Boids Pseudocode

This is an explanation of the boids algorithm explained with the use of pseudocode. It is mostly the standard algorithm as described by Reynolds [1], with a few of my own tweaks thrown in. It should be enough to get you started with programming your own boids simulation and making up your own extra routines.

If you have any queries regarding this or have difficulty understanding any parts of the explanation, please contact me.

Overview

The boids program has the following structure:

```
initialise_positions()

LOOP
        draw_boids()
        move_all_boids_to_new_positions()
END LOOP
```

The `initialise_positions()` procedure puts all the boids at a starting position. I put them all at random locations off-screen to start with, that way when the simulation starts they all fly in towards the middle of the screen, rather than suddenly appearing in mid-air.

The `draw_boids()` procedure simply draws one 'frame' of the animation, with all the boids in their current positions. I won't explain the 3D perspective mapping here, this is described in any good computer graphics text (such as [2]). Note that the boids algorithm works just as well in two dimensions as it does in three, and makes for a nice easy way to start out.

The procedure I have called `move_all_boids_to_new_positions()` contains the actual boids algorithm. Note that all it involves is simple vector operations on the positions of the boids. Each of the boids rules works independently, so, for each boid, you calculate how much it will get moved by each of the three rules, giving you three velocity vectors. Then you add those three vectors to the boid's current velocity to work out its new velocity. Interpreting the velocity as how far the boid moves per time step we simply add it to the current position, arriving at the following pseudo-code:

```
PROCEDURE move_all_boids_to_new_positions()

        Vector v1, v2, v3
        Boid b

        FOR EACH BOID b
                v1 = rule1(b)
                v2 = rule2(b)
                v3 = rule3(b)

                b.velocity = b.velocity + v1 + v2 + v3
                b.position = b.position + b.velocity
        END

END PROCEDURE
```

We'll now look at each of these three rules in turn.

The Boids Rules

Rule 1: Boids try to fly towards the centre of mass of neighbouring boids.

The 'centre of mass' is simply the average position of all the boids. I use the term centre of mass by analogy with the corresponding physical formula (however we ignore individual masses here and treat all boids having the same mass).

Assume we have `N` boids, called b_1, b_2, ..., b_N. Also, the position of a boid `b` is denoted `b.position`. Then the 'centre of mass' `c` of all `N` boids is given by:

```
c = (b_1.position + b_2.position + ... + b_N.position) / N
```

Remember that the positions here are vectors, and `N` is a scalar.

However, the 'centre of mass' is a property of the entire flock; it is not something that would be considered by an individual boid. I prefer to move the boid toward its 'perceived centre', which is the centre of all the other boids, not including itself. Thus, for `boid_J (1 <= J <= N)`, the perceived centre `pc_J` is given by:

```
pc_J = (b_1.position + b_2.position + ... + b_J-1.position +
        b_J+1.position + ... + b_N.position) / (N-1)
```

Having calculated the perceived centre, we need to work out how to move the boid towards it. To move it 1% of the way towards the centre (this is about the factor I use) this is given by `(pc_J - b_J.position) / 100`.

Summarising this in pseudocode:

```
PROCEDURE rule1(boid b_J)

        Vector pc_J

        FOR EACH BOID b
                IF b != b_J THEN
                        pc_J = pc_J + b.position
                END IF
        END

        pc_J = pc_J / N-1

        RETURN (pc_J - b_J.position) / 100

END PROCEDURE
```

Thus we have calculated the first vector offset, `v1`, for the boid.

Rule 2: Boids try to keep a small distance away from other objects (including other boids).

The purpose of this rule is to for boids to make sure they don't collide into each other. I simply look at each boid, and if it's within a defined small distance (say 100 units) of another boid move it as far away again as it already is. This is done by subtracting from a vector `c` the displacement of each boid which is near by. We initialise `c` to zero as we want this rule to give us a vector which when added to the current position moves a boid away from those near it.

In pseudocode:

```
PROCEDURE rule2(boid b_J)

        Vector c = 0;

        FOR EACH BOID b
                IF b != b_J THEN
                        IF |b.position - b_J.position| < 100 THEN
                                c = c - (b.position - b_J.position)
                        END IF
                END IF
        END

        RETURN c

END PROCEDURE
```

It may seem odd that we choose to simply double the distance from nearby boids, as it means that boids which are very close are not immediately "repelled". Remember that if two boids are near each other, this rule will be applied to both of them. They will be slightly steered away from each other, and at the next time step if they are still near each other they will be pushed further apart. Hence, the resultant repulsion takes the form of a smooth acceleration. It is a good idea to maintain a principle of ensuring smooth motion. If two boids are very close to each other it's probably because they have been flying very quickly towards each other, considering that their previous motion has also been restrained by this rule. Suddenly jerking them away from each other, such that they each have their motion reversed, would appear unnatural, as if they bounced off each other's invisible force fields. Instead, we have them slow down and accelerate away from each other until they are far enough apart for our liking.

Rule 3: Boids try to match velocity with near boids.

This is similar to Rule 1, however instead of averaging the positions of the other boids we average the velocities. We calculate a 'perceived velocity', pv_J, then add a small portion (about an eighth) to the boid's current velocity.

```
PROCEDURE rule3(boid b_J)

        Vector pv_J

        FOR EACH BOID b
                IF b != b_J THEN
                        pv_J = pv_J + b.velocity
                END IF
        END

        pv_J = pv_J / N-1

        RETURN (pv_J - b_J.velocity) / 8

END PROCEDURE
```

That's all there is to it :) The three rules are fairly simple to implement.

Further tweaks

The three boids rules sufficiently demonstrate a complex emergent flocking behaviour. They are all that is required to simulate a distributed, leaderless flocking behaviour.

However in order to make other aspects of the behaviour more life-like, extra rules and limitations can be implemented.

These rules will simply be called in the `move_all_boids_to_new_positions()` procedure as follows:

```
PROCEDURE move_all_boids_to_new_positions()

        Vector v1, v2, v3, v4, ...

        FOR EACH BOID b
                v1 = rule1(b)
                v2 = rule2(b)
                v3 = rule3(b)
                 v4 = rule4(b)
                    .
                   .
                   .

                b.velocity = b.velocity + v1 + v2 + v3 + v4 + ...
                b.position = b.position + b.velocity
        END
```

```
END PROCEDURE
```

Hence each of the following rules is implemented as a new procedure returning a vector to be added to a boid's velocity.

Goal setting

Reynolds [1] uses goal setting to steer a flock down a set path or in a general direction, as required to ensure generally predictable motion for use in computer animations and film work. I have not used such goal setting in my simulations, however here are some example implementations:

Action of a strong wind or current

For example, to simulate fish schooling in a moving river or birds flying through a strong breeze.

```
PROCEDURE strong_wind(Boid b)
        Vector wind

        RETURN wind
END PROCEDURE
```

This function returns the same value independent of the boid being examined; hence the entire flock will have the same push due to the wind.

Tendency towards a particular place

For example, to steer a sparse flock of sheep or cattle to a narrow gate. Upon reaching this point, the goal for a particular boid could be changed to encourage it to move away to make room for other members of the flock. Note that if this 'gate' is flanked by impenetrable objects as accounted for in Rule 2 above, then the flock will realistically mill around the gate and slowly trickle through it.

```
PROCEDURE tend_to_place(Boid b)
        Vector place

        RETURN (place - b.position) / 100
END PROCEDURE
```

Note that this rule moves the boid 1% of the way towards the goal at each step. Especially for distant goals, one may want to limit the magnitude of the returned vector.

Limiting the speed

I find it a good idea to limit the magnitude of the boids' velocities, this way they don't go too fast. Without such limitations, their speed will actually fluctuate with a flocking-like tendency, and it is possible for them to momentarily go very fast. We assume that real animals can't go arbitrarily fast, and so we limit the boids' speed. (Note that I am using the physical definitions of *velocity* and *speed* here; velocity is a vector and thus has both magnitude and direction, whereas speed is a scalar and is equal to the magnitude of the velocity).

For a limiting speed `vlim`:

```
PROCEDURE limit_velocity(Boid b)
        Integer vlim
        Vector v

        IF |b.velocity| > vlim THEN
                b.velocity = (b.velocity / |b.velocity|) * vlim
        END IF
END PROCEDURE
```

This procedure creates a unit vector by dividing `b.velocity` by its magnitude, then multiplies this unit vector by `vlim`. The resulting velocity vector has the same direction as the original velocity but with magnitude vlim.

Note that this procedure operates directly on `b.velocity`, rather than returning an offset vector. It is not used like the other rules; rather, this procedure is called after all the other rules have been applied and before calculating the new position, ie. within the procedure `move_all_boids_to_new_positions`:

```
b.velocity = b.velocity + v1 + v2 + v3 + ...
limit_velocity(b)
b.position = b.position + b.velocity
```

Bounding the position

In order to keep the flock within a certain area (eg. to keep them on-screen) Rather than unrealistically placing them within some confines and thus bouncing off invisible walls, we implement a rule which encourages them to stay within rough boundaries. That way they can fly out of them, but then slowly turn back, avoiding any harsh motions.

```
PROCEDURE bound_position(Boid b)
        Integer Xmin, Xmax, Ymin, Ymax, Zmin, Zmax
        Vector v

        IF b.position.x < Xmin THEN
                v.x = 10
        ELSE IF b.position.x > Xmax THEN
                v.x = -10
        END IF
        IF b.position.y < Ymin THEN
                v.y = 10
        ELSE IF b.position.y > Ymax THEN
                v.y = -10
        END IF
        IF b.position.z < Zmin THEN
                v.z = 10
        ELSE IF b.position.z > Zmax THEN
                v.z = -10
        END IF

        RETURN v
END PROCEDURE
```

Here of course the value `10` is an arbitrary amount to encourage them to fly in a particular direction.

Perching

The desired behaviour here has the boids occasionally landing and staying on the ground for a brief period of time before returning to the flock. This is accomplished by simply holding the boid on the ground for a breif period (of random length) whenever it gets to ground level, and then letting it go.

When checking the bounds, we test if the boid is at or below ground level, and if so we make it perch. We introduce the Boolean `b.perching` for each boid `b`. In addition, we introduce a timer `b.perch_timer` which determines how long the boid will perch for. We make this a random time, assuming we are simulating the boid eating or resting.

Thus, within the `bound_position` procedure, we add the following lines:

```
        Integer GroundLevel

        ...

        IF b.position.y < GroundLevel THEN
                b.position.y = GroundLevel
                b.perching = True
        END IF
```

It is held on the ground by simply not applying the boids rules to its behaviour (obviously, as we don't want it to move). Thus, before attempting to apply the rules we check if the boid is perching, and if so we

decrement the timer `b.perch_timer` and skip the rest of the loop. If the boid has finished perching then we reset the `b.perching` flag to allow it to return to the flock.

```
PROCEDURE move_all_boids_to_new_positions()

        Vector v1, v2, v3, ...
         Boid b

        FOR EACH BOID b

                IF b.perching THEN
                        IF b.perch_timer > 0 THEN
                                b.perch_timer = b.perch_timer - 1
                                NEXT
                        ELSE
                                b.perching = FALSE
                        END IF
                 END IF


                v1 = rule1(b)
                v2 = rule2(b)
                v3 = rule3(b)
                 ...

                b.velocity = b.velocity + v1 + v2 + v3 + ...
                 ...
                b.position = b.position + b.velocity
        END

END PROCEDURE
```

Note that nothing else needs to be done to simulate the perching behaviour. As soon as we re-apply the boids rules this boid will fly directly towards the flock and continue on as normal.

A detail I implement here is that the lower bound for the boids' motion is actually a little above ground level. That way the boids are actually discouraged from going too near the ground, and when they do go to the ground they land gently rather than ploughing into it as there is an upward push from the bounding rule. They also land less often which stops them becoming too lazy.

Anti-flocking behaviour

During the course of a simulation, one may want to break up the flock for various reasons. For example the introduction of a predator may cause the flock to scatter in all directions.

Scattering the flock

Here we simply want the flock to disperse; they are not necessarily moving away from any particular object, we just want to break the cohesion (for example, the flock is startled by a loud noise). Thus we actually want to *negate* part of the influence of the boids rules.

Of the three rules, it turns out we only want to negate the first one (moving towards the centre of mass of neighbours) -- ie. we want to make the boids move away from the centre of mass. As for the other rules: negating the second rule (avoiding nearby objects) will simply cause the boids to actively run into each other, and negating the third rule (matching velocity with nearby boids) will introduce a semi-chaotic oscillation.

It is a good idea to use non-constant multipliers for each of the rules, allowing you to vary the influence of each rule over the course of the simulation. If you put these multipliers in the `move_all_boids_to_new_positions` procedure, ending up with something like:

```
PROCEDURE move_all_boids_to_new_positions()

        Vector v1, v2, v3, ...
         Integer m1, m2, m3, ...
```

```
        Boid b

        FOR EACH BOID b

                ...

                v1 = m1 * rule1(b)
                v2 = m2 * rule2(b)
                v3 = m3 * rule3(b)
                ...

                b.velocity = b.velocity + v1 + v2 + v3 + ...
                ...
                b.position = b.position + b.velocity
        END

END PROCEDURE
```

then, during the course of the simulation, simply make `m1` negative to scatter the flock. Setting `m1` to a positive value again will cause the flock to spontaneously re-form.

Tendency away from a particular place

If, on the other hand, we want the flock to continue the flocking behaviour but to move away from a particular place or object (such as a predator), then we need to move each boid individually away from that point. The calculation required is identical to that of moving towards a particular place, implemented above as `tend_to_place`; all that is required is a negative multiplier:

```
Vector v
Integer m
Boid b

...

v = -m * tend_to_place(b)
```

So we see that each of the extra routines are very simple to implement, as are the initial rules. We achieve complex, life-like behaviour by combining all of them together. By varying the influence of each rule over time we can change the behaviour of the flock to respond to events in the environment such as sounds, currents and predators.

Auxiliary functions

You will find it handy to set up a set of Vector manipulation routines first to do addition, subtraction and scalar multiplication and division. For example, all the additions and subtractions in the above pseudocode are vector operations, so for example the line:

pc_J = pc_J + b.position

will end up looking something like:

pc_J = Vector_Add(pc_J, b.position)

where `Vector_Add` is a procedure defined thus:

```
PROCEDURE Vector_Add(Vector v1, Vector v2)

        Vector v

        v.x = v1.x + v2.x
        v.y = v1.y + v2.y
        v.z = v1.z + v2.z

        RETURN v
```

```
    END PROCEDURE
```

and the line:

$$pc_J = pc_J / N-1$$

will be something like:

$$pc_J = Vector\_Div(pc_J, N-1)$$

where `Vector_Div` is a scalar division:

```
    PROCEDURE Vector_Div(Vector v1, Integer A)

        Vector v

        v.x = v1.x / A
        v.y = v1.y / A
        v.z = v1.z / A

        RETURN v

    END PROCEDURE
```

Of course if you're doing this in two dimensions you won't need the z-axis terms, and if you're doing this in more than three dimensions you'll need to add more terms :)

References

[1] Craig W. Reynold's home page, *http://www.red3d.com/cwr/*
[2] *Computer Graphics, Principles and Practice* by Foley, van Dam, Feiner and Hughes, Addison Wesley 1990

Return to the main Boids page.