking time out of their busy schedules to provide valuable and constructive feedback to make this article more concise, informative, and correct. However, we would be pleased to hear your views on the following:

- Is the article clear to follow?
- Are the examples and tasks achievable?
- Do you understand the objects?
- Did we missed anything?
- Any surprises?

The practicals provide a basic introduction for getting started with cloth effects. So if you can provide any advice, tips, or hints during from your own exploration of simulation development, that you think would be indispensable for a student's learning and understanding, please don't hesitate to contact us so that we can make amendments and incorporate them into future practicals.

Recommended Reading

Code Complete: A Practical Handbook of Software Construction, Steve McConnell, ISBN: 978-0735619678

Clean Code: A Handbook of Agile Software Craftsmanship, Robert C. Martin, ISBN: 978-0132350884

Game Inverse Kinematics: A Practical Introduction (2nd Edition) Kenwright. ISBN: 979-8670628204

Kinematics and Dynamics Paperback. Kenwright. ISBN: 978-1539595496

Game Collision Detection: A Practical Introduction (Paperback). Kenwright. ISBN: 978-1511964104

Game C++ Programming: A Practical Introduction (Paperback). Kenwright. ISBN: 978-1516838165

Computational Game Dynamics: Principles and Practice (Paperback). Kenwright. ISBN: 978-1501018398

Game Physics: A Practical Introduction (Paperback). Kenwright. ISBN: 978-1471033971

Game Animation Techniques: A Practical Introduction (Paperback). Kenwright. ISBN: 978-1523210688