

Collision Detection and Resolution

Ming-Hwa Wang, Ph.D.

COEN 396 Interactive Multimedia and Game Programming

Department of Computer Engineering

Santa Clara University

Overview

- collision detection and resolution can implement solidness of objects in physical simulation
- collision detection will determine if and when two objects collide, and collision resolution will figure out where each object should be once a collision is detected (calculating how objects move after the collision is the job of the physics)

Collision Detection

some objects can move very fast, and some objects can have very complicated geometry; besides, collision detection is very costly because every object should be tested against every other object for a possible collision $O(n^2)$

- overlap testing
 - overlap testing detects whether a collision has already occurred; if two objects overlap, they are in collision; it is a discrete test and is the most common techniques, yet exhibits the most error
 - overlap testing is actually a containment problem; it is test if any part of an object is inside any part of another object; it is simple with sphere and boxes, but it can be difficult with polygons; the volume of polygon objects must be approximated by simpler geometrics shapes to make the problem tractable in real time
 - results:
 - find the time the collision took place by bisection
 - the collision normal vector (for computing the collision response and preventing objects from further interpenetrating)
 - limitations of overlap testing: the speed of the fastest object in the scene multiplied by the time step must be less than the size of the smallest collidable object in the scene; either need to keep objects from moving too fast, or simulation step size need to be reduced to satisfy the constraint; both of those options might be undesirable
- intersection testing
 - intersection testing predicts if a collision will occur in the future; the simulation can be carefully moved forward to the time of impact, often more accurately and efficiently than overlap testing
 - intersection testing can be viewed as a visibility problem; test the geometry of an object swept in the direction of travel against other swept/extruded geometry
 - sphere-sphere collision: the sphere of radius r_P moving from the point P_1 at time $t = 0$ to the point P_2 at time $t = 1$ collides at time t with another sphere of radius r_Q moving from the point Q_1 to Q_2 ; $t = \frac{-(\mathbf{A} \cdot \mathbf{B}) - ((\mathbf{A} \cdot \mathbf{B})^2 - |\mathbf{B}|^2(|\mathbf{A}|^2 - (r_P + r_Q)^2))^{1/2}}{|\mathbf{B}|^2}$, where $\mathbf{A} = P_1$

- Q_1 and $\mathbf{B} = (P_2 - P_1) - (Q_2 - Q_1)$; the smallest distance ever separating the centers of the two spheres is $d^2 = |\mathbf{A}|^2 - (\mathbf{A} \cdot \mathbf{B})^2 / |\mathbf{B}|^2$

- limitations:
 - prediction methods aren't very compatible with networked game because future predications rely on knowing the exact state of world at the present time; due to packet latency, the current state is not always coherent
 - intersection testing assumes a constant velocity and zero acceleration over the simulation step

Dealing with Complexity

- simplified geometry: if a complex object can be roughly approximated with a simpler shape, testing will be cheaper (but some parts of the object might come into collision without being able to detect it)
 - Minkowski sum: the Minkowski sum of object X and another object Y can be created by sweeping the origin of X over all points within Y ; $X \oplus Y = \{A + B: A \in X \text{ and } B \in Y\}$
 - determine overlap by taking the origin of the sphere and testing if it is within the new volume
 - to perform the intersection testing, the point becomes a line, going from the center of the sphere at time t_0 at the center of the sphere at time t_1 ; we then test this line to see if it intersects with the new volume
 - bounding volumes: a simple geometric shape that fully encapsulates an object
 - if there is no collision with the bounding volume (cheaper to test), it is known that there is no collision with the object; use bounding volumes when approximate collision detection suffices, or use increasingly complex bounding volumes when accuracy matters
 - the simplest bounding volume is a sphere (no orientation); two spheres overlap each other if the distance between their centers is less than the sum of their radii; in the case of visibility, an extruded sphere is a capsule, so each of these objects can be tested against the other extruded objects to determine overlap
 - the next most common bounding volume is a box
 - axis-aligned bounding box (AABB): the faces line up with the 3 axes; usually results in a loose fit around an object, but the resulting tests against the box are simplified and cheaper to perform
 - oriented bounding box (OBB): a tighter fitting box that is oriented in a manner to best encapsulate the object
- achieving linear complexity instead of $O(n^2)$
 - the world is partitioned with a simple grid, each object must only be tested against objects in the same or neighboring grid cells

- if there are N collidable objects, a 2D grid will need to be at least $N^{1/2} \times N^{1/2}$ in size, and a 3D grid will need to be at least $N^{1/3} \times N^{1/3} \times N^{1/3}$; this will on average result in one object per grid cell, which should support linear time complexity (on average)
- problems:
 - if objects vary in size and don't fit inside a single grid; either increase the cell size or more grid cells "further out" must be tested
 - if all of the objects move into the same grid cell, then the time complexity has reverted back to $O(n^2)$
- ❖ plane sweep: leverages the temporal coherence of objects to roughly stay in the same location from frame to frame, thus reducing the problem to linear; any objects that have overlapping bounds in all axes should be examined more closely for a collision; the time consuming aspect of this algorithm is collecting and then sorting (in $O(n \log n)$) the bounds on each axis every frame; since objects display coherence from frame to frame, we can sort each axis list once, and then use insertion sort to quickly repair any of the bounds that became slightly out of order; the result is nearly linear time

Terrain Collision Detection

- ✚ a flat plane as ground: the simplest type of terrain; if the character is standing, the bottom of each foot should rest at the terrain's y coordinate; if the character jumps into the air and falls to the ground, the character will hit the ground when his foot attempts to go below the terrain; if it does go below, the foot is in collision with the terrain and should be placed at the terrain height
- ✚ a polygon mesh (in x- and z-axes) with height field (in y-axis): treat the terrain as a 2D planar mesh and use the nature of the 2D uniform grid to locate the exact rectangle cell that contains the point Q (represent the point of the object that should rest on the terrain); once we have identified the rectangle cell, we must determine which triangle contains point Q by comparing with the dividing line (which divide the rectangle into 2 triangles)
 - ❖ if $Q_z > Q_x$, upper left triangle
 - ❖ if $Q_z = Q_x$, lower right triangle
 - ❖ if $Q_z > 1 - Q_x$, upper right triangle
 - ❖ if $Q_z = 1 - Q_x$, lower right triangle
$$Q_y = (-N_x Q_x - N_z Q_z + \mathbf{N} \cdot \mathbf{P}_0) / N_y$$
, where \mathbf{P}_0 is one of the triangle's vertices
- ✚ triangulated irregular networks (TINs): if the terrain is a non-uniform polygonal mesh, use spatial partition techniques (e.g., octrees) to narrow down what part of the terrain we're interested in, producing a subset of candidate polygons, then test each polygon until one is found that contain the point Q
 - ❖ barycentric coordinate: represent point Q in terms of a weighted sum of each triangle vertex; the 3 weights must add up to 1.0, so $w_0 = 1 - w_1 - w_2$

$$\begin{array}{|c|} \hline w_1 \\ \hline \end{array} = \frac{1}{(V_1^2 V_2^2 - (\mathbf{V}_1 \cdot \mathbf{V}_2)^2)} \begin{array}{|c|} \hline V_2^2 \\ \hline \end{array} \begin{array}{|c|} \hline -\mathbf{V}_1 \cdot \mathbf{V}_2 \\ \hline \end{array} \begin{array}{|c|} \hline \mathbf{S} \cdot \mathbf{V}_1 \\ \hline \end{array}$$

$$\begin{array}{|c|} \hline w_2 \\ \hline \end{array} = \frac{1}{(V_1^2 V_2^2 - (\mathbf{V}_1 \cdot \mathbf{V}_2)^2)} \begin{array}{|c|} \hline -\mathbf{V}_1 \cdot \mathbf{V}_2 \\ \hline \end{array} \begin{array}{|c|} \hline V_1^2 \\ \hline \end{array} \begin{array}{|c|} \hline \mathbf{S} \cdot \mathbf{V}_2 \\ \hline \end{array}$$

where $\mathbf{S} = \mathbf{Q} - \mathbf{P}_0$, $\mathbf{V}_1 = \mathbf{P}_1 - \mathbf{P}_0$, and $\mathbf{V}_2 = \mathbf{P}_2 - \mathbf{P}_0$

if the texture coordinates (s_0, t_0) , (s_1, t_1) , and (s_2, t_2) are associated with vertices \mathbf{P}_0 , \mathbf{P}_1 , and \mathbf{P}_2 , the texture coordinates (s, t) at the point Q are given by $s = w_0 s_0 + w_1 s_1 + w_2 s_2$ and $t = w_0 t_0 + w_1 t_1 + w_2 t_2$

Collision Resolution

- ✚ the procedure for collision resolution has 3 parts:
 - ❖ prologue callback function: if the collision should not affect the position or trajectories of the objects, the function returns false; the prologue may also trigger other events, e.g., sound effect, sending collision notification, etc.
 - ❖ collision: the objects will be placed at the point of impact and new velocity will be assigned using physics or some other decision logic
 - ❖ epilogue: any post-collision affects must be propagated; by sending a collision epilogue event to each object, with the object determining what effects to trigger
- ✚ resolving overlap testing: 4 steps in resolving the collision:
 - ❖ extract the collision normal: constructed by using the two closest points on each surface (with spheres, the collision normal is the difference between the centers of each sphere at the point of impact)
 - ❖ extract the penetration depth
 - ❖ move the tow objects apart to a penetration depth of zero if needed
 - ❖ compute the new velocities using Newtonian physics
- ✚ resolving intersection testing: simpler because objects never actually penetrate; without overlap, there is no need to detect the penetration depth and move the object apart