Particles
Simulation Homework

• Build a particle system based either on F=ma or procedural simulation
  – Examples: Smoke, Fire, Water, Wind, Leaves, Cloth, Magnets, Flocks, Fish, Insects, Crowds, etc.

• Simulate a Rigid Body
  – Examples: Angry birds, Bodies tumbling, bouncing, moving around in a room and colliding, Explosions & Fracture, Drop the camera, Etc...
Particle

• A particle is simply a point in space with some attributes
• The attributes are what makes different kinds of particles
  – Mass (m)
  – Position (x)
  – Velocity (v)
  – External Force (F)
  – Color, Animal type, Etc.
Particle Motion

• **Dynamic**
  – A particle with a non-zero initial velocity tends to keep moving with that velocity (Newton’s 1st Law)
  – Its motion changes whenever unbalanced external forces are applied to it (Newton’s 2nd Law)

• **Kinematic**
  – An “infinite mass” particle can move along a prescribed path or animated curve directed by an artist
    • **Static** – a kinematic particle with zero velocity
Newton’s Second Law

- The net force on an object is equal to the rate of change of its linear momentum \( P=mv \)

\[
F = \frac{dP}{dt} = \frac{d(mv)}{dt} = ma
\]

- The last equality holds if the object has constant mass
Newton’s Second Law

\[ F = ma = m\ddot{x} \]

• This is a second order differential equation in position

• Higher order differential equations can be analyzed and solved by rewriting them as a system of first order equations:

\[ \dot{x} = v \]
\[ \dot{v} = F/m \]
Forces
Types of Forces

• A particle system that simulates water requires gravity as an external force, as well as internal forces for incompressibility and advection

• Dust particles (or leaves) require air currents and wind as external forces

• If particles are used to model cloth, we require elastic or spring forces between them

• If each particle is fish, they need attractive forces to school and repulsive forces to avoid collisions

• Etc...
Types of forces

• Constant forces (e.g. gravity)

• Time dependent forces (e.g. wind)

• Position dependent forces (e.g. force fields, spatially varying wind)

• Velocity dependent forces (e.g. drag, friction)

• Position & Velocity dependent forces (e.g. springs)
Gravity

- $F_{grav} = -mg$
  - $g = 9.8 \text{ m/s}^2$ is a constant
  - $m$ is the mass of the body/particle

- Simple ballistic motion...
Wind

- Position and time dependent force
- $f_{\text{wind}} = f(\mathbf{x}, t)$
Magnetism

- Assign the particles a magnetic monopole attribute $q$

- $|f_{\text{magnet}}| = \frac{\mu q_1 q_2}{4\pi r^2}$
  - $q_1$ and $q_2$ are magnitudes of magnetic monopoles, $r$ is the distance between the poles, and $\mu$ is a constant
  - Also need to add a direction between particles
  - Like poles repel and unlike poles attract
Drag Force

- Velocity dependent force (linear in velocity)
- The faster the velocity, the larger the drag
  - think molasses or honey

\[ f_{drag} = -k_{drag} \nu_{rel} \]
- where \( k_{drag} \) is the drag coefficient, and \( \nu_{rel} \) is the particle’s velocity relative to the fluid it is in
Spring Force (no damping)

- Hooke’s Law
- \( F_{spring} = -kx \)
- Linearization of the spring forces for small displacements
Spring Force (with damping)

\[ F_{spring} = -kx - kd \dot{x} \]

- Adds an exponential decay to the amplitude of oscillation
- It is a good practice to add some damping to physical systems to keep them from going unstable
  - and for realism
Question #1

LONG FORM:
• Briefly discuss various types of forces that can be used in video game simulations
• Answer short form question below

SHORT FORM:
• Can you think of a use for forces in your video game? Briefly explain
• (Notice I’m starting to assume you have a video game idea. Do you? ☺️)
Collision Detection
Collisions

- As particles move around under the influence of gravity, drag, and other forces, how do they interact with other objects?
- This is where collisions come into play
- How do we detect collisions?
  - Check to see if a particle is inside some object
Example: Plane

- Consider for example using a plane to represent the ground (or a wall)
- Define the plane by a point $\hat{p}$ and normal $\hat{n}$
- Given our particle position $\hat{x}$, we calculate
  $$s = (\hat{x} - \hat{p}) \cdot \hat{n}$$
- $\hat{x}$ is outside the plane if $s > 0$ and inside if $s < 0$
- Normal to the plane/object is given by $\hat{n}$
Example: Box

- Use a plane for each of the six faces of the box.
- If the particle is inside all 6 faces, it is inside the box.

- To find the normal, one has to identify the closest of the 6 planes.
- This is given by the value of $s$ closest to zero.

- Can be used for other convex polyhedra as well.
Example: Sphere

- Define a sphere with a center \( \hat{c} \) and a radius \( r \)
- Given a point \( \hat{q} \), calculate \( s = |\hat{q} - \hat{c}| - r \).
- \( \hat{q} \) is outside the sphere if \( s > 0 \) and inside if \( s < 0 \)
- Normal at the point is \( (\hat{q} - \hat{c})/|\hat{q} - \hat{c}| \)
Collision Response
Collision Response

• Do something to take the bodies from a “colliding state” to a “non-colliding state”

• What properties should a good response algorithm have?
  – Remove interpenetrations
  – Conserve linear and angular momentum
  – Have the correct relative velocities based on the material properties of the colliding bodies
  – Should look plausible!
Collision Response (Notation Key)

- $c_R$ is the coefficient of restitution
  - 0 is completely inelastic; objects stick together
  - 1 is completely elastic; objects bounce without losing any kinetic energy
  - Between 0 and 1 means some energy is lost due to deformation, damage, sound, heat, etc.
- $m_a$ is the mass of the first object
- $m_b$ is the mass of the second object
- $u_a$ is the velocity of the first object before impact
- $u_b$ is the velocity of the second object before impact
- $v_a$ is the velocity of the first object after impact
- $v_b$ is the velocity of the second object after impact
Collision Response (Formulas)

\[ c_R = -\frac{v_b - v_a}{u_b - u_a} \] (definition)

\[ m_a u_a + m_b u_b = m_a v_a + m_b v_b \] (momentum conservation)

• Two equations in two unknowns, solve...
  
  • \( v_a = \frac{(m_a u_a + m_b u_b - m_b c_R (u_a - u_b))}{(m_a + m_b)} \)
  
  • \( v_b = \frac{(m_a u_a + m_b u_b - m_a c_R (u_b - u_a))}{(m_a + m_b)} \)

• We can also look at this in terms of an impulse. The impulse required to change the velocity of object a is
  
  \[ j = m_a (v_a - u_a) \]

• An equal and opposite impulse is applied to object b
If one object is infinitely heavy...

- Useful for kinematic objects (stationary or moving)
- Make $m_b$ infinite
- $v_b = u_b$ (doesn’t change)

$$v_a = \frac{(m_b u_b - m_b c_R (u_a - u_b))}{m_b} = u_b - c_R (u_a - u_b)$$

- If $u_b = 0$ (stationary object, e.g. ground plane), then this further simplifies to $v_a = -c_R u_a$
Higher Spatial Dimensions

• The prior equations describe collision in 1D only
• In 3D, they describe the collision in the normal direction, i.e. on the components of velocity (dot product-ed) into the normal direction
• The tangential components of the velocity do not change, unless there is collisional friction
• Since most surfaces can be locally approximated as being planar, let’s consider point plane collisions....
Point-Plane Collision Response

- Collision only affects the normal component of velocity
- As such, split the velocity into a normal and tangent component:

\[
\vec{V}_N = (\vec{V} \cdot \vec{N})\vec{N} \\
\vec{V}_T = \vec{V} - \vec{V}_N
\]
More Collision Detection

• Need to detect that it’s colliding with the wall, and not separating

• Make sure it is heading into the wall with: \( \vec{V} \cdot \vec{N} < 0 \)
Collision Response

• Adjust the normal velocity of the particle to account for the collision
• Leave the tangential velocity unchanged
• Probably also want to adjust the position of the particle to move it to the surface of the object (if it is inside)

\[
\vec{V}' = \vec{V}_T - c_R \vec{V}_N
\]
Friction

- Let $j_n$ be the collision impulse in the normal direction.

- The new tangential velocity is $\vec{V}'_T = \vec{V}_T - \frac{\mu |j_n| \vec{V}_T}{m|\vec{V}_T|}$, where $\mu$ is the coefficient of kinetic friction:

$$
\vec{V}' = \max \left( 0, \left( 1 - \frac{\mu |j_n|}{m|\vec{V}_T|} \right) \right) \vec{V}_T - c_R \vec{V}_N
$$

- The clamping ensures that friction slows a particle down without changing its direction.

- Static friction can be modeled by first applying this formula with the (typically larger) static friction coefficient:
  - If the max clamps to 0, static friction stopped the object from moving.
  - If not, then recompute the formula with the smaller kinetic friction coefficient.
Question #2

LONG FORM:
• Briefly discuss implementing collisions
• Answer short form question below

SHORT FORM:
• Can you think of a good use for collisions in your video game? Briefly explain
ODEs
Ordinary Differential Equations (ODEs)

- An ODE is an equation containing a function of one independent variable $t$ and its derivatives:
  $$ f(t, y, y', y'', ...) = 0 $$

- First order ODE’s have at most one derivative:
  $$ f(t, y, y') = 0 $$

- If we can isolate the derivative term, we call it an explicit ODE (otherwise its implicit):
  $$ y' = f(t, y) $$
Well-Posed vs. Ill-Posed ODEs

• Model problem
• linear ODE $y' = \lambda y$
  – solution is $y = y_0 e^{\lambda(t-t_0)}$
  – 3 kinds of solutions
    • $\lambda > 0$ ill-posed (a)
    • $\lambda = 0$ mildly ill-posed (b)
    • $\lambda < 0$ well-posed (c)

• Ill-posed problems can not (should not) be solved (with any reasonable assurances) on the computer
Well-Posed vs. Ill-Posed ODEs

• Scalar ODE \( y' = f(t, y) \)
  – Derivatives \( \frac{df}{dy} = \lambda \) must be negative (or \( \leq 0 \) for mild well-posedness) for all values of \( t \) and \( y \) we are concerned with

• Systems of ODEs \( \vec{y}' = f(t, \vec{y}) \)
  – All eigenvalues of the Jacobian matrix \( J = \frac{df}{d\vec{y}} \) must be negative (or \( \leq 0 \)) for all \( t \) and \( \vec{y} \) we are concerned with

• Poor choices of the forces in \( F=ma \) can lead to ill-posed problems!
Numerical Methods for ODEs
Numerical Approximation of Derivatives

\[ y' \approx \frac{(y^{n+1} - y^n)}{(t^{n+1} - t^n)} \]

or

\[ y' \approx \frac{(y^{n+1} - y^n)}{\Delta t} \]

where \( \Delta t = t^{n+1} - t^n \)
Time discretization

\[
\frac{y^{n+1} - y^n}{\Delta t} = f(t^n, y^n)
\]

or...

\[
y^{n+1} = y^n + \Delta t \cdot f(t^n, y^n)
\]

• This method is called **Forward Euler**

• Start at some initial time \( t^0 \) with initial value \( y^0 \)

• Recursively compute the values for the next time step using the values from the current time step

• \( \Delta t \) can be either fixed or adaptively varied for better accuracy and stability
Example

Forward Euler on \( y' = -y \) for \( y^0 = 1, t^0 = 0 \)
\( \Delta t = .5 \) is stable

\( \Delta t = 3 \) is unstable
Forward Euler: Stability

- Consider model equation $y' = \lambda y$ with $\lambda < 0$
  - Recall the analytic solution is exponential decay: $y(t) = y_0 e^{\lambda(t-t_0)}$

- Forward Euler’s method applied to the model equation is $y^{n+1} = y^n + \Delta t \lambda y^n = (1 + \Delta t \lambda)y^n$

- So $y^n = (1 + \Delta t \lambda)^n y^0$, and the solution decays when $|1 + \Delta t \lambda| < 1$

- Thus, $-2 < \Delta t \lambda < 0$ is needed for stability
  - We have $\Delta t \lambda < 0$ trivially, since $\lambda < 0$

- Time step restriction is $\Delta t < \frac{2}{|\lambda|}$
Forward Euler: Accuracy

- $O(\Delta t^2)$ error in each time step (shown via Taylor series)
- $O\left(\frac{1}{\Delta t}\right)$ time steps to get to an $O(1)$ final time
- $O(\Delta t^2) \times O\left(\frac{1}{\Delta t}\right) = O(\Delta t)$ total error
- 1st order accurate
More Numerical Methods for ODEs
Runge-Kutta Schemes

• Runge-Kutta (R.K.) builds on Forward Euler (F.E.)
• Achieves better accuracy by predicting solutions using F.E.
  – and then uses averaging to get new solutions
• Different prediction and averaging schemes give rise to different R.K. schemes
• 1st order (accurate) R.K. is same as F.E.
2nd Order (Accurate) Runge Kutta

• Take two successive F.E. steps:
  \[ \frac{y^{n+1} - y^n}{\Delta t} = f(t^n, y^n) \quad \text{and} \quad \frac{y^{n+2} - y^{n+1}}{\Delta t} = f(t^{n+1}, y^{n+1}) \]

• Average the initial and final states:
  \[ y^{n+1} = \frac{1}{2} y^n + \frac{1}{2} y^{n+2} \]

• If the solution is well behaved for each F.E. step, then since linear interpolation is well behaved, the result is well behaved
3rd Order (Accurate) Runge Kutta

• Take two successive F.E. steps:
  \[
  \frac{y^{n+1} - y^n}{\Delta t} = f(t^n, y^n) \quad \text{and} \quad \frac{y^{n+2} - y^{n+1}}{\Delta t} = f(t^{n+1}, y^{n+1})
  \]

• Average the initial and final states:
  \[
  y^{n+1/2} = \frac{3}{4} y^n + \frac{1}{4} y^{n+2}
  \]

• Take another F.E. step:
  \[
  \frac{y^{n+3/2} - y^{n+1/2}}{\Delta t} = f(t^{n+1/2}, y^{n+1/2})
  \]

• Then average again: 
  \[
  y^{n+1} = \frac{1}{3} y^n + \frac{2}{3} y^{n+3/2}
  \]

• 3\textsuperscript{rd} order R.K is not only more accurate but has some better stability properties
Even More Numerical Methods for ODEs
Backward Euler

\[ y^{n+1} = y^n + \Delta t \, f(t^{n+1}, y^{n+1}) \]

• Equation is implicit in \( y^{n+1} \), so generally need to solve a nonlinear equation to find \( y^{n+1} \)
• Newton iteration…. linearize, solve, linearize, solve, etc.
• Some applications (that allow for larger errors) only use one linearize and solve cycle
• Sometimes \( f \) is already linear in \( y \)
• Accuracy – 1st order (same as forward Euler)
Backward Euler: Stability

• Consider model equation \( y' = \lambda y \) with \( \lambda < 0 \)

• Backward Euler applied to the model equation is
\[
y^{n+1} = y^n + \Delta t \lambda y^{n+1} = (1 - \Delta t \lambda)^{-1} y^n
\]

• So \( y^n = (1 - \Delta t \lambda)^{-n} y^0 \) and the solution decays when \( |1 - \Delta t \lambda| > 1 \)
  – Always true!

• Unconditionally stable - works for all \( \Delta t \)

• No time step restriction…
Backward Euler vs. Forward Euler

- Backward Euler (B.E.) is unconditionally stable
  - i.e. one can take very large time steps, whereas Forward Euler (F.E.) requires smaller time steps
- B.E. might excessively damp out the solution, whereas F.E. might blow up (i.e., NaNs)
- Each B.E. time step may be much harder to solve than a F.E. time step
  - B.E. is more theoretically challenging and uses more CPU time
- Not always clear which is better...
Trapezoidal Rule

\[ y^{n+1} = y^n + \Delta t \frac{f(t^n, y^n) + f(t^{n+1}, y^{n+1})}{2} \]

- 2\text{nd} order accurate
- Unconditionally stable
- Need to solve for \( y^{n+1} \) just like Backward Euler
- One can take very large time steps since it is stable
- Sometimes bad oscillatory behavior if \( \Delta t \) is too big
The Best Numerical Methods....
Back to our problem...

\[
\dot{x} = v \\
\dot{v} = F/m
\]

- We can solve for velocity at one accuracy level lower than for positions (a multivalue method)
- Treating it as a standard system is overkill
- E.g., standard **constant acceleration** equations
  - \( \ddot{x}^{n+1} = \ddot{x}^n + \Delta t \dot{v}^n + \frac{\Delta t^2}{2} \ddot{a}^n \) **piecewise quadratic** position
  - \( \ddot{v}^{n+1} = \ddot{v}^n + \Delta t \ddot{a}^n \) **piecewise linear** velocity
  - \( \ddot{a}^{n+1} = \ddot{a}^n \) **piecewise constant** acceleration (constant from time \( n \) to just before time \( n+1 \))
Newmark Methods

\[ \ddot{x}^{n+1} = \ddot{x}^n + \Delta t \dot{v}^n + \frac{\Delta t^2}{2} [(1 - 2\beta)\ddot{a}^n + 2\beta \ddot{a}^{n+1}] \]

\[ \dot{v}^{n+1} = \dot{v}^n + \Delta t [(1 - \gamma)\ddot{a}^n + \gamma \ddot{a}^{n+1}] \]

- Most popular multi-value method in computational mechanics
- Actually a lot of methods in disguise
- Different choice of \( \beta \) and \( \gamma \) makes a specific method
- \( \beta \) and \( \gamma \) both identically 0 gives the standard constant acceleration case
Newmark Methods

• Second order accurate if and only if $\gamma = 1/2$
• Trapezoidal Rule when $\beta = 1/4$

\[
\dot{x}^{n+1} = \dot{x}^n + \Delta t \ddot{v}^n + \frac{\Delta t^2}{2} \frac{(a^n + \ddot{a}^{n+1})}{2}
\]

\[
\ddot{v}^{n+1} = \ddot{v}^n + \Delta t \frac{(a^n + \ddot{a}^{n+1})}{2}
\]

– Substitute the acceleration terms from the second equation into the first, to see that the first equation is equivalent to

\[
\dot{x}^{n+1} = \dot{x}^n + \Delta t \frac{(\ddot{v}^n + \ddot{v}^{n+1})}{2}
\]
A Newmark Method...

1. \[ \ddot{v}^{n+1/2} = \ddot{v}^n + \frac{\Delta t}{2} \ddot{a}(t^n, \ddot{x}^n, \ddot{v}^{n+1/2}) \]

2. Modify \( \ddot{v}^{n+1/2} \) in some cases, e.g. collisions

3. \[ \ddot{x}^{n+1} = \ddot{x}^n + \Delta t \ddot{v}^{n+1/2} \]

4. \[ \ddot{v}^{n+1} = \ddot{v}^{n+1/2} + \frac{\Delta t}{2} \ddot{a}(t^{n+1}, \ddot{x}^{n+1}, \ddot{v}^{n+1}) \]

5. Modify \( \ddot{v}^{n+1} \) in some cases, e.g. collisions
Implicit Solve

• Steps 1 and 4 are implicit in $\vec{v}^{n+1/2}$ and $\vec{v}^{n+1}$ respectively
• Typically the equations are linear in $v$, so we only need to solve a single matrix system
• The matrix is generally symmetric positive definite (SPD) and we can use fast solvers such as conjugate gradients (CG) for solving the system

• Note that in the first step we are using $\vec{x}^{n}$ instead of $\vec{x}^{n+1/2}$
  – The equations are typically highly nonlinear in $x$
Question #3

LONG FORM:
• Tell me everything about.... Nvm, have a nice day!

SHORT FORM:
• 😊