# Revision

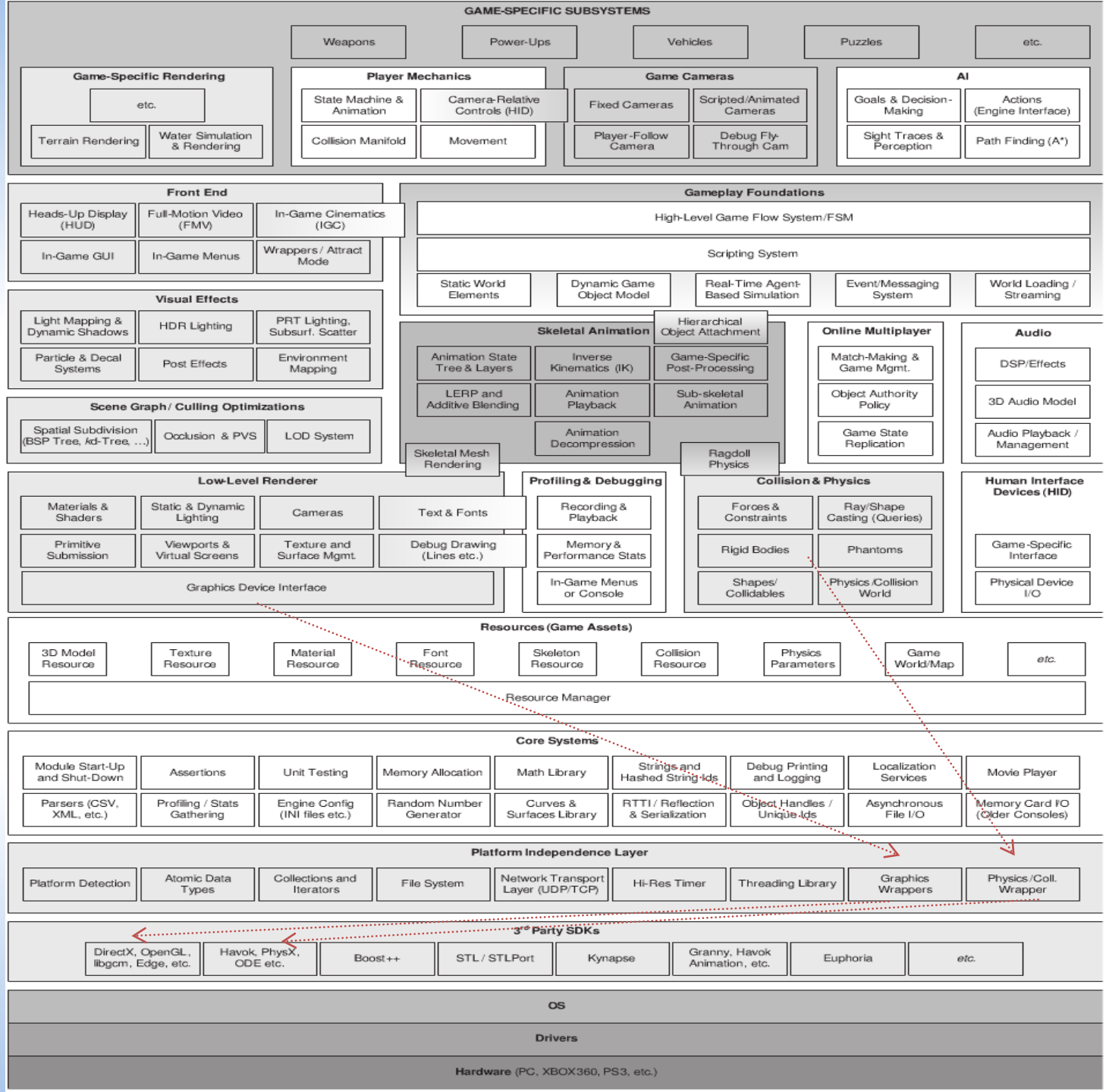
Summary of the Subject

# Game Engines

- What is a game engine?
  - ➢ The core software of the game
    - Describes a set of code used to build the game application
    - Software suites that provide the technological underwiring for games
      - Everything you see on the screen and interact with within the game world is powered by the game engine
    - A framework comprised of a collection of different tools, utilities, and interfaces that hide the low-level details of the various tasks that make up the game
    - Exists to abstract the details of doing common game-related tasks (e.g. rendering, physics, input) so that developers can focus on the details that make their games unique

# Game Engines

- Game engine
  - ➢ No clear boundary between the game and game engine
    - Hard-coded game logic or game rules make it difficult to reuse that software for other games
  - ➢ Ideally, a game engine should be extensible
    - Can be used as the foundation for many different games without major modification
    - However, many trade-offs involved
      - Depends on what the engine will be used for
        - » E.g. different genres have different characteristics and focus
      - Trade-off between generality and optimality
  - ➢ Data-driven architecture is what differentiates a game engine from a program that is a game but not an engine


GAME-SPECIFIC SUBSYSTEMS
Weapons
Power-Ups
Vehicles
Puzzles
etc.
Game-Specific Rendering
etc.
Terrain Rendering
Water Simulation & Rendering
Player Mechanics
State Machine & Animation
Camera-Relative Controls (HID)
Collision Manifold
Movement
Game Cameras
Fixed Cameras
Scripted/Animated Cameras
Player-Follow Camera
Debug Fly-Through Cam
AI
Goals & Decision-Making
Actions (Engine Interface)
Sight Traces & Perception
Path Finding (A*)
Front End
Heads-Up Display (HUD)
Full-Motion Video (FMV)
In-Game Cinematics (IGC)
In-Game GUI
In-Game Menus
Wrappers / Attract Mode
Visual Effects
Light Mapping & Dynamic Shadows
HDR Lighting
PRT Lighting, Subsurf. Scatter
Particle & Decal Systems
Post Effects
Environment Mapping
Scene Graph / Culling Optimizations
Spatial Subdivision (BSP Tree, kd-Tree, ...)
Occlusion & PVS
LOD System
Gameplay Foundations
High-Level Game Flow System/FSM
Scripting System
Static World Elements
Dynamic Game Object Model
Real-Time Agent-Based Simulation
Event/Messaging System
World Loading / Streaming
Skeletal Animation
Hierarchical Object Attachment
Animation State Tree & Layers
Inverse Kinematics (IK)
Game-Specific Post-Processing
LERP and Additive Blending
Animation Playback
Sub-skeletal Animation
Animation Decompression
Skeletal Mesh Rendering
Ragdoll Physics
Online Multiplayer
Match-Making & Game Mgmt.
Object Authority Policy
Game State Replication
Audio
DSP/Effects
3D Audio Model
Audio Playback / Management
Low-Level Renderer
Materials & Shaders
Static & Dynamic Lighting
Cameras
Text & Fonts
Primitive Submission
Viewports & Virtual Screens
Texture and Surface Mgmt.
Debug Drawing (Lines etc.)
Graphics Device Interface
Profiling & Debugging
Recording & Playback
Memory & Performance Stats
In-Game Menus or Console
Collision & Physics
Forces & Constraints
Ray/Shape Casting (Queries)
Rigid Bodies
Phantoms
Shapes/ Collidables
Physics/Collision World
Human Interface Devices (HID)
Game-Specific Interface
Physical Device I/O
Resources (Game Assets)
3D Model Resource
Texture Resource
Material Resource
Font Resource
Skeleton Resource
Collision Resource
Physics Parameters
Game World/Map
etc.
Resource Manager
Core Systems
Module Start-Up and Shut-Down
Assertions
Unit Testing
Memory Allocation
Math Library
Strings and Hashed String Ids
Debug Printing and Logging
Localization Services
Movie Player
Parsers (CSV, XML, etc.)
Profiling / Stats Gathering
Engine Config (INI files etc.)
Random Number Generator
Curves & Surfaces Library
RTTI / Reflection & Serialization
Object Handles / Unique Ids
Asynchronous File I/O
Memory Card I/O (Older Consoles)
Platform Independence Layer
Platform Detection
Atomic Data Types
Collections and Iterators
File System
Network Transport Layer (UDP/TCP)
Hi-Res Timer
Threading Library
Graphics Wrappers
Physics/Coll. Wrapper
3rd Party SDKs
DirectX, OpenGL, libgcm, Edge, etc.
Havok, PhysX, ODE etc.
Boost++
STL / STLPort
Kynapse
Granny, Havok Animation, etc.
Euphoria
etc.
OS
Drivers
Hardware (PC, XBOX360, PS3, etc.)

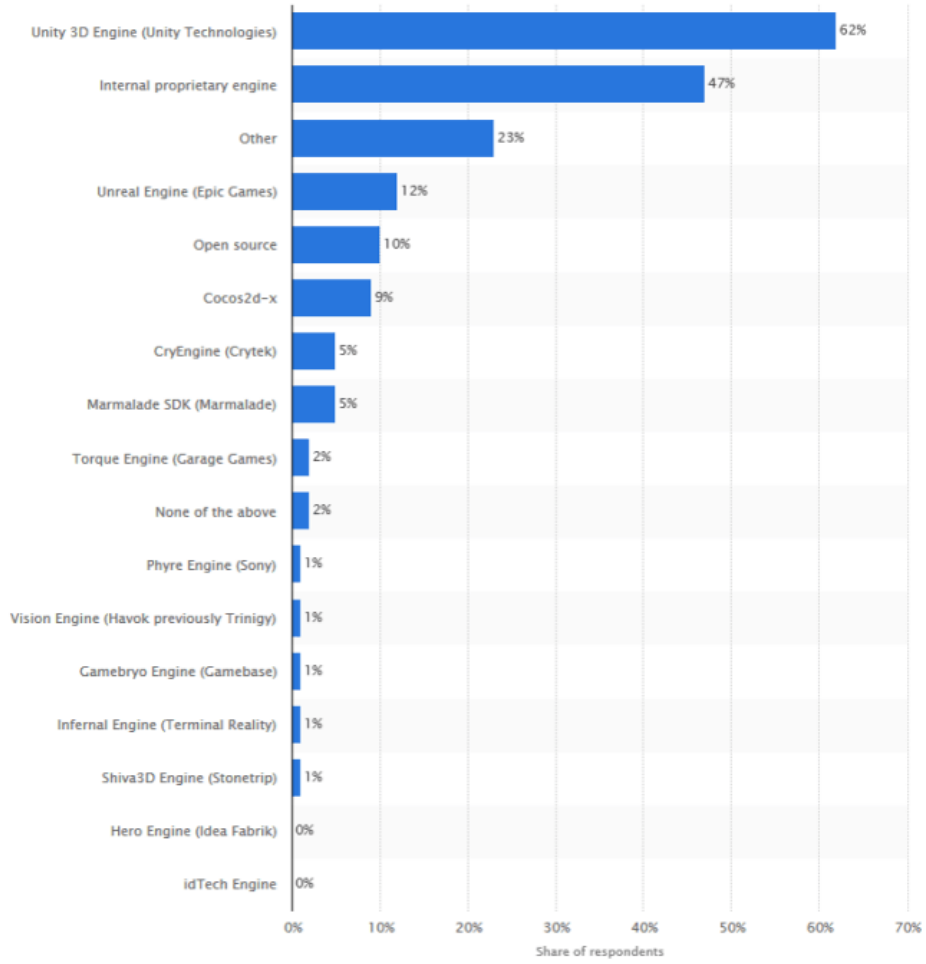
# Game Engines

- Choosing a game engine
  - Each engine has its advantages and disadvantages
  - Depends on the developer's needs in terms of budget, schedule, and the game itself
  - Several criteria to consider
    - Quality and performance
      - Technologies used for rendering, animation, simulation of physics, AI, etc
    - Ease of use
      - Each game engine has its own editor, user community forums usually have vigorous discussions about the plusses and minuses

# Game Engine Survey (2016)

**What game engines do you currently use?**

This statistic illustrates the most popular game engines used for video game development in the United Kingdom (UK) in 2014. The greatest share of respondents reported using Unity 3D Engine, at 62 percent.



# The Rendering Loop

- The rendering loop
  - For the GUI of normal applications
    - Contents on screen are mostly static
    - Only re-draw sections of screen that change via rectangle invalidation
    - Some older games use similar approaches
      - (CPUs were much slower then)
  - In real-time 3D graphics
    - Moving camera in 3D scene
    - Entire screen contents continually change
    - Use a loop to present user with a series of still images in rapid succession


http://www.stibbe.net/History/Games_Speech/Pong.htm

# The Game Loop

- Interacting subsystems
  - Rendering, input/output systems, animation, collision detection, physics simulations, networking, audio, etc.
  - Most require periodic servicing
    - The servicing rate for each subsystem may vary
- The game loop
  - Master loop that updates everything
  - Needs some form of time management

# Time

- Frame rate
  - How rapidly the sequence of still 3D frames is presented to the viewer, typically measured in frames-per-second
- Frame time
  - Also known as time delta/delta time ($\Delta t$ or $dt$)
  - Amount of time that elapses between frames
  - Can measure by simply subtracting time from previous frame with time at current frame
    - Then made available to all engine subsystems
    - Note that using current frame time for upcoming frame isn't necessarily accurate

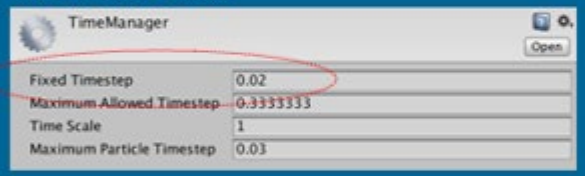
# Time

- Frame time (cont.)
  - Frame rate spikes
    - Certain frames take longer than others
  - Running average
    - Allows game to adapt to varying frame rates, softening effects of spikes
    - The longer the averaging interval
      - The less responsive the game will be to varying frame rates, but spikes will have less of an impact
  - Governing the frame rate
    - Rather than guessing, try to guarantee every frame duration
    - Put main thread to sleep until target frame time has elapsed

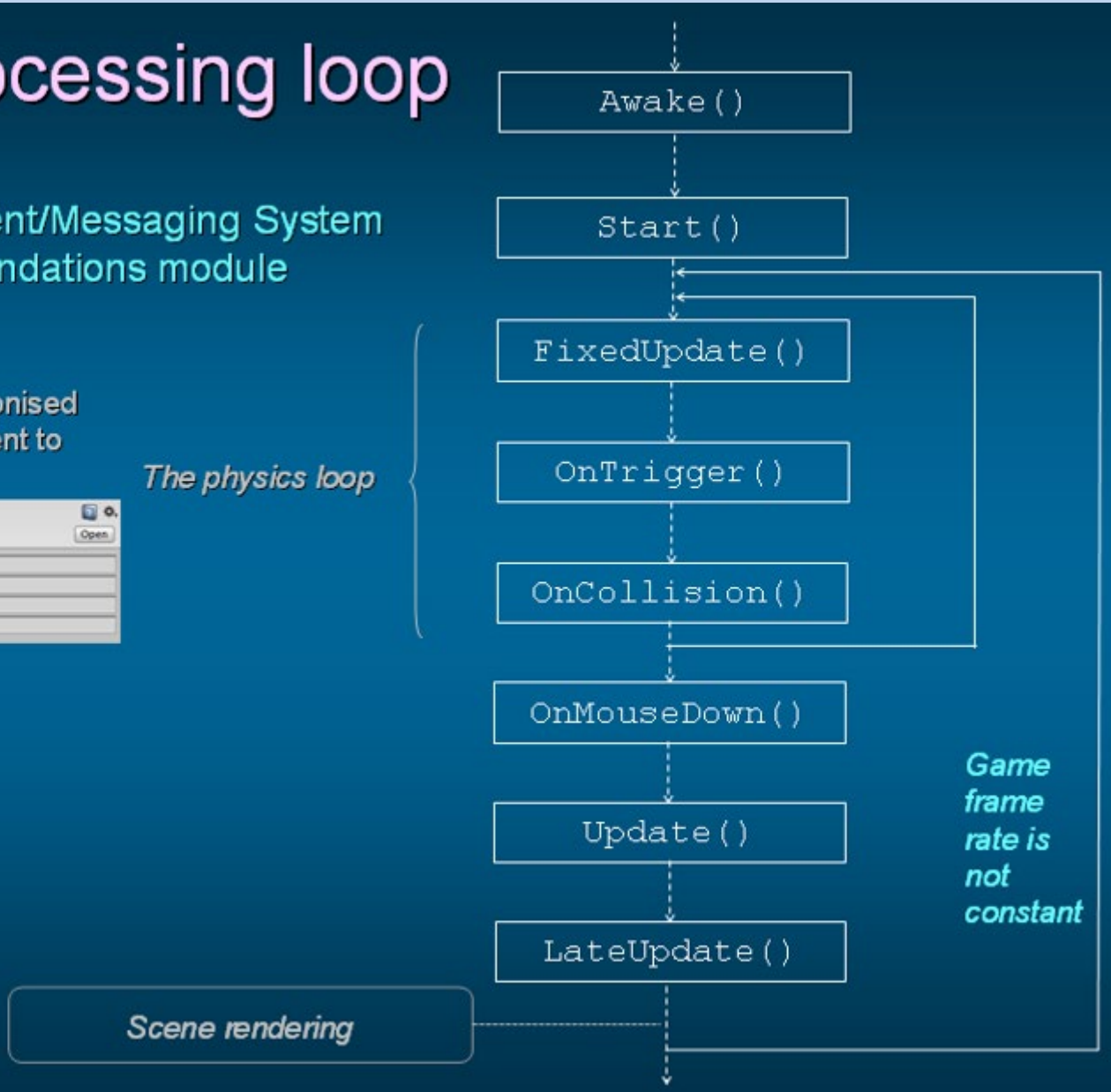
# Event processing loop

Controlled by the Event/Messaging System of the Gameplay Foundations module

The physics loop is synchronised by a timer that is independent to the frame rate




Edit -> Project Settings -> Time




https://docs.unity3d.com/Manual/ExecutionOrder.html

# Unity Essentials

- **GameObject**
  - Unity's GameObject class represents anything that can exist in a Scene
    - The base class of all entities in Unity scenes
    - Building blocks for scenes
  - Acts as a container for **Components**
    - Determine how the GameObject looks and what it does
    - Always has a Transform component
      - Position, rotation and scale
  - Provides a collection of methods
    - Can work with these in code, e.g.,
      - Setting and checking properties
      - Adding/removing components

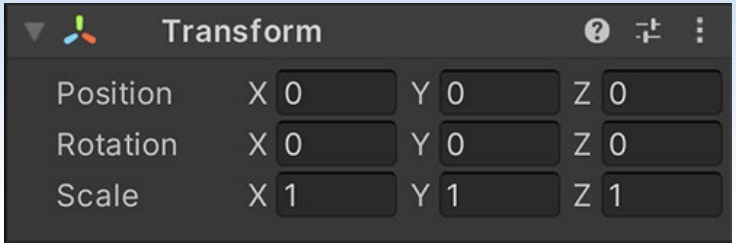
# Unity Essentials

- **Components**
  - ➢ Functional pieces of every GameObject
    - Contain editable properties that define the behaviour of a GameObject
    - With a GameObject selected, components attached to it and their properties appear in the Inspector window
  - ➢ Can attach many components to a GameObject
    - But every GameObject can only have one Transform component
    - Composition relationship rather than inheritance
  - ➢ Create a component with a script to customise behaviour

# Scripts

- Scripting
  - ➢ Create your own components using scripts
    - Trigger game events, modify component properties over time, respond to user input, etc.
    - On creation, the name of the new script used as the class name
      - Must be the same to enable the script component to be attached
    - Scripts are a kind of blueprint
      - When attached to a GameObject, it creates a new instance of the object defined by the blueprint
  - ➢ Initialisation is not done using a constructor
    - Construction of objects handled by the editor
    - **Do not define a constructor**
      - Defining a constructor for a script component will interfere with the normal Unity operation

# Scripts



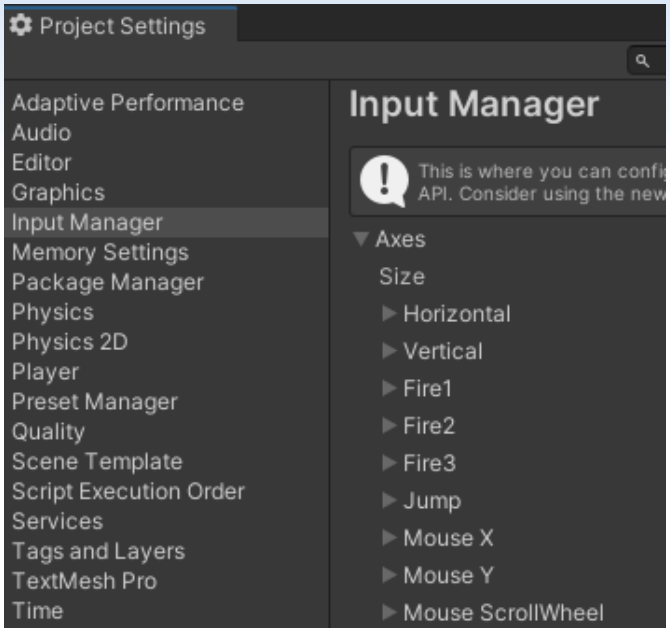

- Transform class
  - Store and manipulate position, rotation and scale
  - Some public variables
    - `Position` – a Vector3 that stores position in world space
    - `Rotation` – a Quaternion that stores rotation in world space
    - `gameObject` – the GameObject this component is attached to
  - Some public methods
    - `Translate()` – move the transform by the translation direction and distance
    - `Rotate()` – applies a rotation, often in Euler angles
    - `LookAt()` – rotates the transform so the forward vector points at target's current position

# Scripts

- Time class
  - Provides numeric values to measure time elapsing while game is running
  - Some important properties
    - `Time.time` – read-only, time (in seconds) since project started playing
    - `Time.deltaTime` – read-only, time (in seconds) elapsed since the last frame. Varies depending on the frames per second rate
    - `Time.timeScale` – controls the rate at which time elapses
    - `Time.fixedDeltaTime` – controls the interval of Unity's fixed timestep loop (for physics)

# Scripts

- Input
  - Every project has several input axes created by default
    - Enables keyboard, mouse, and joystick input regardless of device
  - An axis receives a value in the range of [–1..1]




```
float move = Input.GetAxis(“Horizontal”);
if(Input.GetButton("Fire1"))
```

  - Alternatively, can get input directly

```
float left = Input.GetKey(KeyCode.A);
if(Input.GetMouseButton(0))
```

# Scripts

- Serialising
  - Public variables are displayed in the Inspector
    - Can change their values at runtime
  - Instead of using public variables
    - Using `[SerializeField]` makes the variable appear in the Inspector, but is a private variable
- Component dependencies
  - A component might depend on other components being attached to the GameObject
  - Can enforce dependency, e.g., `[RequireComponent(typeof(Rigidbody))]`

# Scripts

- **Coroutines**
  - When a normal method is called
    - It runs to completion, then returns control to the calling method
    - Any action that takes place within the method must happen within a single frame update
  - A coroutine
    - A method that can pause its execution, return control to Unity, then continues where it left off on the following frame
    - Allows a task to be spread across several frames

# Scripts

- Interaction between objects
  - Messaging system
    - Call a method that is implemented in another script attached to the same object, or another object, and pass a parameter to it
  - Public method
    - Get a component of the target object (which is a script) using the component's name and call its public method explicitly

# Scripts

```
// A script attached to "enemy" object
. . .
public int health = 5;
  .   .   .
void HitByLaser ( int damage ) {
        health -= damage;
}
```

```
  .   .   .
private RaycastHit hitInfo;  // a structure initialized by Raycast() if the ray hits an object
  .   .   .
void Update () {
    .   .   .
     if ( Physics.Raycast (transform.position, directiononOfFire, out hitInfo, 20) ) {
           hitInfo.transform.SendMessage( "HitByLaser",  hDamage );
     }
    .   .   .
    // another way to call HitByBullet() that is a member function of another class
    GameObject.Find("enemy").SendMessage("HitByLaser", hDamage);
}
```

# Scripts

```
public class TargetHit : MonoBehaviour {
    public int health = 5;
    .   .   .
    public void HitByBullet ( int damage ) {
        health -= damage;
    }
}
```

```
  .     .     .
private RaycastHit hitInfo;
  .     .     .
void Update () {
    .     .     .
      if( Physics.Raycast (transform.position, directiononOfFire, out hitInfo, 20) )  {
            TargetHit  target = hitInfo.transform.gameObject.GetComponent<TargetHit>();
            target.HitByBullet( hDamage );
      }
    .     .     .
}
```

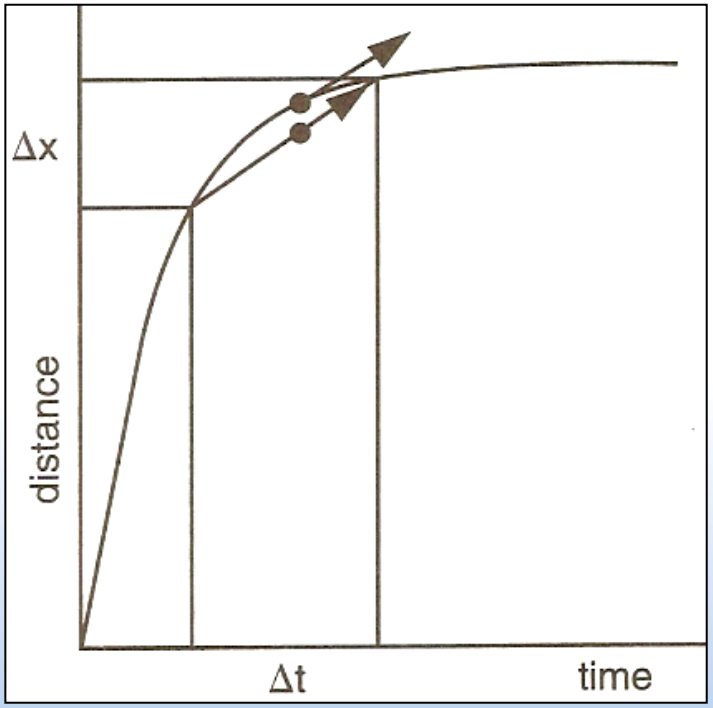
# Dynamics of Particles

- Dynamics of particles (point mass)
  - From a physics simulation point of view
    - Only consider its position only
  - The object's orientation, size, shape and structure
    - Irrelevant in the context of a particle
    - E.g., simulating a bullet's motion
- Kinematics
  - The study of motion, not considering that which causes it
- Kinetics
  - Deals with forces and interactions that produce or affect motion

# Dynamics of Particles

- Linear motion
  - Velocity (instantaneous velocity)
    - Shrink the time interval (Δt) closer and closer to 0
    - The limit as Δt approaches 0
    - Differentiate with respect to time

$$\vec{v} = \lim_{\Delta t \to 0} \frac{\Delta \vec{s}}{\Delta t} = \frac{d\vec{s}}{dt}$$



# Dynamics of Particles

- Linear motion
  - Acceleration
    - Rate at which velocity is changing
      - An object accelerating quickly, is rapidly increasing in velocity
      - Decelerating, decreasing in velocity (i.e. slowing down)
    - A positive value represents speeding up
    - Zero acceleration means no change in velocity
    - A negative value represents slowing down
    - Differentiate velocity with respect to time

$$\vec{a} = \lim_{\Delta t \to 0} \frac{\Delta \vec{v}}{\Delta t} = \frac{d\vec{v}}{dt} = \frac{d^2\vec{s}}{dt^2}$$

# Dynamics of Particles

- Force and motion
  - ➢ Kinetics
    - Why does a particle accelerate?
  - ➢ Force
    - An influence which tends to change the constant velocity motion of an object
  - ➢ Newton's laws of motion
    - Three physical laws
      - Provide relationships between forces acting on an object and the motion of the object

Use the force

# Dynamics of Particles

- Newton's first law (Law of Inertia)
  - An object will remain at rest, or continue to move at a constant velocity, unless acted upon by a net force
    - A stationary object will not move until a net force acts upon it
    - An object that is in motion will not change its velocity (will not accelerate) until a net force acts upon it
- Inertia
  - The resistance of an object to any change in its state of motion

# Dynamics of Particles

- Newton's second law (Law of Acceleration)
  - Any change in motion involves an acceleration
    - The Newton's first law is a special case of the second law for which the net force is zero
  - A net force acting on an object produces acceleration that is proportional to the object's mass

$$\vec{F} = m\vec{a}$$

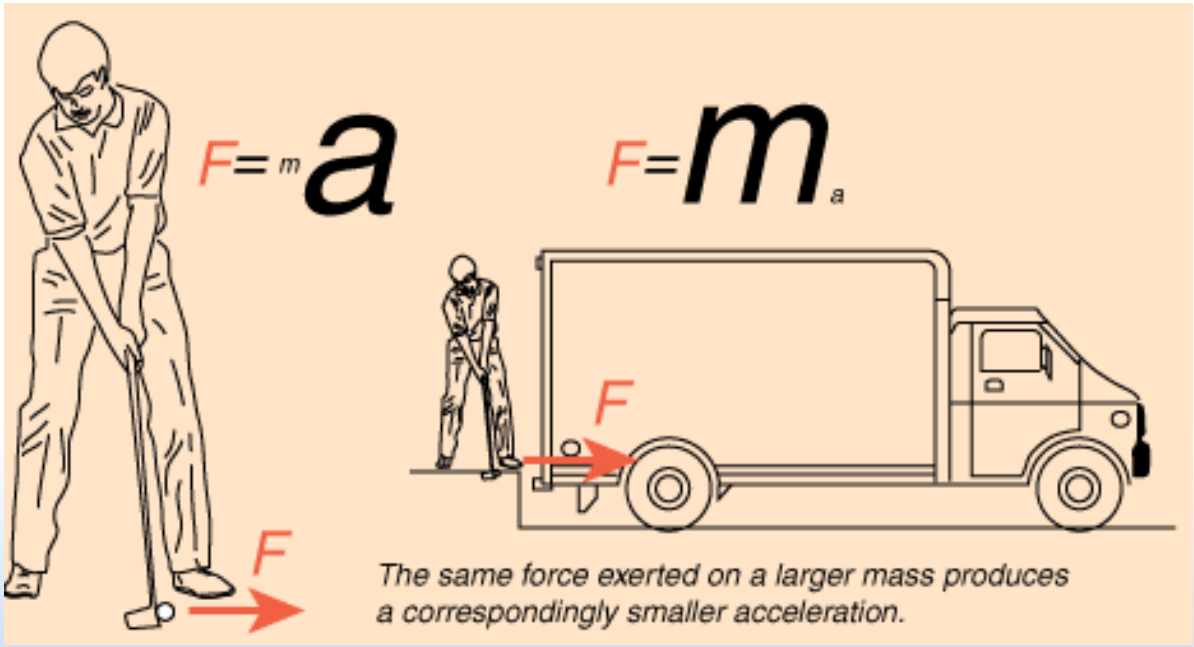
  - Mechanism by which forces alter the motion of an object
    - A force is something that changes the acceleration of an object
  - Apply a force to a particle

```
acceleration = force/mass;
```

# Dynamics of Particles

- Mass
  - Property of an object that makes it resist acceleration (inertia)
  - Measured in kg

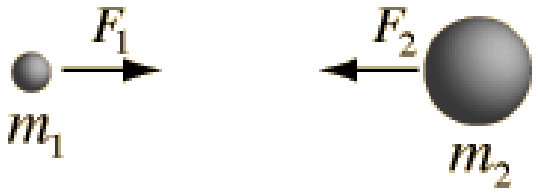
$$\vec{F} = m\vec{a}$$




http://hyperphysics.phy-astr.gsu.edu/hbase/newt.html#fma

# Dynamics of Particles

- Newton's third law
  - If object 1 exerts a force on object 2, object 2 also exerts a force on object 1
    - The forces are equal in magnitude and opposite in direction
  - A mechanism for calculating collision responses




$$\vec{F}_1 = -\vec{F}_2$$

# Dynamics of Particles

- Unity
  - ➢ Rigidbody
    - Used to control an object's movement and position through Unity's physics engine
    - In a script, the `FixedUpdate()` method is recommended as the place to apply forces and change Rigidbody settings
      - The reason being physics updates are carried out in measured time steps that don't coincide with the frame update
      - `FixedUpdate()` is called immediately before each physics update and so any changes made there will be processed directly

# Dynamics of Particles

- Rigidbody class
  - Some public variables
    - `isKinematic` – controls whether physics affects the rigidbody
    - `mass` – the rigidbody's mass
    - `useGravity` – controls whether gravity affects the rigidbody
    - `velocity` – rate of change of the rigidbody's position
  - Some public methods
    - `AddForce` – adds a force to the rigidbody
    - `AddExplosionForce` – simulates an explosion force
    - `AddForceAtPosition` – a force at a position results in a force and a torque applied to the rigidbody
    - `GetAccumulatedForce` – returns the accumulated force on the rigidbody before the simulation step

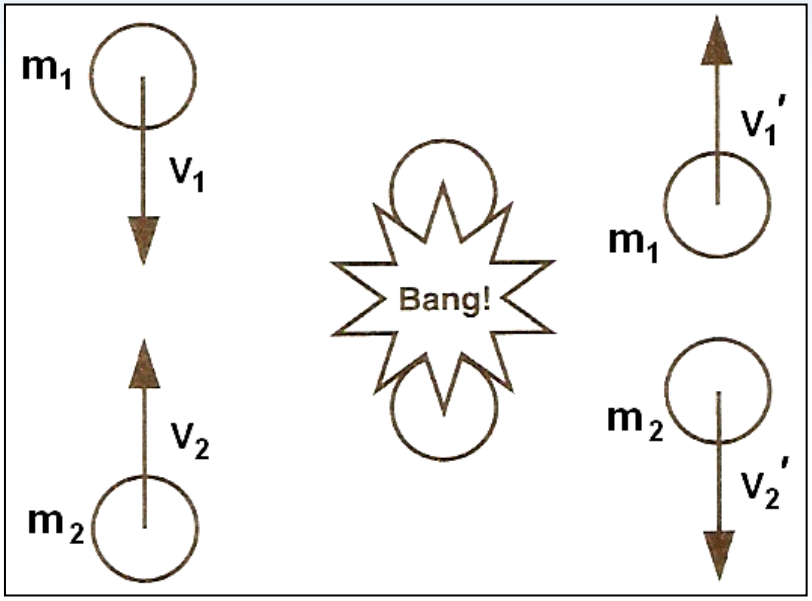
# Dynamics of Particles

- Elastic collision
  - Conservation of momentum

$$m_1\vec{v}_1 + m_2\vec{v}_2 = m_1\vec{v}_1' + m_2\vec{v}_2'$$

  - Conservation of kinetic energy

$$\frac{1}{2}m_1v_1^{\,2} + \frac{1}{2}m_2v_2^{\,2} = \frac{1}{2}m_1v_1'^{2} + \frac{1}{2}m_2v_2'^{2}$$




$$v_1' = \frac{(m_1 - m_2)v_1}{m_1 + m_2} + \frac{2m_2v_2}{m_1 + m_2}$$

$$v_2' = \frac{(m_2 - m_1)v_2}{m_1 + m_2} + \frac{2m_1v_1}{m_1 + m_2}$$

# Dynamics of Particles

- Coefficient of restitution, e
  - Represents the elasticity of a collision
    - "Bounciness"
  - Collisions in the real world are neither perfectly elastic or inelastic
    - Perfectly inelastic collision, e = 0
      - » Difference in final velocities is zero
    - Perfectly elastic collision, e = 1
      - » Velocities before and after collision equal in magnitude opposite in direction

$$e = \frac{-(v_1' - v_2')}{v_1 + v_2}$$

$$v_1' = \frac{(m_1 - em_2)v_1 + (1+e)m_2 v_2}{m_1 + m_2}$$

$$v_2' = \frac{(m_2 - em_1)v_2 + (1+e)m_1 v_1}{m_1 + m_2}$$

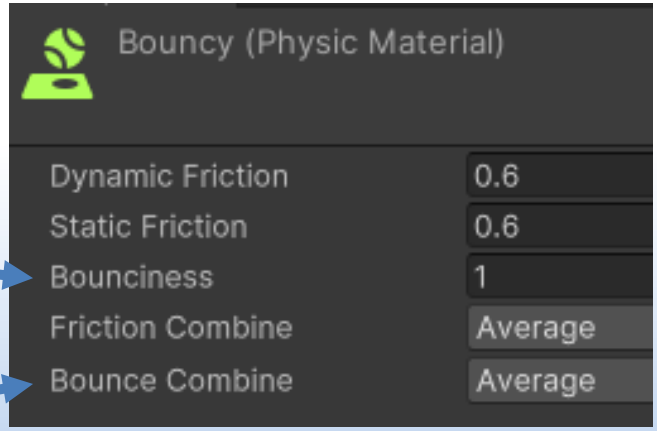
http://en.wikipedia.org/wiki/Coefficient_of_restitution

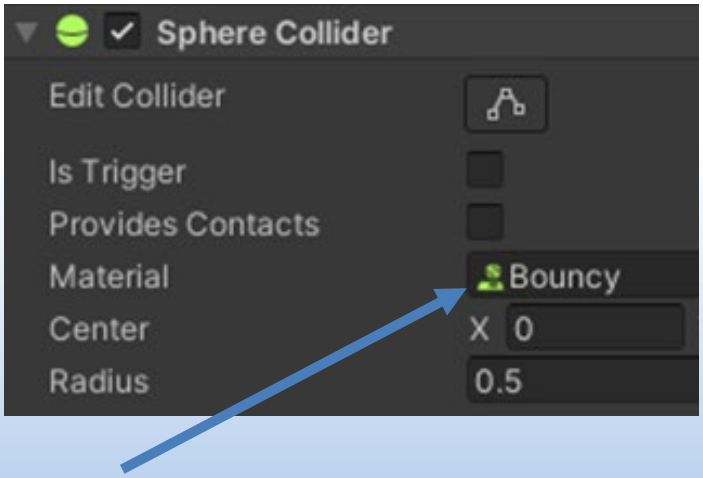
# Dynamics of Particles

- Unity
  - ➢ Physic material
    - Adjusts friction and bouncing effects of colliding GameObjects
    - Applied to a GameObject's Collider
    - Two colliding objects may have different coefficient of restitutions
      - Can set a method for combining them

Coefficient of restitution

Combining



# Dynamics of Particles

- Unity
  - `ForceMode`
    - To specify how to apply a force using `Rigidbody.AddForce()`
    - Ignores mass
      - `Acceleration` : adds a continuous acceleration to the rigidbody
      - `VelocityChange` : Adds an instant velocity change to the rigidbody
    - Uses mass
      - `Force` : adds a continuous force to the rigidbody
      - `Impulse` : adds an instant impulse to the rigidbody

# Collision Detection

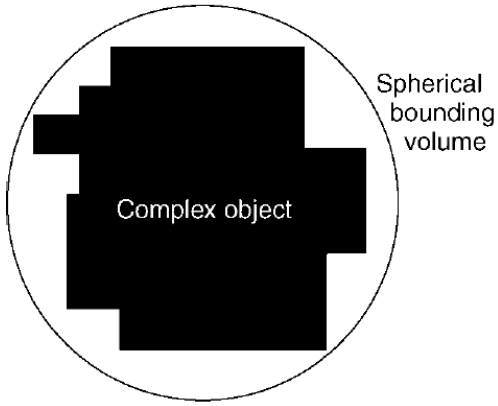
- Unity
  - `Collider`
    - Base class for all colliders, e.g.,
      - `BoxCollider, SphereCollider, CapsuleCollider, MeshCollider`
    - If an object with a Collider needs to be moved during gameplay, should attach a Rigidbody component
      - The Rigidbody can be set to be kinematic if you don't want the object to have physical interaction with other objects
    - Some properties and methods
      - `isTrigger`: specifies if the collider used as a trigger
      - `OnCollisionEnter`: called when this collider touches another collider

# Collision Detection

- Computational complexity
  - Key problems
    - Too many possible collisions
    - Expensive checks
  - To reduce the number of checks
    - Phase 1: Coarse collision detection (broad phase)
      - Find sets of objects *likely* to be in contact
    - Phase 2: Fine collision detection (narrow phase)
      - Check whether candidate collisions (from broad phase) are actually in contact
      - If so, obtain contact information (contact generation)

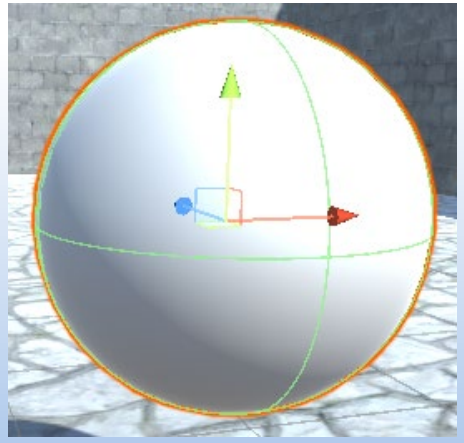
# Bounding Volumes

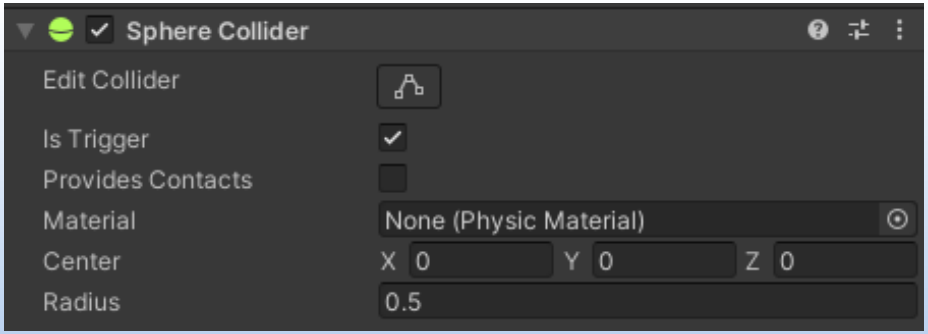
- Bounding volumes
  - An area of space known to contain the object
  - Large enough for the object to be inside
    - Ideally, as close fitting as possible
  - Simple shape used
    - Simplifies collision tests
    - Simplifies repositioning
    - Minimises data storage overheads
  - Concept
    - If the bounding volumes of two objects do not touch
      - The objects inside them cannot be in contact



# Bounding Volumes

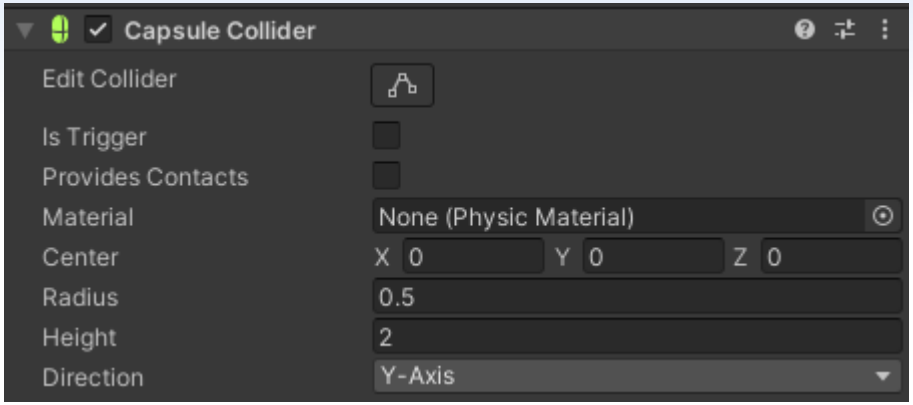
- Bounding sphere
  - ➢ Easy to represent, only needs
    - Centre and radius
  - ➢ Invariant under rotation
    - When moving, only its position needs to be updated
  - ➢ Easy to check whether two spheres overlap
  - ➢ Unity: `SphereCollider`

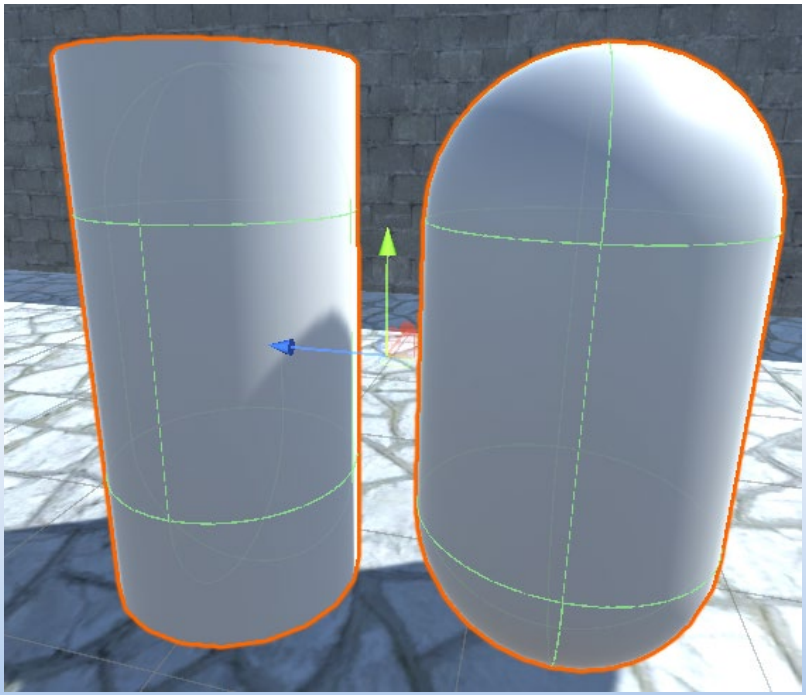


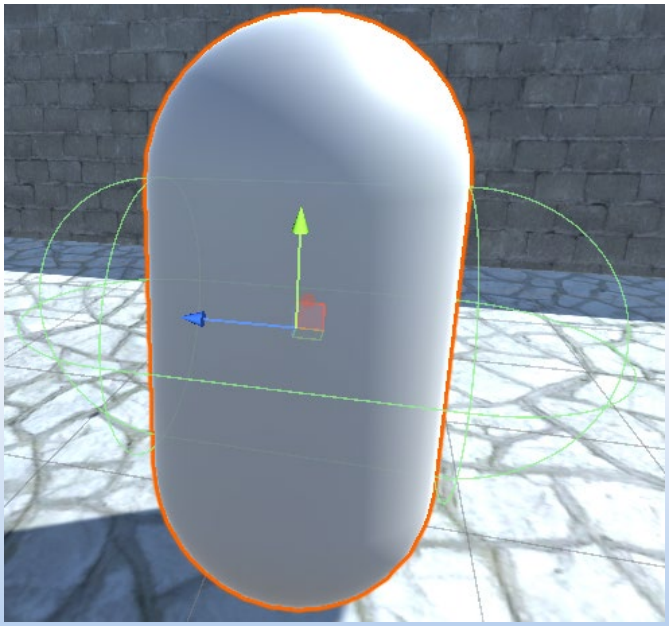


# Bounding Volumes

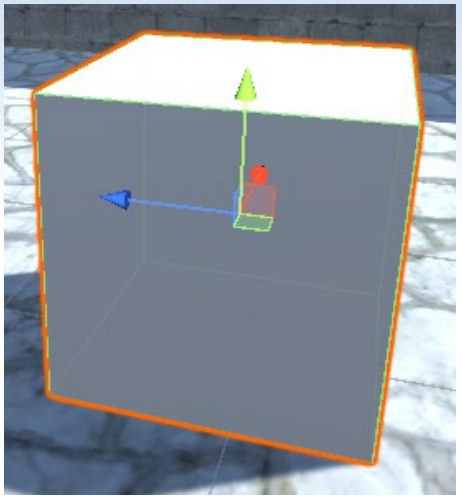
- Unity
  - `CapsuleCollider`
    - Used for cylinders as well
    - Can set Direction



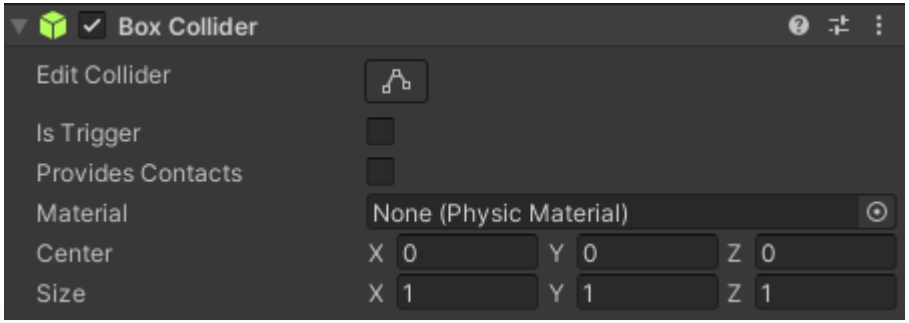



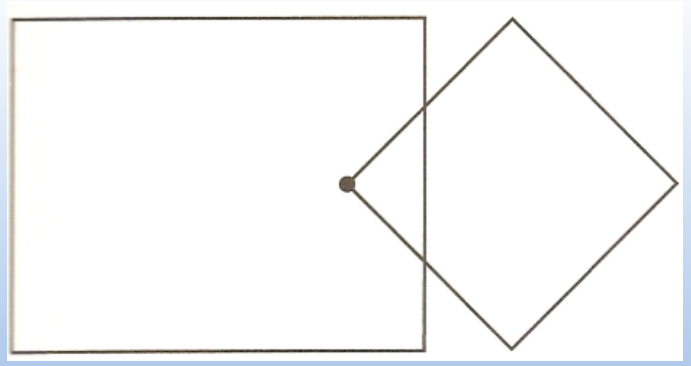


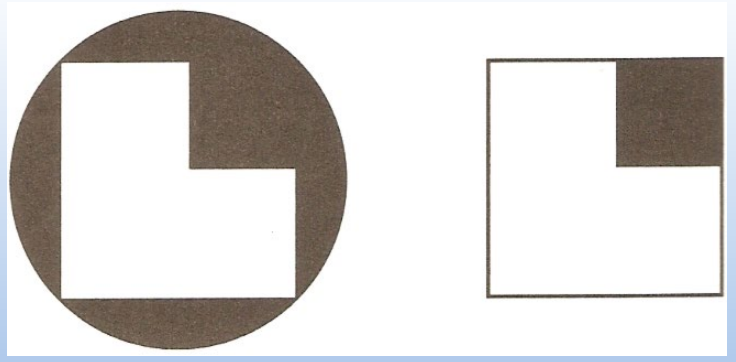
# Bounding Volumes



- Collision check
  - More complex than spheres or cylinders
  - Two bounding boxes are colliding if the vertex of one is inside the other
  - Requires more computation compared to spheres or cylinders
- Unity: `BoxCollider`

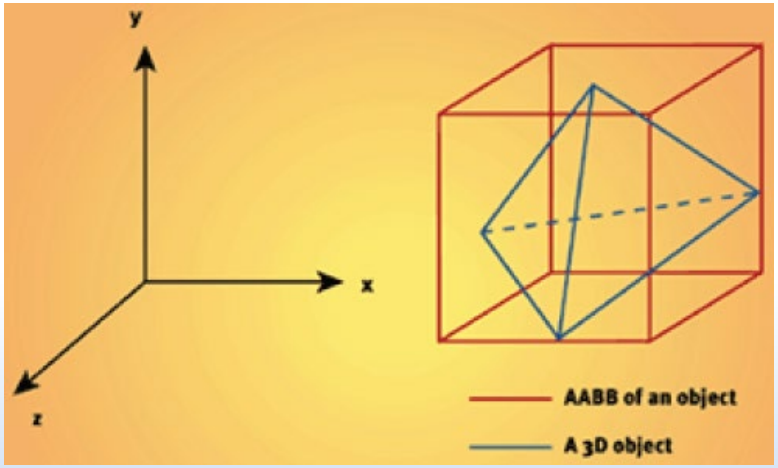




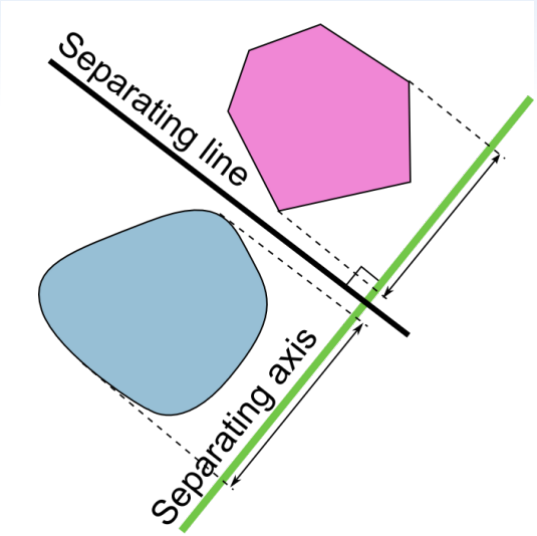
# Bounding Volumes

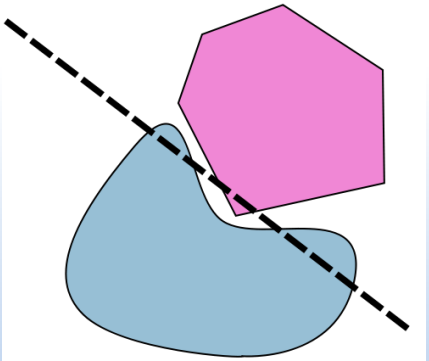
- Two types of bounding boxes in game engines
  - Axis-Aligned Bounding Boxes (AABBs)
  - Orientated Bounding Boxes (OBBs)
- AABBs
  - Aligned to world axes
  - Each box face perpendicular to one coordinate axis
  - AABBs do not rotate
  - Good for static objects that are aligned to axes and do not rotate



# Bounding Volumes

- Separating axis theorem
  - ➢ Two ***convex*** shapes do not intersect
    - If an axis can be found along which the projection of the shapes do not overlap
    - If such an axis does not exist and shapes are convex then they intersect
  - ➢ Used in most collision detection systems
    - Only applies to convex shapes



Not convex

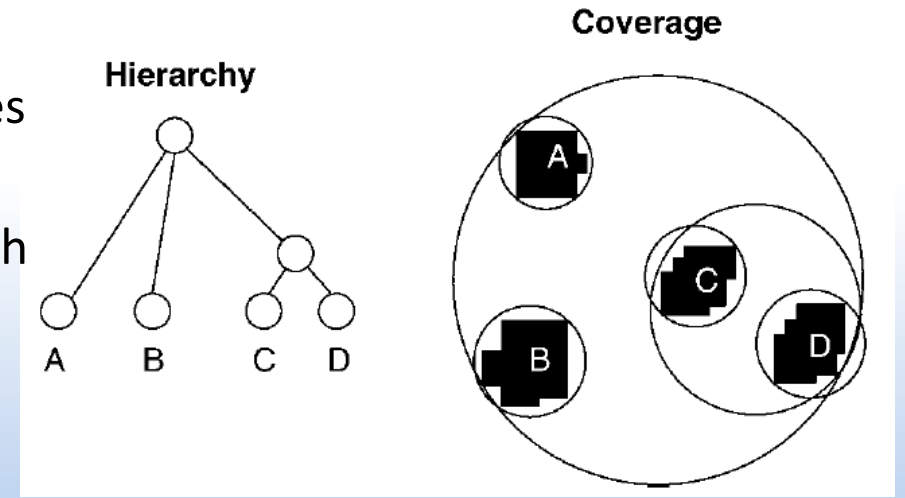
http://en.wikipedia.org/wiki/Separating_axis_theorem

# Bounding Volume Hierarchies

- Bounding volumes alone
  - Still involves checking every pair of objects

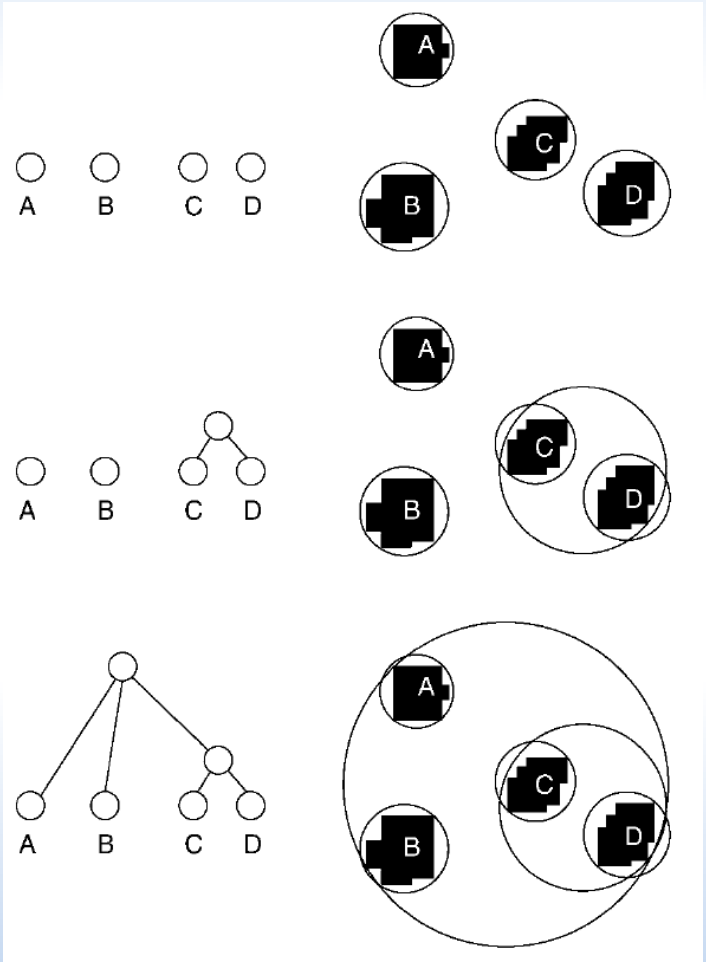
$$N_c = \frac{n!}{2(n-2)!}$$

- Bounding volume hierarchy (BVH)
  - Helps to avoid checking every object pair
  - Each object (in its bounding volume) is a leaf of a tree data structure
    - These connect to parent nodes
    - Each parent has its own bounding volume large enough to enclose all descendents



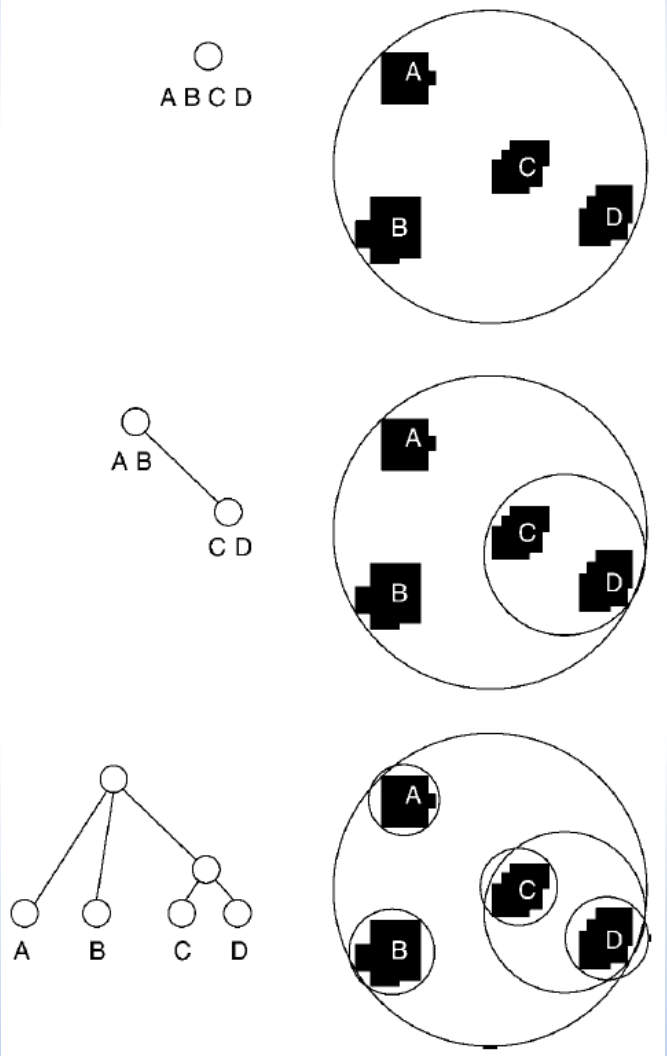
# Bounding Volume Hierarchies

- Building the hierarchy
  - Static environments
    - Hierarchies can be constructed offline
  - Dynamic environments
    - Hierarchies need to be recalculated in game
  - Bottom-up
    - Starts with bounding volumes of individual objects
    - Parents added
    - Continue until only one node left in the list



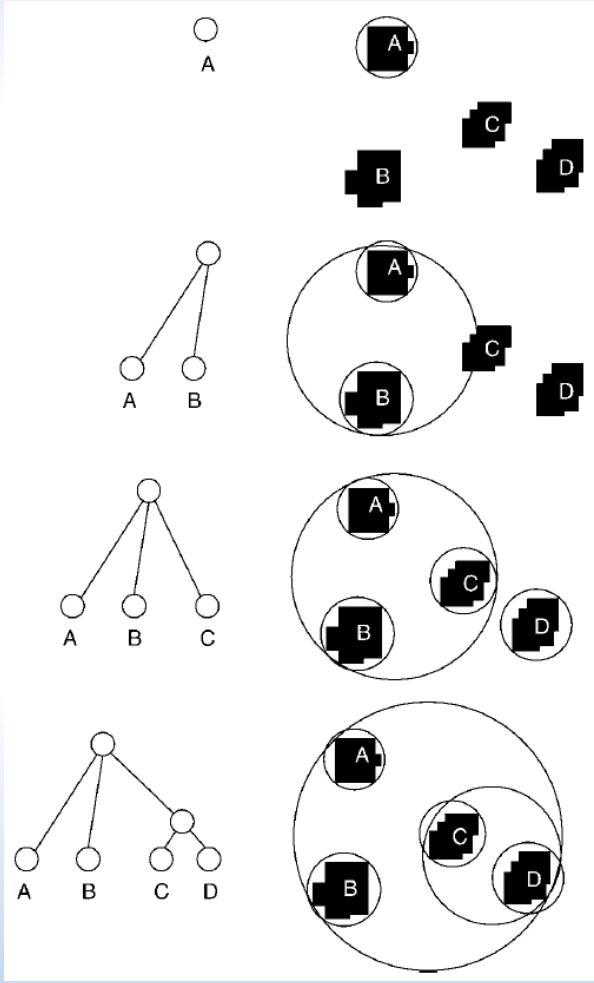
# Bounding Volume Hierarchies

- Top-down
  - Also starts with bounding volumes of individual objects
  - Each iteration, objects in each group separated into two groups
  - Continue until only one object in each group



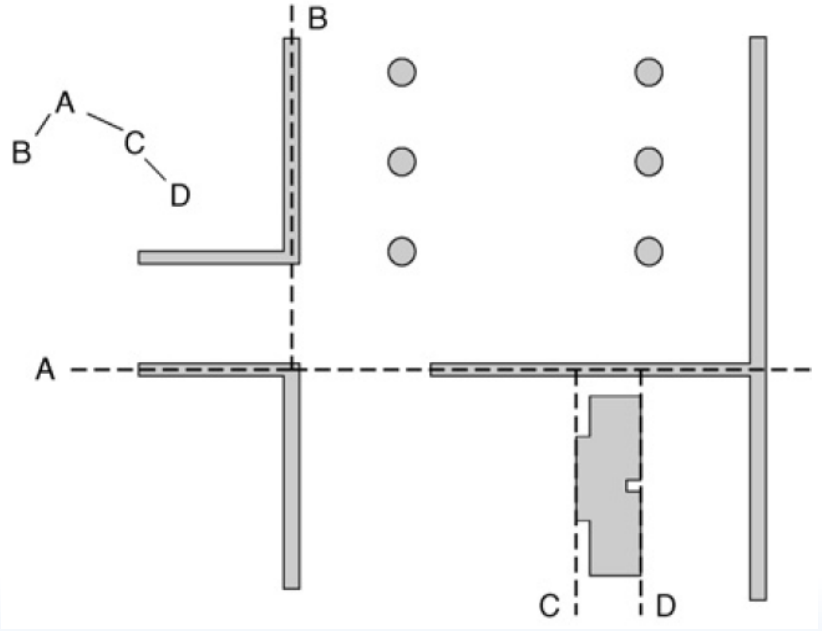
# Bounding Volume Hierarchies

- Insertion
  - Only approach suitable for use during the game
  - Can adjust hierarchy without having to rebuild it completely
  - Starts with existing tree (or empty tree)
  - Objects added/inserted into the tree



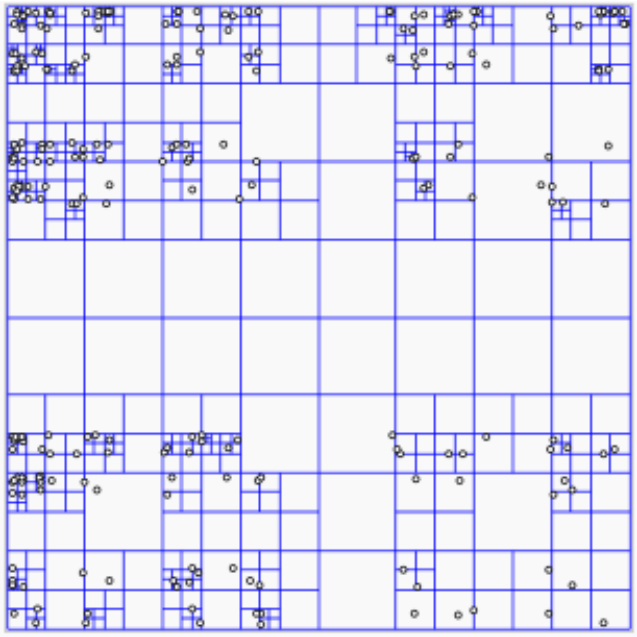
# Spatial Data Structures

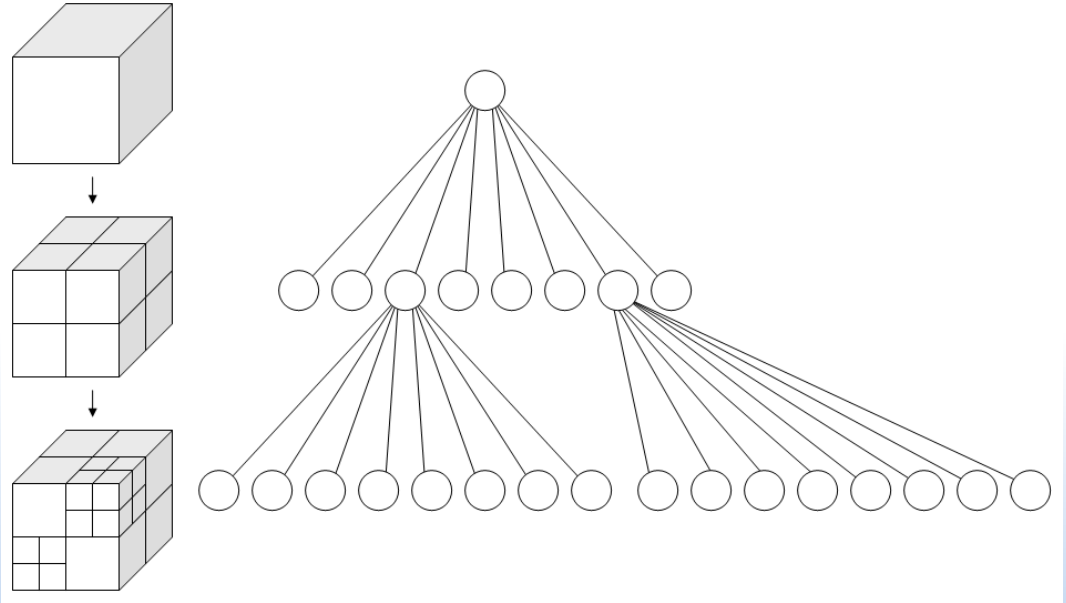
- Binary tree
  - Built by recursively subdividing objects or polygons in a scene using planes
  - Unlike BVH, each node uses a plane to divide space
  - Each parent only has a maximum of two children
- Two phases
  - Building the BSP tree
    - Can be done offline
  - Traversing the BSP tree
    - At run-time



# Spatial Data Structures

- Particularly useful for
  - Outdoor scenes in games where most objects are on the ground
  - Less for indoor games
    - Cannot easily use to detect collisions with walls



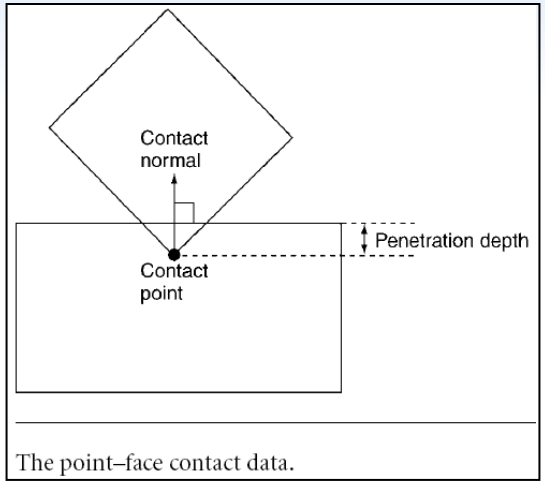


http://en.wikipedia.org/wiki/Quadtree

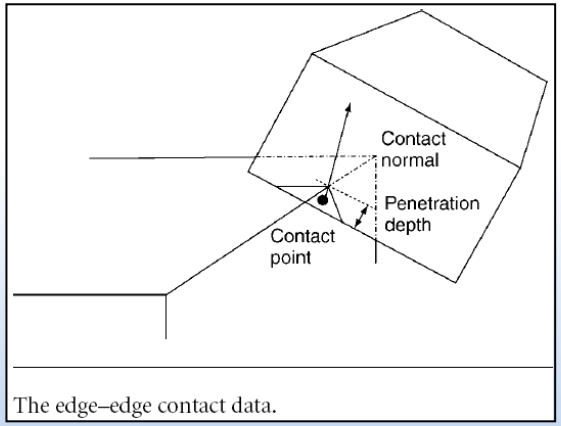
http://en.wikipedia.org/wiki/Octree

# Fine Collision Detection

- Contact data
  - Collision point
    - Point of contact between two objects
    - Objects will be interpenetrating somewhat, may be a number of possible points
  - Collision normal
    - Direction in which impact impulse will be felt between two objects
  - Penetration depth
    - The amount that two objects are interpenetrating
    - Measured along the direction of the collision normal passing through the collision point



The point–face contact data.



The edge–edge contact data.

# Fine Collision Detection

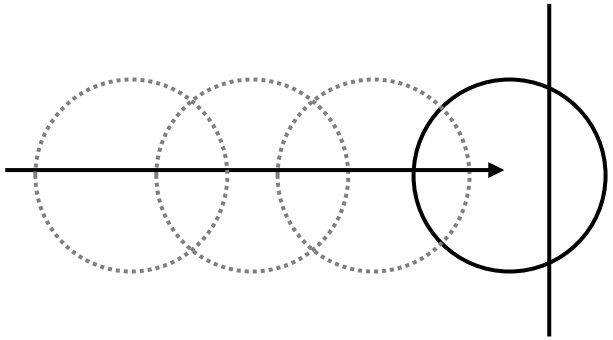
- Unity
  - `Collision`
    - Describes a collision and passed to the following event methods:
      - `OnCollisionEnter()`
      - `OnCollisionStay()`
      - `OnCollisionExit()`
    - Some properties
      - `contactCount`: number of contacts for this collision
      - `relativeVelocity`: the relative linear velocity of the two colliding objects
      - `gameObject`: the GameObject whose collider you are colliding with
    - Public method
      - `GetContacts()`: retrieves all contact points for this collision

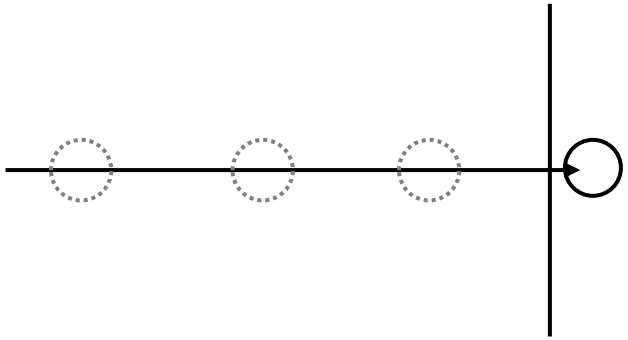
# Fine Collision Detection

- Unity
  - `ContactPoint`
    - Describes a contact point where the collision occurs
    - Properties
      - `impulse`: impulse applied for collision resolution
      - `normal`: normal of the contact point
      - `point`: point of contact
      - `separation`: distance between colliders at the point of contact
      - `thisCollider` : the first collider in contact at the point
      - `otherCollider` : the other collider

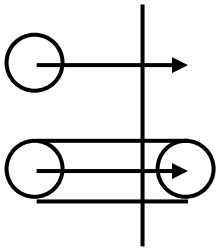
# Collision Detection

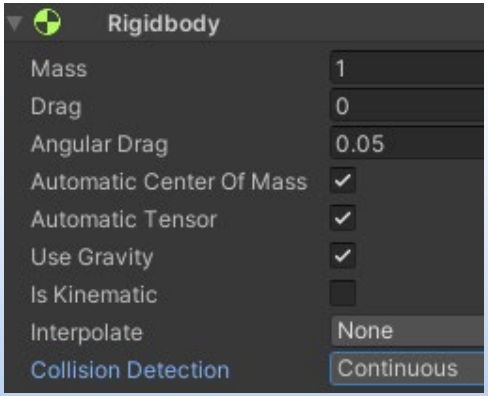
- Discrete collision detection
  - ‘Tunneling’ problem
    - Potential problem for small objects moving at high speeds





  - Solution
    - Ray casting
    - Sweep shapes
    - Continuous collision detection
      - Predict time of impact along current path





# Collision Queries

- Ray cast
  - ➢ Cast a directed line segment
    - From a starting point to an endpoint
      - Some game engines do not support infinite rays
    - Line segment tested against collidable objects
      - Returns contact point(s)
      - Typically want the closest intersection point
      - Spatial data structures speed up collision tests
  - ➢ Example applications
    - Weapons, direct line of sight, movement queries, distance to ground

# Collision Queries

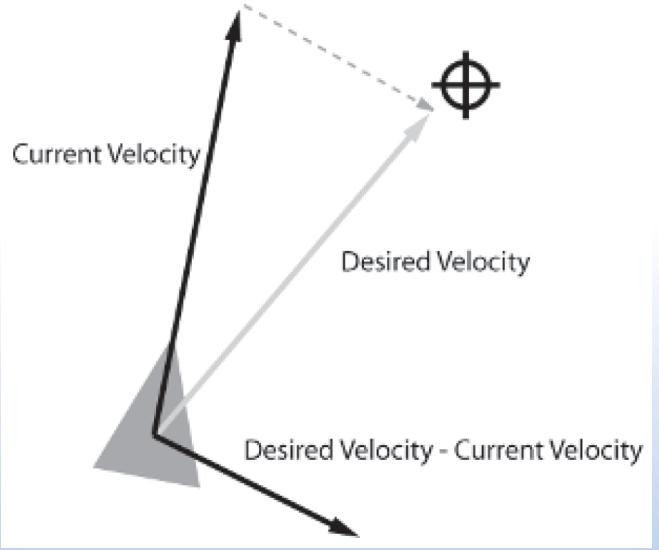
- Shape cast
  - Casting a shape along a directed line segment
    - Shape usually a sphere
  - Example applications
    - Sliding a character forward on uneven terrain
    - Determining whether an object can move between obstacles
    - Virtual camera collision
- Volume queries
  - Determine which collidable objects lie within some specific (invisible) volume
    - Like a zero distance shape cast, but unlike casts can be persistent and takes advantage of temporal coherence

# Movement

- Movement
  - Can be broken down into
    - Action selection
      - The part of the agent's behaviour responsible for choosing its goals and deciding what plan to follow
    - Steering
      - This layer is responsible for calculating the desired trajectories required to satisfy the goals and plans set by the action selection layer
    - Locomotion
      - Represents the more mechanical aspects of an agent's movement

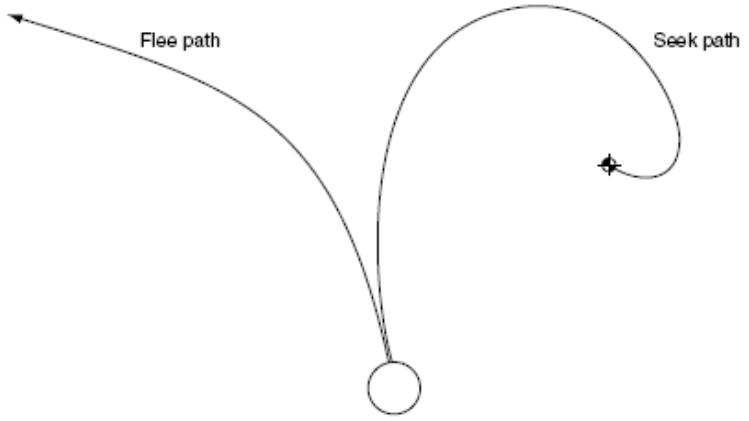
# Steering Behaviours

- Steering behaviours
  - ➢ Commonly implemented in games
    - Simple components, quick to implement and compute
    - Simple to understand and implement but can produce seemly complex movement patterns
  - ➢ Seek
    - Tries to match the position of the agent with the target's position
    - Finds the direction to the target and heads toward it



# Steering Behaviours

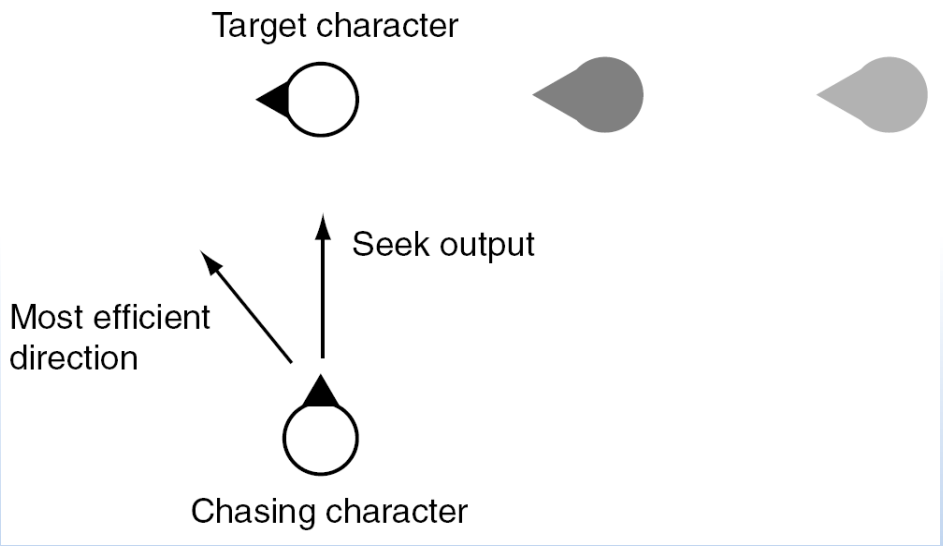
- Flee
  - Opposite of seek
  - Instead of steering agent toward target position
    - Reverse direction and make agent run away
  - Might only want to flee when the target is within a certain range

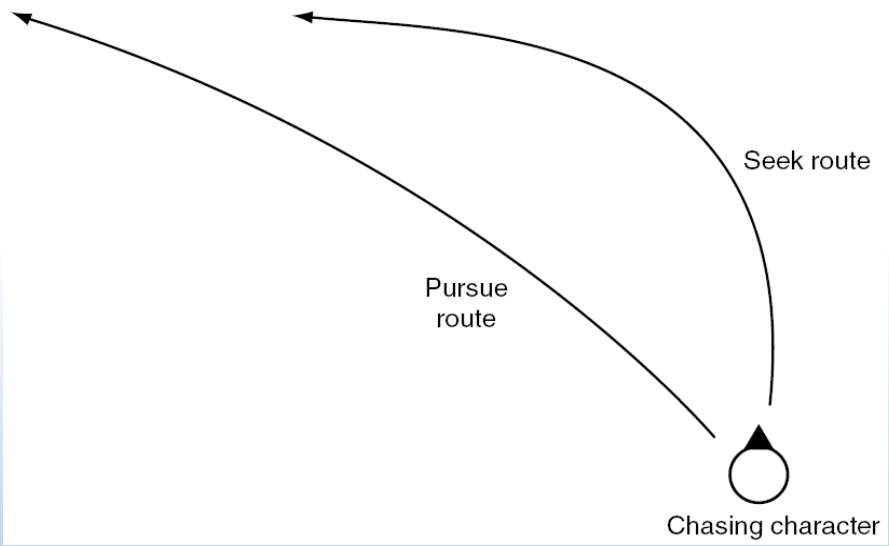



- Arrive
  - Seek is fine if the agent is chasing the target (constantly moving)
    - Doesn't stop gracefully, likely to overshoot an exact spot
    - May oscillate backward and forward on successive frames trying to get there

# Steering Behaviours

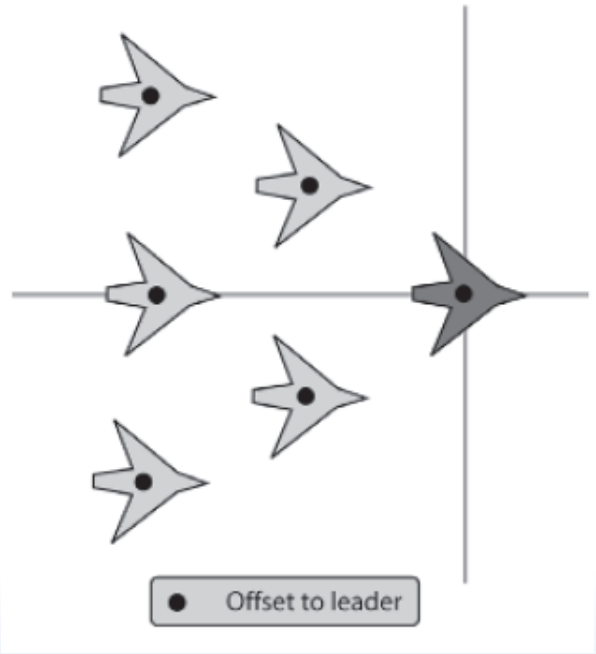
- Pursuit
  - Useful when agent required to intercept moving target
  - Seek only tracks current target position
  - Pursuit predicts where the target will be in future and aims there



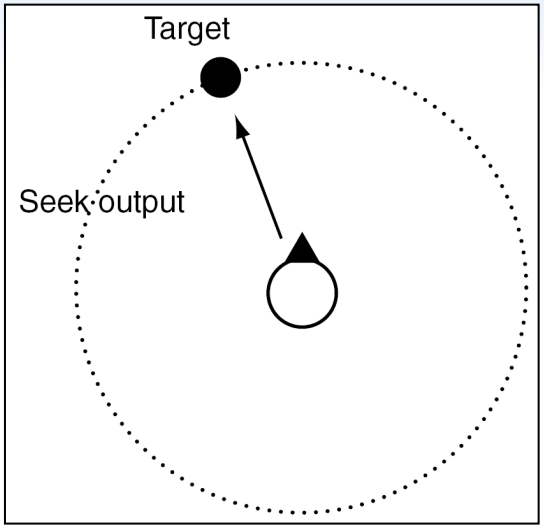
# Steering Behaviours

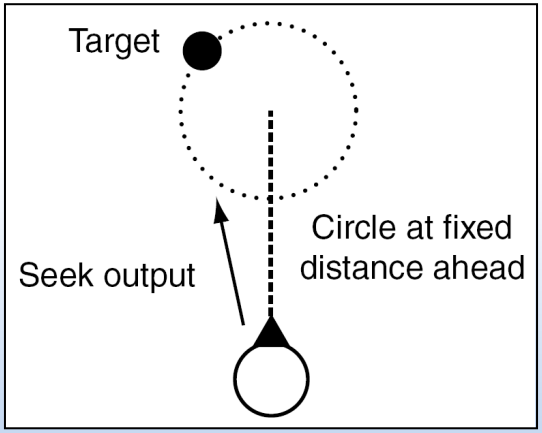
- Evade
  - Opposite behaviour to pursuit
  - This time evader flees from the estimated pursuer's future position
- Offset pursuit
  - Keep the agent positioned at a specified offset from the target
  - Useful for creating formations
  - Use in conjunction with the arrive behaviour instead of seek



# Steering Behaviours

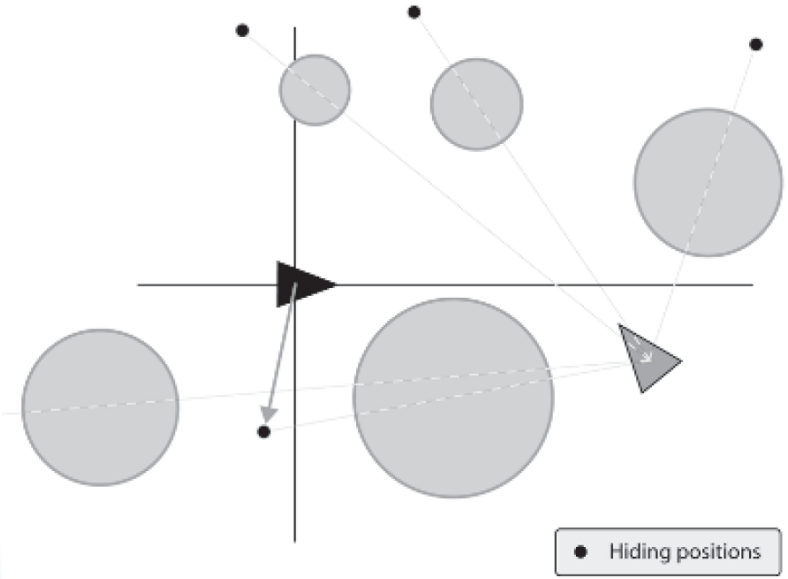
- Wander
  - Steering behaviour that gives the impression of random walk
  - Naïve approach
    - Random direction each time step
    - Produces jittery behaviour, no ability to achieve long persistent turns
  - Better approach
    - Move the circle out in front of the agent and shrink it down
    - Orientation changes a lot smoother



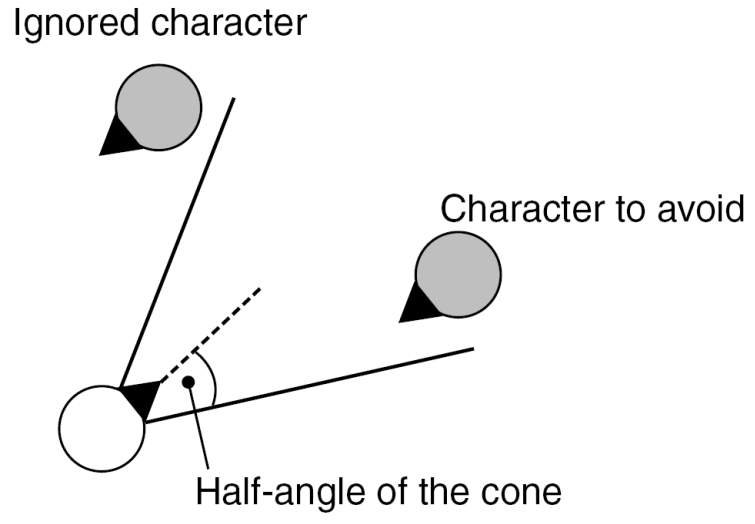
# Steering Behaviours

- First, find a hiding spot for each obstacle
- Then, determine the distance to each spot
  - Use arrive to steer to the closest
  - If no appropriate hiding place is found
    - » Use evade
- Might only want to hide if within the agent's field of view



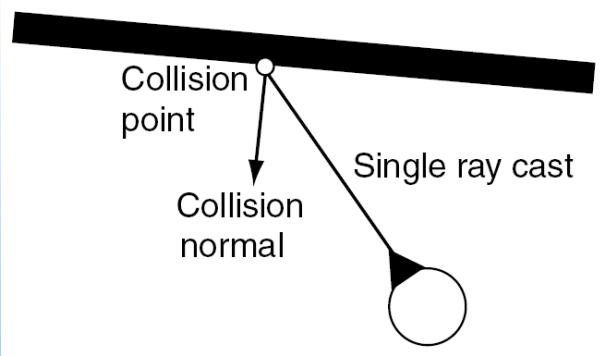
# Collision Avoidance

- Collision avoidance
  - Avoid other agents
    - Both moving and stationary agents
  - Simple approach
    - Use a variation of evade or separation behaviour, that only engages if target within a cone



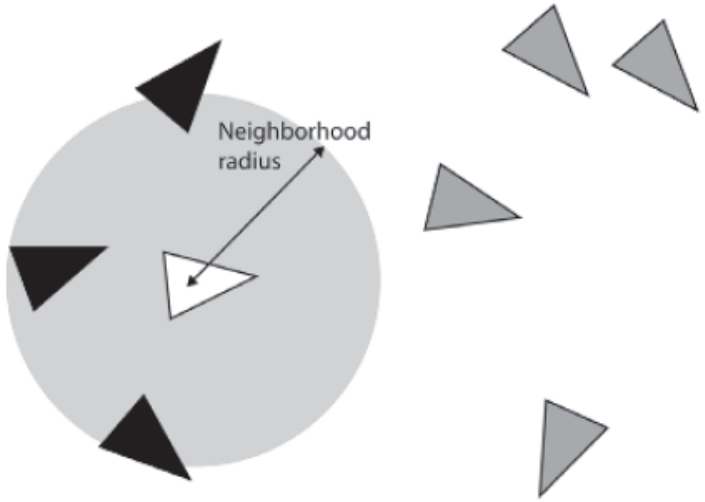
# Collision Avoidance

- Obstacle and wall avoidance
  - The previous approach
    - Only works for agents or objects that can comfortably fit within a bounding sphere
  - To avoid large obstacles
    - Agent casts one or more rays out in the direction of motion
    - Short rays can be used
    - If ray collides with obstacle, then a target is created that has to be avoided



# Combining Steering Behaviours

- Group behaviour
  - ➢ Steering behaviour that
    - Takes into consideration some, or all, other agents in the game world
    - Agent will consider all other agents within a circular area
  - ➢ Flocking and swarming
    - The flocking algorithm relies on blending three steering behaviours
      - Separation
      - Alignment (and velocity matching)
      - Cohesion

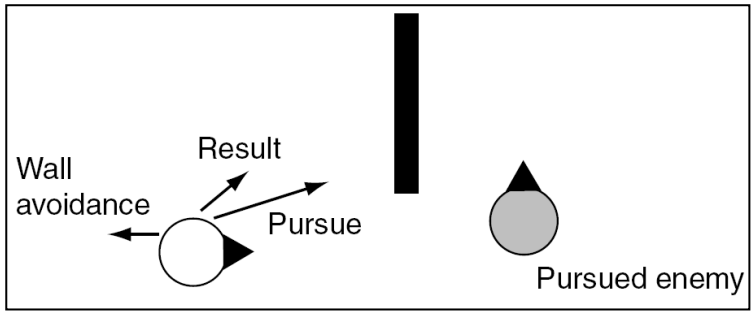


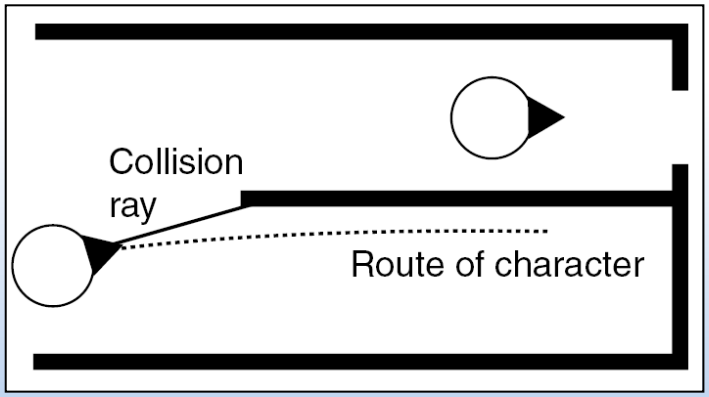
# Combining Steering Behaviours

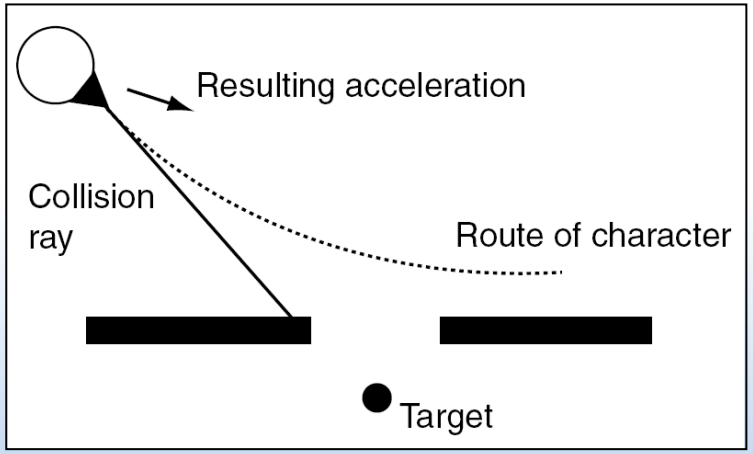
- Blended steering behaviour
  - Can achieve more complex movement
  - Combine steering behaviours using a set of weights and/or priorities
    - Weighted truncated sum
      - Combine each steering behaviour by a weighted sum, then truncate by maximum allowable velocity or acceleration
    - Priorities
      - Some steering behaviours more important than others
    - Weights or priorities may change over time, or may even be switched off

# Combining Steering Behaviours

- Other problems
  - Constrained environments
    - Movement in indoor environments greatly constrains an agent's movements
  - Nearsightedness
    - Steering behaviors act locally based on immediate surroundings

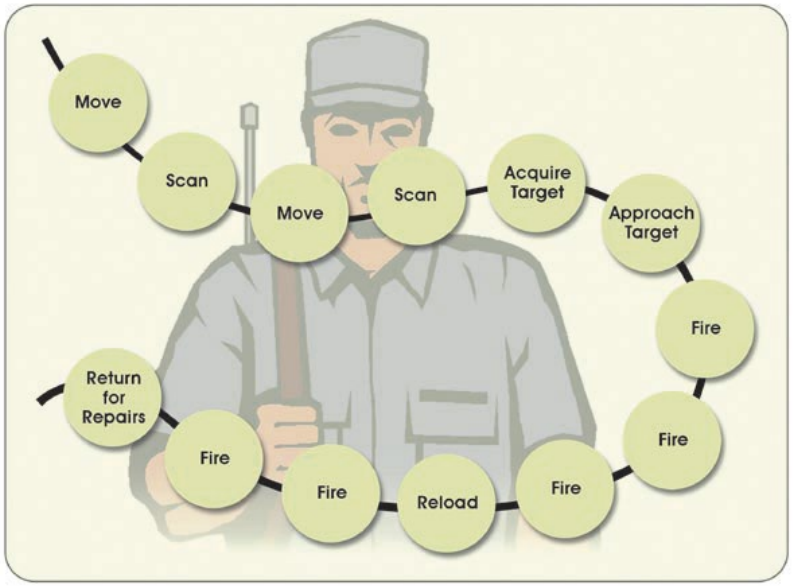


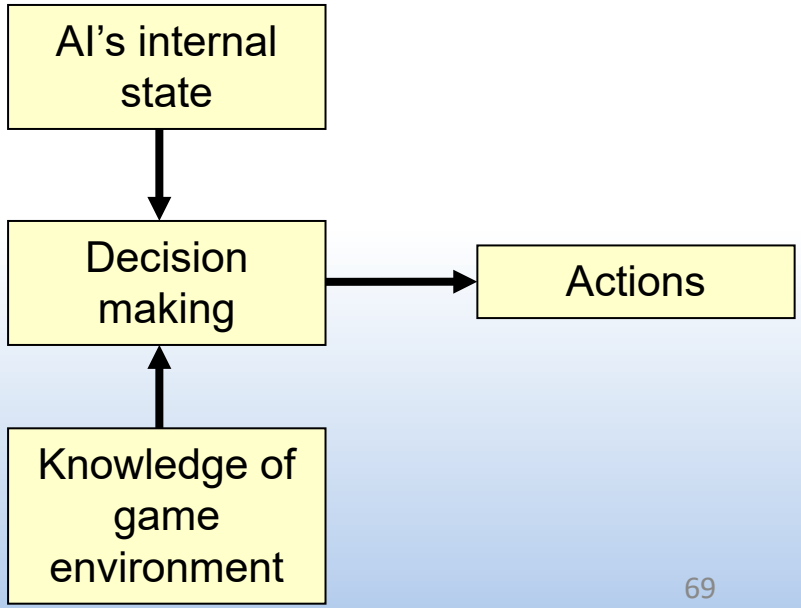



# Controlling Behaviour

- Behaviour
  - A context specific sequence of actions
  - Dynamic sequences
    - Actions in the sequence should vary depending on circumstances (internal state and relevant game environment)



# Controlling Behaviour

- Finite state machines
  - Most common behavioural modelling algorithm
    - Conceptually simple but can create flexible AI with little overhead
  - Consists of
    - A number of states
      - Normally, actions or behaviours associated with each state
      - Only one state is “active”
      - Connected together by transitions
    - Transitions between those states
      - Each transition leads from one state to another
      - Each has a set of associated conditions, if condition met, new state becomes active

# State Machines

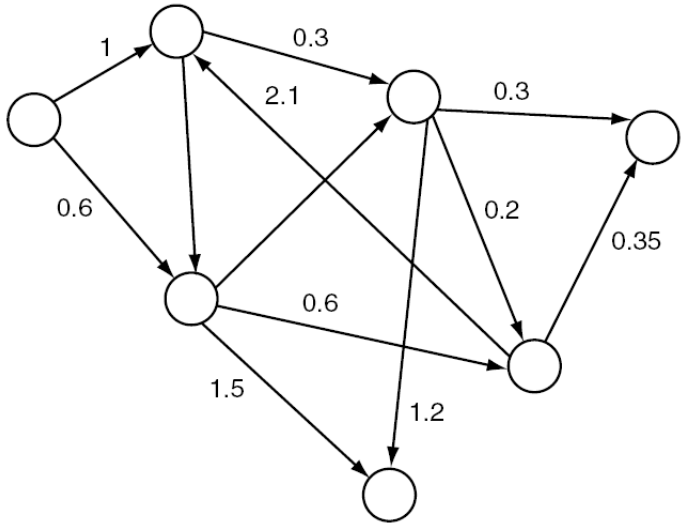
- Hierarchical state machines
  - Complex games
    - Have over 100 states
    - Complex behaviours might be implemented across a number of states
    - Difficult to manage and error-prone
  - Hierarchical state machines
    - Useful when dealing with a large number of states
    - Define top level states
      - Then move into sub-states to elaborate details
    - Easier to understand
      - Less states at top level, more efficient
      - Potentially result in faster game performance

# State Machines

- Advantage of state machines
  - Only one state active at a time
  - Limited to a single action
    - Prevents going to two locations at once or firing a weapon at two different targets simultaneously
- Problem
  - How to handle
    - Interrupted goals
    - More than one goal at a time

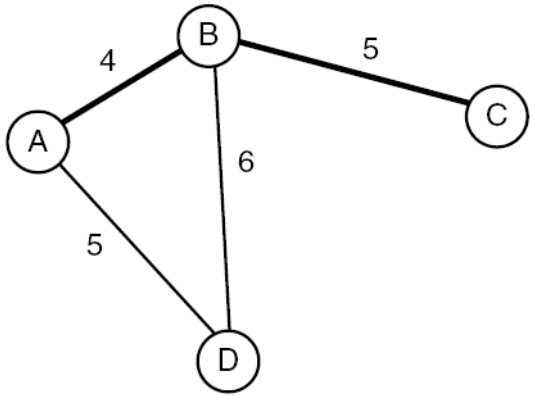
# The Pathfinding Graph

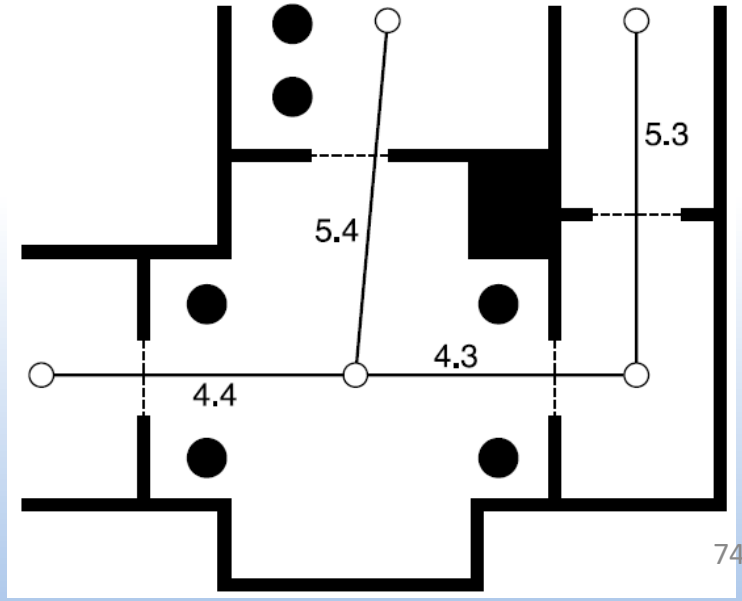
- Directed weighted graphs
  - Connections and costs in one direction only
    - If can return, considered as two connections
  - Connections now ordered
    - E.g., from node 1 to node 2



# The Pathfinding Graph

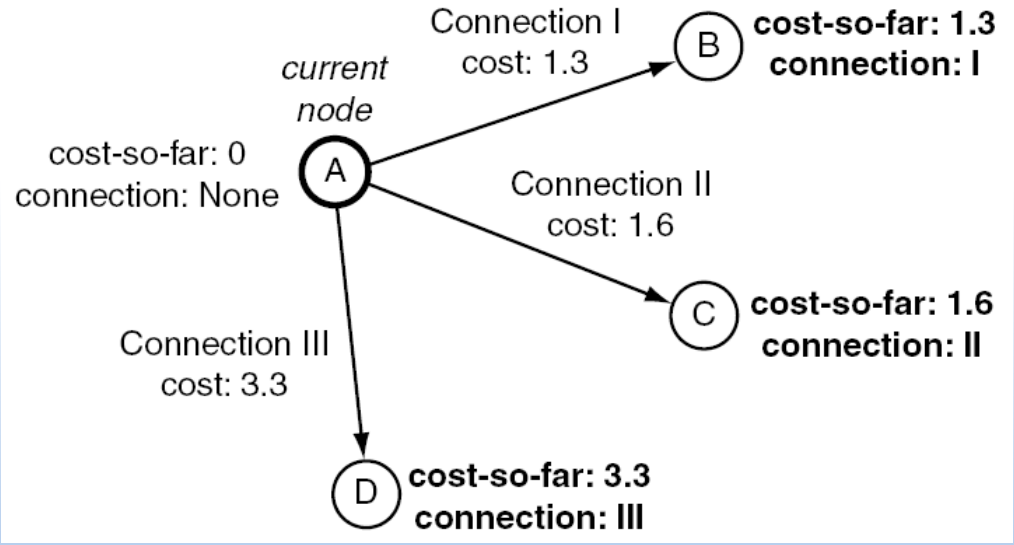
- Total path cost
  - Sum connection costs
- Nodes are representative points
  - Can be centre of a region
  - Or some other position



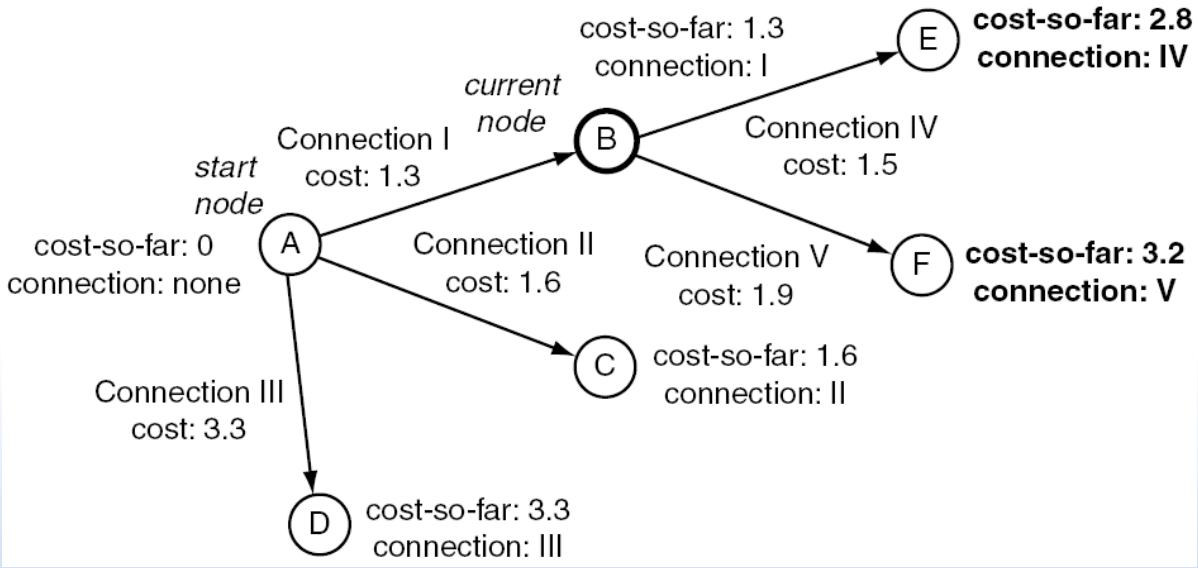
# Dijkstra

- Processing the current node
  - Trace all the outgoing connections from the current node
  - For each connection, find the end node and store
    1. Total cost of path thus far (‘**cost-so-far**’), and
    2. The **connection** it arrived there from
  - Need to record the cost-so-far and connection for each node



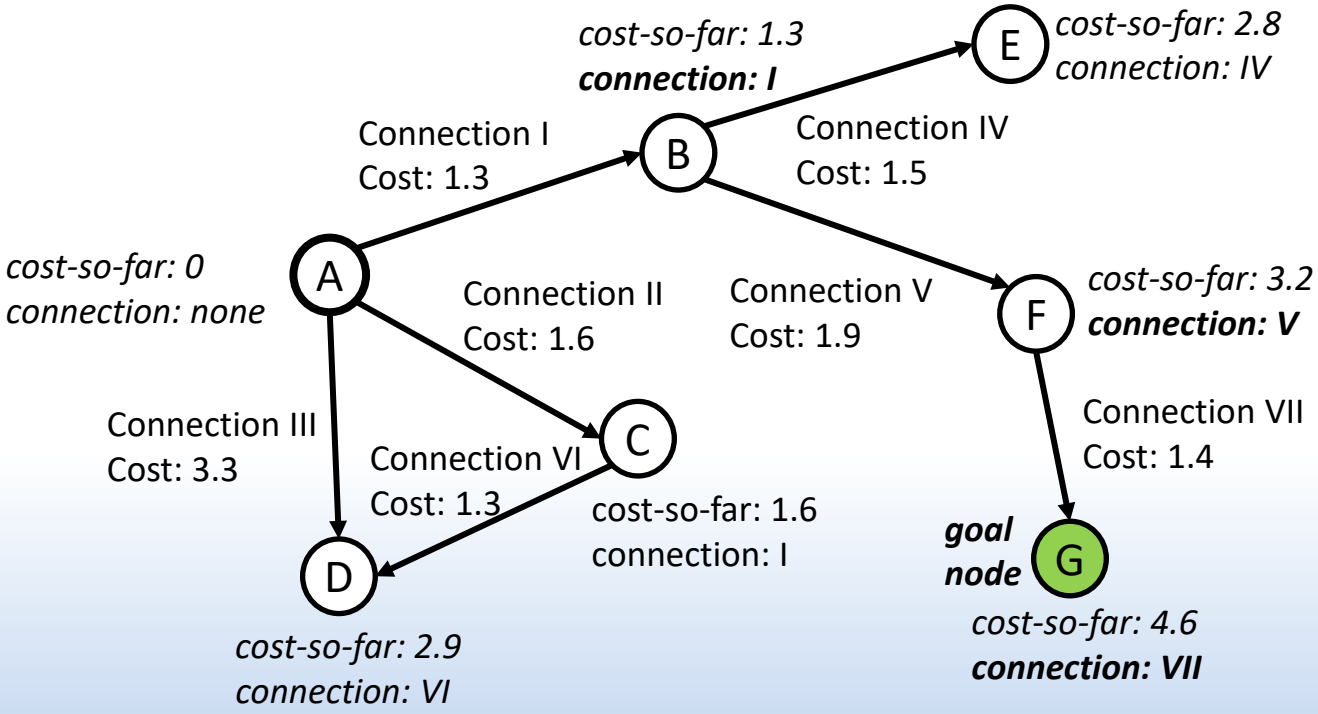
# Dijkstra

- For iterations after the first
  - Cost-so-far for the end node of each connection, is the
  - Sum of **connection cost + cost-so-far** of the current node



# Dijkstra

- Retrieving the path
  - Start at the goal node, work backward through the connections until the start node, then reverse the order




Connections working back from the goal: **VII, V, I**

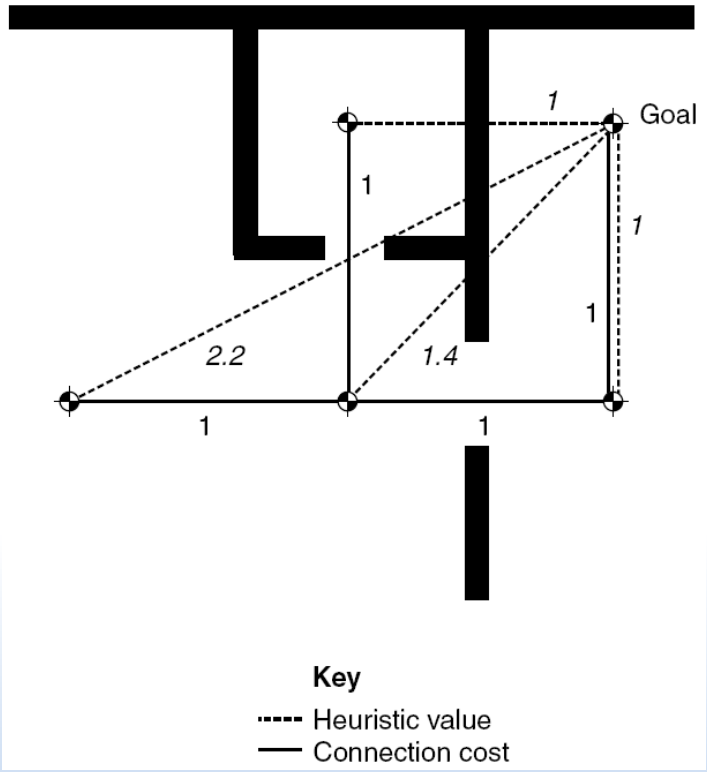
Final path: **I, V, VII**

# A* Pathfinding

- The algorithm
  - Like Dijkstra, works in iterations
  - Processing current node
    - Difference from Dijkstra? Store another value called the "**estimated-total-cost**"
      - Estimate of total cost for a path from the start node, through the current node, onto the goal
      - Estimated-total-cost = **cost-so-far + heuristic**
      - The heuristic is an estimate of the cost from that node to the goal
        - » Cannot be negative
      - This estimate is not part of the algorithm and is generated by a separate piece of code

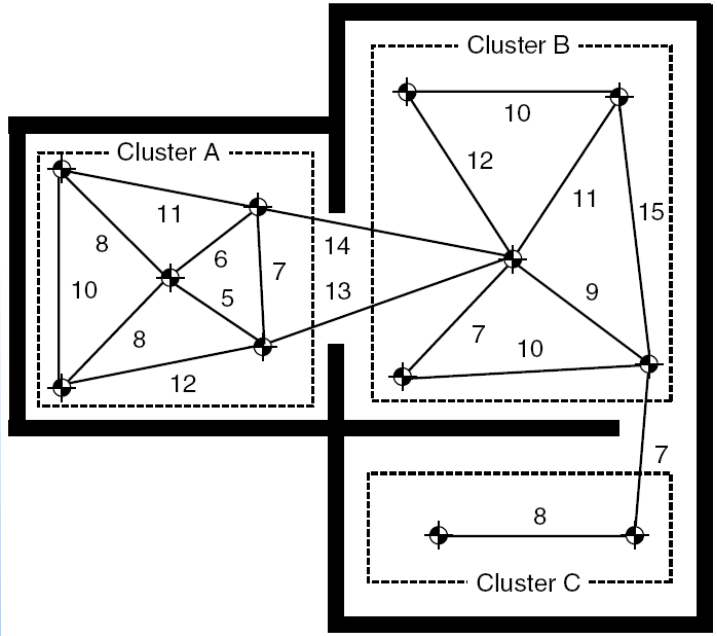
# A* Pathfinding

- Common heuristics in games
  - ➢ Euclidean distance
    - If connection cost represents distance
    - Distance “as the crow flies”, measured between two points through obstructions
    - Either accurate or guaranteed to underestimate
    - In outdoor settings with few constraints, can be very fast and accurate



# A* Pathfinding

- Cluster heuristic
  - Group nodes together in clusters
  - Nodes in clusters represent highly interconnected region
  - Often done manually
  - Lookup table gives smallest path length

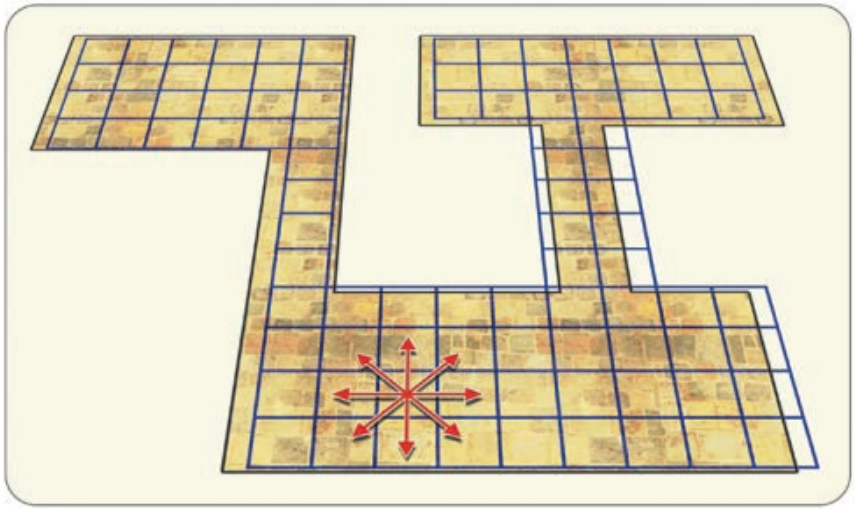


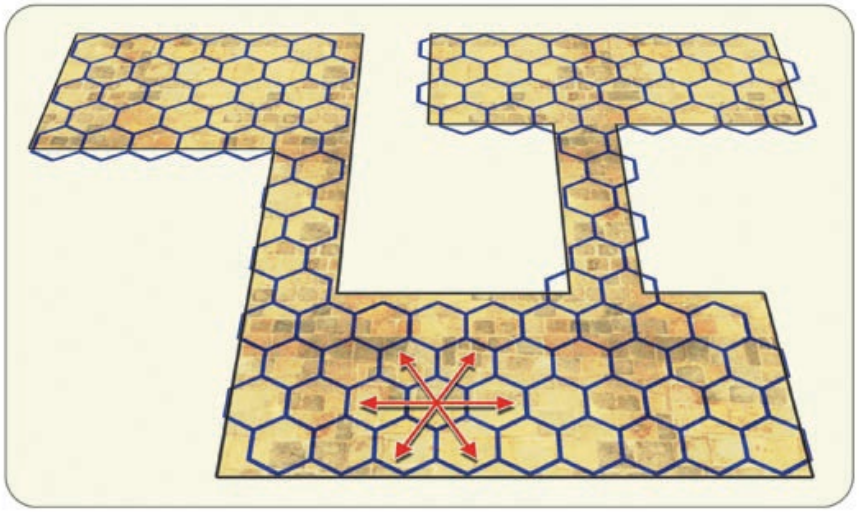

| | A | B | C |
|---|---|---|---|
| A | x | 13 | 29 |
| B | 13 | x | 7 |
| C | 29 | 7 | x |

Lookup table

# Pathfinding Networks

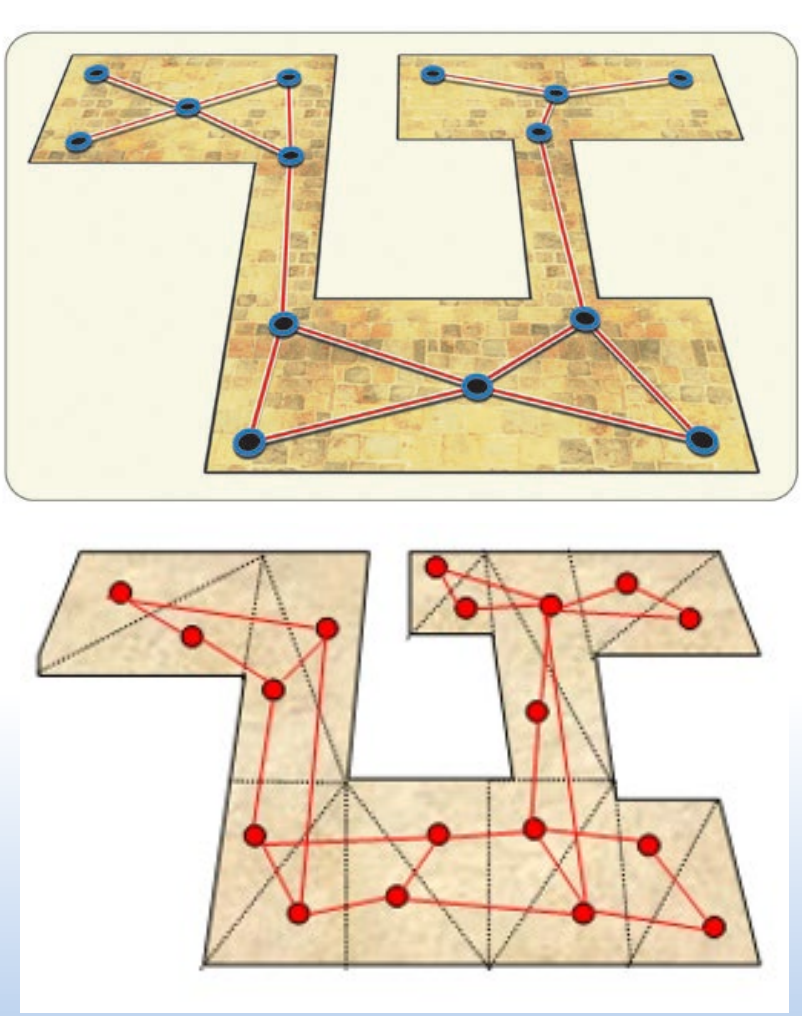
- Rectangular grids (also known as tiles)
  - Can be square or hexagonal
  - Can be used to automatically generate path-finding networks
  - Easy to generate algorithmically and determine connectivity
  - Easy to implement
    - Can use a simple 2D array





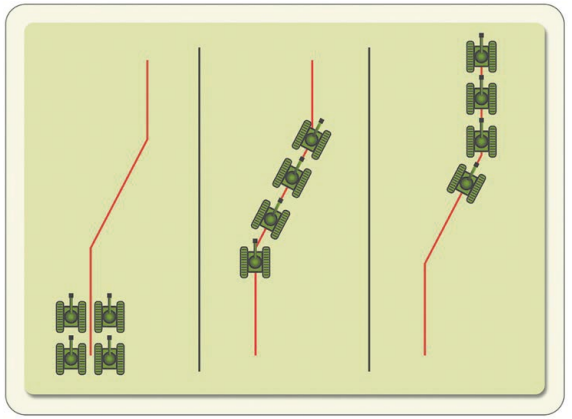
# Pathfinding Networks

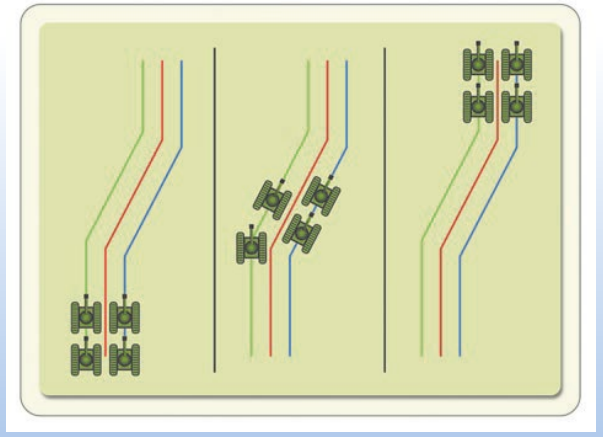
- Arbitrary grids
  - Nodes only placed at important locations
  - Usually done manually
  - Advantages
    - Nodes can be placed exactly along the desired movement for path following
    - Needs to store less data
  - Disadvantage
    - Manual work required
- Navigation mesh
  - Divides the environment into a mesh, assign nodes to the mesh



# Advance Techniques

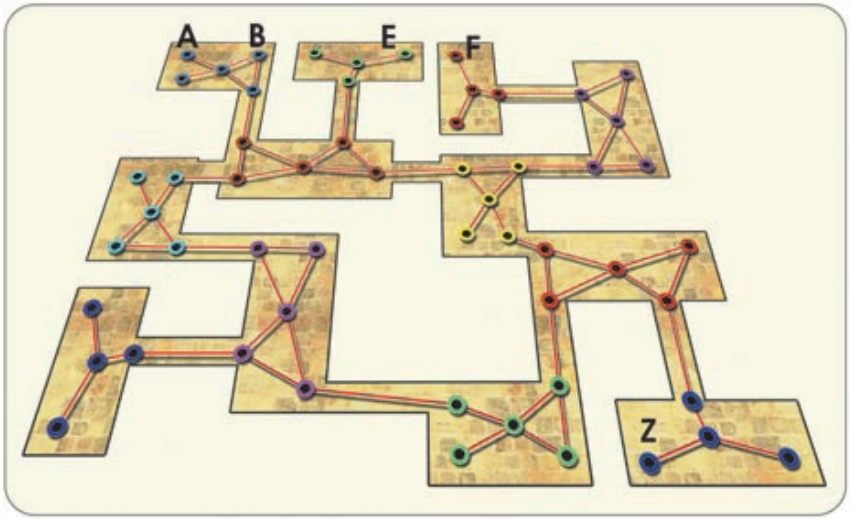
- Problem
  - Units will all line up in a row (called ‘antlining’)
    - Takes long time for last unit to arrive
    - If enemy near destination, group easily destroyed
  - To address antlining
    - Offset the individual unit paths
    - Formation much tighter while moving and at destination



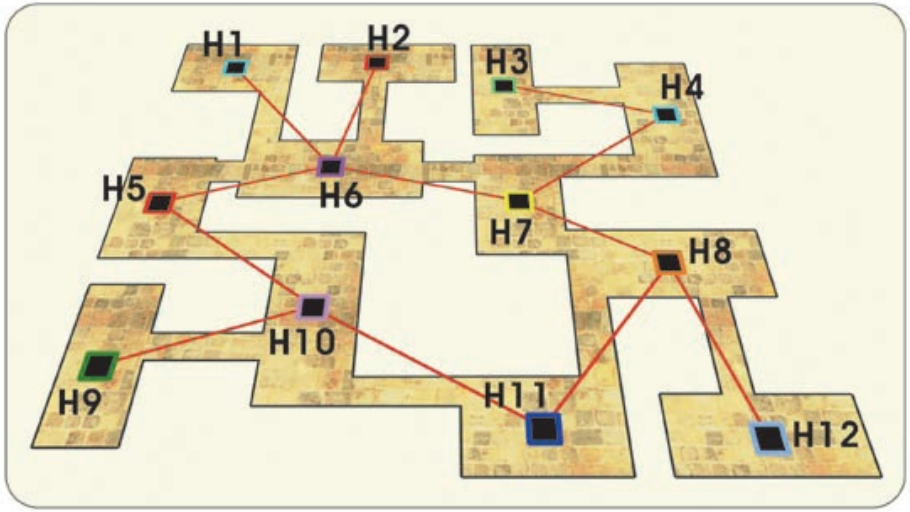


# Advance Techniques

- Hierarchical path-finding
  - Plan overview of route first then refine as needed




Original network contains 56 nodes and high level of detail



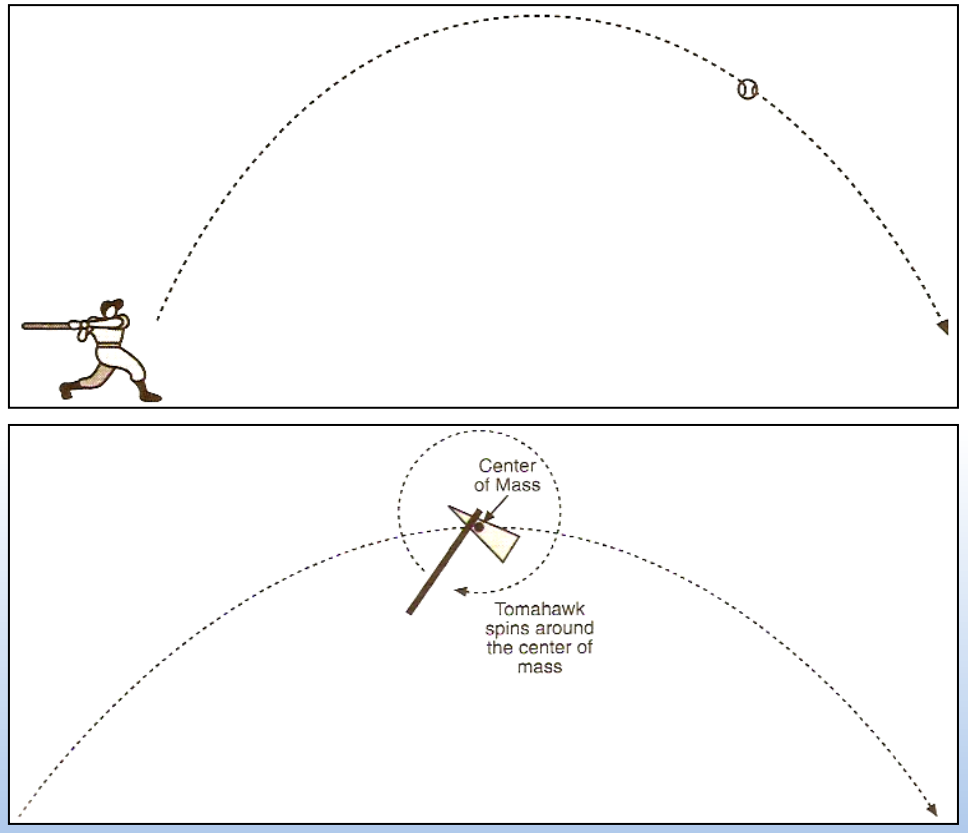

Hierarchical network contains less detailed representation and fewer nodes (12 nodes)

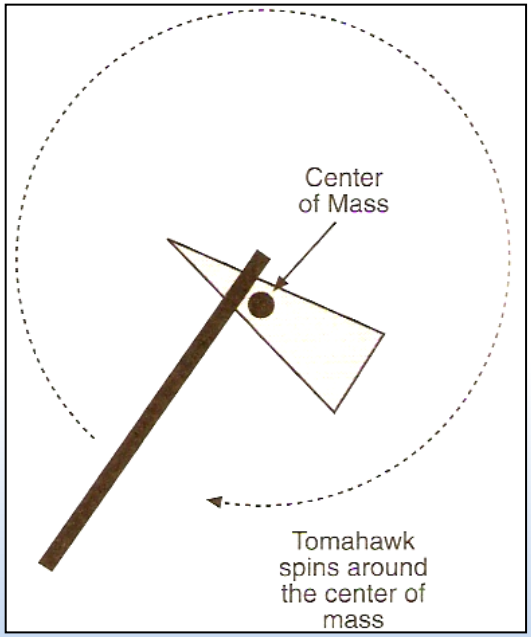
# Rigid Bodies

- Centre of mass
  - A specific point at which the system's mass behaves as if it were concentrated
    - Splitting the object in two through this point produces two objects with the same mass
    - Can balance the object at this point
  - Often also called the “centre of gravity”
  - Not always the geometric centre
  - Fixed position in relation to object (for rigid bodies)
    - Not necessarily in contact with the object

# Rigid Bodies

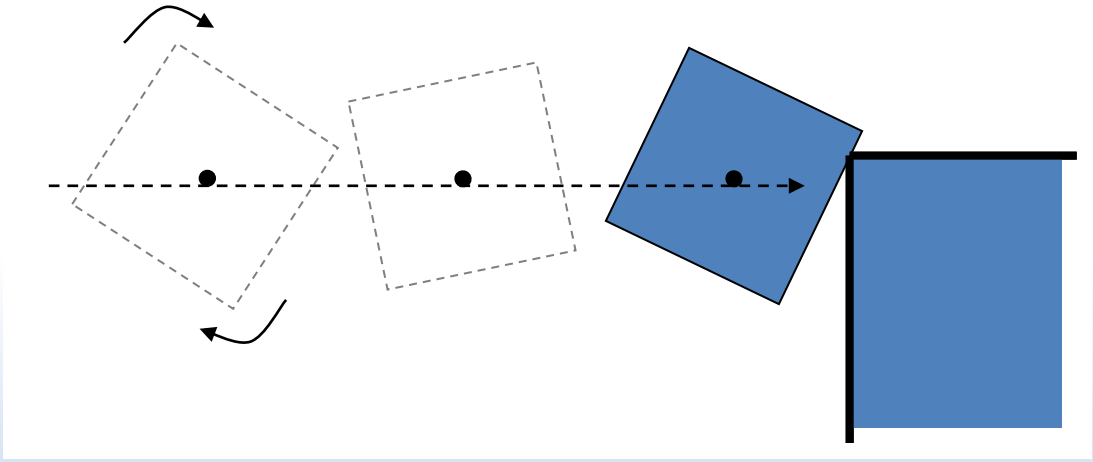
- The centre of mass behaves like a particle
  - Same linear motion formulae as for particles
  - Allows separation of linear motion and angular motion calculations



# Rigidbody Dynamics

- Force
  - Need to be able to apply force at a distance away from the centre of mass
    - E.g. at a point on the surface of the rigid body

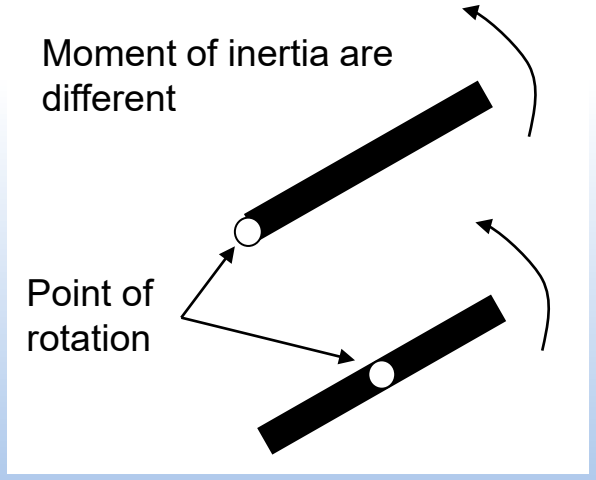


# Rigidbody Dynamics

- Moment of inertia
  - Similar role in rotational dynamics as mass in linear dynamics
  - Used to determine the relationship between
    - Angular momentum and angular velocity
    - Torque and angular acceleration
  - It depends on the mass distribution and the axis of rotation

$$I = \sum_{i=1}^{n} m_i r_i^2$$

$$\tau = I\alpha$$



Thanks for attending