Game Physics

Chris GauthierDickey
CIS 399 - Intro to Game Programming
Why Physics?

• Humans are experts and knowing what to expect. The use of physics helps our games behave the way our brains think they should!
• Because it’s fun!
Collision Detection

• First we must side-track into collision detection…

• Physics isn’t as meaningful without collision detection!

• We divide our algorithm into two phases:
  – Broad phase: test the intersection of objects at a high level
  – Narrow phase: exact collision of polygons
Broad phase

• Naïve approach:
  – Compare all objects each frame:
    • All-pairs algorithm: $\Theta((n^2-n)/2)$
    • Not bad for small number of objects
    • Easy to implement
    • Compare bounding box each frame (for broad phase)
      – Box can be a sphere, axis-aligned bounding box, object aligned bounding box, or something else
Broad Phase continued

• Spheres are easy to test, AABB are also easy to test, but change with object orientation, OBB can be pre-calculated

• However, scalability often comes through hierarchy
  – Build a hierarchy of bounding boxes around the object, testing the topmost level first, and then lower levels for more accurate intersections
Ogre Collision Detection

• Ogre implements collision detection at the SceneManager level
• Different SceneManagers optimize their collision detection routines
• We query the SceneManager to see if anything has collided
• Ogre only implements broad-phase detection
// this returns the query object
IntersectionSceneQuery* iQuery =
    mSceneMgr->createIntersectionQuery();

// this executes the query
IntersectionSceneQueryResult& queryResult = iQuery->execute();

// You can also retrieve the last query without recalculating
queryResult = iQuery->getLastResults();

// to destroy the query call:
mSceneMgr->destroyQuery(intersectionQuery);
Pairs of results

- This code gets the first element of the list then gets the first item in the pair, and then calls getName() on it. We can also access second, which is the second item in the pair. The result is a list of pairs, so we can do this:

  String name = queryResult.movables2movables.begin()->first->getName();

- Note that the pair is the pair of items which may have collided.
Physics

• Thanks to Newton, we can simulate simple physics in our game.
• We’re really interested in a very small part of physics: dynamics.
• Full blown dynamics are quite difficult, but we can simplify our model to add realism to the game.
Simplifying Assumptions

• We consider only objects with uniform mass
• We assume we have a center of mass already defined
• We don’t consider rotational effects (yet)
  – These require more complex collision detection than what Ogre gives us
Newton’s Second Law

• $F = ma$
  
  – Force equals mass times acceleration.
  
  – To calculate the position of an object, we consider all forces on the object and determine its acceleration ($a = F/m$). We then can integrate to find the velocity. Integrate again to find its position.
Vector Calculus

- Vector Calculus works by considering each element of the vector, instead of a single scalar value.
- This means that when we calculate forces, we just use vectors, instead of scalars. Further, the center of mass is a vector, not a scalar.
Adding Physics to the Game

• We need a center of mass for the object
• We need to keep track of the direction the object is moving, separate from its orientation
  – Orientation just determines which way is ‘forward’ and which direction the engines ‘push’
  – Direction is the sum of the forces acting on it
How do we update()?

• Now, instead of updating our position by our orientation, we update it by our direction.

• Each tick, we calculate the acceleration:
  – $a = \frac{F}{M}$

• We then integrate to find velocity

• Integrate again to find position
Euler method

• Instead of integrating, we use the Euler method:
  – Recall that $a = \frac{dv}{dt}$
  – So $F = m \frac{dv}{dt}$
  – So $\frac{dv}{dt} = \frac{F}{m}$
  – $dv = \left(\frac{F}{m}\right)dt$
  – Remember that in integrating we take the number of slices towards the limit of infinity
Approximation

• Thus we can approximate by imagining small time steps:
  \[ \Delta v = (F/m) \Delta t \]
  – Therefore:
    • \[ v_{t+\Delta t} = v_t + (F/m) \Delta t \]
  – and
    • \[ s_{t+\Delta t} = s_t + \Delta t(v_{t+\Delta t}) \]
More Details

• Euler method is really an approximation.

• Taylor theorem tells us we can approximate the value of a function at some point by knowing some initial conditions and its derivatives at another point.

  – We expand \( y(x) \) to:

\[
y(x + dx) = y(x) + (dx) y'(x) + \left( \frac{dx^2}{2!} \right) y''(x) + \left( \frac{dx^3}{3!} \right) y'''(x) + \ldots
\]
Taylor Expansion

• Note that the more derivatives we find, the more accurate our approximation is.
• Euler method only takes the first derivative:
  \[ v_{t+\Delta t} = v_t + \frac{F}{m} \Delta t \]
• The obvious question:
  – Can we gain more accuracy by expanding further?
More Accuracy

• Euler method loses accuracy because we chop off the other terms after the first derivative.

• Other methods further calculate the expansion:
  – Improved Euler method takes the next derivative (expansion)
  – Runge-Kutta method takes 4 expansions
Physics for Our Game

• The main reason we need more accuracy is if the derivative changes with respect to its inputs
  – This occurs, for example, if we have a drag coefficient

• Euler method is considered unstable if \( \Delta t \) is too large.

• However, we're modeling space craft, so we can be fairly accurate with the basic Euler method.