Simulating Physics, Generating 2D terrains

CS134

Chris Pollett

Sep. 20, 2004

Introduction

- Physics
- Parallelism
- The Laws of Motion
- Force and acceleration
- Implementing Forces
- Preserving Physics
- Terrain Generation

Physics

- Games are more successful if objects move at least loosely like in the real world. Motion in the real world has been well-studied by physicists.

- As far as computer simulation goes there are three important characteristics to consider with regard to physics: (a) parallelism -- many things happen at same time, (b)homogeneity -- assume physics on Mars just like on Earth, (c) local -- things far apart do not influence each other much.

Parallelism

- Pop does not use threads to simulate many things happening at same time. Reason: why threads grant priority and real world parallelism are different.
- Objects are store in arrays and for loop used to cycle through these arrays to compute updates.
- Objects (mainly critters) in Pop have two functions used in this process update() and move(dt).

Why update and move?

- Update is supposed to look at all the forces, collisions,etc and figure out what will be new aggregate force vector on the object.
- Move actually moves object to a new position
- We calculate each object's update first
- Then we move each object
- This avoids issues of whose value getting changed first affecting outcomes of events.
- Gives actions the appearance of occurring in parallel.

The Laws of Motion

- The laws of motion are homogeneous -- the same everywhere.
- This is implemented in Pop by having all critters derive from the same base class and by making move(dt) non-virtual.
- Thus, after all the forces have been calculated on the object in an update, the actual motion can then be determined by Newton's Laws.

Motion Refresher

- position -- where object is
- velocity -- rate of change of position
- acceleration -- rate of change in velocity
- Force = mass * acceleration. Force on object is vector sum of individual forces applied.

So…

acceleration = Force/mass

velocity = velocity + dt*acceleration

position = position + dt * velocity

Pop Mass

- Critters each have _mass, _density, and _radius fields.
- Internally, critters ensure that _mass is proportional to the product of the _density*_radius^3.
- We'll talk latter about changing density. Default is 1.

Force and Acceleration

- Each critter has a number of cForce *_pforce objects on it.
- To make a critter feel a force we call _pforce->force(this). I.e., we're using the Strategy pattern. This returns a numerical vector which we can think of as the quantity of this particular force applied to this critter at this time.
- We store all of these forces in an MFC CTypedPtrArray(CObArray, cForce*) _forceArray.
- To determine all the forces acting on a critter update calls feelforce()

feelforce

```
void cCritter::feelforce()
{
	cVector forcesum;
	for(int i=0; i<_forcearray.GetSize(); i++ )
	{
		forcesum += _forcearray.GetAt(i)->force(this);
	}
	_acceleration = forcesum/mass();
}
```

Can override this method. Might do if want to simulate steering forces.

There is also a feellistener() method which figures out keyboard inputs' effects on velocity and acceleration

Implementing Forces

- cForce::force() just returns the zero vector.
- Other forces that inherit from cForce are:

  cForceGravity, cForceDrag (has subclass: cForceVortex), cForceObject (cForceObjectSeek, cForceObjectSpringRod), cForceClass (cForceClassEvade, cForceEvadeBullet)
- Let's look at how some of these might be implemented.

Gravity

- F = G *m1*m2/ D^2 where G is a constant. When m2 is Earth and when D doesn't change much (I.e., stays close to radius of earth) G*m2*D^2 is constant and we get:
- F= g*m.
- So to specify g class cForceGravity has a _pulldirection and an _intensity.

cForceGravity::force

```
cVector cForceGravity::force(cCritter
   *pCritter)
{
    return _intensity*pcritter->mass() *
       _pulldirection;
}
```

Drag

This is a force in a resistive media (water/air). _windvector is used to hold the wind velocity/current. Again we have an _intensity. The force method looks like:

```
cVector cForceDrag::force(cCritter *pCritter)
{
    Real area = pcritter->radius()*pcritter->radius();
    return  cVector(area*_intensity*(_windvector - pcritter->velocity()));
}
```

cForceObject

- Used to model a critter's reactions to some other critter.
- Has a cCritter *_pnode reference
- Can use this reference to compute distance between two critter and use to model spring forces. This is done in cForceObjectSpringRod.
- Can also use to get one critter to chase/evade another critter. this is done in cForceObjectSeek and cForceClassEvade.
- If want to reference more than one critter than use cForceClass.
- Another interesting force is cForceWaypoint. (see 7.8 in book).

Preserving Physics

- You should be able to make any change you want to the critters motion without having to override move(dt). For example,
  - Can create steady cForces to simulate constant acceleration
  - to apply an impulse change the critter's velocity using setVelocity.
  - to teleport the critter can use moveTo()

Terrain Generation

- Algorithms depend on types of terrain.
- Today will consider random dungeon generation
- Random topography.

Random dungeons

- Imagine world lives on some flat playing field with a given x and y size.
- Pick several random x values (vertical lines) less than the x size.
- Pick several random y values (horizontal line) less than the y size.
- These line induce a set of rectangles on the world. Can choose among these, randomly discarding the ones that are too small. These are the rooms.

More Random Dungeons

- For each room choose at random whether or not a given wall has a door. Then choose at random along the wall where the door is.
- Starting from one room. Connect an as yet non connected room to those already connected.
- To do this view the corners of each room as well as the doors as vertices in a graph. Let there be an edge between any two vertices if there is not a room in-between.
- Using this graph and path finding algorithms connect doors.

Random topography

- Again imagine world as a having some fixed x and y extents.
- Pick some random point on this world as well as how high they are.
- Now iteratively erode these spikes unto neighboring points.
- Good for generating volcano like topography.
- Can also use random line techniques for more ``plate tectonic'' like scenery.