CSC8503

Advanced Game Technologies

Module Overview

- Fourteen Tutorials divided between four topics:
  - Physics Simulation in Real-Time
  - Artificial Intelligence
  - GPU Computation
  - Network Programming
- Three weeks…

Module Overview

- Begin today with Physics, which lasts the whole of this week
- We'll cover all 14 tutorials by next Friday
- Means you still get a full week of uninterrupted practical work to do the coursework, after having learned all the subject matter

Lecture Timetable

- This week:
  - Monday 28th, 10:00AM (now)
  - Tuesday 29th, 10:00AM, 1:00PM
  - Wednesday 30th, 10:00AM
  - Thursday 1st, 10:00AM
  - Friday 2nd, 10:00AM, 1:00PM
- Next week:
  - Monday 5th, 11:00AM
  - Tuesday 6th, 10:00AM, 1:00PM
  - Wednesday 7th, 10:00AM
  - Thursday 8th, 10:00AM
  - Friday 9th, 10:00AM, 1:00PM

Practical Timetable

- 14:00-16:00, every weekday (excl. marking day)

Coursework

- Specification is now online.
- Did anyone make the mistake of using the specification from last year, on account of not looking at the dates?

Coursework

- Physics underpins EVERYTHING
- Get your physics right
- Without the physics, you can't get enough marks to pass the coursework portion of the module
- Focus of this entire first week is physics

Additional Note

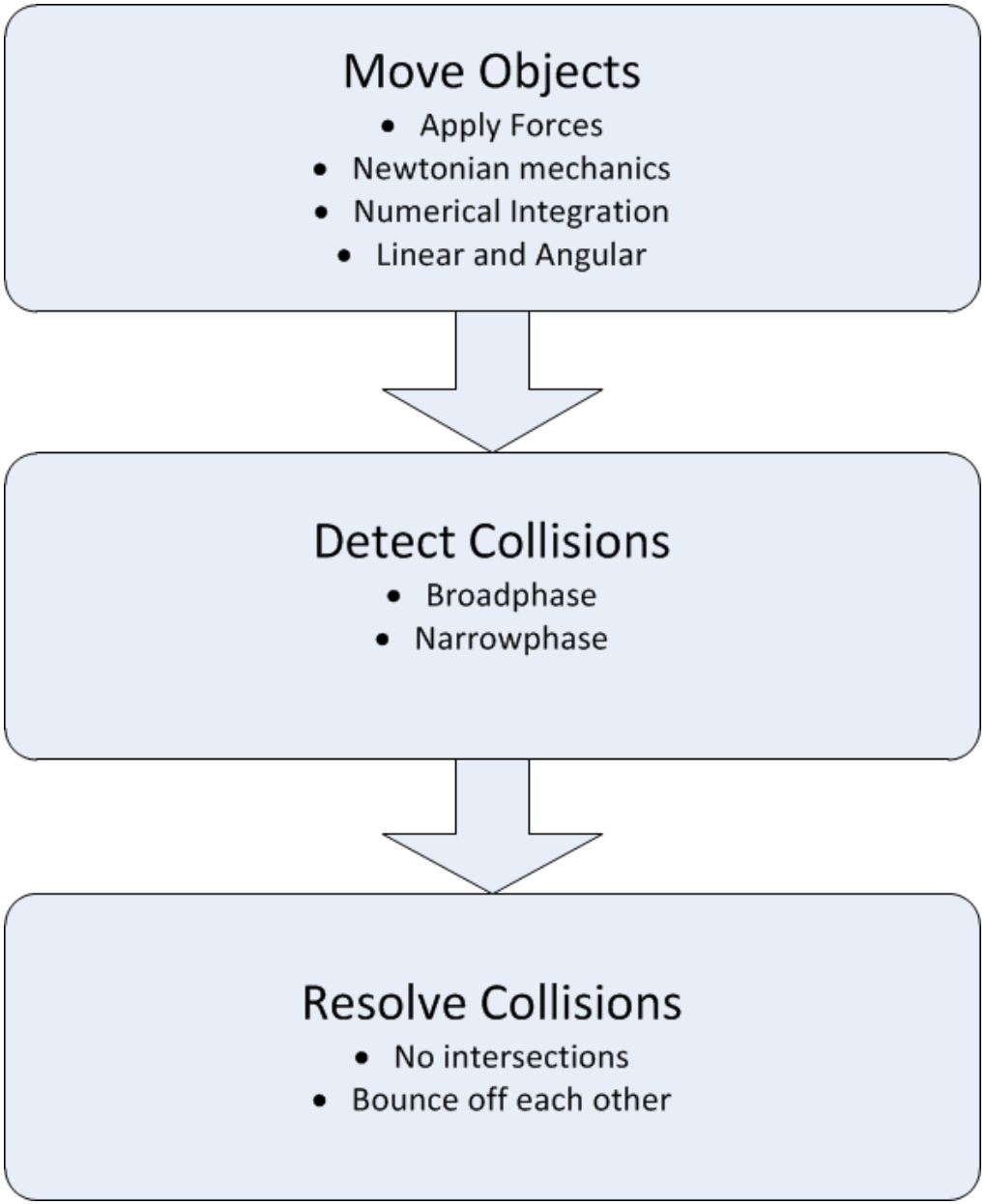
- Industry visit to ZeroLight
- Monday 12th December, 11:00-14:00

Physics Tutorial 1: Newtonian Dynamics

New Concepts

- The Physics Engine - Overview
- Linear Newtonian Dynamics
  - Newton's Laws
  - Conservation of Momentum
  - Scalars and Vectors
  - Forces
- Rotational/Angular Newtonian Dynamics
  - Torque
  - Inertia
- Physical Representation of Virtual Objects
  - Particles
  - Rigid Bodies
  - Soft Bodies
  - Physical Appearance vs. Graphical Appearance

What is a Physics Engine?


Move Objects
Apply Forces
Newtonian mechanics
Numerical Integration
Linear and Angular
Detect Collisions
Broadphase
Narrowphase
Resolve Collisions
No intersections
Bounce off each other

The Physics Engine: Overview

- Physics engine **must** be a compartmentalised subsystem
- In most modern games, physics comes second only to rendering in terms of processor scheduling
- The physics update cycle is normally threaded away from all other game engine tasks
- Operates on a **fixed** time step
  - Dynamic time steps can lead to breakages

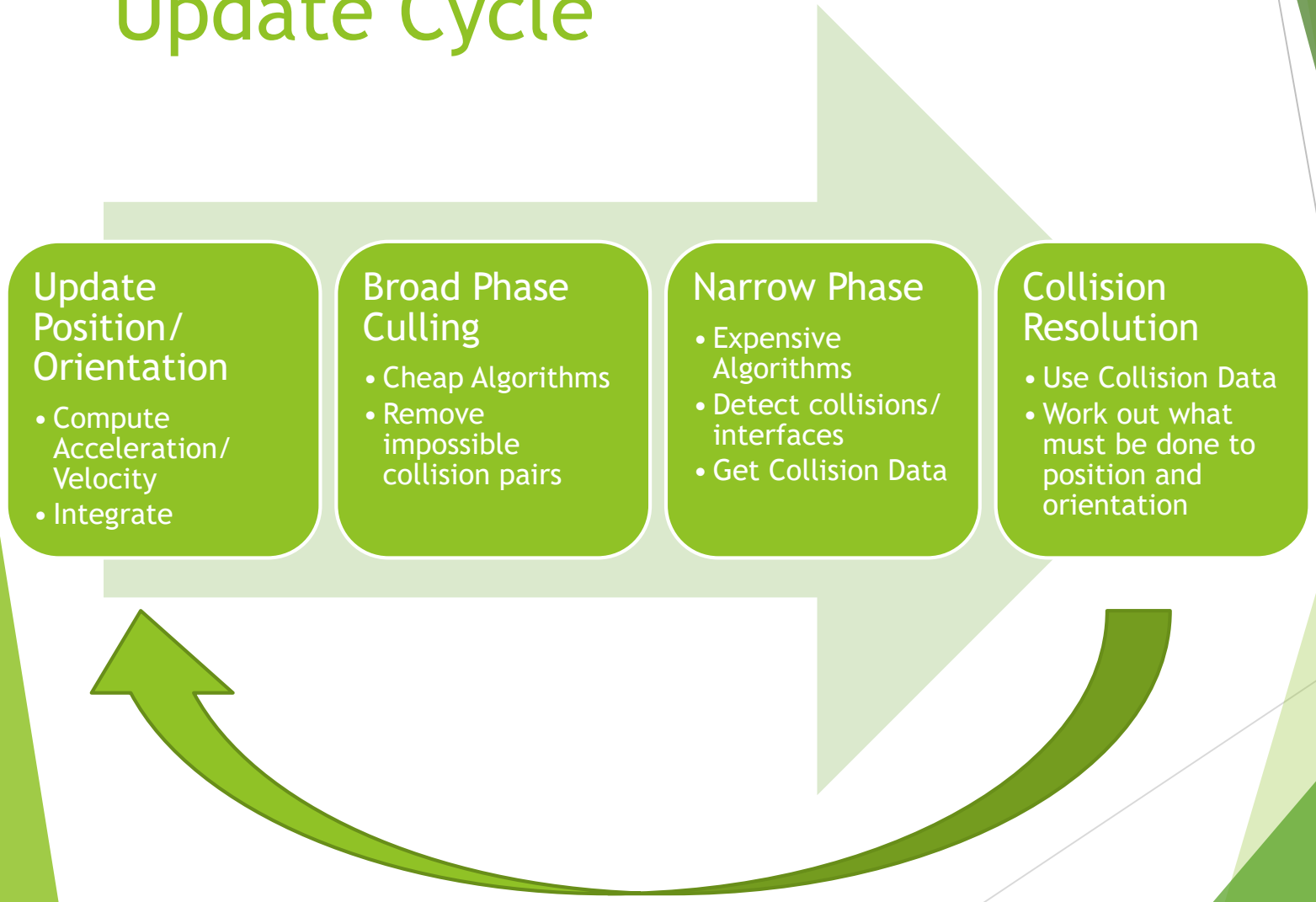
The Physics Engine: Overview

- On the subject of dynamic time steps:
  - Though we're a bit dismissive of them, they actually can work, and are used semi-regularly. In the framework there is a function which allows you to dynamically update your time step
  - The approach we take to constraint solvers is slightly predicated on fixed time steps – things like the Baumgarte offset, which are a function of time, need recomputing dynamically in a dynamically time stepped system
  - The core problem with them is networking; if the physics updates aren't consistent between clients and the server, terrible things happen.

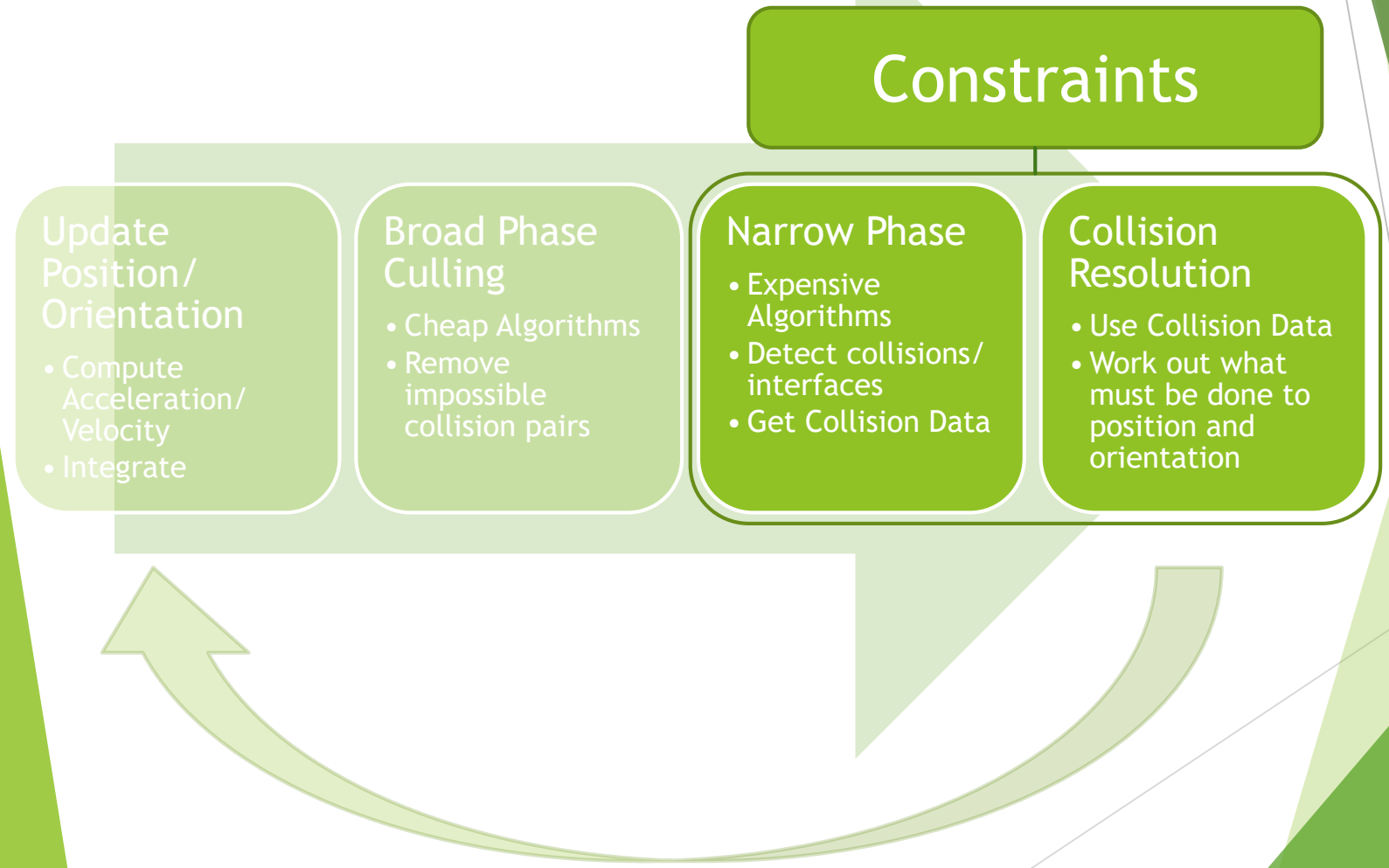
The Physics Engine: Overview

- Responsibilities of the Physics Engine:
  - Update/Maintain positions and orientations of items
  - Determine what collisions are possible (broad phase)
  - Determine what collisions have happened (narrow phase)
  - Generate data regarding those collisions
  - Resolve those collisions

The Physics Engine: Update Cycle



The Physics Engine: Update Cycle



The Physics Engine: Update Cycle

- Constraint-based solvers, really, relate mostly to the last step – collision resolution
- But how you resolve collisions is directly connected to how you set them up in the first place
- As such, we need to introduce the concept of constraints *early*, so we aren't caught with our pants down trying to implement a constraint-based resolution system *later*

A note on Code Samples

- Many of the lecture notes in this series include illustrative code samples
- Some of these are just that – *illustrative*. They're giving you an idea of what we're talking about
- In that sense, this module differs significantly from other work you've done, where everything in the code needed to be embedded in your project
- In your physics system, it's up to you to reason about what to implement – and it's up to us to equip you to make those decisions

A note on Code Samples

- You are actively encouraged to explore commercial physics engines – in the industry, you will normally be using such an engine
- Your coursework **won't** employ a commercial physics engine – everything in there you'll either write yourself, or draw from the tutorial sample code
- The purpose of teaching you how a physics engine works, from start to finish, is to equip you to understand the complexities involved
- Not just intellectual complexity but *computational* complexity – more-so than graphics, inefficient physics can hammer your processor scheduling and, consequently, perceived frame rate – stationary objects may as well not be rendered

Linear Newtonian Dynamics

Newton's First Law

- A body will remain at rest or continue to move in a straight line at a constant speed unless acted upon by a force.

- This is the foundation of all dynamics.
- Objects at rest remain at rest until they have some reason to stop resting.
- Objects moving at a given velocity remain at that velocity until there's a reason for that velocity to change.

- Collision
- Applied Force
- Removal of a balancing force (see Third Law)

Newton's Second Law

- The acceleration of a body is proportional to the resultant force acting on the body, and is in the same direction as the resultant force.
- $F = ma$

- First law tells us when something needs to change.
- Second law tells us that needs to change.

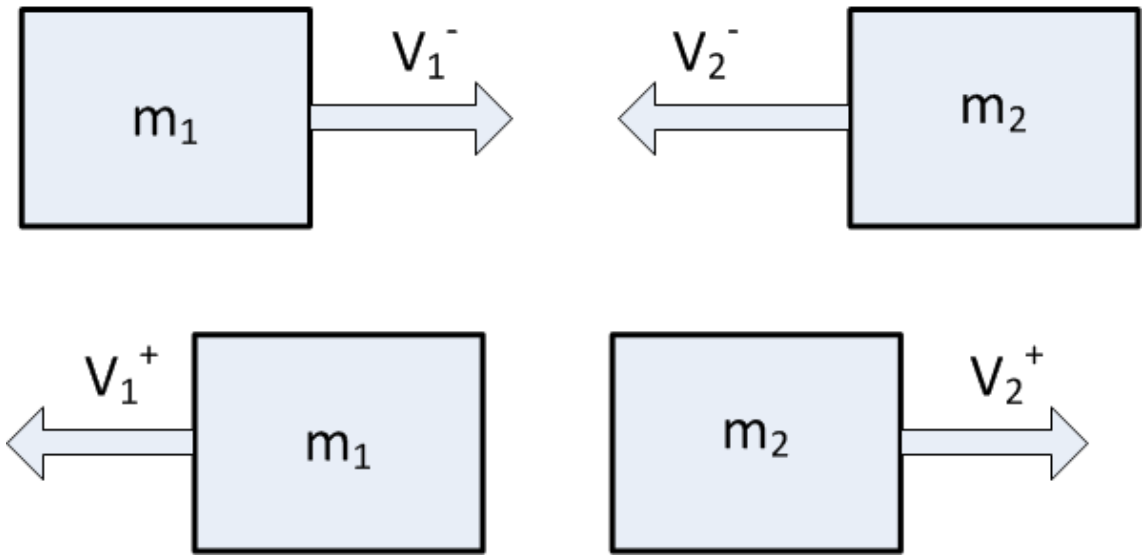
Newton's Third Law

- For every action there is an equal and opposite reaction.

- Third law is book-keeping.
  - For the first two laws to be true, there needs to be balance.
  - Objects sit at rest all around this lab, despite there always being a gravitational force on them.
  - Ergo, there's a counter-force applied against them by the object they're resting on. It must be equal, not greater than, or they'd fly.
  - First person to mention the Meissner Effect fails the module.

- Question: Do we need to compute this every frame?

Conservation of Momentum



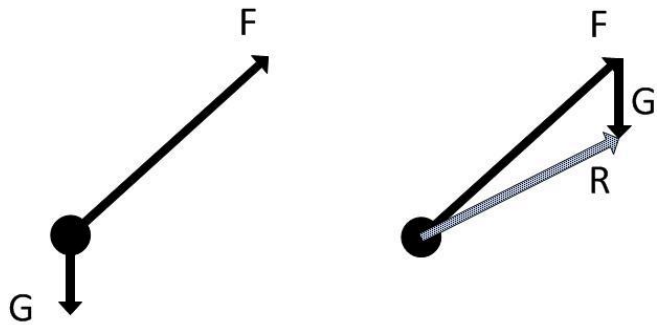
$$m_1 v_1^- + m_2 v_2^- = m_1 v_1^+ + m_2 v_2^+$$

This law is fundamental to how objects interact with each other. The first three have basically told us about the forces acting on objects; this one addresses forces imparted between objects

Vectors and Scalars

- Almost all of our computation relating force to motion is performed using vectors.
- As in graphics, when we use the term vector, we mean something which has both **magnitude** and **direction**.
  - We don't mean the "STL Vector" container class.
- When we use the term scalar, we mean something which **only has magnitude**.
- Consider the **momentum** example in the previous slide:
  - Is mass a vector, or a scalar?
  - Is velocity a vector, or a scalar?
  - Is momentum a vector, or a scalar?
- Is speed a vector, or a scalar?

Resolving Multiple Forces



- An object of mass m is subject to two forces:
  - $F = \begin{bmatrix} 2 \\ 5 \\ 1 \end{bmatrix}$
  - $G = \begin{bmatrix} -2 \\ 2 \\ 2 \end{bmatrix}$
- What is the resultant force, R, on the object?
- What is its acceleration?

A note on Acceleration, Velocity and Displacement

- Acceleration is the change in velocity over time. If something has acceleration (or deceleration), it is moving faster (or slower).
- Velocity is the change in displacement over time. If something has velocity, it is moving.
- Displacement is the mechanical term for an object's position (position is generally used in our tutorials). If something has had a velocity, its displacement has changed.
- These relationships will be explored this afternoon

Rotational/Angular Newtonian Dynamics

Why do we need angular motion?

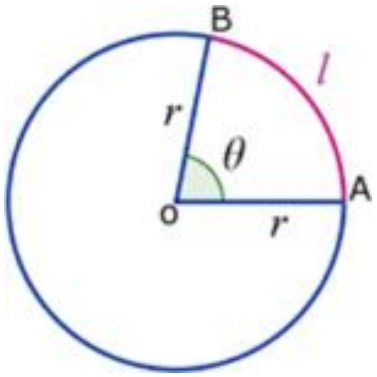
- Objects in our game need more than just a position – they need orientation.
  - Even spheres, if textured, need an orientation to ensure textures map correctly
  - Snooker balls – which aren't even textured, though may be mapped for lighting effects – need orientation, and rate of change of orientation, even more – spin happens.
- Tracking and updating the orientation of objects is a fundamental element of physics
  - Without it, our objects never rotate in response to stimuli.
  - Imagine a collision between two cubes – how believable would that be without rotation?

What does it mean in computational terms?

- It’s another layer of complexity
- We’re adding computational expense to each object’s updates
- It’s often seen as ‘more difficult’ than linear motion
- But really, it has direct analogue – we’re just solving the same problem for a different property of the object

A note on Radians

- Radians are a physicist's unit of angle, based on the radius of a circle
  - Circumference of a circle is $2\pi r$
  - So, for a circle of radius r, its radius will fit along the circumference 2π times
  - If we draw analogue to degrees (360° in a circle), that means there are 0.01745… radians in a degree, or 57.2957795… degrees in a radian
  - An angle given in radians is the ratio of the arc swept by an angle θ radians to radius r of the corresponding circle



$$\theta = \frac{l}{r}$$

A note on Radians

- Why do we care?
  - The radian is the fundamental unit of angular motion
  - Trigonometric functions expect inputs to be handled in radians in most cases
  - Keeping everything in radians allows us to leverage the above ratio to avoid conversions to and from degrees

- Sometimes angles will be greater than a full rotation when computed – a quick fix can be used to ensure these situations don't break your maths:
  - If $\theta > 2\pi$ $\quad \theta -= 2\pi$
  - If $\theta < -2\pi$ $\quad \theta += 2\pi$

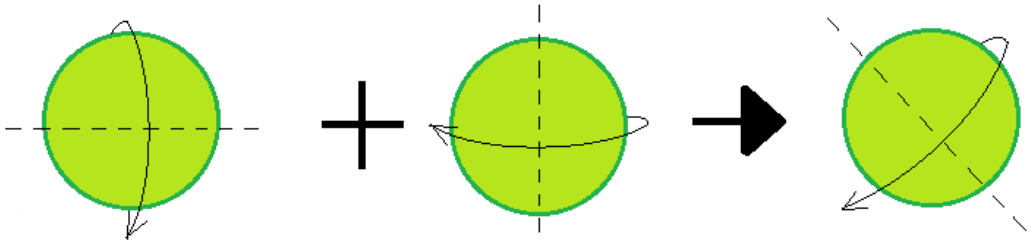
A note on Quaternions

- You will live, eat, sleep and breathe quaternions for the remainder of the degree programme
- These were introduced in the Skeletal Animation tutorial in your graphics module, but we'll briefly go over them here

- We're reminded that a quaternion if a four-element vector of the form (a, b, c, w).
- The **orientation** of an object is normally stored as a quaternion, and the maths represents it using the symbol Θ

A note on Quaternions

- So, let our orientation Θ be of the form (a, b, c, w)
- The orientation of our object itself can be described, in a physical sense, in terms of a three-element vector which defines the angle about which it has rotated (denoted $\hat{\boldsymbol{n}}$), and an angle around that axis which it has rotated (denoted θ).
- This can be tricky to get our heads around, as normally we think about objects rotating in multiple directions at once – but the trick is realising that a single angle can describe all rotation, if we define the axis about which the rotation occurs ourselves, e.g.:



- We reiterate this mainly because it's possible to gloss quaternions over a bit in graphics and still 'get' what animation does; not so in physics

A note on Quaternions

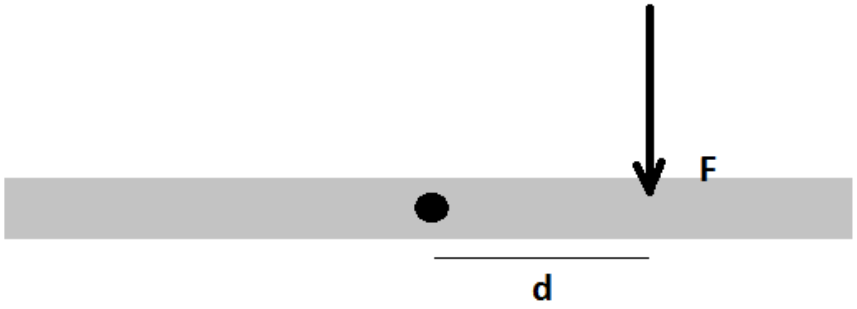
- So, once we've established our axis of rotation $\hat{n}$, where $\hat{n} = (x, y, z)$, and our angle of rotation θ about that axis, our orientation Θ can be calculated as:

$$\Theta = \left(x \sin\left(\frac{\theta}{2}\right), y \sin\left(\frac{\theta}{2}\right), z \sin\left(\frac{\theta}{2}\right), \cos\left(\frac{\theta}{2}\right)\right)$$

- Why bother with quaternions, though? Aren't matrices easier to visualise?
  - Lower memory footprint
  - More operations to create, but fewer operations required to chain updates
  - Your physics simulation is one, massive chain of updates

Fundamentals of Angular Motion: Torque

- Torque is the term in physics defined as the cross product of a force applied to an object and the distance that force is applied from the object's pivot point



$$\tau = F \times d$$

- This is a cross product because we're acting across three dimensions

Fundamentals of Angular Motion: Torque

- In this way, torque is our angular analogue to force in linear motion
- This relationship between torque and force also indicates how we can unify the various forms of motion based on a single variable
- But what is our analogue to acceleration?

A note on Angle, Angular Velocity, and Angular Acceleration

- As noted in our discussion of radians, θ is our angle. This is analogous to displacement
- ω denotes our angular velocity (the change in angle θ over time)
- α denotes our angular acceleration (the change in angular velocity ω over time)
- Like their linear cousins, they'll be discussed this afternoon

Fundamentals of Angular Motion: Inertia

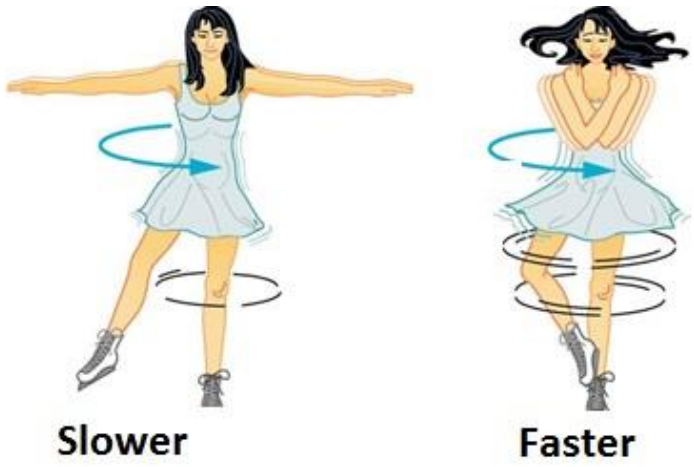
- Now we have an analogue to acceleration, and an analogue to force, we need to establish how they're connected
- The relationship between torque and angular acceleration is referred to as the **moment of inertia.** Torque is the product of inertia and angular acceleration

$$\tau = I\alpha$$

- Inertia represents the resistance a body has to change of state of angular velocity

Fundamentals of Angular Motion: Inertia

- The moment of inertia depends on distribution of mass about the axes of the rotating object
- The wrinkle: Inertia is a complex property. A scalar value I wouldn't contain enough information to describe these properties of an object
- The answer: Inertia must be represented as a matrix



Fundamentals of Angular Motion: Inertia

- The inertia matrix, or **inertia tensor**, for an object in three dimensions takes the form:

$$\begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{bmatrix}$$

- As noted before, we work in vectors when handling three-dimensional motion; $\tau = I\alpha$ becomes

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

Fundamentals of Angular Motion: Inertia

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

- Well, that's a nice formula, but what does any of it mean?

- It's not actually that complicated
  - All axes need to be considered, and the effect of torque on one axis isn't always limited to the same axis of angular acceleration
  - So, I_{xx} represents the effect τ_x has on α_x. I_{xy} represents the effect τ_x has on α_y, while conversely I_{yx} represents the effect τ_y has on α_x - and so on.

Fundamentals of Angular Motion: Inertia

- Important properties of the inertia matrix:
  - The diagonal elements (I_{xx}, I_{yy}, and I_{zz}) must **never** be zero – which is another way of saying that τ_x must always have some effect on α_x, etc.
  - The matrix must be symmetrical (e.g., $I_{xy} = I_{yx}$, $I_{xz} = I_{zx}$, and $I_{yz} = I_{zy}$)

Fundamentals of Angular Motion: Calculating α

- So, we've established how to determine torque:

$$\tau = F \times d$$

- And the relationship between torque and angular acceleration via the inertia matrix:

$$\tau = I\alpha$$

- But how do we actually compute the angular acceleration itself?

- We need to keep in mind that torque and angular acceleration are both three-element vectors, and the inertia matrix is, well, a matrix

Fundamentals of Angular Motion: Calculating α

- If $\tau = I\alpha$ then:

$$\alpha = I^{-1}\tau$$

- Where I^{-1} is the inverse of the inertia matrix

- The inverses of diagonal matrices are easy to compute, as each diagonal element is replaced by its reciprocal
- The inverses of other matrices require a full matrix inversion computation, which is highly expensive
- BUT assuming the bodies in our simulation don't change, or change in a predictable fashion (e.g., a spaceship which always breaks the same way), we can handle the generation of inverse matrices before compiling our program, or during loading

A note on Symmetry

- The inverse inertial matrices of symmetrical objects are diagonal and, therefor, easier to compute than their asymmetrical cousins

- As a result, objects in games which aren't really symmetrical (such as barrels and bricks) are often considered symmetrical for the purposes of angular motion

A note on Symmetry

- This is especially true for objects whose inertia tensor is likely to change during runtime (to mitigate the cost of those pesky re-computations)

- A completely symmetrical object has a specific, key property: torque about an axis only causes rotation about that axis – so all the non-diagonal elements of the inertia matrix can be set to zero, e.g.:

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

A note on Symmetry

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

- As to why the diagonal elements must be non-zero, consider the equation $\tau = I\alpha$.
  - If I_{xx} is the effect τ_x has on α_x, then for any non-zero value of τ_x, α_x needs to be infinite to satisfy $I_{xx} = 0$.
  - This is a Bad Thing.
  - Additionally, breaks several laws of thermodynamics.
  - Which is also a Bad Thing.

Special Cases: Spheres and Cuboids

- Many objects in your game can be abstracted to solid spheres, or solid cubes, of uniform mass distribution
- This simplifies computation significantly, and can be a tidy way of improving performance without too adversely affecting accuracy
- You should be able to recall the following

Special Cases: Spheres and Cuboids

- Consider a solid sphere of radius *r* and mass *m*

- The effect of an applied force (any value of torque at radius *r*) will be identical across all axes (there is no purer shape than a sphere). That effect is given by the equation

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

- And the inertia matrix of the sphere takes the form:

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

Special Cases: Spheres and Cuboids

- Consider a solid cuboid of length *l*, height *h*, and width *w*.

- The effect of an applied force (torque) differs for each axis, such that:

$$I_{xx} = \frac{1}{12} m(h^2 + w^2),$$

$$I_{yy} = \frac{1}{12} m(l^2 + w^2),$$

$$\text{and } I_{zz} = \frac{1}{12} m(h^2 + l^2)$$

- Where the inertia matrix of the sphere takes the form:

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

A note on Asymmetry

- So, we already know asymmetrical objects potentially generate more computational expense
- Why bother with them?
- Well, almost no objects in the universe (of a scale we can meaningfully interact with, much less see) are completely symmetrical.
  - There's an argument to be made for the cores of neutron stars – but if you're meaningfully interacting with one of those, you're having a Very Bad Day.

A note on Asymmetry

- This means it's entirely feasible that a game will have to handle objects which aren't totally symmetrical in a fashion the player finds believable

- The handout includes the equations which actually determine the inertia matrix of asymmetrical objects, but the key characteristics of such a matrix are:
  - Non-zero diagonal elements
  - At least some other elements are non-zero, but symmetrical (e.g., $I_{xy} = I_{yx}$, etc.)
- The inverses of these matrices are expensive to compute on the fly. If at all possible, try and do so beforehand

Physical Representation vs. Graphical Representation

Physical Representation

- Three general ways in which the physical nature of an item in our environment can be modelled:

Physical Representation: Particle Systems

- A particle based physics system doesn't care about collisions, only motion
- All objects in the environment are particles – points in space.
- They might have mass (depending on the system being solved), but they have no radius

- Because they have no radius, they can't represent torque; no torque, no angular motion
- Because they have no radius, they can't collide
- These limitations make particle systems more meaningful to graphical effects than physics

Physical Representation: Rigid Bodies

- Most physical calculations in video games are applied to rigid bodies (and the engine technology you'll be developing for your coursework is largely expected to revolve around rigid bodies).

- Rigid bodies have an actual physical presence – a sphere, a cuboid, a heightmap, a puppy dog, a spaceship, etc.
- Their defining characteristic is that they **don't deform** – a balloon in the real world can be squeezed and stretched, even after it's inflated, while a balloon represented in a rigid body physics system couldn't

Physical Representation: Rigid Bodies

- Because they have physical presence, they have dimensionality – radius, width, height, etc – and thus have a centre of motion
- So they can collide
- And they can rotate (because they can experience torque)
- Computing physical properties of rigid bodies is significantly less expensive than computing physical properties of soft bodies.

Physical Representation: Soft Bodies

- Cloth, cushions, water-bombs, water itself – anything that deforms when subject to a force – is most accurately represented as a soft body.

- Soft bodies must be discretised in a physics engine – which often makes them analogous to many small bodies, interconnected in some fashion (we'll discuss this later in the module). This carries with it significant expense.
- They're used regularly in video game technology, but invariably focused on situations where application enhances user experience/immersion
- You don't use them 'just for kicks'

Physical Representation

	Velocity	Volume	Angular Motion	Deformation
Particle	Y	N	N	N
Rigid Body	Y	Y	Y	N
Soft Body	Y	Y	Y	Y

- It should be noted that a fully-featured physics system will support all these forms of object representation
- This is because all three forms have a place in many modern games

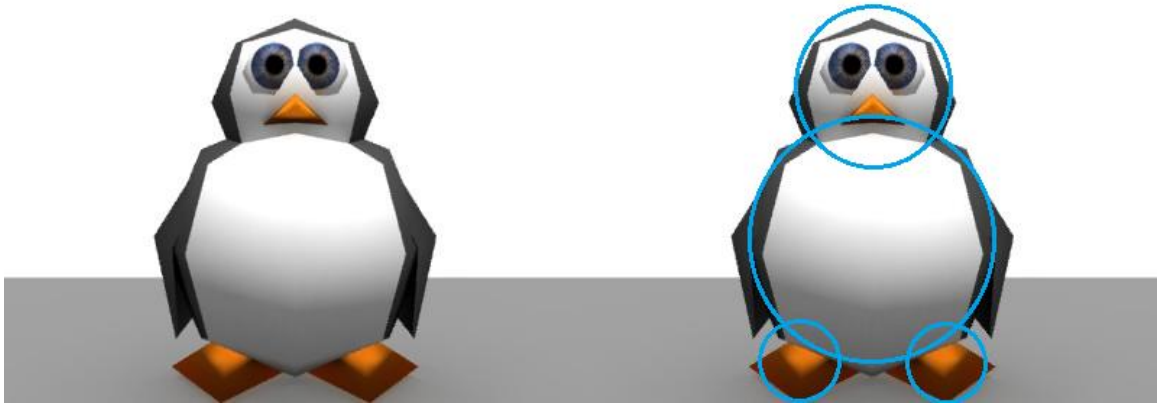
Physical Representation

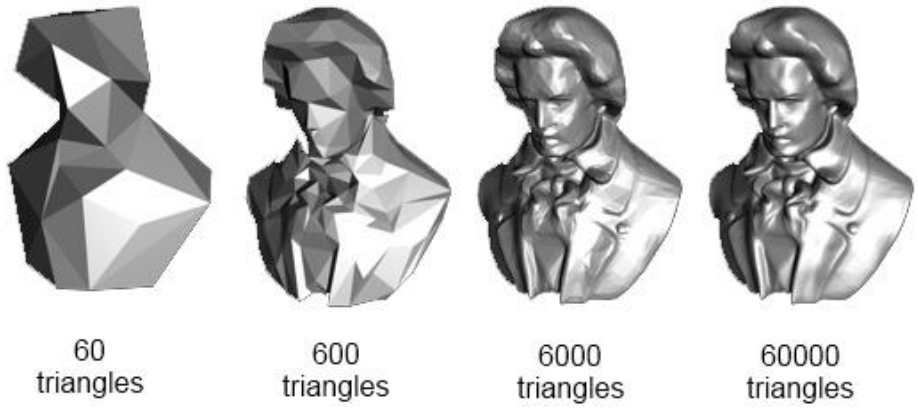
- A key point to take away from this discussion regarding physical representation of objects is that it **does not need to map to graphical representation**
- The physical representation of an object needs to be detailed enough for its interactions with the environment to appear believable – it does not need to be as detailed as the graphical model

- As much as Newtonian mechanics, this principle of ‘smoke and mirrors’ underpins game physics

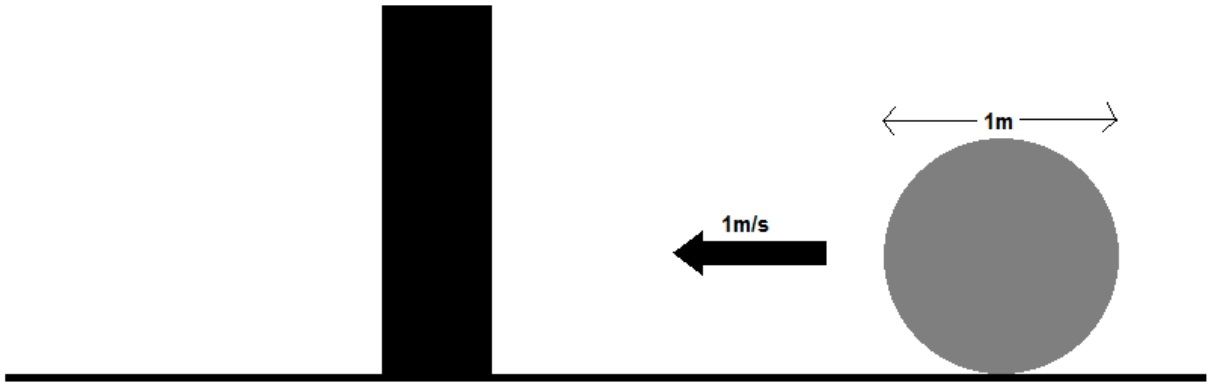
Physical Representation

- Consider the complexity of an object as a function of its number of components (faces, primitives, etc.) – this is as true in physics as it is in graphics, and to a more pronounced degree
- We reach a point of diminishing returns in physics far quicker





On Scaling



- Consider the above scenario
- What happens if our ball moves 1km/s?
- What happens if our ball moves 1 ball-width/s?
- Unified scaling

Summary

- Reviewed Linear Newtonian Motion
- Developed a mild headache

- Reviewed Angular Newtonian Motion
- Developed a migraine

- Reviewed Physical Representation of Objects
- Saw cute penguins to offset migraine/headache

Practical

- Look at the framework code
- Right now, there's a lot missing – you're going to fix that this week
- Begin by looking at how the physics engine integrates with the rest of the game engine
- Prepare for tomorrow, it's going to be a doozy