CS 428 – Notes on Procedural Animation for Project 3

Doug DeCarlo

These notes describe a method for the procedural animation of *agents* (or "critters") that will appear to wander around as if they're alive. These methods come from the area of computer graphics that overlaps with the study of artificial life [1]. Methods such as these are in common use in games, simulations and even production.

The complexity of the motion originates from a *model* that describes how the agents respond to their environment. Quite simple models can produce very believable behavior, such as Reynold's model for flocking behavior [3], which served as the basis used to largely automate the wildebeest stampede sequence in Disney's *The Lion King*. Clearly, models that are used in games like *The Sims* are much more intricate, as they must encapsulate a great deal of world knowledge, incorporate some notion of agent perception, and have a fun interface for their control.

The following describes a model that produces believable wandering motions, suitable for the animation of crawling bugs. It includes the following aspects:

- avoiding collisions with objects in the scene
- wandering

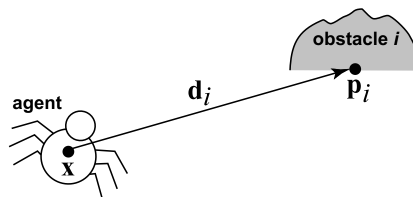
The model used here is an analogy to physics: agents move like particles in a force field. (All masses are set to unity, so the model deals with accelerations instead of forces.) Behaviors are realized by accelerations "induced" by the environment. For example, an acceleration that points away from an object (and increases in strength as that object is approached) produces a behavior that avoids collisions with that object. While behaviors of real organisms don't work this way (one would hope!), this model is still a sound engineering tool for producing realistic behaviors. An animation is produced by integrating the differential equations of motion, which define the acceleration $\mathbf{a}(t)$ as the second derivative of the position $\mathbf{x}(t)$ with respect to time t:

$$\mathbf{a}(t) = \frac{d^2\mathbf{x}(t)}{dt^2}$$

Both behaviors listed above can be described using accelerations that are determined based on the direction and distance to a scene object i, as:

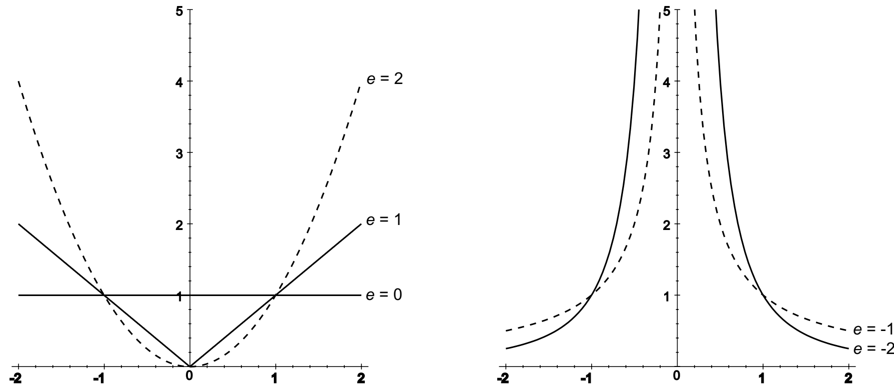
$$\mathbf{a}_i(t) = k_i \frac{\mathbf{d}_i}{\|\mathbf{d}_i\|} \|\mathbf{d}_i\|^{e_i}$$

where $\mathbf{d}_i$ is the vector that stretches from the agent's position $\mathbf{x}$ to the location $\mathbf{p}_i$ of object i as in the following:



In this case, $\mathbf{d}_i = \mathbf{p}_i - \mathbf{x}$. The formula for $\mathbf{a}_i(t)$ defines (as a function of $\mathbf{x}$) an *acceleration field* where the accelerations radiate from $\mathbf{p}_i$ (they point towards $\mathbf{p}_i$ when k_i is positive, and away when k_i is negative). In fact, k_i is simply a scaling constant for the acceleration field for object i. The exponent e_i controls the the relative magnitudes of the accelerations based on the distance $\|\mathbf{d}_i\|$ from the agent to the obstacle as $\|\mathbf{d}_i\|^{e_i}$. As seen in the graphs below, positive values of e_i produce zero accelerations

at the object; negative values of e_i produce larger accelerations as the object is approached; $e_i = 0$ produces accelerations with constant magnitude (with a singularity at the object itself).



Acceleration plots for various exponents e

Negative values of e_i are useful for obstacle avoidance (they also have negative k_i), while positive values of e_i are useful for attracting the agent to an object. This is because it's only worth avoiding an object that is nearby, while it's only worth seeking a desired object that is far away.

A suitable wandering behavior can be constructed using attraction. One way involves creating an attractive location in the scene that moves around randomly (at regular intervals). By confining the random location to a certain area, this can also serve to confine the agent to that area of the environment.

Integration

With all the accelerations computed, they are added up to produce the total acceleration at time t:

$$\mathbf{a}(t) = \sum_i \mathbf{a}_i(t)$$

This is numerically integrated (twice) to get the position. The differential equation given above is a second-order equation (it involves second derivatives); but can be written as two coupled first-order equations as:

$$\begin{pmatrix} \mathbf{a}(t) \\ \mathbf{v}(t) \end{pmatrix} = \frac{d \begin{pmatrix} \mathbf{v}(t) \\ \mathbf{x}(t) \end{pmatrix}}{dt}$$

(This means we need to maintain values of the position *and* velocity at any particular moment.) These first-order equations can be solved using the Euler integration method, which uses linear approximations to these functions. (Substantially better methods are available, but are also more complicated [2].) This leads to the following, which determine the velocity and position at the later time $t + \Delta t$ given values of $\mathbf{x}(t)$, $\mathbf{v}(t)$ and $\mathbf{a}(t)$:

$$\begin{aligned} \mathbf{v}(t+\Delta t) &= \mathbf{v}(t) + \mathbf{a}(t)\Delta t \\ \mathbf{x}(t+\Delta t) &= \mathbf{x}(t) + \mathbf{v}(t)\Delta t \end{aligned}$$

Getting it working

Euler integration has its problems—a sufficiently small value of Δt is required (the solution will diverge, otherwise). When the animation is real-time (with t measuring the actual time elapsed), it's

very possible that several integration steps will be required to bring the environment up-to-date. This avoids use of a single (perhaps too large) Δt.

Euler integration has another problem—its linear approximations lead to increased "energy" of the system. If left on its own, the velocities produced by the simulation will steadily increase, resulting in an incorrect (but amusing) animation. The simplest correction is to damp the system, which takes a little energy out (like friction). Adding in an acceleration (which is equivalent to a viscous drag force) that points in the opposite direction of the velocity is sufficient:

$$\mathbf{a}_d(t) = -k_d \mathbf{v}(t) \qquad k_d > 0$$

Even with this damping, another problem can occur. For negative values of e_i, the acceleration increases very quickly as the object is approached. Sometimes an object can be approached too closely (such as when Δt is a little too large), and a *huge* acceleration is produced that sends the agent flying out past Neptune. One possible solution to this problem enforces an upper bound on the magnitude of $\mathbf{a}(t)$.

There are a lot of parameters to adjust here; each scene object has its own k_i and e_i, and the enire scene has a value for k_d, and the upper bound on the acceleration magnitude. A lot of "tuning" is involved to get this working nicely. One difficulty is in getting the obstacle avoidance accelerations strong enough to prevent slight object collisions. As the object (or the agent) gets larger, this becomes more apparent, as right now, they are both modeled as points. One possibility would be to modify the computation of $\mathbf{d}_i$ to be the distance between the closest points on the agent and object (or at least, an approximation to this which models the agent and object as circles).

References

[1] Steven Levy. Artificial Life: A Report from the Frontier Where Computers Meet Biology. Vintage Books, 1993.

[2] W. Press, S. Teukolsky, W. Vetterling and B. Flannery. Numerical Recipes in C: The Art of Scientific Computing. Cambridge University Press, 1992.

[3] Craig W. Reynolds. Flocks, Herds, and Schools: A Distributed Behavioral Model. In *Computer Graphics*, 21(4), SIGGRAPH 1987 Conference Proceedings, pp. 25–34.