Chapter 13. Flocking Boids

Flocking is a computer model for the coordinated motion of groups (or flocks) of entities called boids. Flocking represents typical group movement, as seen in bird flocks and fish schools, as combinations of simple steering behaviours for individual boids based on the position and velocities of nearby flockmates.

Although individual flocking behaviours (sometimes called rules) are quite simple, they combine together to give boids and flocks very interesting overall behaviours, which would be extremely complicated to program explicitly.

Flocking is often grouped with *Artificial Life* algorithms because of its use of *emergence*: complex global behaviours arise from the interaction of simple local rules. A crucial part of this complexity is its unpredictability over time: a boid 'flying' in a particular direction may be doing something completely different in a few minutes time.

Flocking is useful for many types of games where groups of things must move in complex, coordinated ways. For instance, this applies to animals, monsters, soldiers, and crowd scenes. Flocking is used in games such as *Unreal* (Epic), *Half-Life* (Sierra), and *Enemy Nations* (Windward Studios).

The Flocking3D application is shown in Figure 1. It involves the interaction of two different groups of boids: the yellow flock are the predators, the orange ones the prey. Over time, the boids in the orange flock are slowly eaten, although they try their best to avoid it. Both flocks must avoid obstacles and stay within the bounds of the scene.

The left-hand image shows an early stage in the system when the predators are chasing prey. The right-hand image is after the prey have all been eaten, and the predators are flocking together more.

The coding was carried out by the author and one of his students, Miss Sirinart Sakarin.

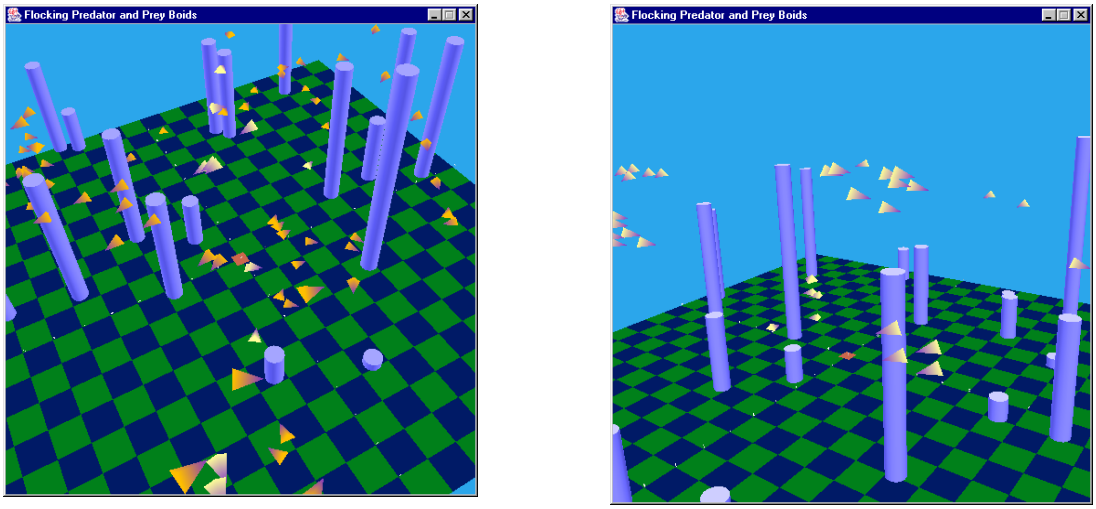

Figure 1. Predator and Prey Flocks in Flocking3D.

The Java and Java 3D techniques illustrated in Flocking3D include:

- the use of inheritance to define a predator and prey as subclasses of a Boid class;
- the use of inheritance to define the predator's and prey's behaviours as subclasses of a FlockingBehavior class;
- the use of synchronized updates to the ArrayList of boids;
- the detaching of boid nodes from the scene graph (when they are 'eaten');
- the creation of a boid shape using an IndexedTriangleArray.

1. Background on Flocking

Flocking was first proposed by Craig Reynolds in his paper:

> "Flocks, Herd, and Schools: A Distributed Behavioral Model"
> C.W. Reynolds, *Computer Graphics*, 21(4), SIGGRAPH'87, pp.25-34.

The basic flocking model consists of three simple steering behaviours:

- *Separation*: steer to avoid crowding local flockmates.
- *Alignment*: steer towards the average heading of local flockmates.
- *Cohesion*: steer to move towards the average position of local flockmates.

These rules are visualised in Figure 2.

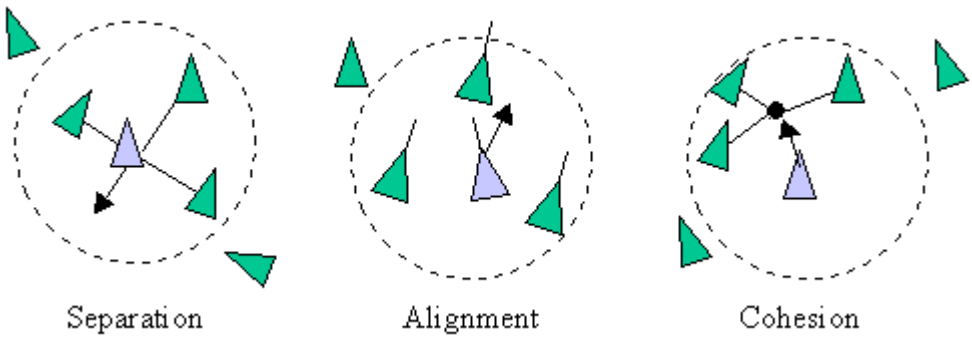

Figure 2. Reynolds' Steering Rules.

The circles in Figure 2 reflect the meaning of 'local flockmates'-- those boids within a certain distance of the boid under consideration.

A more elaborate notion of neighbourhood limits it to the space in front of the boid's current forward direction, which more accurately reflects how real-world flock members interact (see Figure 3).

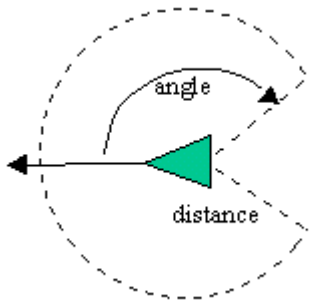

Figure 3. Boid Neighbourhood Based on Distance and Angle.

Over the years, and even in Reynolds' original work, many variants of the basic rules and additional rules have been suggested, including obstacle avoidance and goal seeking.

More Information

The best place for online information about flocking is Craig Reynolds' page (http://www.red3d.com/cwr/boids/). It contains hundreds of links to relevant information, including flocking in games, virtual reality, computer graphics, robotics, art, and artificial life. There is a special section on software using flocking, and a separate page of applets (http://www.red3d.com/cwr/boids/applet/).

Conrad Parker has put together a very clear explanation of boid algorithms with pseudocode examples (http://www.vergenet.net/~conrad/boids/pseudocode.html). He describes Reynolds' steering rules, and additional techniques for goal setting, speed limiting, keeping the flock inside a bounded volume, perching, and flock scattering. His pseudocode was a major influence on the design of our boids steering rules.

Steven Woodcock has written two articles about flocking for *Games Programming Gems* Volume 1 and 2:

- "Flocking: A Simple Technique for Simulating Group Behavior", S. Woodcock, *Game Programming Gems*, Charles River Media, 2000, Section 3.7, pp.305-318.
- "Flocking with Teeth: Predators and Prey", S. Woodcock, *Game Programming Gems II*, Charles River Media, 2001, Section 3.11, pp.330-336.

The first describes Reynolds' basic steering rules, while the second introduces predators and prey, and static obstacles. Both articles come with C++ source code.

2. Flocking3D

As mentioned above, Flocking3D creates predator and prey boids. Each boid behaviour includes:

- Reynolds' steering rules for separation, alignment, and cohesion;
- perching on the ground occasionally;
- avoiding static obstacles;
- staying within the scene volume;
- a maximum speed (although the prey boids have a higher maximum than the predators);
- when a predator is hungry, it chases prey boids. If it gets close enough to a prey boid, it is 'eaten', and removed from the scene graph;
- prey boids try to avoid predators.

UML Diagrams for Flocking3D

Figure 4 shows the UML diagrams for all the classes in the Flocking3D application. Only the class names are shown, and superclasses that are a standard part of Java or Java 3D (e.g. JPanel, Shape3D) are omitted.

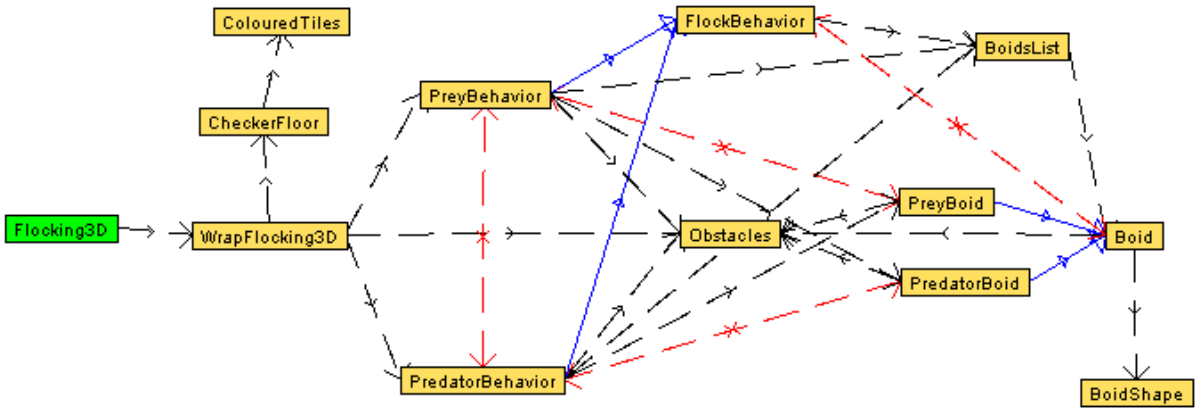

Figure 4. The Flocking3D Classes.

Flocking3D is the top-level JFrame for the application.

WrapFlocking3D creates the 3D world, and is similar to earlier 'Wrap' classes. It creates the predator and prey behaviours, and the obstacles.

PreyBehavior and PredatorBehavior can be thought of as flock managers: they create the boids which make up their flock, and handle rule evaluation at run-time. They are subclasses of the FlockBehavior class.

PreyBoid and PredatorBoid represent individual boids, and are subclasses of the Boid class. BoidShape is a subclass of Shape3D and holds the on-screen boid shape.

CheckerFloor and ColouredTiles are the same classes as in previous examples.

3. WrapFlocking3D

The flocking code is located in addFlockingBoids().

```
private void addFlockingBoids(int numPreds, int numPrey,
                                    int numObs)
{ // create obstacles
  Obstacles obs = new Obstacles(numObs);
  sceneBG.addChild( obs.getObsBG() );  // add obstacles to scene

  // make the predator manager
  PredatorBehavior predBeh = new PredatorBehavior(numPreds, obs);
  predBeh.setSchedulingBounds(bounds);
  sceneBG.addChild( predBeh.getBoidsBG() );  // add preds to scene

  // make the prey manager
  PreyBehavior preyBeh = new PreyBehavior(numPrey, obs);
  preyBeh.setSchedulingBounds(bounds);
  sceneBG.addChild( preyBeh.getBoidsBG() );  // add prey to scene

  // tell behaviours about each other
  predBeh.setPreyBeh( preyBeh );
  preyBeh.setPredBeh( predBeh );

} // end of addFlockingBoids()
```

The number of predators, prey, and obstacles are read from the command line in Flocking3D.java, or assigned default values.

The two behaviours are passed references to each other so that a steering rule can consider neighbouring boids of a different type.

4. Obstacles

The Obstacles class creates a series of blue cylinders placed at random places around the XZ plane. The cylinders have a fixed radius, but their heights can vary between 0 and MAX_HEIGHT (8.0f). A cylinder is positioned with a TransformGroup, and then added to a BranchGroup for all the cylinders; this BranchGroup can be retrieved by calling getObsBG().

At the same time that a cylinder is being created, a BoundingBox is also calculated:

```
height = (float)(Math.random()*MAX_HEIGHT);
lower = new Point3d( x-RADIUS, 0.0f, z-RADIUS );
upper = new Point3d( x+RADIUS, height, z+RADIUS );
bb = new BoundingBox(lower, upper);
```

The BoundingBox is added to an ArrayList which is examined by boids calling isOverLapping():

```
public boolean isOverlapping(BoundingSphere bs)
// Does bs overlap any of the BoundingBox obstacles?
{ BoundingBox bb;
  for (int i=0; i < obsList.size(); i++) {
    bb = (BoundingBox)obsList.get(i);
```



```
      if( bb.intersect(bs) )
        return true;
    }
    return false;
  } // end of isOverlapping()
```

A boid calls isOverLapping() with a BoundingSphere object representing its position in space at that time.

5. BoidShape

As Figure 1 shows, a boid is represented by a spear-head shape. A prey boid has an orange body with a purple 'nose, while predators have a yellow body.

Even with something as simple as this shape, it is useful to do a preliminary sketch before resorting to programming. I usually draw a simple side/front/top CAD-style diagram, as in Figure 5. The points (p0, p1, etc.) will become the points in the GeometryArray.

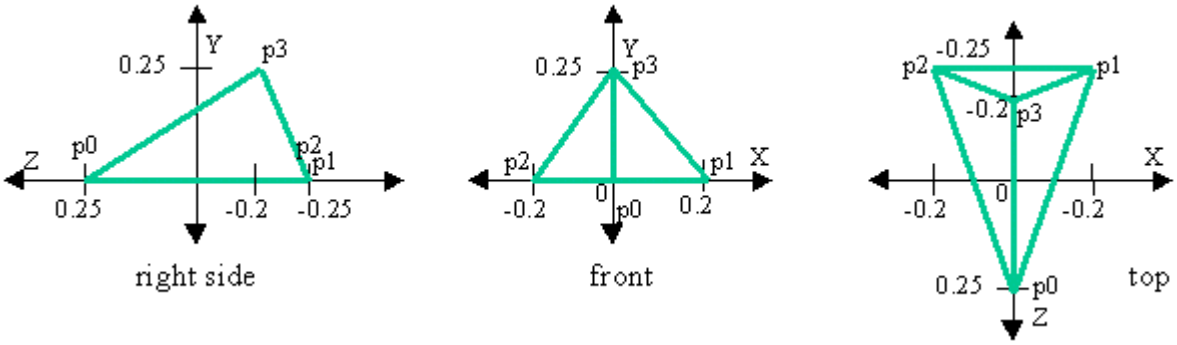

Figure 5. Sketches for a Boid Shape.

The various sides of the shape reuse the same points, which suggests the use of an IndexedGeometry (or subclass) to represent the shape. IndexedGeometry allows the sides of the shape to be specified by indices into an array of points, which means that less points are needed when creating the shape.

The spear-head is really just four triangles, and so BoidShape uses an IndexedTriangleArray:

```
IndexedTriangleArray plane =
  new IndexedTriangleArray(NUM_VERTS,
        GeometryArray.COORDINATES | GeometryArray.COLOR_3,
        NUM_INDICES );
```

There are four vertices, and 12 indices (4*3) used to specify the shape since there are four triangles.

First the points are stored in an array, and then the indices of the points array are used to define the sides in another array:

```
// the shape's coordinates
Point3f[] pts = new Point3f[NUM_VERTS];
```



```
pts[0] = new Point3f(0.0f, 0.0f, 0.25f);
pts[1] = new Point3f(0.2f, 0.0f, -0.25f);
pts[2] = new Point3f(-0.2f, 0.0f, -0.25f);
pts[3] = new Point3f(0.0f, 0.25f, -0.2f);

// anti-clockwise face definition
int[] indices = {
   2, 0, 3,      // left face
   2, 1, 0,      // bottom face
   0, 1, 3,      // right face
   1, 2, 3  };   // back face

plane.setCoordinates(0, pts);
plane.setCoordinateIndices(0, indices);
```

Some care must be taken to get the ordering of the indices correct. For example, they must be listed in a counter-clockwise order for the front of the face to be pointing outwards.

The point colours are set in the same way: an array of Color3f objects, and an array of indices into that array:

```
Color3f[] cols = new Color3f[NUM_VERTS];
cols[0] = purple;   // a purple nose
for (int i=1; i < NUM_VERTS; i++)
  cols[i] = col;   // the body colour

plane.setColors(0,cols);
plane.setColorIndices(0, indices);
```

Indices arrays can be reused (as here), which may allow some graphics cards to do further optimisations on the shape's internal representation.

Our boids do not change shape or colour, although this is quite common in flocking systems -- perhaps the boid gets bigger as it gets older or eats, and changes colour to indicate a change to its age or health. From a coding perspective, this requires a mechanism for adjusting the coordinate and/or colour values inside BoidShape. The safe way to do this, as discussed in the last chapter, is to use GeometryUpdater, maintained as an inner class of BoidShape. The GeometryArray's updateData() method would be called when the shape and/or its colour had to be changed.

6. The Boid class and its Subclasses.

The public and protected methods and data of the Boid class and its PredatorBoid and PreyBoid subclasses are shown in Figure 6.

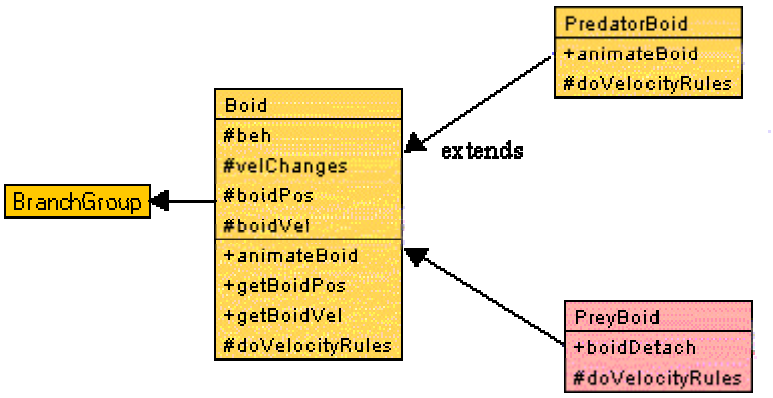

Figure 6. The Boid Class and Subclasses.

Boid represents a boid by a BranchGroup, TransformGroup and BoidShape (a Shape3D node). The TransformGroup is used to move the boid about in the scene, and the BranchGroup is required so that the boid can be removed from the scene (e.g. after being eaten). Only a BranchGroup can be detached from the scene graph. Boid is a subclass of BranchGroup.

```
public class Boid extends BranchGroup
{ ...
    :
  private TransformGroup boidTG = new TransformGroup();
    :
  public Boid(...)
  {  ...
    // set TG capabilities so the boid can be moved
    boidTG.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    boidTG.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
    addChild(boidTG);  // add TG to Boid BG
      :
    // add the boid shape to the TG
    boidTG.addChild( new BoidShape(boidColour) );
  } // end of Boid()
   :
} // end of Boid class
```

Boid Movement

The crucial boid attributes are its current position and velocity. These are initially set to random values (within a certain range), and used to position and orientate the boid via a call to moveBoid():

```
  protected Vector3f boidPos = new Vector3f();
  protected Vector3f boidVel = new Vector3f();
     :
  // used for repeated calculations
```



```
private Transform3D t3d = new Transform3D();

public Boid(...)
{ ...
  boidPos.set(randPosn(),(float)(Math.random()*6.0), randPosn());
  boidVel.set( randVel(), randVel(), randVel());
  :
  moveBoid();
  :
} // end of Boid()


private void moveBoid()
{ t3d.setIdentity();   // reset t3d
  t3d.rotY( Math.atan2( boidVel.x, boidVel.z) );  // around y-axis
           // atan2() handles 0 value input arguments
  t3d.setTranslation(boidPos);
  boidTG.setTransform(t3d);   // move the TG
} // end of moveBoid()
 :
```

The boid position and velocity vectors (boidPos and boidVel) are protected since subclasses will very probably need to access them.

moveBoid() uses the current velocity to calculate a rotation around the y-axis, and positions the boid with boidPos. A limitation of moveBoid() is that the boid does not rotate around the x- or z- axes, which means a boid always remains parallel to the XZ plane.

Animating the Boid

animateBoid() is the top-level method for animating a boid, and is called by the FlockBehavior class at each update for every boid. The method can be overridden by Boid subclasses.

```
// global constants for perching
private final static int PERCH_TIME = 5;   // how long to perch
private final static int PERCH_INTERVAL = 100;
                           // how long between perches
   :
// global variables for perching
private int perchTime = 0;
private int perchInterval = 0;
private boolean isPerching = false;
   :

public void animateBoid()
{ if (isPerching) {
    if (perchTime > 0) {
      perchTime--;  // perch a while longer
      return;       // skip rest of boid update
    }
    else {   // finished perching
      isPerching = false;
      boidPos.y = 0.1f;   // give the boid a push up off the floor
      perchInterval = 0;  // rest perching interval
    }
```



```
    }
    // update the boid's vel & posn, but keep within scene bounds
    boidVel.set( calcNewVel() );
    boidPos.add(boidVel);
    keepInBounds();
    moveBoid();
  } // end of animateBoid()
```

animateBoid() begins with code related to perching (the resting of a boid on the scene floor). keepInBounds() contains the rest of the perching code:

```
    perchInterval++;
    if ((perchInterval > PERCH_INTERVAL) &&
         (boidPos.y <= MIN_PT.y)) {   // let the boid perch
      boidPos.y = 0.0f;         // set down on the floor
      perchTime = PERCH_TIME;  // start perching time
      isPerching = true;
    }
```

A boid can initiate perching every PERCH_INTERVAL time steps, and perching lasts for PERCH_TIME steps. The perchInterval counter is incremented from 0 up past PERCH_INTERVAL, and is used to judge when perching can next begin. isPerching is a boolean, set to true when perching is started, and perchTime is set to PERCH_TIME. At each update, perchTime is decremented until it reaches 0, which causes perching to stop.

If the boid is not perching, then it calculates its new velocity with calcNewVel(), which is used to update boidVel and boidPos. These values are checked in keepInBounds() to decide if they move the boid outside of the scene's boundaries. If they do, then they are modified to move the boid back in from the boundary.

Back in animateBoid(), the boid in the scene is moved by calling moveBoid().

Velocity Rules

calcNewVel() calculates a new velocity for a boid by executing all of the velocity rules for steering the boid, and combining their results.

An important design aim is that new velocity rules can be easily added to the system (e.g. by subclassing Boid), and so calcNewVel() does not make any assumptions about the *number* of velocity rules being executed.

The solution is to have each velocity rule add its result to a global ArrayList, called velChanges. calcNewVel() iterates through the list to find all the velocities, which it adds together to get a total velocity vector.

There are two other issues: obstacle avoidance, and limiting the maximum speed. If the boid has collided with an obstacle, then the velocity change to avoid the obstacle takes priority over the other velocity rules.

The new velocity is limited to a maximum speed (i.e. magnitude) so a boid cannot attain light-speed, or something similar, by a combination of the various rules.

```
  protected ArrayList velChanges = new ArrayList(); // globals
  protected FlockBehavior beh;
     :
```



```
private Vector3f calcNewVel()
{ velChanges.clear();   // reset velocities ArrayList

  Vector3f v = avoidObstacles();         // check for obstacles
  if ((v.x == 0.0f) && (v.z == 0.0f))    // if no obs velocity
    doVelocityRules();    // then carry out other velocity rules
  else
    velChanges.add(v);    // else only do obstacle avoidance

  newVel.set( boidVel );     // re-initialise newVel
  for(int i=0; i < velChanges.size(); i++)
    newVel.add( (Vector3f)velChanges.get(i) );  // add vels

  newVel.scale( limitMaxSpeed() );
  return newVel;
}  // end of calcNewVel()


protected void doVelocityRules()
// override this method to add new velocity rules
{
  Vector3f v1 = beh.cohesion(boidPos);
  Vector3f v2 = beh.separation(boidPos);
  Vector3f v3 = beh.alignment(boidPos, boidVel);
  velChanges.add(v1);
  velChanges.add(v2);
  velChanges.add(v3);
} // end of doVelocityRules()
```

avoidObstacles() always returns a vector, even when there is no obstacle to avoid. The 'dummy' vector has the value (0,0,0) which is checked in calcNewVel().

To reduce the number of temporary objects, calcNewVel() reuses a global newVel Vector3f object in its calculations.

doVelocityRules() has protected visibility so that subclasses can readily extend it to add new steering rules. Aside from executing a rule, it is also necessary to add the result to the velChanges ArrayList. The three rules executed in Boid are Reynolds' for cohesion, separation, and alignment.

beh is a reference to the FlockBehavior subclass for the boid. Velocity rules which require the checking of flockmates, or boids from other flocks, are stored in the behaviour class, which acts as the flock manager.

Another advantage of this ArrayList approach is the ease of testing different rules (and their combinations), which is just a matter of commenting out the unwanted add () calls to velChanges.

Obstacle Avoidance

Obstacle avoidance can be computationally expensive, and can easily cripple a flocking system involving hundreds of boids and tens of obstacles. The reason is two-fold: the need to keep looking ahead to detect a collision *before* the boid image intersects with the obstacle, and the need to calculate a rebound velocity which mimics the physical reality of a boid hitting the obstacle.

As usual, a trade-off is made between the accuracy of the real-world simulation, and the need for fast computation. In Flocking3D, the emphasis is on speed, so there is no look-ahead collision detection, and no complex rebound vector calculation.

The boid is represented by a bounding sphere, whose radius is fixed but center moves as the boid moves. The sphere is tested for intersection with all the obstacles, and if an intersection is found, then a velocity is calculated based on the negation of the boid's current (x,z) position, scaled by a factor to reduce its effect.

```
private Vector3f avoidObstacles()
{ avoidOb.set(0,0,0);   // reset
  // update the BoundingSphere's position
  bs.setCenter( new Point3d( (double)boidPos.x,
                   (double)boidPos.y, (double)boidPos.z) );
  if ( obstacles.isOverlapping(bs)) {
    avoidOb.set( -(float)Math.random()*boidPos.x, 0.0f,
                 -(float)Math.random()*boidPos.z);
    // scale to reduce distance moved away from the obstacle
    avoidOb.scale(AVOID_WEIGHT);
  }
  return avoidOb;
}  // end of avoidObstacles()
```

There is no adjustment to the boid's y- velocity which means that if it hits an obstacle from the top, its y-axis velocity will be unchanged (i.e. it will keep moving downwards), but will change its x- and z- components.

avoidOb is a global Vector3f object used instead of creating a new temporary object each time that obstacle checking is carried out.

The bs BoundingSphere object is created when the boid is first instantiated.

Staying in Bounds

keepInBounds() checks a bounding volume defined by two points, MIN_PT and MAX_PT, representing its upper and lower corners. If the boid's position is beyond a boundary then it is relocated to the boundary, and its velocity component in that direction is reversed.

The code fragment below shows what happens when the boid has passed the upper x-axis boundary:

```
if (boidPos.x > MAX_PT.x) {     // beyond upper x-axis boundary
  boidPos.x = MAX_PT.x;         // put back at edge
  boidVel.x = -Math.abs(boidVel.x);   // move away from boundary
}
```

The same coding approach is used to check for the upper and lower boundaries along all the axes.

6.1. PreyBoid

A prey boid has an orange body and wants to avoid being eaten by predators. It does this by applying a new velocity rule for detecting and evading predators. It also has a higher maximum speed than the standard Boid, so may be able to outrun an attacker.

Since a PreyBoid can be eaten, it must be possible to detach the boid from the scene graph.

```
public class PreyBoid extends Boid
{
  private final static Color3f orange = new Color3f(1.0f,0.75f,0.0f);

  public PreyBoid(Obstacles obs, PreyBehavior beh)
  { super(orange, 2.0f, obs, beh);     // orange and higher max speed
    setCapability(BranchGroup.ALLOW_DETACH);   // prey can be 'eaten'
  }

  protected void doVelocityRules()
  // Override doVelocityRules() to evade nearby predators
  { Vector3f v = ((PreyBehavior)beh).seePredators(boidPos);
    velChanges.add(v);
    super.doVelocityRules();
  }  // end of doVelocityRules()

  public void boidDetach()
  { detach(); }
}
```

The benefits of inheritance are clear, making PreyBoid very simple to define.

doVelocityRules() in PreyBoid adds a rule to the ones in Boid, and so calls the superclass' method to evaluate those rules as well.

seePredators() is located in the PreyBehaviour object, the manager for the prey flock. The method looks for nearby predators and returns a 'flee' velocity. seePredators() is inside PreyBehavior because it needs to examine a flock.

6.2. PredatorBoid

A predator has a yellow body and can get hungry. Hunger will cause it to do two things: the predator will eat a prey boid which is sufficiently close, and will trigger a velocity rule to make the predator head towards nearby prey groups.

```
public class PredatorBoid extends Boid
{ private final static Color3f yellow = new Color3f(1.0f, 1.0f,0.6f);
  private final static int HUNGER_TRIGGER = 3;
                      // when hunger affects behaviour
  private int hungerCount;

  public PredatorBoid(Obstacles obs, PredatorBehavior beh)
  { super(yellow, 1.0f, obs, beh);   // yellow boid, normal max speed
    hungerCount = 0;
  }

  public void animateBoid()
  // extend animateBoid() with eating behaviour
```



```
  { hungerCount++;
     if (hungerCount > HUNGER_TRIGGER)   // time to eat
       hungerCount -= ((PredatorBehavior)beh).eatClosePrey(boidPos);
     super.animateBoid();
  }

  protected void doVelocityRules()
  // extend VelocityRules() with prey attack
  { if (hungerCount > HUNGER_TRIGGER) {  // time to eat
      Vector3f v = ((PredatorBehavior)beh).findClosePrey(boidPos);
      velChanges.add(v);
    }
    super.doVelocityRules();
  }

} // end of PredatorBoid class
```

The eating of prey is not a velocity rule, so is carried out by extending the behaviour of animateBoid(). eatClosePrey() is located in PredatorBehavior because it examines (and modifies) a flock, which means that it should be handled by the flock manager. The method returns the number of prey eaten (usually 0 or 1), reducing the predator's hunger.

The movement towards prey *is* a velocity rule so is placed in an overridden doVelocityRules() method. Since findClosePrey() is looking a flock, it is carried out by PredatorBehavior.

7. BoidsList

The FlockBehavior class is a flock manager, and consequently must maintain a list of boids. The obvious data structure is an ArrayList, but there is a subtle issue: boids may be deleted from the list, as when a prey boid is eaten. This can cause a synchronization problem because of the presence of multiple behaviour threads in the application.

For example, the PredatorBehavior thread may delete a prey boid from the prey list at the same time that PreyBehavior is about to access the boid in the self-same list. The solution is to synchronize the deletion and accessing operations, so they cannot occur simultaneously. This is the purpose of the BoidsList class:

```
public class BoidsList extends ArrayList
{
  public BoidsList(int num)
  {  super(num);  }

  synchronized public Boid getBoid(int i)
  // return the boid if it is visible; null otherwise
  { if (i < super.size())
      return (Boid)get(i);
    return null;
  }

 synchronized public boolean removeBoid(int i)
 // attempt to remove the i'th boid
 { if (i < super.size()) {
```



```
      super.remove(i);
      return true;
    }
    return false;
  }

}  // end of BoidsList class
```

Another consequence of the dynamic change of the boids list is that code should not assume that its length stays the same. This means, for instance, that for-loops using the list size should be avoided. An alternative is a while-loop which uses a null result from calling getBoid() to finish.

8. FlockBehavior and its Subclasses

The public and protected methods and data of the FlockBehavior class and its PredatorBehavior and PreyBehavior subclasses are shown in Figure 7.

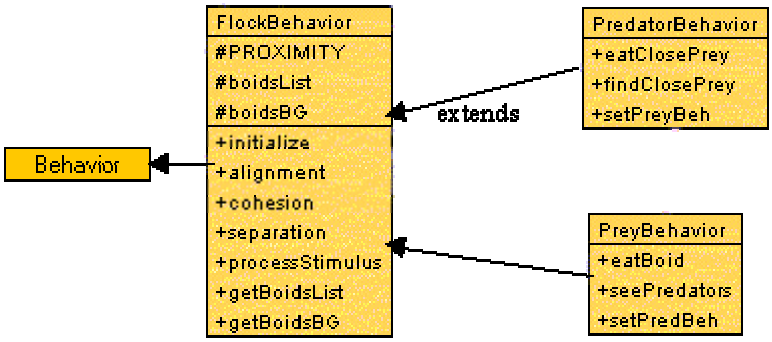

Figure 7. FlockBehavior and its Subclasses.

FlockBehavior has two main tasks:

1. to periodically call animateBoid() on every boid;
2. to store the velocity rules which require an examination of the entire flock.

FlockBehavior does not make boids – that is done by its subclasses. PredatorBehavior creates a series of PredatorBoids, while PreyBehavior handles PreyBoids. But the subclass behaviours do use the inherited boidsList list for storage.

Task (1) is done from within processStimulus():

```
public void processStimulus(Enumeration en)
{ Boid b;
  int i = 0;
  while((b = boidsList.getBoid(i)) != null) {
    b.animateBoid();
    i++;
  }
  wakeupOn(timeOut);   // schedule next update
```



```
  }
```

Note the coding style of the while-loop, which does not use the ArrayList's size.

Velocity Rules Again

FlockBehavior is a general purpose flock manager, so stores the basic velocity methods used by all boids: Reynolds' cohesion, separation, and alignment rules.

All the rules have a similar implementation since they all examine nearby flockmates, building an aggregate result during the process. This result is converted into a velocity and scaled before being returned.

As explained at the start of the chapter, Reynolds' notion of flockmates is based on a distance measure and an angle spread around the forward direction of the boid. However, the rules in Flocking3D only utilise the distance value, effectively including flockmates from all around the boid. The angle measure was dropped because of the overhead of calculating it, and because the behavior of the boids seems realistic enough without it.

The cohesion() method is shown below. It calculates a velocity which encourages the boid to fly towards the average position of its flockmates.

```
public Vector3f cohesion(Vector3f boidPos)
{ avgPosn.set(0,0,0);     // the default answer
  int numFlockMates = 0;
  Vector3f pos;
  Boid b;

  int i = 0;
  while((b = boidsList.getBoid(i)) != null) {
    distFrom.set(boidPos);
    pos = b.getBoidPos();
    distFrom.sub(pos);
    if(distFrom.length() < PROXIMITY) {   // is boid a flockmate?
      avgPosn.add(pos);     // add position to tally
      numFlockMates++;
    }
    i++;
  }
  avgPosn.sub(boidPos);     // don't include the boid itself
  numFlockMates--;

  if(numFlockMates > 0) {   // there were flockmates
    avgPosn.scale(1.0f/numFlockMates);   // calculate avg position
    // calculate a small step towards the avg. posn
    avgPosn.sub(boidPos);
    avgPosn.scale(COHESION_WEIGHT);
  }
  return avgPosn;
}
```

avgPosn and distPosn are global Vector3f objects to reduce the creation of temporary objects when the method is repeatedly executed.

The while-loop uses getBoid() to iterate through the boids list. The loop's complexity is O(n), where n is the number of boids. Since the method is called for every boid, checking the entire flock for cohesion is $O(n^2)$. If there are *m* velocity rules, then each update of the flock will have a complexity of $O(m*n^2)$. This is less than ideal, and one reason why flocking systems tend to have few boids.

The solution lies with the design of the boids list data structure, which should utilise boid position in its ordering. A search algorithm using spatial information should be able to find a nearby boid in constant time (O(1)), reducing the cost of a flock update to linear proportions: O(m*n).

In cohesion(), nearness is calculated by finding the absolute distance between the boid (boidPos) and the one under consideration. The PROXIMITY constant can be adjusted to include different numbers of boids as flockmates.

The choice of scaling factor (COHESION_WEIGHT in this case) is a matter of trial-and-error, which was carried out by running the system with only the cohesion rule, and observing the behaviour of the flock.

8.1. PreyBehavior

PreyBehavior has three tasks:

1. to create its boids (e.g. PreyBoids) and store them in the boids list;
2. to store the velocity rules specific to PreyBoids, which require an examination of the entire flock;
3. to delete a PreyBoid when a predator eats it.

Boid creation is done in createBoids() which is called from the behaviour's constructor.

```
private void createBoids(int numBoids, Obstacles obs)
{ // preyBoids can be detached from the scene
  boidsBG.setCapability(BranchGroup.ALLOW_CHILDREN_WRITE);
  boidsBG.setCapability(BranchGroup.ALLOW_CHILDREN_EXTEND);

  PreyBoid pb;
  for(int i=0; i < numBoids; i++){
    pb = new PreyBoid(obs, this);
    boidsBG.addChild(pb);   // add to BranchGroup
    boidsList.add(pb);      // add to BoidsList
  }
  boidsBG.addChild(this);   // store the prey behaviour with its BG
}
```

boidsBG is the inherited BranchGroup for the collection of boids in the scene, and must have its capabilities changed to allow prey boids to be detached from it at run-time.

When a PreyBoid object is created, it is passed a reference to the obstacles (passed to the behaviour from WrapCheckers3D), and a reference to the behaviour itself, so that its velocity rules can be accessed.

createBoid() creates a scene branch like that shown in Figure 8.

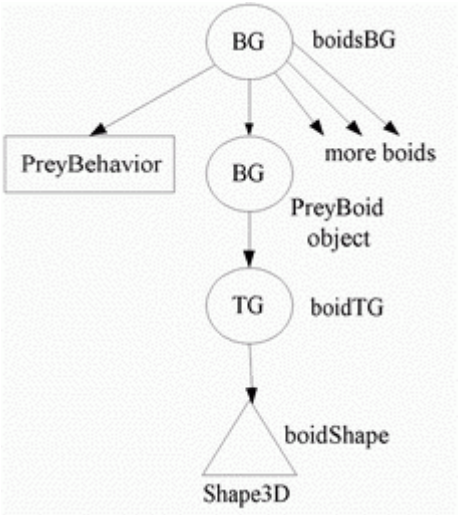

Figure 8. Scene Branch for PreyBoid nodes.

Velocity Rules and Other Flocks

PreyBehavior's task (2) is complicated by the need to access the other flock (the predators), which means that it must have a reference to PredatorBehavior.

WrapFlocking3D passes in a reference to the PredatorBehaviour object by calling PreyBehavior's setPredBeh():

```
public void setPredBeh(PredatorBehavior pb)
{ predBeh = pb; }
```

predBeh is a global, used by the seePredators() velocity rule.

seePredators() is coded in a similar way to the rules in FlockBehavior: a loop tests each of the boids in the flock for proximity. If a predator is within range, then the velocity is set to a scaled move in the opposite direction.

```
public Vector3f seePredators(Vector3f boidPos)
{
  predsList = predBeh.getBoidsList();  // refer to pred list
  avoidPred.set(0,0,0);  // reset
  Vector3f predPos;
  PredatorBoid b;

  int i = 0;
  while((b = (PredatorBoid)predsList.getBoid(i)) != null) {
    distFrom.set(boidPos);
    predPos = b.getBoidPos();
    distFrom.sub(predPos);
    if(distFrom.length() < PROXIMITY) {   // is pred boid close?
      avoidPred.set(distFrom);
      avoidPred.scale(FLEE_WEIGHT);  // scaled run away
      break;
    }
    i++;
  }
  return avoidPred;
```



```
} // end of seePredators()
```

An important difference from earlier rules is the first line, where a reference to the predators list is obtained by calling getBoidsList() in PredatorBehavior.

Goodbye Prey

PreyBehavior contains an eatBoid() method that is called by PredatorBehavior when the given boid has been eaten.

```
public void eatBoid(int i)
{ ((PreyBoid)boidsList.getBoid(i)).boidDetach();
  boidsList.removeBoid(i);
}
```

Deleting a boid involves its detachment from the scene graph and its removal from the boids list.

8.2. PredatorBehavior

PredatorBehavior has similar tasks to PreyBehavior:

1. to create its boids (e.g. PredatorBoids) and store them in the boids list;
2. to store the velocity rules specific to PredatorBoids, which require an examination of the entire flock;
3. to eat prey when they're close enough.

Boid creation is virtually identical to that done in PreyBehavior, except that PredatorBoids are created.

PredatorBoid has a method, setPredBeh(), which allows WrapFlocking3D to pass it a reference to PreyBehavior.

The velocity rule is findClosePrey(), which involves the predator in calculating the average position of all the nearby PreyBoids, and moving a small step towards that position. The code is similar to other rules, except that it starts by obtaining a reference to the prey list by calling getBoidsList() in PreyBehavior.

```
preyList = preyBeh.getBoidsList();  // get prey list
 :
int i = 0;
while((b = (PreyBoid)preyList.getBoid(i)) != null) {
  pos = b.getBoidPos();
  :
}
:
```


Lunch Time

Task (3) for PredatorBehavior, eating nearby prey, is not a velocity rule, but a method called at the start of each predator's update in animateBoid(). However, the coding is similar to the velocity rules: it iterates through the prey boids checking for those near enough to eat.

```
public int eatClosePrey(Vector3f boidPos)
{ preyList = preyBeh.getBoidsList();
  int numPrey = preyList.size();
  int numEaten = 0;
  PreyBoid b;

  int i = 0;
  while((b = (PreyBoid)preyList.getBoid(i)) != null) {
    distFrom.set(boidPos);
    distFrom.sub( b.getBoidPos() );
    if(distFrom.length() < PROXIMITY/3.0) { // boid v.close to prey
      preyBeh.eatBoid(i);   // found prey, so eat it
      numPrey--;
      numEaten++;
      System.out.println("numPrey: " + numPrey);
    }
    else
      i++;
  }
  return numEaten;
}
```

The reference to PreyBehavior is used to get a link to the prey list, and also to call eatBoid() to remove the i^{th} boid. When a boid is removed, the next boid in the list will become the new i^{th} boid, so the i index must not be incremented.

9. Other Coding Approaches

Flocking3D was influenced a little by the Java 3D flocking code of Anthony Steed, which he developed back in 1998 as part of a comparison with VRML (http://www.cs.ucl.ac.uk/staff/A.Steed/3ddesktop/).

He represents each boid as a TransformGroup, and utilises a BoidSet object (a BranchGroup subclass) for a flock. An interesting feature is the use of morphing to create wing flapping, as a transition between three different TriangleArrays.

A FlockBehavior class uses WakeupOnFramesElapsed(0) to trigger boid updating, and a FlapBehavior class for the wings is triggered by WakeupOnTransformChanged events when the boid moves.

Flock dynamics include perching, speed limiting, proximity detection, and inertia. There is only a single kind of boid, and no obstacles in the scene.

