Flocking Over 3D Terrain

Joel Gompert
Constraint Systems Laboratory
Department of Computer Science and Engineering
University of Nebraska-Lincoln
jgompert@cse.unl.edu

December 11, 2003

Abstract

A method is presented for animating herds of animals that can follow terrain while being efficient enough to run in real-time. This method involves making simple modifications to Reynolds' agent-based flocking algorithm. The modifications use only local properties of the terrain, and thus have low complexity. This method focuses on using terrain that can be described as an elevation grid, but it may be extendible to arbitrary terrain. The flocking algorithm with these modifications produces naturally behaving herds that follow the terrain. They will swerve around hills and attempt to follow paths that reduce energy expenditure. The terrain-following rule added to the flocking algorithm has a constant parameter that can be adjusted to produce different behaviors. Empirical analysis shows how this parameter affects energy expenditure of the animals while traveling over the terrain.

Acknowledgments: Thanks to my advisor, Dr. Berthe Y. Choueiry. Thanks to Christopher Kline [5] for a sample flocking implementation which I occasionally referred to while creating my own implementation. This work is supported by the CAREER Award #0133568 from the National Science Foundation.

Contents

List of Tables

List of Figures

1 Introduction

To animate a large number of objects by hand would be a tedious job. To make the job easier, we would like to try to automate as much of the process as possible. For the case of flocks or herds of animals, Reynolds introduced a simple agent-based approach to animate a flock of animals through space [10]. In this method, each animal makes its own decisions on how to move, according to a small number of simple rules that consider the neighboring members of the flock. Reynolds suggested that further modifications could lead to a herd model by giving the animals the ability to follow terrain.

The challenge is how to give the animals the ability to follow terrain while keeping the complexity low enough to still allow for real-time simulation of the herd. We would like the herd to move in natural-looking paths. Also, we would like the animals to travel fastest on flat ground, more slowly when navigating downhill and slowest when moving uphill. We present here a simple method to satisfy all of these requirements, by modifying Reynolds' basic flocking algorithm. The modifications use only local properties of the terrain to minimize complexity, namely the gradient and the heigt of the terrain. This method only considers terrain that can be described as an elevation grid, in other words, terrain that is given by a function $h(x, y)$ that returns the height at each horizontal position. However, it may be

simple to extend this method to be applicable to arbitrary terrain. This is an area for future work, as described in Section 11.

Section 2 discusses physical simulation. Section 3 gives a short description of Reynolds' flocking algorithm, followed by an explanation of the modifications that we need to make to this algorithm. Sections 4, 5, and 6 discuss the main behaviors that this paper focuses on, which are goal seeking, terrain following, and obstacle avoidance, respectively. Section 7 discusses the time-complexity of the algorithm, and Section 8 discusses some implementation issues.

In our implementation of the algorithm, we identify several parameters that allow us to control the behavior of the flock. Section 9 describes these parameters. Sections 10 and 11 discuss the results and future work respectively.

2 Simulation basics

Simulating the behavior of an animal in the flock amounts to updating the animal's position (Eq 1) and velocity (Eq 2) at each time step Δt. This applies not only to our animals but to any moving physical body we are simulating.

$$\vec{p} \longleftarrow \vec{p} + \vec{v} \cdot \Delta t + \frac{1}{2}\vec{a} \cdot (\Delta t)^2 \tag{1}$$

$$\vec{v} \longleftarrow \vec{v} + \vec{a} \cdot \Delta t \tag{2}$$

In these equations, $\vec{p}$ is the position, $\vec{v}$ is the velocity, $\vec{a}$ is the acceleration, and Δt is the elapsed time between updates. These equations constitute a forward Euler approximation to a set of differential equations. We cannot simply find the explicit solution to the differential equations because the acceleration $\vec{a}$ will be a function of the positions and velocities of other similar dynamic bodies in the animal's environment. When we have multiple bodies interacting in this fashion, then the differential equations become unsolvable. However, by using this simple Euler approximation, then the calculation remains the same, no matter how complex the system becomes as we add more bodies. Also note that, in Equation 1, if one cannot afford the time it takes to compute the last term involving the acceleration, one may simply ignore it in most situations. Typically, in a real time simulation, we want to animate the simulation at some frame rate, say 30 frames per second or higher. So Δt should be small, and the $(\Delta t)^2$ term may be negligible depending on the application. This term becomes significant only for large accelerations. Thus, there is a tradeoff here between speed and accuracy.

Now we want to be able to determine the acceleration. The acceleration is going to be a result of forces exerted by the animal, based on its decisions on how to move, as well as outside forces acting on it. At each time step,

we want to update the acceleration, based on these forces, according to Newton's law:

$$\vec{a} \longleftarrow \frac{\vec{F}}{m}, \tag{3}$$

where $\vec{F}$ is the net force and m is the mass of the body. Each time through the update loop of our program, we merely need to, for each body, calculate the net force acting on it, and update the acceleration, position, and velocity of the body using these three motion formulas. Now, the only difference between an animal and an inanimate object is that for an animal, at least some of the forces acting on the animal originate from the animal itself, while inanimate objects react only to outside influences.

It should be noted that this forward Euler method of integration is not the only possible method. We chose Forward Euler for its simplicity. Integration methods fall into one of two categories: implicit and explicit. Forward Euler is an example of an explicit method. There are more complex explicit methods that result in more accurate results, however it turns out that all explicit methods become unstable when Δt becomes too large, while implicit methods (such as reverse Euler) remain stable. The downside to using implicit methods is that they make the math much more complicated. In addition, implicit methods require explicit knowledge of the derivatives of the forces on each body. This becomes quite complex when one has mul-

tiple bodies whose derivatives depend on each other. This requires solving a system of equations. So, in this implementation, we use forward Euler in order to maintain simplicity. Our observations showed that the simulation remained stable as long as Δt remained below approximately 100ms.

3 Reynolds' Flocking Algorithm

The animals ('boids') in Reynolds' flocks move according to three simple rules. Each of these rules results in an acceleration. The accelerations from the various behavioral rules can be combined in various ways and used to animate the animals. These three simple rules can produce a natural-looking flock behavior:

1. *Separation*: accelerating away from nearby members of the flock that are closer than some threshhold distance,

2. *Alignment*: adjusting an animal's velocity to match the average velocity of its neighbors, and

3. *Cohesion*: accelerating towards the center of mass of the visible flockmates.

Each behavior here is executed for each animal individually.

Reynolds also describes a few other possible behaviors that could be added, that we will also use. These are *obstacle avoidance* and *goal seeking*. Obstacle avoidance produces an acceleration to steer around obstacles, and goal seeking produces an acceleration in the direction of some goal location. There are other possibilities for behaviors.[1] For our purposes we will use the following set of behaviors: (1) Separation, (2) Alignment, (3) Cohesion, (4) Obstacle avoidance, and (5) Goal seeking. In Section 3.1 we will begin to discuss obstacle avoidance, and in Section 4, we will discuss goal-seeking. We will also make some additions to allow for following the terrain, which are explained in Section 5.

3.1 Obstacle avoidance

Reynolds discusses various methods of obstacle avoidance [9], which we will not discuss here in detail. Reynolds divides the methods into three groups, those based on *geometric models*, those based on *image processing*, or those based on *geometrical path-planning*. The methods are listed below:

- Obstacle Avoidance Techniques Based on Geometric Models:

    1. repulsive force field,

    2. steer away from center,

[1] For example, Flake discusses another behavior that he calls view clearing [2].

3. steer along surface ("curb feeler"), and

4. steer towards nearest silhouette edge.

- Obstacle Avoidance Techniques Based on Image Processing:

1. fuzzy silhouette,

2. obstacle density image, and

3. z-buffer image.

For this implementation we used a method similar to 'steer away from center,' that will be discussed in Section 6.

3.2 Implementation

In the implementation of a physical simulation, one must choose the size of the time-step Δt. One option is to choose a fixed Δt and then synchronize the simulation to the system clock using a delay when an execution of the main loop of the simulation takes less time than Δt. This solution will cause problems if it is executed on a machine that takes longer than Δt to go once through the main loop. Another option is to let the main loop run as often as it can, and each time through, set Δt to the amount of time that has elapsed since the previous execution of the main loop, according to the system clock.

In this main loop, we need to update each body's acceleration, position, and velocity according to our motion equations (1) and (2), and the frame of animation needs to be drawn. For this implementation, every animal is updated every frame. However, in order to speed up the frame-rate in a real-time simulation, one may update some fraction of the herd each frame. For example, one may wish to update half of the herd at a given frame and the other half at the next frame. In that case Δt for each animal becomes the amount of time since the last time that animal's behavior was updated. While, the position and velocity of each animal should be updated every frame, it may be benificial to divide the behavioral updates to acceleration over multiple frames in this manner.

4 Goal seeking

We experimented with a goal-seeking behavior. First we tried a simple attractive force pulling towards the goal in the form

$$\vec{F} \longleftarrow K(\vec{p_g} - \vec{p_a}) \tag{4}$$

where $\vec{p_g}$ is the position of the goal point, $\vec{p_a}$ is the position of the animal, and K is an arbitrary constant. This gives a force in the direction of the goal, proportional to the distance to the goal. Along with the scaling constant K,

$\vec{F}$ can be clipped off at a maximum magnitude. Alternatively, instead of a linear function, one can use a constant magnitude, quadratic, exponential, or a more complex function.

This force towards the goal did not, however, provide the desired results, because it tends to draw the boids into a full-speed flight towards the goal. This situation prevents the boids in the back from being able to catch up with the ones in front of the herd, because the ones in front are at the maximum speed.

We propose here a novel and improved goal-seeking behavior that seems to solve this issue. Instead of the boid's position being attracted to the goal, the *boid's velocity is attracted towards a velocity pointing in the direction of the goal.* This target velocity is defined as the *cruising speed* C_s. This cruising speed is a constant that is less than the maximum speed. Define the target cruising *velocity* $\vec{v_g}$ as:

$$\vec{v_g} \longleftarrow C_s \frac{\vec{p_g} - \vec{p_a}}{||\vec{p_g} - \vec{p_a}||}. \tag{5}$$

Now create a force pointing from the current velocity towards the desired velocity.

$$\vec{F} \longleftarrow K(\vec{v_g} - \vec{v}) \tag{6}$$

Thus, instead of trying to get to the goal as quickly as possible, the

boids travel at some slower cruising velocity, allowing boids to occasionally go faster when necessary, as when stragglers need to catch up. The resulting behavior seems more realistic as well, since a flock or herd should travel at some cruising speed, not the maximum speed, unless emergency conditions arise.

5 Terrain following: from flocks to herds

In this section we will discuss the modifications to the flocking algorithm necessary to create herd behavior. First, the animals must be confined to a two-dimensional space, namely the surface of the terrain, which curves through 3-space. In the approach presented here, we use a two-dimensional flocking algorithm, and the terrain is modeled as a height field. So, effectively the animals are flocking on a two dimensional plane that has, at every point, a 'height' property, which the animals use to adjust their behavior. Now, although everything is modeled internally as being two-dimensional, when rendering the animation, the animals are still drawn at their correct height on the terrain.[2]

[2]In this implementation, the boids always adjust their position to be a constant height above the surface of the terrain. A future implementation may allow more complex behavior (e.g. jumping or falling).

In this model, the horizontal plane is (x, y) and the height of the terrain at each point is $z = h(x, y)$. We also define the gradient $\vec{g}(x, y)$ of the height at each point on the terrain. The gradient of $h(x, y)$ is a vector whose components are the function's partial derivatives, or $\vec{g}(x, y) = (\frac{dh}{dx}, \frac{dh}{dy})$, evaluated at the point (x, y). The gradient $\vec{g}$ gives us a vector pointing in the direction of the most rapid increase of height, thus pointing in the direction of 'uphill'. The magnitude of the gradient is the slope of the terrain in that direction, which we will denote as $s = ||\vec{g}||$. At a local maximum or minimum of the terrain's height, and on perfectly level terrain, the gradient is exactly zero $(0, 0)$. At the exact point of a sheer cliff, there is a discontinuity, and the gradient at this point is undefined. For this paper we assume we have some efficient way of computing the gradient at any point on the terrain. This is normally simple to compute. For example, if the normal vector of the terrain at point p is the vector $\vec{n} = (n_x, n_y, n_z)$, then the gradient at p is $\vec{g}(p) = (-n_x/n_z, -n_y/n_z)$, and the slope of the terrain in the direction of the gradient is

$$s = ||\vec{g}|| = \sqrt{(\frac{-n_x}{n_z})^2 + (\frac{-n_y}{n_z})^2} = \frac{\sqrt{{n_x}^2 + {n_y}^2}}{n_z}. \tag{7}$$

5.1 3D Correction

Since the animals are confined to a two-dimensional surface, we use a two-dimensional flocking algorithm. Local properties of the terrain such as height and gradient are used to apply forces to the animals or otherwise modify the two-dimensional flocking algorithm.

Using a 2D flocking algorithm requires a correction to be applied to each animal's velocity because an animal with a constant 2D (x, y) velocity traveling over hills will be moving through 3-space faster when going uphill or downhill than when moving across level terrain. However, this is not the effect we are seeking. This problem can be fixed by applying a correcting factor to the 2D velocity before using it to update the animal's position each frame. This correcting factor takes into account the slope of the terrain in the direction of travel, which is $s_v = \frac{\vec{v} \cdot \vec{g}}{||\vec{v}||}$, where $\vec{v}$ is the 2D velocity and $\vec{g}$ is the gradient.

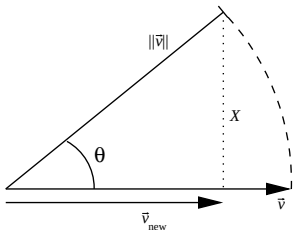

Figure 1: *Rotating velocity by the slope.*

Figure 1 illustrates how we want the velocity to be rotated to point up (or down) the slope of the terrain. The goal is to keep the final 3D velocity correct for a given $\vec{v}$. In this figure, the angle θ corresponds to the slope of the terrain in the direction of travel, and $\vec{v}_{new}$ is the corrected 2D velocity, given the slope. We can solve for $\vec{v}_{new}$ as a function of $\vec{v}$ and the slope s_v.

$$X = \tan\theta ||\vec{v}_{new}|| = s_v ||\vec{v}_{new}|| \tag{8}$$

$$||\vec{v}_{new}||^2 + X^2 = ||\vec{v}||^2 \tag{9}$$

$$\vec{v}_{new} = \frac{\vec{v}}{\sqrt{{s_v}^2 + 1}} \tag{10}$$

Thus, when updating the animal's position, instead of simply updating the 2D position by the velocity $\vec{v}$, we must first multiply $\vec{v}$ by the correcting factor $\frac{1}{\sqrt{{s_v}^2+1}}$.

Note that, to be more accurate, the same correcting factor needs to be applied to acceleration, except that the slope in the direction of the acceleration must be used (s_a). However, applying the correction factor to acceleration does not produce a noticeable difference.

Applying this correcting factor allows us to make the animals' motion over the terrain to look much more natural. In particular it causes the 2D

velocity to be translated correctly to a 3D velocity.[3]

5.2 Natural Behavior Over 3D Terrain

The application of the correcting factor of Section 5.1 will not change the animals' overall 2D routes as they travel over the terrain. At this point, the herd will travel straight over hills or whatever terrain features are in its way as in Path 1 in Figure 2. However, we do not usually want the herd to go straight over the top of a hill, but we would like it to swerve in a natural-looking route around the hill such as Path 2 or 3. In order to get this more natural-looking behavior, we want to add to the basic flocking algorithm a new behavior rule that takes the terrain into account.

5.2.1 Natural Routes

One possibility is to pre-compute the 'best' route. There are many articles on computing shortest routes over terrain. For example, Kapoor describes a method for computing the geodesic shortest path across the surface of a polygonal mesh such as terrain [4]. The shortest path is not necessarily what we would like. The shortest path in Figure 2 is closer to Path 2 than to Path 1 or 3. However, we may prefer that the animals follow a path closer

[3]Note that this does not mean that the animals will always travel at a constant speed. It simply guarantees that the animated speed of the animal will match the expected speed.

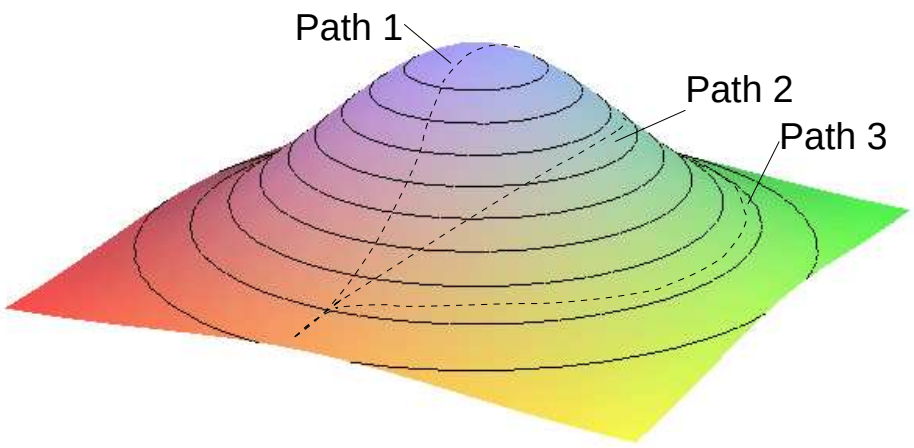

Figure 2: *Possible paths to get past a hill.*

to Path 3.

Although the shortest path would possibly be best for robots or vehicles when time is the most important factor, animals and humans naturally choose behaviors that minimize energy expenditure. For example, different gaits are employed because they use less energy at different speeds. Running is more efficient than walking at higher speeds, and walking is more efficient than running at slower speeds [7]. Similarly, it seems reasonable that animals would choose paths that minimize their energy expenditure.

Also, in comparison to the field of general relativity, a body in space follows a geodesic shortest path in the space-time continuum, without doing any shortest-route planning. It merely follows, at all times, a straight line

in the locally flat space-time. It seems reasonable, then, that the animals in the herd can follow a reasonable path by looking only at the properties of the locally flat terrain, i.e. the gradient at the current position. Admittedly, this is a simplification since, in reality, animals could see a hill in front of them and react to it before the animal feels the change in gradient beneath its feet. However, this seems to be an acceptable approximation.

5.2.2 Energy Expenditure

From the above discussion, it becomes clear that we need to consider how the terrain affects energy expenditure. Naturally, walking on level terrain uses less energy than walking uphill, and any hiker knows that walking down a steep hill also uses more energy than walking on level terrain.

Generally, to conserve energy, the herd should try to follow contour lines to avoid traveling up or down. However this is not a hard rule, because energy expended increases with distance traveled, so following a contour line around a wide, short hill might make the distance to travel so long that a more direct route over the hill would be more economical. Looking at Figure 2, Path 3 has less vertical travel than Path 2, but 3 is a longer path. Also starting and goal points may often be at different altitudes, or have a ridge or valley between them. When it becomes necessary to get to the

other side of a ridge, the herd will want to pass over the lowest point of the ridge without going too far out of its way.

5.2.3 Drag Force

My solution to this problem is to have the terrain create a force on the animals. This force resists any motion that is not following the contour lines. The force should increase with the magnitude of the gradient at the current position. For example, to deflect the animals from going straight over a steep hill, there should be a force acting on the animals in the direction of 'downhill' (i.e., in the opposite direction of the gradient). However, if the animals are already following a contour line (traveling orthogonally to the gradient), then we need to avoid pushing them downhill, away from the contour line, so the force should be proportional to the animal's component of velocity in the direction of the gradient, as shown in Figures 3 and 4. This can be compared to drag as a result of moving through a viscous fluid. Drag is a force acting in the direction opposite that of velocity, and is proportional to the magnitude of velocity as in $\vec{F_d} = -B\vec{v}$. So, this force from the terrain can be thought of as a drag force that acts only parallel to the gradient (or orthogonal to the contour lines). And the drag coefficient B grows with the magnitude of the gradient.

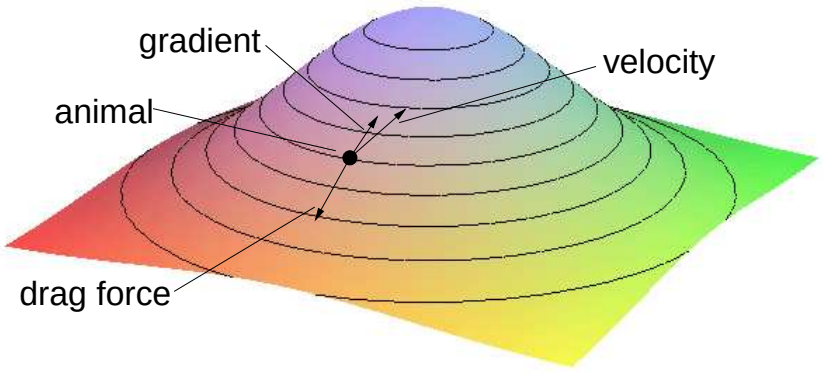

Figure 3: *Drag force while traveling uphill*

Notice that the velocities in both figures, Figure 3 and 4, have the same magnitude, however they produce different drag forces because only the component of the velocity perpendicular to the contour lines is used when computing this drag force. The result of this force is to deflect the herd's motion away from the direction of the gradient, and to deflect it more strongly as the terrain becomes steeper. So, the herds will take a more direct route when moving over a wide, short hill, and will swerve more widely around a steeper hill.

The question now is how this drag coefficient B should be computed as a function of slope. B should increase as slope increases, but how? A linear relationship seems reasonable, but can this be justified? Consider the effect gravity has on bodies on the surface of the terrain.

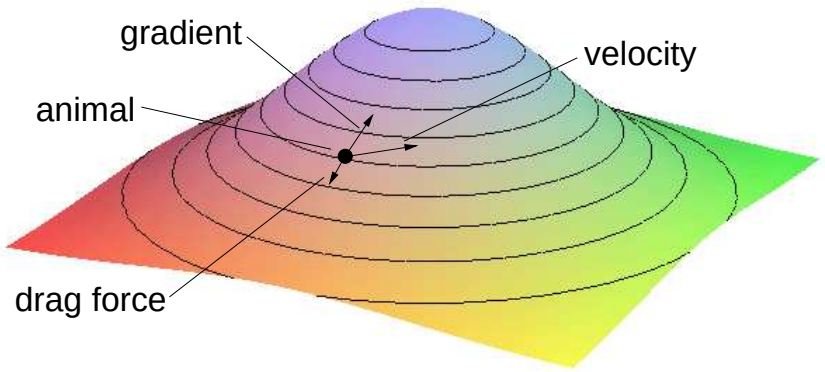

Figure 4: *Drag force while following the contour more closely*

On an inclined plane as shown in Figure 5, if gravity exerts a constant force $\vec{F}$ downward on a body, it will produce a force parallel to the inclined plane, and one perpendicular to it, of magnitudes $||\vec{F}|| \sin\theta$ and $||\vec{F}|| \cos\theta$ respectively, where θ is the angle the inclined plane makes with the horizontal. The slope of the plane is $s = \tan(\theta)$.

It seems reasonable that there are two things that affect the speed at which animals will travel up or down hill: (1) the extra energy required to travel up or down a slope and (2) less traction friction on a steeper slope which will make an animal more cautious and want to travel more slowly.

The parallel force is proportional to the effort that is required to ascend the plane However, the amount of traction on the hill is proportional to the perpendicular, or normal, force. The animal's ability to go up the hill

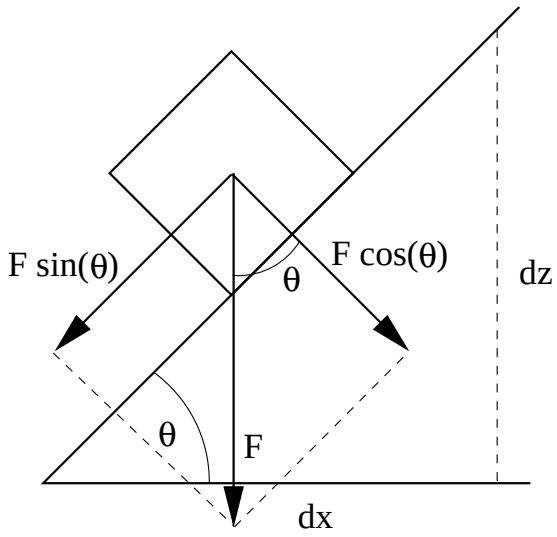

Figure 5: *Gravity's influence on an object on an inclined plane*

should be inversely proportional to the amount of traction, thus inversely proportional to the normal force.

So, if ability to travel up the inclined plane is directly proportional to the parallel force and inversely proportional to the perpendicular force, then, introducing some arbitrary constant K, the aversion to going up the hill should be in this form:

$$K\frac{||\vec{F}||\sin(\theta)}{||\vec{F}||\cos(\theta)} = K\tan(\theta) = Ks. \tag{11}$$

Thus it seems reasonable to use a linear function of the slope for the coefficient of our drag force and so we make this assignment:

$$B = Ks. \tag{12}$$

The fact that the drag coefficient B goes to infinity as the slope goes to infinity also seems reasonable.

So, to summarize, drag force is normally a force proportional to velocity, or $\vec{F} = -B\vec{v}$. However we want the drag force to act only parallel with the gradient, so we take the dot product of velocity with the normalized gradient, thus

$$\vec{F} = -B\frac{\vec{v}\cdot\vec{g}}{||\vec{g}||^2}\vec{g} \tag{13}$$

where $\vec{g}$ is the gradient. Now in Equation (12), we said that the coefficient B is Ks, but we know $s = ||\vec{g}||$. So the drag force becomes

$$\vec{F} = -Ks\frac{(\vec{v}\cdot\vec{g})\vec{g}}{||\vec{g}||^2} = -K\frac{(\vec{v}\cdot\vec{g})\vec{g}}{||\vec{g}||}. \tag{14}$$

And we define the force to be zero if the magnitude of the gradient is zero.

Now, the only question that remains to be answered is what value to use for the multiplying constant K. We implemented all the modifications to the flocking algorithm described thus far. We then performed tests, with the herds traveling from one location to a goal point located some distance away using a simple goal-seeking behavior. We let the constant K vary from 0 to 75, and for each value we had the herds travel over smooth, randomly-generated terrains and calculated an estimate for the average amount of energy expended over the trip. For each value of the constant, we gathered

data from about 300 trips in order to obtain a reliable mean. For each trip, I computed the energy expended for each member of the herd, and took the average over all the members of the herd when they reached the goal. (Reaching the goal was defined as the average position of the herd being within a certain arbitrary allowable error radius ϵ of the goal.) The results of this are shown in the graph of Figure 6.

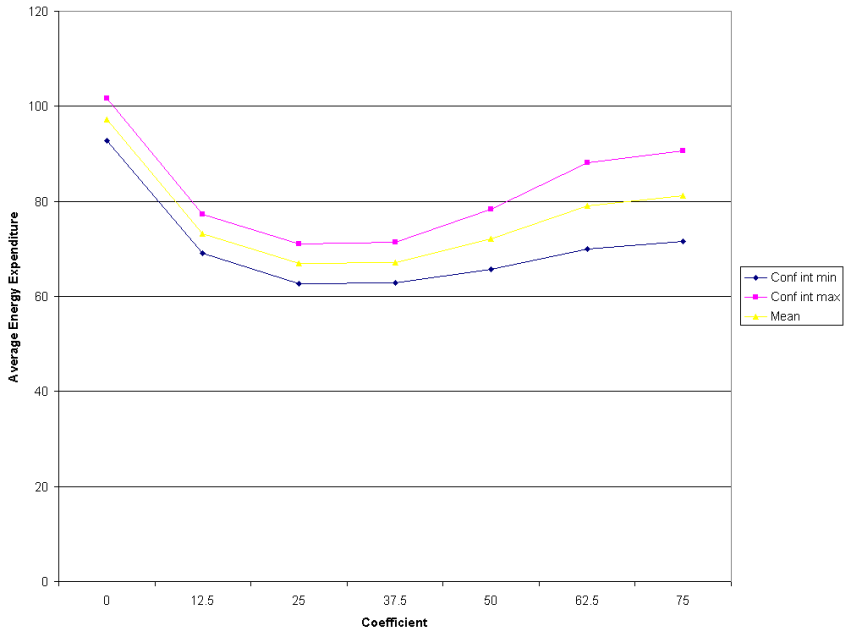

Figure 6: *Energy expenditure as a function of the coefficient K*

The energy expenditure for each animal was computed by determining, at each time step, the amount of energy expended at that time step, and summing over the entire trip. For a given time step, the energy expended

was computed as a function of distance traveled and slope of the terrain in the direction of travel. The energy expended should be linear with distance traveled in a small time step.

It has been found that the most economical slope (at least for humans) is downhill at about a -10% slope [6]. Thus our energy function should reach a minimum around there. As the slope increases to infinity, the required energy should have an asymptote at some (finite) maximum value, since traveling straight up (an infinite slope) should only require a finite amount of energy, just as climbing a vertical ladder requires a finite amount of energy. And, similarly, as the slope decreases to negative infinity, the required energy should have an asymptote at another, smaller, maximum value for downhill travel, just as climbing down a vertical ladder requires a finite amount of energy less than that of climbing up the ladder. This required energy as a function of slope is illustrated in Figure 7. This figure shows energy expenditure per distance traveled per unit of mass of the animal. The maximum at the left was placed at 1/5 of the maximum at the right because it has been found that downward breaking action expends about 1/5 as much energy as upward motion [6].

By measuring an average energy expenditure for various values for the constant K, we can find values that result in low energy expenditure. The

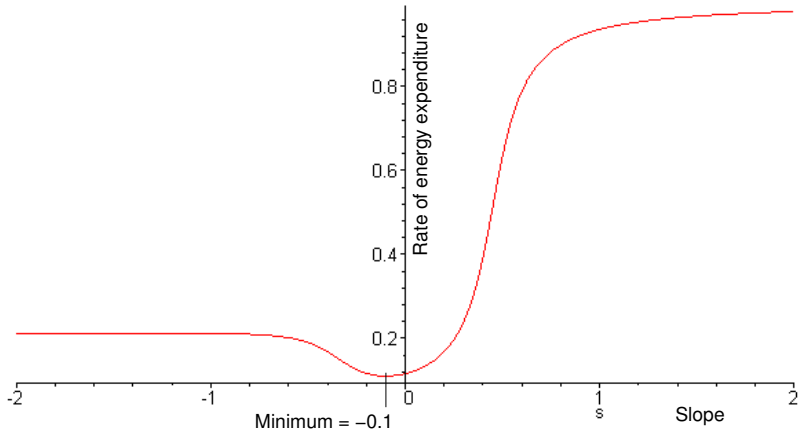

Figure 7: *Energy expenditure as a function of slope.*

value zero for the constant would eliminate any effect the terrain has on the animals, and the herd will travel in a straight line over the terrain. It would seem reasonable that increasing the value above zero, would decrease the energy expenditure, up to a point. Above some point, the animals would be constrained to the contour lines so much that they would be taking much longer paths than necessary and thus would be expending too much energy. Above some higher value, the animals would no longer be able to accelerate themselves fast enough to climb hills, and they might never be able to reach their goal, making the energy expenditure measure infinite. This possibility of becoming 'stuck' when K is too large necessitated inserting a limit in the experimental runs. Either there needs to be a time limit or an expended-energy limit, beyond which the herd is assumed to be stuck. The results for

my implementation are shown in the chart of Figure 6.

Of course it is not necessary to always choose the value that minimizes the energy expenditure. In fact, different values may be chosen for different desired animal behaviors. Slow moving cattle, for example, may choose to stick close to the contour lines, while energetic mountain goats may prefer to hop over the top of a steep hill. A single animal may even have different values for K at different times depending on its mood or situation. Also note that different values may be used for negative and positive slope. A larger K for positive slopes causes the animals to travel more slowly uphill than downhill.

6 Avoiding trees

Avoiding obstacles such as trees on the terrain can be implemented as an obstacle avoidance flocking behavior mentioned in Section 3.1. For sake of comparison, I designed, implemented and evaluated two different methods. The first method, discussed in Section 6.1, was a simple repulsive force pointing away from the trees. The second method, discussed in Section 6.2, was a steering force towards the nearest silhouette edge. This second method produced more realistic-looking results. Experiments here involved only small objects which might represent objects like trees or boulders. Future

work might involve larger objects like a long fence. More complex objects such as large concave objects would require more complicated algorithms to navigate.

6.1 Repulsive Force

The flock did not appear to behave naturally when modeling the trees as repulsive forces. The force should fall off as the distance to the tree increases (possibly an inverse square relation similar to forces in nature). Thus, there is some radius from the tree where the force becomes significant. If the flock heads straight towards a tree at cruising speed, instead of swerving or splitting around the tree, it seems to hit this invisible radius around the tree and bounces backwards before going around the tree. The effect is similar to what it might be like to drop a handful of bouncy balls onto a metal cylinder. They would all bounce back and outward when colliding with the cylinder, before continuing on their way.

6.2 Steering Towards the Nearest Silhouette Edge

This method produced much more natural-looking behavior, but may not work correctly for areas with dense collections of obstacles. A simple approximate way to accomplish this is illustrated in Figure 8 in two dimensions for

convex polygonal obstacles. First, each vertex of the polygon is projected onto the line perpendicular to the velocity, shown as a dotted line in the figure. Of the projected values, we determine the maximum and minimum values. If they are both positive or both negative, then the obstacle is to one side or the other, so we can safely ignore it. Of the maximum and minimum values, the one with the smaller absolute value is the nearest silhouette edge. The animal should steer towards this edge, or better, past the edge by an distance at least the radius of the animal in order to clear the obstacle.

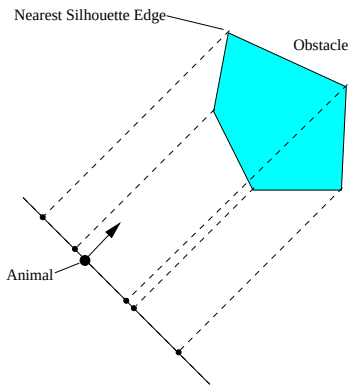

Figure 8: *Projection of an obstacle*

Before doing this check, one can reduce computation by first eliminating obstacles that lie entirely behind the animal, since they are irrelevant. This can be done while filtering out all obstacles that are not within the animal's range of vision.

This method for steering around obstacles works for most cases. How-

ever, it might fail in some situations when an animal is too close to an obstacle, but usually by the time the animal gets that close to an obstacle, it has already steered towards the nearest silhouette edge, thus avoiding the troublesome situations.

A possible improvement to this method can be made by taking into account the direction to the goal in addition to just the current direction of travel.

Figure 9 shows a screenshot of animals navigating in the presense of tree-like obstacles.

7 Complexity

Reynolds' flocking algorithm has a complexity of $O(n^2)$, since each animal must consider every other animal in the flock, even if just to determine whether each animal is within the range of vision. We can consider the workload as two parts. For each animal, (1) we must decide which animals are 'neighbors' of the animal, and (2) the interactions with those neighbors must be computed. The first part, computing the neighbor lists for all the animals, is $O(n^2)$ in the size of the flock. The second part, computing interactions between neighbors, is $O(n^2)$ in the size of the neighbor list.

The first part could be sped up by some sort of spatial sorting, as

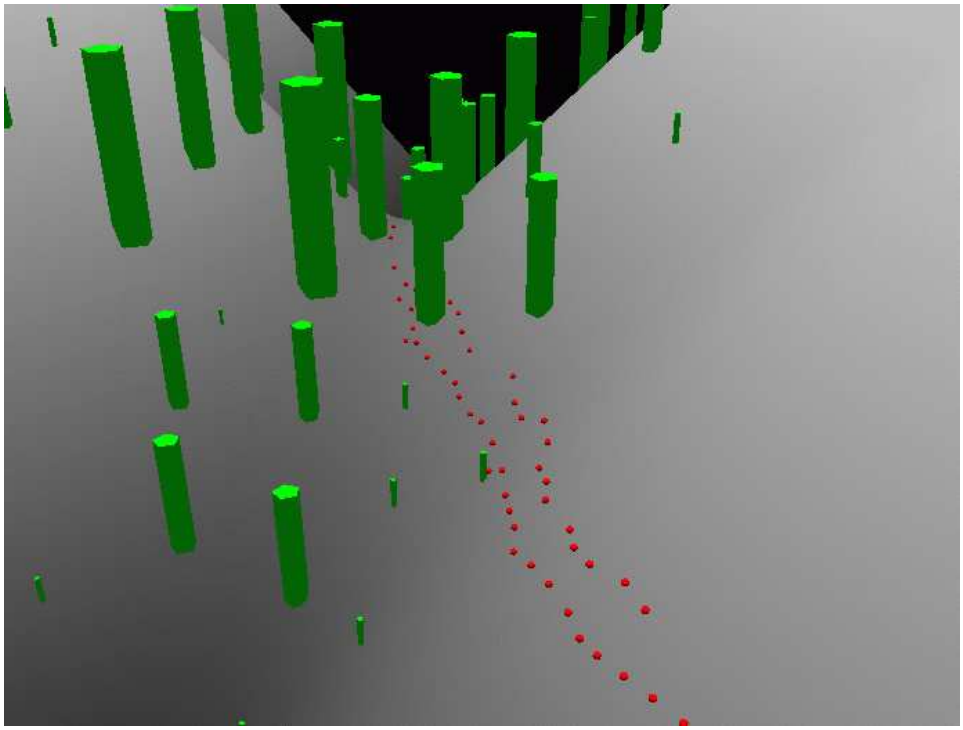

Figure 9: *Screenshot of animals flocking with obstacles.*

Reynolds discusses in his 1987 paper [8], where each animal is placed in a 'bin' and computing the neighbor list requires looking at only the animals in the nearby bins. This only speeds up the task of computing the neighbor list. The amount of time required to actually compute the interactions between neighbors is not changed. The simplest way to improve this second part is to just reduce the visible range of each animal in the flock, thus reducing the effective 'neighborhood.'

8 Implementation

The ideas presented in this paper were implemented in C++ on a Windows machine using the Microsoft Direct3D API. When tested on a 2.53GHz Pentium 4 processor, with 50 animals in a flock, a one-frame update of all the animals took about 10ms.

8.1 Collision handling

If the simulation is to look at all realistic then we must handle collisions properly so that solid objects are not passing through each other. Because the simulation is really a discrete-time approximation, we will sometimes have solid objects overlapping the same space. A simple but crude solution is just to force the objects to move so they are not overlapping. After computing the next update to the objects' positions, if two objects are found to be overlapping, then one or both objects should artificially be 'backed up' so that the objects are no longer overlapping. At this time any changes in velocity, energy, etc. that result from the collision should be made. This is not an ideal solution because this artificial movement of the objects produces small errors that can accumulate over time. These errors become larger if the objects are moving at higher speeds so that their overlaps are larger.

Another solution is described by Bourg [1]. If, after the Δt update to

position is calculated, an overlap is detected, then the simulation is backed up to the point before the update and a smaller time-step (e.g., $\frac{\Delta t}{2}$) update is attempted. In case there is still an overlap, then we repeat the above-described procedure with smaller and smaller time steps until there is no overlap. Note that we need to specify some small threshold distance between two objects such that if two objects are closer together than the threshold distance, but not overlapping, then they are defined to be colliding.

This method of dividing the time step and recalculating should provide much more accurate simulations. However, if our desire is to produce real-time simulations, then this is unlikely to be possible. Generally, the real-time simulator will be computing updates as rapidly as possible and will not have the time to divide time steps and recompute.

In addition to these concerns, there is another problem. If an object is traveling fast enough, then it may jump through an obstacle with no overlap ever occurring. If overlap is our only way to detect collisions, then an object may pass through an obstacle without our even detecting it.

If an object is traveling at a speed below $\frac{L}{\Delta t}$ (where L is the length of the object in the direction of travel), then there will be no overlap between the object's positions at consecutive time-steps. However, if the object is traveling above that speed, there will be a gap between the consecutive

positions. The size of this gap is $||\vec{v}|| \cdot (\Delta t - L)$ where $\vec{v}$ is the velocity of the object. If this value is positive, then there is a gap, and if the gap is large enough for an obstacle to fit inside, then it is possible for the object to jump 'through' the obstacle with no overlap occurring. A solution to this would be to test whether the line segment connecting the two consecutive positions intersects with the obstacle. An even more precise strategy would be to test for an intersection between the obstacle and the volume swept out through space as the object moves between the consecutive positions.

8.2 Flock Size

When running a simulation in real-time, it is important to keep the frame rate high. Unfortunately the ability to do this is dependent on the complexity of the computation, the implementation, and the hardware on which the simulation is run. Ignoring for the moment any annoyance that a low frame-rate causes the user of a real-time simulation, there seems to be a lower limit on the update rate for the flocks of animals. With my implementation, the animals' acceleration needs to be updated based on behavior at least about ten times per second in order for normal flocking behavior to occur. If all the animals are updated every frame, and the frame-rate is artificially lowered to about ten per second or less, then the animals fail to flock together as they

should. Also, I implemented a feature that allows every animal's velocity and position to be updated every frame, but behavior is recalculated every i-th frame, where the set of animals is randomly divided at generation time into i equal groups, and frame-rate up to a point. Indeed, when i is set large enough to cause each animal's behavior to be updated at or less than about ten times per second, then the results are the same as when the original simulation was slowed down artificially. The reason for this breakdown in the simulation at such low frame-rates is the use of forward Euler's for the simulation. Since this is an explicit integration technique, then it become unstable when δt becomes too large.

Since the complexity of the flocking algorithm is quadratic in the number of animals in the flock, the flock size drastically affects the frame-rate. Since there seems to be a lower limit on the allowable frame-rate, this places a restriction on the maximum flock size. Of course this depends heavily on the implementation and the hardware the simulation is run on. On my machine, with my implementation, the largest reasonable flock size was 140.

As discussed in Section 9.3.4 the process of computing which of the members of the flock are neighbors to each animal can take a large percentage of the simulations computation time, especially when trigonometric functions are needed to compute the viewing angle. It should be kept in mind, then,

that this may be a major limiting factor in the frame rate, and thus the flock size.

9 Parameters

This implementation has many parameters that can be adjusted to create the desired results. The main set of these parameters is listed here:

- *Animal size*: For this implementation, we defined the simulation parameters in terms of the size of the animals. By defining our unit of length as the radius of an animal, then the other parameters may be given in animal-radius units. The animal size can then be considered no longer a parameter but merely a constant, because all other parameters are defined in terms of the animal size.

- *Maximum speed* (v_{max}): the animals' top speed

- *Cruising speed* (v_c): The cruising speed is the natural speed the animals will attempt to maintain when heading towards some goal over flat terrain.

- *Maximum acceleration* (a_{max}): the animals' top acceleration

- *Comfort-zone radius* (C): The comfort-zone radius defines a circle around each animal. If a second animal enters the first animal's comfort zone, the first animal will move away.

- *Danger-zone radius* (D): The danger-zone radius specifies a smaller, concentric circle inside the comfort-zone radius. This danger-zone radius aids in defining the comfort zone and specifies the radius at which two animals are much too close, and they must accelerate (a_{max}) away to avoid a collision. If a second animal enters the outer edges of the first animal's comfort-zone, the first animal will use minimal acceleration to move away. However, if the second animal manages to get closer, then the acceleration increases until it reaches the maximum possible acceleration at the danger-zone radius.

- *Visible flock-mate range* (R_f): Visible flock-mate range is simply the maximum distance at which a second animal may be from the first animal and still be used in calculating the first animal's behavior.

- *Visible obstacle range* (R_o): Similarly, the visible obstacle range is the maximum distance an obstacle can be from an animal and still be used in calculating the animal's behavior for avoiding the obstacle. These two visible-range parameters (R_f and R_o) may be set equal to each

other; however, for maximum flexibility, we have separated the two. And they will be discussed separately in this paper.

- *Viewing angle* (θ_v): The animal's field of view, which is measured as an angle from the direction of the current velocity.

- *Drag factor* (f_d): A drag force causing the animals to follow the contour lines of the terrain more closely.

Figure 10 shows which parameters are dependent on each other. In the figure, an arrow from A to B indicates that the value of B depends on the value of A.

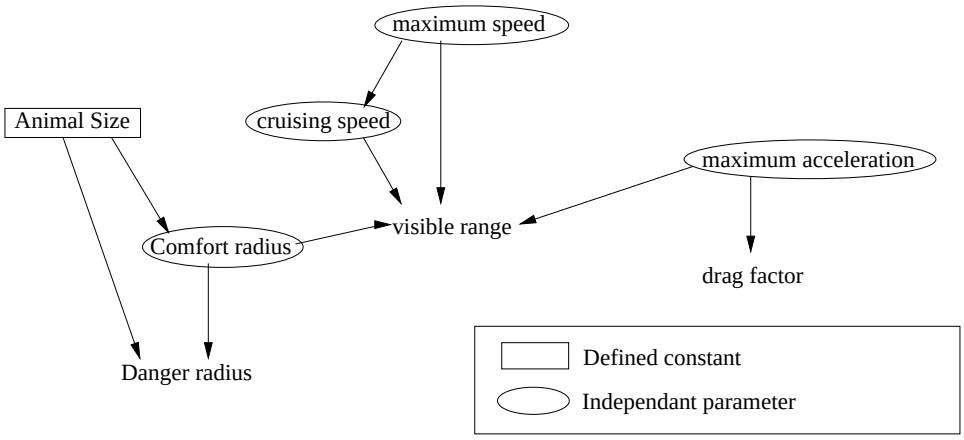

Figure 10: *Dependency of the parameters.*

Section 9.1 describes the set of parameters that may be considered independently. Section 9.2 describes the set of parameters that determines

the animals' abilities to react quickly to events (such as collisions) and gives the relation connecting these parameters. Section 9.3 discusses parameters individually and landmark values for those parameters.

9.1 Independent parameters

A few of the parameters can be set independently from the others. As stated earlier, the animal size is a constant defining the unit of length. After setting this constant, the following parameters may be set independently from the other parameters and define the basics of the animal behaviors:

- Maximum speed (v_{max})
- Cruising speed (v_c)
- Maximum acceleration (a_{max})
- Comfort zone radius (C)

Section 9.2 explains the relationship between these parameters. Speed and the comfort zones C and D may likely be the first parameters to be chosen since they so greatly define the behavior of the flock by defining its rate of movement and its spacing.

9.2 Animal reaction

The animals' ability to react to events like impending collisions is governed by two quantities: the time the animals have to react and the rate at which they are able to react. For an impending collision, the time that an animal has to react is, of course, dependent on the animal's speed and the distance to the obstacle. The ability to react in that amount of time is dependent on the animal's ability to accelerate.

For example, if an animal is moving straight toward a tree, then whether or not a collision occurs is dependent on:

- how close the animal needs to get to the tree to notice it (R_o),
- how fast the animal is going (up to v_{max}), and
- the animal's maximum acceleration a_{max}.

These are related to each other as shown, assuming that the animal is heading directly towards an obstacle and needs to accelerate orthogonally to its original velocity by a certain distance to sidestep the obstacle.

$$\text{Reaction time}(t) = \frac{R}{||\vec{v_0}||} \tag{15}$$

$$\text{maximum distance able to be cleared} = \frac{1}{2}a_{max} \cdot t^2 = \frac{1}{2}a_{max}(\frac{R}{||\vec{v_0}||})^2 \tag{16}$$

where R is the visible range and $\vec{v_0}$ is the initial velocity. It seems, then, that adjusting the velocity and the visible range is more effective than adjusting acceleration, since they have a quadratic relationship with the ability to react, while acceleration has only a linear relationship.

On the other hand, if, instead of sidestepping the obstacle, we would like the animal heading towards an obstacle to be able to decelerate to a stop before the collision, then the following inequality must hold:

$$||\vec{v_0}||^2 \leq 2a_{max}R \tag{17}$$

assuming that the acceleration a_{max} is in the opposite direction of the velocity. Thus, the ratio

$$\frac{2a_{max}R}{||\vec{v_0}||^2} \tag{18}$$

must be held above unity.

Of course, when setting these parameters, we need to allow for more than the exact ability to react. We must give some room for error to allow for inaccuracies and imprecision in the simulation. At minimum we have inaccuracy from the discrete-time nature of the simulation, and we have imprecision from finite-precision floating-point numbers.

9.3 Inference of landmark values through empirical analysis

This section describes some of the parameters individually in more detail. Reasonable bounds are considered for these parameters in order to determine landmark values.

9.3.1 Comfort zone

The comfort zone C of the animal is the radius that an animal tries to keep between itself and its flock-mates. Clearly this is dependent on the animal size as it must be larger than the animal. If two animals' centers get closer than this distance, then the animals will accelerate apart. Also the danger zone radius D must be larger than the animal size but smaller than the comfort zone radius.

9.3.2 Visible flock-mate range

This parameter R_f controls how far away two animals can get from each other and still affect each other's behavior. The range of possible values for this parameter are discussed here.

In order to improve the real-time performance of the simulation, the visible range of the animals should be reduced as much as possible to reduce the number of other flock members that each animal has to consider. How-

ever, this raises the question of how far can the visible range practically be lowered.

First, it is clear that reducing the visible range below the comfort zone radius will significantly change the flock's behavior. If the animal cannot see as far out as the edges of its comfort zone, it will not know if another animal has entered its comfort zone. And if the two animals have the same parameters, then the two animals may be closer to each other than they should be.

Another factor limiting the minimum acceptable value for the visible range is the maximum speed v_{max} of the animals. If the animal's top speed is increased, then the minimum value for the visible range should be increased. The reason for this is that the ratio of visible range to speed gives the reaction time for collisions. And this ratio must be kept above some minimum allowed reaction time. This was covered in more detail in Section 9.2.

As for the maximum value for the visible flock-mate range, it can be set arbitrarily large, keeping in mind that as R_f is increased, then each animal has a larger neighborhood of flock-mates to consider which will slow the simulation. Beyond some point, however, raising the value will have no

further effect[4]. Increasing the visible range beyond the diameter of the flock will have no effect on the flock's behavior. An approximate value for this largest distance might possibly be computed for a flock at rest. When at rest, the flock is squeezed together as tight as possible while keeping from invading each others' comfort zones. Indeed, in simulation they typically arrange themselves in a tight group with approximately a hexagonal packing lattice[5]. This makes sense since hexagonal packing becomes optimal for packing circles inside a larger circle for large enough n (in this case, number of animals). The area that this group covers should increase approximately linearly with n. Thus the diameter of the group should vary proportionally with $\sqrt{n}$. The downside to this is that by forming a lattice, the animals are arranged in straight rows, which may be undesirable. Perhaps, to avoid this phenomena, an additional behavior module is needed for random grazing behavior when a herd is not traveling.

9.3.3 Visible obstacle range

This parameter R_o controls the distance from an animal within which obstacles (such as trees) affect an animal's behavior.

[4]It is worth mentioning that this statement hold only for the visible flock-mate range. The visible obstacle range has different landmark values, as we discuss in Section 9.3.3.

[5]Gauss proved that this is the most compact lattice pattern.

In my implementation of obstacle avoidance, if an animal is heading towards an obstacle, then the animal will attempt to steer towards the edge of the obstacle that provides the shortest path to the goal. Figure 11 shows an example of animals avoiding obstacles.

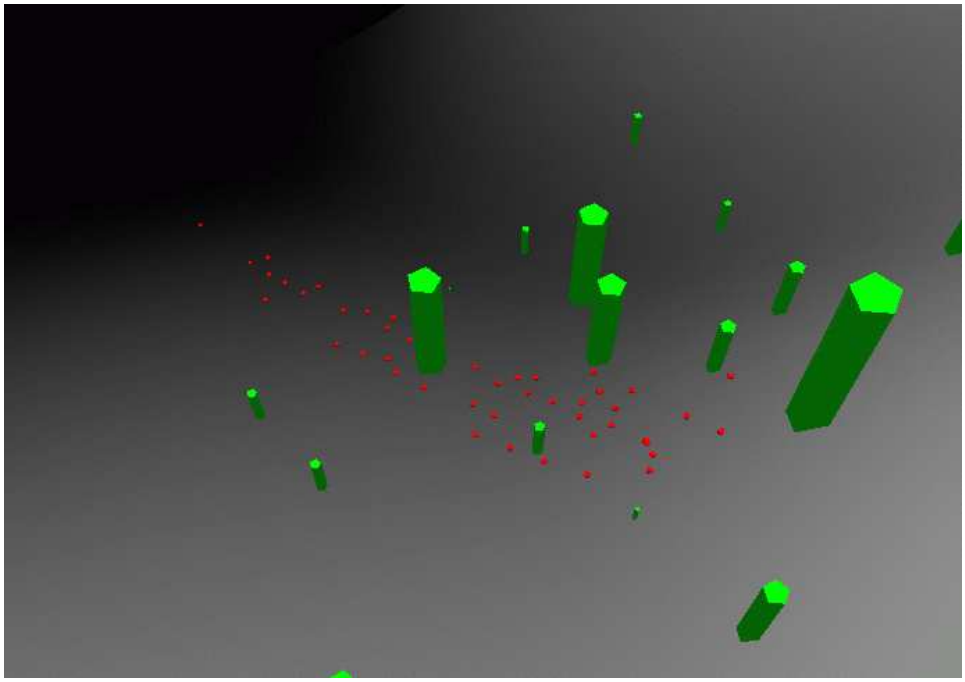

Figure 11: *Screenshot showing animals avoiding obstacles.*

This particular method of obstacle avoidance described in Section 6 creates a limit to the maximum acceptable value for the visible obstacle range. The reason is that if an animal is facing an obstacle, then it will attempt to steer towards the nearest edge that minimizes the path distance to the

goal. If the obstacles are positioned densely enough, or if the visible obstacle range is large enough, then the animal may 'see' a solid wall of obstacles. As soon as it has steered away from one obstacle, it will be facing another obstacle, and may steer back to the original obstacle or to yet another obstacle. This may continue indefinitely and may prevent the animals from making progress. This problem can be eliminated by limiting the visible obstacle range so that only the nearest obstacles are visible.

9.3.4 Viewing Angle

The viewing angle θ_v indicates how far behind itself an animal can 'see.' This is given as an angle from directly ahead, so the parameter can range from 0 to π. However, in order for a traveling flock to keep the proper spacing, then two animals side by side need to be able to see each other. Thus, the viewing angle must be at least $\frac{\pi}{2}$. This angle need not represent merely peripheral vision, may include any way of sensing neighbors, such as turning the head to scan a wider angle, and hearing.

Filtering by the viewing angle presents a small problem since it requires calculating the angle between two vectors, which requires a trigonometric operation. This operation will have to be calculated $O(n^2)$ number of times every frame. In my implementation, results showed that this trigonometric

operation was using most of the computation time. This can be sped up by storing cosine of the viewing angle.

9.3.5 Drag factor

The drag factor d_f is a multiplying constant to a force that helps the animals travel more closely to the contour lines of the terrain. Setting this parameter to zero eliminates this force, so that the animals travel in a straight line over the terrain, ignoring the slope. Increasing the parameter results in a stronger force keeping the animals from traveling across contour lines. If the parameter is increased too high, then the animals will be held to contour lines so strongly that they will not be able to climb even shallow hills. In this case, the animals may easily become stuck. Figure 13 shows an example of animals following contour lines as their group splits to get around a hill.

The drag factor normally resists the animals' movements away from contour lines. However, a negative value for d_f will actually accelerate the animals away from the contour lines, producing unrealistic behavior. In this case the animals will accelerate rapidly up and down the steepest hills and never follow contour lines.

9.4 Summary

Table 1 lists the parameters discussed above and their landmark values, summarizing our analysis.

Table 1: *Parameter landmarks.*

		Landmark values	
Parameter		minimum	maximum
animal size		0	∞
max speed	v_{max}	cruising speed	∞
cruising speed	v_c	0	max speed
max acceleration	a_{max}	0	∞
comfort zone	C	animal size	∞
danger zone	D	animal size	comfort zone
visible flock-mate	R_f	comfort zone	farthest flock-mate
visible obstacle	R_o	animal size	∞
viewing angle	θ_v	$\frac{\pi}{2}$	π
drag factor	f_d	0	~55

9.5 Varying Behaviors

By varying different parameters, different behaviors can emerge. Within reasonable ranges, a fairly realistic herding behavior occurs. However, by setting extreme values, odd behavior can occur. All of these results were obtained from a flock of animals that all had the same parameter values.

- *Ghost-like*: By reducing the animals' vision of each other to nil, then each animal completely ignores the other. And, if the physics engine

does not handle the collisions, then the behavior is like a group of independent ghosts that can pass right though each other. The vision can be reduced to nothing by setting either the visible range or the viewing angle to zero.

- *Trapped by obstacles*: By including enough obstacles within the visible obstacle range so that the animals will see a solid wall of obstacles, then the animals will no longer to be able to make progress and will be, in a sense, trapped.

- *Overwhelmed by terrain*: By setting the drag force high enough, or creating terrain steep enough, then the animals will no longer be able to climb the slope. Thus the animals can become trapped by the terrain and not be able to make progress.

- *Swarm of insects*: By allowing the animals to see each other only for the purpose of trying to move to the center of the flock, then it can create an effect that looks similar to a swarm of flying insects as they appear to move randomly or perhaps in an orbit around the center of mass of the flock.

Table 2: *Parameters used to create various behaviors.*

Behavior	**Parameters**
Ghost-like	$R_f = 0$ or $\theta_v = 0$
Trapped by obstacles	R_o small enough, and enough trees
Overwhelmed by terrain	f_d large enough
Swarm of insects	$R_f = 0$ for avoid-collision behavior

10 Results

These modifications to the flocking algorithm produce pleasing results. The animals swerve around hills and dips in the terrain. They travel more slowly when traveling downhill and more slowly still when traveling uphill. Figure 12 shows a screen-shot of a herd moving over the terrain towards the camera. Each animal is represented by a small sphere.

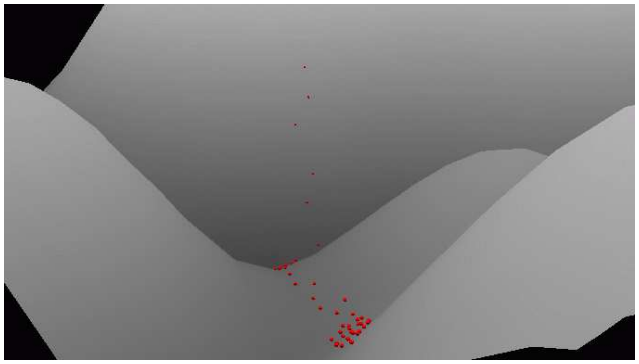

Figure 12: *Screenshot of herds moving over terrain*

Figure 13 shows another screenshot from a top-down view. In this picture, the herd is moving towards the bottom right. This demonstrates that the herd can divide into two groups to go around a hill. They will regroup on the other side of the hill.

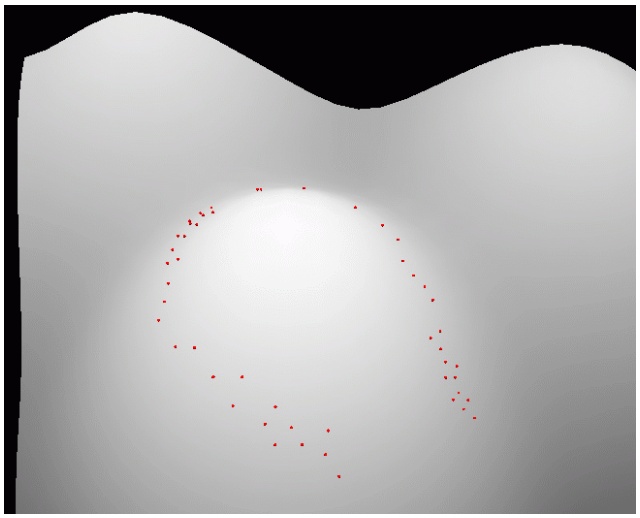

Figure 13: *Screenshot of herd dividing into two group to go around a hill.*

Since the 3D correction (Section 5.1) and the drag force (Section 5.2.3) for following terrain are simply functions of the slope at the animal's current position, then computing this is constant-time for each animal, and thus for the herd as a whole it is O(n) in the number of animals in the herd.

11 Future Work

There are many avenues for future exploration. Among the possible issues raised are these listed here:

How do we handle a long ridge in the terrain with a distant gap? Should the animals be smart enough to travel along the ridge to pass through the gap in it.

This method is limited in that it was created specifically for terrain described as an elevation grid. Perhaps modifications can be made to apply this method to arbitrary terrain which might include features such as sheer cliffs, overhangs, land bridges, or tunnels.

This method consists of a two-dimensional flocking algorithm with corrections applied to it. Perhaps another method should be explored using a three-dimensional flocking algorithm with constraints so the animals' are positioned on the terrain and their headings are tangent to the terrain.

Future progress could be done to give the animals higher-level behavior. At this point, they merely navigate to a pre-determined goal location. A higher-level behavior control module could make decisions about behavior. Such behaviors could include finding food and water, migrating, and avoiding danger. This brings up many complex issues such as how do the animals find food and water? What sort of representation of the terrain should be

used to use to make behavioral decisions? Would this involve some sort of spatial database system? This could be used to allow the animals to analyze the terrain in front of them to make more intelligent decisions about where to travel. What kind of memory should the animals have? Is one animal in the herd the leader? If so, is there any challenging of the leader's position? Is there some kind of decision making at the herd-level? How much influence do individual animals have?

Experiments could be done concerning the interaction of different species. Especially interesting would be the integration of predators and prey, and the factors that determine whether the ecological system is stable.

This method was designed for use with point-mass objects. Further work could include the use of objects with a more complex representation or with complex dynamics. Also involuntary motion like slipping and sliding down a hill has not yet been implemented here.

This method simply uses distance to determine if neighbors are visible. A more complex version might analyze the terrain and any obstacles in the area to determine if vision is obstructed.

12 Conclusion

Reynolds' flocking algorithm creates flocking behavior by combining a small set of simple rules that each animal follows. By adding additional rules, various behaviors can be created. In order to create the behavior of herds moving over terrain we have made three modifications to the basic flocking algorithm.

1. The position, velocity, and acceleration vectors in the flocking algorithm are restricted to the 2D (x, y) plane.

2. At each time step, the position of an each animal is updated no longer by just its velocity $\vec{v}$ but by $\frac{\vec{v}}{\sqrt{{s_v}^2+1}}$.

3. An additional behavioral rule is added: an acceleration equal to $-K\frac{(\vec{v}\cdot\vec{g})\vec{g}}{||\vec{g}||}$.

The constant K controls how tightly the herd will follow the contour lines of the terrain. K can be adjusted to allow low energy expenditure and other terrain-following behaviors.

A Source code

```
//////////////////////////////////////////////////////////////////////
// Boid.h: interface for the Boid class.
//   A Boid object represents one animal in the simulation
// Joel Gompert
//////////////////////////////////////////////////////////////////////

#if !defined(AFX_BOID_H__4A9416A7_C323_4E53_87BB_A1E6DD0D0123__INCLUDED_)
#define AFX_BOID_H__4A9416A7_C323_4E53_87BB_A1E6DD0D0123__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "Vector2D.h"
#include "Entity.h"
#include "Mesh.h"
#include "Terrain.h"
#include "Flock.h"

class Flock;

class Boid : public PhysicalEntity
{
public:

   // Constructor #1
   // creates an individual boid with randomized position,
   // velocity, and orientation.
   Boid(LPDIRECT3DDEVICE9 CreationDevice, Flock* f, Terrain* t);

   // Constructor #2
   // creates an individual boid with specific position and velocity.
   Boid(LPDIRECT3DDEVICE9 CreationDevice, Flock* f, Terrain* t,
           const Vector2D& startpos, const Vector2D& startvel);

   ////////////////////////////////////////////////////
   // functions required for PhysicalEntity
   D3DXVECTOR3 GetGroundContactPoint();
```

```
void GetMeshes(Mesh** MeshArray);
int GetMeshCount();
unsigned int GetAction(float TimeStep, Msg** SentMsgs);
unsigned int SendMsg(Msg* RecievedMsg, Msg** SentMsgs);
//////////////////////////////////////////////////////

// Destructor
virtual ~Boid() {}

// Update_Behaviors needs to be called frequently.
// This is where the animal makes decisions and updates
// its behavior and current acceleration
void Update_Behaviors();

// Update needs to be called every frame.  This is where
// the animal's position is updated and animated
void Update(float elapsed_time);

// helper stuff for controlling the rate at which
// Update_Behaviors() is called
int behavior_update_delay;
       // number of frames between behavior updates

int last_behavior_update;
             // number of frames since last behavior update

// accessor function
// retrieves the current 2D position of the animal
Vector2D getPos() const;

// for some experiments we want the boids in the flock
// in different groups, we'll call the groups 'teams'
int         team;

// a pointer to the terrain object
Terrain* terrain;

//////////////////////////////////
// Temporary addition for calculating energy expenditure
D3DXVECTOR3 lastPos;
```

```
   float energy;
   void reset();
   float TERRAIN_DRAG_FACTOR;
   float DOWNHILL_TERRAIN_DRAG_FACTOR;
   ////////////////////////////////////////////////////

   // this lets the animal know which flock it is a member of
   Flock* parentFlock;

private:

   // this is the 3D model that appears in the
   // animation for this animal
   Mesh*    m_mesh;

   // current 2D acceleration, velocity and position of the animal
   Vector2D m_2acceleration;
   Vector2D m_2velocity;
   Vector2D m_2position;

   // this is scratch space used by each boid
   static list<Boid *> neighborList;

   // finds all animals in parentFlock which are to be considered
   // neigbors of this animal and stores this
   // list of animals in neighborList
   void computeNeighbors();

   // computes new acceleration for the boid each frame
   // it is called by Update_Behaviors()
   Vector2D navigator();

   // These functions provide all of the basis behaviors that are
   // combined to create the overall behavior of the animal
   Vector2D keepDistance();
      // keeps comfort space between animals

   Vector2D matchHeading();
      // adjusts velocity to match neighbors
```

```
   Vector2D steerToCenter();  // maintains flock cohesion
   Vector2D avoidObstacles(); // steer arounds trees
   Vector2D gotoGoal();    // steers animal towards the goal
   //Vector2D avoidCircularObstacle(Vector2D pos, float radius);

   // this is the function that combines the basis behaviors
   // it combines valueToAdd with the running total contained
   // in accumulator and returns the magnitude of the resulting
   // value of accumulator
   float accumulate(Vector2D& accumulator,
            const Vector2D& valueToAdd);
};

#endif//!defined(AFX_BOID_H__4A9416A7_C323_4E53_87BB_A1E6DD0D0123__INCLUDED_)
```

```
//////////////////////////////////////////////////////////////////////
//
// Boid.cpp: implementation of the Boid class.
//     A Boid object represents one animal in the simulation
// Joel Gompert
//////////////////////////////////////////////////////////////////////

#include "Boid.h"
#include "utils.h"
#include "forest.h"
#include <cmath>

////////////////////////////////////////////
// stuff for profiling
#include "profiler/profiler_lowlevel.h"
#include "profiler/profiler_highlevel.h"

// Definitions for the program zones we will use in
// this file (there are also some zones in the
// 'profiler' subdirectory)
Define_Zone(boid_update);
Define_Zone(boid_navigator);
Define_Zone(boid_accumulate);
Define_Zone(boid_compute_neigbors);
Define_Zone(boid_avoid_obstacles);
Define_Zone(boid_keep_distance);
Define_Zone(boid_match_heading);
Define_Zone(boid_steer_to_center);
Define_Zone(boid_goto_goal);
// end of profiling stuff
////////////////////////////////////////////


//////////////////////////////////////////////////////////
// Here are the parameters that control the animal behavior
// at this point they are constants, but they could be different
// for each individual animal
const float BOID_MAX_SPEED = 10.00f;
const float BOID_CRUISING_SPEED = 0.7f * BOID_MAX_SPEED;
const float BOID_MAX_ACCELERTION = BOID_MAX_SPEED * 2.0f;
```

```
const float BOID_RADIUS = .05f;
const float BOID_COMFORT_RADIUS = BOID_RADIUS * 8.0f;
const float BOID_DANGER_RADIUS = BOID_RADIUS * 4.0f;
const float BOID_VISIBLE_RANGE = BOID_RADIUS * 200.0f;
const float BOID_VISIBLE_TREE_RANGE = 10.4f;
const float BOID_VIEWING_ANGLE = PI*3.0f / 4.0f;
   // Maximum angle (from the vector pointing directly ahead)
   // that a boid can see
const float BOID_INITIAL_TERRAIN_DRAG_FACTOR = 30.0f;
const float BOID_INITIAL_DOWNHILL_DRAG_FACTOR =
         BOID_INITIAL_TERRAIN_DRAG_FACTOR / 5;
//////////////////////////////////////////////////////////////


// declaration necessary for static members
list<Boid *> Boid::neighborList;


//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

// Constructor #1
// creates an individual boid with randaomized position,
// velocity, and orientation.
Boid::Boid(LPDIRECT3DDEVICE9 CreationDevice, Flock* f, Terrain* t)
{
   parentFlock = f;
   terrain = t;

   ////////////////////////////////////////////////////
   // set up the 3D model for this boid
   m_mesh = new Mesh();
   LPD3DXMESH NewMesh;

   // for now animals will be represented by simple spheres
   D3DXCreateSphere(CreationDevice, BOID_RADIUS, 10,10,
            &NewMesh, NULL);

   m_mesh->SetD3DXMesh(NewMesh);
```

```
// set up the material for this boid
// for now, a simple red color will do
D3DMATERIAL9 NewMaterial;
memset(&NewMaterial,0, sizeof(NewMaterial));
NewMaterial.Ambient.a = 1;
NewMaterial.Ambient.b = 0;
NewMaterial.Ambient.r = 1;
NewMaterial.Ambient.g = 0;
NewMaterial.Diffuse = NewMaterial.Ambient;
m_mesh->SetMaterials(1, &NewMaterial);
////////////////////////////////////////////////////

team = 0;

// set a random position
m_2position = Vector2D(
   3.0f * (float(rand()) / float(RAND_MAX)) +3.5f,
   3.0f * (float(rand()) / float(RAND_MAX)) +3.5f);

// pick a random angle...
float r = 2*PI * (float(rand()) / float(RAND_MAX));
// ...and a random speed...
float speed = BOID_MAX_SPEED * (float(rand()) / float(RAND_MAX));
// ...to form a random velocity
m_2velocity = Vector2D((float)sin(r), (float)-cos(r)) * speed;

// initial acceleration is zero
m_2acceleration = Vector2D(0,0);

// set the PhysicalEntity 3D positon
Position = D3DXVECTOR3(m_2position.x, m_2position.y,
   terrain->GetHeight(m_2position.x, m_2position.y) + BOID_RADIUS*2);

////////////////////////////////////
// Temporary addition for calculating energy expenditure
lastPos = Position;
energy = 0;
TERRAIN_DRAG_FACTOR = BOID_INITIAL_TERRAIN_DRAG_FACTOR;
DOWNHILL_TERRAIN_DRAG_FACTOR = BOID_INITIAL_DOWNHILL_DRAG_FACTOR;
```

```
   ////////////////////////////////////////////////////
}

// Constructor #2
// creates an individual boid with specific position and velocity.
Boid::Boid(LPDIRECT3DDEVICE9 CreationDevice, Flock* f, Terrain* t,
                const Vector2D& startpos, const Vector2D& startvel)
{
   parentFlock = f;
   terrain = t;

   ////////////////////////////////////////////////////
   // set up the 3D model for this boid
   m_mesh = new Mesh();
   LPD3DXMESH NewMesh;

   //Radius = BallRadius;

   // for now animals will be represented by simple spheres
   D3DXCreateSphere(CreationDevice, BOID_RADIUS, 10,10, &NewMesh, NULL);
   m_mesh->SetD3DXMesh(NewMesh);

   // set up the material for this boid
   // for now, a simple red color will do
   D3DMATERIAL9 NewMaterial;
   memset(&NewMaterial,0, sizeof(NewMaterial));
   NewMaterial.Ambient.a = 1;
   NewMaterial.Ambient.b = 0;
   NewMaterial.Ambient.r = 1;
   NewMaterial.Ambient.g = 0;
   NewMaterial.Diffuse = NewMaterial.Ambient;
   m_mesh->SetMaterials(1, &NewMaterial);
   ////////////////////////////////////////////////////

   team = 0;

   // set position and velocity
   m_2position = startpos;
   m_2velocity = startvel;
```

```
   // initial acceleration is zero
   m_2acceleration = Vector2D(0,0);

   // set the PhysicalEntity 3D positon
   Position = D3DXVECTOR3(m_2position.x, m_2position.y,
      terrain->GetHeight(m_2position.x, m_2position.y) + BOID_RADIUS*2);

   ////////////////////////////////////////////////////////
   // Temporary addition for calculating energy expenditure
   lastPos = Position;
   energy = 0;
   TERRAIN_DRAG_FACTOR = BOID_INITIAL_TERRAIN_DRAG_FACTOR;
   DOWNHILL_TERRAIN_DRAG_FACTOR = BOID_INITIAL_DOWNHILL_DRAG_FACTOR;
   ////////////////////////////////////////////////////////
}

// Update() must be called every frame, but
// Update_Behaviors() may be called less often,
// but preferably still at least several times a second
// Update_Behaviors needs to be called frequently.  This is where the animal
// makes decisions and updates its behavior and current acceleration
void Boid::Update_Behaviors()
{
   // at this point just get new acceleration based on basis behaviors
   m_2acceleration = navigator();
}

// Update needs to be called every frame.  This is where the animal's position
// is updated and animated
void Boid::Update(float elapsed_time)
{
   Profile_Scope(boid_update);

   ////////////////////////////////////////////////////////////////
   // Drag Force behavior

   // we want to take into account the gradient
   // of the terrain at the boid's current position
   Vector2D grad(terrain->GetGradient(this->m_2position.x,
                 this->m_2position.y));
```

```
// slope in the direction of the gradient
float slope = grad.magnitude();

//m_2acceleration -= grad * 4 / (1+ slope*slope);// * slope;

if(slope > 0) // if the slope is zero, then nothing to do
{
   //Vector2D unitgrad = grad / slope; // normalized gradient

   // component of current velocity in the direction of the gradient
   float dot = DotProduct(this->m_2velocity, grad);
   Vector2D new_a;
   if(dot > 0) // boid is traveling uphill
   {
      new_a = (-TERRAIN_DRAG_FACTOR * dot / slope) * grad ;
   }
   else if(dot < 0) // boid is traveling downhill
   {
      new_a = (-DOWNHILL_TERRAIN_DRAG_FACTOR * dot / slope) * grad;
   }

   m_2acceleration += new_a;
}

//////////////////////////////////////////////////////////////////////
// update position

// component of the gradient in the direction of the current velocity
// in other words, the slope in the direction the boid is travelling.
float grad_velcomp = DotProduct(this->m_2velocity.normalized(), grad);

// the 1/sqrtf(s*s + 1) adjustments are there to account for
// the fact that the two dimensional velocity and acc are actually
// moving the boids over 3D terrain.  This effectively reduces the speed
// by a function of the slope in that direction. A possible
// optimization would be to approximate the 1/sqrt(s*s + 1) factor

m_2position += m_2velocity*elapsed_time /
      sqrtf(grad_velcomp*grad_velcomp + 1);
```

```
//////////////////////////////////////////////////////////////////////


// update velocity
m_2velocity += m_2acceleration * elapsed_time;

// Cap off velocity at maximum allowed value
if (m_2velocity.squared_magnitude() > (BOID_MAX_SPEED*BOID_MAX_SPEED))
{
   m_2velocity.setMagnitude(BOID_MAX_SPEED);
 }

// set the Physical entity positon
Position = D3DXVECTOR3(m_2position.x, m_2position.y,
   terrain->GetHeight(m_2position.x, m_2position.y) + BOID_RADIUS*2);

//////////////////////////////////////////////////////////
// Temporary addition for calculating energy expenditure
//D3DXVECTOR3 disp = Position - lastPos;
//lastPos = Position;
//float distance = sqrtf(D3DXVec3LengthSq(&disp));

//if(distance > 0)
//{
// float movement_slope = disp.z / sqrtf(D3DXVec3LengthSq(&disp) - SQUARED(disp.z)

// // energy expended is linear with distance traveled, and slope determines the c
// // come up with some multiplying factor as a function of the slope.
// float slope_factor = -(atanf(6.0f * (movement_slope + 0.34f)) - 5.0f * atanf(8.
//
// //// lets try some piecewise linear function
// //// less than x1, the energy expenditure is at some maximum for downhil moveme
// //// at slope of x2, energy expenditure is at a minimum
// //// greater than x3, the energy expenditure is at some maximum for uphill move
// //const float x1 = -1.0f, x2 = -0.1f, x3 = 1.0f;
// //if(movement_slope < x1) // below this we just use a constant
// //{
// // slope_factor = 3;
// //}
// //else if(movement_slope < x2) // in the range [x1, x2)
```

```
   // //{
   // // const float m = (1.0f-3.0f)/(x2 - x1);
   // // const float b = 3.0f - m*(x1);
   // // slope_factor = m * movement_slope + b; // (y = m*x + b)
   // //}
   // //else if(movement_slope < x3) // in the range [x2, x3)
   // //{
   // // const float m = (4.0f-1.0f)/(x3 - x2);
   // // const float b = 1.0f - m*x2;
   // // slope_factor = m * movement_slope + b; // (y = m*x + b)
   // //}
   // //else // in the range [x3, infinity)
   // //{
   // // slope_factor = 4;
   // //}

   // energy += distance * slope_factor;
   //}
   ////////////////////////////////////////////////////////

   // see if it is time to update behavior
   last_behavior_update = (last_behavior_update + 1) %
               behavior_update_delay;
   if(0 == last_behavior_update)
      Update_Behaviors();
}

// this is the function that combines the basis behaviors
// it combines valueToAdd with the running total contained in accumulator
// and returns the magnitude of the resulting value of accumulator
float Boid::accumulate(Vector2D& accumulator, const Vector2D& valueToAdd)
{
   Profile_Scope(boid_accumulate);

   // we want to come up with some combination of these two
   // vectors but we do not want the final result's magnitude
   // to be greater than 1.0

   // so first, we add the two together and check if we're within
   // the limit.
```

```
   Vector2D temp = accumulator + valueToAdd;
   if(temp.squared_magnitude() <= 1.0)
   {
      // we are, so go ahead and return this result
      accumulator = temp;
      return temp.magnitude();
   }
   else
   {
      // we're not, so we want to add on just enough of the valueToAdd
      // to bring our magnitude to 1.0
      // to do this, we want to find a value t such that
      // the magnitude of accumulator + t*valueToAdd  is 1.0
      // solving for this value t requires solving a quadratic
      // equation. Here are the 3 coefficients:
      float a = valueToAdd.squared_magnitude();
      float b = 2 * DotProduct(valueToAdd, accumulator);
      float c = accumulator.squared_magnitude()-1;
      // quadratic formula
      float t = (float) (-b + sqrt(b*b - 4*a*c))/(2*a);

      accumulator += valueToAdd*t;
      return 1.0f;
   }
}

// finds all animals in parentFlock which are to be considered
// neigbors of this animal and stores this list of animals in neighborList
void Boid::computeNeighbors()
{
   Profile_Scope(boid_compute_neigbors);

   // loop through all the members of the flock
   list<Boid *>::iterator i;
   for(i = (parentFlock->members.begin());
         i != parentFlock->members.end(); i++)
   {
      Boid *pBoid = *i;

      // do not bother adding this boid itself to its own
```

```
      // list of its neighbors
      if(pBoid == this)
         continue;

      // get vector pointing from this boid to its flock-mate
      Vector2D dir = pBoid->m_2position - this->m_2position;

      // if this other boid is visible then it is a neighbor
      if( dir.squared_magnitude() <= SQUARED(BOID_VISIBLE_RANGE) &&
            // if it is close enough
          this->m_2velocity.angle(dir) < BOID_VIEWING_ANGLE)
            // and if it is in the field of view
      {
         // then add it to the list of neighbors
         neighborList.push_back(pBoid);
      }
   }
}

// computes new acceleration for the boid each frame
// it is called by Update_Behaviors()
Vector2D Boid::navigator()
{
   Profile_Scope(boid_navigator);

   Vector2D acc(0,0);
   float mag;

   computeNeighbors();



      mag = accumulate(acc, avoidObstacles());

   if(mag < 1.0)
      mag = accumulate(acc, keepDistance());

   if(mag < 1.0)
      mag = accumulate(acc, matchHeading());
```

```
   if(mag < 1.0)
      mag = accumulate(acc, steerToCenter());

   if(mag < 1.0)
      mag = accumulate(acc, gotoGoal());

   // IMPORTANT:
  // Since the FlockCentering, CollisionAvoidance, and VelocityMatching modules
  // return a vector whose magnitude is a percentage of the maximum acceleration,
  // we need to convert this to a real magnitude
   acc *= BOID_MAX_ACCELERTION;

   neighborList.clear();

   return acc;
}

extern Forest* MainForest;
// steer arounds trees
Vector2D Boid::avoidObstacles()
{
   Profile_Scope(boid_avoid_obstacles);

   Tree** Trees;
   int TreeCount = MainForest->GetTrees(Trees); // Get the list of trees
   int i;
   Vector2D force(0,0); // accumulative force of all the trees
   float closest_dist_squared;
   bool tree_found = false;

   for(i=0; i<TreeCount; i++) // for each tree
   {
      // get the posititon and radius of this tree
      Vector2D treepos = Vector2D(Trees[i]->Position.x,
                  Trees[i]->Position.y);
      float radius = (Trees[i]->Height / 10.0f) + (BOID_RADIUS);
      // we add BOID_RADIUS, because we want the boid to aim not
      // for the exact edge of the tree, but to allow enough
      // room for boid to get past the edge
```

```
// disp is the vector pointing from the
// boid's position to the tree's center
Vector2D disp = treepos - this->m_2position;

// get the squared distance to the tree
float disp_mag_squared = disp.squared_magnitude();

if(tree_found && closest_dist_squared < disp_mag_squared)
   continue; // with next tree

// check emergency condition of colliding with the tree.
if(disp_mag_squared <= SQUARED(radius))
{
   // in this case we want to accelerate directly away

   // add a force directly away from the tree
   force = - disp * (100 * SQUARED(BOID_RADIUS) /
               disp_mag_squared);

   tree_found = true;
   closest_dist_squared = disp_mag_squared;
   continue; // with next tree
}

// otherwise we are not colliding yet

// check if the direction to the tree is behind the boid.
// if so, we can ignore it and just go on to the next tree
if(DotProduct(disp, this->m_2velocity) <= 0)
   continue; // with next tree

// if we get to this point we know that the
//  direction to the tree make an angle
// with the velocity vector of less than 90 degrees.


// find the projection of the tree onto the line
// perpendicular to disp
float projected_r;
if(disp_mag_squared < 4 * SQUARED(radius))
```

```
   // estimate if we are getting "close" to the tree
{
   // we're getting too close to the circle for
   // approximations, we need to be more exact
   projected_r = sqrtf(SQUARED(radius) * disp_mag_squared /
         (disp_mag_squared - SQUARED(radius)));
}
else if(disp_mag_squared < SQUARED(BOID_VISIBLE_TREE_RANGE))
{
   // use a bigger radius than the actual
   // radius just to give some comfort space
   projected_r = radius * 1.3f;
}
else // the tree is not within visible range
{
   continue; // with next tree
}

// here comes the tricky part.  Basically we want to
// project the circle of the tree onto a line perpendicular
// to the velocity.  If the tree projects entirely to the left
// or to the right of the current position, then the boid is
// not heading on a collision course

// this is the distance to the tree
float disp_mag = sqrtf(disp_mag_squared);

// use it to calculate the unit vector pointing to the tree
  Vector2D unit_disp = disp / disp_mag;
   // unit vector pointing towards center of tree

// d = unit vector pointing to the tree but rotated 90 degrees
Vector2D d = Vector2D(unit_disp.y, -unit_disp.x);
   // rotate 90 degrees

// projection of the velocity on a line perpendicular
// to the direction to the tree

// we need to calculate the 2 points to the either
// side of the tree that we will project
```

```
Vector2D point1 = (d * projected_r) + disp;
Vector2D point2 = (-d * projected_r) + disp;

// rotate the velocity 90 degrees so that we have a vector
// perpendicular to the velocity
Vector2D right(m_2velocity.y, -m_2velocity.x);
// we do not need to normalize it because we only care about
 relative distances

// p1 and p2 are the projections of point1 and point2
float p1 = DotProduct(point1, right);
float p2 = DotProduct(point2, right);

// if one projection is positive and the other is negative
// then the velocity is pointing at the tree.
// Otherwise we can ignore this tree
if((p1<0 && p2>0) || (p1>0 && p2<0))
{
   // okay, we're looking at the tree.  Now, which
   // edge of the tree is closer to the current
   // direction the boid is facing.
   Vector2D new_vel;

   // use the point that is closer to the velocity
   // direction (the one whose projection is
   // closest to zero)
   if(fabs(p1) > fabs(p2))
      new_vel = point2;
   else
      new_vel = point1;

   // we've got a vector pointing the direction we want.
   // Now to set its magnitude.
   // we can just use the boid's current speed.
   new_vel.setMagnitude(this->m_2velocity.magnitude());

   // now apply a force pushing the current velocity
   // towards this new velocity
   // (use the radius/distance ratio so that nearer
   // trees have a greater impact)
```

```
         force = 10000*(new_vel - m_2velocity)*(radius/disp_mag);
         tree_found = true;
         closest_dist_squared = disp_mag_squared;
      }
   }

   // limit at magnitude 1
   if(force.squared_magnitude() > 1)
      force.setMagnitude(1.0f);

   return force;
}

// keeps comfort space between animals
// provides an acceleration away from nearest neighbors if
// they are too close
// it assumes that neighborList already contains the list
// of neighbor boids
Vector2D Boid::keepDistance()
{
   Profile_Scope(boid_keep_distance);

   if(neighborList.size() == 0)
      return Vector2D(0,0);

   Vector2D change(0,0);

   list<Boid *>::iterator i;
   for(i = neighborList.begin(); i != neighborList.end(); i++)
   {
      Boid* pBoid = *i;

      // dist is a vector pointing to the other boid
      Vector2D dist = pBoid->m_2position - this->m_2position;
      // get the squared distance to the other boid
      float mag2 = dist.squared_magnitude();

      // check if the other boid is invading our comfort zone
      if(mag2 < SQUARED(BOID_COMFORT_RADIUS))
      {
```

```
            // okay we're too close, a collision may be immenent
            // decide how strongly we need to accelerate away
            float newmag = (SQUARED(BOID_COMFORT_RADIUS) - mag2) /
                     (SQUARED(BOID_COMFORT_RADIUS) -
                      SQUARED(BOID_DANGER_RADIUS));
            // cap it off at 1
            if(newmag > 1.0f) newmag = 1.0f;
            dist.setMagnitude(newmag);
            change -= dist;
         }
      }
      if(change.squared_magnitude() > 1)
         change.setMagnitude(1.0f);
      return change;
   }

   // adjusts velocity to match average of neighbors
   // assumes that neighborList already contains a list
   // of the neighbors
   Vector2D Boid::matchHeading()
   {
      Profile_Scope(boid_match_heading);

      // if there are no neighbors, nothing to do
      if(neighborList.size() == 0)
         return Vector2D(0,0);

      // we'll use ave to find the average velocity of the neighbors
      Vector2D ave(0,0);
      int count = 0;

      list<Boid *>::iterator i;
      for(i = neighborList.begin(); i != neighborList.end(); i++)
      {
         Boid* pBoid = *i;
         if(pBoid->team == this->team)
         {
            ave += pBoid->m_2velocity;
            count++;
         }
```

```
   }
   if(count == 0)
      return Vector2D(0,0);

   ave /= (float)count; // finish computing the average

   Vector2D change = (ave - this->m_2velocity) / (BOID_MAX_SPEED/2);
   if(change.squared_magnitude() > 1)
      change.setMagnitude(1);
   return change;
}

// maintains flock cohesion
// accelerates toward the average position of the neighbors
// assumes that neighborList is already filled with the list of neighbors
Vector2D Boid::steerToCenter()
{
   Profile_Scope(boid_steer_to_center);

   // if no neighbors, then nothing to do
   if(neighborList.size() == 0)
      return Vector2D(0,0);

   // we'll use ave to find the average position of the neighbors
   Vector2D ave(0,0);
   int count = 0;

   list<Boid *>::iterator i;
   for(i = neighborList.begin(); i != neighborList.end(); i++)
   {
      Boid* pBoid = *i;
      if(pBoid->team == this->team)
      {
         ave += pBoid->m_2position;
         count++;
      }
   }
   if(0 == count)
      return Vector2D(0,0);
```

```
   ave /= (float)count; // finish computing average

   Vector2D change = (ave - this->m_2position)/50;
   if(change.squared_magnitude() > 1)
      change.setMagnitude(1);
   return change;
}

// steers animal towards the goal
Vector2D Boid::gotoGoal()
{
   Profile_Scope(boid_goto_goal);

   // get a vector pointing towards the goal
   Vector2D newV = (parentFlock->team_goals[this->team] - m_2position);

   // adjust this vector to cap its magnitude off at BOID_CRUISING_SPEED
   // we'll use this new vector as our new target velocity
   if(newV.squared_magnitude() > SQUARED(BOID_CRUISING_SPEED))
      newV.setMagnitude(BOID_CRUISING_SPEED);

   // find out the difference between our current velocity and
   // our target velocity.  This will be our new adjustment
   Vector2D change = (newV - this->m_2velocity)/(BOID_MAX_SPEED/2);
   if(change.squared_magnitude() > 1)
      change.setMagnitude(1);
   return change;
}

// accessor function
// retrieves the current 2D position of the animal
Vector2D Boid::getPos() const
{
   return this->m_2position;
}

//////////////////////////////////////////////////////
// functions required for PhysicalEntity
D3DXVECTOR3 Boid::GetGroundContactPoint()
{
```

```
   D3DXVECTOR3 Result;
   Result = Position;
   Result.z -= BOID_RADIUS;
   return Result;
}

void Boid::GetMeshes(Mesh** MeshArray)
{
   MeshArray[0] = m_mesh;
}

int Boid::GetMeshCount()
{
   return 1;
}

unsigned int Boid::GetAction(float TimeStep, Msg** SentMsgs)
{
   Update(TimeStep);
   return 0;
}

unsigned int Boid::SendMsg(Msg* RecievedMsg, Msg** SentMsgs)
{
   return 0;
}
//////////////////////////////////////////////////////

//
void Boid::reset()
{
   m_2position = Vector2D(3 * (float(rand()) / float(RAND_MAX)) +3.5f,
            3 * (float(rand()) / float(RAND_MAX)) +3.5f);

   m_2velocity = Vector2D(0,0);
   m_2acceleration = Vector2D(0,0);

   Position = D3DXVECTOR3(m_2position.x, m_2position.y,
      terrain->GetHeight(m_2position.x, m_2position.y) + BOID_RADIUS*2);
```

```
    ////////////////////////////////////
    // Temporary addition for calculating energy expenditure
    lastPos = Position;
    energy = 0;
    ////////////////////////////////////////////////////
}
```

```
//////////////////////////////////////////////////////////////////////
// Flock.h: interface for the Flock class.
//   A flock contains a list of boids
//   The Flock is not responsible for deleting dynamically
//   allocated boids
// Joel Gompert
//////////////////////////////////////////////////////////////////////

#if !defined(AFX_FLOCK_H__AB70E4E2_1566_4EF1_8991_00EB8A769A65__INCLUDED_)
#define AFX_FLOCK_H__AB70E4E2_1566_4EF1_8991_00EB8A769A65__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include <list>
using namespace std;

#include "Entity.h"
#include "Mesh.h"
#include "Boid.h"

class Boid;

class Flock : public AbstractEntity
{
public:
   ////////////////////////////////
   // Constructors and destructors
   ////////////////////////////////
   Flock();
   virtual ~Flock();

   // a helper function to release the memory of all the boids in the flock
   void cleanup();

   ////////////////////
   // flocking functions
   ////////////////////
   void Update(float elapsed_time); // updates all members of the flock
```

```
   //////////////////////////
   // miscellaneous functions
   //////////////////////////

   void AddTo(Boid* boid); // adds a boid to the flock
   int GetCount() const {return members.size();}
      // get number of boids in the flock
   void RemoveFrom(Boid* boid);
      // removes a boid from the flock and releases the boid's memory

   Vector2D ave2Pos(int team); // gives the average 2D position of the flock
   D3DXVECTOR3 ave3Pos(int team);
      // returns the average 3D position of the flock

   ////////////////////////////////////////////////////
   // functions required for PhysicalEntity
   unsigned int GetAction(float TimeStep, Msg** SentMsgs);
   unsigned int SendMsg(Msg* RecievedMsg, Msg** SentMsgs);
   ////////////////////////////////////////////////////

   ////////////////////////////////////////////////////
   // Temporary addition for calculating energy expediture
   float aveEnergy();
   ////////////////////////////////////////////////////

   friend Boid;

   Vector2D team_goals[2];
private:

   list<Boid *> members; // the members of the flock
};

#endif // !defined(AFX_FLOCK_H__AB70E4E2_1566_4EF1_8991_00EB8A769A65__INCLUDED_)
```

```
///////////////////////////////////////////////////////////////////////
// Flock.cpp: implementation of the Flock class.
//   A flock contains a list of boids
// Joel Gompert
///////////////////////////////////////////////////////////////////////

#include <fstream> // for outputting energy data

#include "Flock.h"

///////////////////////////////////////////////////////////////////////
// Construction/Destruction
///////////////////////////////////////////////////////////////////////

Flock::Flock()
{
   // set up some example goal positions
   team_goals[0] = Vector2D(35,35);
   team_goals[1] = Vector2D(50,50);
}

Flock::~Flock()
{

}

// calls Update() on all members of the flock
void Flock::Update(float elapsed_time)
{
   list<Boid *>::iterator i;
   for(i = members.begin(); i != members.end(); i++)
   {
      (*i)->Update(elapsed_time);
   }
}

// adds a boid to the flock
void Flock::AddTo(Boid* boid)
{
   // there are two steps to this because the flock
```

```
   // and the boid must point to each other
   members.push_back(boid); // let the flock know about the boid
   boid->parentFlock = this; // let the boid know which flock it is in
}

// here's a helper function to release the memory of all the boids in the flock
void Flock::cleanup()
{
   while(!members.empty())
   {
      Boid *pBoid = members.front();
      members.pop_front();
      delete pBoid;
   }
}

// removes a boid from the flock and releases the boid's memory
void Flock::RemoveFrom(Boid* boid)
{
   // first we have to find this boid in the list
   list<Boid *>::iterator i;
   for(i = members.begin(); i != members.end(); i++)
   {
      Boid *pBoid = *i;

      if(pBoid == boid)
      {
         // here it is, remove it and delete it
         members.erase(i);
         delete pBoid;
         return;
      }
   }
   return;
}

// gives the average 2D position of the flock
Vector2D Flock::ave2Pos(int team)
{
   Vector2D ave(0,0);
```

```
   int count = 0;

   list<Boid *>::iterator i;
   for(i = members.begin(); i != members.end(); i++)
   {
      Boid *pBoid = *i;
      if(team == pBoid->team)
      {
         ave += pBoid->getPos();
         count++;
      }
   }

   return (ave / (float)count);
}

// returns the average 3D position of the flock
D3DXVECTOR3 Flock::ave3Pos(int team)
{
   D3DXVECTOR3 ave(0,0,0);
   int count = 0;

   list<Boid *>::iterator i;
   for(i = members.begin(); i != members.end(); i++)
   {
      Boid *pBoid = *i;
      if(team == pBoid->team)
      {
         Vector2D p = pBoid->getPos();
         ave += D3DXVECTOR3(p.x, p.y,
               pBoid->terrain->GetHeight(p.x, p.y));
         count++;
      }
   }

   return (ave / (float)count);
}

//////////////////////////////////////////////////////
// functions required for PhysicalEntity
```

```
unsigned int Flock::GetAction(float TimeStep, Msg** SentMsgs)
{
   const float some_error_constant = 10.0f;

   // here we can check if the flock has reached its goal.
   // if the average position of the flock is within some
   // error constant distance from the goal
   //if(aveEnergy() > 350 || (ave2Pos(0) - team_goals[0]).squared_magnitude() < some_
   //{
   // // output the current average energy expended
   // ofstream fout;
   // fout.open("energy.dat", ios_base::app);
   // fout << aveEnergy() << '\t';
   //

   // Boid *pBoid = (Boid *)members.front();
   //
   // pBoid->terrain->Initialize(40, 40);
   // pBoid->terrain->NewRandomSmoothTerrain();
   // pBoid->terrain->NormalizeHeight();

   // float df = pBoid->TERRAIN_DRAG_FACTOR;
   // df += 12.5f;
   // if(df > 75.0f)
   // {
   //    fout << endl;
   //    df = 0;
   // }

   // fout.close();

   // list<Boid *>::iterator i;
   // for(i = members.begin(); i != members.end(); i++)
   // {
   //    pBoid = *i;
   //    pBoid->reset();
   //    pBoid->TERRAIN_DRAG_FACTOR = df;
   // }
   //}
```

```
   return 0;
}
unsigned int Flock::SendMsg(Msg* RecievedMsg, Msg** SentMsgs)
{
   return 0;
}
//////////////////////////////////////////////////////

//////////////////////////////////////////////////////
// Temporary addition for calculating energy expediture
float Flock::aveEnergy()
{
   float ave = 0;
   int count = 0;

   list<Boid *>::iterator i;
   for(i = members.begin(); i != members.end(); i++)
   {
      Boid *pBoid = *i;
      if(0 == pBoid->team)
      {
         ave += pBoid->energy;
         count++;
      }
   }

   return (ave / (float)count);
}
//////////////////////////////////////////////////////
```

References

[1] David M. Bourg. *Physics for Game Developers*. O'Reilly & Associates, 2001.

[2] Gary William Flake. *The Computational Beauty of Nature*. The MIT Press, Cambridge, Massachusetts, 1999.

[3] Joel Gompert. Flocking over 3d terrain. Technical report, University of Nebraska-Lincoln, 2002.

[4] Sanjiv Kapoor. Efficient computation of geodesic shortest paths. In *ACM Symposium on Theory of Computing*, 1999.

[5] Christopher Kline. web.media.mit.edu/~ckline/cornellwww/boid/Boid.c++. Cornell University.

[6] A. E. Minetti. Mechanical determinants of grade walking energetics. *Journal of Physiology*, 472:725–735, 1993.

[7] A.E. Minetti and R. McN. Alexander. A theory of metabolic costs for bipedal gait. *Journal of Theoretical Biology*, 186:467–476, 1997.

[8] Craig W. Reynolds. Flocks, herds, and schools: A distributed behavior model. In *SIGGRAPH '87 Conference Proceedings*, pages 25–34, 1987.

[9] Craig W. Reynolds. Not bumping into things: Notes on "obstacle avoidance" for the course on physically based modeling. In *SIGGRAPH '88 Conference Proceedings*, 1988.

[10] Craig W. Reynolds. Steering behaviors for autonomous characters. In *Proceedings of Game Developers Conference*, pages 763–782, San Francisco, California, 1999. Miller Freeman Game Group. http://www.red3d.com/cwr/papers/1999/gdc99steer.html.