# Contents


**1 Physics 3**

1.1 Newtonian Dynamics . . . 3

1.1.1 Newton's First law . . . 3

1.1.2 Newton's Second law . . . 3

1.1.3 Newton's Third law . . . 3

1.1.4 Conservation of Momentum . . . 3

1.1.5 Torque . . . 3

1.1.6 Inertia . . . 3

1.2 Physics Engine . . . 4

1.2.1 Orientation . . . 4

1.2.2 Object state . . . 5

1.2.3 Physics Representation . . . 5

1.2.4 Environment Scaling . . . 6

1.3 Numerical integration . . . 6

1.3.1 Explicit Euler . . . 6

1.3.2 Implicit Euler . . . 6

1.3.3 Semi-implicit Euler . . . 6

1.3.4 Verlet . . . 7

1.3.5 Runge-Kutta (Midpoint) method . . . 7

1.3.6 Other methods . . . 7

1.4 Constraints . . . 7

1.4.1 Note on differentiation . . . 8

1.4.2 Constraint in a single dimension . . . 8

1.4.3 Constraint in three dimensions . . . 8

1.4.4 Adding Energy . . . 9

1.4.5 Constraint Drift and Baumgarte correction . . . 9

1.4.6 Calculating $\lambda$ . . . 9

1.5 Collision Detection . . . 10

1.5.1 Broadphase: Bounding volume test . . . 10

1.5.2 Broadphase: World Partitioning . . . 11

1.5.3 Broadphase: Sort and Sweep . . . 11

1.5.4 Narrowphase: Separating Axis Theorem (SAT) . . . 11

1.5.5 Sphere-Sphere test . . . 13

1.5.6 AABB-AABB test . . . 14

1.5.7 Sphere-Plane test . . . 14

1.6 Collision Manifolds . . . 15

1.6.1 Clipping method . . . 15

1.7 Collision Response . . . 17
1.7.1 Impulse method . . . 17
1.7.2 Penalty method . . . 18
1.7.3 Soft Bodies . . . 19
1.7.4 Constraint based collision response . . . 19
1.7.5 Constraints as Friction . . . 20
1.8 Solvers . . . 20
1.8.1 Gauss-Seidel . . . 21
1.8.2 Constraint Drift . . . 22

**2 Artificial Intelligence 23**
2.1 Finite State Machine . . . 23
2.1.1 Hierarchical FSM . . . 24
2.1.2 Behaviours and Types . . . 24
2.1.3 Fuzzy State Machines . . . 24
2.2 Path Planning . . . 24
2.2.1 A* algorithm . . . 25
2.2.2 Computational cost . . . 26
2.2.3 Spline following . . . 26
2.3 Crowd management . . . 26
2.3.1 Fixed Group . . . 26
2.3.2 Embedded Map Data . . . 27
2.3.3 Flocking . . . 27
2.3.4 Ant Colony Optimisation . . . 27
2.4 Decision making . . . 28
2.4.1 Decision trees . . . 28
2.4.2 Goal Oriented Action Planning . . . 28
2.4.3 Machine Learning . . . 28

**3 Networking 29**
3.1 Socket protocols . . . 29
3.2 Topologies . . . 29
3.3 Constraints of Network Gaming . . . 30
3.3.1 Zoning . . . 30
3.3.2 Interest Management . . . 31
3.3.3 Dead Reckoning . . . 31

**4 Massively parallel and Heterogeneous computing 33**
4.1 GPU computation . . . 33

Course material: `https://research.ncl.ac.uk/game/mastersdegree/gametechnologies/`

# 1 Physics

## 1.1 Newtonian Dynamics

### 1.1.1 Newton's First law

*In an inertial reference frame, an object either remains at rest or continues to move at a constant velocity, unless acted upon by a force.*

- In a games physics engine forces that would typically dampen the velocity of an object in motion (e.g. friction, air resistance, etc.) are abstracted to simple damping factors

### 1.1.2 Newton's Second law

*In an inertial reference frame, the vector sum of forces on an object is equal to the mass of the object multiplied by the acceleration of the object.*

$$F = ma$$

### 1.1.3 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 on the first body.*

### 1.1.4 Conservation of Momentum

*In a closed system, the total momentum is constant.*

$$p = mv$$

In a collision:

$$p^i = p^f$$
$$p_1^i + p_2^i = p_1^f + p_2^f$$
$$m_1^i v_1^i + m_2^i v_2^i = m_1^f v_1^f + m_2^f v_2^f$$

### 1.1.5 Torque

- Result of a force applied to an object a given distance from a pivot point
- Torque produced by force $F$ at distance from pivot $d$: $\tau = dF$

### 1.1.6 Inertia

- Moment of inertia is the resistance of a rigid body to change in rotational motion
- Moment of inertia defines how mass is distributed about each axis
- Inertial tensor contains influence of torque in a given axis on the acceleration in a given axis.

$$\begin{bmatrix} \tau_x \\ \tau_y \\ \tau_z \end{bmatrix} = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{bmatrix} \begin{bmatrix} \alpha_x \\ \alpha_y \\ \alpha_z \end{bmatrix}$$

- In the inertia tensor the diagonal elements must not be zero and the non-diagonal elements must be symmetrical
- For symmetrical objects the inertia tensor only has non-zero elements in the diagonal, therefore for any given axis only torque applied in that axis can influence angular acceleration in that axis
- Solid sphere of radius $r$ and mass $m$:

$$i = \frac{2mr^2}{5}$$

$$I = \begin{bmatrix} i & 0 & 0 \\ 0 & i & 0 \\ 0 & 0 & i \end{bmatrix}$$

- Solid cuboid of dimensions $(h, w, l)$ and mass $m$:

$$I_{xx} = \frac{1}{12}m\left(h^2 + w^2\right)$$
$$I_{yy} = \frac{1}{12}m\left(l^2 + w^2\right)$$
$$I_{zz} = \frac{1}{12}m\left(h^2 + l^2\right)$$

$$I = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix}$$

- Asymmetrical objects are more computationally expensive than symmetrical (due to non-diagonal inertia tensor) and is not always worthwhile in its benefit to the simulation

## 1.2 Physics Engine

- Role of physics engine:
  1 Move objects
  2 Detect collisions
  3 Resolve collisions

### 1.2.1 Orientation

- Stored as quaternion
- $\Theta = \left(xsin\left(\frac{\theta}{2}\right), ysin\left(\frac{\theta}{2}\right), zsin\left(\frac{\theta}{2}\right), cos\left(\frac{\theta}{2}\right)\right)$
  where $(x, y, z)$ is the axis of rotation and $\theta$ is the angle of rotation

### 1.2.2 Object state

- Position, $s = \int v \, dt$
- Linear Velocity, $v = \int a \, dt$
- Linear Acceleration, $a = \frac{F}{m}$
- Force, $F$
- Mass, $m$
- Orientation, $\theta = \int \omega \, dt$
  (Represented as quaternion)
- Angular Velocity, $\omega = \int \alpha \, dt$
- Angular Acceleration, $\alpha = \frac{\tau}{I}$
- Torque, $\tau$
- Inertia, $I$

### 1.2.3 Physics Representation

**Particle**

- Single point in space
- Has linear motion but no angular motion
- Has no volume
- Can be used for particle systems (e.g. smoke, fog, fire, etc.) or for when speed of calculations is more important than physical accuracy (e.g. distant objects, off camera objects, etc.)

**Rigid Body**

- Objects are defined as a set of shapes in space
  Complex objects built up of several individual rigid bodies
- Has linear and angular motion
- Has volume and mass
- Shape is constant/non-deformable
- Used when accurate collision detection and response is required

**Soft Body**

- Represents objects that can change shape/deform
- Otherwise similar to rigid body
- More expensive than rigid bodies (both computationally and memory wise)

| | Velocity | Angular | Volume | Deformation | Collision |
|---|---|---|---|---|---|
| Particle | ✓ | ✗ | ✗ | ✗ | (✗) |
| Rigid Body | ✓ | ✓ | ✓ | ✗ | (✓) |
| Soft Body | ✓ | ✓ | ✓ | ✓ | (✓) |

Table 1: Comparison of features of different physical representations

- Physics shapes are often different from graphical shapes
- Physical shape of an object is often an approximation made from several small shapes that provides collision detection and response that is sufficiently believable for the player

#### 1.2.4 Environment Scaling

- Environment scaling is important in ensuring physical accuracy in the system
- Time step must be small enough to ensure that interfaces between objects are detected in a timely manner
- Too slow a time step, too high a speed or too small a object can all lead to interfaces being detected incorrectly (either not at all or with incorrect normal and penetration depth)
- Use of a common scaling system (i.e. not relative to the object (or mesh))

## 1.3 Numerical integration

- Obtaining new value based on its rate of change
- Multiple uses other than physics simulation

#### 1.3.1 Explicit Euler

$$
\begin{aligned}
v_{n+1} &= v_n + a_n \Delta t \\
s_{n+1} &= s_n + v_n \Delta t
\end{aligned}
$$

- Evaluates entirely within current time step
- Tends to be unstable unless a very short time step is used

#### 1.3.2 Implicit Euler

$$
\begin{aligned}
v_{n+1} &= v_n + a_{n+1} \Delta t \\
s_{n+1} &= s_n + v_{n+1} \Delta t
\end{aligned}
$$

- Addresses approximation issues of Explicit Euler by using future state of system
- Knowing future value of acceleration is expensive (to the extent of not typically being useful)

#### 1.3.3 Semi-implicit Euler

$$
\begin{aligned}
v_{n+1} &= v_n + a_n \Delta t \\
s_{n+1} &= s_n + v_{n+1} \Delta t
\end{aligned}
$$

- Best of both explicit and implicit Euler
- Uses current acceleration but next velocity
- Still fast to execute but more stable than explicit Euler

#### 1.3.4 Verlet

$$v_n = \frac{s_n - s_{n-1}}{\Delta t}$$
$$s_{n+1} = s_n + v_n \Delta t + a_n \Delta t^2$$

- Does not calculate velocity
- Second derivative approach
- Similar computational cost to semi-implicit Euler
- Can reverse simulation
- More suited to the penalty method of collision response
- Requires the initial state to be provided (requires last two positions)

#### 1.3.5 Runge-Kutta (Midpoint) method

- Euler methods overlook the change in velocity that occurs throughout a time step and only consider the values at the start and end of a step
- Runge-Kutta methods take changes in velocity during the time step into account via several iterations of Euler integrations
- In RK2 the velocity is predicted at the point half way through the time step

$$s_{n+1} = s_n + v_{n+0.5} \Delta t$$

#### 1.3.6 Other methods

- Several more complex iterative integration methods exist

  (e.g. higher orders of Runge-Kutta)
- Additional accuracy of more complex integration will likely not be noticed in gaming physics simulations
- Better suited for mathematics, engineering and scientific simulations where there is a weaker real time constraint or lower agent simulations

### 1.4 Constraints

- Define what must not happen in the simulation
- General equation:

$$\dot{C} = JV$$

### 1.4.1 Note on differentiation

- A partial derivative (denoted by $\partial$) is one in which only a single value of a multivariate function is changed

  i.e. for function $f(x, y, z)$ the partial derivative $\frac{\partial f}{\partial x}$ only changes variable $x$, variables $y$ and $z$ remain constant

- The vector differential $\nabla$ is used to represent a series of partial differentials for every variable in a system

  e.g. $\nabla = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}\right)$

### 1.4.2 Constraint in a single dimension

- Consider a constraint restricted in a single axis:

$$C(x, y) = \frac{1}{2}\left((x - y)^2 - L^2\right)$$

- Constrains two points $x$ and $y$ along a single axis to be distance $L$ apart
- Differentiate $C$ with respect to time $t$ using multivariate chain rule:

$$\frac{dC}{dt} = \frac{\partial C}{\partial x}\frac{dx}{dt} + \frac{\partial C}{\partial y}\frac{dy}{dt}$$

- Rearrange and substitute dot notation differentials:

$$\dot{C} = (x - y)\dot{x} + (y - x)\dot{y}$$

- This gives:

$$J = (x - y \quad y - x)$$
$$V = (\dot{x} \quad \dot{y})$$

- Force $F$ required to satisfy constraint:

$$F = J^T\lambda$$

  where $J^T$ is the transposed Jacobian

### 1.4.3 Constraint in three dimensions

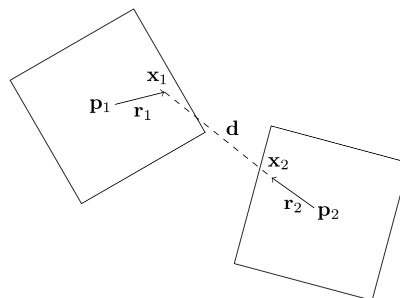



Figure 1: 3D constraint example

- Consider a constraint that keeps points $x_1$ and $x_2$ a fixed distance $L$ apart:

$$C(x_1, q_1, x_2, q_2) = \frac{1}{2}\left((x_2 - x_1)^2 - L^2\right)$$

  where $q_1$ and $q_2$ are orientations of the two cubes.
- Differentiate $x_1$ and $x_2$ as vector from position of cube with respect to time:

$$\frac{dx_1}{dt} = v_1 + \omega_1 \times r_1$$
$$\frac{dx_2}{dt} = v_2 + \omega_2 \times r_2$$

- Obtain:

$$\dot{C} = -d \cdot v_1 - (r_1 \times d) \cdot \omega_1 + d \cdot v_2 + (r_2 \times d) \cdot \omega_2$$

- Gives Jacobian $J$:

$$J = \begin{pmatrix} -d^T & -(r_1 \times d)^T & d^T & (r_2 \times d)^T \end{pmatrix}$$

### 1.4.4 Adding Energy

- Energy is constant when $\dot{C} = 0$
- Can set $\dot{C}$ to a vector function to add or remove energy from the system
- Known as bias vector $\zeta$
  Can relate to position, orientation and time

### 1.4.5 Constraint Drift and Baumgarte correction

- Can introduce a correction factor $\beta$ to deal with accumulated error due to iterative updates:

$$\dot{C} = JV - \beta C$$

- Errors often tied to nature of constraints and time, as such express correction as:

$$\dot{C}(t) + \beta C(t) = 0$$

- $\beta$ is clamped to the range:

$$0 < \beta < \frac{1}{\Delta t}$$

### 1.4.6 Calculating $\lambda$

- Force $F$ required to satisfy constraint:

$$F = J^T \lambda$$

  where $\lambda$ is a constant such that force $F$ balances/solves the constraint
- Need to calculate $\lambda$ for specific objects
- Consider Newton's second law ($f = ma$):

$$F = M \cdot V$$
$$M = \begin{bmatrix} m_1 & 0 & 0 & 0 \\ 0 & I_1 & 0 & 0 \\ 0 & 0 & m_2 & 0 \\ 0 & 0 & 0 & I_2 \end{bmatrix}$$

  Matrix $M$ describes distribution of mass for both objects ($m$ are masses, $I$ are inertia tensors) and $\dot{V}$ is the acceleration equivalent to $V$

- Approximate $\dot{V}$ numerically over a small time step $\Delta t$:

$$\dot{V} \approx \frac{V_2 - V_1}{\Delta t}$$

- Multiply by mass matrix $M$:

$$J^T \lambda = M\dot{V}$$
$$J^T \lambda = \frac{M(v_2 - V_1)}{\Delta t}$$

- Multiply by $JM^{-1}$ ($M$ is easily invertible):

$$JM^{-1}J^T \lambda = \frac{\zeta - JV}{\Delta t}$$

- Rearrange for $\lambda$:

$$\lambda = \frac{\zeta - JV}{JM^{-1}J^T \Delta t}$$

## 1.5 Collision Detection

- Performed in two stages:

  **Broadphase**
  - Determine if it is possible for two objects to have collided
  - Reduce the list of collision pairs by excluding those where a collision is certainly impossible
  - Computationally fast tests as accuracy is not important at this stage

  **Narrowphase**
  - Determine if two objects have collided
  - Accurate tests
  - Often tests for simple shapes can be used, depending on the specifics of the object
    Game object often made up of several simple collision shapes

- Collision response data to be extracted from collision detection:

  **Penetration Depth,** $p$
  Distance the two objects interface/overlap

  **Collision Normal,** $N$
  Normal along which the two objects collided

  **Contact Point,** $P$
  The point at which the two objects are interfacing

### 1.5.1 Broadphase: Bounding volume test

- Simplest (but effective) test
- Test for intersection of bounding volumes
- Typically either sphere or axis aligned bounding box due to efficiency of test

#### 1.5.2 Broadphase: World Partitioning

- Split world into partitions and only test for collision between objects in the same partition
- Objects may reside in multiple partitions (if they are on the border of several partitions)
- Common techniques are:

  **Fixed World Partitioning**

  - World is split into predetermined partitions
  - Efficiency over brute force broadphase dependant on distribution of objects

  **Binary Search Partitioning**

  - Recursively partition the world
  - Continue to split the world until either a maximum depth or number of objects in a division is reached
  - Distributes partitions according to object distribution so maintains efficiency regardless of world state
  - Octree and quadtree are common implementations, dividing each partition into 8 or 4 sub-partitions respectively

#### 1.5.3 Broadphase: Sort and Sweep

- Project bounding volumes onto a single axis and create possible collision pairs from overlapping volumes
- Workflow:

  1 Sort objects based on their position along the sort axis

  2 Traverse the list, when the start of object $O_i$ is found and other objects that are encountered before the end of $O_i$ are potential collision pairs

#### 1.5.4 Narrowphase: Separating Axis Theorem (SAT)

- If two objects are not interfacing then it should be possible to draw a plane between them
- Only works for convex shapes
- If a single can where a plane can be drawn between two objects then the two objects are certainly not interfacing
- Test by projecting the shapes onto a test axis and checking for overlap between them

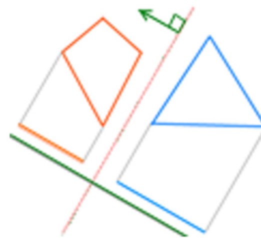


Figure 2: SAT collision detection

**Generating potential separating axes**

- Assuming only polygonal shapes the potential collision axes are defined by the normals of each face of the collision shapes
- Number of potential separating axis is the number of faces of each shape combined
- As soon as a single separating axis is found in which the projections of the objects do not overlap the algorithm can exit (without having to test all axes)
- Parallel axis do not need to be checked multiple times

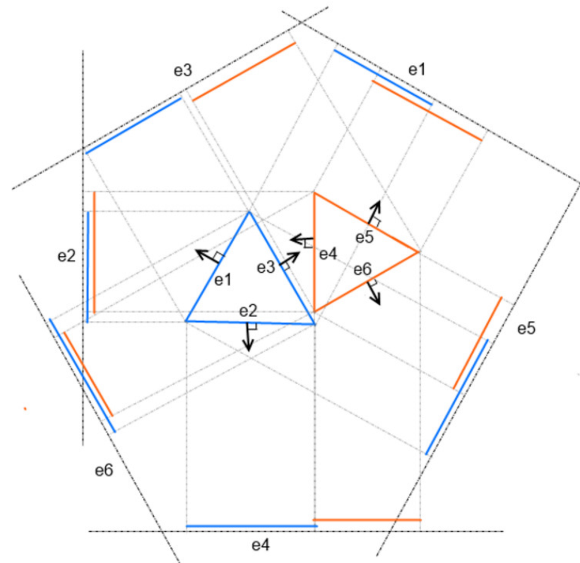
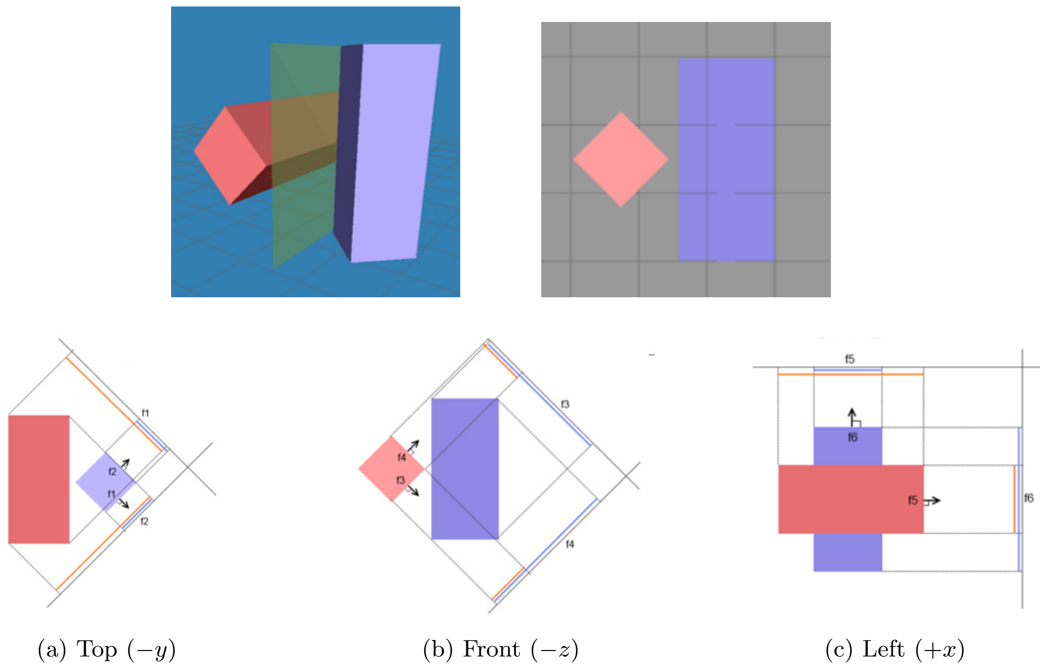



Figure 3: SAT potential collision axes

**Edge-Edge collisions in 3D**

- Figure 4 shows a case where SAT generates a false positive when two 3D objects are not actually colliding but all potential axis tests show them to be overlapping
- Generate additional potential separating axis using the cross product of every permutation of faces on both object
- Creates a set of new potential separating axes which are orthogonal to pairs of faces
- Vast increase in number of potential separating axes

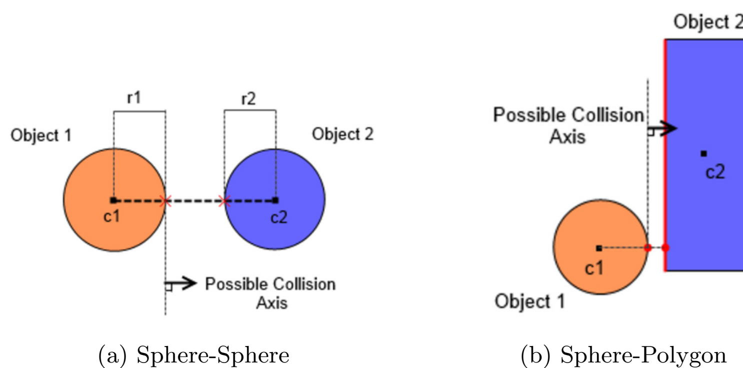


(a) Top $(-y)$ (b) Front $(-z)$ (c) Left $(+x)$

Figure 4: SAT in 3D

**Spheres and Curves**

- Curves have an infinite number of edges
- As shapes are convex only one axis must be tested to check if shapes are colliding
- For sphere-sphere simply test the distance between the closest point of object 1 with the closest point of object 2
- For sphere-polygon iterate through all faces of the polygon to find the face on which the closest point to the sphere resides, the normal of this face is the potential separating axis




(a) Sphere-Sphere (b) Sphere-Polygon

### 1.5.5 Sphere-Sphere test

- Two spheres of radii $r_1$ and $r_2$ at positions $S_1$ and $S_2$ have collided if:

$$d < r_1 + r_2$$
$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

where $d$ is the distance between the circle centres.

- For computational savings can be simplified to:

$$d^2 < (r_1 + r_2)^2$$
$$d^2 = (x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2$$

- Collision response data:

$$p = r_1 + r_2 - d$$
$$N = |S_1 - S_2|$$
$$P = S_1 - N(r_1 - p)$$

#### 1.5.6 AABB-AABB test

- Two AABB of dimensions $(w_1, h_1, l_1)$ and $(w_2, h_2, l_2)$ at positions $(x_1, y_1, l_2)$ and $(x_2, y_2, z_2)$ are interfacing the three conditions are all true:

$$|x_2 - x_1| < \frac{1}{2}(w_1 + w_2)$$
$$|y_2 - y_1| < \frac{1}{2}(h_1 + h_2)$$
$$|z_2 - z_1| < \frac{1}{2}(l_1 + l_2)$$

- Computationally cheap, but limited as bounding boxes must be axis aligned

#### 1.5.7 Sphere-Plane test

- Sphere at position $S$ of radius $r$ intersects an infinite plane with normal $N$ at distance $d$ from the origin if:

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

- Collision response data:

$$p = r - (N \cdot S - d)$$
$$N = N_{plane}$$
$$P = S - N(r - p)$$

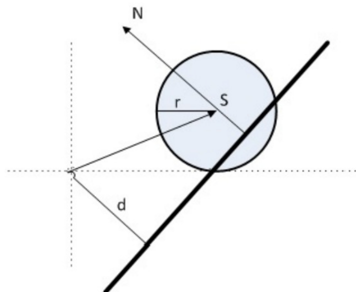



Figure 6: Sphere-Plane collision test

## 1.6 Collision Manifolds

- Specify multiple points of contact for an interface rather than just one
- Summation of surface area between two colliding objects
- Manifold is created as a 2D surface, even though by the time a collision has been detected there will be an intersecting volume between the two object

### 1.6.1 Clipping method

- Obtain a manifold that describes the initial collision points accurately
- Omit any contact points that are a result of collisions after the objects intersect each other
- With convex object typically there will be only one or two contact points

#### Worked example

- Using hypothetical collision shown in figure 7
- Have collision normal and penetration depth from SAT

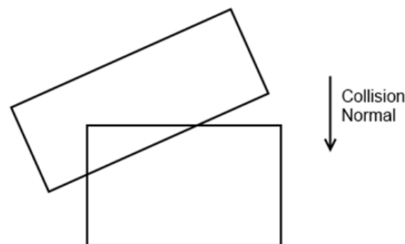



Figure 7: Example collision

Workflow:

1 Identify Significant Faces
   - Identify the two significant faces that are intersecting
   - For each shape:
     - Select the vertex furthest along the collision normal (shown in red in figure 8)
     - Select the face that includes this vertex that has a normal closest to the collision normal
   - The faces selected for both shapes are the significant faces

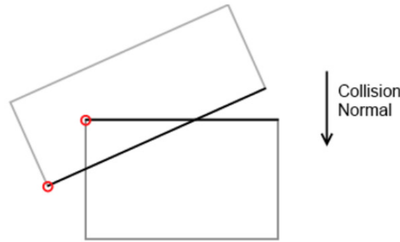



Figure 8: Significant faces

2 Determining Incident and Reference Faces

- The reference face will be the point of reference when clipping
- The incident face will be the set of vertices being clipped
- The face with a normal closest to the collision normal becomes the reference face
  The other face is the incident face

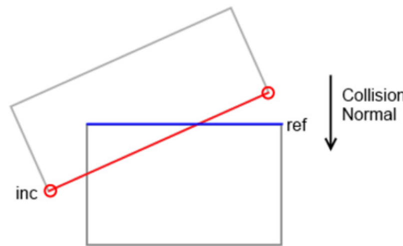



Figure 9: Incident and Reference faces

3 Adjacent face clipping

- Clip the incident face with all adjacent face of the reference face
- Using the normal of adjacent faces and any vertex on the face to produce a plane equation
- Use Sutherland-Hodgman Clipping algorithm
- Clipped edges will have new vertices added as shown on the left hand side in figure 10

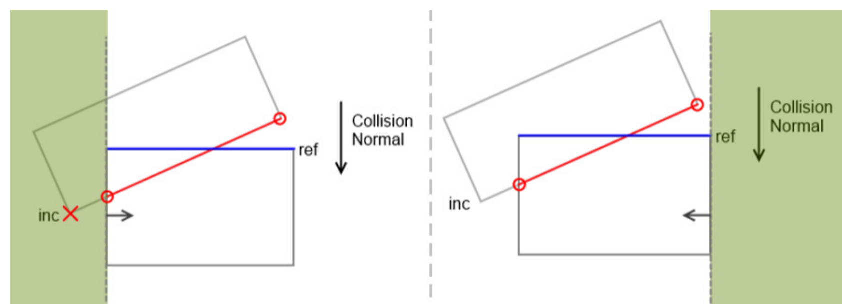



Figure 10: Adjacent face clipping (clipping region shaded green)

4 Final clipping

- Final clipping is in the plane of the reference face
- Remove all points in clipping plane (as opposed to non-destructive clipping in previous stage)
- This leaves the original collision points and removes any that occur after the two objects intersect each other

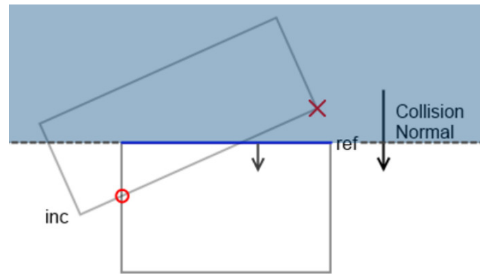



Figure 11: Final clipping (clipping region shaded blue)

## 1.7 Collision Response

- Resolve collisions using:
  - The collision manifold
  - The collision normal
  - The penetration depth
- General methods of collision response:

**Projection methods**
Control the position of colliding objects

**Impulse methods**
Control the velocity of colliding objects

**Penalty methods**
Control the acceleration of colliding objects

### 1.7.1 Impulse method

- Apply an impulse $J$ to each object:

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

- Affect velocity of each object using impulse:

$$\Delta v = \frac{J}{m}$$

**Linear impulse**

- Calculate relative velocity of colliding entities along collision normal:

$$\begin{aligned} v_{ab} &= v_a - v_b \\ v_n &= v_{ab} \cdot N \end{aligned}$$

- Final relative velocity dependant on coefficient of elasticity $\epsilon$
  - $\epsilon = 1$ is a purely elastic collision (no loss of velocity in collision)

  - $\epsilon = 0$ is a purely inelastic collision (no velocity transfer, objects stick together)

- Impulse calculated as:

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

- Ensure conservation of momentum plus impulse to resolve collision:

$$m_a v_a^f = m_a v_a^i + JN$$
$$m_b v_b^f = m_b v_b^i + JN$$

- Velocity updates:

$$v_a^f = v_a^i + \frac{J}{m_a} N$$
$$v_b^f = v_b^i + \frac{J}{m_b} N$$

**Angular impulse**

- Need to take into account velocity of contact points due to angular velocity:

$$v_{ca} = v_a + \omega_a r_a$$
$$v_{cb} = v_b + \omega_b r_b$$

- Addition to impulse calculation:

$$J = \frac{-(1+\epsilon)v_{ab} \cdot N}{N \cdot N \left(\frac{1}{m_a} + \frac{1}{m_b}\right) + \left[\left(I_a^{-1}(r_b \times N)\right) \times r_a + \left(I_b^{-1}(r_b \times N)\right) \times r_b\right] \cdot N}$$

- Ensure conservation of momentum:

$$I_a \omega_a^f = I_a \omega_a^i + r_a \times JN$$
$$I_b \omega_b^f = I_b \omega_b^i + r_b \times JN$$

- Angular velocity updates:

$$\omega_a^f = \omega_a^i + \frac{r_a \times JN}{I_a}$$
$$\omega_b^f = \omega_b^i - \frac{r_b \times JN}{I_b}$$

### 1.7.2 Penalty method

- Model collision as a spring
- Spring equation with damping:

$$F = -kx - cv$$

$F$ is the force extorted by the spring

$k$ is the spring constant

$x$ is the displacement of the spring from it s rest position

$c$ is the damping coefficient

$v$ is the velocity of the moving object attached to the spring

- Calculate force for collision:

$$F = -kx - c(N \cdot V_{ab})$$
$$F = ma$$

- Calculate acceleration of each object:

$$a_1 = \frac{F}{m_1}$$
$$a_2 = \frac{F}{m_2}$$

#### 1.7.3 Soft Bodies

- Represent soft bodies as a mesh of nodes connected by springs
- Commonly use the graphical mesh as the node mesh
- Expensive to simulate
- Cannot usually use springs to model deformable objects where the object is permanently deformed (possible depending on situation, e.g. cloth can be torn by simply removing constraints then splitting and filling in the graphical mesh)

#### 1.7.4 Constraint based collision response

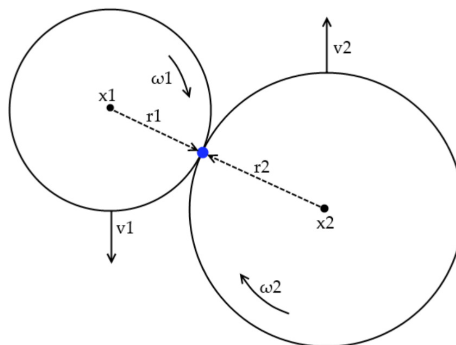



Figure 12: Collision constraint example

- Inequality restraints restrict the axis, direction, area, etc. in which a constraint can act
  Do this by clamping $\lambda$:
$$\lambda_- \leq \lambda \leq \lambda_+$$
- Contact constraint only allows constraint force to push out (does not affect other constraints)

$$\lambda_- = 0$$
$$\lambda_+ = \infty$$

- Collision constraint:
$$C = (x_x + r_2 - x_1 - r_1) \cdot N$$

- Contact point $p$ is defined as:
$$p = x + r$$
- Obtain Jacobian:
$$J = \begin{bmatrix} -N^T \\ -(r_1 \times N)^T \\ N^T \\ (r_2 \times N)^T \end{bmatrix}$$
- Collision is resolved by solving this constraint with the contact constraint on $\lambda$
- Two options for using the collision manifold:
  1. – Iterate through all contact points in manifold
     – Majority of collision resolved by first point, majority of remainder by second point, etc.
     – Slight loss of accuracy
  2. – Divide the constraint resolution between contact points

### 1.7.5 Constraints as Friction

- Define tangents to surfaces that are in contact $u_1$ and $u_2$ and their normal $n$:
$$u_1 \times u_2 = n$$
- Define collision constraints:
$$C_1 = (x_2 + r_2 - x_1 - r_1) \cdot u_1$$
$$C_2 = (x_2 + r_2 - x_1 - r_1) \cdot u_2$$
- Constraints restrict movement of the contact points alone the surface in any direction
- Obtain Jacobians:
$$J_1 = \begin{bmatrix} -u_1^T \\ -(r_1 \times u_1)^T \\ u_1^T \\ (r_2 \times u_1)^T \end{bmatrix} J_2 = \begin{bmatrix} -u_2^T \\ -(r_1 \times u_2)^T \\ u_2^T \\ (r_2 \times u_2)^T \end{bmatrix}$$
- Approximate limits on $\lambda$ based on frictional force due to gravity:
$$-\mu mg \le \lambda \le \mu mg$$
  where $m$ is object mass, $g$ is acceleration due to gravity and $\mu$ is a constant which is tuned to make the constraint believable

## 1.8 Solvers

- As more constraints are added to a physics system the need to solve all constraints simultaneously arises
- Objects may be affected by multiple constraints which may work against each other if solving constraints sequentially

- Global solver represents constraints as a series of linear equations to be solved
  e.g.

$$\begin{aligned} 4x + y &= 23 \\ x - z &= 6 \\ 2z + y &= 3 \end{aligned}$$

- Represent system of equations in $Ax = b$ notation:

$$\begin{bmatrix} 4 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & 1 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 23 \\ 6 \\ 3 \end{bmatrix}$$

  $A$ is the coefficient matrix
  $x$ is the solution (unknown) vector
  $b$ is the constant vector

- In a system of $i$ equations with $j$ unknown the general form of $Ax = b$ is:

$$\begin{bmatrix} a_{11} & a_{21} & \cdots & a_{i1} \\ a_{12} & a_{22} & \cdots & a_{i2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1j} & a_{2j} & \cdots & a_{ij} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_j \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_i \end{bmatrix}$$

- Can solve $Ax = b$ systems using multiple methods:

  **Jacobi**
    - Parallelisable
    - Slow to converge

  **Gauss-Seidel**
    - Simplest to implement

  **Successive-Over-Relaxation**
    - Faster convergence over Gauss-Seidel

  **Conjugate Gradient**
    - Fast convergence
    - Relatively complex

### 1.8.1 Gauss-Seidel

- Iterate through each row of $A$ matrix (i.e. each constraint) and solve constraints relevant to that object
- Generate a value indicating how much the velocity of the object should change
- Constraint is therefore not solved in each iteration but made to converge towards the solution over multiple iterations
- Convergence is not guaranteed in each frame due to the non-deterministic amount of time taken to solve different numbers of constraints
- Limit number of solver iterations to prevent too much time spent on solver
- Non-convergence in a single frame is unimportant as the error can always be corrected in subsequent frames and the system will eventually converge
- Order in which constraint are solved matters
  In order to converge to the same value the constraints must be solved in the same order every iteration

#### 1.8.2 Constraint Drift

- Errors can accumulate and feed forward into subsequent frames
- Commonly seen in stacked objects (in which they can bee seen to sink into each other)
- Baumgarte Stabilisation adds additional forces to the system to compensate for accumulated error
  Additional forces eventually bring solution back into convergence
- Additional energy to be added often requires trial and error tuning
  Too little energy and the system will still drift
  Too much and the system will become unstable

# 2 Artificial Intelligence

- Must balance randomness and predictability
- Must make the game sufficiently challenging that it is enjoyable to play without being unbeatable
- For example, may introduce "rubber banding" to keep a certain number of cars in a racing game near the player
- Optimisations
  - As objects get further away from the player the complexity of their behaviour can be reduced, however the same outcome must still be reached
    (e.g. a high level of detail task may be "walk to market, sell fish, buy bread, go home" but if the player never sees the NPC performing those actions then as long as there is more fish and less bread in the market they will never know if they didn't)
  - AI computations are often not time critical so can be performed over multiple frames or multiple cores

## 2.1 Finite State Machine

- Implementation types:

  **Hard coded switch statements**
  - Switch case controlling state behaviours and test for state transfer
  - OK for fast prototyping but is unmanageable for a large/complex state machine
  - Relatively computationally fast

  **Hard coded state pattern**
  - Each state is a data type that contains the transfer tests and behaviours
  - Allows addition, removal and modification of states without the risk of invalidating the rest of the FSM

  **Interpreted state pattern**
  - Similar to hard coded state patterns
  - States, behaviours and transfer conditions populated dynamically
    e.g. from definition files or scripts

- State diagrams often used in design of an FSM

  **States**
  - Represented as nodes
  - Shows the state and possibly actions

  **Transitions**
  - Represented as edges
  - Shows the possible transfers between states

**Transfer conditions**

- Represented as edge labels
- Shows the conditions required for a transition to take place

- When designing an FSM need to take into account possibility of state oscillations and add hysteresis to transfer conditions where appropriate

#### 2.1.1 Hierarchical FSM

- Hierarchical FSMs allow more complex state machines and AI behaviour to be created
- Essentially adding another FSM as a node in a parent FSM

#### 2.1.2 Behaviours and Types

- AI types can be used to define an AI for a specific type of agent
- Types are defined by a combination of behaviours (an individual state machine)
- Allows for easy scaling and reuse of AI functionality

#### 2.1.3 Fuzzy State Machines

- State machine that can combine the behaviours of multiple states
- States not restricted to being either active or inactive
  Can be a certain degree of active
- Can reduce the complexity of a state machine by allowing multiple states to be active, therefore fewer states need to exist
- State transfer tests may be more complex to implement

### 2.2 Path Planning

- Represent environment as a graph
  - Waypoints are nodes
  - Path between waypoints are edges
- Nodes can be arranged in a regular pattern or in an environment specific manner (e.g. junctions are nodes and roads are edges)
- More sophisticated approach is to use a navigation mesh, in which nodes bound regions based on the shape of the environment
- Edges are weighted to represent the difficulty of traversing between two waypoints
- For navigation alone the data required for each node is:
  - Unique ID
  - List of connected nodes

  - Position
  - Passibility flag

- Environment can be represented in any way as long as the important information above is included in the representation

#### 2.2.1 A* algorithm

- Based on a heuristic for the cost of traversing from a given node to the target node
- $f = g + h$

  $f$ is the total cost for the node under consideration

  $g$ is the cost of the shortest path from the start node to the current node

  $h$ is the estimated cost of the path from this node to the end 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
- The A* algorithm maintains two lists of nodes:

  **Open list**

    - Nodes that algorithm is aware of but not yet explored
    - Have a computed $f$ value

  **Closed list**

    - Nodes that have been explored
    - All nodes in closed list have been moved from open list
    - Final path is constructed from nodes in the closed list
- A* also requires storing a reference to a parent node in each node

  Parent node is used to denote the parent which gives the shortest path back to the start node from any given node on the open and closed lists
- 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

#### 2.2.2 Computational cost

- Environment representation can be very memory costly for complex terrains
- Dynamic terrains will require the navigation map to be updated whenever the environment changes
- Dynamic loading can make memory management for the navigation graph more complex
- Path finding is often very computationally expensive especially with a high number of possible unique paths
- The use of a heuristic path planner should be assessed and avoided if possible
- Can use a hierarchical navigation graph to split up large environments
- Commonly used or high level (in a hierarchical approach) paths may be precomputed

#### 2.2.3 Spline following

- Entities follow a predetermined path in the environment defined by a series of curves
- Commonly used in games that feature a predefined track
  (e.g. racing games)
- Can use multiple different paths to provide a sense of randomness to the player

### 2.3 Crowd management

- Simply using A* for every agent is a large scale simulation is not possible:
  - Computationally expensive
  - Not guaranteed to be correct
    Multiple agents may end up attempting to take the same path at the same time
- Crowd navigation focuses on managing the flow of a large number of agents such that they tend towards a direction rather then following an exact path
- Tend to reuse the same data used in path finding

#### 2.3.1 Fixed Group

- Workflow:
  1 Define arrangement of entities around a centre point
  2 Perform a navigation algorithm on the centre point
  3 Move the centre point along the path
  (also moving all entities relative to this point)
  4 Apply a small force to all entities in the direction of the new location
  5 Reduce applied force as centre point approaches destination
- Well suited to small groups of agents

- Can cause issues if one entity becomes separated from the group (e.g. they enter a building)
  - Entity can become trapped
  - One solution is to navigate the separated entity back to the squad using A* and switch back to fixed group navigation when they arrive back at the group
  - Could also apply forces to entities that prevent them for getting too close to objects that can cause this issue

#### 2.3.2 Embedded Map Data

- Good for navigation of large numbers of entities to a small selection of predetermined points
- Path between event node to the destination is precomputed and embedded into the map data
- Navigation then becomes a simple lookup based on the entity position which is sufficiently computationally cheap to scale to large numbers well
- Limited number of destinations (which must be known at design stage)
- High memory footprint as map size and number of destinations increases

#### 2.3.3 Flocking

- Efficient update of many entities (boids) in a flock
- On updating an entity, consider:

  **Separation**

  - How far the entity is from it's nearest neighbours
  - Manages flock density
  - Keeps entities from moving too close together such that they all occupy the same space

  **Alignment**

  - Average direction of flock
  - (normalised vector sum)

  **Cohesion**

  - Move towards average position of the flock
  - Keeps flock together

- Weighting can be applied to the three factors above to change the way the flock behaves
- Alignment can be used to set the path of the flock
  (e.g. as the result of a navigation algorithm)

#### 2.3.4 Ant Colony Optimisation

- Start with no knowledge of the environment
- Ants explore terrain while depositing a pheromone trail
- Other ants are more likely to follow an existing pheromone trail
- An optimal route by definition has more frequent traversals so over time has a stronger pheromone trail

- As a route becomes non-optimal ants stop taking it and the pheromone trail becomes weak
- Important to balance exploration (searching paths with little or weak pheromone trails randomly) and exploitation (preference for the route with the strongest pheromone trail)

## 2.4 Decision making

### 2.4.1 Decision trees

- Decision making often expressed as a tree of states
- Cheap method for simple AI entities

### 2.4.2 Goal Oriented Action Planning

- Using a graph search to determine a series of actions that lead to a given event
- Graph nodes are states
- Edges are actions

### 2.4.3 Machine Learning

- Use machine learning to alter game play based on actions of the player
- Requires a training phase where player input should (ideally) result in a given outcome
- Training obtains optimal values for the specific implementation (for a GA it is the best solution, for ANN it is the neuron connection weightings)
- Requires significant testing and (depending on the way it is integrated) state guards to prevent the game behaving in an undesirable way

# 3 Networking

## 3.1 Socket protocols

**Stream (TCP)**

- Packets are received in order
- Reliable packet delivery
- Slower
- Better suited for transport of "important" data
  e.g. leaderboard data, in game transactions, etc

**Datagram (UDP)**

- No guaranteed order
- No guaranteed delivery
- Faster
- Better suited for frequent, time critical updates
  e.g. updating player positions in a game, etc

## 3.2 Topologies

**Client-Server**

- Clients connect to a managed server
- Can guarantee behaviour of server
- Can cause bandwidth issues if player is far from server geographically

**Client-Client/Server**

- Clients connect to another client running as a server
- Still requires some kind of discovery service (e.g. through a managed server that only has this role)
- Can help to reduce bandwidth issues caused by geolocation

**Peer-peer**

- Clients connect direct to one another
- Still requires some kind of discovery service (e.g. through a managed server that only has this role)
- Can help to reduce bandwidth issues caused by geolocation

## 3.3 Constraints of Network Gaming

Two main issues:

**Real time**

- If an event is triggered by player $U_1$ at time $t$, then all players should see the event happen at time $t$
- Consider if an element of the simulation need to be updated across all clients
  (e.g. typically gravity is constant and does not need to be updated over the network)
- Consider if an element needs to be updated in real time or if a slower periodic (synchronising) update would suffice
  (e.g. the position of a skybox determined by time of day can be updated locally on all clients and occasionally synchronised with the server)

**Consistency**

- If player $U_1$ affects player $U_2$, then all players should see $U_1$ affecting $U_2$
- Simple solution is to use TCP over UDP
- Design of game can be such that when consistency is desired the real time requirement is relaxed

### 3.3.1 Zoning

- Reducing the number of updates that are sent based on the location of objects in the game world
- Determining if clients might interact with each other
- Analogous to broadphase in collision detection
- Zones often distributed across several servers

**High Level Zoning**

- Analogous to fixed world space partitioning in collision detection
- Split world into several distinct zones
- Clients in each zone should not be able to affect those in other zones
- Assumes an even distribution of players across all zones

**Spatial Zoning**

- Aims to subdivide the world into zones without having to stop on a loading screen while the player changes zones
- Server must pre-empt when the player will change between zones
- Player leaving one zone is disconnected from one server and joins another
- Density and shape of zones can be modified based on the rate at which players are entering leaving them
- Level design can be used to prevent players from simply appearing in a world or the environment of the new zone suddenly appearing when players switch zones

#### 3.3.2 Interest Management

- Determining if clients actually do interact with one another
- Analogous to narrowphase in collision detection

**Behavioural Modelling**

- Adjusting the area of interest based on the behaviours of the client
- (e.g. a client flying a plane will have a larger area of interest than a client driving a jeep)

**Publish-Subscriber Model**

- Common method of interest management
- A client will subscribe to relevant updates from objects it is interested in/affected by
- Reduces network traffic at the cost of increased server load
- Can play a part in making network games cheat proof

**Aura-Nimbus Approach**

- Client has two radii:

  **Aura**
  Range of influence

  **Nimbus**
  Range of interest

- If client $A$ is within the aura of client $B$ and $B$ is in the nimbus of $A$ then $A$ receives updates from $B$
- If all aura and nimbi overlap then no performance advantage is seen, rather a performance overhead will be incurred

#### 3.3.3 Dead Reckoning

- In distributed games there is no single global state that can be said to be true for all players
- Need to keep individual players world states in sync to a believable standard
- Can base world synchronisation on 0th, 1st or 2nd order differential of position
- 0th order (i.e. position itself) is not used for dead reckoning, rather frequent state regeneration
- The choice between velocity and acceleration depends on which of the two is the least volatile, the least volatile value should be chosen to reduce the chances of lost packets causing the world states to become too far out of sync

**Convergence of World Views**

- If two world states are too far diverged then resynchronising them is impossible, game design should prevent this from such a state being possible

  (e.g. player $A$ thinks they have killed player $B$ but $B$ thinks they have dodged player $A$'s attack)

- Can use frequent state regeneration on variables that denote the state of other clients

  (e.g. a client that has killed another player may wait for that player to confirm that they are dead)

- Server may frequently broadcast the positions of all entities and any that are out of sync can interpolate their position with that broadcast by the server to synchronise with the rest of the world states

# 4 Massively parallel and Heterogeneous computing

- Splitting large scale numerical tasks over multiple computational devices
- In the case of GPU computing this is just on the graphics card
- Heterogeneous computing frameworks allow targeting a wide range of devices (e.g. CPUs, GPUs, crypto-coprocessors, FPGAs, etc.)

## 4.1 GPU computation

- GPU arranged in several grids of blocks, each block executes a set of threads
- Threads and block have their own (shared) memory
- Global/constants memory accessible to all blocks
- A copy to/from the GPU has an overhead which is not overly dependant on the size of that data, for this reason small copies are often less efficient than single large copies
- Data available per thread is limited
- The idea problem for GPU computation is one that is embarrassingly parallel (e.g. updating the positions of many game entities or running a particle system)