Animation – A broad Brush

Traditional Methods
- Cartoons, stop motion

Keyframing
- Digital inbetweens

Motion Capture
- What you record is what you get

Simulation
- Animate what you can model (with equations)

Computer Animation

Animation Techniques

Keyframing

From “The computer in the visual arts”, Spalter, 1999

Keyframing

Traditional animation technique
Dependent on artist to generate ‘key’ frames
Additional, ‘inbetween’ frames are drawn automatically by computer

How are we going to interpolate?

Figure 10.4 Three keyframes. Three keyframes representing a ball on the ground, at its highest point, and back on the ground.

From “The computer in the visual arts”, Spalter, 1999
Linear Interpolation

Simple, but discontinuous velocity

Nonlinear Interpolation

Smooth ball trajectory and continuous velocity, but loss of timing

Easing

Adjust the timing of the inbetween frames. Can be automated by adjusting the stepsize of parameter, t.

Style or Accuracy?

Interpolating time captures accuracy of velocity
Squash and stretch replaces motion blur stimuli and adds life-like intent
Traditional Motivation
Ease-in and ease-out is like squash and stretch
Can we automate the inbetweens for these?

Animation Techniques

Procedural Techniques

Examples
Inanimate video game objects
- GT Racer cars
- Soapbox about why this is so cool
Special effects
- Explosions, water, secondary motion
- Phantom Menace CG droids after they were cut in half
**Procedural Animation**

Very general term for a technique that puts more complex algorithms behind the scenes.

Technique attempts to consolidate artistic efforts in algorithms and heuristics.

Allows for optimization and physical simulation.

---

**Procedural Animation Strengths**

- Animation can be generated ‘on the fly’
- Dynamic response to user
- Write-once, use-often
- Algorithms provide accuracy and exhaustive search that animators cannot

---

**Procedural Animation Weaknesses**

- We’re not great at boiling human skill down to algorithms
  - How do we move when juggling?
- Difficult to generate
- Expensive to compute
- Difficult to force system to generate a particular solution
  - Bicycles will fall down

---

**Particle Systems**

- Particle systems provide a powerful framework for animating numerous similar elementary “objects” at the same time. Those objects are called particles. Using a lot of particles with simple physics allow us to model complex phenomena such as:
  - Fireworks
  - Waterfalls
  - Smoke
  - Fire
  - Flocking
  - Clothes, etc.
Introduction

Typical Particle system animation routine

ParticleSystem()

1. Animate a particle System
2. **While** animation not finished
3. **Do** Delete expired particles
4. Create new particles
5. Simulate Physics
6. Update particle attributes
7. Render particles

Particle

typedef struct // Create A Structure For Particle

{ bool active; // Active (Yes/ No)
  float life; // Particle Life
  float fade; // Fade Speed
  float r; // Red Value
  float g; // Green Value
  float b; // Blue Value
  float x; // X Position
  float y; // Y Position
  float z; // Z Position
  float xi; // X Direction
  float yi; // Y Direction
  float zi; // Z Direction
  float xg; // X Gravity
  float yg; // Y Gravity
  float zg; // Z Gravity
}

particles; // Particles Structure
Example - Firework

During the rocket phase, all particles flock together. The speed of the particles inside the illusory rocket is determined by the initial launch speed to which we subtract the influence of gravity.

Physics

\[ F = m \cdot a \]
\[ a = \frac{F}{m} \]
\[ a = g = 9.81 \text{ m/s} \]
\[ a(t + dt) = - gz \] where \( z \) is upward unit vector
\[ v(t+dt) = v(t) + a(t) \, dt \]
\[ x(t+dt) = x(t) + v(t)dt + \frac{1}{2} a(t^2)dt \]

Particle system - Applications

Using this general particle system framework, there are various animation effects that can be simulated such as force field (wind, pressure, gravity), viscosity, collisions, etc.

Rendering particles as points is straightforward, but we can also draw tiny segments for giving the illusion of motion blur, or even performing ray casting for obtaining volumetric effects.
The QuadParticles Class

Although many particle systems can be modeled with points and lines, moving to quadrilaterals (quads) combined with textures allows many more interesting effects. The texture can contain extra surface detail, and can be partially transparent in order to break up the regularity of the quad shape.

A quad can be assigned a normal and a Material node component to allow it to be affected by lighting in the scene. The only danger with these additional features is that they may slow down rendering by too much. For example, we want to map the texture to each quad (each particle), but do not want to use more than one QuadArray and one Texture2D object.

Forces

\[ A = \frac{F}{m} \]

- Particle masses won’t change
- But need to evaluate \( F \) at every time step.
- The force on one particle may depend on the positions of all the others

Typically, have multiple independent forces.

- For each force, add its contribution to each particle.
- Need a force accumulator variable per particle.
- Or accumulate force in the acceleration variable, and divide by \( m \) after all forces are accumulated
Example forces
- Earth gravity, air resistance
- Springs, mutual gravitation
- Force fields
  - Wind
  - Attractors/Repulsors
  - Vortices

Earth Gravity
- \( f = -9.81 \times \text{particle mass in Kg} \times Y \)

Drag
- \( f = -k \times v \)

Uniform Wind
- \( f = k \)

Simple Random Wind
- After each timestep, add a random offset to the direction

Noisy Random Wind
- Acts within a bounding box
- Define a grid of random directions in the box
- Trilinear interpolation to get \( f \)
- After each timestep, add a random offset to each direction and renormalize

Attractors/Repulsors
- Special force object at position \( x \)
- Only affects particles within a certain distance
- Within the radius, distance-squared falloff
  - if \( |x-p| < d \)
    - \( v = (x-p)/(x-p) \)
    - \( f = \pm k/|x|^2 \times x \)
  - else
    - \( f = 0 \)
- Use the regular grid optimization from lecture
**Emitters**

**What is it?!**
- Object with position, orientation
- Regulates particle “birth” and “death”
- Usually 1 per particle system
  - More than 1 can make controlling particle death inconvenient

**Emitters**

**Regulating particles**
- At “birth,” reset the particle’s parameters
  - Free to set them arbitrarily!
- For “death,” a few possibilities
  - If a particle is past a certain age, reset it.
  - Keep an index into the particle array, and reset a group of K particles at each timestep.
- Should allocate new particles only once!
  - Recycle their objects or array positions.

**Emitters**

**Fountain**
- Given the emitter position and direction, we have a few possibilities:
  - Choose particle velocity by jittering the direction vector
  - Choose random spherical coordinates for the direction vector

**render()**
- Store rotation info per-particle
- Keep meshes facing “forward” along their paths
- Can arbitrarily pick “up” vector

**Demo**
- [http://www.delphi3d.net/download/vp_sprite.zip](http://www.delphi3d.net/download/vp_sprite.zip)
Rendering

Render billboards
- Want to represent particles by textures
- Should always face the viewer
- Should get smaller with distance
- Want to avoid OpenGL’s 2d functions

Render billboards (one method)
- Draws an image-plane aligned, diamond-shaped quad
- Given a particle at p, and the eye’s basis (u,v,w), draw a quad with vertices:
  q0 = eye.u
  q1 = eye.v
  q2 = -eye.u
  q3 = -eye.v
- Translate it to p
- Will probably want alpha blending enabled for smoke, fire, pixie dust, etc. See the Red Book for more info.

Simulation Loop Recap

A recap of the loop:
- Initialize/Emit particles
- Run integrator (evaluate derivatives)
- Update particle states
- Render
- Repeat!

Particle Illusion Demo
- www.wondertouch.com