Contents

1 Equations and their Solutions

1.1 Newton's First Law

When viewed in an inertial reference frame, an object is either at rest or moves at a constant velocity, unless acted upon by an external force.

- Objects in a scene that are to be kept in a constant position (e.g. ground) should be immune to physics calculations.
- Movable objects should have a "rest" state in which no physics calculations are performed for that object until something will cause it to move.

  This both makes the simulation less computationally intensive and reduced the chance of jitter and error caused by two nearly equal floating point numbers.

1.2 Newton's Second Law

The vector sum of the forces acting on an object is equal to the total mass of that object multiplied by the acceleration of the object.

$$F = ma$$

1.3 Newton's Third Law

When one body exerts a force on a second body, the second body simultaneously exerts a force equal in magnitude and opposite in direction to that of the first body.

1.4 Law of Conservation of Momentum

In a closed system, total momentum is consistent.

$$p = mv$$

Ignoring all other forces if two objects of mass m_n travelling towards each other with momentum u_n, the total momentum of the system is:

$$p_{initial} = m_1u_1 + m_2u_2$$

After they collide the two objects will have two new velocities while keeping the same masses:

$$p_{final} = m_1v_1 + m_2v_2$$
$$p_{final} = p_{initial}$$

1.5 SUVAT equations

s Displacement

u Initial velocity

v Final velocity

a Acceleration

t Time

$$\begin{aligned}
v &= u + at \\
s &= ut + \frac{1}{2}at^2 \\
s &= \frac{1}{2}(u+v)\,t \\
v^2 &= u^2 + 2as \\
s &= vt - \frac{1}{2}at^2
\end{aligned}$$

Constant acceleration is assumed, this is not an issue in simulation as updates are calculated in time steps sufficiently small that constant acceleration can be assumed.

1.6 Numerical Integration

Using time step: $\Delta t = t_{x+1} - t_x$

1.6.1 Explicit Euler

- Simple
- Computationally cheap
- Unstable at high Δt

$$\begin{aligned}
v_{x+1} &= v_x + a_x \Delta t \\
s_{x+1} &= s_x + v_x \Delta t
\end{aligned}$$

1.6.2 Implicit Euler

- Similar to explicit Euler but uses rate of change in next state
- Stable
- Computationally expensive

$$\begin{aligned}
v_{x+1} &= v_x + a_{x+1} \Delta t \\
s_{x+1} &= s_x + v_{x+1} \Delta t
\end{aligned}$$

1.6.3 Semi-implicit Euler

- A combination of explicit and implicit Euler
- Stable
- Avoid expensive operations

$$\begin{aligned}
v_{x+1} &= v_x + a_x \Delta t \\
s_{x+1} &= s_x + v_{x+1} \Delta t
\end{aligned}$$

1.6.4 Verlet

- Similar computational cost to Semi-implicit Euler
- Entities must store their historical positions (in a FIFO buffer)
- Reversible

Substitute $v = u + at$ into Euler equations:

$$\begin{aligned} s_{x+1} &= s_x + v_{x+1}\Delta t \\ &= s_x + (v_x + a_x\Delta t)\Delta t \\ &= s_x + v_x\Delta t + a_x\Delta t^2 \end{aligned}$$

This can be solved for v_x using the past two displacements:

$$v_x = \frac{s_x - s_{x-1}}{\Delta t}$$

The substituted back into:

$$\begin{aligned} s_{x+1} &= s_x + v_x\Delta t + a_x\Delta t^2 \\ &= s_x + (s_x - s_{x-1}) + a_x\Delta t^2 \end{aligned}$$

1.7 Drag

In simple simulations the forces that act against an object to eventually bring it to a stop (e.g. friction, air resistance, etc.) can be represented by a damping factor $0 < d < 1$.

This factor is used to attenuate the velocity of a moving entity in each physics frame.

This factor can be varied as a cheap way of simulating different environments.

1.8 Rest

When the velocity of an entity is close to zero, set the velocity to zero.

This is done to avoid excessive computation for entities at rest.

1.9 Gravity

Gravity is simple to handle as a constant that is added to acceleration in the y axis.

This constant can be varied as a cheap way of simulating different environments.

2 Collision Detection and Response

2.1 Detection

Collision detection is a naturally expensive operation for scenes with large numbers of entities.

Without optimisation it becomes an N^2 problem (where N is the number of entities).

An approach to reducing the number of comparison operations is to separate collision detection into two phases: broadphase and narrowphase.

2.1.1 Broadphase

Broadphase collision detection reduces the number of collision checks required by culling checks for entities that cannot have collided.

Bounding Box Test

Tests for collisions using a bounding box defined by the extremes of an entity in each axis.

In the case where there is no other entity "close" to the one being tested, this replaces the complex collision check with a much simpler one.

Works well in scenes with low numbers of entities.

World Partitioning

Partition the world into several subsections, converting the large N^2 problem into several smaller n^2 problems.

Provision has to be made for an entity to exist in multiple partitions if it is on a boundary.

Entities are sorted into partitions based on their size and position.

If performing a simple geographical division then it is possible for the distribution of entities to mean this is no faster (or sometimes slower) than performing a bounding box check (e.g. if all entities are in the same partition).

Binary Search Partitioning

Binary search partitioning (BSP) solves the problem of uneven entity distribution in world partitioning by recursively subdividing partitions based on the number of entities in a partition.

An Octree is a common approach to this in which the world is split into progressively smaller cubes.

The world is split into several partitions, if the number of entities in a partition is above a set threshold then the partition is split into several sub partitions. This process repeats until the number of entities in each sub partition is below the set threshold.

The threshold must be chosen well: too high and the benefits of BSP are not visible, too low and the process becomes inefficient due to the number of sorting operations when partitions are divided.

Sort and Sweep

Sort and sweep can be used in conjunction with any other method of culling checks where collision is impossible.

In this process bounding boxes of all entities are projected onto a single axis and any pairs of entities that do not overlap are culled from more expensive comparisons.

This can be performed during the test for partition occupancy in world partitioning or BSP.

2.1.2 Narrowphase

Accurately check the remaining pairs of entities.

Narrowphase checks should also provide the necessary data to calculate the collision response:

Contact point: P
: Point at which the interface was detected.

Contact normal: N
: A unit vector describing the direction in which the entities must be moved to resolve the interface.

Penetration depth: p
: The minimum distance the entities must be moved to resolve the interface, i.e. the distance required for them to be touching, not intersecting.

Due to the mature of the time stepped physics model, entities will intersect (interface) before they are detected. i.e. Collisions are not detected, intersections are.

The time step must be small enough such that the contact point is on the correct side of both entities, otherwise the interface will be resolved in the opposite direction.

Sphere-Sphere detection

Two spheres are said to have collided if the following equation is true:

$$d < r_1 + r_2$$

Where d is the distance between the centres of the spheres and r_1 and r_2 are the radii of each sphere.

The collision response data can be extracted as follows:

$$\begin{aligned} p &= r_1 + r_2 - d \\ N &= |S_1 - S_2| \\ P &= S_1 - N \times (r_1 - p) \end{aligned}$$

Sphere-Plane detection

Using the plane equation:

$$\begin{aligned} ax + by + cz + d &= 0 \\ -ax_0 - by_0 - cx_0 &\equiv d \end{aligned}$$

Where $N = a, b, c$ is the normal to the plane, (x_0, y_0, z_0) is a point on the plane and any given point on the plane is given by (x, y, z).

A sphere and plane are said to have collided if the following equation is true:

$$N \cdot S + d < r$$

Where N is the normal to the plane, S is the position of the sphere, d is the distance from the centre of the sphere to the nearest point on the plane and r is the radius of the sphere.

The distance d can be calculated by:

$$d = -(x \cdot N)$$

Where x is any point on the plane.

The collision response data can be extracted as follows:

$$\begin{aligned} p &= r - (N \cdot S + d) \\ P &= S - N \times (r - p) \end{aligned}$$

Note that in this case an infinite plane is assumed.

2.2 Response

Three methods:

Projection
Moves colliding objects far enough apart for them not to be colliding, operating on position alone.

Impulse
Calculates new velocity for each object based on the law of conservation of momentum.

Penalty
Calculates new acceleration for each object by applying force required to push objects apart.

2.2.1 Impulse method

Impulse J is the force F applied over a time Δt.

$$\begin{aligned} J &= F\Delta t \\ J &= ma\Delta t && F = ma \\ J &= m\frac{\Delta v}{\Delta t}\Delta t = m\Delta v && a = \Delta v/\Delta t \end{aligned}$$

As momentum $p = mv$, impulse J is simply the change in momentum for an object of constant mass.

For two colliding objects A and B, the velocity is given by:

$$\begin{aligned} v_{ab} &= v_a + v_b \\ v_N &= v_{ab} \cdot N \end{aligned}$$

Where v_N is the total velocity in the direction of the collision normal N.

The final velocity along the normal after the collision is proportional to the coefficient of elasticity ϵ. Where $\epsilon = 1$ is an elastic collision (i.e. all momentum is conserved) and $\epsilon = 0$ is an inelastic collision (i.e. all momentum is dissipated).

$$\begin{aligned} v_N^{final} &= -\epsilon\, v_N^{initial} \\ \left(v_a^{final} + v_b^{final}\right) \cdot N &= -\epsilon\, \left(v_a^{initial} + v_b^{initial}\right) \cdot N \end{aligned}$$

Momentum injected into the system must be balanced to maintain the law of conservation of momentum:

$$\begin{aligned} m_a v_a^{final} &= m_a v_a^{initial} + J_N \\ m_b v_b^{final} &= m_b v_b^{initial} - J_N \end{aligned}$$

The impulse J is given by:

$$J = \frac{-\left(1+\epsilon\right) v_{ab} \cdot N}{N \cdot N \left(\frac{1}{m_a} + \frac{1}{m_b}\right)}$$

Then the new velocities for each object are computed by:

$$\begin{aligned} v_a^{final} &= v_a^{initial} + \frac{J}{m_a} N \\ v_b^{final} &= v_b^{initial} - \frac{J}{m_b} N \end{aligned}$$

2.2.2 Penalty method

See Springs first (3).

Assume that:

- A "virtual" spring exists between the colliding objects
- The penetration depth p is the displacement of the spring

Then the relative velocity of the colliding objects is given as:

$$v = N \cdot (v_a - v_b)$$

Therefore the collision can be modelled as a spring using equation:

$$F = -kp - c\,(N \cdot (v_a - v_b))$$

This calculates the resultant force of the collision, the new acceleration values are then calculated by:

$$a_a^{final} = a_a^{initial} + \frac{F}{m_a}$$
$$a_b^{final} = a_b^{initial} + \frac{F}{m_b}$$

The value of k behaves in the same way as that of ϵ in the impulse method.

The value of c determines how much the object reacts to the spring, i.e. a higher value gives a more "bouncy" collision.

3 Springs

A spring has rest length to which it will return when any force compressing or extending it has been removed.

The force required to displace a spring by distance x from its rest length is given by Hooke's law of Elasticity:

$$F = -kx$$

Where k is the spring constant.

In a simulation an additional damping factor is added to ensure that objects affected by the spring eventually come to rest:

$$F = -kx - cv$$

Where c is a damping factor and v is the velocity of an object in the axis of the spring.

4 Soft Bodies

Soft bodies (such as cloth) are modelled as a collection of rigid bodies bound by "virtual" springs.

A piece of cloth at rest will have no tension in all springs (i.e. all springs at their rest length), when it is moved the springs closest to the point at which the cloth is being moved will begin to act upon their neighbouring meshes which populates back to almost all of the surface.

Simulating soft bodies using springs can lead to amplification of movement within the system if damping factors are not properly tuned.

5 Choice and States

Game AI is described as any decision making process within the game and is typically implemented using a finite state machine (FSM).

An FSM is a collection of states that describe what an entity is supposed to be doing and a set of conditions that will cause a transition to another state.

5.1 Implementation methods

Hard coded switch

States are enumerated and transition conditions are checked based on the current state in a single conditional structure.

Simple and efficient for small FSMs, becomes unmanageable for a larger/more complex FSM.

Hard coded state pattern

All states are child classes of a base class `State`, each state class is responsible for managing its own behaviour and state.

Reduces complexity of large conditional structures.

Interpreted state pattern

Similar to above but states, behaviours and transitions are read from a file and the structure is constructed at game startup.

5.2 Issues

Missed transitions

If the logical conditions for a state transition do not evaluate to true, however the transition should have happened.

E.g. comparison using equality at a spatial boundary.

Complexity

A growing number of states can make an FSM complex, this issue is partially negated by hierarchical FSMs.

State oscillations

Occurs when no hysteresis is implemented in the transition conditions for continuous variables.

As a result states constantly change due to over compensation of the behaviour of the activated state.

5.3 Hierarchical FSMs

Applying a hierarchy to the states of an FSM can simplify the set of transitions and overall structure of the FSM.

Divide groups of states that correspond to a single NPC behaviour into their own parent states.

Transition conditions for each active state in the hierarchy must be checked.

6 Path Finding

6.1 A* algorithm

- Algorithm uses a best first approach to build up an optimal path from a starting node to a target node
- Operates over a graph described by nodes and edges connecting them
- Nodes and edges can be marked as impassible in which case they are not considered by the algorithm when searching for a path
- Edges can have weights adjusted to represent the cost of traversing them (e.g. to reflect difficulty in traversing a certain type of terrain)

The A* algorithm requires several key features:

Node structure
Several additional pieces of information about nodes on the graph need to be stored:

Parent node
The shortest path from this node to the start node.

***g*-value**
The cost of the shortest path from the start node to this node.

***h*-value**
The estimated cost of the path from this node to the end node.

***f*-value**
The estimated cost of the shortest path from the start node to end node that traverses this node.

$$f = g + h$$

Heuristic equation
An equation used to estimate the cost of travelling from a given node to the target node.

This **must** be either an underestimate or the exact cost. Providing a heuristic that is an overestimate will result in a path other than the optimal path being found.

Open list
A list of nodes that the algorithm is aware of, i.e. those that are connected to nodes in the closed list.

The open list is sorted by ascending *f*-value, this allows the node that is assumed to have the cost form start to end node to be selected from the top of the list.

Closed list
A list of nodes that have been considered as part of the shortest path by the algorithm.

Every node will have been on the open list at some point.

The shortest path always be made up of a subset of nodes from the closed list.

The workflow of the algorithm is as follows:

1 Add start node to the open list

2 For each node P on the open list

    1 Move P to the closed list

```
    2 If P is the end node the shortest path has been found, then break from the loop
    3 For each node Q connected to P
        1 Skip Q if it is not traversable
        2 Calculate $g$ and $f$ scores for Q
        3 If Q is on either the open list or closed list and the calculated $g$-value is less than its
          current $g$-value then update the $g$ and $f$ scores and set its parent to P
        4 If Q is not on any list then add it to the open list and set its parent to P
4 If a path has been found the build the path by traversing the parents of the last node added to the
  closed list adding each node to a list, then reversing this list to obtain the path from start to end
```

6.2 Goal Oriented Action Planning

A speciality of path finding that combines path finding with a finite state machine in order to find the most optimal series of state changes that lead to a final state given an initial state.

- Each possible state is a node on a graph and state changes are edges
- Each state change has a cost associated with it
- The way states are encoded allows a similarity comparison which can be used to calculate the h-score

It then follows that the optimal series of state changes can be determined by performing A* over the graph of states and state transitions.

6.3 Optimisations and Issues

- When using Euclidean distance as a heuristic in A* it is common to greatly underestimate the lowest distance.

  Underestimation of the heuristic can result in greater computational cost of the entire path finding algorithm due to the increased number of paths that are considered before the optimal path is found.

  As a result more complex heuristics are typically used which take into account specifics of the game and its world.
- Nodes or edges that are not traversable can be marked with a boolean flag to prevent them from being considered by the A* algorithm (as opposed to assigning an infinite weight).
- Groups of objects that move in an identical path together (e.g. squads in an RTS) only one instance of the path finding must be performed for the entire group and the positions of individuals computed using positions relative to the centre, leader, etc.
- A* is an expensive algorithm and in the worst case will perform a full exploration of the search space.
- A* only works in practice for single agent path finding. Multi agent path finding is PSPACE difficult.
- Paths should only be recomputed when the best path is likely to change.

  Usually this would be once when the entity in motion first starts to move and an additional time for each event that changes parts of the graph that form parts of the path the entity is taking.

7 Fluid and Wave simulation

The correct way to simulate fluids is typically based around the Navier-Stokes equation. Such methods are too computationally expensive to be used in real time simulations, as such graphical ricks are typically used to create the effect of bodies of water in games.

7.1 Dimensional reduction heightmaps

If only the surface of the fluid body is to be considered then it can be represented using a simple heightmap where a y coordinate (the height) is mapped to a series of x and z coordinates in a fixed grid.

7.2 Column approach

If the fluid body is represented as a collection of cuboid of fixed x and z dimensions with the y dimension equivalent to the height of one point on the heightmap then the fluid volume can be defined by the fixed area in the XY plane multiplied by the height.

The fluid velocity is then the change in height with respect to time. A reduction to a single dimension which greatly simplified the implementation of the Navier-Stokes equation.

One issue to be addressed is the range to which the height of a fluid body should be clamped.

Without proper boundary clamping energy can build up in a system which causes undesired behaviour, however incorrectly set boundaries will detract from the realism of the simulation in the case where a boundary is reached (e.g. flat plateaus being generated where several fluid bodies are above the height threshold).

7.3 Functional approach

A simpler method is to model the movement of waves on the fluid bodies surface using a series of periodic functions, typically sinusoidal functions, overlapped on top of each other to determine the height of a particular point in the heightmap.

This can be applied continuously to give the effect of rippling waves, etc. or applied using a finite state machine to simulate the water being disturbed by a player.

This technique can be used to generate a realistic simulation with very little computational cost.

7.4 Optimisations and Issues

- Any implementation using the Navier-Stokes approach will be computationally intensive.
- If representing the surface of a fluid body as a soft body then additional engineering challenges need to be addressed in the handling of multiple sustained collisions when objects either sink, float on or bounce off the surface of the fluid.
- If using a heightmap to represent the surface of the fluid then tessellation can be used in the graphical pipeline to increase the quality of the simulated water surface.
- If objects are to be submerged in a fluid then the issue of submersion can be addressed by attaching a spring between the entity being submerged and the surface of the fluid.

  Once the entity sinks far enough into the fluid body it breaks the spring and frees the column of water.

  This approach works well when representing the fluid surface as a soft body or when functional rippling is triggered by an entity entering the fluid body as it provides realistic response from the fluid surface.

- Buoyancy can be addressed in several ways, the simplest is to apply a fixed acceleration in the y axis to counteract the fixed acceleration due to gravity.

  Depending on the value relative to gravity this can either slow (for entities denser than the fluid) or reverse (for entities less dense than the fluid) the rate of submersion.