
Recap: In the previous lectures, we have discussed techniques for computing the motion of a single agent. Today we will discuss techniques for planning and coordinating the motion of multiple agents. In particular, we will discuss three aspects of this problem:

- **Particle systems**: Modeling (unintelligent) motion for a collection of objects
- **Flocking behavior**: Where all agents are moving as a unit
- **Crowd behavior**: Where a number of agents, each with their own desired destination, are to move without colliding with each other.

Today, we will discuss particle systems and flocking behavior.

**Particle Systems**: A *particle system* is a technique that uses a large number of small graphical artifacts, called *particles*, to create a large-scale, typically “fuzzy,” visual effect. Examples of applications of particle systems include many amorphous natural phenomena such as fire, smoke, water, clouds, and explosions. In these examples particles are born and die dynamically, but there are also variants of particle systems where particles are persistent. These are used to produce static phenomena such as galaxies or “stringy” phenomena such as hair, grass, and other plants.

A fundamental principle that underlies particle systems is that our brains tend to cluster an aggregation of similar objects into the impression of a single impression. Thus, the many individual water drops that cascade down a flowing fountain are perceived as a flowing stream of water (see Fig. 1). (Note that particle systems do not really belong in this lecture on artificial intelligence, since their behavior is based solely on physical laws, and not on any model of intelligent behavior. Nonetheless, we have decided to discuss them here because they do represent a common technique for generating interesting patterns of movement in games and other applications of interactive computer graphics.)

Particle systems are almost as old as computer games themselves. The very earliest 2-dimensional games, such as *Spacewar!* and *Asteroids* simulated explosions through the use of a particle system. The power of the technique, along the term “particle system,” came about in one of the special effects in the second Star Trek movie, where a particle system was used to simulate a fire spreading across the surface of a planet (the genesis effect).

**How Particle Systems Work**: One of the appeals of particle systems is that they are extremely simple to implement, and they are very flexible. Particles are created, live and move according
to simple rules, and then die. The process that generates new particles is called an *emitter*. For each frame that is rendered, the following sequence of steps is performed:

**Emission:** New particles are generated and added to the system. There may be multiple sources of emission (and particles themselves can serve as emitters).

**Attributes:** Each new particle is assigned its (initial) individual properties that affect how the particle moves over time and is rendered. These may change over time and include the following:

- **Geometric attributes:** initial position and velocity
- **Graphics attributes:** shape, size, color, transparency
- **Dynamic attributes:** lifetime, influence due to forces such as gravity and friction

**Death:** Any particles that have exceeded their prescribed lifetime are removed from the system.

**Movement:** Particles are moved and transformed according to their dynamic attributes, such as gravity, wind, and the density of nearby particles.

**Rendering:** An image of the surviving particles is rendered. Particles are typically rendered as a small blob.

In order to create a natural look, the process of emitting particles and the assignment of their attributes is handled in the probabilistic manner, where properties such as the location and velocity of particles being determined by a random number generator. Many game engines (including Unity) provide flexible systems for generating particle systems, offering a multitude of options that can be set by the designer.

Particle system can be programmed to execute any set of instructions at each step, but usually the dynamic properties of particles are very simple, and do not react in any complex manner to the presence of other particles in the system. Because the approach is procedural, it can incorporate any computational model that describes the appearance or dynamics of the object. For example, the motions and transformations of particles could be tied to the solution of a system of partial differential equations. In this manner, particles can be used to simulate physical fluid phenomena such as water, smoke, and clouds.

**Updating Particles in Time:** There are two common ways to update particles over time.
Closed-form function: Every particle is represented by a parametric function of time (and coefficients that are given by the particle’s initial motion attributes. For example, given the particle’s initial position $p_0$, its initial velocity vector $\vec{v}_0$, and some fixed field force $\vec{g}$ (representing, for example, the affect of gravity as a vector that points downwards) the position of the particle at time $t$ can be expressed in closed form as:

$$p(t) = p_0 + \vec{v}_0 t + \frac{1}{2} \vec{g} t^2.$$  

(If you have ever taken a course in physics, this formula will be familiar to you. If not, don’t worry how it was derived. It is a simple consequence of Newton’s basic laws of motion.) On the positive side, this approach requires no storage of the evolving state of the particle, just its initial state and the elapsed time. On the negative side, this approach is very limited, and does not allow particles to respond to each other or their environment.

Discrete physical integration: In this type of system, each particle stores in current physical state. This state consists of the particle’s current position, which is given by a point $p$, its current velocity, which is given by a vector $\vec{v}$, and its current acceleration, which is given by a vector $\vec{a}$.

Acceleration can be thought of as the accumulated effect of all the forces acting on the particle. (For example, the force of gravity decreases the vertical component of the velocity. On the other hand, the force of air resistance or friction tends to decrease the particle’s absolute velocity, without altering its direction.) We can then update the state over a small time interval, $\Delta t$. Think of $\Delta t$ as the elapsed time between consecutive frames, or a fixed update time, such as 0.1 seconds. By the basic laws of kinematics, (1) the total vector sum $\vec{F}$ of forces alters acceleration, (2) acceleration changes the object’s velocity over $\Delta t$, and (3) velocity changes the object’s position in space over $\Delta t$:

$$\begin{align*}
(1) \quad \vec{a} &\leftarrow \frac{\vec{F}}{m}, \\
(2) \quad \vec{v}' &\leftarrow \vec{v} + \vec{a} \cdot \Delta t \quad \text{and} \quad (3) \quad p' &\leftarrow p + \vec{v} \cdot \Delta t,
\end{align*}$$

where $p'$ and $\vec{v}'$ denote the particle’s new position and velocity, respectively. (Note that, except for time, these are 3-dimensional vector quantities.)

Doing this in Unity: The Unity physics engine will take care of these operations automatically, whenever you attach to it a Rigidbody component (and assuming that isKinematic is not enabled). When isKinematic is disabled (the object is under control of the physics engine) then the body can be moved by applying forces.

```csharp
Rigidbody rb = GetComponent < Rigidbody > ();
rb.AddForce(Vector3.up * 10f); // apply an upward force
```

The function AddForce has a second optional argument that controls the manner in which the force is applied. These include

\footnote{When dealing with particles, it is common to ignore the object’s mass. In general, the mass $m$ of an object is its resistance to change its velocity as a consequence of a force. The acceleration $\vec{a}$ due to a force $\vec{F}$ is given by $\vec{a} = \frac{\vec{F}}{m}$, which is derived from the well-known formula $\vec{F} = m \vec{a}$. Note that acceleration is a vector quantity, where the direction is given by the direction of the force that is acting on the body.}
• Force: Add a continuous force to the rigidbody, using its mass
• Acceleration: Add a continuous acceleration to the rigidbody, ignoring its mass
• Impulse: Add an instant force impulse to the rigidbody, using its mass
• VelocityChange: Add an instant velocity change to the rigidbody, ignoring its mass

There is also a function AddTorque that adds a rotational force. The argument is a vector quantity, where the direction gives the axis of rotation and the magnitude gives the strength of the force (according to the right-hand rule, I believe).

When isKinematic is enabled (not under the control of the physics engine) the object can be moved either by changing transform.position or rigidbody.position directly. The Unity manual says that the latter is more efficient. The same applies to transform.rotation and rigidbody.rotation. Altering the position and rotation will cause the object to “teleport” instantly to the new position. Unity manual also explains that there is a smoother way to apply motions to kinematic rigid bodies, through the functions movePosition and moveRotation.

```csharp
void Update () {
    Vector3 velocity = ... // current linear velocity
    Vector3 angVeloc = ... // current (Euler) angular velocity
    float DeltaT = Time.deltaTime; // elapsed time
    rb.MovePosition(transform.position + velocity * DeltaT);
    Quaternion deltaRot = Quaternion.Euler(angVeloc * DeltaT);
    rb.MoveRotation(rb.rotation * deltaRot);
}
```

**Flocking Behavior:** Next, let us consider the motion of a slightly smarter variety, namely *flocking*. We refer to flocking behavior in the generic sense as any motion arising when a group of agents adopt a decentralized motion strategy designed to hold the group together. Such behavior is exemplified by the motion of groups animals, such as birds, fish, insects, and other types of herding animals (see Fig. 2).

![Fig. 2: An example of complex emergent behavior in flocking.](image)

In contrast to full crowd simulation, where each agent may have its own agenda, in flocking it is assumed that the agents are homogeneous, that is, they are all applying essentially the
same motion update algorithm. The only thing that distinguishes one agent from the next is their position relative to the other agents in the system. It is quite remarkable that the complex formations formed by flocking birds or schooling behavior in fish can arise in a system in which each creature is following (presumably) a very simple algorithm. The apparently spontaneous generation of complex behavior from the simple actions of a large collection of dynamic entities is called *emergent behavior*. While the techniques that implement flocking behavior do not involve very sophisticated models of intelligence, variants of this method can be applied to simple forms of crowd motion in games, such as a crowd of people milling around in a large area or pedestrians strolling up and down a sidewalk.

**Boids:** One of the earliest models and perhaps the best-known model for flocking behavior was given by C. W. Reynolds from 1986 with his work on “boids.” (The term is an intentional misspelling of “bird.”) In this system, each agent (or *boid*) determines its motion based on a combination of four simple rules:

**Separation:** Each boid wishes to avoid collisions with other nearby boids. To achieve this, each boid generates a *repulsive potential field* whose radius of influence extends to its immediate neighborhood. Whenever another boid gets too close, the force from this field will tend to push them apart.

**Alignment:** Each boid’s direction of flight is aligned with nearby boids. Thus, local clusters of boids will tend to point in the same direction and hence will tend to fly in the same direction.

**Avoidance:** Each boid will avoid colliding with fixed obstacles in the scene. At a simplest level, we might imagine that each fixed obstacle generates a repulsive potential field. As a boid approaches the object, this repulsive field will tend to cause the boid to deflect its flight path, thus avoiding a collision. Avoidance can also be applied to predators, which may attack the flock. (It has been theorized that the darting behavior of fish in a school away from a shark has evolved through natural selection, since the sudden chaotic motion of many fish can confuse the predator.)

**Cohesion:** Effects such as avoidance can cause the flock to break up into smaller subflocks. To simulate the flocks tendency to regroup, there is a force that tends to draw each boid

![Fig. 3: Forces that control flocking behavior.](image-url)
towards the center of mass of the flock. (In accurate simulations of flocking motion, a boid cannot know exactly where the center of mass is. In general the center of attraction will be some point that the boid perceives being the center of the flock.)

**Boid Implementation:** Next let us consider how to implement such a system. We apply the same discrete integration approach as we did used for particle systems. In particular, we assume that each boid is associated with a state vector \((p, \vec{v})\) consisting of its current position \(p\) and current velocity \(v\). (We assume that the boid is facing in the same direction as it is flying, but, if not, a vector describing the boid’s angular orientation can also be added to the state.) We think of the above rules as imposing forces, which together act to define the boid’s current acceleration. Given this acceleration vector \(a\) caused by the boid forces, we apply the update rules described earlier for particle systems:

\[
\vec{v}' \leftarrow \vec{v} + \vec{a} \cdot \Delta t \quad \text{and} \quad p' \leftarrow p + \vec{v} \cdot \Delta t,
\]

in order to obtain the new position \(p'\) and new velocity vector \(\vec{v}'\).

How are these forces computed? First observe that, for the sake of efficiency, you do not want the behavior of each boid to depend on the individual behaviors of the \(n - 1\) boids, since this would be much to costly, taking \(O(n^2)\) time for each iteration. Instead, the system maintains the boids stored in a *spatial data structure*, such as a grid or quadtree, which allows each boid to efficiently determine the boids that are near to it. Rules such as separation, avoidance, alignment can be based on a small number of nearest neighbor boids. Cohesion requires knowledge of the center of the flock, but this can be computed once at the start of each cycle in \(O(n)\) time.

Since these are not really natural forces, but rather heuristics that are used to guiding motion, it is not essential to apply kinematics rigorously in order to determine future motion. Rather, each rule naturally induces a directional influence. (For example, the force of avoidance is directed away from the anticipated point of impact while the force for alignment is in the average direction that nearby boids are facing.) Also, each rule can be associated with a strength whose magnitude depends, either directly or inversely, on the distance from the point of interest. (For example, avoidance forces are very strong near the obstacle but drop off rapidly. In contrast, the cohesive force tends to increase slowly as the distance from the center of flock increases.) Thus, given a unit vector \(\vec{u}\) pointing in the induced direction and the strength \(s\) given as a scalar, we can compute the updated acceleration vector as \(\vec{a} = c \cdot s \cdot \vec{u}\), where \(c\) is a “fudge factor” that can be adjusted by the user that is used to model other unspecified physical quantities such as the boid’s mass.

One issue that arises with this or any dynamical system is how to avoid undesirable (meaning unnatural looking) motion, such as collisions, oscillations, and unrealistically high accelerations. Here are two approaches:

**Prioritize and truncate:** Assume that there is a fixed maximum magnitude for the acceleration vector (based on how fast a boid can change its velocity based on what sort of animal is being modeled). Sort the rules in priority order. (For example, predator/obstacle avoidance is typically very high, flock cohesion is low.) The initial acceleration
vector is the zero vector (meaning that the boid will simply maintain its current velocity). As each rule is evaluated, compute the associated acceleration vector and add it to the current acceleration vector. If the length of the acceleration vector ever exceeds the maximum allowed acceleration, then stop and return the current vector.

**Weight and clamp:** Assign weights to the various rule-induced accelerations. (Again, avoidance is usually high and cohesion is usually low.) Take the weighted sum of these accelerations. If the length of the resulting acceleration vector exceeds the maximum allowed acceleration, then scale it down.

The first method has the virtue that, subject to the constraint on the maximum acceleration, it processes the most urgent rules first. The second has the virtue that every rule has some influence on the final outcome. Of course, since this is just a heuristic approach, the developer typically decides what sort of approach yields the most realistic results.