# INFT3960 – Game Production

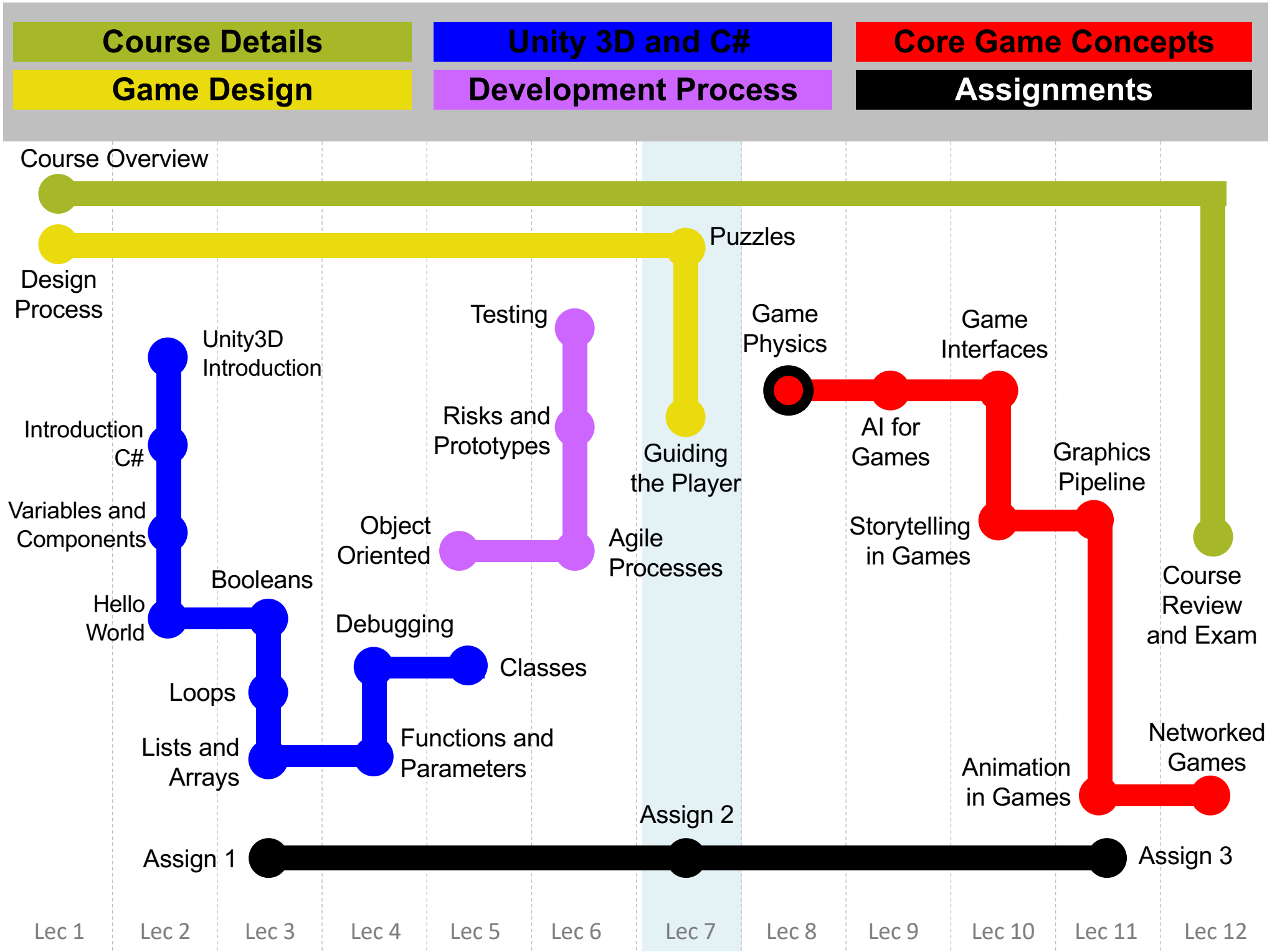
**Week 08**

**Module 8.1**

**Game Physics**

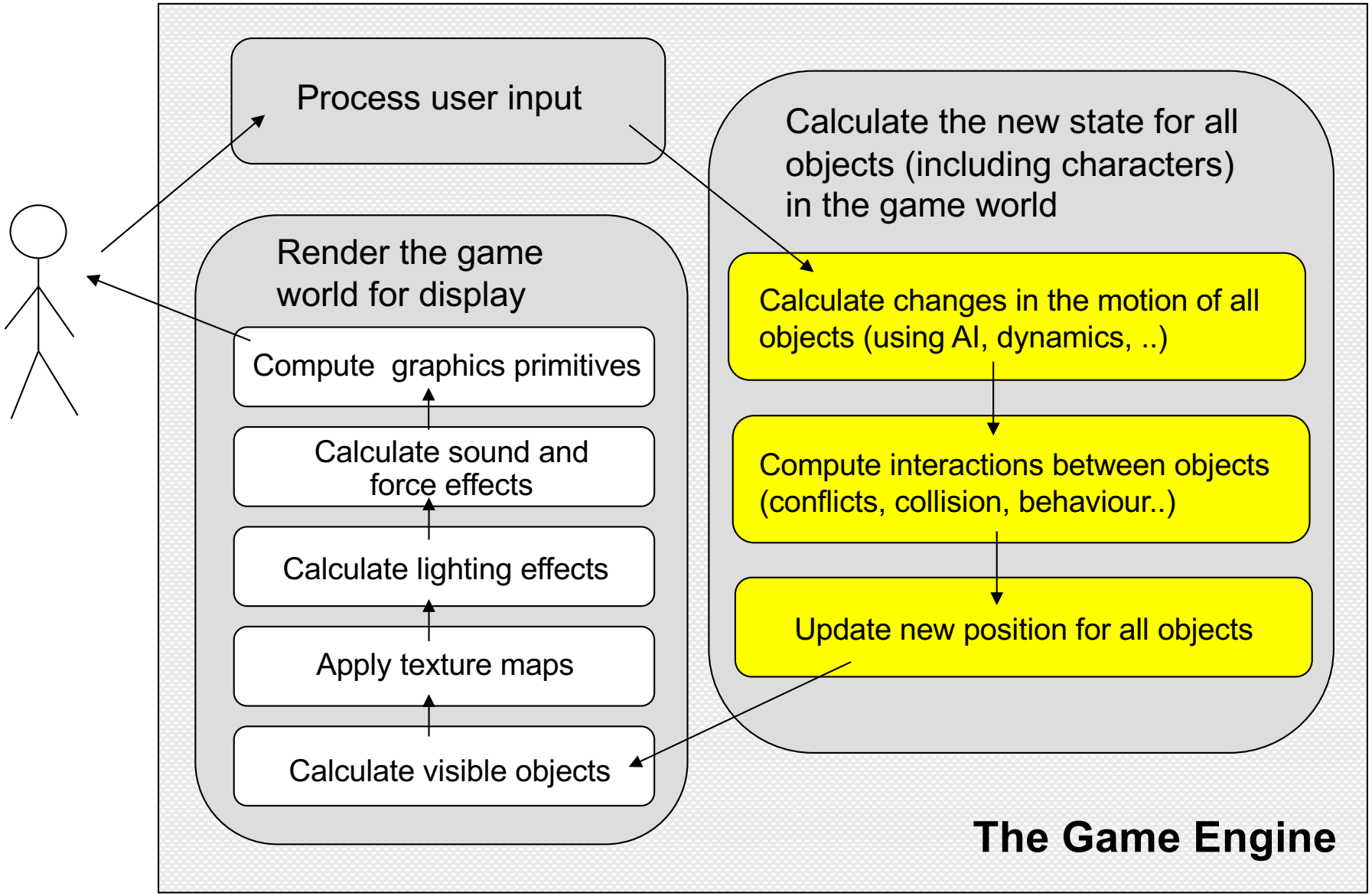
# Course Overview

| Lec | Start Week | Modules | Topics | Assignments |
|---|---|---|---|---|
| 1 | 3 Aug | Mod 1.1, 1.2 | Course Overview, Design Process | |
| 2 | 10 Aug | Mod 2.1, 2.2, 2.3, 2.4 | Unity3D Introduction, Introduction C#, Variables and Components, Hello World | |
| 3 | 17 Aug | Mod 3.1, 3.2, 3.3 | Booleans, Loops, Lists and Arrays | Assign 1<br>21 Aug, 11:00 pm |
| 4 | 24 Aug | Mod 4.1, 4.2 | Functions and Parameters, Debugging | |
| 5 | 31 Aug | Mod 5.1, 5.2 | Classes, Object Oriented | |
| 6 | 7 Sep | Mod 6.1, 6.2, 6.3 | Agile Processes, Risks and Prototypes, Testing | |
| 7 | 14 Sep | Mod 7.1, 7.2 | Puzzles, Guiding the Player | Assign 2<br>18 Sep, 11:00 pm |
| 8 | 21 Sep | Mod 8.1 | Game Physics | |
| | | | | |
| 9 | 12 Sep | Mod 9.1 | AI for Games | |
| 10 | 19 Oct | Mod 10.1, 10.2 | Game Interface, Storytelling in Games | |
| 11 | 26 Oct | Mod 11.1, 11.2 | Graphics Pipeline, Animation in Games | Assign 3<br>1 Nov, 11:00pm |
| 12 | 2 Nov | Mod 12.1, 12.2 | Networked Games, Course Review | |


Course Details
Unity 3D and C#
Core Game Concepts
Game Design
Development Process
Assignments
Course Overview
Puzzles
Design Process
Testing
Game Physics
Game Interfaces
Unity3D Introduction
Risks and Prototypes
Guiding the Player
AI for Games
Graphics Pipeline
Introduction C#
Variables and Components
Object Oriented
Agile Processes
Storytelling in Games
Course Review and Exam
Booleans
Hello World
Debugging
Classes
Loops
Functions and Parameters
Lists and Arrays
Networked Games
Animation in Games
Assign 2
Assign 1
Assign 3
Lec 1
Lec 2
Lec 3
Lec 4
Lec 5
Lec 6
Lec 7
Lec 8
Lec 9
Lec 10
Lec 11
Lec 12

# Game Physics – Topics

- What is Physics?
- Some History
- Simple Motion
- Collision Detection
- Complex Simulations




Note that physics is a key ingredient of the game engine (sometimes it is even called the physics engine)

# What is Physics

Physics involves the study of how objects behave in the real world.

Often these same principles are used when trying to determine how objects will behave in the game world.

# What is Physics

Physics along with Computer Graphics and Artificial Intelligence is strongly grounded in mathematics.

Unfortunately many calculations based on the laws of physics can be slow.

This can be detrimental if you need to maintain interactive rates in a game.

# What is Physics

However much like graphics there are two main approaches to solving this problem.

1. Use fast hardware
2. Simplify the model

# What is Physics

Use fast hardware

Many calculations can be performed in hardware. There are even dedicated physics cards available (Physics Processing Units).

# What is Physics

Use fast hardware

Many calculations can be performed in hardware. There are even dedicated physics cards available (Physics Processing Units).

Simplify the model

Fortunately in a game world the accuracy of the physics is often not as critical as in the real world. Therefore shortcuts and approximations can be used to make calculations faster.

# How is Physics Used

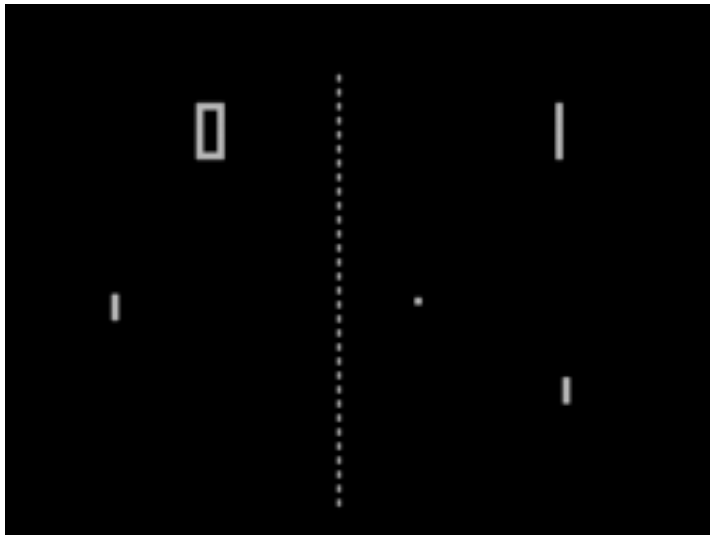
Physics can be used to model things, such as:

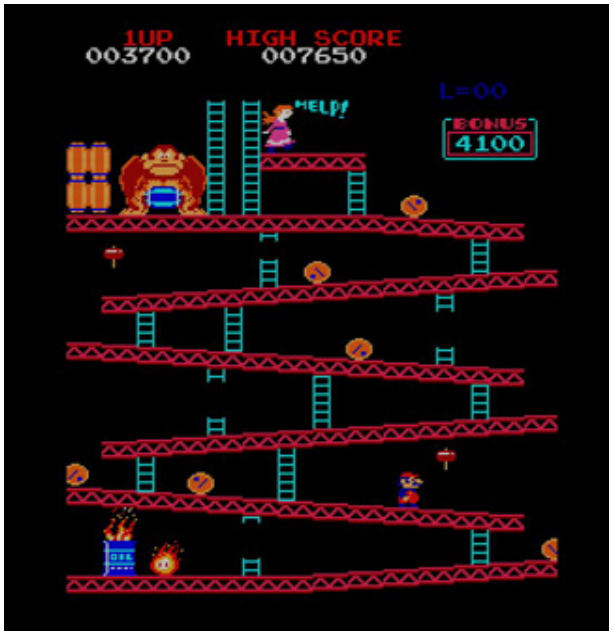
- the flight path of a missile or rocket
- the pitch, yaw and roll, and lift for airplanes
- the buoyancy and fluid drag on a ship
- the braking and cornering behaviour of cars
- the result when objects like billiard balls collide
- the way human limbs move together
- the way water and other fluids flow
- the effect of springs
- the effect of forces, such as gravity
- the way things accelerate
- the behaviour of smoke
- the effects of weather

# Some History

Many early games relied on simple physics to implement the gameplay.

e.g. Pong (velocity, simple collision and forces)
e.g. Donkey Kong (gravity, jumping and climbing)

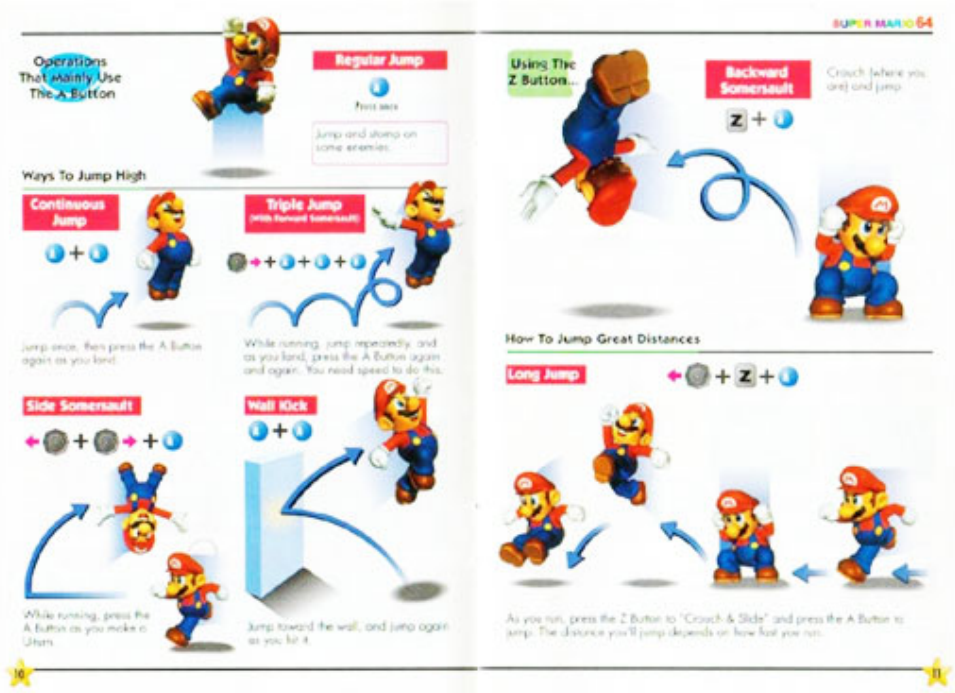




# Some History

Eventually 3D worlds were introduced along with physics calculations in 3D also became possible

e.g. Doom (running, jumping, rockets and explosions)
e.g. Mario 64 (jumping, sliding, bouncing)

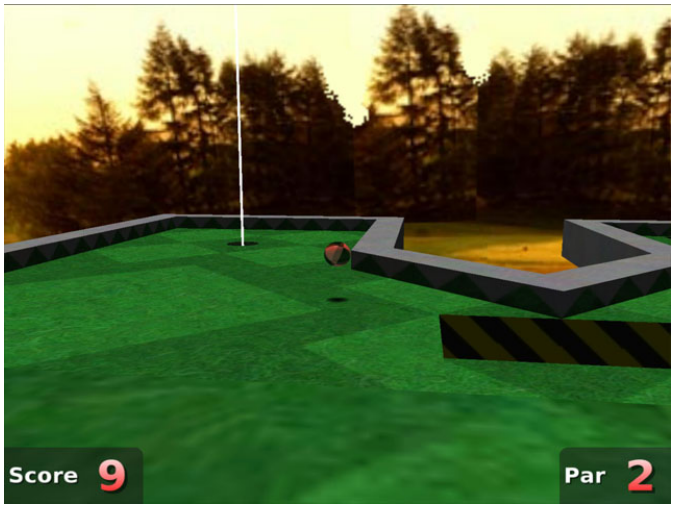



# Some History

There was some divergence in the use of physics :

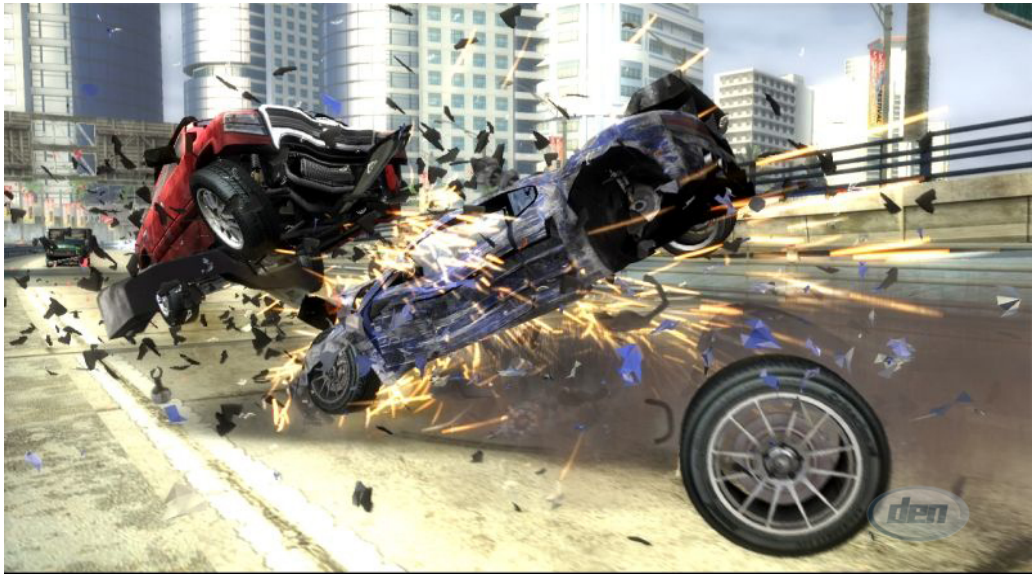
In many arcade games - crazy things could happen - there was no attempt to map reality

In simulation games - very realistic things would happen - often these games relied on realistic game worlds



# Some History

In arcade games, the physics were used to enhance gameplay with less emphasis on recreating real world situations. e.g. Spectacular acrobatics often occurred at the push of a button. e.g. *Burnout Revenge*



# Some History

Simulation games tried to use physics as realistically as possible, leading to unforgiving, but rewarding gameplay experiences.

These distinctions are most obvious in sports games were the players now expect the simulation to be as close as possible to the real world. e.g. *Grand Turismo*

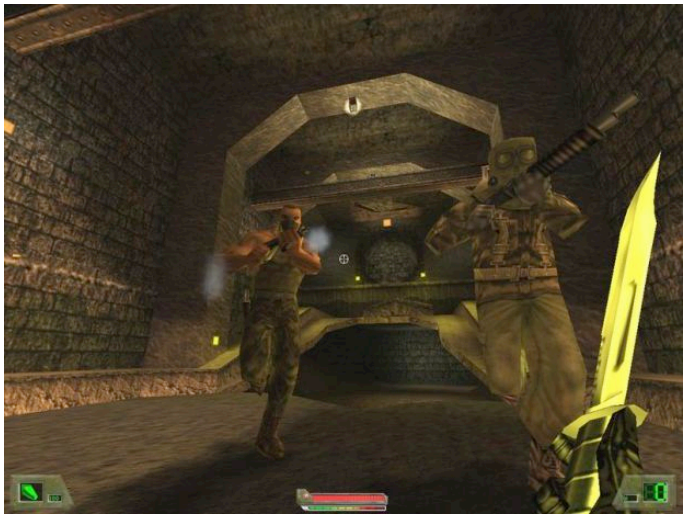
*Grand Turismo 1*


*Grand Turismo 5*

# Some History

Most character movements in early games were pre-recorded animations.

Physics began to be used to produce animations in real time, especially to show damage in shooters.
e.g. *Soldier of Fortune.*

# Some History

*Ragdoll physics* was introduced explicitly to show character injuries which depended on the situation.

This made games less reliant on pre-recorded animation and helped to increase engagement

# Some History

Even more accurate physics to drive real time character animation where characters explicitly interact with the environment were being seen in games
e.g.Grand Theft Auto IV

# Some History

The more interactivity that has entered games, the greater the reliance on physics. Many modern games rely on sophisticated physics modelling

e.g. Half-Life 2 (gravity, weight, magnetism, and buoyancy)


# Some History

Some modern games still rely on exaggerated physics modelling to create compelling gameplay

e.g. Rocket League


# Some History

While key objects and characters were influenced by physics it is only more recently that it has been possible to apply real-time physics to the environment as well. So that characters can interact realistically with the full environment. For example, structures damaged, foliage moved, etc  e.g. Crysis

https://www.rankred.com/best-physics-game/

# Simplifying Things

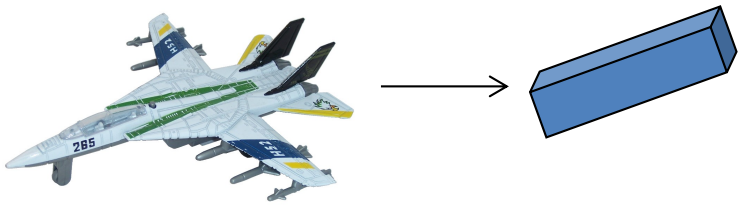
The more realistic the game world becomes the more you can expect accurate physics to play a part in the simulation.

In our games we will keep things simple and primarily use simple Newtonian physics to calculate the motion of objects (cars, airplane, projectiles, people).

# Simple Motion

Physics also becomes more complex when we considered more complicated shapes, or objects made up of parts.

One common simplification is to consider objects as a single rigid body and disregard it's real size, shape and parts in any calculation.



This is the approach we will take in this course.

# Time & Space

Physics are usually dealing in a model of space and time that is continuous.

In a game world both space and time are more discrete.

So calculations may be simplified and just performed using integers rather than worrying about decimal places.

# 2D Space

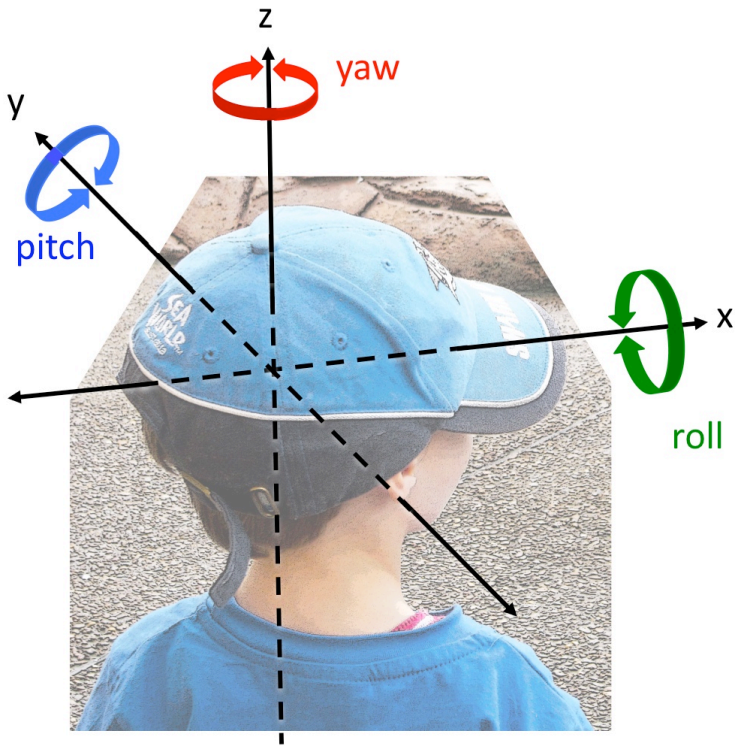
Especially in 2D games the space typically reflects the layout of onscreen pixels - using a 2D coordinate system with discrete (x,y) spatial locations.


This helps to simplify the calculations and is often accurate enough to satisfy gameplay requirements.

# 3D Space

In 3D games the space may need to be more accurately represented and so floating point representations of a continuous (x,y,z) space is commonly used.

Note that floating point calculations are slower than integer calculations although with modern hardware this is not such an issue.



# The Size of Space

In both 2D and 3D space the gameworld is never infinite in scope.

There are always boundaries which have to be accounted for in the calculations - what will happen when a character reaches the end of the world?

# Time

There are two traditional ways that time is integrated into physical calculations.

The first is to use the change of frames.

This is less accurate as the frame rate can depend on the speed of the hardware the game is running on.

Changes in frame rate can produce unexpected side effects in the motion.

(e.g. slow motion when frame rate is slow)

# Time

The second approach is to measure the passage of actual time.

This is a more reliable way to incorporate time into the physics calculations and is not affected by hardware speed.

It removes problems with inconsistent frame rates.

But can introduce jumps in movement (e.g. sudden shifts in position and apparent changes in speed when frame rate is slow or variable)

# Typical Motion

Some of the more typical elements required to create the motion required in games are :

- velocity
- acceleration
- momentum
- inertia
- mass
- forces

$$d = vt$$
$$v = u + at$$
$$d = ut + at^2/2$$
$$v^2 = u^2 + 2ad$$

# Typical Motion

## Kinematics

- Study of the motion of objects *without* taking into account mass or force
- Basic quantities: position, time
- Basic equations:

$$d = vt$$

$$v = u + at$$

$$d = ut + at^2/2$$

$$v^2 = u^2 + 2ad$$

*where:*

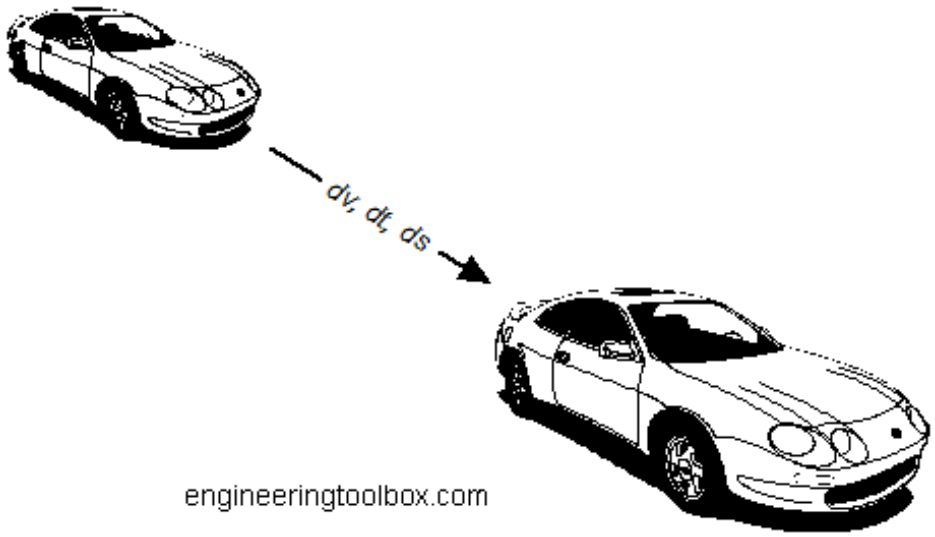
*t - (elapsed) time*
*d - distance (change in position)*
*v - (final) velocity (change in distance per unit time)*
*a - acceleration (change in velocity per unit time)*
*u - (initial) velocity*

https://web.cs.wpi.edu/~rich/courses/imgd4000-d09/lectures/D-Physics.pdf

# Typical Motion

velocity

Describes how fast and in what direction things move - usually to update the position of moving objects in games.



# Typical Motion

acceleration

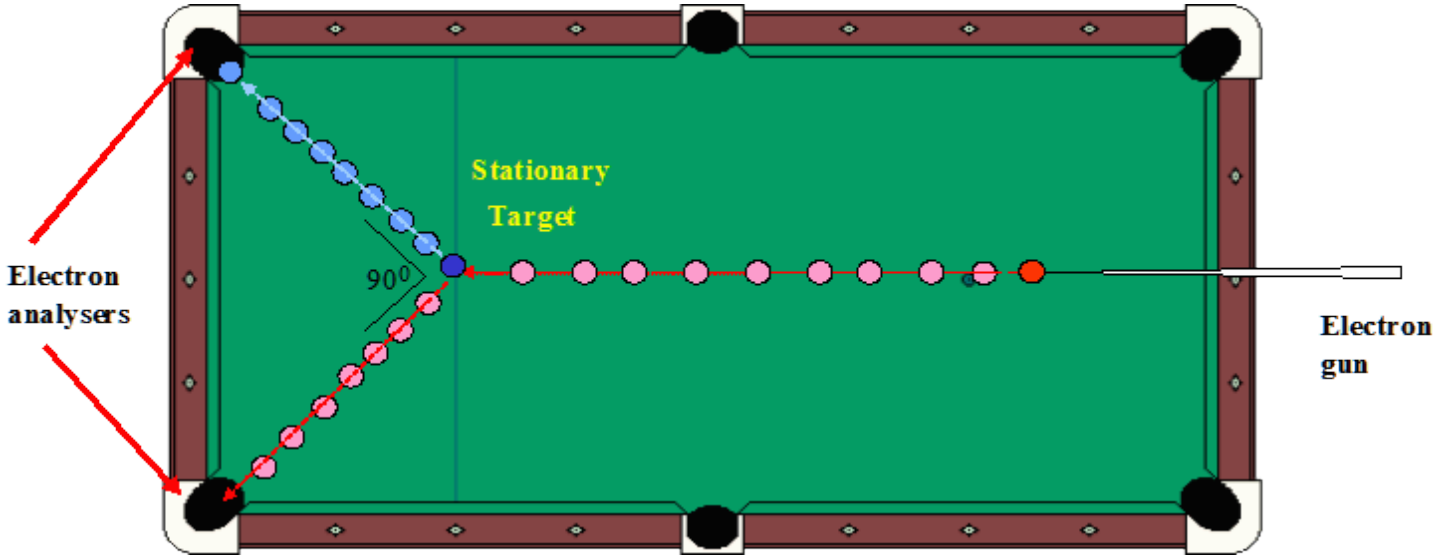
Describes how quickly the velocity of a thing is changing. Things only accelerate when forces are applied to them e.g. gravity.

Objects like cars, space ships and falling objects often experience acceleration in games. It is also possible to have a negative acceleration, or deceleration. This can occur due to forces such as friction.

# Typical Motion

momentum

The conservation of this helps explains how things move after they collide.

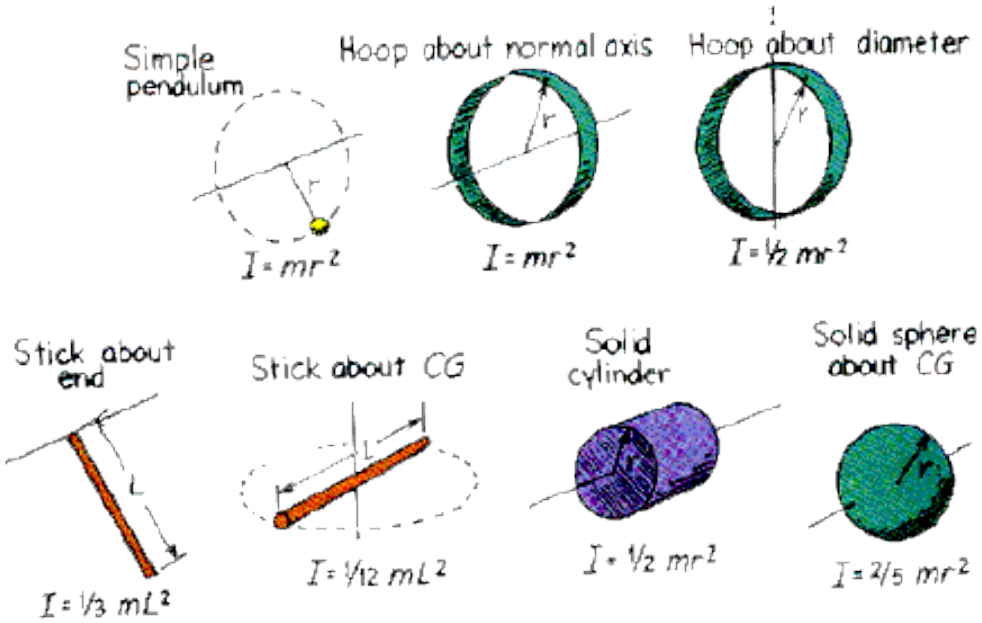


# Typical Motion

## inertia

This is the tendency for objects to stay still unless something happens to them – (e.g. a force is applied)

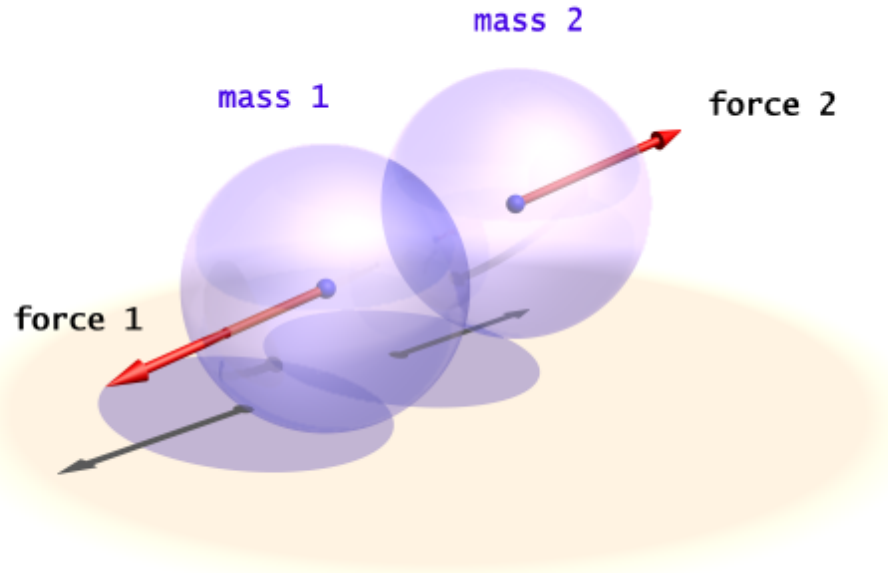
This concept proves very useful in games as nothing happens to lots of objects and this takes no time to calculate.



# Typical Motion

mass

This basically describes how much matter an object has which is important to know because this will influence the motion of an object e.g. what happens when forces act on it.



# Typical Motion

gravity

This a force acting between two objects - usually to make an object fall (accelerate) down to earth in games.

In games this might be adjusted (often higher) to make gameplay more interesting or space friendly

# Typical Motion

friction

This is a force that acts on moving objects to slow them down - it makes an object decelerate.

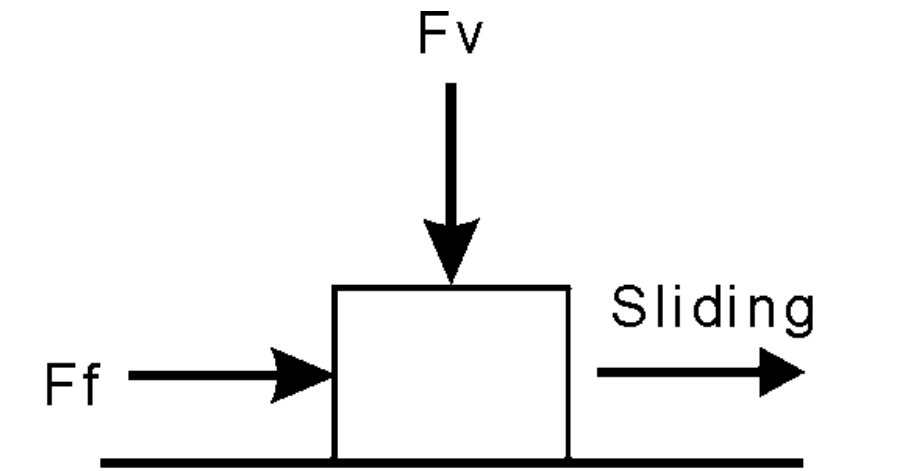



**Fig. 3.1 Friction Coefficient**
**Ff = Cf x Fv**

# Typical Motion

Some of the more typical elements required to create the motion required in games are :

- velocity
- acceleration
- momentum
- inertia
- mass
- forces

These concepts are all part of what is sometimes called *Newtonian Mechanics.*

# Typical Motion

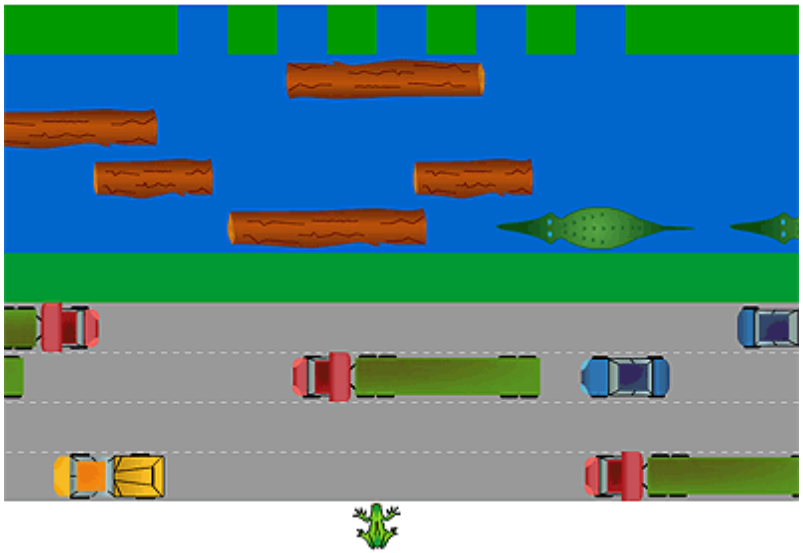
Newton's laws of motion have proved to be a very useful approximation of how things move in the real world.

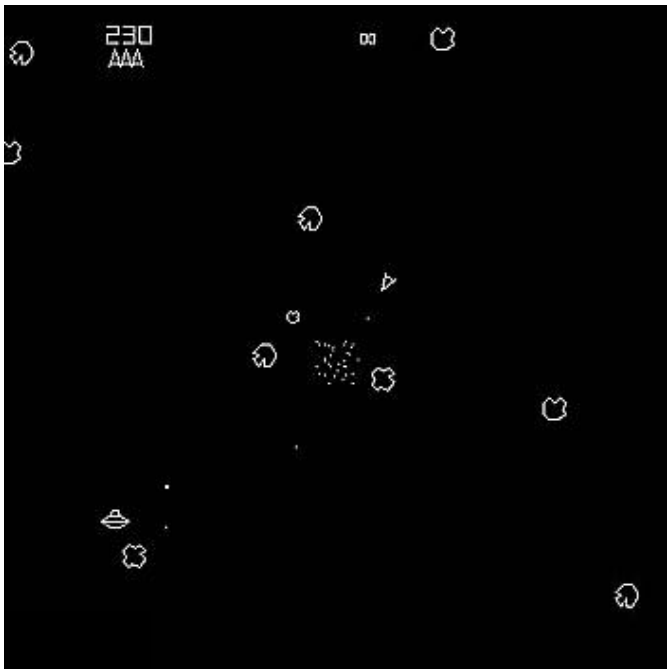
The calculations are relatively simple (especially in 2D)

Although we make some approximations - e.g. assume objects have no shape and that the mass is all centered, we also ignore some forces that have little noticeable effect on movement e.g. air resistance and material properties.

# Collision Detection

Collisions between objects in the game world are usually important events. Collisions often affect the motion of objects and also the state of the gameplay.





# Collision Detection

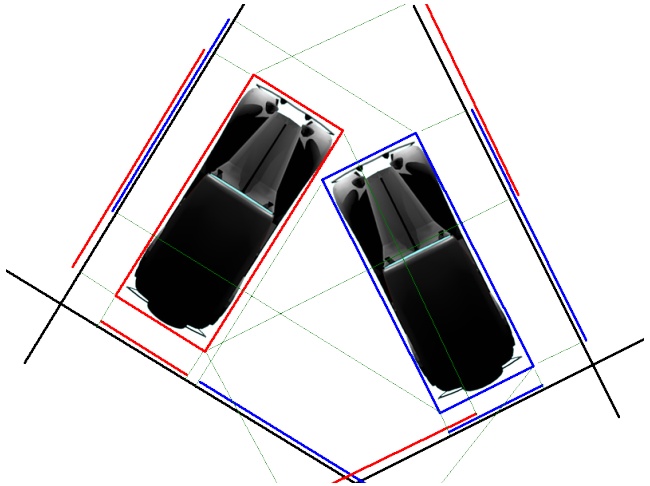
Examples

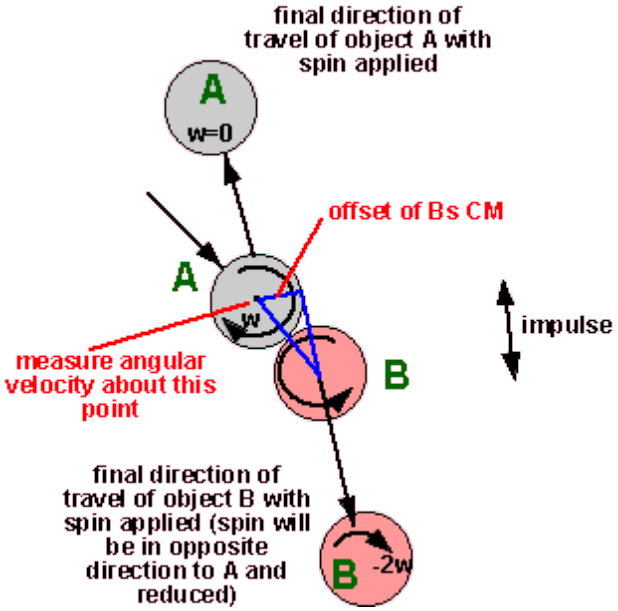
characters colliding with walls,

character touching items to pick them up

balls hitting the ground

bullets hitting a target, etc.





# Collision Detection

We therefore will also need to understood how to determine when objects are colliding.

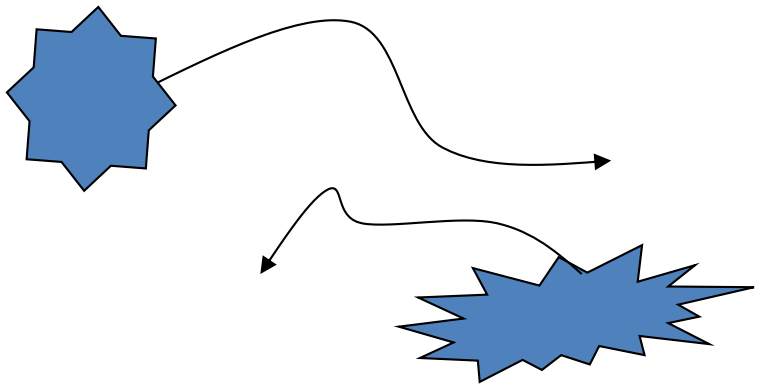
Calculating how very complex objects collide is time consuming.

Again in interactive games we will usually need to find some faster, easier ways to simplify these calculations.

# Geometry

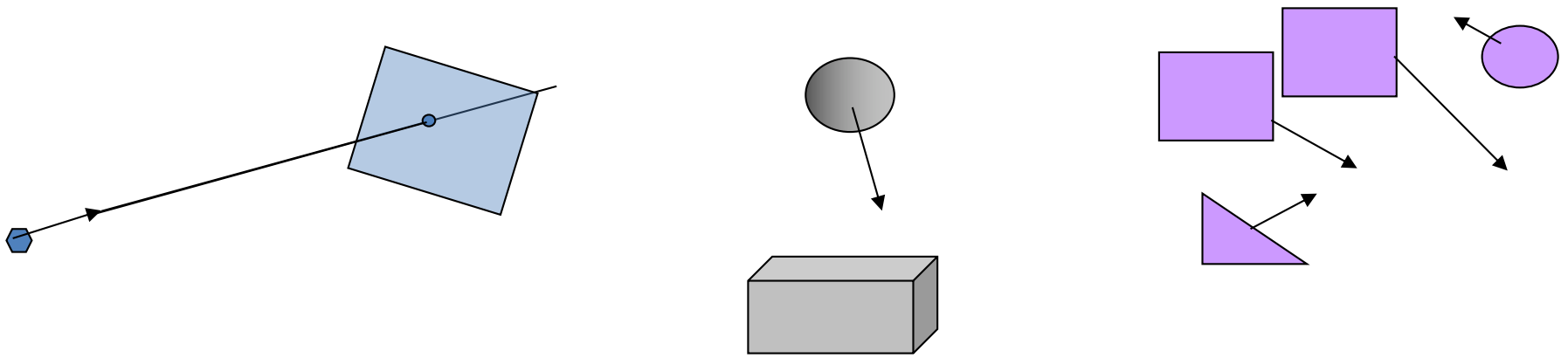
Collision detection is really a check to see if the geometry of items are intersecting.

If the geometry is complex then so are the calculations



# Geometry

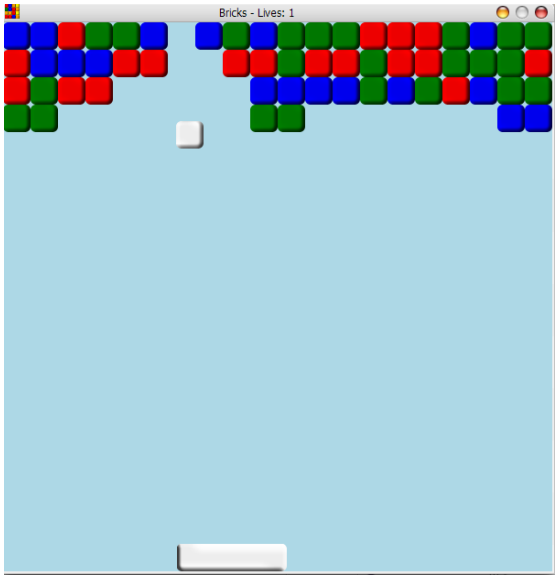
Collision detection involves a lot of geometry calculations. It involves checking to see if intersections occur between points, lines and other simple shapes such as triangles, quadrangles and 3D shapes like boxes and spheres



# Collision Detection

These calculations are simpler in 2D than 3D.

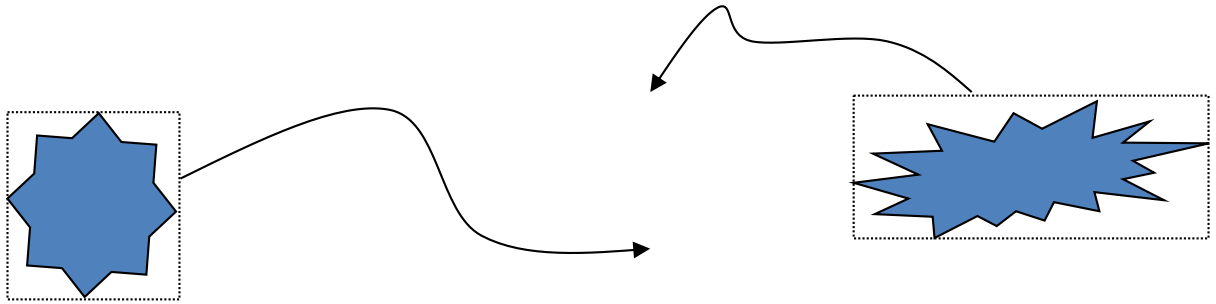
In general the simpler the geometry the quicker the detection process is. Of course the more objects in a scene the more possible collisions that need to be checked.




# Bounding Boxes

One simple way to check for collision in 2D is to ignore the real shape and just use a box that bounds the geometry of the object.

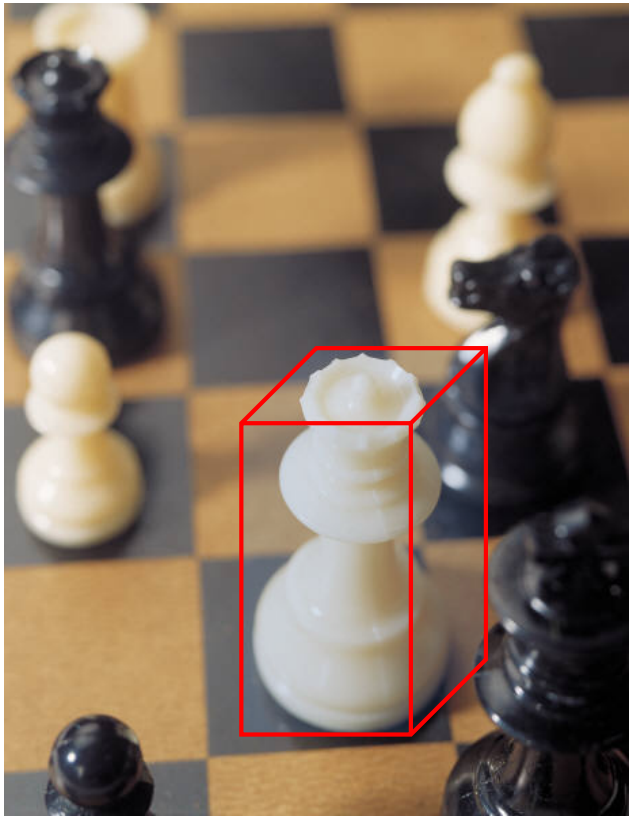
Then check if the two boxes overlap. (This is often accurate enough)



# Collision Detection

In 3D the same principle applies, except a bounding volume is used.

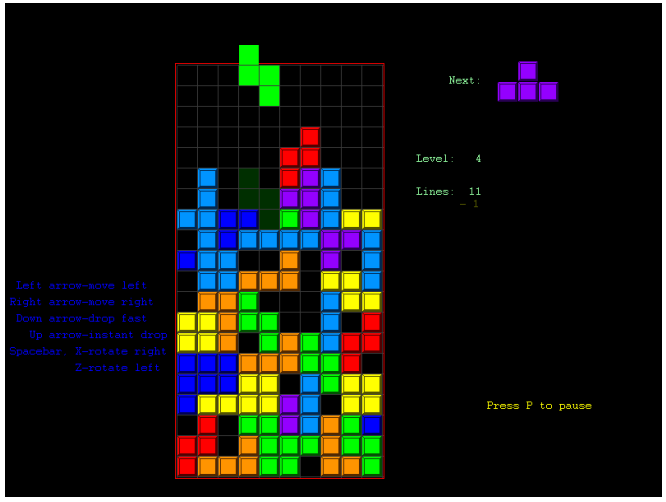
Again making sure the model conforms to a box shape helps with the realism of this simple collision detection approach.



# Collision Detection

Even where more realistic collision detection is required it is often useful to perform bounding box checks first.

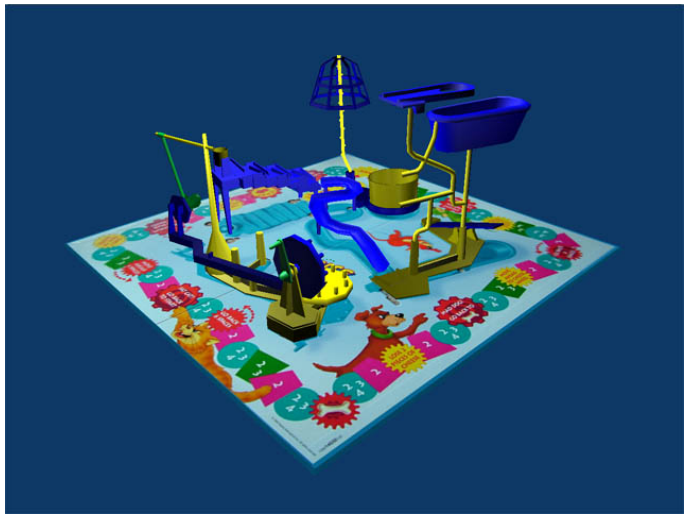
Only where a possible collision is occurring do more thorough, time consuming checks need to occur.




# Collision Detection

Apart from bounding boxes, the geometry can also be approximated by spheres (ellipsoids) or by using a convex hull (a shape that stretches around the object like an elastic band).

Again this simpler geometry makes the calculations faster then using the more complex geometry of the actual objects.



# Complex Simulations

Apart from simple motion and collision there are a number of more complex applications of the principles and laws developed in physics.

- Particle physics
- Fluid dynamics
- Animating characters
- Lighting

# Complex Simulations

We shall briefly discuss a few of these now:

- Particle physics
- Fluid dynamics

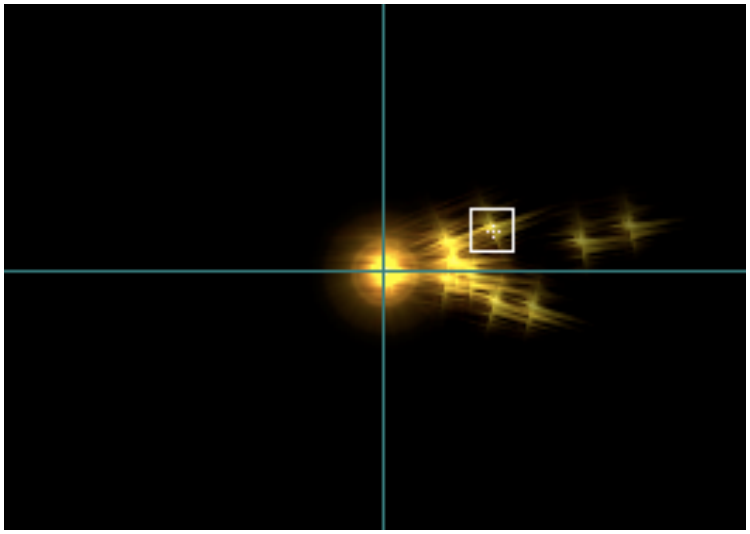
And come back to the others later:

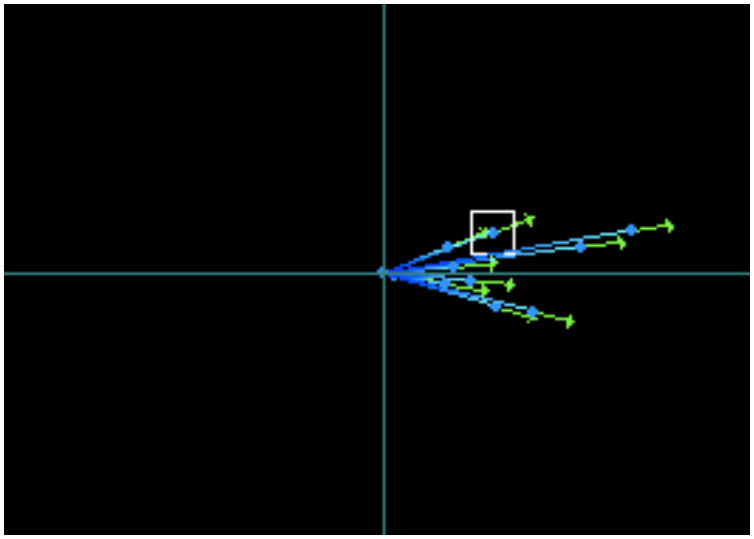
- Animating characters
- Lighting

# Particle Physics

Many real world phenomenon, such as the behaviour of gases can be modelling using particles (or lots of small moving pieces).

The same approach can be used in a game world.



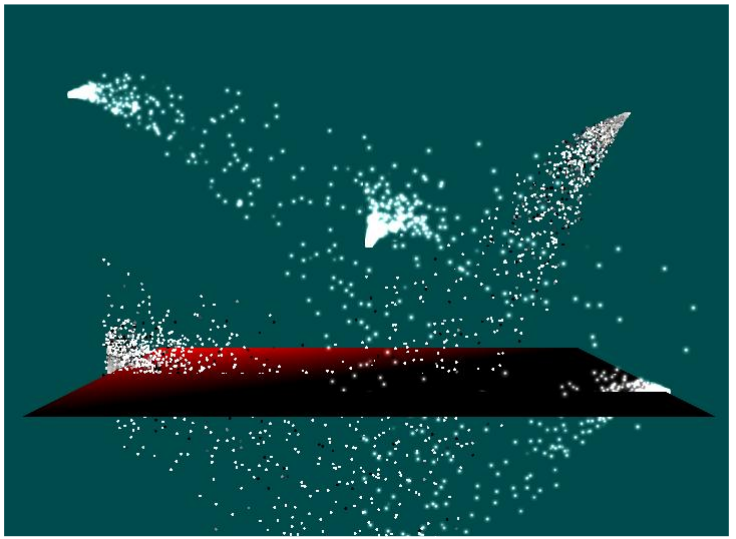


# Particle Physics

Explosions are commonly modelled in this way in games. Other example include smoke, dust, moving water, rain, etc.

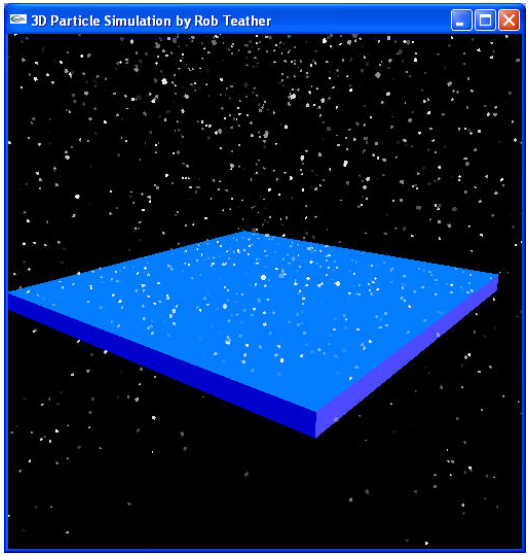
# Particle Physics

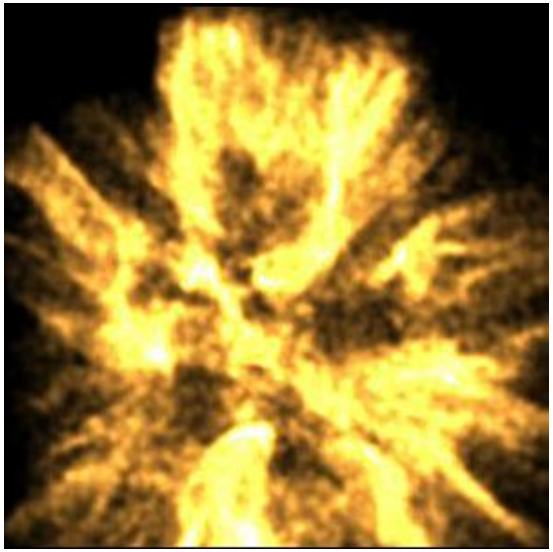
Particle simulations usually involve a lot of small round objects (particles) moving and colliding.

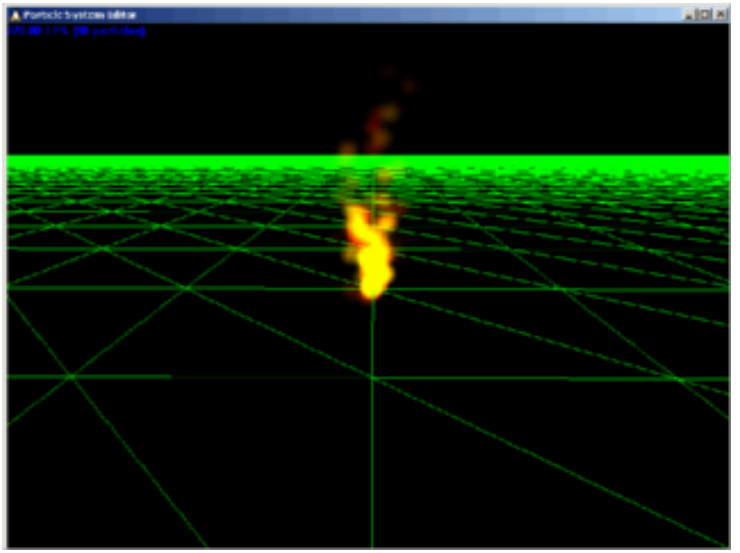


# Particle Physics

The principles are much the same as we have discussed (motion and collision) except there is usually some element of randomness provided in the number of particles and the initial motion (velocity and direction) of generated particles.



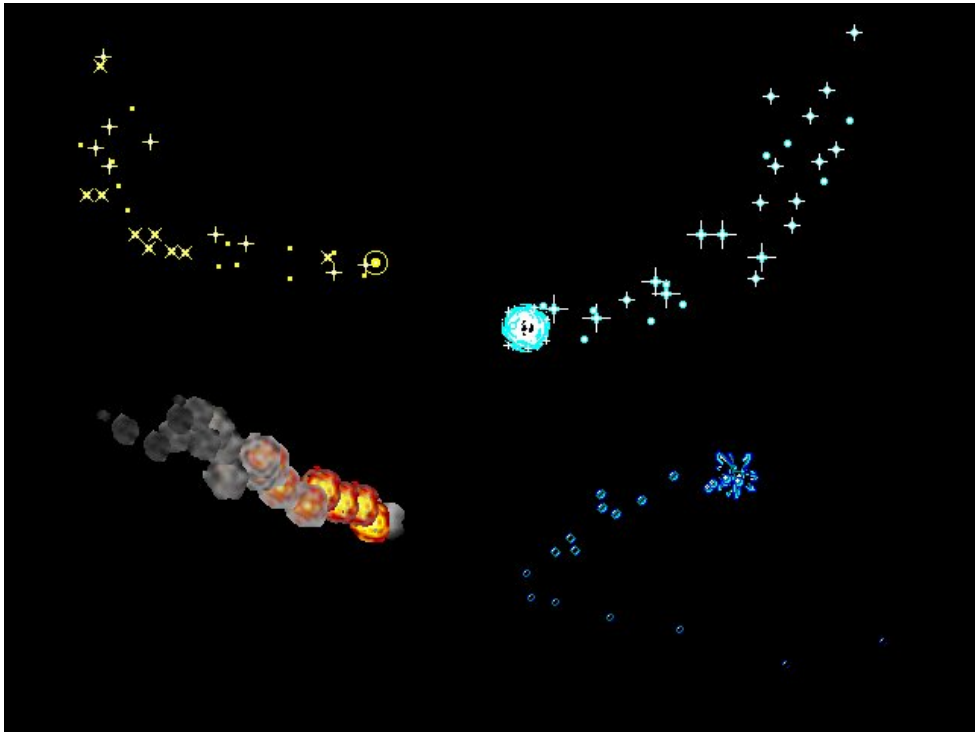





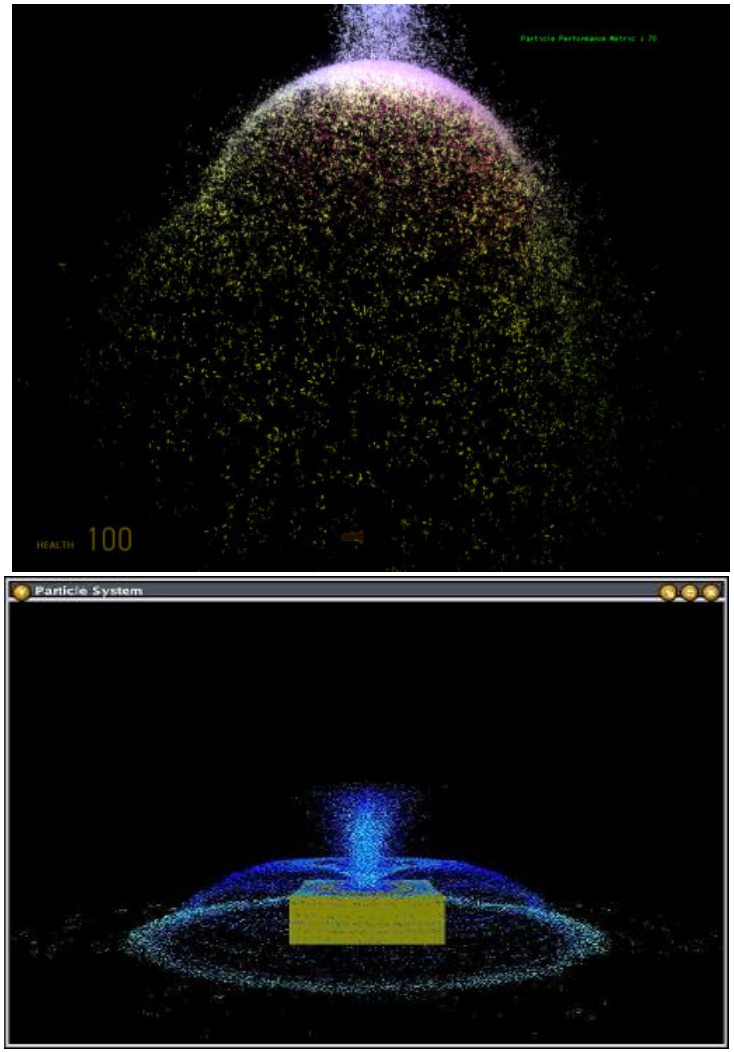
# Particle Physics

The size and number of particles will determine the realism and also the calculation cost.

The design of the particle sprite (image) is also critical in determining the final visual effect.



# Particle Physics




Remember you have to check for collisions between every object in the scene - lots of particles - lots of possible collisions.

Although many particles only last a few frames (e.g. explosions) - that is particles disappear or decay over time.

# Particle Physics in Unity3D

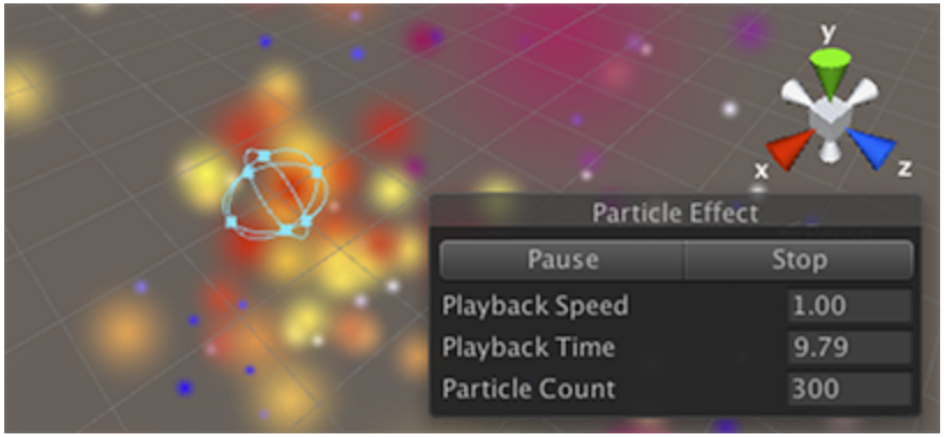
https://docs.unity3d.com/Manual/ParticleSystems.html

## Using Particle Systems in Unity

Other Versions

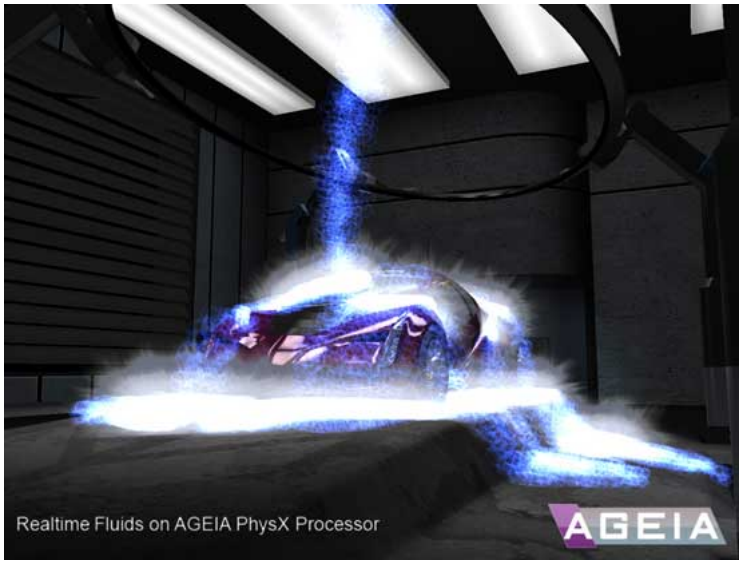
Unity implements Particle Systems with a component, so placing a Particle System in a Scene is a matter of adding a pre-made GameObject (menu: **GameObject** > **Create General** > **Particle System**) or adding the component to an existing GameObject (menu: **Component** > **Effects** > **Particle System**). Because the component is quite complicated, the Inspector is divided into a number of collapsible sub-sections or **modules** that each contain a group of related properties. Additionally, you can edit one or more systems at the same time using a separate Editor window accessed via the **Open Window** button in the Inspector. See documentation on the Particle System component and individual Particle System modules to learn more.

When a GameObject with a Particle System is selected, the Scene view contains a small **Particle Effect** panel, with some simple controls that are useful for visualising changes you make to the system's settings.

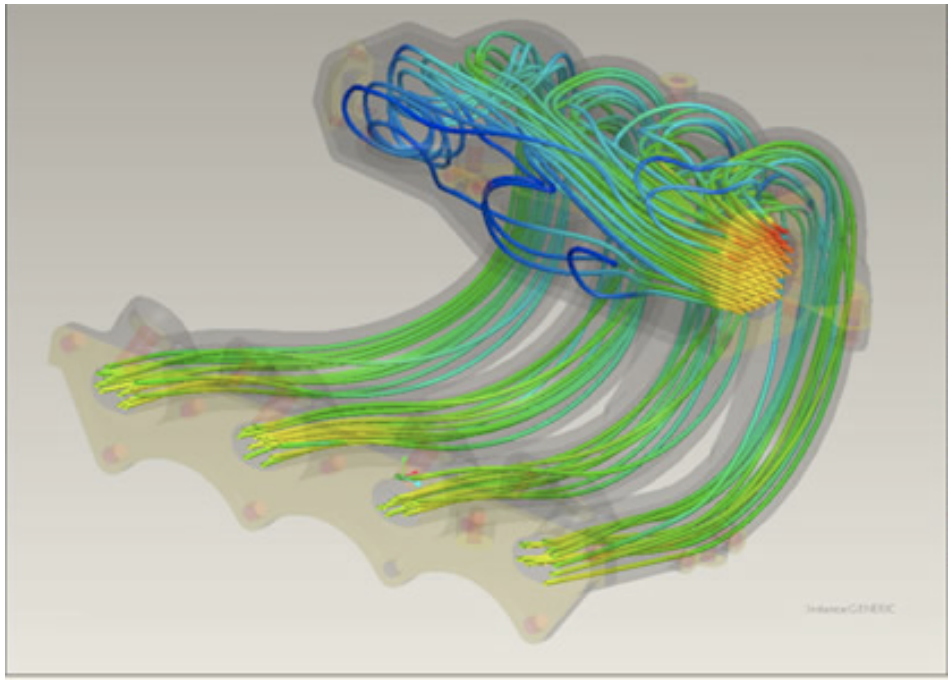


# Fluid Dynamics

Fluid dynamics is a traditional field, where super computers where used to model the behaviour of fluids e.g. how it flows and moves, turbulence etc. (although the same computational techniques can be used for gases)

# Fluid Dynamics



The number of complex vector calculations typically required to model fluids have in the past been too slow for interactive graphics.

Although the power of modern machines is seeing this change as specialised physics processing units are developed.

# Fluid Dynamics

Although these techniques produce more realistic fluid flows and gas effects - there are simpler faster techniques (e.g particle methods) to create the illusion of fluids and gases.

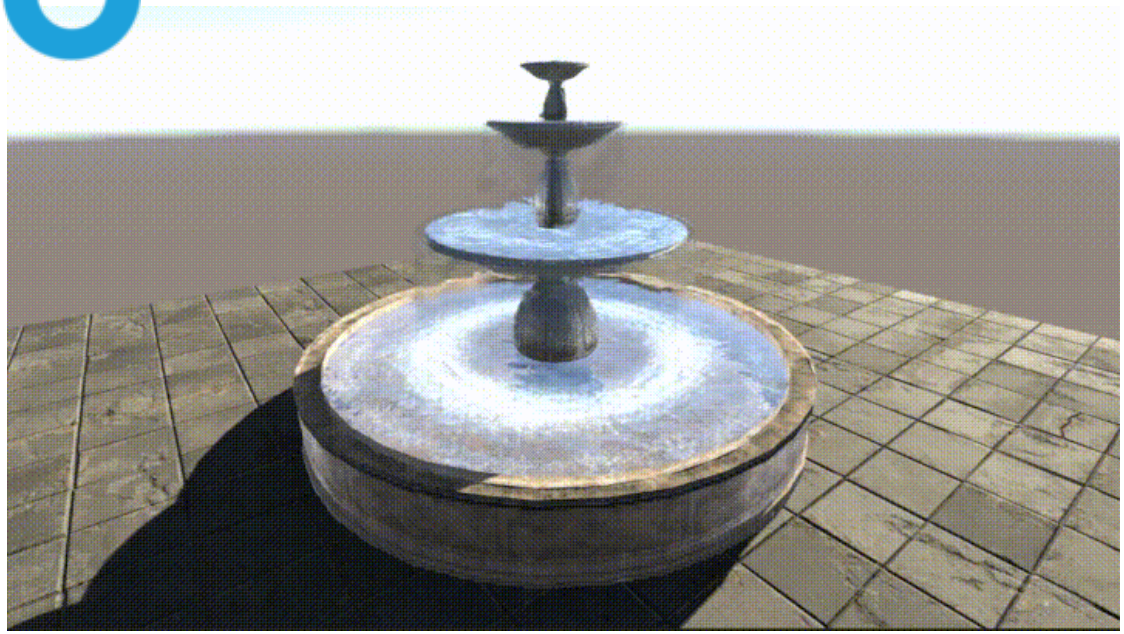
Although particle techniques may produce aesthetically less pleasing results.

# unity – Asset

https://forum.unity3d.com/threads/fluvio-fluid-dynamics-for-unity.95255/

Fluvio

This asset supports fluid like simulations- but is based onuses particle systems



# Summary

Physics relies on mathematical formulations – many of which take some time to calculate.

While you can avoid a lot of maths, concepts like vectors and coordinate systems are important to understand.

It is often good to simplify these physics to ensure faster updates.

Collision detection is one example where simpler geometries can help reduce calculation times.

Particle physics can be used to simulate more complex effects.

Unity provides specialised support for all these physics.