

Java Gaming & Graphics Programming



Killer Game Programming

in Java

O'REILLY®

Andrew Davison

Flocking Boids

Flocking is a computer model for the coordinated motion of groups (or flocks) of entities called boids. Flocking represents group movement—as seen in bird flocks and fish schools—as combinations of steering behaviors for individual boids, based on the position and velocities of nearby flockmates. Though individual flocking behaviors (sometimes called *rules*) are quite simple, they combine to give boids and flocks interesting overall behaviors, which would be complicated to program explicitly.

Flocking is often grouped with *Artificial Life* algorithms because of its use of *emergence*: complex global behaviors 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 do something different a few moments later. Flocking is useful for games where groups of things, such as soldiers, monsters, or crowds move in complex, coordinated ways.



Flocking appears in games such as *Unreal* (Epic), *Half-Life* (Sierra), and *Enemy Nations* (Windward Studios).

Flocking was first proposed by Craig Reynolds in his paper “Flocks, Herd, and Schools: A Distributed Behavioral Model,” published in *Computer Graphics*, 21(4), SIGGRAPH'87, pp. 25–34.

The basic flocking model consists of three simple steering behaviors (or rules):

Separation

Steer to avoid crowding local flockmates.

Alignment

Steer toward the average heading of local flockmates.

Cohesion

Steer to move toward the average position of local flockmates.

These rules are illustrated in Figure 22-1.

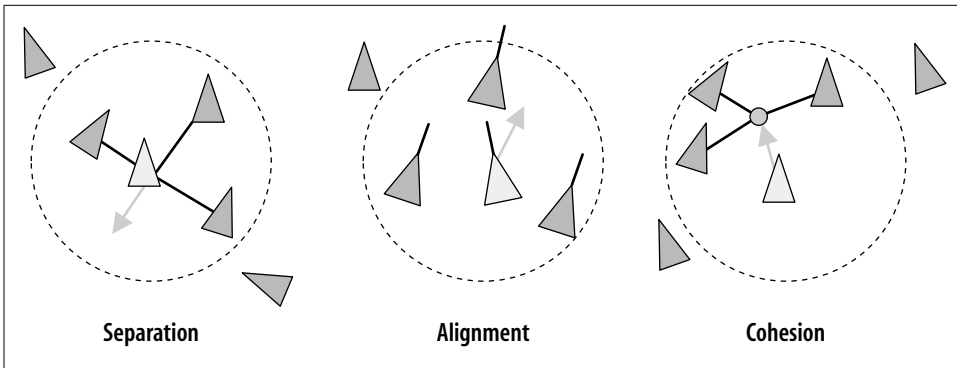


Figure 22-1. Reynolds' steering rules

The circles in Figure 22-1 surround the center boid's local flockmates. Any boids beyond a certain distance of the central boid don't figure in the rule-based calculations.

A more elaborate notion of neighborhood only considers flockmates surrounding the boid's current forward direction (see Figure 22-2). The extent of neighborhood is governed by an arc on either side of the forward vector. This reduced space more closely reflects how real-world flock members interact.

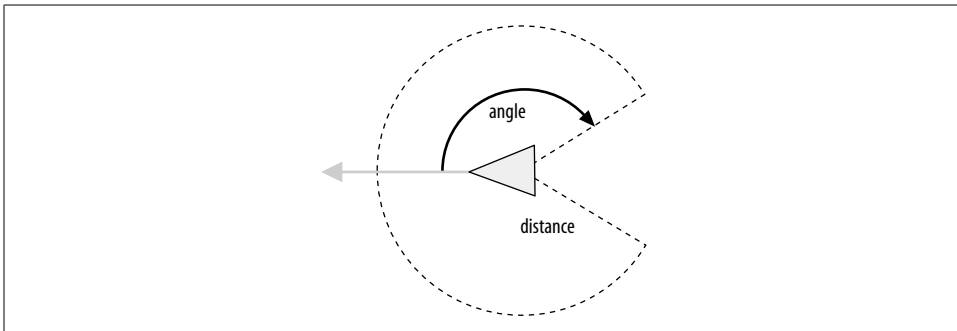


Figure 22-2. Boid neighborhood based on distance and angle

Many other rules have been proposed over the years, including ones for obstacle avoidance and goal seeking. Reynold's web site (<http://www.red3d.com/cwr/boids/>) contains hundreds of links to relevant information, including flocking in games, virtual reality, computer graphics, robotics, art, and artificial life. The plethora of links at Reynold's site can be a tad daunting. A good starting point is Conrad Parker's web page explaining 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 my boids' steering rules.

Two other good starting points are Steven Woodcock’s articles “Flocking: A Simple Technique for Simulating Group Behavior” in *Game Programming Gems* and “Flocking with Teeth: Predators and Prey” in *Game Programming Gems II*. The first paper describes Reynolds’ basic steering rules, and the second introduces predators and prey, and static obstacles. Both articles come with C++ source code.

A Flocking Application

My Flocking3D application is shown in Figure 22-3. It involves the interaction of two different groups of boids: the yellow flock are the predators, and the orange ones the prey. Over time, the boids in the orange flock are slowly eaten though they try their best to avoid it. Both flocks must avoid obstacles and stay within the bounds of the scene.

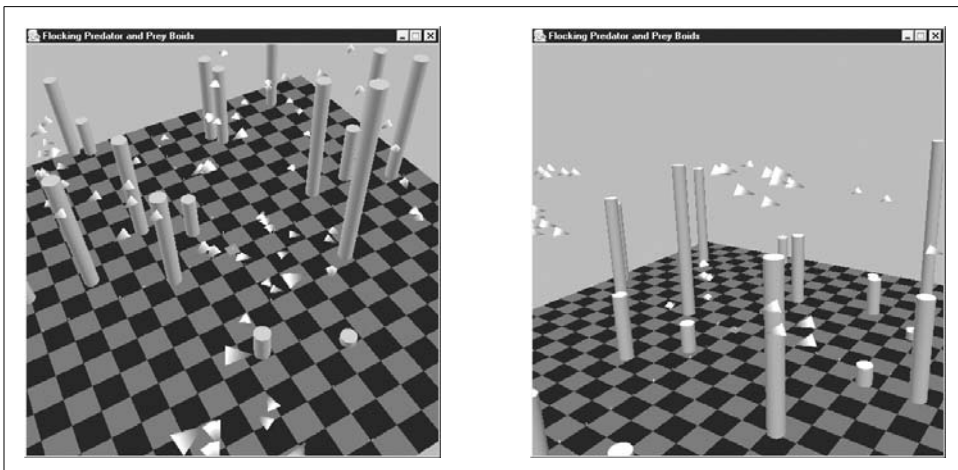


Figure 22-3. Predator and prey flocks in Flocking3D

The lefthand image shows an early stage in the system, when the predators are chasing prey. The righthand image was taken after all the prey have been eaten, and the predators are flying in groups.



The code in this chapter was developed by one of my students, Miss Sirinart Sakarin, and me; the code can be found in *Flocking3D*.

Boid behavior 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 (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 will eat the prey.
- Prey boids try to avoid predators (they have an aversion to being eaten).

The implementation techniques illustrated in this example include:

Inheritance

Inheritance is used to define the boids (as subclasses of `Boid`) and their behaviors (as subclasses of `FlockingBehavior`).

Geometry building

The boid shape (a sort of arrowhead) is built using Java 3D's `IndexedTriangleArray`.

Synchronized updates

Updates to the boids must be controlled, so changes to the flock in each time interval only become visible when every boid in the flock has been changed.

Scene graph detaching

When a boid is eaten, it's removed from the scene graph by having its `BranchGroup` detached from its parent node.

Figure 22-4 shows the class diagrams for the application. Only the class names are shown; superclasses that are a standard part of Java or Java 3D (e.g., `JPanel`, `Shape3D`) are omitted.

Flocking3D's Ancestor

Flocking3D was influenced by Anthony Steed's Java 3D flocking program, developed back in 1998 as part of a comparison with VRML (<http://www.cs.ucl.ac.uk/staff/A.Steed/3ddesktop/>). Each boid in his work is represented as a `TransformGroup` and utilizes a `BoidSet` object (a `BranchGroup` subclass) for a flock. An interesting feature is his use of morphing for wing flapping, implemented as a transition between three `TriangleArrays`. A `FlockBehavior` class uses `WakeupOnFramesElapsed(0)` to trigger boid updates, and a `FlapBehavior` object for the wings is triggered by `WakeupOnTransformChanged` events when the boid moves. Flock dynamics include perching, speed limiting, proximity detection, and inertia. There's only one kind of boid, and no obstacles in the scene.

Flocking3D is the top-level `JFrame`. `WrapFlocking3D` creates the 3D scene, the predator and prey behaviors, and the obstacles. `PreyBehavior` and `PredatorBehavior` can be thought of as flock managers: they initialize the boids that make up their flock and handle rule evaluation at runtime; they're subclasses of the `FlockBehavior` class.

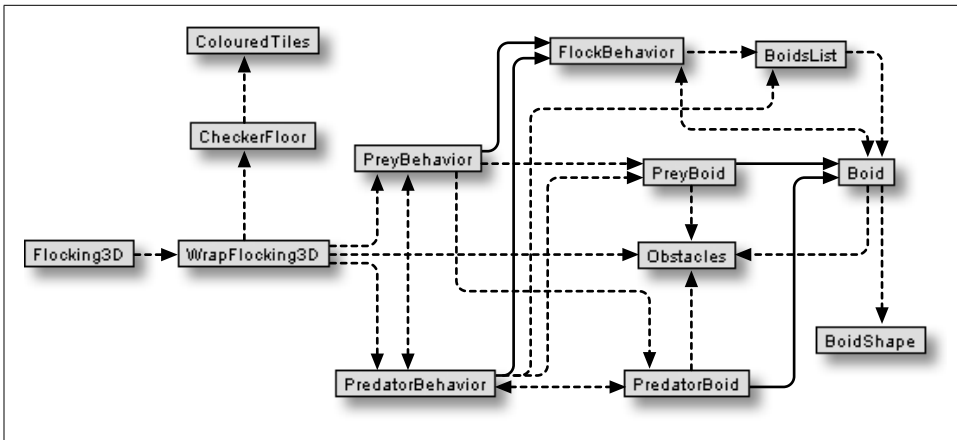


Figure 22-4. The Flocking3D classes

PreyBoid and PredatorBoid represent individual boids and are subclasses of the Boid class. BoidShape is a subclass of Shape3D and manages the boid shape. CheckerFloor and ColouredTiles are the same as in previous examples.

Scene Creation

WrapFlocking3D is like previous Wrap classes in that it creates a 3D scene inside a JPanel, made up of a checkerboard floor, blue sky, lighting, and an OrbitBehavior node to allow the user to adjust the viewpoint.

The additions are the obstacles and the two flocks, both created 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 behaviors 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` or are assigned default values. The behaviors are passed references to each other so their steering rules can consider neighboring boids of the other type. Here's a typical call to `Flocking3D`:

```
java Flocking3D 10 200 15
```

This will make `addFlockingBoids()` create 10 predators, 200 prey, and 15 obstacles.

Adding Obstacles

The `Obstacles` class creates a series of blue cylinders placed at random locations 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; the `BranchGroup` is then retrieved by calling `getObsBG()`.

At the same time that a cylinder is being created, a `BoundingBox` is 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);
```

A boid checks an obstacle's bounding box to avoid colliding with it. The bounding boxes for all the obstacles are added to an `ArrayList`, which is examined by boids when they call `isOverlapping()`. A boid calls `isOverlapping()` with a `BoundingSphere` object representing its current position:

```
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()
```

The `isOverlapping()` method sacrifices efficiency for simplicity: the bounding sphere is checked against every obstacle's bounding box in the scene. An obvious improvement would be to order the bounding boxes in some way to reduce the number that needs to be tested. However, this would complicate the code and wouldn't produce much improvement for the small number of obstacles used in my examples.

The boid shape

A boid is represented by a spearhead shape, which is shown from three different directions in Figure 22-5. A prey boid has an orange body with a purple nose, while predators are completely yellow.

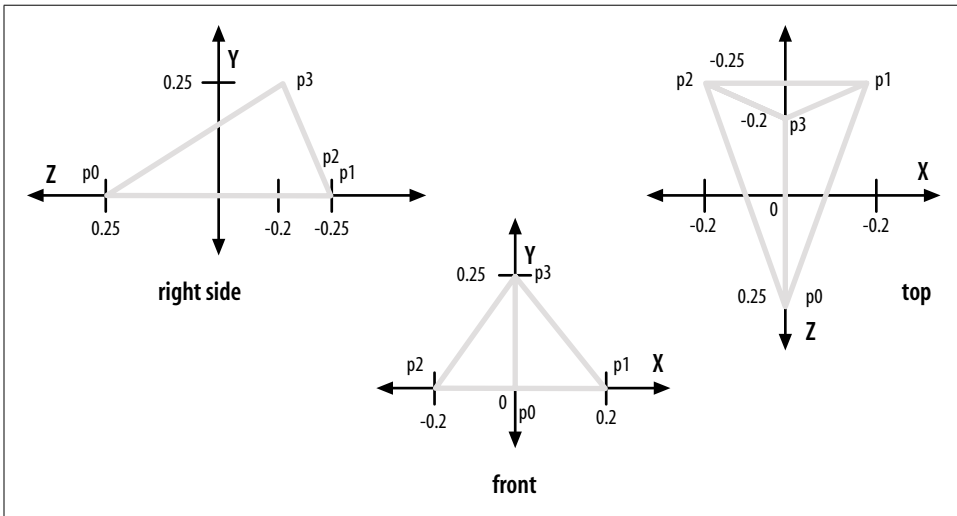


Figure 22-5. CAD sketches for a boid shape

Even with something as simple as this shape, doing a preliminary sketch before resorting to programming is useful. I usually draw a simple side/front/top CAD-style diagram, as in Figure 22-5. The points in the diagrams (p0, p1, etc.) will become the points in the resulting `GeometryArray`.

Different sides of the shape reuse the same points, which suggests that Java 3D's `IndexedGeometry` (or a subclass) should represent the shape. `IndexedGeometry` allows sides to be specified by indices into an array of points, which means fewer points are needed when creating the shape.

The spearhead is made up of four triangles, so my `BoidShape` class uses a Java 3D `IndexedTriangleArray`:

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

The shape is made from four triangles, but the sharing of sides means there's only four different vertices (labeled as p0, p1, p2, and p3 in Figure 22-5). As a consequence, `NUM_VERTS` is 4 in the code above. Each vertex has an (x, y, z) coordinate, so the `IndexedTriangleArray` will use 12 indices (4 vertices each require 3 values). Consequently, `NUM_INDICES` has the value 12.

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. The points for a face must be listed in counterclockwise order for the front of the face to point toward the viewer. This is an example of the “righthand rule” for orienting normals (discussed back in Chapter 16).

The point colors are set in the same way with an array of Java 3D `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 color

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

The array holding the indices (i.e., `indices`) can be reused, which may allow some graphics cards to do further optimizations on the shape’s internal representation.

My boids don’t change shape or color though this is quite common in other flocking systems; perhaps the boid gets bigger as it gets older or eats and changes color to indicate a change to its age or health. From a coding perspective, this requires a mechanism for adjusting the coordinate and/or color values inside `BoidShape`. The safe way to do this, as discussed in the last chapter, is to use a Java 3D `GeometryUpdater`, maintained as an inner class of `BoidShape`. The `GeometryArray`’s `updateData()` method would be called when the shape and/or its color had to be changed.

Types of Boids

The public and protected methods and data of the Boid class and its PredatorBoid and PreyBoid subclasses are shown in Figure 22-6.

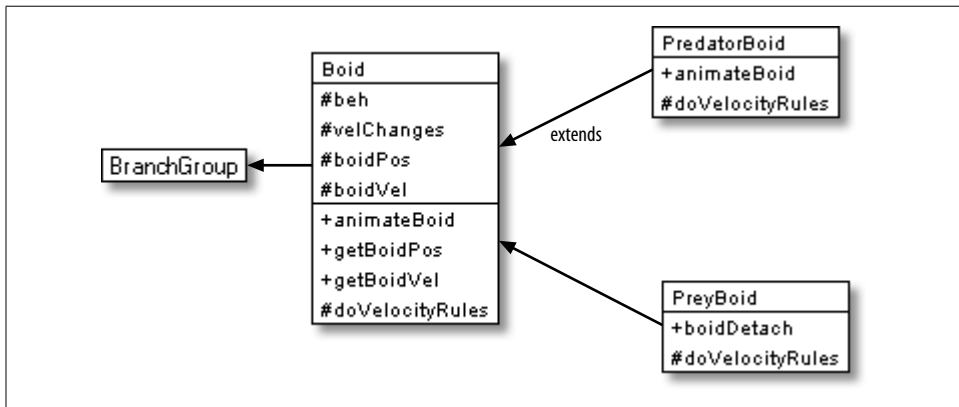


Figure 22-6. The Boid class and subclasses

Boid represents a boid using a BranchGroup, TransformGroup, and BoidShape (a Shape3D node). The TransformGroup moves the boid about in the scene, and the BranchGroup permits the boid to be removed from the scene (e.g., after being eaten). Only a BranchGroup can be detached from a scene graph. The subgraph is built in the Boid() constructor:

```
private TransformGroup boidTG = new TransformGroup(); // global

public Boid(...)
{ // other initialization code, and then ...

    // 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 BranchGroup

    // add the boid shape to the TG
    boidTG.addChild( new BoidShape(boidColour) );
} // end of Boid()
```

Boid Movement

The most important attributes for a boid are its current position and velocity. These are set to random values (within a certain range) in the Boid constructor:

```
// globals
protected Vector3f boidPos = new Vector3f();
protected Vector3f boidVel = new Vector3f();
```

```

// code in the Boid constructor
boidPos.set(randPosn(),(float)(Math.random()*6.0), randPosn());
boidVel.set( randVel(), randVel(), randVel());

```

The `moveBoid()` method uses them to orient and position the boid in the scene:

```

// global used for repeated calculations
private Transform3D t3d = new Transform3D();

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, so subclasses that need to access or change their values can do so easily. `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 doesn't rotate around the x- or z-axes, which means a boid always remains parallel to the XZ plane. For example, a boid can't turn downward into a nose dive or lean over as it turns sharply. This restriction can be removed by adding extra `TransformGroup` nodes above the boid's current `TransformGroup` to handle the other forms of rotation.

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. `animateBoid()` can be overridden by `Boid` subclasses (e.g., by `PredatorBoid`) to modify the animation activities.

`animateBoid()` begins with code related to boid *perching*, which is when a boid is stationary on the floor:

```

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

```

`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;
}

```

Perching sessions occur at fixed intervals, and each session lasts for a fixed amount of time. During the `PERCH_INTERVAL` time, the boid must fly about, until the interval is met; then the boid can perch for `PERCH_TIME` time units before it must start flying again. `animateBoid()` is called repeatedly by the `FlockBehavior` object, and `perchInterval` records the number of calls. When the count exceeds the number stored in `PERCH_INTERVAL`, perching is initiated.

`isPerching` is set to true when perching starts, and `perchTime` is set to `PERCH_TIME`. Each time that `animateBoid()` is called after that, `perchTime` is decremented. When `perchTime` reaches 0, perching stops. The `perchInterval` counter is reset to 0, and the perching interval can start being measured off again.

This code illustrates how the timing of boid activities can be implemented using counters. This is possible since `animateBoid()` is called at fixed intervals by `FlockingBehavior`.

If the boid is not perching, then `calcNewVel()` calculates the boid's new velocity and updates `boidVel` and `boidPos` with this velocity. `keepInBounds()` checks these values to decide if they specify a new position for the boid outside the scene's boundaries. If they do, then the values are modified so that when they're applied to the boid, it will stay within the boundary. Back in `animateBoid()`, the boid in the scene is moved by `moveBoid()`.

Velocity Rules

`calcNewVel()` calculates a new velocity for a boid by executing all of the velocity rules for steering the boid. Each rule returns a velocity, and these velocities are summed by `calcNewVel()` to get a total.

An important design aim is that new velocity rules can be easily added to the system (e.g., by subclassing `Boid`), so `calcNewVel()` doesn't make any assumptions about the number of velocity rules being executed. Each velocity rule adds its result to a global `ArrayList`, called `velChanges`. `calcNewVel()` iterates through the list to find all the velocities.

Two other issues are 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 value, so a boid cannot attain the speed of light, or something similar by the 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 “no obstacle” vector has the value $(0, 0, 0)$, which is detected by `calcNewVel()`.



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

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

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

Obstacle Avoidance

Obstacle avoidance can be computationally expensive, easily crippling a flocking system involving hundreds of boids and tens of obstacles. The reason is two-fold: the algorithm has to keep looking ahead to detect a collision before the boid image intersects with the obstacle and, therefore, should calculate a rebound velocity that 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’s no look-ahead collision detection and no complex rebound vector calculation. A boid is allowed to hit (and enter) an obstacle and rebounds in the simplest way possible.

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;
}
```

There's 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'll keep moving downward) but it will change its x- and z-components.

Instead of creating a new temporary object each time obstacle checking is carried out, the code utilizes a global `Vector3f` object called `avoidOb`. 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's relocated to the boundary, and its velocity component in that direction is reversed.

The following code fragment 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 approach is used to check for the upper and lower boundaries along all the axes.

The Prey Boid

A prey boid has an orange body and wants to avoid being eaten by predators. It does this by applying a velocity rule for detecting and evading predators. It has a higher maximum speed than the standard `Boid`, so it 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, as it's simple to define `PreyBoid`. `doVelocityRules()` in `PreyBoid` adds a rule to the ones present in `Boid` and calls the superclass's method to evaluate those rules as well.

`seePredators()` is located in the `PreyBehavior` 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.

The Predator Boid

A predator gets hungry. Hunger will cause it to do two things: eat a prey boid, if one is sufficiently close (as defined by the `eatClosePrey()` method), and trigger a velocity rule to make the predator subsequently pursue 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 behavior
  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 behavior
  { 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

```

Eating prey isn't a velocity rule, so it is carried out by extending the behavior of `animateBoid()`. `eatClosePrey()` is located in `PredatorBehavior` because it examines (and modifies) a flock, which means that it's handled by the flock manager. The

method returns the number of prey eaten (usually 0 or 1), reducing the predator's hunger.

The movement toward prey is a velocity rule, so it is placed in the overridden `doVelocityRules()` method. Since `findClosePrey()` is looking at a flock, it's carried out by `PredatorBehavior`.

Grouping the Boids

The `FlockBehavior` class is a flock manager and, consequently, must maintain a list of boids. The obvious data structure for the task is an `ArrayList`, but there's a subtle problem: boids may be deleted from the list, as when a prey boid is eaten. This can cause synchronization problems because of the presence of multiple behavior 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 same boid in the list. The solution is to synchronize the deleting and accessing operations so they can't 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 the list's length stays the same. This means, for instance, that for loops using the list size should be avoided. If a for loop was utilized, then a change in the boids list may cause a boid to be processed twice, or skipped, because of its index position changing.

Flock Behavior

The public and protected methods and data of the FlockBehavior class and its PredatorBehavior and PreyBehavior subclasses are shown in Figure 22-7.

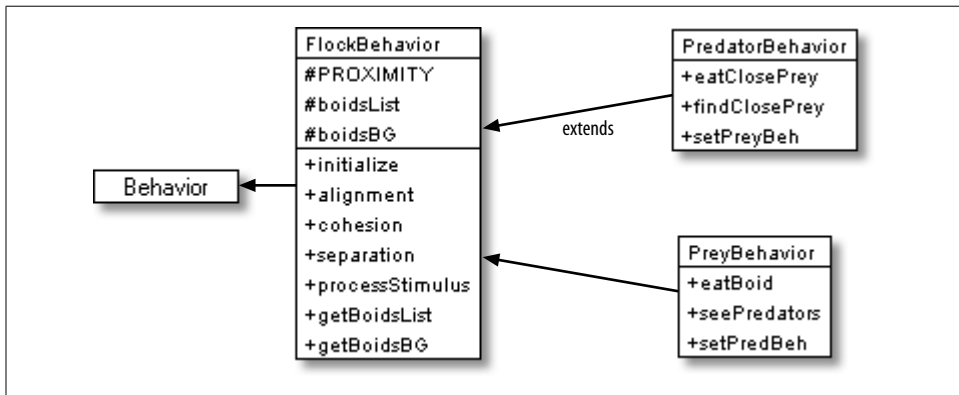


Figure 22-7. FlockBehavior and its subclasses

FlockBehavior has two main tasks:

- To call `animateBoid()` on every boid periodically
- To store the velocity rules, which require an examination of the entire flock

FlockBehavior doesn't create boids since that's handled by its subclasses, and the information required for each type of boid is too specialized to be located in the superclass. **PredatorBehavior** creates a series of **PredatorBoids**, and **PreyBehavior** handles **PreyBoids**. But the subclass behaviors do use the inherited `boidsList` list for storage.

Animate the Boids

The calls to `animateBoid()` are carried out in `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
}
```

Velocity Rules Again

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

the rules have a similar implementation since they examine nearby flockmates and build an aggregate result. This is converted into a velocity and scaled before being returned.

As explained at the start of this chapter, Reynolds' notion of flockmates is based on a distance measure and an angle around the forward direction of the boid. However, the rules in `Flocking3D` only utilize the distance value, effectively including flockmates from all around the boid. The angle measure is dropped because of the overhead of calculating it and because the behavior of the boids seems realistic enough without it. A boid using an angle component is only influenced by the boids within its field of vision. The general effect is that a flock is more affected by changes to boids near the front and less affected by changes toward the back.

The `cohesion()` method is shown below. It calculates a velocity that encourages the boid to fly toward 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.

A well-designed boids list data structure will reduce the overhead of the calculations; for example, one simple optimization is to utilize boid position in the list's ordering. A search algorithm using spatial information should find a nearby boid in constant time ($O(1)$), reducing the cost of a flock update to $O(m*n)$. For example, if there are 1,000 boids and 10 rules, then the current algorithm takes time proportional to $O(10^7)$, whereas the improved version would be $O(10^4)$, potentially a 1,000-fold improvement

In `cohesion()`, nearness is calculated by finding the absolute distance between the boid (`boidPos`) and each neighboring boid. The `PROXIMITY` constant can be adjusted to change the number of neighbors.

The choice of scaling factor (`COHESION_WEIGHT` in this case) is a matter of trial and error, determined by running the system with various values and observing the behavior of the flock. Part of the difficulty when deciding on a good value is that the rules interact with each other to produce an overall effect. For example, the cohesion rule brings boids together while the separation rules keep them apart. This interplay is what makes boid behavior hard to predict and so interesting.

The Prey's Behavior

`PreyBehavior` has three tasks:

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

Boid creation is done in `createBoids()`, called from the class'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 behavior with its BG
}
```

boidsBG is the inherited BranchGroup holding the boids in the scene; boidsBG’s capabilities must allow changes to its children so prey boids can be detached from it at runtime (after they’re eaten).

When a PreyBoid object is created, it’s passed a reference to the obstacles (passed to the behavior from WrapFlocking3D) and a reference to the behavior itself, so its velocity rules can be accessed.

createBoid() creates a scene branch such as that shown in Figure 22-8.

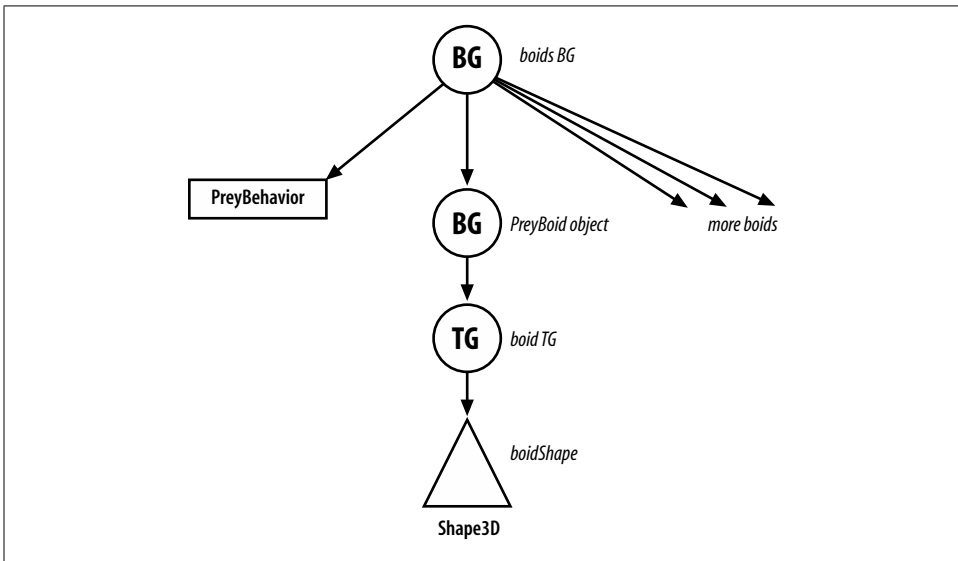


Figure 22-8. Scene branch for PreyBoid nodes

Velocity Rules and Other Flocks

PreyBehavior’s second task is complicated by the need to access the other flock (the predators), which means that it must reference the PredatorBehavior object. WrapFlocking3D delivers the reference via 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 predator flock for proximity. If a predator is within range, the velocity will be 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 in this code from earlier rules is the first line, where a reference to the predators list is obtained by calling getBoidsList() in PredatorBehavior. The prey examine this list to decide if they should start running.

Goodbye Prey

PreyBehavior contains an eatBoid() method, called by PredatorBehavior when the given boid has been eaten:

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

eatBoid() deletes a boid by detaching it from the scene graph, and removing it from the boids list.

The Predator's Behavior

PredatorBehavior has similar tasks to PreyBehavior:

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

Boid creation is almost identical to that done in `PreyBehavior`, except that `PredatorBoids` are created instead of `PreyBoids`. `PredatorBoid` has a method, `setPredBeh()`, which allows `WrapFlocking3D` to pass it a reference to the `PreyBehavior` object.

The velocity rule is implemented by `findClosePrey()`: the method calculates the average position of all the nearby `PreyBoids` and moves the predator a small step toward 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(); // get prey boid position
    // use pos to adjust the predator's velocity
}
```

Lunch Time

The third task for `PredatorBehavior`—eating nearby prey—isn't 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;
} // end of eatClosePrey()
```

The reference to `PreyBehavior` is used to get a link to the prey list and to call `eatBoid()` to remove the i^{th} boid. When a boid is removed, the next boid in the list becomes the new i^{th} boid, so the i index mustn't be incremented.