Collisions

CS134

Chris Pollett

Oct 11, 2004.

Introduction

- The cCritter::collide method
- Collision handling
- Colliding spheres

The cCritter::collide method

- A cCritter has a virtual BOOL collide(cCritter *pcritter) method.
- This checks if pcritter is touching the caller and returns TRUE/FALSE accordingly.
- In the TRUE case, one also executes code to handle the result of the collision.

Some points about collide

- Want to keep physics symmetric. That is, we only want to call a collide method once for each pair of critters.
- Need a mechanism to decide if we are going to call pcritteri->collide(pcritterj) or pcritterj->collide(pcritteri)

Default collide

- The cCritter::collide implements (a) the law of conservation of momentum, (b) the law of conservation of energy, and (c) the requirement that two objects can’t be in the same place at the same time (cCritter’s behave like fermions)
- Standard implementation assumes critters behave like spheres.

Other implementations

- cCritterWall’s collide is different since long narrow walls are not like disks/spheres. The code for this is slightly complicated. Will not discuss today.
- cCritterBullets damage other object then explode.

Example of overriding collide

```
BOOL cCritterChild::collide(cCritter *pcritter)
{
    BOOL collided = cCritter::collide(pcritter);
    if(collided)
    {
    //do something
    }
    return collided;
}
```

More examples of overriding collide

```
BOOL cCritterArmedPlayer::collide(cCritter
   *pcritter)
{
    BOOL collided = cCritter::collide(pcritter);
    if(collided && _sensitive && !pcritter-
       >IsKindOf(RUNTIME_CLASS(cCritterWall)))
        damage(1);
    return collided;
}
```

Another Example

```
BOOL cCritterBasket::collide(cCritter *pcritter)
{
    if(contains(pcritter))
    {
        pcritter->die();
        return TRUE;
    }
    else
        return FALSE;
}
```

cCritterBullet::collide() Rough Sketch

- Makes use of cCritterBullet::isTarget to check:
  - If the pcritter is one of bullet’s targets and bullet’s touching it, damage pcritter and die.
  - If pcritter is a target and you’re not touching it, do nothing
  - If pcritter isn’t a target, collide with it normally

Collision Handling Overview

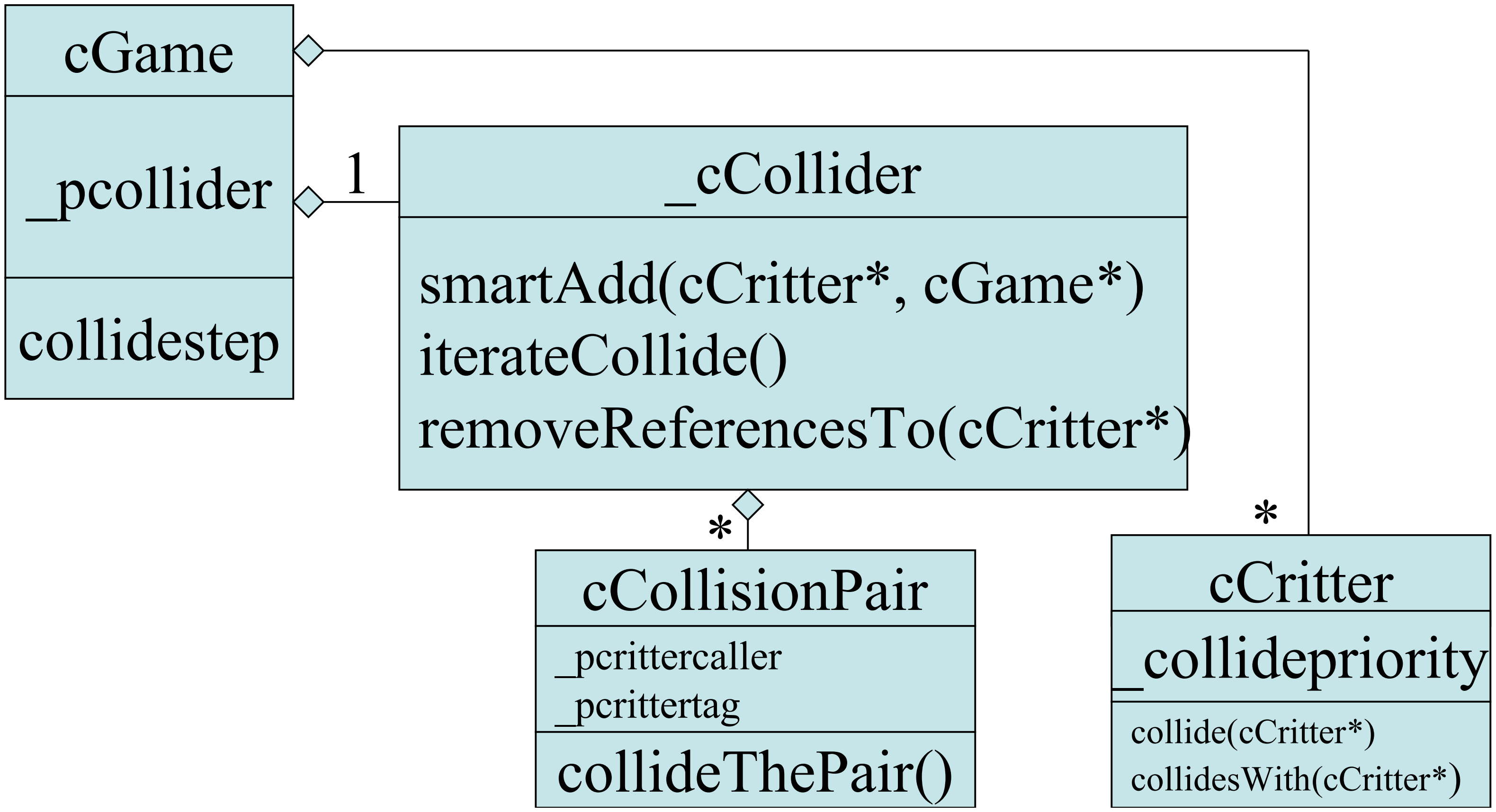
- Have a utility class for holding pairs of critters, this class says which critter has priority to control behavior of collision
- Maintain a collection of all pairs of critters which can collide
- For each game step, iterate through this collection of pairs
- For each candidate pair, collide the critters by letting the higher priority critter call its collide method on the other.

The N-Squared Problem

- If had N critters and did nothing clever, then checking for all possible collisions would involve N^2 checks.
- Don’t need to check if a critter collides with itself, so above architecture reduces the problem to N(N-1)/2 checks
- Can further try to restrict irrelevant possibilities (Ex: wall to wall collisions).
- Pop works well when there is less than 500 or so pairs to check.

A collision-handling architecture

- Here are the relevant collision classes:



How the collide classes work together

- cCollisionPair has two cCritter* members called _pcrittercaller and _pcritterarg. cCollider::collideThePair calls _pcrittercaller->collide(_pcritterarg).
- cCollider holds a collection of cColliderPair objects. A cGame has a cCollider _pcollider object. When you add a critter to the game, cCollider::smartAdd looks at all possible pairs that include the new critter and creates a cColliderPair object for the relevant pair. When a critter, p, is deleted _pcollider->removeReferencesTo(p) is called.
- In each update, cGame::collideStep calls _pcollider->iterateCollide() which in turn calls collideThePair for each cColliderPair.

Collision priority

How does smartAdd know what the relevant pairs are? And how does it know which member of the pair should have which role?

Answer: Each cCritter has a Real _collidePriority. Given two critters the one with the higher priority will be set up to have its collide called. Each cCritter also has a collidesWith(cCritter *pcritterOther) method which smartAdd uses to figure out if the given critter can collide with pcritterOther.

More on Collision Priority

- Order of priority of some common critter types: Walls, Bullets, Player, Other critters.
- collidesWith returns an int. As will probably want to override, some useful return codes are: cCollider::DONTCOLLIDE, cCollider::COLLIDEASCALLER, cCollider::COLLIDEASARG, cCollider::COLLIDEEITHERWAY
- Usually, collidesWith compares the _collidepriority values and returns the appropriate code.

N^3 Issues

- cCollider stores cCollosionPair’s as lists.
- To delete a pair from a cCollider involves deleting roughly N objects from a collection of N^2/2 objects.
- If used an array would and did this in a silly fashion each time delete an object would need to move everything over by 1.
- So would have O(N^3) work. Agh!!

Colliding Spheres

- We collide object using physics.
- Conservation of momentum means:

  M_1*V_{1out} + M_2*V_2out} = M_1*V_{1in} +M2*V_{2In}
- Notice the velocities are **vectors**, so have 3 components.
- We are also assuming energy is conserved (elastic collisions). This means:
- 1/2*(M_1||V_{1out}||^2 +M_2||V_{2out}||^2) = 1/2*(M_1||V_{1in}||^2 +M_2||V_{2in}||^2)

More on Colliding spheres

- Values for velocities that satisfy the first constraint give a line. Values for velocities which satisfy the second constraint give an ellipse.
- So the intersection gives two solutions, a pre and post collision solution.
- Book works out formula of solutions (massratio := M_2/M_1):
  - V_{1out} = [(1-massratio)*V_{1in} + 2*massratio*V_2in]/(1+massratio)
  - V_{2out} = [2*V_{1in}+(massratio - 1)*V_{2in}]/(1+massratio)

Making Fermions

- Code has been added to cCritter::collide so as to move pairs of critters apart along the line connecting their centers so they don’t overlap after a collision.