

Optional:  
Introduction to the C# language

**CONTACT ME IF YOU HAVEN'T USED A  
PROGRAMMING LANGUAGE WITH TYPE  
DECLARATIONS**

# C# can be thought of as

- C++ with seatbelts and more modern features
- A redesign of Java
  - Trying to solve a lot of the same issues as Java
  - But got to benefit from seeing what worked and didn't work in Java
- Java, but better for games
  - Math tends to be faster and easier because of user-defined value types
    - Allowing intermediate values to be stack allocated rather than heap allocated
    - So less garbage collection
- The actual history is more complicated, but from your standpoint, the above are reasonable viewpoints

# C# is a lot like C++ but without

## No **explicit memory management**

- No destructors
- No delete operator
- No memory leaks
- Just allocate objects and let the system worry about them

## No **include files**

- One source file with both interface and implementation

## No **explicit pointers**

- No \*, ->, or &
- No pointer arithmetic
- (Mostly) no worrying about reference versus value parameters
- No reference counting
- Acts just like Scheme/Racket, Java, Python, Javascript, etc.

# C# is a lot like C++ but without

## No **memory corruption**

- All operations are type-safe and checked at run-time when necessary
- No buffer-overflow attacks
- No evil errors where variables mysteriously change their values for no apparent reason

## No worrying about whether to **pass** a parameter **by value**, **reference**, or as a **pointer**

- And no corrupting memory because you chose wrong

## No weird **text-level macros**

- There is a macro facility, but it's less error prone

## No wacky Turing-complete **template system** that only 5 people in the world understand

- Templates are actually cool, but you should be grateful you won't have to learn them for this class

# And it includes a bunch of more modern features

## Type declarations

- Instead of:

```
List<int> list  
= new List<int>();
```

- Just say:

```
var list  
= new List<int>();
```

## Complicated for loops

- Instead of:

```
for (int i = 0;  
     i < foo.Length;  
     i++) {
```

```
    var item = foo[i];
```

...

```
}
```

- Just say:

```
foreach (var item in foo)
```

...

# Superficial differences between C++ and C#

# null

- Like C++, but spelled in **lowercase**
- Means a null reference

# bool and string

## bool

- Like C++
- However, can't use numbers and pointers as booleans
  - if (number)  
becomes:  
if (number!=0)
  - if (reference)  
becomes  
if (reference!=null)

## string

- Mostly like C++
- But can use strings in switch statements
- **No char \*** variables
- Strings are immutable
  - Can't say
    - mystring[5] = 'c';
- Strings and chars are full unicode

# Arrays

- Mostly like C++, but
  - Say:
    - `int [] a;`
  - Instead of:
    - `int a [];`
  - And say:
    - `int [] a = new int[5];`
  - Rather than:
    - `int a[5];`
- You can ask an array for its length:
  - `a.Length`
  - Or: `a.Count`, which works for things besides arrays (lists, hash tables, etc.)
- Array references are **checked at run-time**
  - `a[i]` will throw an exception if  $i < 0$  or  $i >= a.Length$

# Virtual vs. override

C++

```
class Parent {  
    public: virtual void M()  
    { ... }  
}
```

C#

```
class Parent {  
    public virtual void M()  
    { ... }  
}
```

```
class Child : Parent {  
    public: virtual void M()  
    { ... }  
}
```

```
class Child : Parent {  
    public override void M()  
    { ... }  
}
```

“virtual” introduces a new virtual function

# Virtual vs. override

C++

```
class Parent {  
    public: virtual void M()  
    { ... }  
}
```

C#

```
class Parent {  
    public virtual void M()  
    { ... }  
}
```

```
class Child : Parent {  
    public: virtual void M()  
    { ... }  
}
```

```
class Child : Parent {  
    public override void M()  
    { ... }  
}
```

“override” means a new method  
for an existing virtual function

# Number types

- int and float are mostly what we'll use in class
- However there are lots of others too
  - Exotic sizes: sbyte, long, short, double long
  - Unsigned numbers: byte, uint, ulong, ushort, double ulong
  - Higher precision floating point: double, decimal
- One gotcha to remember:
  - 7.5 means a double
  - **7.5f means a float**
- So:
  - float x = 7.5;
- Gives a compiler warning. It wants to see:
  - float x = 7.5f;
- C++ works this way too, but you might not have used floats before

# Basic class structure

- **Single inheritance**
  - Can't have two parent classes at once
  - Although we'll talk about interfaces, which give you a lot of the functionality of multiple inheritance
- Refer to other constructors using **this** and **base**, rather than by name
  - E.g. in code on right
    - base(x) instead of Parent(x)
    - this(0) instead of Child(0)
- No private inheritance
- No friend classes

```
class Parent {  
    public Parent(int x)  
    { ... }  
    ...  
}  
  
class Child : Parent {  
    public Child(int x)  
        : base(x) // Call Parent's constructor  
    { ... }  
  
    public Child()  
        : this(0) // Call Child's other constructor  
    { ... }  
    ...  
}
```

# The Math class

- Remember that everything has to be in a class
- So there are no global procedures, only methods
- So you have to put things like sin and cos into a bogus class whose only purpose in life is to hold sin and cos
  - Math is called a “static class”
  - Because it’s just there to hold a set of static methods
  - You can’t make an instance of it
- Math.Sin(x)
- Math.Cos(x)
- Math.Abs(x)
- Math.Min(x,y)
- Math.Max(x,y)
- Math.Log(x)
- Etc.

# Reference versus value types

- Class declaration makes a “reference type”
  - Allocated on heap
  - Garbage collected
  - Can contain any kind of data
  - Always passed by reference
- Struct declaration makes a “value type”
  - Always passed by value
  - Stored on stack or in-line in another object
    - So doesn’t need GC
  - Can only contain numbers, booleans, and other value types
- Nullable value types
  - int? means an int that can also be null

# New features

# The object data type

- All types are subtypes of **object**
  - Numbers
  - Strings
  - Classes
  - Method pointers
  - Everything
- So if you declare a variable to be of type object, you can **store any data in it**
- This is different than Java because in C#: no int/Integer distinction

# foreach

- Lets you easily iterate through a collection of objects
- Works with **arrays, lists, virtually any collection class**
- Note: can say “var” instead of *type*

**Example  
(summing elements of an array):**

```
int sum=0;  
foreach (var n in array)  
    sum += n;
```

**General syntax**

```
foreach (type var in collection)  
    statement
```

```
foreach (type var in collection) {  
    statement;  
    statement;  
    ...  
}
```

# Properties

- Let you define members of classes that **look like fields**
- But **call methods** (code) when you read them and write them
- Does the equivalent of **accessor methods** in C++, but cleaner
- Get and set **code can be left blank** if you just want the property to behave like a normal field.

Syntax

```
int MyProperty {  
    get {  
        ... code to get value ...  
    }  
    set {  
        ... code to change value ...  
    }  
}
```

Now you can use MyProperty as if it were a field

- `MyProperty = MyProperty + 1;`

# Property example

- We often want to give clients
  - **Access to the values** of fields
  - **Without the ability to change them**
- We can do that with properties by
  - Keeping the field private
  - Making a property public
  - But not giving it a set method

```
class TreeNode {  
    private TreeNode parent;  
    private List<TreeNode>  
        children;  
  
    public TreeNode Parent {  
        get { return parent; }  
    }  
    ...  
}
```

# Expression methods

- In cases where the body of a method is just “return *whatever*”
- You can remove the { }’s and return and just say:  
=> *whatever*;

```
class TreeNode {  
    private TreeNode parent;  
    private List<TreeNode>  
        children;  
  
    public TreeNode Parent {  
        get => parent;  
    }  
    ...  
}
```

# Expression methods

- And if it's in a property and the property only has a get method, then you can also remove the { get } part

```
class TreeNode {  
    private TreeNode  
    parent;  
  
    private List<TreeNode>  
    children;  
  
    public TreeNode Parent  
    => parent;  
  
    ...  
}
```

# Property example

- We might want to let the client **set** the field
- But then we have to worry about them
  - Setting the parent field
  - **Without updating** the parent's children fields
- We can fix that by giving them a **set method** that updates the children fields automatically
  - Client doesn't even have to know there is a children field
  - Much less that it's a list

```
class TreeNode {  
    private TreeNode parent;  
    private List<TreeNode> children;  
  
    public TreeNode Parent {  
        get => parent;  
  
        // Set method gets called with the new  
        // value in the variable value  
        set {  
            // Remove it from the previous parent  
            if (parent!=null)  
                parent.children.Remove(this);  
  
            // Add it to the new parent  
            value.children.Add(this);  
            parent = value;  
        }  
        ...  
    }  
}
```

# Properties

## C++

```
class A {  
    private int m_x;  
    public int GetX() {  
        return m_x;  
    }  
    private void SetX(int v)  
    {  
        m_x = v;  
    }  
}
```

## C#

```
class A {  
    private int m_x;  
    public int X {  
        get => m_x;  
        private set => m_x = value;  
    }  
}
```

# Run-time type checking

- Type casts like `(int)x` are checked automatically at run time
  - An `InvalidCastException` is thrown if `x` isn't an int
  - (or whatever you're casting it to)
- The **is** operator
  - Lets you check in advance whether a cast would work
  - Example:  
`if (x is string)`  
    `DoSomething((string)x);`  
`else`  
    `DoSomethingElse();`
- The **as** operator
  - Like C++'s `dynamic_cast`
  - Returns value cast to new type if possible
  - Or null otherwise
  - Example:  
`var s = x as string;`  
`if (s != null)`  
    `DoSomething(s);`  
`else`  
    `DoSomethingElse();`
- Pattern matching
  - The `as + if` combo above can be written more readably as:  
`if (x is string s)`  
    `DoSomething(s);`  
`else`  
    `DoSomethingElse();`

# Generic types

- Like **C++ templates**, but simpler (and more limited)
- **Classes** that are **parameterized** by other **types**
- Take one or more type parameters enclosed in **<> brackets**

Useful built in generic types:

- **List<T>**
  - A generic list of type T
  - List<int> is a list of integers
  - List<bool> is a list of booleans
  - Etc.
- **Dictionary< TKey, TValue >**
  - A hash table who keys are type TKey
  - And whose values are type TValue

# Assertions

Like in C++, but

- They're static methods
- in the class Debug
- In the System.Diagnostics namespace

```
using System.Diagnostics;
```

```
class Foo {  
    void Bar() {  
        ... do some stuff ...  
        Debug.Assert(a==b);  
        ... do some more stuff...  
    }  
}
```

# Lambda expressions

- These are in C++ too, but more complicated
- Basic syntax:
  - `(args ...) => value`
  - `(args ...) => { code ... }`
  - `() => { code ... }`
- Lambdas have **types**
  - `Func<In,Out>` is a function that takes an In and returns an Out
  - `Func<In1, In2, Out>` takes two arguments, etc.
  - `Action<In>` is a function that takes an argument of type In and returns void
  - Action is a function that takes no arguments and returns void

## Examples

- **`(int i) => i+1`**
  - Function that adds 1 to its argument
  - The int part can be omitted when the compiler can figure it out for itself
  - So: `i => i+1`
- **`(int i, int j) => i+j`**
  - Adds arguments
- `(int[] a, int i) => a[i] > 0`
  - True if ith element of a is positive
- **`(int[] a) => {  
 foreach (var e in a)  
 if (e>0) return true;  
 return false;  
}`**
  - Returns true if a has a positive element

# Lambda example from homework

```
private void WriteBracketedExpression(  
    string start,  
    Action generator,  
    string end)  
{  
    Write(start);  
    indentLevel++;  
    NewLine();  
    generator();  
    indentLevel--;  
    NewLine();  
    Write(end);  
}
```

```
private void WriteList(ICollection list)  
{  
    if (list.Count == 0)  
        Write("[ ]");  
    else  
        WriteBracketedExpression(  
            "[ ",  
            () =>  
            {  
                foreach (var item in list)  
                {  
                    WriteObject(item);  
                    Write(", ");  
                }  
            },  
            " ]");  
}
```

# Attributes

- C# lets you to annotate classes, methods, and fields with **custom metadata** called attributes
- You can add an attribute to a declaration by putting **[attribute]** immediately before it.
- Unity uses attributes extensively to let you customize its behavior

# Examples

- `public float FooBar;`
  - Unity will save the FooBar field to disk, and also let you edit it in the inspector.
- **[HideInInspector]**  
`public float FooBar;`
  - It's still public, and still serialized, but Unity won't display it in the editor.
- **[NonSerialized]**  
`public float FooBar;`
  - Unity will leave it alone completely;
- **[MenuItem("MyMenu/Log something")]**  
`static void LogSomething() {  
 Debug.Log("Bla bla bla")  
}`
  - Unity will add LogSomething to the menu in the editor

**things Ian uses a lot in his code**

# String interpolation

- "The sum of the array is {sum}"
  - Is a normal string
  - It's a fixed set of characters, including the {sum} part
- `$"The sum of the array is {sum}"`
  - Note the \$ at the beginning
  - Is really a call to the `string.Format` method
  - It substitute the value of sum in for the substring `{sum}`

# Tuple types

- Convenient way to make types without having to write a declaration for them
  - int is an integer
  - (int, int) is a tuple with two integers in it
  - (int, string) is a tuple with an integer followed by a string
- Useful for returning multiple values from a method
  - Just return one tuple

```
public (int, string) Method() {  
    ... some code ...  
    return (i, myString);  
}
```

```
public void OtherMethod() {  
    var (n, s) = Method();  
    // now n is the number  
    // and s is the string  
}
```

# Fancy tuple types

- **Dictionary<int, string>** is a hash table mapping integers to strings
- **Dictionary<(int, int), string>** is a hash table mapping pairs of ints to strings
- **Dictionary<int, (string, string)>** is a hash table mapping integers to pairs of strings
- It's just easier than having to make a separate class definition for pairs of integers or strings

# Iterators

- All **collection types** in C# have interfaces for iterating over their elements
  - We'll talk more about these because Unity abuses them to interesting effect
- They give you a way to
  - **Ask for one element at a time**
  - **Generate one element at a time**
- They get used to implement foreach
- We'll talk more about them later

# Linq (“language-integrated queries”)

- A library **built on top of iterators** to let you do
  - Database access
  - Scheme/racket-style map/fold/filter operations
    - But they get called Select/Aggregate/Where because that's what they're called in SQL
- **Easy way to write loops** over collections
- But you have to be willing to write  **$\lambda$  expressions**
  - Because Select/Aggregate/Where/etc. take functions as their arguments

# Example

- Suppose you have a list of strings and you want to compute how many characters are in all of them
- You could write it this way:
  - `var sum = 0;  
foreach (var str in list)  
 sum += str.Length;`
- But instead you could just say:
  - `list.Select(str => str.Length).Sum()`
- How to read this:
  - **List** generates a series of strings
  - **str => str.Length** generates the length of a string
  - **.Select(str => str.Length)** generates a series of their lengths
  - **.Sum()** adds them all up and returns one number

# More examples

- How many zeros are there in a list of numbers?
  - `list.Where(e => e == 0).Count()`
- Give me an array of just the positive numbers from the list
  - `list.Where(e => e>0).ToArray()`
- Change an array of type object (`object[]`) that happens to contain numbers into an `int[]`:
  - `array.Select(e => (int)e).ToArray()`
  - Or just: `array.Cast<int>().ToArray()`

# More examples

- Are any/all the elements positive?
  - `list.Any(e => e>0)`
  - `list.All(e => e>0)`
- What's the first positive number in the list?
  - `list.First(e => e>0)`
- What's the smallest positive number in the list?
  - `list.Where(e => e>0).Min()`
- How many numbers are in the list, ignoring duplicates?
  - `list.Distinct.Count()`



# CS 376

## Game design and development



# What's this course about?

- Game **programming**
    - Less about game design
    - Very little about art (3d modeling, etc.)
  - Focus on
    - **General architecture**
    - **Background** concepts
  - Less on **genre-specific** or game-specific code
- Example topics
- Game **architecture**
    - Component-based
    - Event-driven
  - **Math** and **physics**
  - Real-time **rendering**
  - **Serializing** media assets
  - Polling **controllers**
  - **Performance** issues

# Will we make games?

- Yes, about **one a week**
- **Simple** games to teach specific concepts about game programming
- We will mostly be designing them for you
  - But if you take **next quarter's course**, you'll be designing your own games



# Class format

- **Flipped classroom**
  - **Lectures** prerecorded: listen to them in advance
  - **Class sessions** will be for discussion and exercises
    - You'll work on the exercises in your **troupe**
  - Regular on-line **microquizzes**
- **Weekly assignments**
  - Machine graded
- **Peer grading**
- **Microquizzes**
  - On canvas
  - Roughly one per lecture deck
- **Quizzes**
  - Real, paper quizzes graded by actual humans
  - In class, one hour each, three of them

# Your troupe

- In-class exercises will be done in groups of 3-4 called **troupes**
  - We'll form troupes this weekend
- We'll form troupes this week
- If there are problems, you can **change troupes**
  - Contact Ian

# tools and materials

# Languages and tools

- Game engine: **Unity**
  - Cross-platform
  - Free, provided you don't put any of your games on a Northwestern server
  - We will standardize on version 2021.3.39f1
    - See syllabus for download link
- Language: **C#**
  - Like Java but different
  - You may not use Javascript



# Languages and tools

- IDE (installed by Unity)
  - **Visual Studio**
    - Most powerful, but windows only
  - **Visual Studio for Mac**
    - Cross-platform
- Source control
  - **GitHub Desktop**
  - Feel free to use a different git version if you want

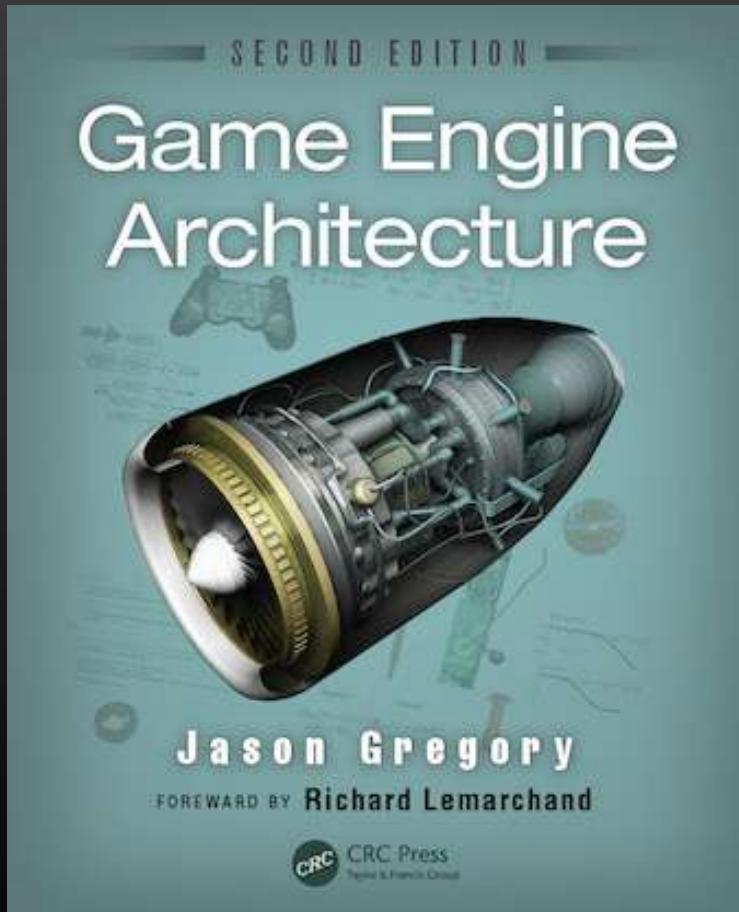


# Buy a game controller!

- The textbooks are optional, but the controller isn't
- You can use **anything that works with your machine**
  - But it needs to have both **buttons and at least one joystick**
- If you have a console, the controller for it probably works with your laptop/desktop

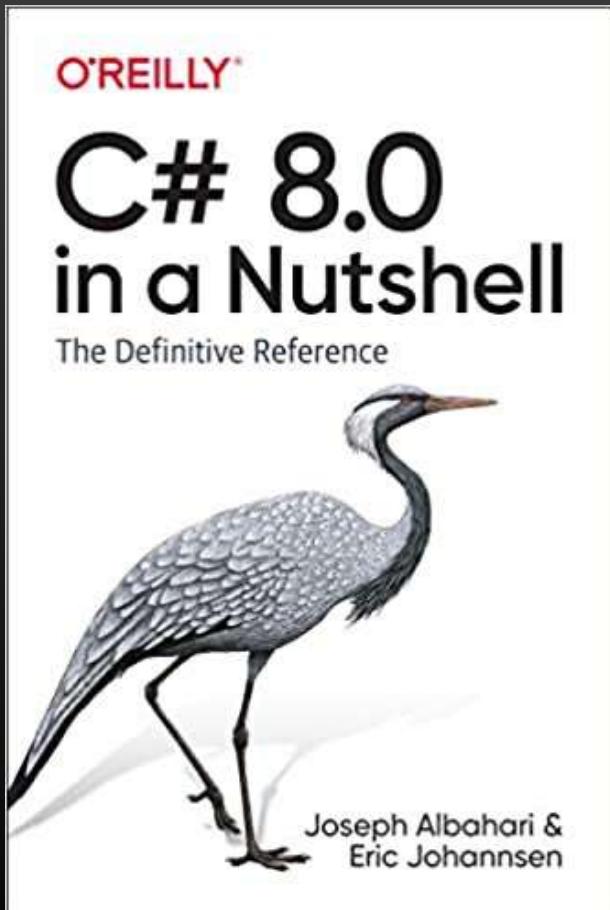


# Primary text (optional, but great)



- Encyclopedic
- Covers way more than we will cover
- But a really good, up-to-date reference on game internals
- Recommended if you're serious about games

# Optional book



- If you're not familiar with C# or Visual Studio
- Feel free to use any other C# book
  - Avoid C# 10
  - If you get a C# 10 book, it's fine but understand that all their discussion about nullable reference types is irrelevant for Unity
- You are responsible for teaching yourself C#
  - We won't spend class time on it

We will not teach you C# or Unity

You need to **self-study**

grading

# Grading

- **Quizzes** (40%)
- **Assignments** (writing games, 40%)
  - Roughly weekly individual assignments
  - Peer graded
  - Bigger, group project game toward the end of the quarter
- Peer **grading** (10%)
  - 10% of your grade is for doing the grading of other people's assignments
  - Anonymous
- **Microquizzes** (10%)
  - Roughly one per slide deck

# Quiz schedule

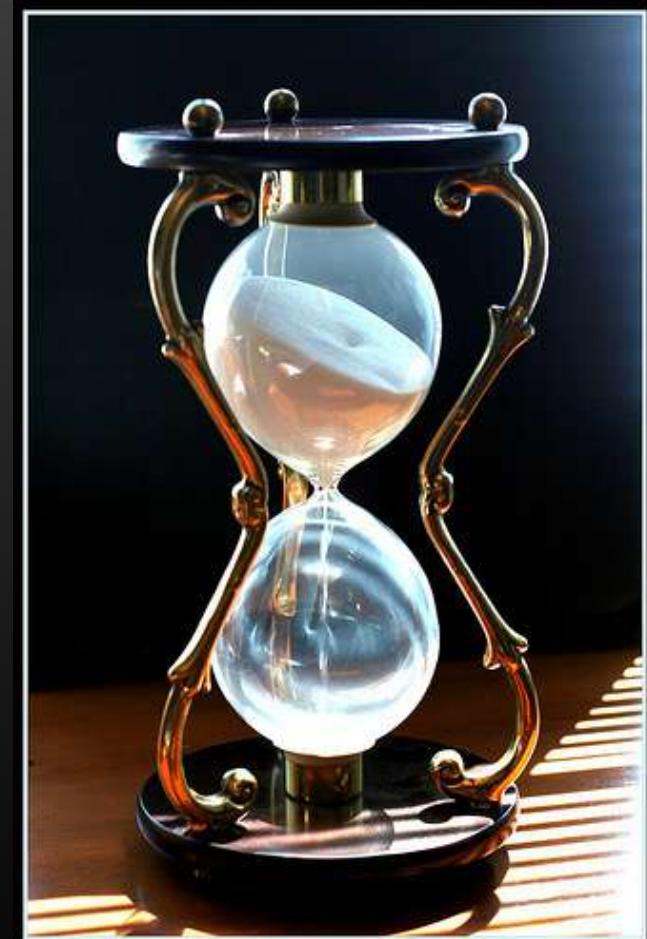
- Q1: Wednesday, **October 19**, in class
- Q2: Monday, **November 21**, in class
  - Monday before Thanksgiving
- Q3: Thursday, **December 8, 7-9pm**
  - We have no control or flexibility over this
  - We are not allowed or able to people take it early
  - Plan your plane flights accordingly

# Quiz structure

- Quizzes will stress concepts, not memorization
- Closed book, closed note
  - This is only because I've always found that student performance decreases for open book tests
- However, we will provide glossaries along with the tests listing:
  - Unity API calls
  - Important equations
- Kinds of questions that are fair game include:
  - Write a short procedure to do something
  - Explain why this short procedure is broken
  - Give me the math to do something
  - Tell me which of this list of things have some specific property

# Late policy

- The late policy is there are **no late submissions**
  - i.e. **no extensions**
  - Sorry; this is the problem with **peer grading**
- If you're not done, **submit what you can** when the deadline comes
- Important: if something terrible happens to you (e.g. you or a loved one are hospitalized), come and talk to me and we'll work something out



# Tentative list of topics

## “zero”-d games

- Basic game architecture
- The Unity architecture
- Input devices
- Messages, event handling, and multitasking
- Scalar math
- Game design and MDA

## 2d games

- Linear spaces
- Projective spaces
- Sprite rendering and animation
- Game AI
- Physics
- Collision detection
- Networking
- Audio

# Tentative list of topics

## 3d games

- Homogeneous coordinates
- Prospective projection
- Triangle-based rendering
- Representation of 3D rotation
- Transforms and kinematic chains
- Performance issues in rendering

# installing software

# Installing on Windows

- Go to the download link in the syllabus
- Choose 2021.3.1of1 LTS
  - Download it and run it
  - When it gives you a lot of checkboxes, make sure the one marked “Visual Studio Community” is checked
- Open Unity
  - Go to Edit>Preferences
  - Select External Tools
  - Under External Script Editor, choose Visual Studio (or Visual Studio for Mac, whichever is listed)

# Installing on Mac

- Apologies if you have an M1 chip
- Go to the download link in the syllabus
- Choose 2021.3.1of1 LTS
- Download it and run it
  - When it gives you a lot of checkboxes, make sure the one marked “Visual Studio for Mac” is checked
  - Let it do its thing
- Open Unity
  - Go to Unity>Preferences
  - Select External Tools
  - Under External Script Editor, choose Visual Studio (or Visual Studio for Mac, whichever is listed)
  - Close the dialog box

# Installing GitHub Desktop

- Go to [desktop.github.com](https://desktop.github.com)
- Download it and run the installer
- You'll need to make a github account if you don't already have one
  - If you don't want to, you can use other versions of git, that's fine
  - GitHub Desktop is just nice and simple
- Start GitHub Desktop
  - Choose File>Clone repository
  - Choose URL
  - Enter `ianhorswill/CS376-student`

# Trying unity

- Go to the CS376-student folder
- Find 2D Particles/Assets/Rigid body.unity
- Open it
  - Unity will think for a long time
- Press the ▶ button in the center of the top of the window
- This is a simple soft-body physics simulator trying to simulate a rigid body
  - Try grabbing things and moving them around
  - Don't be surprised if things go crazy; physics is hard
  - We'll talk a lot more about that later in the quarter
- Press the ▶ button again to stop it

# Simulation and lifeworlds

# This is a weird lecture

- Not representative of the rest of the course
- Lots of philosophy
- But it brings up themes that will matter later in the course when we talk about
  - Game design
  - Game development (writing the actual simulation)

# Video games are often simulations



# Simulations have a standard structure

- **Data objects** that represent **world objects**
  - **State** variables to represent the state of the simulated world object
  - **Update** methods to compute new state from
    - Current state
    - States of other objects
    - Input from user
- Mechanisms for handling **object interactions**
  - Collisions
  - Attraction
  - Attacks

# Simulations have a standard structure

```
loop forever {  
    clear the screen  
    draw all the objects  
    update all the objects  
    handle all the pairwise object interactions  
}
```

# Is chess really a medieval warfare sim?



# Is monopoly really a capitalism simulator?



# Is Scrabble an anything sim?



# Any yet almost all games look like this

```
loop forever {  
    clear the screen  
    draw all the objects  
    update all the objects  
    handle all the pairwise object interactions  
}
```

# Mimesis

- Most arts are imitative or **mimetic**: they are **imitations of nature**
  - There are always **exceptions**
    - Painting is a mimetic art
    - But there are still **non-figurative** paintings
- The process of imitation is called **mimesis**
- Video games are an imitative art
  - But they're **more like theater** than painting
  - They're **processes** that unfold in time

# Distortion

# Distortion

- Plato (*The Sophist*) pointed out that there are **different kinds** of imitative art
  - **Faithful reproduction**
  - **Intentionally distorted** in order to appear correct
- Of course, in practice, this is a **spectrum**; no art is a perfect reproduction



# Realism and distortion in games

- Games are usually highly distorted
  - Made **easier**
    - In real life, you have one hit point
  - **Simplified**
    - You don't want SimCity to simulate the rusting of individual sewer valves
  - Made more **real seeming**
    - Random number generators are often tweaked to feel more random than actual randomness



# Sim sub-genres versus arcade

- “**Sim**” games
  - **Accurate simulations** of something
  - Sim racers
  - Sim flight simulators
- “**Arcade**” games
  - **Distorted simulations** intended to be more engaging
    - Especially for casual players
  - Arcade racers
  - Arcade flight sims

# Gran Turismo 6



# MarioKart 8

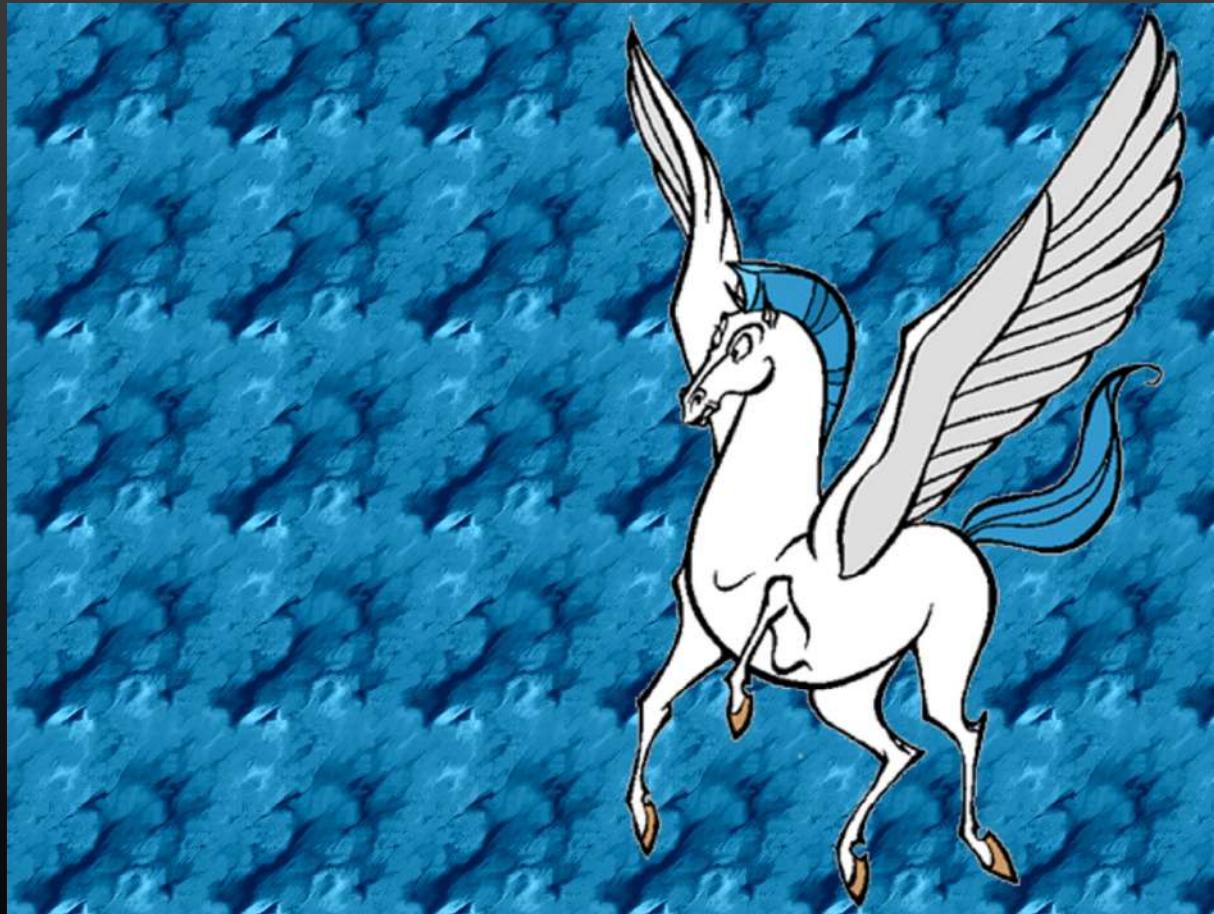


Of course,  
**realism isn't binary**

# Is this realism?



# How about this?



Or this?



# No simulation is ever perfect

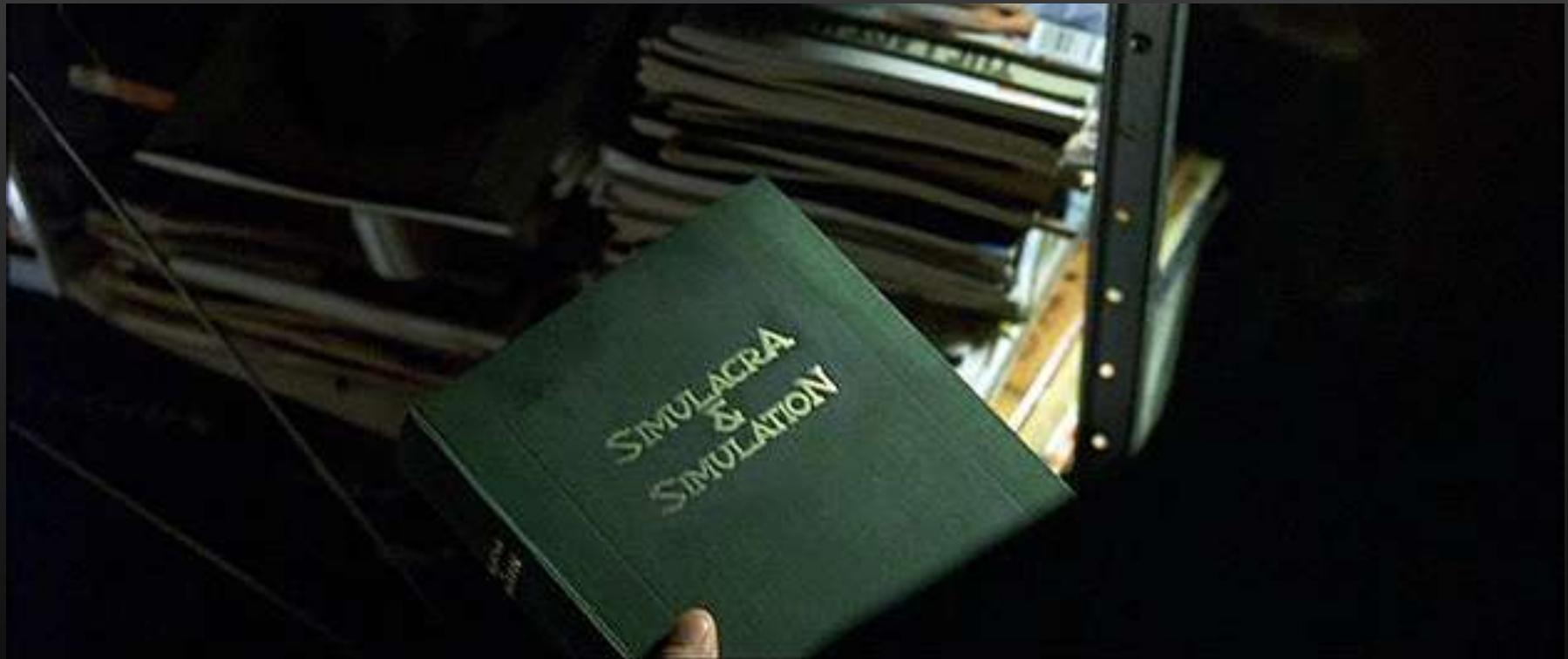
- We **don't actually know** what reality is
  - People writing “accurate” physics sims in 1400, 1800, and today would write sims with very different behavior
- Our simulations couldn’t include abstractions like **tables, chairs, and humans**
  - They’d just have **quarks** and other subatomic particles

# Representation and abstraction

But wait, if tables don't technically exist, are we **living in a fantasy?**

- No, we're thinking in terms of representations that are **useful approximations**
- But it does mean that a lot of our thinking is (necessarily) **disconnected from the details of reality**

# *The Matrix*



# Hyperreality

Jean Baudrillard and Umberto Eco:

- Disneyland's Main St. is a fantastic, hyper-realistic simulation of old-time America
- But no street in America ever actually looked like that
- It's a simulation of a romantic ideal that never existed, a simulacrum, in his terminology



# Lifeworlds

# Lifeworlds

- The **world you live in** as distinct from physics
- The **business** world
- The **academic** world
- The **Poker** world
- The **music** world
- The **literary** world

# Elements of a lifeworld

- **Care**

A lifeworld comes prepackaged with fears, goals, aspirations, and other values that it expects you to adopt

- Staying alive
- Getting promoted

- **Equipment**

A lifeworld is populated with objects, people, forces of nature, and so on, that all relate back to your concerns

- **Actions**

It provides you with the potential for meaningful action

- **Consequences**

It specifies the meaningful consequences of your possible actions

# Games are lifeworlds

- They're like **simulations of lifeworlds**
- They come **prepackaged with care**
  - Players make **meaningful choices** in them
- But there might not actually be a thing being simulated, so perhaps the simulation **just is** the lifeworld

# MarioKart 64 as a lifeworld

- **Care**  
You want to get across the finish line before the other players
- **Equipment**  
The track, your Kart, power ups, the finish line, other players
- **Actions**  
Acceleration and braking, activating a powerup
- **Consequences**  
The rules of the simulation



# Games as formal systems

# Games as formal systems

- Take the **rules for Monopoly**
- Replace all the **capitalism words** like “property”, “buy”, “pay”, “collect”, “rent” with their equivalent words in the **Akkadian language of ancient Babylon**
- Teach the game to someone who
  - Speaks English
  - Hasn’t studied any semitic languages
- They are playing monopoly
  - But have no idea it’s about capitalism
- These are the raw **game mechanics**, divorced from the game’s evocative fiction

# Reskinning a game

- Take Dungeons and Dragons™
- Change all **fantasy terms** ⇒ **Star Wars™** terms
  - Magic ⇒ force
  - Gold piece ⇒ credit
  - Horse ⇒ flying car
  - Orc ⇒ Gamorrean
  - Bow ⇒ laser carbine
  - Healing potion ⇒ bacta
  - Hit point ⇒ hit point
- The mechanics (and so formal system) is **exactly the same**
- And yet it will **seem different** to the players

# Flavor text

- Flavor text is **evocative terminology**
  - Backstory/lore
  - Setting-appropriate terminology for mechanical
- But has **no mechanical effects** on the game
- Games are roughly **mechanics + flavor text**

# Flavor assets

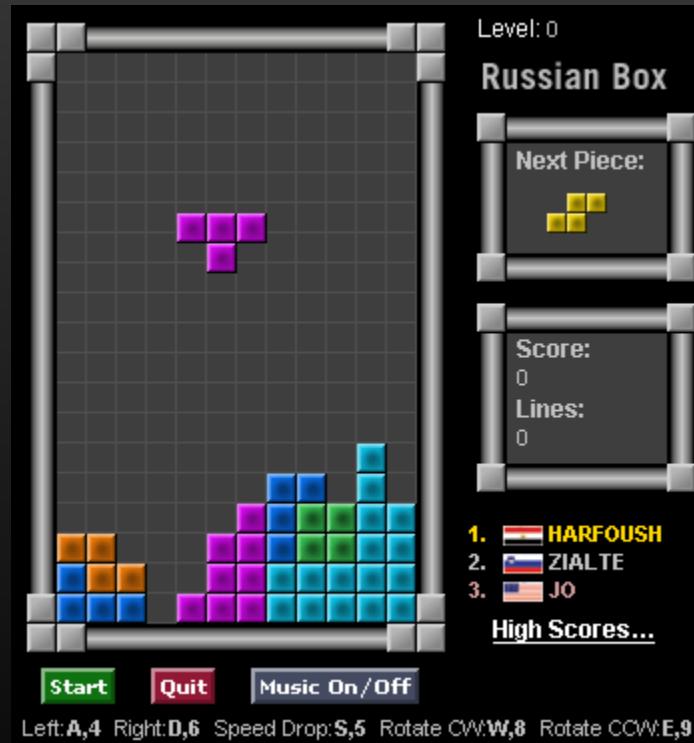
- Most games could be made with **brightly colored geometric shapes** for all the on-screen objects
  - Characters
  - Walls
  - Power-ups
- But in practice, we usually use **assets that suggest, the world and emotions** we're trying to evoke
- These are also flavor
  - They have no mechanical effect
  - But they're important to the game experience

game = formal system + flavor

# Mechanics vs. flavor

- Different games put different emphasis on the two
- **One isn't better or more important than the other**
- But it's important to cultivate the skill of **separating the two in your mind**
  - What are the mechanics of the game, divorced from the flavor?
  - How do the mechanics serve or interfere with the fiction of the game?
  - How does the flavor serve or interfere with it?
  - What would a reskinned version of the game look like?

# Some games have very little flavor



# Some games are mostly flavor



*nū*

# Game architecture

# Digital games

The core of a typical digital game is

- A **simulator** that animates game world frame by frame
- A database of **assets**
  - Game levels
  - Sprites (2d images)
  - 3d models
  - Shaders
  - Animations
  - Sound files
  - Configuration information
  - Saved games

# Asset databases

# Assets

- Roughly, **anything that isn't code**
- Typically stored as **discrete source files**
  - Each level, model, sound file, etc. is a separate file on the hard drive
  - In some cases, each might be its own folder
- These have to be **read in at runtime**
  - Called **deserialization** (transforming a series of bytes into a complex data structure)
- Or sometimes written out again
  - Called **serialization** (transforming the data structure back into a linear sequence of bytes)

# Asset databases can be huge

- FFVI Remake is ~**100GB**
- Storage devices are **relatively slow**
  - NVMe drive: 4 GB/sec (0.5 minutes to transfer FFVI)
  - Hard drive: 0.1 GB/sec (15 minutes)
  - Blu ray drive: 0.05 GB/sec (30 minutes)
- In practice, it's **much slower** than that
  - **Deserialization** software must transform the data into the format it's stored in RAM, and that can be complicated
  - For example: **shaders are often JIT compiled** for your specific graphics card when they're loaded from disk

# Asset databases can be huge

- Assets are often **much larger than RAM**
  - So you have to load just the stuff you need now
  - And load other stuff as you need it
  - But that can interrupt gameplay
- Platform TRCs (essentially app store requirements) **limit the number of seconds** you can spend loading a level
  - So you may not even be able to load a whole level in at once
- So you may need to dynamically **load assets in the background** during gameplay

# Performance issues

- Load data in **separate thread** from main thread
  - Let the screen continue to update during loading
  - But care must be taken to make sure there aren't **race conditions** (we'll talk about these later)
- "**Streaming**": load data in the background while the player is playing
  - Still must **load enough in advance to start the level**
  - And **predict what you will need** in advance so you can load it proactively
- Package all assets for a level into **one file**, to reduce seek time
  - Put the **stuff you need first at the beginning** of the file
- If CPU faster than disk channel, then **store compressed version** on disk

# Build-time asset processing

- **Format** conversion
  - Translate from e.g. Adobe format to an efficient format for the game
- **Precompute** expensive calculations
  - E.g. lighting, sample quaternion splines
- **Pack** assets together
  - Multiple sprites => one texture file
  - All assets for a given level in one file
- **Sort** assets to improve streaming performance

# Format issues

- Adobe and Maya **file formats are designed for flexibility**
  - Support for all kinds of features like NURBS that wouldn't be used in a game
  - Textures can be any kind of image file
  - Meshes may be represented in XML or some other textual format
  - Lots of extra information not needed for displaying the asset
- **GPUs require very specific formats**
  - No jpeg compression
  - Have to be a power of 2
  - Models have to be triangle meshes in a very specific format
  - Coordinates have to be single-precision binary floats
- The **build pipeline** will typically
  - Strip data that won't be used at runtime
    - E.g. surface normal if you aren't using lighting calculations
  - Convert media assets from text to binary formats
  - Get the data as close as possible to the native GPU format

# Simulation

# Simulations have a standard structure

- **Data objects** that represent **world objects**
  - **State** variables to represent the state of the simulated world object
  - **Update** methods to compute new state from
    - Current state
    - States of other objects
    - Input from user
- Mechanisms for handling **object interactions**
  - Collisions
  - Attraction
  - Attacks

# Simulations have a standard structure

```
loop forever {  
    clear the screen  
    draw all the objects  
    update all the objects  
    handle all the pairwise object interactions  
}
```

# Representing game objects

# A typical game object

- **Pose** information  
Where the object is and how it's pointed. Often represented by a transform matrix.
- **Physics** state  
Momentum, angular momentum, etc.
- **Gameplay/AI state** information  
Hit points or whatever
- Render **geometry**  
Shape(s) to be drawn on screen, represented as a set of polygons
- Render **material(s)**  
Color and texture of the object, specified by shaders, texture maps, color information, etc.
- **Collision** geometry  
Simplified geometry used for determining contact between objects

# Game objects in an OOP framework

- **Separate classes** for different kinds of data
- Game object's class is **combination of relevant parent classes**, e.g. (in C++):

```
class Monster : public GameObjectBase,  
Renderable, Collidable, PhysicsObject
```

- Then you initialize the fields of the different classes appropriately

# Problems with the OOP approach

- Requires **multiple inheritance**
  - Doesn't work well with Java or C#
- Multiple inheritance has **issues**
  - What if both PhysicsObject and Renderable have fields called "enabled"?
- **Explosion of classes**
  - You need to declare a different class for every possible combination of parent classes
  - In the worst case, you end up having a **different class for every object**

# Component-based architecture

- There's **just one game object class**: GameObject
- It has **one field**: a list of **components**
  - Okay, maybe it has more too, but mostly it's just a list of components
  - In Unity, it also has a name, a tag (another string), and a list of child objects
- **Class hierarchy** of components
  - Renderable mixin class => **Renderer** component class
  - Collidable mixin class => **Collider** component class

Wait, if everything is all  
**one class**

**How do you know** whether a game object is a **monster or the player?**

# Type testing in component-based systems

- Objects are **implicitly typed** by the components they contain
- To test if something is a monster, check if it contains a **monster-related component**

```
if (maybeMonster.GetComponent<MonsterAI>() != null)  
    Attack(maybeMonster);
```

# Serialization

# Initializing game objects using code

```
class Monster : Whatever {  
    public Monster(int lots, int of, int arguments) {  
        this.field1 = value;  
        ...  
        this.fieldn = value;  
    }  
}
```

```
void StartGame() {  
    new Monster(args....);  
    new Monster(args ...);  
    new Player(args...);  
    ....  
}
```

# Initializing GameObjects from files

```
class Monster : Whatever {  
    public Monster(args ...) { ... }  
    static public FromFile(string path) {  
        ... open the file ...  
        ... read the data for the constructor args ...  
        return new Monster(args ...)  
    }  
}
```

# Saving GameObjects to files

```
class Monster : Whatever {  
    public Monster(args ...) { ... }  
    static public FromFile(string path) {  
        ... open the file ...  
        ... read the data for the constructor args ...  
        return new Monster(args ...)  
    }  
  
    public void ToFile(string path) { ... }  
}
```

# This is usually called **serialization**

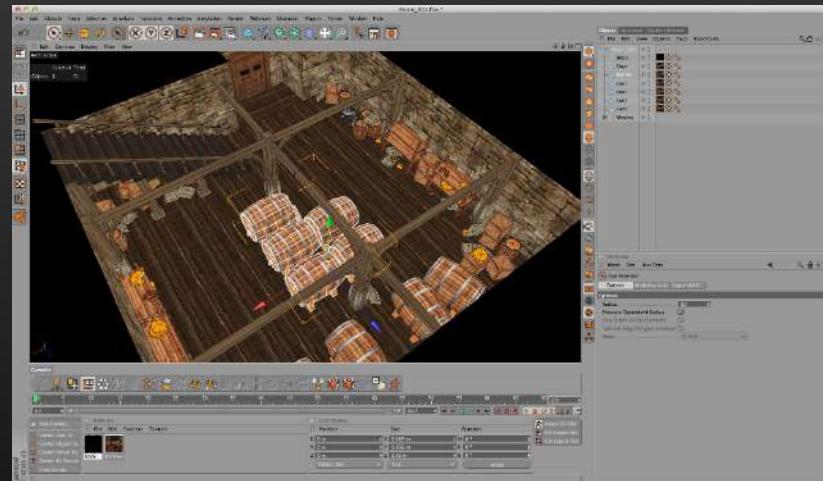
```
class Monster : Whatever {  
    public Monster(args ...) { ... }  
    static public Deserialize(string path) {  
        ... open the file ...  
        ... read the data for the constructor args ...  
        return new Monster(args ...)  
    }  
  
    public void Serialize(string path) { ... }  
}
```

# Serialization

- **Serializer**
  - Converts game object to a **stream of bytes**
  - Can be saved to a **file** or transmitted over a **network**
- **Deserializer**
  - Converts stream of **bytes back into the game object**
  - Or **updates fields** in an existing game object (for network updates)

# Level editors

- **Compiled with class definitions** from game
- **Deserializes** game objects from disk files
- Lets user **edit** them
- **Serializes** game objects back to disk file(s)



# A basic serializer

- Manually write serializers for **primitive types** (e.g. int)
  - E.g. write out the bytes of the int
- Every class has a Serialize **virtual method**
- To serialize an object
  - **Recursively serialize** its fields
- Deserialization is the same, but in reverse

```
class A : Whatever {  
    Type Field1;  
    Type Field2;  
    ...  
  
    public override void  
    Serialize(Stream out) {  
        Field1.Serialize(out);  
        Field2.Serialize(out);  
        ...  
    }  
}
```

# What are some **problems** with this approach?

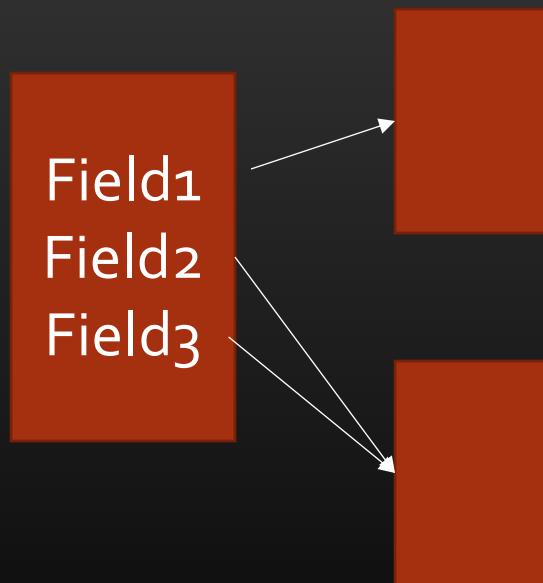
- Manually write serializers for **primitive types** (e.g. int)
  - E.g. write out the bytes of the int
- Every class has a Serialize **virtual method**
- To serialize an object
  - **Recursively serialize** its fields
- Deserialization is the same, but in reverse

```
class A : Whatever {  
    Type Field1;  
    Type Field2;  
    ...  
  
    public override void  
    Serialize(Stream out) {  
        Field1.Serialize(out);  
        Field2.Serialize(out);  
        ...  
    }  
}
```

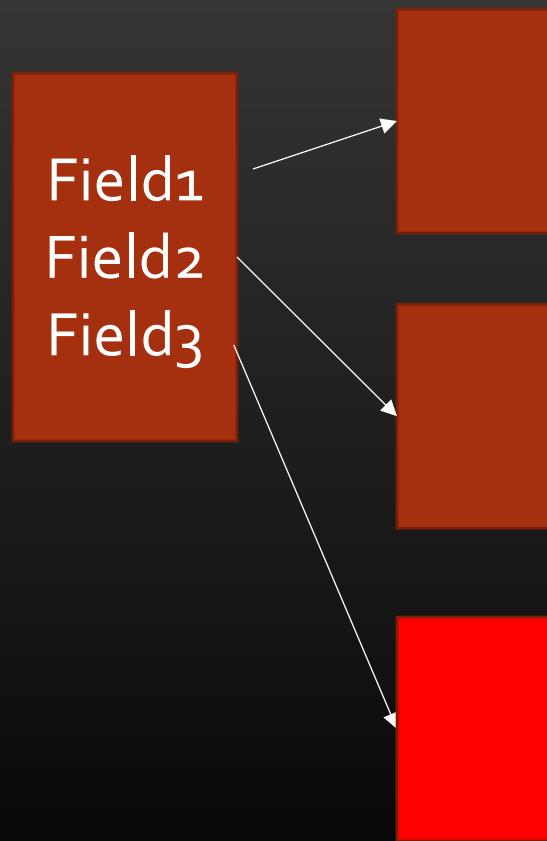
# Problem 1: graphs

- We implicitly assumes the object and its fields formed a **tree**
- We're basically doing a **depth-first walk** of it
- But we're doing the tree version of the DFS, **not the graph version**

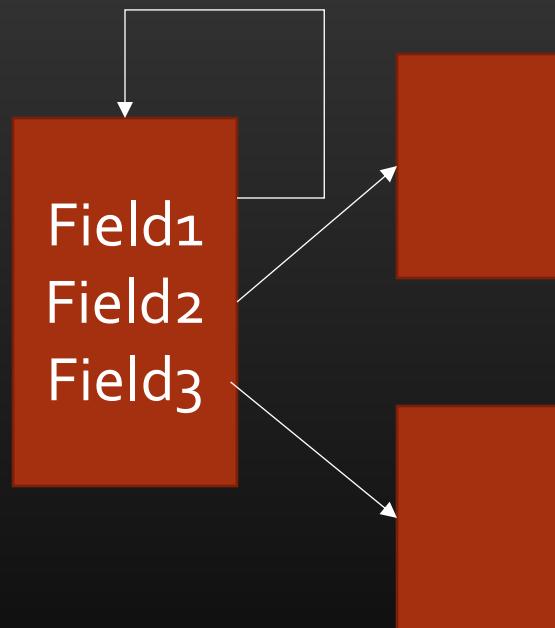
# Serialize this object



# And now deserialize it



# And what about this?



How do you fix it?

# Handling DAGs and cycles

When **writing** objects

- Keep a **hash table**
  - Of the objects you've already written
  - Along with a **serial number**
- When you want to write an object
  - If it's in the table
    - Write a **magic code** and the object's serial number
  - If it's not
    - Make a new serial number
    - Add it to the table
    - Write a **different magic code** and the serial number
    - Recursively write the fields as normal

# Handling DAGs and cycles

When **reading** objects

- Keep a **hash table**
  - Of the objects you've already read
  - Along with their serial numbers
- When you want to read an object
  - **Read the magic code**
  - If it's the **"already in the table"** code
    - Read the object's serial number
    - Look it up in the table
  - If it's the **"new object"** code
    - Read its serial number
    - Recursively read the fields as normal
    - Add it to the table

# Problem 2: subclassing

- What if a field can be filled with **values of different types**?
- Solution:
  - Keep a **hash table of deserializers** for the different types
  - Before writing an object, **write its type**
  - When reading an object
    - First **read its type**
    - **Look up** the type's deserializer in the hash table
    - Call it

# Rendering

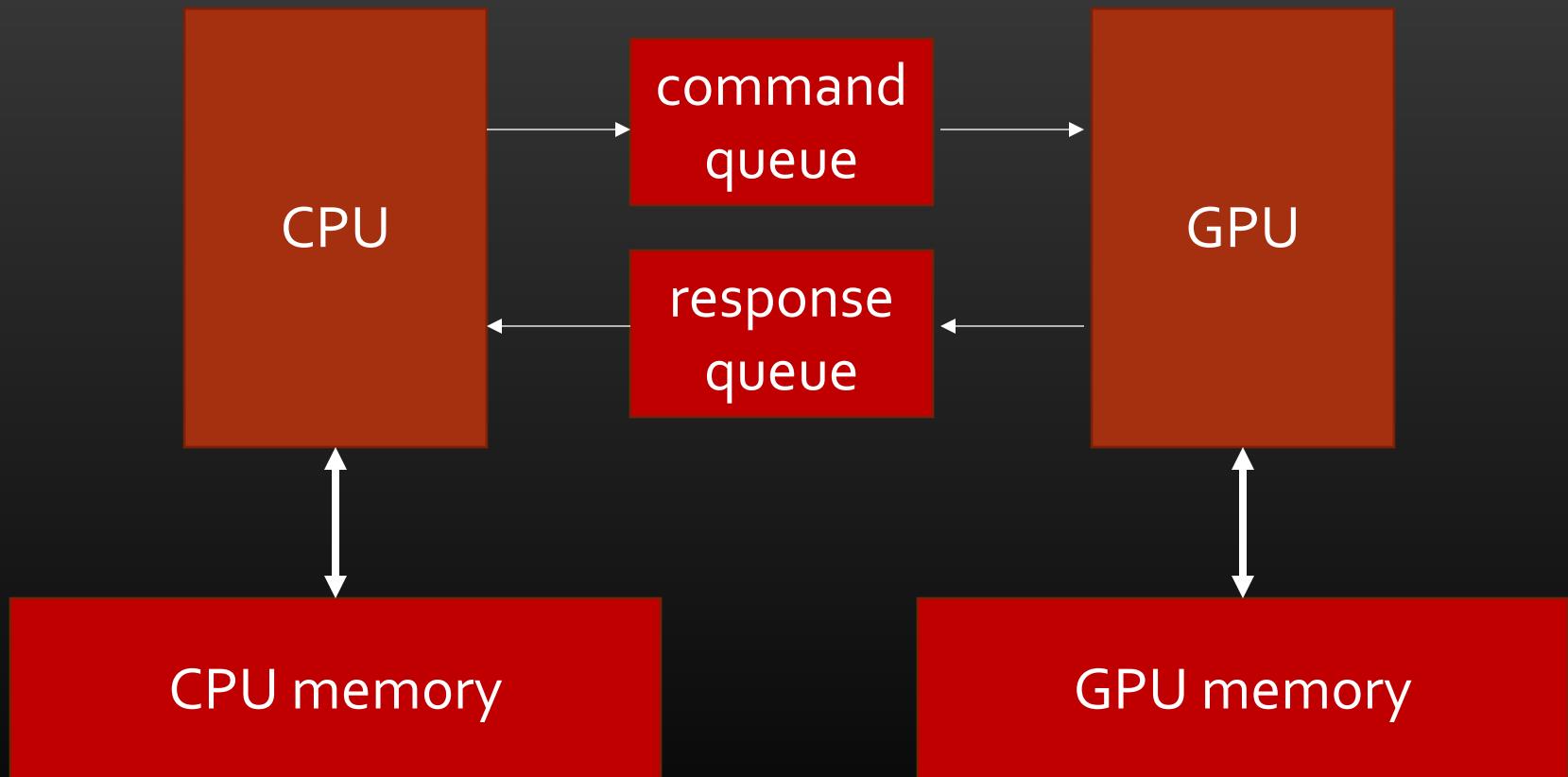
# Rendering

- Rendering is the process of **drawing** the next frame
- It mostly involves **drawing all the objects**
- Drawing an object involves
  - A **mesh**: the triangles to draw
    - For a sprite, this is two triangles arranged in a quad (rectangle)
  - A **shader**: the programs to run in the GPU to do the drawing
  - A set of shader **parameters**
    - Coordinate **transform**
    - **Color**
    - **Texture** map
    - **Lighting** information

# Graphics Processing Units (GPUs)

- Your GPU (usually) has its own **separate memory**
  - Frame buffer (the **screen**)
  - Shaders
  - Meshes
  - Parameter blocks for shaders
- The GPU is really **fast**
- **Communication** between CPU and GPU is really **slow**
- So we try to **minimize communication**
- And let them run independently
  - In **parallel**
  - **Asynchronously**

# CPU/GPU architecture



# Controlling the GPU inefficiently

- **Use shader** 7, pass number 0
- **Use mesh** number 7
- **Set the transform** of parameter block 4 to XXX
- **Set the color** of parameter block 4 to YYY
- **Use parameter block** 4
- **Run** the shader
- **Use shader** 7, pass number 0
- **Use mesh** number 8
- **Set the transform** of parameter block 4 to ZZZ
- **Set the color** of parameter block 4 to YYY
- **Use parameter block** 4
- **Run** the shader

# Controlling the GPU efficiently

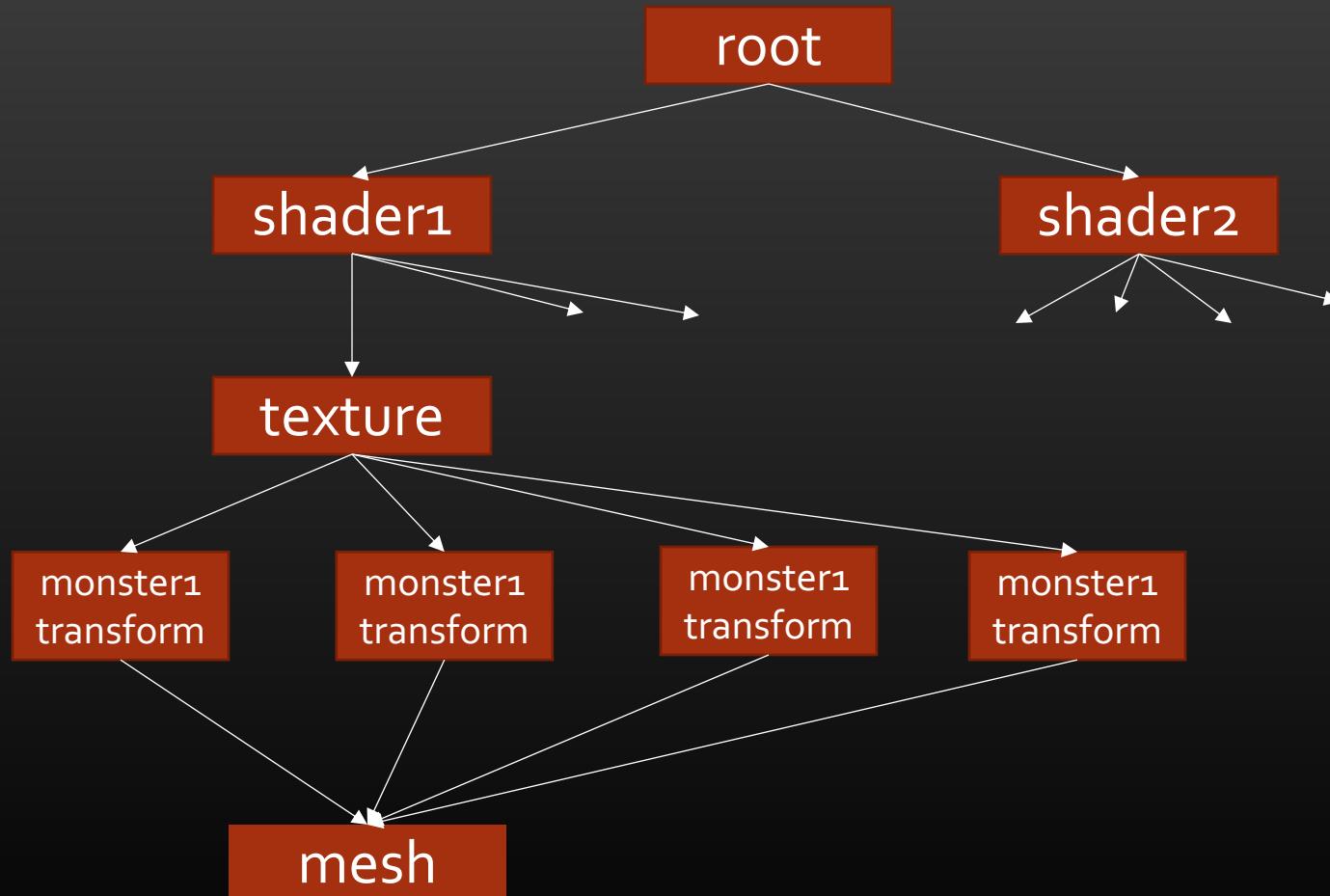
- **Use shader** 7, pass number 0
- **Use mesh** number 7
- **Set the transform** of parameter block 4 to XXX
- **Set the color** of parameter block 4 to YYY
- **Use parameter block** 4
- **Run** the shader
- **Set the transform** of parameter block 4 to ZZZ
- **Run** the shader

# Sorting objects based on GPU info

- We want to **change as few settings as possible** between run calls
  - Changing a setting often involves waiting for the GPU's pipelines to empty, which is very expensive
- So games do a lot of work to **sort/group** game objects based on their **GPU settings**
- The most important way is to arrange the objects in a **DAG** called the **scene graph**
  - As we'll see later, there are **other reasons** for doing this too
  - In particular, arranging objects in **hierarchies of transforms**

# Scene graph

(how could we optimize this more?)



How do we draw a  
scene graph?

# Depth-first walk

**Render(graphnode) {**

**Change setting** specified by node (e.g. texture)

    if (node is a leaf)

        Send run command

    else

        For each child

**Render(child)**

**Undo change** in setting

# Optimization

- The naïve algorithm spends **a lot of time undoing** things
- So graphics drivers are **optimized** to detect cases like
  - Set texture
  - Run
  - ~~Undo set texture~~
  - Set texture
- And they often use **stacks** to keep track of undo information

# Depth-first walk with stack

**Render**(graphnode) {

**Push** the setting we're about to change  
    Change setting specified by node (e.g. texture)

    if (node is a leaf)

        Send run command

    else

        For each child

            Render(child)

**Pop the stack** into the setting we changed

# Polling and event handling

# Core GameObject methods

- **Startup**
- **Update**
- Shutdown
- **Serialize/deserialize**
- Event handling (we'll talk about that later)

# Startup

- When the game **starts**, it
  - **Deserializes** all the objects for the current level
  - Calls all the objects' **startup** methods
    - To let them do any special initialization they want to do
  - Enters the **update/draw loop**
- This continues until the game or level **terminates**

# Polled update (“ticking”)

- Update methods called at **regular intervals**
- Used for update operations that
  - Are **quick**
  - Have to **happen constantly**
- Typically little control over the order in which objects are ticked
- Different types
  - Once **per frame**
  - Once **per physics update** cycle
  - **Early/late** updates
    - Called before/after normal updates for when you need to do something before or after all the normal updates

# Event handlers

- Methods **called irregularly in response to specific events**
  - Object **collisions**
  - Object moving into/out of a designated **trigger area**
  - Object **creation/destruction**
  - **Programmer-defined** events
    - See enemy
    - Get hit
- Some games also treat core methods as events
  - Update
  - Startup
  - Etc.

# Event handling mechanisms

- **Virtual methods**

- Every object has a “something hit me” method
- Engine calls it when something hits it
- **Unity** essentially uses this for most things

- **Event queue**

- Every object has a queue of **event notifications**
- When something hits it, the engine puts a “you were hit” notification in the queue
- Object polls (periodically checks) the queue

- **Callbacks**

- Every object has a field which is a **list of methods to call** if the object is hit
- Objects can put their methods in the list if they want to hear about collisions with that object
- When something hits the object, the engine calls all of them
- Unity uses this for its **UI system**

# Callbacks in C#

- C# has a **special kind of field** called an event
  - Holds a list of callbacks
- Declared as:  
***event type EventName;***
- You can add methods to it by saying:  
***EventName += method;***
- You can call the callbacks by saying:  
***EventName()*** or ***EventName(args ...)***

For more information, see chapters 30 and 31 of the C# textbook or google “C# events”

# Parallelism

# A very important number for real-time animation

- 60 frames per second = **16ms per frame**
- You have **16ms to do everything**
  - Update routines
  - Physics
  - Event handling
  - Audio
  - Rendering

# Multithreading

- You can improve things somewhat by splitting jobs between **different cores of the CPU**
  - Split big chunks of work into separate threads
    - **Update** thread
    - **Render** thread
    - **Physics** thread
    - **Audio** thread
    - **Level loading** thread
  - And hope that helps
- However, **coordination** between threads is very **expensive**
  - So this only helps when you have tasks that can run independently without a lot of interaction

# Asynchronous operations

- What if the object wants to do something that **takes a long time** (e.g. process a mouse drag)?
  - All updates have to run collectively within 16ms
  - Can't have an **update** routine wait for it to end
  - Everything in the **game would stop**
- Often need some mechanism for **long-running operations**
- Different games use different approaches
  - **Coroutines**
    - Can "temporarily return" and continue where they left off later
  - **Separate threads**
    - Use actual parallel processing
    - Update, rendering, and physics are often separate threads
  - **Job systems**
    - Queue of long-running operations
    - Separate thread to run them one at a time

# Example: path planning

- Planning a path for a character is expensive
  - Need to use Dijkstra (A\*, actually) to search a big graph
  - Don't want that in the update method
- So the path planner runs in a separate thread
  - Has a queue of path planning requests
  - Game object's Update method adds request to queue
  - Eventually, planner thread processes it
  - Planner thread sends "path planned" event back to the game object
    - And includes the path

# Readings

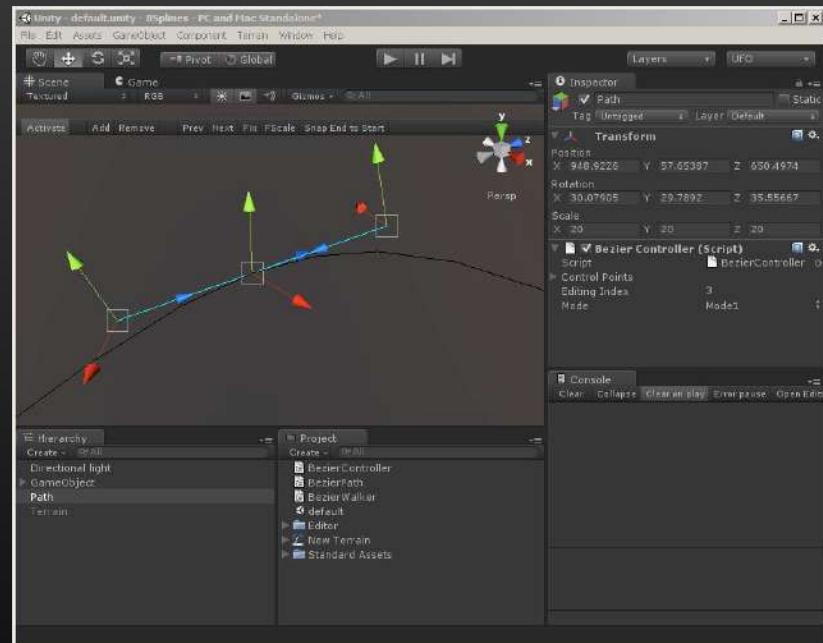
- Game Engine Architecture, ch. 1.6-1.7, 7.1-7.5

# The architecture of **Unity3D**

# Game and editor

## The Unity editor

- Has **your code** compiled into it
- Has **your level** loaded
- Lets you **edit** the objects in the level
- **Serializes** the level back to the level file when you save
- Lets you **run and pause** your game inside of it



# Caveat: your game is really running *inside* the editor

So if your code goes into an **infinite loop**, and the game freezes, then so does the editor!

- You will need to **kill the editor** itself
  - **Force Quit** on Mac
  - **Task manager** on Windows
- When you do this, **unsaved changes** to your level (aka your scene file) will be **lost**
- So you may want to save your scene file before you run your code

# Assets

- All the data associated with your game
  - **Code** (“scripts”)
  - Level files (aka **scenes**)
  - **Media** assets
    - Images
    - 3D models
    - Animation data
    - Sound files
- Stored in the **Assets folder** of your project
  - Can be organized however you like
  - However any subfolder(s) named Editor are special
    - Contain scripts for extending the Unity Editor itself
    - Not included in the game itself

# Serialization

- **Central** to the internal operation of the engine
  - **Loading** stuff, obviously
  - But is also used in **creating new objects**
    - Cloning existing objects
    - Representation of object templates ("prefabs")
- Also central to the **editor**
  - **Level files** are serialized collections of GameObjects
  - **Every time you change your code** files, the editor
    - Serializes your objects to disk
    - Blows the objects away (but not the files)
    - Recompiles your code
    - Deserializes the objects into new objects made with the new code
- Unfortunately, Unity's serializer is known for being kind of **crappy** (see appendix)

# GameObjects

# UnityEngine.Object

- **Different from `object`**, aka `System.Object`
- Has a **name** field (a string)
- Useful static methods
  - **`Destroy(Object)`**  
Nukes the object
  - **`FindObjectOfType<Type>()`**  
Finds an object of *Type* and returns it
  - **`FindObjectsOfType<Type>()`**  
Returns an array of all objects of *Type*

# GameObject

- Subclass of **UnityEngine.Object**
- Collection of **Components**
- Contains
  - A **name** (string)
  - A collection of **Components**
    - One of those is always the Transform
      - And the Transform can contain child objects
  - Some **other stuff** we'll ignore for the moment (a "tag", a layer)

# GameObject hierarchy

- GameObjects can have other GameObjects (**children**) within them
- So a GameObject is really
  - A list of **Components**, and
  - A list of **child GameObjects**
- In other words, the GameObjects in a given level file (aka a scene) form a **tree** called the **hierarchy**
  - You can see the hierarchy for the scene you're working on the left in the Unity editor
- Full disclosure: the GameObjects don't technically have the children in them
  - The GameObjects have Transforms
  - Transforms have child Transforms
  - Which them have pointers back to their respective game objects

# Object creation and destruction

- **Object.Instantiate(*GameObject*)**
  - Makes a clone of GameObject
- **Destroy(*GameObject*)**
  - Does exactly what it sounds like.

# Prefabs

- Files containing **serialized GameObjects**
- Can be used as **templates for creating GameObjects** just by repeatedly deserializing from them
- **GameObject.Instantiate(*prefab*)**
- **GameObject.Instantiate(*prefab, position, rotation*)**
  - Creates a new game object by deserializing the data in the prefab
  - We'll talk more about this later

# Useful static methods

- **GameObject.Find(*string*)**
  - Returns the GameObject with the specified **name**
- **GameObject.FindWithTag(*string*)**
  - Returns the first GameObject it finds with the specified string in its **tag** field
- **GameObject.FindGameObjectsWithTag(*string*)**
  - Returns an array of all the GameObjects with the specified string in their **tag** fields

# Components

# Component

- **Basic unit** of functionality
- Can be **combined together** into GameObjects
- Can access the other components of its GameObject using **GetComponent<Type>()**.

# The Transform component

- Specifies relationship between the GameObject's **local coordinate system** and the world coordinate system
- We'll talk more about this later, but the Transform specifies:
  - **Position**
  - **Rotation**
  - **Scale**
- **Every game object** has a Transform even if it's not an object that's rendered on the screen
  - Exception: **UI objects** have a **RectTransform** instead
- Every GameObject and Component has a **transform field** that holds the Transform.

# GameObject hierarchy

- GameObjects are arranged in a **tree**
- In particular, their **Transforms** are arranged in a tree
  - A child GameObject's transform specifies its position, rotation, and scale **relative to** that of **its parent**
- So the object hierarchy is essentially a **simplified scene graph**
  - Or, alternatively, the transform hierarchy is forced into doing double-duty as a container for the level

# Common components

- **Camera**
  - Specifies how to render the scene
- **Transform**
  - Specifies where to render an object
- **Renderer/SpriteRenderer**
  - Actually draws the object
- **Light**
  - Specifies position and color of a light source
- **Skybox**
  - Specifies background
- **Collider/Collider2D**
  - Tells physics the dimension of the object
  - Or specifies a region of space to be monitored for objects entering and exiting ("triggers")
- **RigidBody/RigidBody2D**
  - Performs physics simulation for the object, updating its position and rotation based on rigid body dynamics

# Accessing Components

- **GetComponent<Type>()**  
Returns the object's component of the specified *Type*, or null if this object doesn't have a component of that *Type*.
- **GetComponentInChildren<Type>()**  
Same, but checks children (and other descendants) if it can't find a component in this game object.
- **GetComponentInParent<Type>()**  
Same, but checks ancestors if it can't find a component in this game object.
- **GetComponents<Type>()**  
Returns an array of all the object's components of that *Type*.
- **GetComponentsInChildren<Type>(),  
GetComponentInParent<Type>()**  
Just what you'd expect.

# Making your own components

- Make a new class that's a subclass of **MonoBehaviour**
  - Note British spelling
- Add **fields** as appropriate
  - Fields declared as **public** will be
    - **Serialized** to the level file
    - **Editable** in the inspector (unless marked **[HideInInspector]**)
    - You can also force private fields to be serialize by marking them **[SerializedField]**
  - Unless they're something Unity can't serialize (see appendix)
- Add **message handlers** as appropriate (e.g. `Update`)
- Add anything else you need

# Initializing your Component

- **Avoid constructors** unless you know what you're doing
  - Your constructor (if you define one) runs before the object is deserialized so
  - Field initializations **may be overwritten**
  - It runs in the load thread, not the main thread, so you **can't call most Unity functions**
- Put initializations into a **Start() message handler** instead

# Attributes

- C# lets you to annotate classes, methods, and fields with **custom metadata** called attributes
- You can add an attribute to a declaration by putting **[attribute]** immediately before it.
- Unity uses attributes extensively to let you customize its behavior

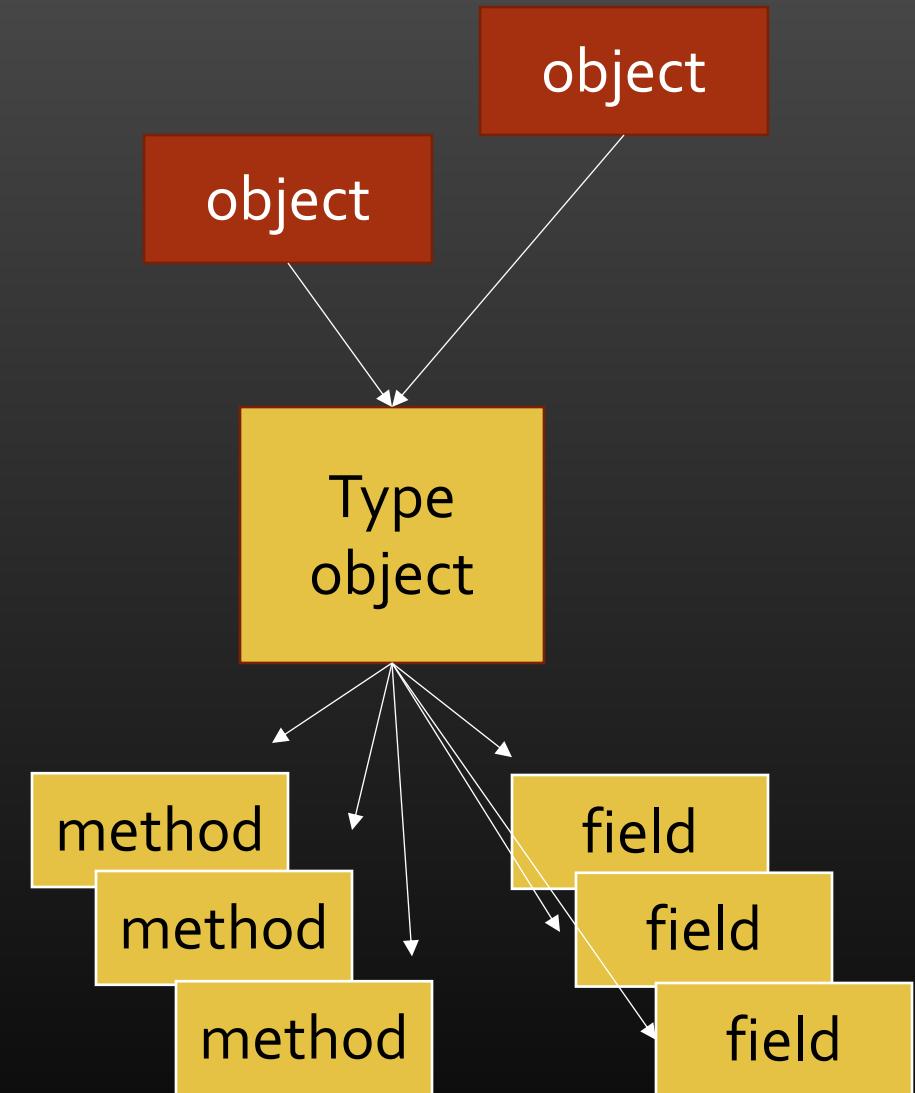
# Examples

- **public float FooBar;**
  - Unity will serialize the FooBar field to the level file, and also let you edit it in the inspector.
- **[HideInInspector]**  
**public float FooBar;**
  - It's still public, and still serialized, but Unity won't display it in the editor.
- **[NonSerialized]**  
**public float FooBar;**
  - Unity will leave it alone completely;
- **[MenuItem("MyMenu/Log something")]**  
static void LogSomething() {  
 Debug.Log("Bla bla bla")  
}

# Update and message handling

# Reflection

- Modern languages keep **metadata** about objects at run-time
  - Types
  - Methods
  - Etc.
- Unity makes extensive use of **reflection**,
  - Ability to **ask objects about their types** and methods at **run-time**



# The Type class

- System has one **Type object** per data type in the system
  - Classes
  - Structs
  - Arrays
  - Etc.
- The **GetType() method** of an object returns its type object
  - Essentially every object stores its type in a hidden field
  - GetType() reads the field
- You can also use the **typeof** operator to get the run-time type object of a type given its name (e.g. **typeof(int)**)
- Type object stores **information about the type's**
  - Name
  - Constructors
  - Fields
  - Methods
  - Properties
  - Events
- Provides methods for **manipulating objects** of its type
  - Calling methods
  - Reading/writing fields
  - Etc.

# Messages

- Unity talks to components using **reflection**
- It calls this sending them “**messages**”
  - This is different from the messages we talked about in the last deck
- You can send a message to a GameObject by calling its **SendMessage("methodname")** method
  - `SendMessage("Update")` will call the `Update()` method on every component

# Common messages

## Startup

- void **Awake()**
  - Called when the object is first created
- void **Start()**
  - Called when all the objects have been deserialized

## Update

- void **Update()**
  - Called for each frame update
- void **LateUpdate()**
  - Called after Update() has been called on all components of all objects
- void **FixedUpdate()**
  - Called after each update of the physics system

# Messages are a pain

- They operate based on **string names**
  - So **typeos aren't detected** by the compiler
- They **bypass visibility** declarations  
(public, private, etc.)
  - I usually declare messages to be **internal**, i.e.:  
internal void Update() { ... }
  - But you can declare them public, private, or protected,  
and it will make no difference

# Colliders

- Represents the **volume occupied by a GameObject**
    - We'll use 2D colliders for the moment
  - Colliders come in two flavors
    - **Regular**  
Used by physics for bouncing and friction
    - **Trigger**  
Ignored by physics; sends message when an object passes through the collider
  - To make something a trigger, set its **trigger field** to true.
- **Collider2D**
    - Base class of collider components
  - **BoxCollider2D**
    - A rectangular region
  - **CircleCollider2D**
    - A circular region
  - **PolygonCollider2D**
    - An arbitrary polygon
  - **EdgeCollider2D**
    - A series of lines that can be bounced off of, but don't fill a volume

# Collider messages

- void **OnCollisionEnter2D**(Collider2D)
  - Called when something bounces off of the component's game object
  - We'll talk about when we get to physics
- void **OnTriggerEnter2D**(Collider2D)
  - Called when an another object enters this object's trigger area
- void **OnTriggerStay2D**(Collider2D)
  - Called when the object is still inside this object's trigger area
- void **OnTriggerExit2D**(Collider2D)
  - Called when the object leaves this object's trigger area
- void **OnMouseDown()**
  - Call when used clicks inside this object's collider

# Execution

# Level load

- The system begins by **loading the level** from your scene file
  - Create all the objects and deserialize all the data into them
  - Send them all an **Awake()** message
  - Send them all a **Start()** message
- **Caveat:** loading **runs in its own thread**
  - Good: lets games (optionally) start running before they finish loading
  - Bad: there are a lot of operations you aren't allowed to perform from within Awake()

# Magic static classes

- **Input**
  - You've already looked at this one
- **Time**
  - Contains all the information about current game time and frame rate
  - **Time.time**
    - How many seconds the game has actually been running for
  - **Time.fixedTime**
    - How long physics *thinks* the game has been running for
  - **Time.deltaTime**
    - Time (in seconds) since the last frame update
  - **Time.fixedDeltaTime**
    - Simulated time between physics updates

# Update/Draw loop

- We'll skip talking about drawing for a while.
- Update is a little complicated
  - The basic update message is **Update()**
    - It's run once per frame
    - But your frame rate depends on how fast your computer is
    - So it runs at a variable frequency
  - **Physics dies** if you have a variable update frequency
    - So it **simulates a fixed update frequency** by repeatedly updating itself until its caught up with the current real time
    - Physics updates send the **FixedUpdate()** message

# Approximate update/draw loop

```
while (true) {  
    Time.time = actual clock time;  
    while (Time.fixedTime < Time.time) {  
        SendMessage("FixedUpdate");  
        Time.fixedTime += Time.fixedDeltaTime;  
    }  
    SendMessage("Update");  
    SendMessage("LateUpdate");  
    ... draw the frame ...  
}
```

# Summary: Weird design decisions in Unity

- The GameObject hierarchy **is the transform hierarchy**
  - So all GameObjects have a Transform
    - Except UI elements, which have a RectTransform
  - Even if they don't appear on the screen or have a well-defined position
- **Message dispatch** is based on spelling
  - No compile-time checking
- **Serialization** is klugy

# Appendix

**Unity's serializer** is terrible

# Problems with Unity's serializer

- Silently **refuses to serialize** certain objects
- Doesn't understand **subclassing**
- Doesn't ?!\$@# understand **null**
- Can only handle **DAGs and cycles** in certain cases

# Basic things to know

- By default only serializes **public fields**
  - But you can override that
- Serialization of **built-in Unity types** basically works
- Serialization of **built-in C# types** sort of works
  - **Primitives**: int, float, strings, etc.
  - **Arrays** thereof
  - **List<Type>**
  - But **not other generic types**
    - e.g. Dictionary<string, string> is ignored by the serializer
  - Subclassing doesn't work
    - So fields of type object will be ignored
  - Null doesn't work
    - If you serialize an array of objects, any nulls get turned into objects with default values for their slots
  - Cyclic structures don't work

# Serializing new classes

- If your class is a subclass of **UnityEngine.Object** (e.g. subclass of MonoBehaviour)
  - Unity will serialize it
- **If not, the serializer will quietly ignore it**
  - If you just say class Employee { fields ... }, it won't serialize it unless you mark it [Serializable]
- Unity will also ignore it if it's a **generic type**

```
class A<T> { ... stuff ... }
```

```
class B : A<int> { ... stuff ... }
```

Fields of type A<int> won't serialize, but fields of type B will.

# Serializing new classes

- In either case, Unity serializes your classes **by value**, not by reference
  - Null doesn't work
  - Subclassing doesn't work
  - Cycles don't work
  - Etc.
- You can force it to serialize by reference by making your class derive from **ScriptableObject**

# Things Unity won't serialize

- Built-in **generic types** (e.g. Dictionary<T<sub>1</sub>, T<sub>2</sub>>)
  - But it will serialize List<T>
- User-defined **generic types** (sort of)
  - If you define a field to be MyClass<int>, it won't serialize it
  - If you say
    - class MyOtherClass : MyClass<int> {}
    - And then declare your field to be of type MyOtherClass, it will serialize it

Modeling **quantity**

# Scalars

- Single numbers are often called **scalars**
- They represent **amounts** of something
  - As opposed to **vectors**, that represent an amount **and a direction**



# The development of numbers

- It's taken a **couple thousand years** for people to agree on what things are valid numbers
  - And even so, there's still controversy in some circles
- There are a few things that everybody agrees on
  - 1, 2, 3, 4 ...
- But other kinds of numbers were more **controversial**
  - **Zero**
  - **Negative** numbers
  - The continuum (i.e. **real** numbers)
  - **Complex** numbers

# But will this be on the test?

- Most of this is **background** information
- There are a bunch of **concepts that come up over and over in class**, and that you need to understand
  - Breaking things down to addition and multiplication
  - Linearity
- But we will only test you on **material that specifically comes up in programming**
  - Number **types**
    - But don't try to memorize the ranges of specific data types, i.e. what the value of double.MaxValue is.
  - **Failure modes** of machine arithmetic
    - Overflow/underflow
    - Floating-point error
    - What NaN is

natural numbers

# The “natural” numbers

- Believed to have been invented for **accounting**
- Who owed whom how many oxen?



# Representation of numbers in computers

- There is an **infinite set of numbers**
- But a computer has only a **finite memory**
- So computers are **limited** in their ability to represent numbers
  - Note: lots of caveats apply here

# Representing natural numbers

(aka unsigned ints, uints)

- Computer data is ultimately represented using **bit patterns**
  - There are many **possible ways** of representing natural numbers,
  - But almost always use base 2 radix notation, aka **binary**
- 0 = 0000
  - 1 = 0001
  - 2 = 0010
  - 3 = 0011
  - 4 = 0100
  - 5 = 0101
  - 6 = 0110
  - 7 = 0111
  - 8 = 1000

# Addition

- Addition captures the **merging** of quantities



+



=



# Arithmetic overflow

- Integer data types generally use **fixed numbers of bits**
    - 4, 8, 16, 32, 64
  - An  $n$ -bit number can represent the range zero to  $2^n - 1$ 
    - Arithmetic operations compute the result **mod  $2^n$**
  - So results outside that range “**wrap around**”
    - 4-bit arithmetic:  
 $10 + 8 = 1010_2 + 1000_2 = 10010_2 = 18$
    - But **truncated** to 4 bits, you get  $0010_2 = 2$
  - C# can check for overflow, but it doesn't by default
- 0 = 0000
  - 1 = 0001
  - 2 = 0010
  - 3 = 0011
  - 4 = 0100
  - 5 = 0101
  - 6 = 0110
  - 7 = 0111
  - 8 = 1000

# Natural number types in C#

- **byte**
  - 8 bits
  - 0-255
- **ushort**
  - 16 bits
  - 0-65,535
- **uint**
  - 32 bits
  - 0-4,294,967,295
- **ulong**
  - 64 bits
  - 0-  
18,446,744,073,709,551,615

- Each of these types has
- **MaxValue** field
    - E.g. `uint.MaxValue`
    - Largest number that can be represented with that type
  - **MinValue** field
    - E.g. `uint.MinValue`
    - Smallest value that can be represented
    - For unsigned numbers, this is always zero
  - **`uint.MaxValue+1`  
= `uint.MinValue = 0`**

# Multiplication

- Multiplication captures **iterated addition**
- Addition and multiplication together give us
  - The ability to **express a lot of functions**
  - A lot of **useful properties**

## Identity elements

$$n + 0 = 0 + n = n$$

$$1n = n1 = n$$

## Commutativity

$$n + m = m + n$$

$$mn = nm$$

## Associativity

$$(l + n) + m = l + (n + m)$$

$$(ln)m = l(nm)$$

## Distributivity

$$l(n + m) = ln + lm$$

# Shifting binary numbers

- If you represent numbers in binary, then **shifting** left/right corresponds to **multiplying/dividing by 2**
- $4 \gg 1 = 0100_2 \gg 1 = 0010_2 = 2$
- $5 \gg 1 = 0101_2 \gg 1 = 0010_2 = 2$
- $5 \ll 1 = 0101_2 \ll 1 = 01010_2 = 10$

# Defining functions in terms of addition and multiplication

- Your computer does most of its math work in terms of arithmetic
- What happens if we limit ourselves to just the functions you can get by **combining addition and multiplication?**
  - i.e. arbitrary combinations of  $+$ ,  $\times$ , **constants**, and **variables**
- What functions do we get

# Polynomials

- The polynomials are precisely the functions that are composed only of **addition and multiplication**
- We're used to seeing them in the standard form
$$a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$$
- But **any combination of addition and multiplication** can be rewritten in the standard form

# Linear functions

- Linear functions are an **incredibly important** special case of polynomials
  - Linear means “**line-like**”
- There are **two senses** of linear that get used
  - Most common sense: **sum of constants times inputs**
    - $f(x) = kx$
    - Functions of multiple variables:  $f(x_1, x_2) = k_1x_1 + k_2x_2$
    - This is the sense we'll use in class
  - **High school** sense:  $y = mx + b$ 
    - This is technically called an **affine** function or inhomogeneous linear function
- We'll talk more about linearity next time
- This is mostly a placeholder

integers

# Inverting addition

- Subtraction **undoes addition**
  - $(n + m) - m = n$
- But can also be defined in terms of **additive inverses**
  - $n + (-n) = 0$
- Inverse elements take us outside the realm of natural numbers and into the **integers**
  - We have to allow **negative numbers**
- This was weirdly **controversial**
  - Descartes grudgingly accepted complex numbers
    - He invented the term “imaginary number”
  - But **never accepted negative numbers!**

# 2's complement integers

- Need a system of bit patterns for **signed integers**
- The **most popular** system is called “2's complement”
- Note: **overflow** causes a **sign change**
- $011 = 3$
- $010 = 2$
- $001 = 1$
- $000 = 0$
- $111 = -1$
- $110 = -2$
- $101 = -3$
- $100 = -4$

# 2's complement has nice properties

- Same addition and subtraction rules as unsigned integers
- You can multiply/divide with shifting
  - Provided you **clone the most significant bit**
  - $-4 >> 1 = 1100_2 >> 1 = 1110_2 = -2$
- $011 = 3$
- $010 = 2$
- $001 = 1$
- $000 = 0$
- $111 = -1$
- $110 = -2$
- $101 = -3$
- $100 = -4$

# Integers types in C#

- **sbyte**
    - 8 bits
    - -128 to 127
  - **short**
    - 16 bits
    - -32,768 to 32,767
  - **int**
    - 32 bits
    - -2,147,483,648 to 2,147,483,647
  - **long**
    - 64 bits
    - -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
- **int.MaxValue+1 = int.MinValue**
  - **int.MinValue-1 = int.MaxValue**

floating point

# Inverting multiplication

- Division **undoes multiplication**
  - $\frac{ab}{b} = a, a \neq 0$
- But can also be defined through **multiplicative inverses**
  - $aa^{-1} = 1$
- Incorporating multiplicative inverses gives us the **rationals**

BTW: division is just as hard for computers as for humans

# Floating-point numbers

- Again, computers **can't represent all rational** numbers because of limited space
- The most common **representation for rationals** is floating-point
  - Basically just "**scientific notation**"
    - E.g.  $6.02214 \times 10^{23}$
  - But in **binary**
    - $0.1011101011 \times 2^{1011}$
  - Composed of two values
    - **Mantissa**: small number to be scaled
    - **Exponent**: multiplier that scales the mantissa

# Floating-point numbers

```
class CrappyFloat {  
    int mantissa;  
    int exponent;  
  
    //This has bugs, but it gets the idea across  
    CrappyFloat Add(CrappyFloat a, CrappyFloat b) {  
        var big = a.exponent>b.exponent?a:b;  
        var little = a.exponent<b.exponent?a:b;  
        var expDifference = big.exponent-little.exponent;  
        return new CrappyFloat(  
            big.mantissa+(little.mantissa >> expDifference),  
            big.exponent  
        );  
    }  
}
```

# Floating-point error

- FP only represents values **approximately**
- And FP operations add further **rounding error**
  - Example in base 10, with 4 digits of precision:  
$$\begin{aligned}100,000 + 15 &= 1 \times 10^5 + 1.5 \times 10^1 \\&= 1 \times 10^5 + 0.00015 \times 10^5 \\&\approx 1.00015 \times 10^5 \\&\approx 1 \times 10^5 \text{ rounded to 4 digits}\end{aligned}$$
  - So **100,000 + 15 = 100,000**
- This **error**
  - **Accumulates** with more operations
  - Is **proportional to the magnitude** of the numbers (bigger numbers have bigger errors)

# Floating-point addition is **not** associative

- **Order** of operation matters because of **rounding**
  - $(100,000 + 50) + 50 \approx 100,000 + 50 \approx 100,000$
  - $100,000 + (50 + 50) \approx 100,000 + 100$   
 $\qquad\qquad\qquad\approx 100,100$
- Ideally want to add numbers from **smallest to largest**
  - **Good luck** with that
  - You **won't have to worry** about that for this course

# Exotic values in IEEE floating point standard

## Infinities

- $\frac{1}{0} = \infty$
- $\frac{-1}{0} = -\infty$
- $k\infty = \infty, k > 0$
- $k\infty = -\infty, k < 0$
- $k(-\infty) = -\infty, k > 0$
- $k(-\infty) = \infty, k > 0$
- $\infty + 1 = \infty$
- $\infty - 1 = \infty$

## NaN ("not a number")

- $\frac{0}{0} = \frac{\infty}{\infty} = \frac{\infty}{-\infty} = -\frac{\infty}{\infty} = \text{NaN}$
- $0\infty = 0(-\infty) = \text{NaN}$
- $\infty + (-\infty)$   
 $= (-\infty) + \infty = \text{NaN}$
- $\infty - (-\infty)$   
 $= (-\infty) - \infty = \text{NaN}$
- $1 + \text{NaN} = \text{Exception}$

# Floating-point in C#

## Float type

- 32 bit IEEE standard float
  - 23 bit mantissa
  - ~7 significant decimal digits

## Double type

- 64 bit IEEE float
  - 52 bit mantissa
  - ~16 decimal digits of accuracy

# Magic values in C# floats

## Exotic values

- `float.PositiveInfinity`
  - The  $+\infty$  bit pattern
- `float.NegativeInfinity`
  - The  $-\infty$  bit pattern
- `float.NaN`
  - The not a number bit pattern

## Extremal values

- `float.Epsilon` =  $1.4 \times 10^{-45}$ 
  - Smallest positive float
  - `float.Epsilon/2=0`
- `float.MaxValue` =  $3.40282347 \times 10^{38}$ 
  - Largest finite float
  - `float.MaxValue+1=float.PositiveInfinity`
- `float.MinValue` =  $-3.402823 \times 10^{38}$ 
  - Smallest finite value
  - `float.MinValue-1 = float.NegativeInfinity`

# Loss of precision warnings

- Compiler complains when you **store a double into a float** variable
  - You're losing digits!
  - Did you really mean to do that?
- Usually not a problem because **everybody uses doubles**
- But **not in games** :-(
  - **Graphics cards use floats** for efficiency
  - So the **Unity API uses floats** everywhere
  - And so you **get this error a lot**

Requires minor adjustments

- **Math** static class
  - Math.Sin, Math.Cos, etc.
  - All **return doubles** :-(
- So Unity has a **Mathf** class
  - Mathf.Sin **returns a float**
- Also **have to type "f"** after a lot of numbers ...

# Numeric literals in C#

- There are **many different number 10s**
- C# has a **notation to specify** which one you mean
  - 10 is an **int**
  - 10**u** is a **uint**
  - 10**U** is a **ulong**
  - 10.**0** is a **double**
  - 10.**0f** is a **float**

# The reals

- Most math uses **real numbers**
- These **can't be represented** in computers
  - $\pi$  has an **infinite number of digits**
- Can only be **approximated using rationals**
- That usually means **floats**
  - Which are rationals where the denominator is always a power of 2
  - So **even less accurate** approximations than regular rationals: can't represent  $1/3$ .

# function approximation

# Approximating functions

- Ultimately, your computer computes functions using
  - Simple **arithmetic**
  - Table **lookup**
  - Combinations of the above with **looping** and **recursion**
- So functions like **sin** and **cos** are always **approximated**
  - Instead of computing  $\sin(x)$ , we compute  $\sinish(x)$ , which is close to  $\sin(x)$

# Local approximation

- Often times, we write the approximation **relative to a value we know exactly**
- Example:
  - $\sin(0) = 0$
  - $\sin(x) \cong x$ , for small values of  $x$
- How do we talk about **how good** an approximation this is?

# Big oh notation

- Measures how fast a function **grows as  $x \rightarrow \infty$** 
  - $f(x) = O(x^n)$  means “f **grows no faster than  $x^n$** ”
  - In particular: for  $x > x_0$ ,  $f(x) \leq kx^n$
- We want the **execution time of an algorithm** to **grow slowly**
- So we want **small exponents**

# Little oh notation

- Measures how fast a function **shrinks as  $x \rightarrow 0$** 
  - $f(x) = o(x^n)$  means “f **shrinks faster than  $x^n$** ”
  - $\lim_{x \rightarrow 0} \frac{f(x)}{x^n} = 0$
- We want the **error in our approximations to shrink fast**
- So we want **high exponents**
  - $o(x^2)$  convergence is better than  $o(n)$  convergence

# Linear approximations

- $\sin(x) \cong x$  is a **linear** approximation
  - $f(x) = x$  is a linear function
- How do we know there isn't a **better** linear approximation?

# Theorem

- For any function  $f: \mathbb{R}^n \rightarrow \mathbb{R}$ , and any input  $x_0$  for  $f$ , there is **at most one linear function**  $Df_{x_0}$  for which:
  - Let  $g(x + \Delta x) = f(x_0) + Df_{x_0}(\Delta x)$
  - $|f(x_0 + \Delta x) - g(x_0 + \Delta x)| = o(\Delta x)$
- Translation: there's at **most one linear function** that approximates a given function at a point with **better than linear error**

# The differential of a function

- The function  $Df_{x_0}$  is called the **differential** or **total derivative** of  $f$  at the point  $x_0$
- You can define it in terms of normal derivatives
  - $Df_{x_0}(x) = (x - x_0)f'(x_0)$ , for one variable
  - $Df_{x_0}(x) = (x - x_0) \cdot \nabla f(x_0)$ , for vector functions
- But it also gives you a different way of thinking about **what a derivative means**: it's about **approximation**
  - You can construct calculus starting from the uniqueness of the total derivative
  - And not mess with all those limits and  $\epsilon - \delta$  proofs

# Taylor polynomials

- And of course, you can do that with fancier polynomials
- A Taylor polynomial is just a **truncated Taylor series**:
  - $f(x_0 + \Delta x) \approx f(x_0) + f'(x_0)\Delta x + \frac{1}{n!}f^n(x_0)\Delta x^n$
- An nth-order Taylor polynomial approximates a function with an **error of  $o(\Delta x^n)$**
- So, again, we can think of derivatives as being about how **polynomials approximate functions**

# complex numbers

# Inverting polynomials

- Finally, we talked about **inverting** combinations of addition and multiplication
  - Let's look at how to **invert a whole combination** of additions and multiplications
  - That is, a **polynomial**
- If we have  $f(x) = k_0 + k_1x + \cdots + k_nx^n$ , how do we compute  $f^{-1}(y)$ ?

# Roots

- Notice we only have to figure out how to **solve for  $f^{-1}(0)$** , aka the **roots** of  $f$
- Why?
  - For other values of  $y$ , we can just define
  - $g(x) = f(x) + y = (k_0 + y) + k_1x + \cdots + k_nx^n$
  - And then find  $g^{-1}(0)$
  - So if we know how to solve for  $0$ , we can solve for anything

# Roots of general polynomials

- There is **no general formula** for finding the roots of an arbitrary polynomial
  - Linear polynomials are easy
  - You learned a formula for quadratics
  - And there's one for cubics too
- For higher-order ones, we have to use **numerical methods**
- For example, **Newton's method**
  - Make an initial guess:  $x_0$
  - Make a better guess:  $x_1 = x_0 - f(x_0)/f'(x_0)$
  - Make a still better guess:  $x_2 = x_1 - f(x_1)/f'(x_1)$
  - Lather, rinse, repeat

# Roots of quadratics

- There is, however, a **closed form** for the roots of quadratics, that we all learn in high school
- For quadratic  $ax^2 + bx + c$ , the roots are:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- Of course, there's the issue of what happens when  **$b^2 - 4ac$  is negative**
  - For quadratics, people were able to **ignore** this issue
  - And just say “then there's **no root**”

# Roots of cubics

- Mathematicians found a **closed form** solution for cubics in the **16<sup>th</sup> century**
- I won't show it to you because it's way too complicated
  - First mathematical proof that rich people had too much free time in the renaissance

# Closed-form solution for cubics

- It had a little **embarrassing feature**
  - You ended up with **square roots of negative numbers**
  - Even when there were real-valued roots
  - It's just that the **negative square roots canceled**
- So mathematicians were left with a **choice**
  1. Declare their magic **formulae invalid**
  2. Declare that **imaginary numbers exist**
- They chose **option 2**
  - But they weren't happy about it
  - They still didn't believe in negative numbers!

# Complex numbers

- Once you decide there can be square roots of negative numbers, you **quickly realize that**
  - All numbers can be written as a **sum of a real number and an imaginary number**
  - All imaginary numbers are **multiples of  $\sqrt{-1}$**
- So all (complex) numbers are **linear combinations of  $1$  and  $\sqrt{-1}$**
- This became European mathematics' **first attempt at representing (2d) space** with mathematical objects
  - Which we will start to talk about next time

Modeling 2D space

# Goal

- We said that computers ultimately do most of their math using **arithmetic**
  - +, -, \*, /
  - And **division is hard**
- So we want to **represent space** using just
  - **Numbers**
  - **Addition**
  - **Multiplication**

# Representing geometry

Need to represent several different **kinds of objects**

- Points
- Vectors (displacements between points)
- Pseudovectors (things perpendicular to other things)
- Directions
- Lines
- Planes (when we get to 3D)
- Distances
- Angles
- Parallelism
- Shape

What we usually see for **representing** them

- Numbers
- “Vectors”
- Matrices
- Dot products

# Linear spaces

- A linear space is a **set together with**
    - **Addition**
    - **Scalar multiplication**
  - That **behaves as you'd expect**
    - Commutativity
    - Associativity
    - $0x = 0$
    - $k(x + y) = kx + ky$
    - $x + (-1)x = 0$
- Examples
- The real numbers
  - **Vectors**
  - **Matrices**
  - **Functions**
    - (Between linear spaces)
  - **Polynomials**

# Linear functions

- Functions that **commute with addition and scalar multiplication**
  - $f(x + y) = f(x) + f(y)$
  - $f(kx) = kf(x)$
- Let you **switch between** doing
  - Performing the **arithmetic first**
  - Or performing the **function first**
- Form a **rich algebra**

# Linear functions form linear spaces

- The **set of linear functions** over a given linear space is **itself a linear space**
  - Or between two linear spaces
- **Arithmetic over functions** is defined in the obvious way
  - $(f + g)(x) = f(x) + g(x)$
  - $(kf)(x) = kf(x)$
  - 0 element = constant function  $f(x)=0$

# Coordinate vectors and bases

# Bases

- **Minimal set** of elements, sums and multiples of which **generate the whole space**
- **Not unique**
  - There are **many possible bases**
- We're mostly going to focus on **orthonormal** bases
  - Basis vectors have **length 1**
  - And are **mutually perpendicular**

# Coordinate vectors

- Bases allow us to represent any element as a **weighted sum of basis elements**

$$x = x_1 b_1 + \cdots + x_n b_n$$
$$y = y_1 b_1 + \cdots + y_n b_n$$

- So we can specify any element just by **specifying the weights** (aka the **coordinates**)
  - $x = (x_1, \dots, x_n)$
  - $y = (y_1, \dots, y_n)$
- Provided we **agree on the basis** for a representation
  - If we **change bases**, we have to **change coordinates**
- This even holds for **infinite dimensional spaces**, but we won't talk about those until we get to **audio**

# Coordinate vectors

- Coordinate vectors are **important** because they allow us to
  - **Represent** any (finite dimensional) linear space
  - As a finite set of **numbers**
- And fortunately, **most of what we want to represent** in graphics is finite-dimensional linear spaces

# Coordinate vectors aren't the original space

- They're a particular **representation** of the original space  $V$ 
  - Determined by a **mapping from  $V$  to  $\mathbb{R}^n$**  (the space of coordinate vectors)
- It's important to **keep them straight** because
  - We'll end up representing **lots of things** with coordinate vectors
    - In particular, graphics ends up representing everything as **4-vectors** and **4x4 matrices**
  - We have to remember that they **don't all represent the same kind of thing**

# Coordinate representation

- Let  $f: V \rightarrow \mathbb{R}^n$  be the function mapping elements of the **linear space to coordinate vectors** representation (using whatever basis we chose)
- What **kind of function** is  $f$ ?
- A **linear** function
  - $f(x + y) = f(x) + f(y)$
  - $f(kx) = kf(x)$

# Linear functions have bases too

- Consider a function  $f$  from an  $n$ -dimensional space  **$V$  to itself**
  - Choose your favorite **basis for  $V$** 
$$b_1, b_2, \dots, b_n$$
  - Define a **set of linear functions**
    - $b_{ij}(kb_i + \text{whatever}) = kb_j$ , for all  $i, j$
- We can **write  $f$  as a weighted sum** of these functions
$$f = a_{11}b_{11} + a_{12}b_{12} + \cdots a_{1n}b_{1n} + \cdots + a_{n1}b_{n1} + a_{nn}b_{nn}$$

# Linear functions have bases too

So

- The  $b_{ij}$  form a basis for functions from  $V$  to itself
  - We can pull the same trick for functions between different spaces
- We can represent  $f$  in terms of just the weights  $a_{ij}$

What do we call that representation?

# Coordinate representation of linear functions

A matrix is just a **coordinate representation** for a **linear function**

$$f(x) = \begin{bmatrix} a_{11} & \cdots & a_{n1} \\ \vdots & \ddots & \vdots \\ a_{1n} & \cdots & a_{nn} \end{bmatrix} x$$

And of course, matrices form linear spaces too

# Coordinate representation of linear functions

- Important because it gives lets us **implement any linear function** using just
  - **Addition**
  - **Multiplication**
- This is very **convenient** because
  - Linear functions are **most of what we want to compute** in graphics
  - CPUs are **really good at addition and multiplication**
  - GPUs are even better
    - They're basically just **big dot product engines**

# Affine spaces

# Bad news: Physical space isn't a linear space :-(

**Has** notions of

- Position
- Distance
- Angle
- Etc.

**Doesn't have** notions of

- Addition
- Multiplication
- “zero”

You **can't add** points in space

# Affine spaces

- Physical space is more like an **affine** space
- Informally: a vector space where you **forget what zero is**
- Formally:
  - A set of **points**
  - A separate set of **vectors** representing **displacements** between the points
  - A set of **transformations** that preserve parallelism of lines
    - Called **affinities** or **translations**
    - Basically adding a vector to a point

# Points and vectors in affine spaces

## Point

- Represents a **location** in space
- No arithmetic operations

$\text{Vector} + \text{Vector} = \text{Vector}$

$\text{Scalar} * \text{Vector} = \text{Vector}$

$\text{Point} + \text{Vector} = \text{Point}$

$\text{Point} + \text{Point}$  *undefined*

$\text{Scalar} * \text{Point}$  *undefined*

## Vector

- Represents a **displacement** (translation) in space
- Arithmetic operations

# Representing points using coordinate vectors

- We can still **represent points using vectors**
- But we have to choose a particular point to represent **zero**
  - Aka the **origin**
- Then **represent a point** as
  - The **coordinate vector** for
  - The **displacement** vector of the point
  - From the **origin**

# Representing points using coordinate vectors

So then a **coordinate system** consists of

- An **origin**
- A set of **basis vectors** (axes)
  - Usually orthonormal in graphics
- This is called a coordinate frame

# Transforming coordinate systems

- We have to worry about **both**
  - Change of axes (change of **basis**)
  - Change of **origin**
- So **transforming coordinate systems** involves
  - A **translation** (addition of a constant vector)
  - Then a change of basis (**rotation** matrix)
- This is **inconvenient**
  - Why?

# Nonlinearity

- Changing coordinate systems **isn't linear**

$$y = Ax$$

- It's an **affine** function

$$y = Ax + \mathbf{b}$$

- This isn't the end of the world, but it **complicates implementation**

Fortunately, there's a **clever hack** to get around it that we'll talk about shortly

# Why am I being so pedantic?

- It actually **ends up mattering** in graphics code
- For example
  - Different kinds of objects **transform differently** when you switch coordinate systems
  - **Vectors** transform using a **change of basis matrix**
  - **Points** transform with that matrix **plus an offset**
    - To account for the difference in origins between the coordinate systems
- **Unity somewhat protects you** from this
  - But there still end up being multiple methods for transforming coordinates
  - And when you write GPU code you actually need to remember to represent different kinds of things differently

# Angles, rotations, and directions

And also oriented areas

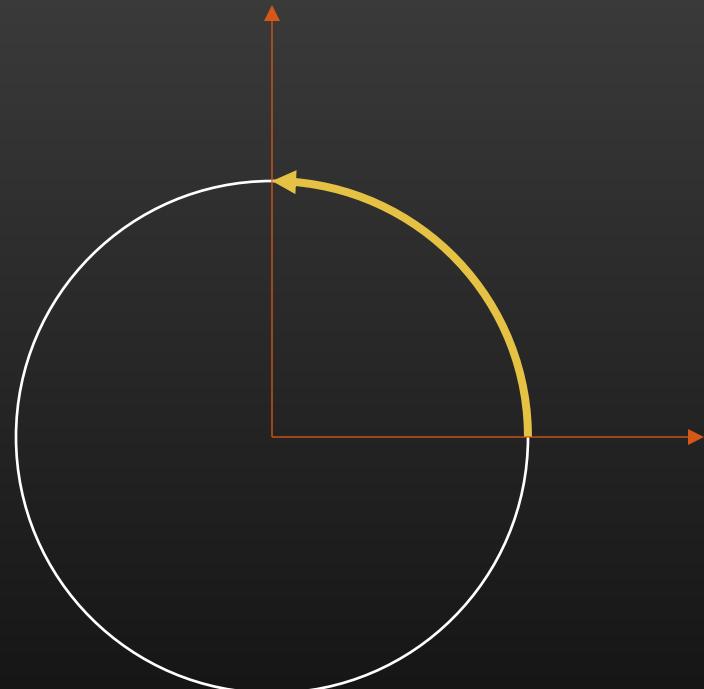
# Representing angles

There are actually **three different ways** of defining angle

- As **distance** around a circle
- As an **inner product**
  - **Projection** of one vector on another
- As an **exterior product**
  - **Area** extruded by one vector along another

# Angle as distance

- Form a **circle** around the two vectors
- **Angle** between two vectors is
  - **Distance** between their intersections
  - Measured **along the perimeter**
  - Normalized by the **radius** of the circle
- **Radians**:  $0 - 2\pi$



# Inner product

- Maps **two vectors  $u, v$  to a real number  $\langle u, v \rangle$**

- **Bilinear**

- Linear in each argument, separately

- **Symmetric**

$$\langle u, v \rangle = \langle v, u \rangle$$

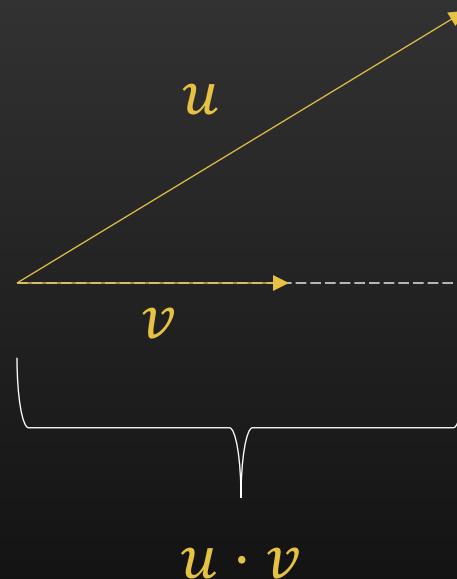
- (this rule gets changed if the coordinates are complex)

- In this class, the inner product will usually be the **dot product**

$$\langle u, v \rangle = u \cdot v = \sum u_i v_i$$

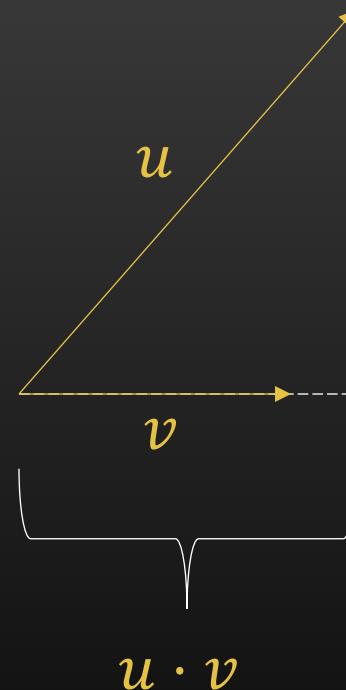
# Inner product as projection

- When one of the vectors is a **unit vector**,
- It measures the **projection** of the other on the unit vector
  - i.e. how far it extends in the direction of the unit vector



# Inner product as projection

- The projection depends on the **angle** between the vectors
- As well as their **lengths**

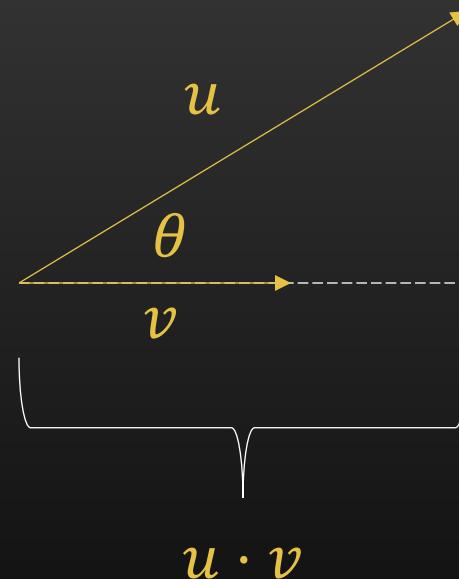


# Inner product as angle

- And if we think of it in terms of **triangles**
- We can see that it's value can be written as

$$u \cdot v = \|u\| \|v\| \cos \theta$$

where  $\|v\|$  means the length of  $v$

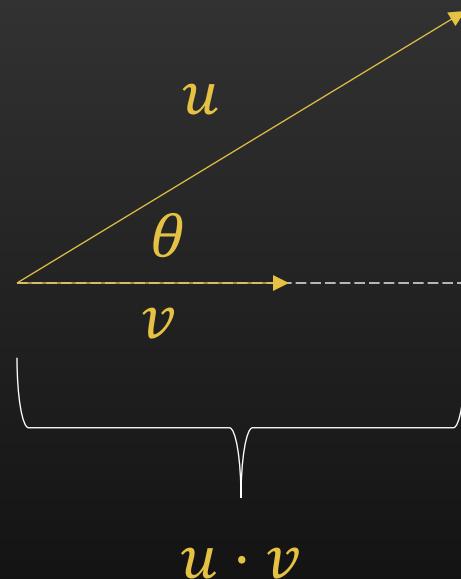


# Inner product as angle

$$\mathbf{u} \cdot \mathbf{v} = \|\mathbf{u}\| \|\mathbf{v}\| \cos \theta$$

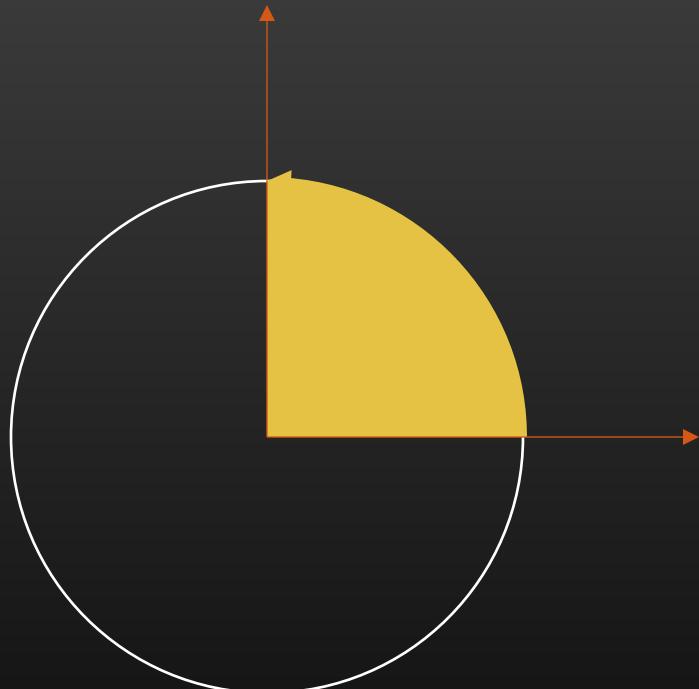
Therefore:

- $\mathbf{v} \cdot \mathbf{v} = \|\mathbf{v}\|^2$
- $\|\mathbf{v}\| = \sqrt{\mathbf{v} \cdot \mathbf{v}}$
- $\theta = \cos^{-1} \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\| \|\mathbf{v}\|}$
- $\mathbf{u} \cdot \mathbf{v} = 0$  when  $\mathbf{u} \perp \mathbf{v}$
- $\mathbf{u} \cdot \mathbf{v}$  is maximal when  $\mathbf{u} \parallel \mathbf{v}$



# Angle as area

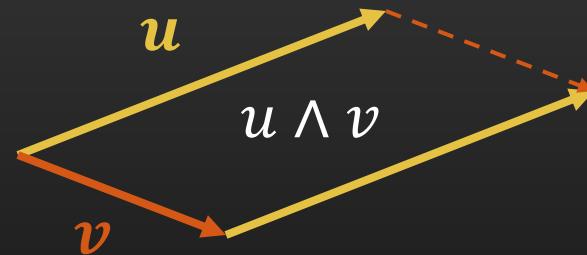
- This is **weirder**
- Form a **circle** around the two vectors
- **Angle** between two vectors is
  - **Area** between their intersections
  - Normalized by the **radius** of the circle



# The exterior product (aka wedge product)

- **Oriented area** created by **sweeping** one vector along another
- Oriented means it has both
  - A **magnitude**
  - A **direction**

We'll relate this to circular areas and angles next week



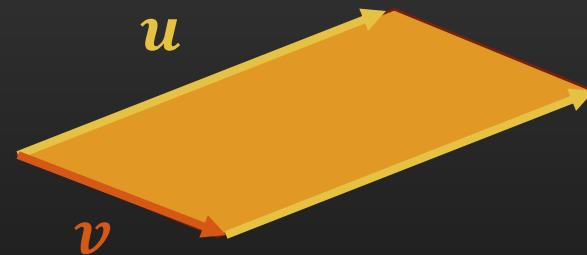
# The exterior product (aka wedge product)

- Start with a vector  $\mathbf{u}$



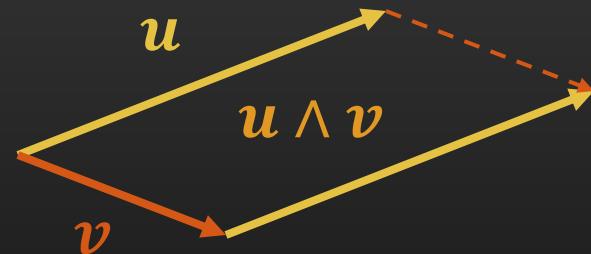
# The exterior product (aka wedge product)

- Start with a vector  $\mathbf{u}$
- Sweep it along another vector  $\mathbf{v}$



# The exterior product (aka wedge product)

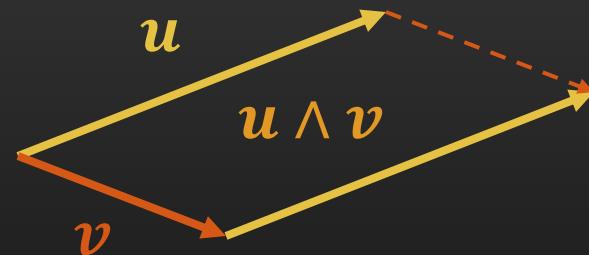
- Start with a vector  $\mathbf{u}$
- Sweep it along another vector  $\mathbf{v}$
- Resulting **area** is the **wedge product**  $\mathbf{u} \wedge \mathbf{v}$



# The exterior product in 2D

In 2D, the area is the **determinant** of the matrix formed by the two vectors:

$$\begin{aligned} u \wedge v &= \left\| \begin{bmatrix} u_1 & u_2 \\ v_1 & v_2 \end{bmatrix} \right\| \\ &= \mathbf{u}_1 \mathbf{v}_2 - \mathbf{u}_2 \mathbf{v}_1 \end{aligned}$$

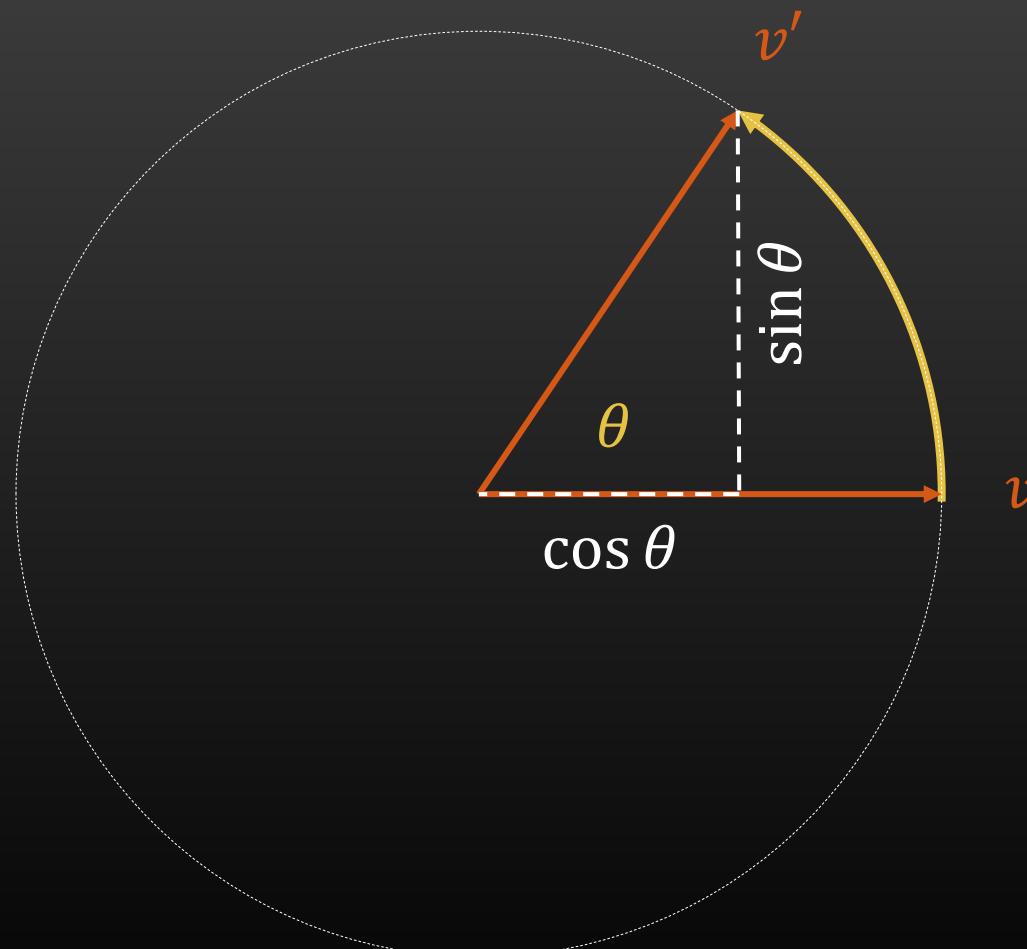


We'll talk about the 3D case later in the quarter

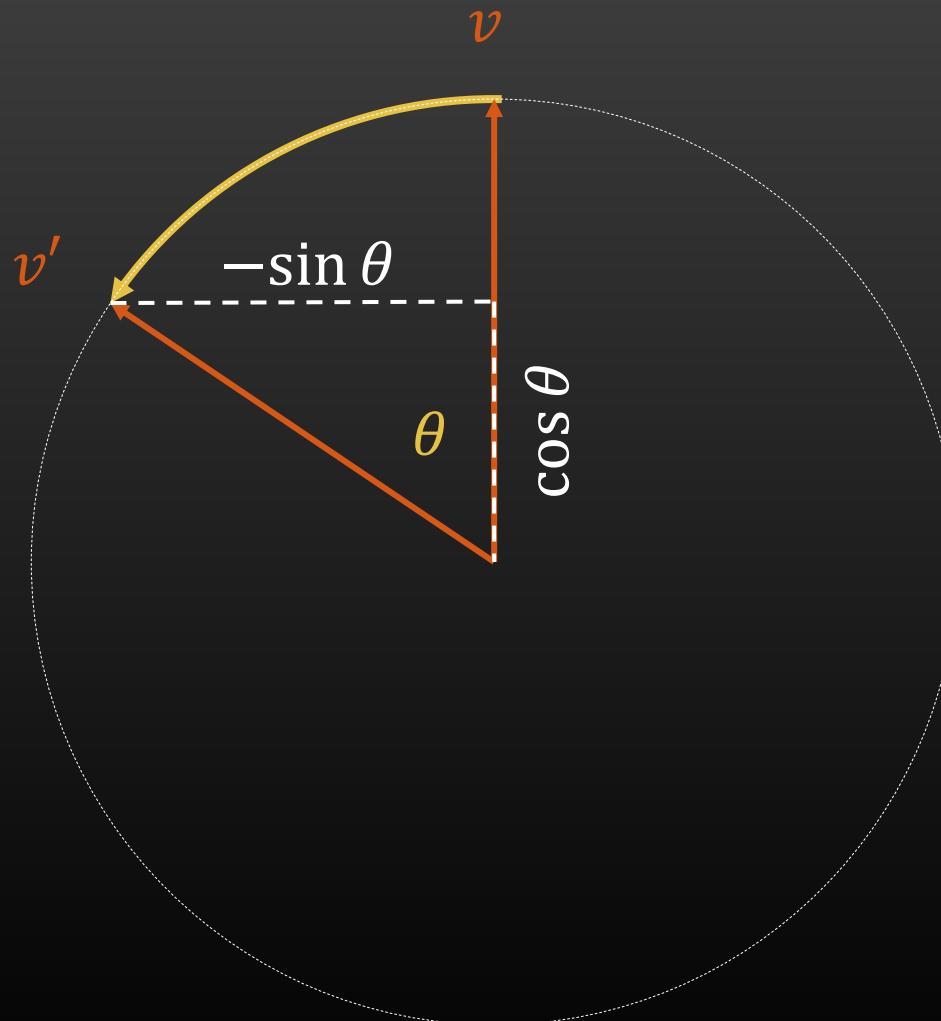
# Representing rotations

- Rotations are relatively **simple** in 2D
- Two **obvious representations**
  - Angle through which to rotate (e.g. in radians)
  - Rotation matrix
- Representing it as an **angle** is fine, but
  - Have to do comparisons  **$mod\ 2\pi$** 
    - More on this later
  - Doesn't help you **compute** the rotation of a vector

# Rotating the x axis



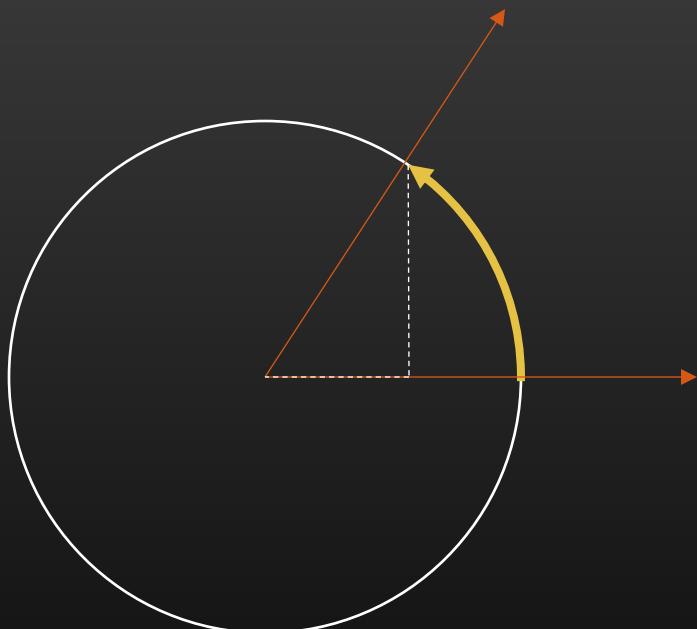
# Rotating the y axis



# Rotation matrix

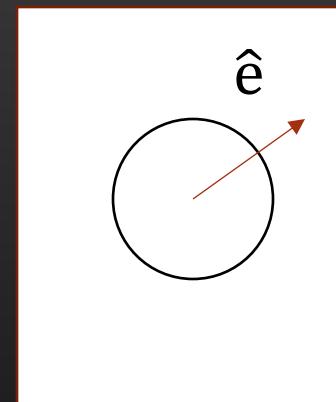
- Rotation is **linear**
  - Doubling the input doubles the output
- We just determined its action on the two axes
- So its **matrix form** is

$$v' = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} v$$

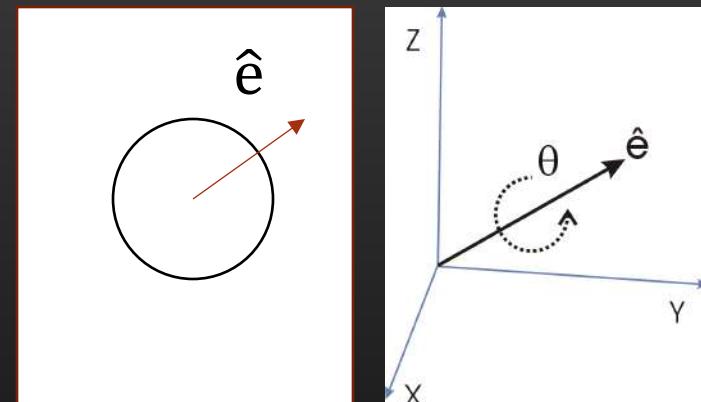


# Representing orientation

- Orientation is how an object is **rotated**
  - In **2d**, this is just the **direction**  $\hat{e}$  the object is “**facing**”
  - In **3d**, there’s another degree of freedom ( $\theta$ ): **rotation about the facing axis**
- We can represent orientation using a **rotation from** some agreed upon **reference orientation**
- So the **pose** of an object is
  - Its **rotation** relative to the reference orientation
  - Plus a **translation** from the origin



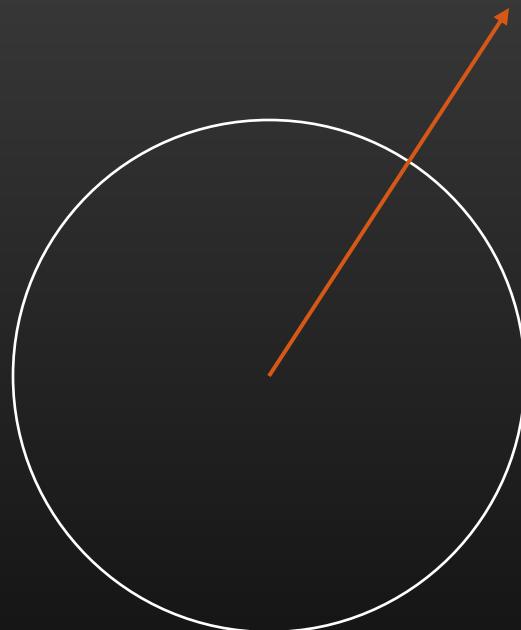
2d



3d

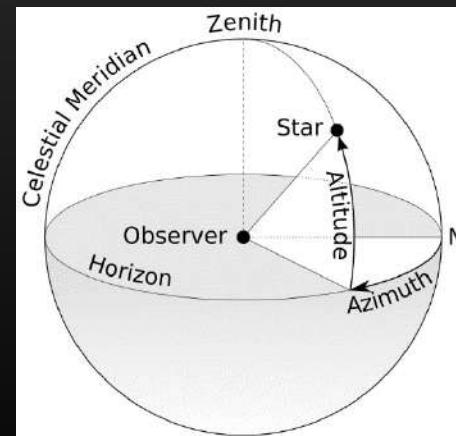
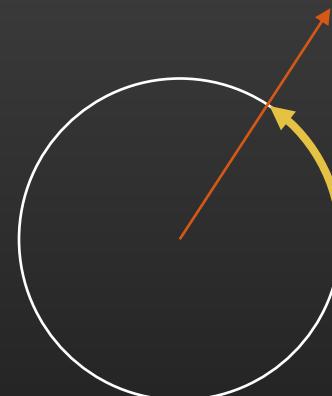
# Representing directions

- How do we represent a **direction** in space?
  - Not a point
  - Not a vector with a specific length
  - **Just the direction**



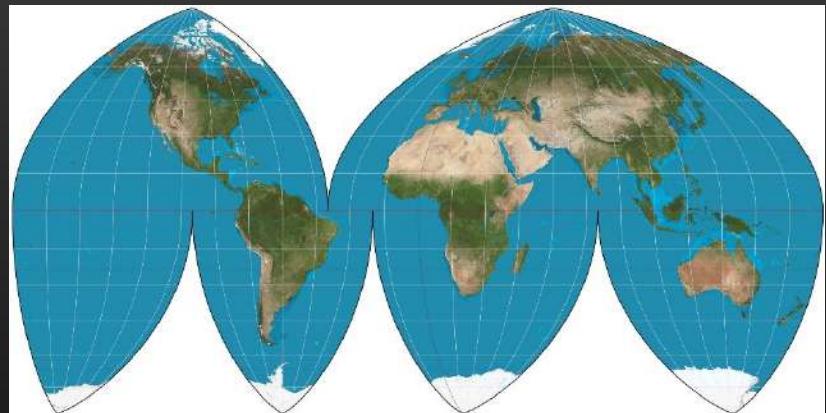
# Representing directions as angles

- In 2D we can represent it with **just an angle**
  - In 3D, it takes two angles
  - Azimuth and attitude
- One problem with this representation is it's **discontinuous**
  - Have to wrap from  $2\pi$  to 0



# Discontinuity

- Ultimately, this is because you **can't**
  - Map a **circular/spherical** thingy in  $n$  dimensions
  - To  $n$  **numbers**
  - Without **discontinuity**
- It's the same reason **flat maps** are always misleading
  - Make things that are adjacent look far apart



# Representing directions as vectors

- Alternatively, we can represent a direction using a **vector parallel to it**
  - Most often, a unit vector
- **Disadvantages**
  - **Non-unique**: many vectors represent the same direction
  - **Redundant**: using a 2-vector to represent a quantity with 1 dof
- Advantage: **continuity**
  - Directions that are close to one another have representations that are close to one another



# Projective coordinates

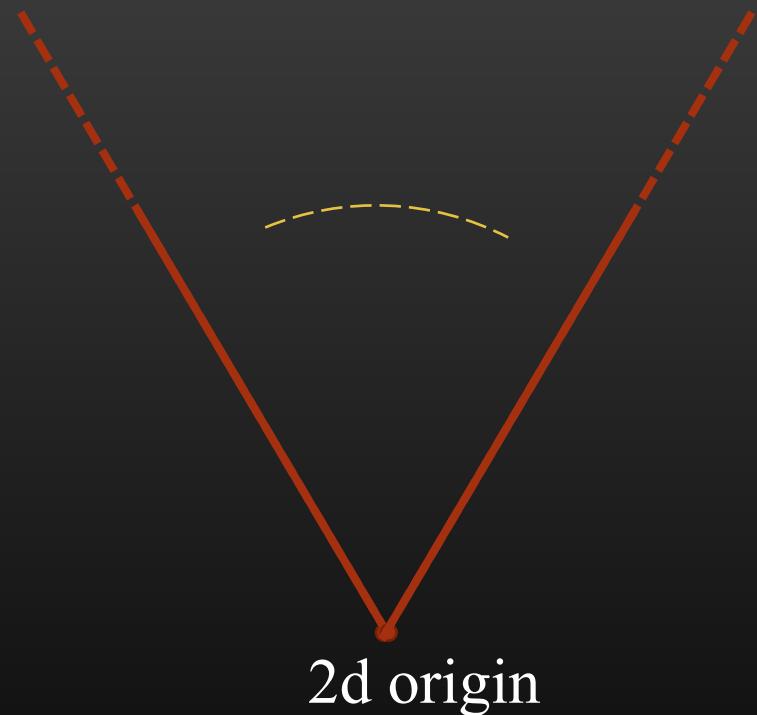
(aka homogeneous coordinates)

# Projective spaces

- This turns out to be a **really useful** technique
- The **projective space** in  $n$  dimensions can be thought of as
  - The set of **lines through the origin** in  $\mathbb{R}^{n+1}$
  - **Vectors from  $\mathbb{R}^{n+1}$** 
    - Not including zero, and
    - Treating parallel vectors as equivalent
- Basic idea is to **add a redundant dimension** to make things easier
- **Computer graphics** typically represents everything internally in terms of projective space
  - Your GPU's basic data type is the **4-vector**

# Simple example

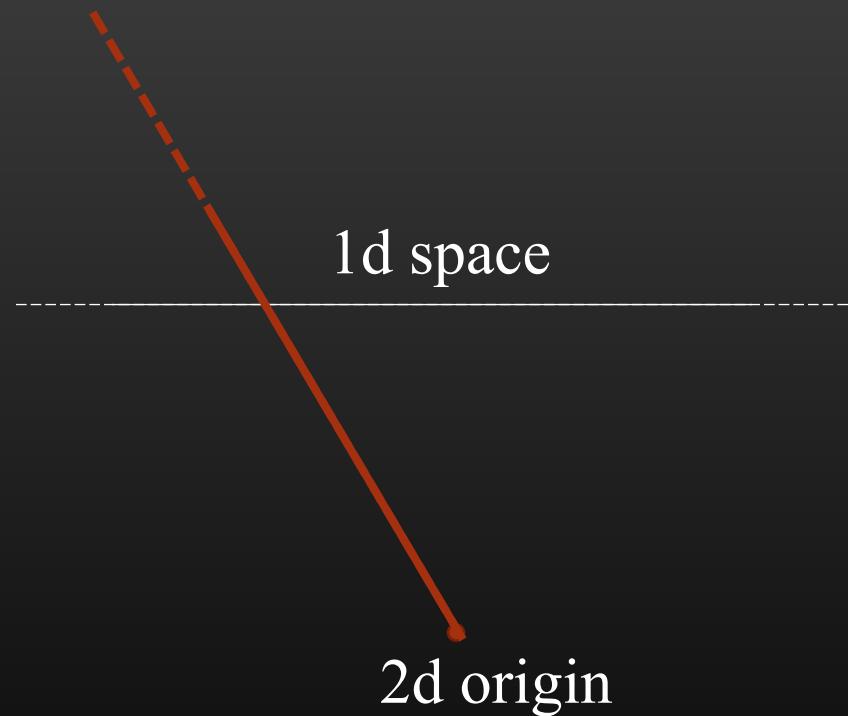
- Let's think about the **1d projective space**, because it's easy to visualize
- It's the space of possible
  - **Lines** through the origin
  - **Directions**
  - In the 2D plane
- Notice that's a **1-dimensional family**
- **Why** would we care?



# Embedding an affine space

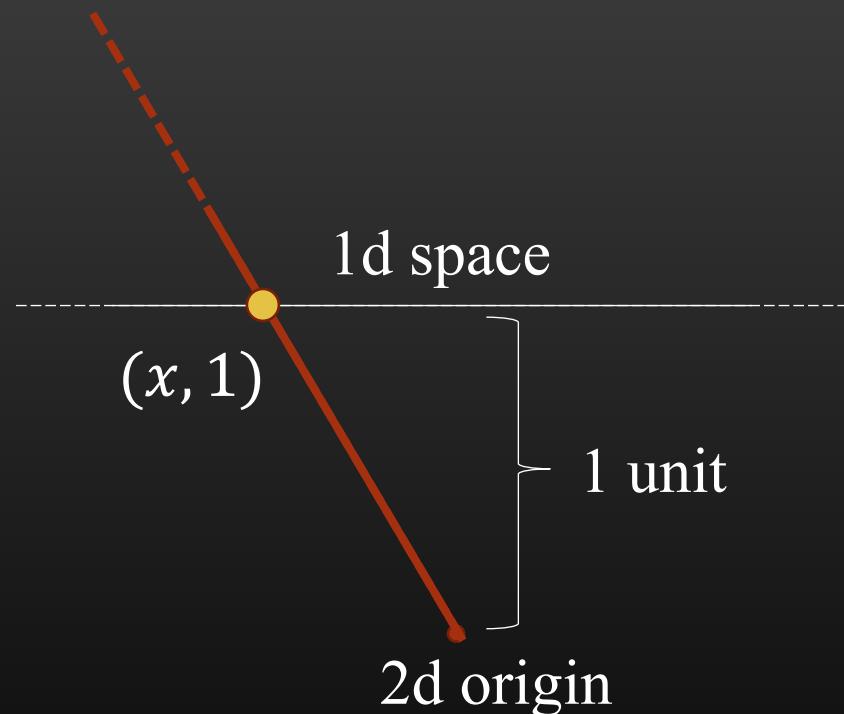
We can **represent** an **affine space** using a projective space

- Take your **1D** affine space
- Draw it as a **line in 2D** space
- Represent
  - A **point** in the **1D** space
  - As the **line** that passes through that point and the origin
- We can do this for **more dimensions**
  - It's just **hard to draw in powerpoint**



# Projective coordinates

- Now we can **represent**
  - A **point**  $x$  in the 1D space
  - Using **any vector** on the line intersecting it
  - Usually the point  $(x, 1)$
- This is a **redundant** coordinate system
  - Can represent any  $x$
  - As any  $(wx, w)$ ,  $w \neq 0$



Who cares?

# Remember how coordinate system transformations weren't linear?

- The **rotation** part was **linear**
- But the **translation** wasn't

# Translation is linear in projective coordinates

- **Represent**  $x$  as  $(x, 1)$ 
  - Or more **generally**  $(wx, w)$ ,  $w \neq 0$
- Suppose we want to **translate** it by amount  $t$ 
  - i.e. add  $t$  to it
- We just do a **matrix multiply**:

$$\begin{bmatrix} 1 & t \\ 0 & 1 \end{bmatrix} (x, 1)^T = (x + t, 1)^T$$

- Or more **generally**:

$$\begin{bmatrix} 1 & t \\ 0 & 1 \end{bmatrix} (wx, w)^T = (w(x + t), w)^T$$

# In 2D

- We **represent**
  - **2D points**  $(x, y)$
  - Using **3 vectors**  $(wx, wy, w)$ , e.g.  $(x, y, 1)$
- A **translation** by  $(t_x, t_y)$  looks like the **matrix**
$$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

# Coordinate transforms

- **Rotation** by  $\theta$  is still **linear**
- We just **pad** out the matrix with zeros and a 1

$$\begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

# Coordinate transforms

- But now we can **rotate and then translate** by multiplying by

$$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta & t_x \\ \sin \theta & \cos \theta & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

- So we can do **both** operations with **one matrix** multiplication

# In 3D

- We **represent**
  - **3D points**  $(x, y, z)$
  - Using **4 vectors**  $(wx, wy, wz, w)$ , e.g.  $(x, y, z, 1)$
- Everything else is **just like 2D**
  - Except **rotation** is a **nightmare**, but we'll talk about that later

# Transform hierarchies

- To find where a character's **hand** is
  - Start with the location of a point in the **hand's coordinate system**
  - Multiply the transform from hand to the **forearm**
  - Multiply by the transform from the forearm to the **upper arm**
  - Multiply by the transform from the upper arm to the **shoulder**
  - Multiply by the transform from the shoulder to the **pelvis**
  - Multiply by the transform from the pelvis to **world coordinates**



# Transform hierarchies

- Oh, and do it for **all the points** on the hand:

$$v'_i = W \left( P \left( S \left( U(F(Hv_i)) \right) \right) \right)$$

for  $0 \leq i <$ large number

- That's a **lot of work!**



# Transform hierarchies

- Oh, and do it for **all the points** on the hand:

$$v'_i = W \left( P \left( S \left( U \left( F(Hv_i) \right) \right) \right) \right)$$

for  $0 \leq i <$ large number

- Oh, but wait, matrix multiplication is **associative!**

$$v' = (WPSUFH)v'$$

- Compute the matrix once
- Multiply all points by only one matrix



And that is how we  
model **space**

(mostly) using only numbers, addition, and multiplication

# Appendix 1

How does this relate to the **Unity API** ?

# Points, vectors, and directions

- Unity doesn't have separate point and vector types
- So you have to **keep track of** when a vector is a vector and when it's **really a point**

# Vector types

- Unity provides **vector types** for 2-4 dimensions
  - **Vector2**
  - **Vector3**
  - **Vector4**
- Internally, your **GPU** does everything with **Vector4**
- But Unity will **autoconvert**
  - **2D to 3D**
    - $(x, y) \rightarrow (x, y, 0)$
  - **3D to 4D**
    - $(x, y, z) \rightarrow (x, y, z, 1)$
    - Unless except in a few cases where it know you mean a vector rather than a point, then  $(x, y, z) \rightarrow (x, y, z, 0)$

# Matrices

- All matrices in Unity (and most graphics systems) are **4×4**
- Basic data type is called **Matrix4x4**
- **Methods**
  - Vector3 **MultiplyPoint**(Vector3)
    - Converts vector to projective coordinates
    - Transforms it
    - Converts it back
  - Vector3 **MultiplyVector**(Vector3)
    - Same, but for vectors representing displacements rather than points.

# The transform component

- Every game object has a built-in component of type **Transform**
  - In the field named **transform**
- Specifies
  - **Parent** coordinate system
  - Matrix from **local coordinates to parent's** coordinates
- But you mostly don't have to deal with the actual matrix

# Transform properties

These are derived from the internal matrix and parent coordinate systems

- **parent**
  - Transform component of the parent object
- **position**
  - Position of the object in global coordinates
- **localposition**
  - Position of the object in relative to parent
- **rotation**
  - Rotation relative to global coordinates
  - Specified as a weird thing called a quaternion, which we will talk about later
- **localrotation**
  - Rotation relative to parent

# Transform properties

These are derived from the internal matrix and parent coordinate systems

- **forward**

- Vector pointing in object's "forward" direction in global coordinates
- Don't use this for 2D, it's really the local Z axis converted to global coordinates

- **up**

- Vector pointing in the object's local Y axis, converted to global coordinates

- **right**

- Vector pointing in the object's local X axis, converted to global coordinates

# Transform properties

These are derived from the internal matrix and parent coordinate systems

- **localToWorldMatrix**

- Matrix that transforms points from local to global coordinates

- **worldToLocalMatrix**

- Matrix that transforms points from global to local coordinates

# Transform methods

These are derived from the internal matrix and parent coordinate systems

- **Vector3 TransformDirection(Vector3)**

- Transforms a direction vector from local to global coordinates
- This works for pseudovectors

- **Vector3 TransformPoint(Vector3)**

- Transforms a point from local to global coordinates

- **Vector3 TransformVector(Vector3)**

- Transforms a displacement vector from local to global coordinates

# Transform methods

These are derived from the internal matrix and parent coordinate systems

- **Vector3 InverseTransformDirection(Vector3)**

- Transforms a direction vector from global to local coordinates

- **Vector3 InverseTransformPoint(Vector3)**

- Transforms a point from global to local coordinates

- **Vector3 InverseTransformVector(Vector3)**

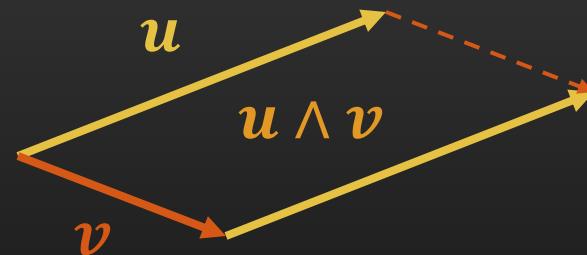
- Transforms a displacement vector from global to local coordinates

# Appendix 2

More on the **exterior product**, for the curious

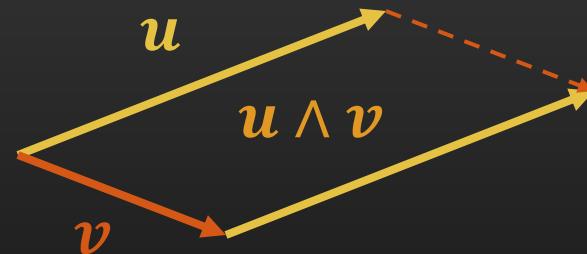
# Properties of the exterior product

- **Bilinearity**  
(linear in both arguments)
- **Antisymmetry**  
 $u \wedge v = -(v \wedge u)$
- Consequently
  - $u \wedge v = 0$  when  $u \parallel v$
  - Maximal when  $u \perp v$
- Kind of the **opposite** of  
the **inner product**



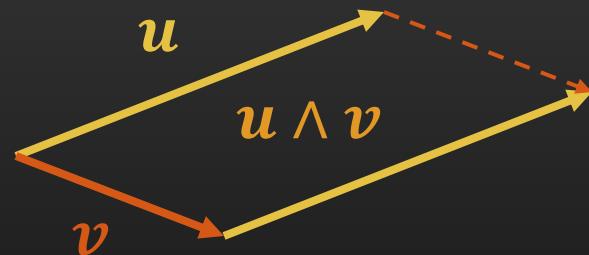
# Exterior product and the cross product

- The wedge product is sort of a **generalization** of the cross product
  - Works in any number of dimensions
- Represents an **area of a plane** formed by two vectors
  - In 3D, there are **2 degrees of freedom** to that plane
  - In 2D there are **zero** (there's only one plane)



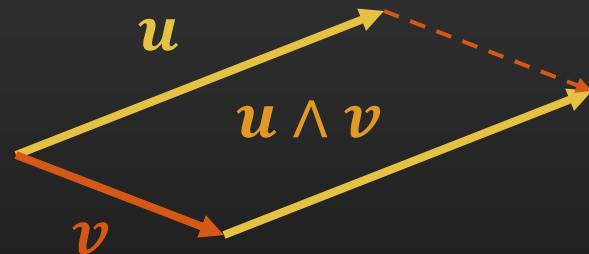
# Bivectors

- So the result of  $\wedge$  looks like
  - A **scalar** in 2D
  - A **vector** in 3D
- It's actually **neither**
- It's a third thing called a **bivector**
- Specifies
  - **Orientation** of the plane
  - Amount of **area** in it



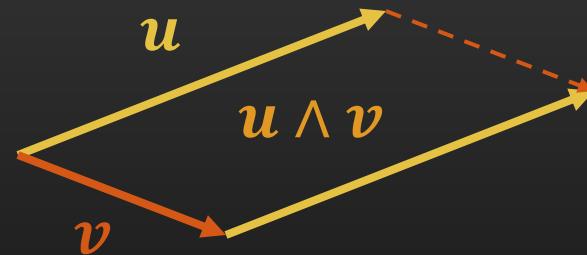
# Bivectors in 2D

- In 2D, there's **only one plane**
- So a **bivector** has
  - **0 degrees** of freedom for the **orientation** of the plane
  - **1 dof** for the **area**
  - **1 dof total**
- So a bivector **looks like a scalar**
  - Also known as a **pseudoscalar**



# Bivectors in 3D

- In 3D, there are many planes
- So a **bivector** has
  - 2 degrees of freedom for the orientation of the plane
  - 1 dof for the area
  - 3 dof total
- So a bivector **looks like a vector**
  - But technically lives in a different linear space
  - And obeys somewhat different laws



Linearity is your friend

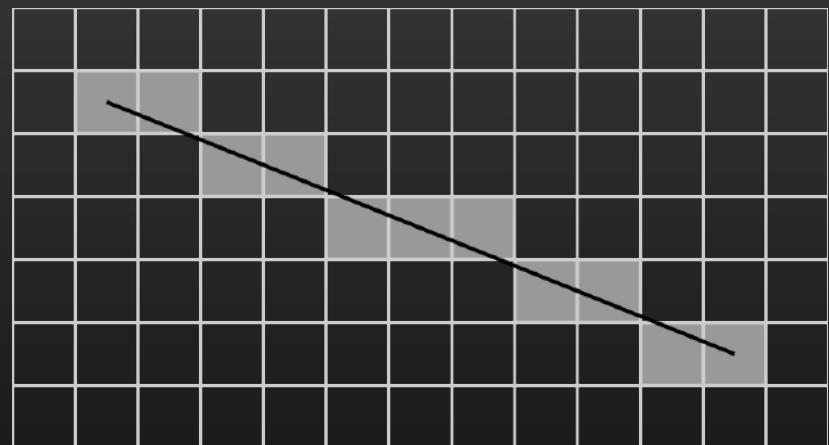
# Basic polygon **rendering**

# Low-level drawing interface

- Under the hood, the hardware typically supports two **primitive shapes**
  - Drawing **lines**
  - Drawing polygons, specifically a **triangles**
- Everything else gets **reduced** to these operations
  - Drawing a complex shape is reduced to drawing a set of polygons
  - Drawing complex polygons is reduced to drawing triangles

# Rasterization

- Lines and triangles are ideal **mathematical objects**
  - **Infinite** resolution
- But your video display is a **raster**
  - **Finite** collection of pixels
- The process of deciding which pixels correspond to a given primitive is called **rasterization**



# Geometry vs. material

- Computer graphics generally divides drawing into **two issues**
  - Material: what **color(s) to draw**
  - Geometry **where to draw** it
- These get handled by separate systems in the GPU
- For the moment, we'll assume the material is always some fixed color

# Coordinate systems

- The **position** field of the **Transform** component of a game object gives the object's position  $w$  in the global coordinate system, aka **world coordinates**
- But **drawing** is done in **pixel coordinates**  $s$  on the screen
- There may also be other coordinate systems in use, but we'll ignore these for the moment

# Projection

- In 2D, the mapping from **world to screen coordinates** consists of
  - A translation
  - A rotation
  - A scaling
- All of these are expressible as **matrices** in projective coordinates
  - And so we can **compose** them using matrix multiplication to make **one matrix**  $M$
- So given
  - World coordinates  $w$
  - Screen coordinates  $s$
  - Projection matrix  $M$
- We have that
$$s = Mw$$

# Linear interpolation (aka “lerp”)

- Let  $\mathbf{a}$  and  $\mathbf{b}$  be **two points in some linear space**
- Let  $s \in [0,1]$  be a real number
- The linear **interpolation** of  $\mathbf{a}$  and  $\mathbf{b}$  is:

$$\text{lerp}(\mathbf{a}, \mathbf{b}, s) = \mathbf{a} + s(\mathbf{b} - \mathbf{a})$$

- So that:
  - $\text{lerp}(\mathbf{a}, \mathbf{b}, 0) = \mathbf{a}$
  - $\text{lerp}(\mathbf{a}, \mathbf{b}, 1) = \mathbf{b}$
  - $\text{lerp}(\mathbf{a}, \mathbf{b}, 0.5) = \text{midpoint of } \mathbf{a} \text{ and } \mathbf{b}$

# Lerp commutes with linear maps

Given some **linear function**  $f$ :

$$\begin{aligned} f(\text{lerp}(\mathbf{a}, \mathbf{b}, s)) &= f(\mathbf{a} + s(\mathbf{b} - \mathbf{a})) \\ &= f(\mathbf{a}) + s(f(\mathbf{b}) - f(\mathbf{a})) \\ &= \text{lerp}(f(\mathbf{a}), f(\mathbf{b}), s) \end{aligned}$$

So we can:

- lerp first, and then apply  $f$ , or
- Apply  $f$  first and then lerp

We get the same result either way

# Line rasterization

# Drawing a point

- Suppose we want to draw **just one point**, given its world coordinates
- We just
  - Compute its **screen coordinates**  $s = Mw$
  - Round to the **nearest pixel**
  - **Set that pixel** to the desired color

# Drawing a line

- When we draw lines on the screen, we're technically drawing **line segments**
- Suppose we want to draw a segment
  - Given its **endpoints**  $w_1, w_2$  in world space
- Draw the screen **projections of all the points** on the segment

# A bad line drawing algorithm

**DrawLine**( $w_1, w_2$ )

For each  $t$  from 0 to 1 (going in some small steps)

$$w = \text{lerp}(w_1, w_2, t)$$

$$s = Mw$$

set pixel at  $s$  to desired color

- This has the problem that you **don't know what step size** to use
  - Too small and you're **wasting time**
  - Too big and you have **gaps in the line**

# Linearity is your friend

- That algorithm had us **lerp'ing, then projecting** (multiplying by  $M$ )
  - That means we do a lot of projecting
- But we know that
  - **Projection is linear** (in projective coordinates)
  - **Lerp is bilinear**
  - So we can **project first, then lerp** (in projected coordinates)
  - Only have to project the endpoints

# A better line drawing algorithm

**DrawLine**( $w_1, w_2$ )

$s_1 = Mw_1$

$s_2 = Mw_2$

For each  $t$  from 0 to 1 (going in some small steps)

$s = \text{lerp}(s_1, s_2, t)$

set pixel at  $s$  to desired color

- In theory, this also makes it easier to decide what the step size is (e.g. you have one step for each position along the X axis)
- But in practice, you really use...

# Line rendering in real life

In real life, you either

- Use the **Bresenham algorithm**
  - An optimized line-drawing algorithm
  - Operates on screen coordinates
    - So you still have to project
  - Uses only integer arithmetic
  - Avoids division
- Or draw line as really **thin triangles**
  - So now let's talk about drawing triangles

# Triangle rasterization

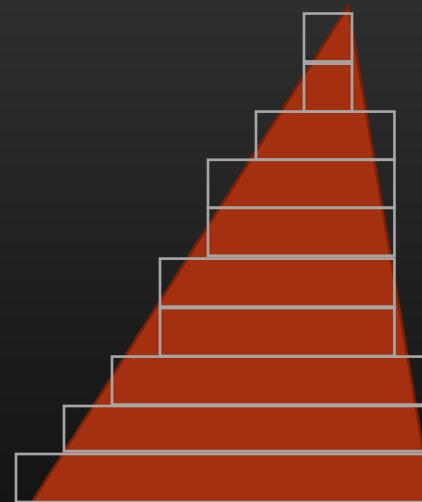
# Drawing a flat-bottom triangle

- **Easy to draw** a flat-bottom or flat-top triangle
  - Triangle with a side **parallel to the X axis**



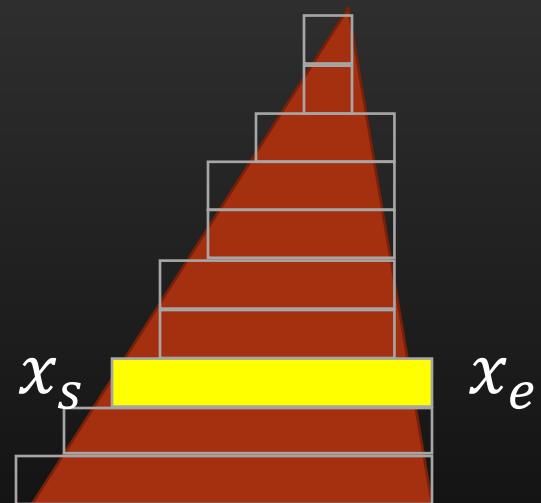
# Drawing a flat-bottom triangle

- **Easy to draw** a flat-bottom or flat-top triangle
  - Triangle with a side **parallel to the X axis**
- Its raster form is a series of **horizontal scan lines**



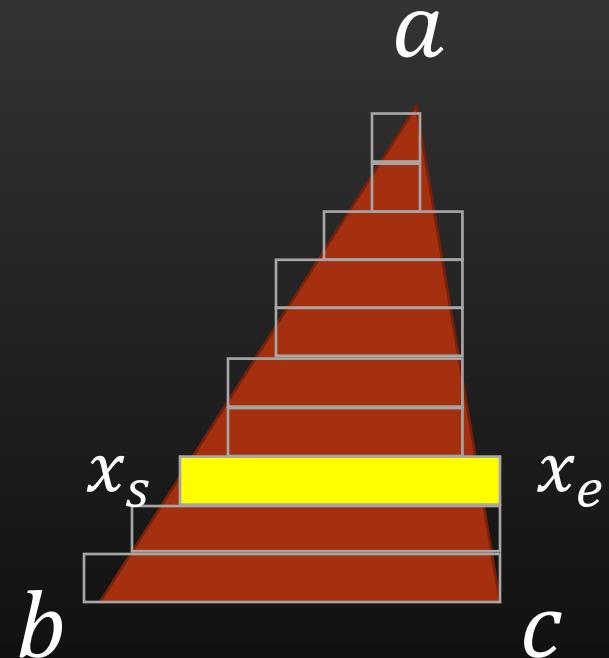
# Sketch of algorithm

- Drawing a scan line is easy
  - Find its Y coordinate
  - And the X coordinates of its start and endpoints
  - Mark all the pixels inbetween
- How do we find  $x_s$  and  $x_e$ ?



# Sketch of algorithm

- Let  $a, b, c$  be the coordinates of the **corners**
- We get
  - $x_s$  by **lerp'ing** from  $a$ 's X coordinate to  $b$ 's X coordinate
  - And  $x_e$  by **lerp'ing** from  $a$  to  $c$ .



# Sketch of algorithm

**DrawFlatTri**( $a, b, c$ )

$$\Delta y = a_y - b_y$$

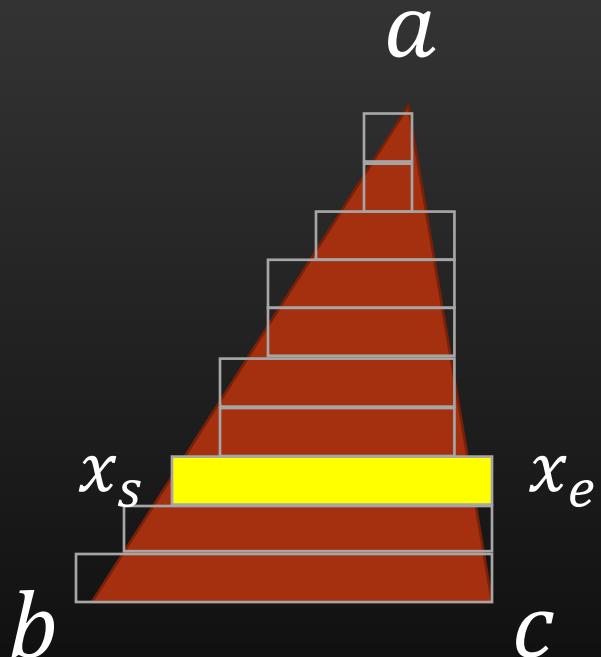
for  $t = 0$  to  $\Delta y$

$$y = b_y + t$$

$$x_s = \text{lerp}(b_x, a_x, \frac{t}{\Delta t})$$

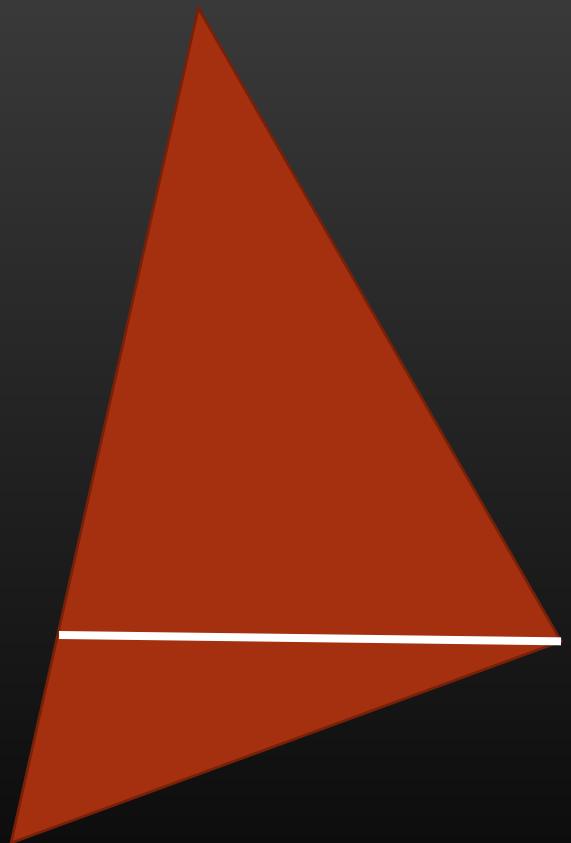
$$x_e = \text{lerp}(c_x, a_x, \frac{t}{\Delta t})$$

fill pixels from  $(x_s, y)$   
to  $(x_e, y)$



# Drawing a general triangle

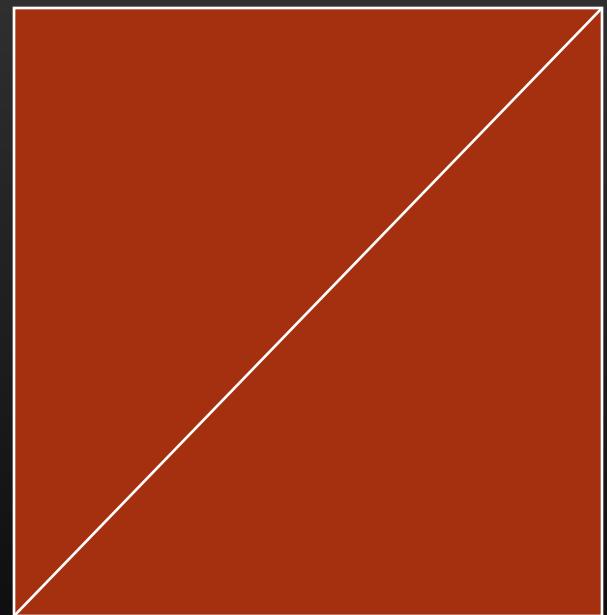
- A general triangle is just **two flat triangles**
  - Flat bottom
  - On top of flat top
- So we can draw it by **reducing it** to simpler cases



Drawing  
convex polygons

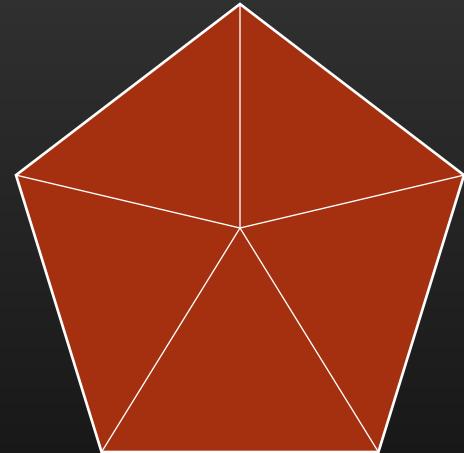
# Drawing a quad

- A quadrilateral (quad) is just **two triangles**



# Drawing a general convex polygon

- Any other convex poly can be drawn as a **triangle fan**
  - Pick an **arbitrary point** in the interior
  - Make tris from that **point and adjacent vertices** of the poly



# Materials and shaders

# Materials

- Great!
- Now we know how to figure out which pixels to color in
- How do we determine **what color to fill it with?**

# Materials

- That's determined by the **material**: the **shader and its parameters**
  - The shader is a **GPU subroutine**
  - Takes a set of coordinates (and any other parameters), and **returns a color**
- These get complicated for 3D, but here are a few **simple kinds** we'll use for 2D

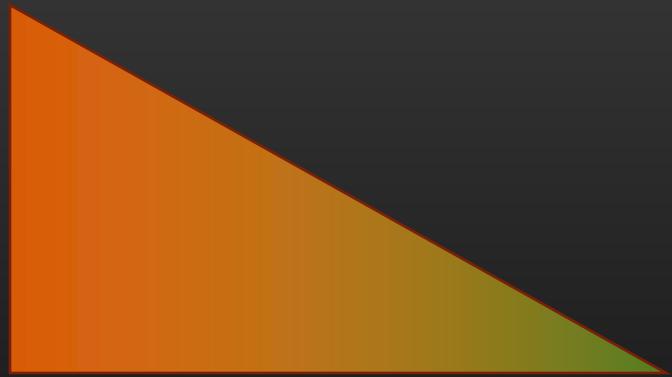
# Simple materials: a fixed color

- Shader **ignores its arguments**
- Always returns a fixed color



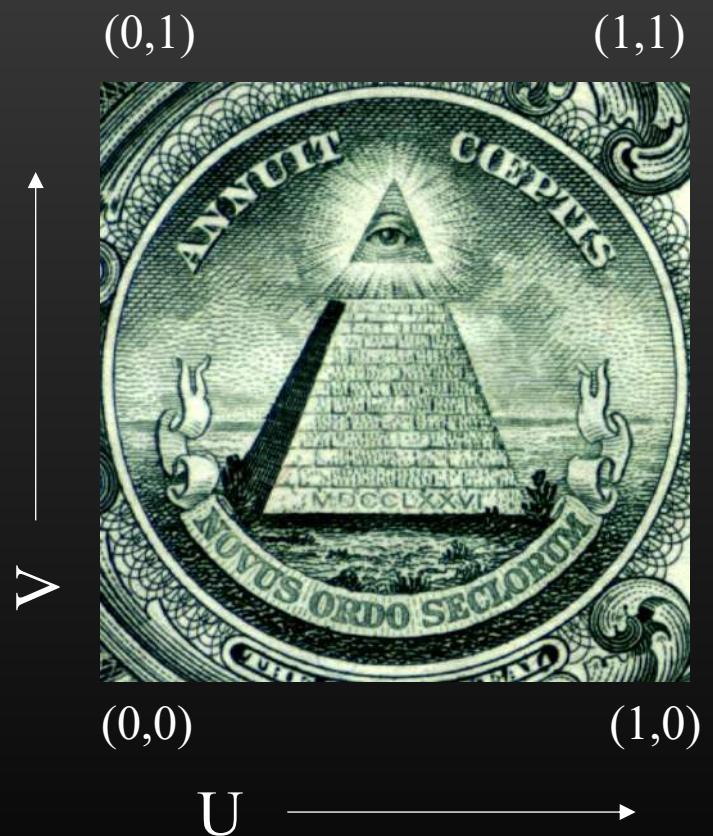
# Simple materials: vertex-interpolated color

- Separate **colors specified for each vertex** (corner)
- Color of an interior pixel is **linearly interpolated** from the colors of the vertices



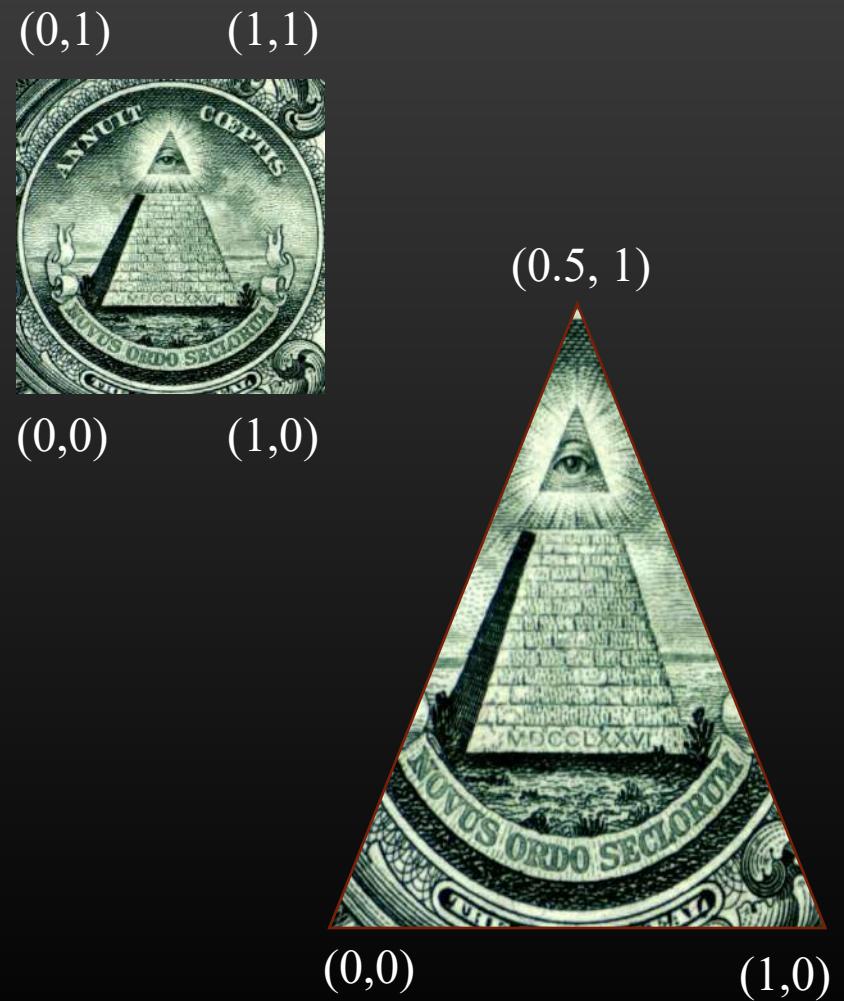
# Simple materials: texture maps

- Load a **raster image** into memory
  - Aka a texture
- Assign it a coordinate system
  - $(0,0)$  = lower left corner
  - $(1,1)$  = upper right corner
  - Called **UV coordinates**
  - To distinguish it from the XYZ coordinates of game objects



# Simple materials: texture maps

- Tag each triangle **vertex** with **UV** coordinates
- **Linearly interpolate** the coordinates for pixels inside the triangle
- Assign each pixel the **color from the image** at the interpolated coordinates



I did warn you that  
linearity was going to  
come up a lot

# Transparency

# Rendering transparent colors

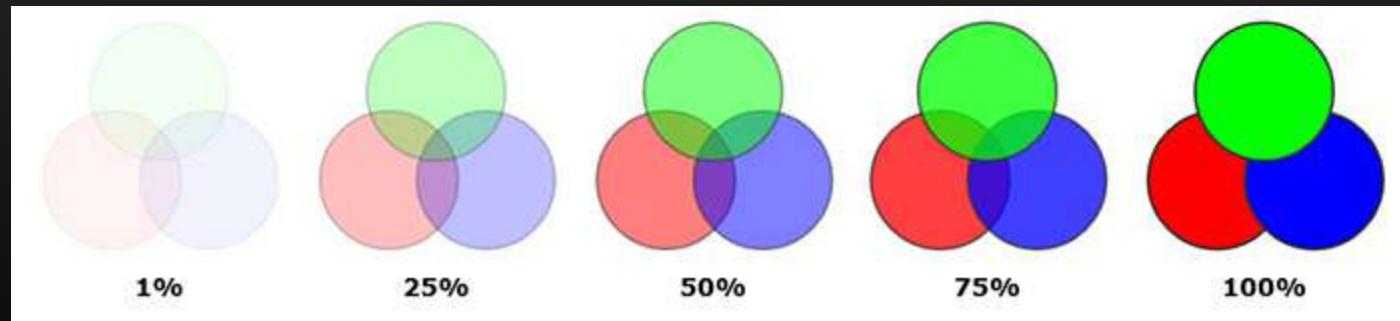
- In addition to **R, G, and B** (red, green, and blue) values
- Colors internally also have an  **$\alpha$  (alpha)** value representing **opacity**
  - 0 means color is completely **transparent**
    - Painting with the color on another color leaves the color below **unchanged**
  - 1 means the color is completely **opaque** (if on a 0-1 scale, else 255)
    - Painting with the color completely **replaces** whatever is below

# Alpha blending

$$O_R = I_\alpha I_R + (1 - I_\alpha) O_R = \text{lerp}(O_R, I_R, I_\alpha)$$

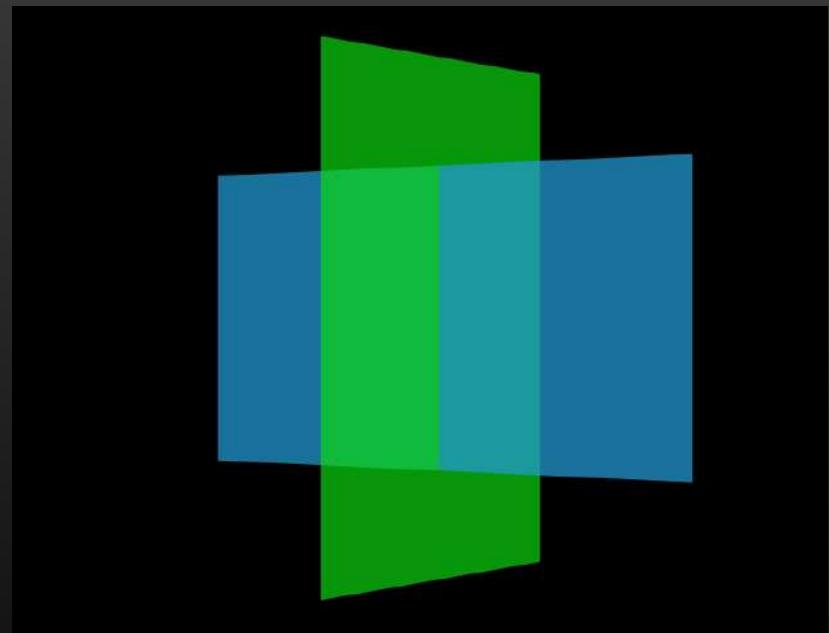
$$O_G = I_\alpha I_G + (1 - I_\alpha) O_G = \text{lerp}(O_G, I_G, I_\alpha)$$

$$O_B = I_\alpha I_B + (1 - I_\alpha) O_B = \text{lerp}(O_B, I_B, I_\alpha)$$

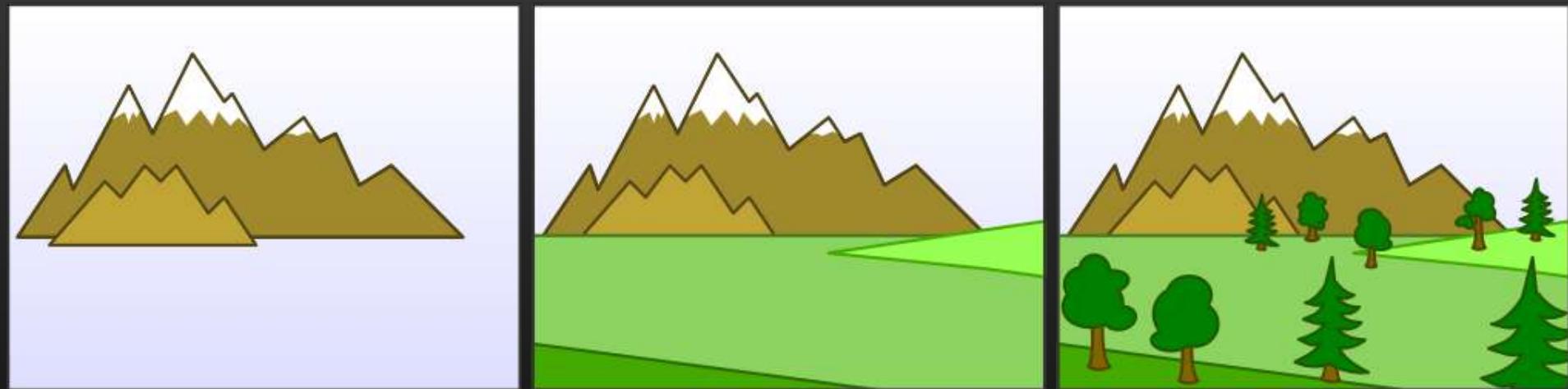


# Depth-ordering

- The problem with alpha blending is that it's **not commutative**
  - Color for **A in front of B**
  - Is different from color for **B in front of A**
- So we not only have to **render all** the surfaces
- We have to render them **back to front**



# Painter's algorithm

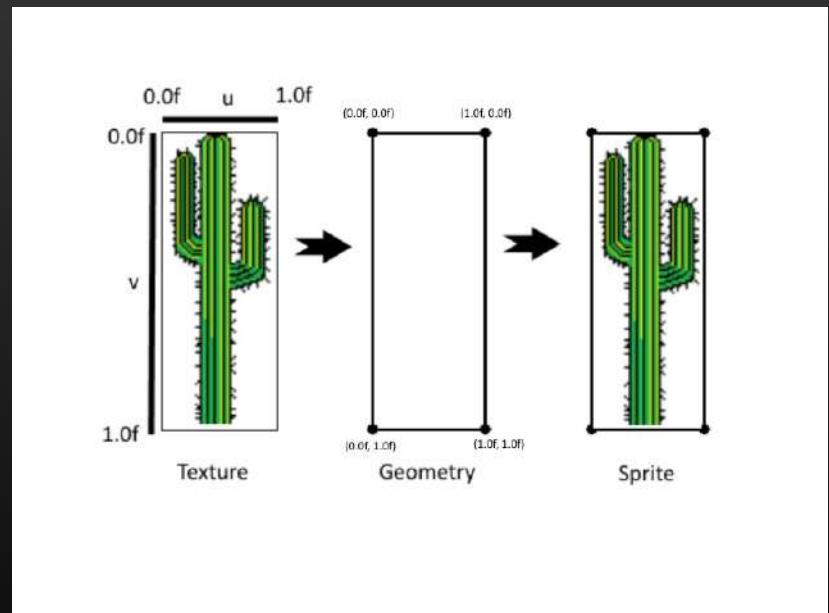


# Sprite rendering

# Sprite rendering

Rendering **2D sprites** is easy

- Just render **flat quads** using the 3D hardware
- **Texture map** them with the sprite
- Use **transparency** to handle non-rectangular shapes
  - Unused pixels left transparent
  - Note if all the **alpha values are either 0 or 1**, but nothing in between, we **don't have to depth sort**



# Sprite sheets

- Sprites are typically formed into **packed textures** called sprite sheets
- This ends up being more efficient
  - Because switching textures slows down the GPU



# Character sprite sheets

- And for **animation loops** for characters
- The different **frames of the animation** are packed into a sprite sheet



# Rendering in Unity

# SpriteRenderer component

- Most 2D games can get by using Unity's **built-in sprite system**
- Add SpriteRenderer component to your GameObject
- Drag image file for sprite to your Assets folder
- Set the SpriteRenderer's **Sprite field** with the sprite you want to render



# Unity Materials

- Unity has a **Material class**
- Material objects **represent**
  - A **shader** to run in the GPU for drawing
  - A collection of **parameter** settings for the shader
- Parameters vary by shader, but include
  - **mainTexture**
    - Texture map to draw
    - Should be of type Texture2D
  - **color**
    - Color to draw in, or to use to tint the Texture
    - Should be of type Color

# GL class and OnRenderObject()

- Unity includes a subset of the **OpenGL** graphics API
- Included in the GL static class
- However, GL can **only be used** from inside specific methods
  - Particularly **OnRenderObject()**
  - Called by Unity renderer after a given GameObject has completed its normal rendering process
  - This is where you can put exotic custom stuff

# Example

```
void OnDrawObject() {  
    myMaterialObject.color = Color.green;  
    myMaterialObject.SetPass(0);  
    GL.Begin(GL.Quads);  
    GL.Vertex3(0,0,0);  
    GL.Vertex3(0,1,0);  
    GL.Vertex3(1,1,0);  
    GL.Vertex3(1,0,0);  
    GL.End();  
}
```

# Configuring the shader

- Start by setting the material's
  - **color**, if the material supports tinting
  - **mainTexture**, if you're using a texture map
- Then call the material's **SetPass(0)** method
  - This will start the shader running

# GL.Begin / GL.End

- Used to issue draw commands for **primitives**
- Call **Begin**, specifying primitive type
  - **GL.LINES**
  - **GL.TRIANGLES**
  - **GL.QUADS**
- **Generate vertices** for each primitive
- Call **End()**

```
GL.Begin(GL.TRIANGLES);  
... vertices for 1st tri ...  
... vertices for 2nd tri ...  
...  
GL.End();
```

# Specifying vertices

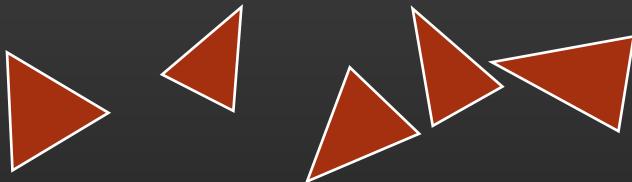
- GL.Vertex/Vertex3, **submit vertices** to the GPU
- You can optionally specify **color** or **texture coordinates**
  - But use GL.Color/TexCoord2 **before calling GL.Vertex**
- The GPU draws a primitive when it has **enough vertices for the current primitive type**
  - 2 for line
  - 3 for triangles
  - 4 for quads

```
void GL.Vertex3(float, float, float)  
void GL.Vertex(Vector3)  
void GL.Vertex(Vector2)  
void GL.Color(Color)  
void GL.TexCoord2(float, float)
```

# Winding order

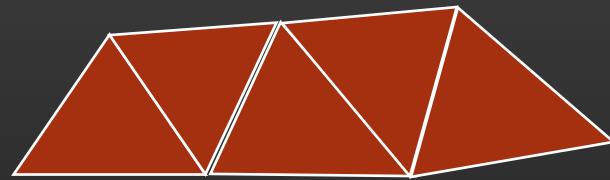
- Unity assumes you provide the vertices of polygons in **clockwise order**
  - For **efficiency reasons** we'll explain when we get to 3D
- You can **start anywhere** on the polygon
  - But you have to move around the polygon in a clockwise order

# Representing a batch of connected triangles



## GL.TRIANGLES

- **General**, but have to specify every vertex of every triangle
- Sequence of **unconnected** triangles
- Sequence of  **$3n$  vertices**
  - $a_0, b_0, c_0, a_1, b_1, c_1, \dots, a_n, b_n, c_n$
  - Where  $i$ th tri is  $a_i b_i c_i$



## GL.TRIANGLESTRIP

- **Efficient** but limited to chains of connected tris
- Sequence of  **$n + 2$  vertices**
  - Each adjacent pair represents a segment
  - $v_0, v_1, \dots, v_{2n}$
  - Where  $i$ th tri is  $v_i, v_{i+1}, v_{i+2}$
- **First tri is clockwise** winding order, subsequent tris are specified with just one extra vertex

# Transform matrices in GL

- GL breaks up the transform matrix into **two parts**
  - **Model/view matrix**  
Specifies the mapping from object coordinates to “camera coordinates”
  - **Projection matrix**  
Specifies the mapping from camera coordinates to screen coordinates
  - We'll go into more detail on these when we get to 3D
- So the real transform applied by the GPU is the **product of these two**

# Matrices on entry to OnRenderObject

- Model/view matrix set to map current **GameObject's local coordinates to coordinates of the current Camera** object
- Projection matrix is set to the projection parameters of the **current Camera** object

# Changing the matrices

- You can change the **model/view** matrix using
  - **GL.LoadIdentity()**  
Resets transform back to identity matrix
  - **GL.MultMatrix(Matrix4x4)**  
Applies additional transform to current transform
- You can change the **projection** matrix using
  - **GL.LoadOrtho()**  
Forces the projection matrix to be a orthographic projection, i.e.  
(x,y,z) in camera coordinates maps to (x, y) in screen coordinates
  - **GL.LoadProjectionMatrix(Matrix4x4)**  
Sets the projection matrix to whatever you like
- And you can reset **both** using
  - **GL.LoadPixelMatrix()**  
Resets both matrices so that coordinates (x,y,z) map to screen pixel  
(x,y)

# Setting the matrices back

- It's generally a bad idea for your OnRenderObject method to mess up the graphics settings
- So if you're going to change the transform, wrap it in:
  - **GL.PushMatrix()**  
Saves the current matrices
  - **GL.PopMatrix()**  
Restores the last saved matrix information

# Physics in 2D

Or: how I spent my summer vacation making particle sims

# Today

- Start talking about how **physics engines** work
- And various ways physics simulation can go **horribly wrong**
- You won't need to build a physics engine for this class
- But you'll want to **understand how they work** so you can understand
  - Their **limitations**
  - Why your stack of crates suddenly **exploded**

# Physics

- Ultimately governs all **behavior of matter**
- In games, **physics simulation** is used for **realistic motion of objects** under the influence of
  - **Collisions**
  - **Gravity**
- **Not generally used for character motion**, save for
  - **Ragdoll** physics: motion of limp bodies
  - Handling **collisions during an animation**

# Models of physics

- **Human scale**
  - Aristotelian
  - Newtonian
- **Atomic scale**
  - Quantum mechanics, QED
- **Astronomical scale**
  - General relativity

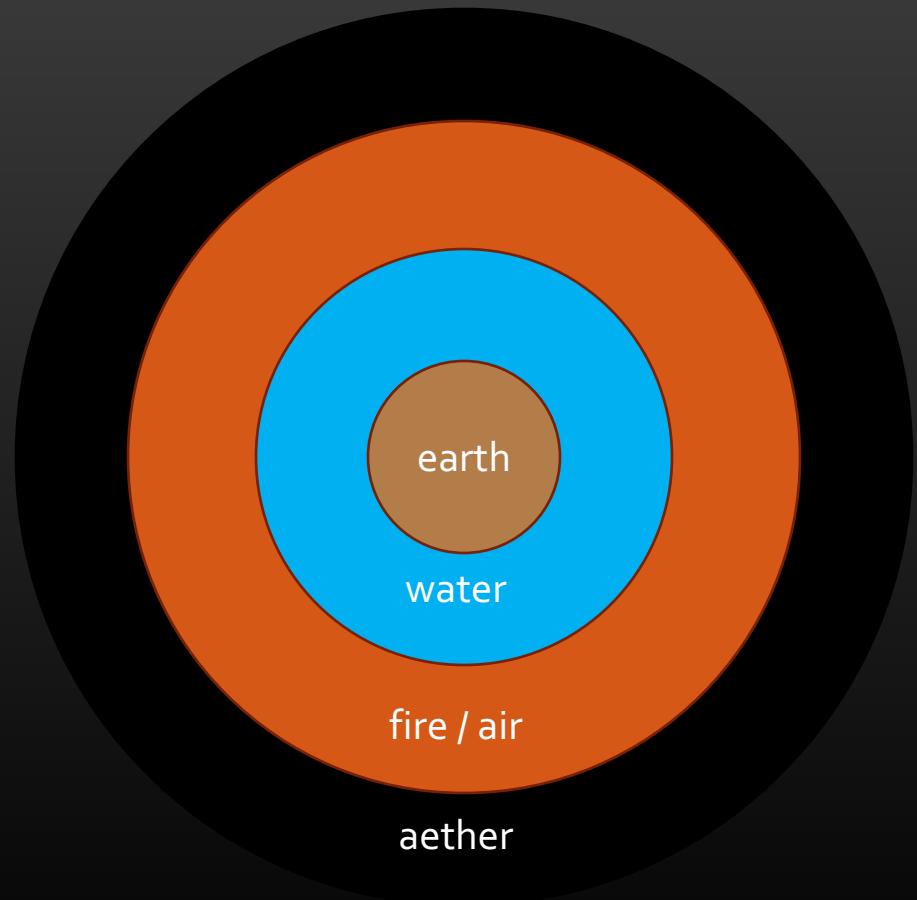
# Aristotelian physics (roughly)

- Objects have
  - A **natural place**
  - A **natural motion**
- Natural state is determined by their **elements**
  - Earth/water: **at rest on land**
  - Fire/air: in the **sky**
  - Aether (celestial objects): **in motion in the heavens**



# Aristotelian physics (roughly)

- An objects can be **disturbed** from its natural state
  - By applying some **impetus** to them
- But **returns** to its natural state when the disturbance ends



# Demo

# Newtonian physics

**One kind of matter** with **one natural state**

- To continue moving in a straight line with **constant direction and speed**
- Disturbances **change the direction** and/or **speed**
- But when the **disturbance ends**, they'll continue with their last direction and speed

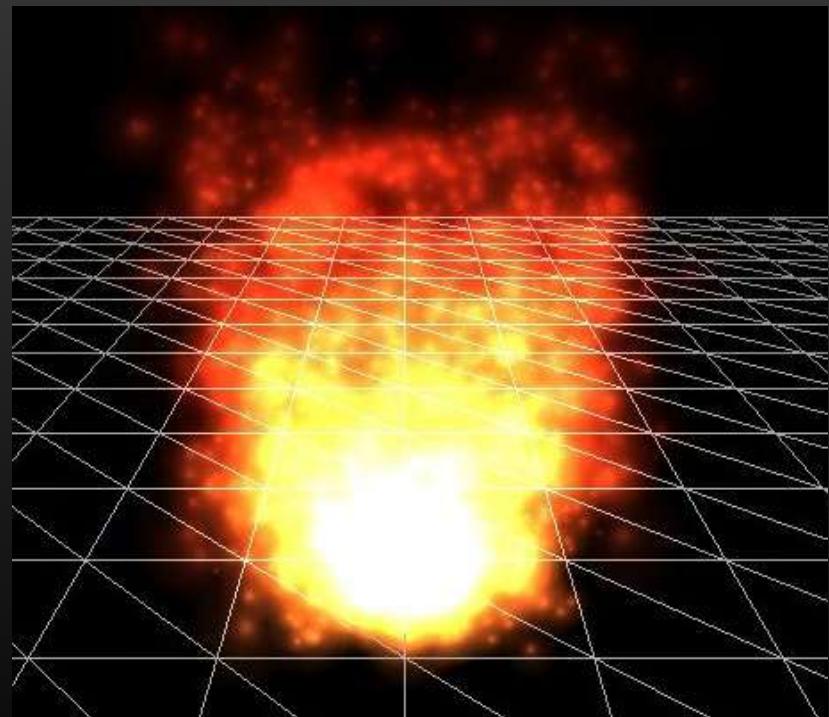
# Aristotelian vs. Newtonian physics

- **Aristotelian** physics
  - Short version: stuff **falls on the ground and stops**
  - **Qualitatively accurate** for everyday life
  - Gives you **no quantitative guidance**
- **Newtonian** physics
  - Summary: stuff moves in **straight lines** unless affected by forces
  - Precise **quantitative model**
  - **Qualitatively inaccurate** for everyday life unless you model gravity, friction, wind resistance, etc.
- So **many games** use something closer to Aristotelian physics (e.g. Mario)

# Newtonian physics of **point particles**

# Particle systems

- We'll start by pretending that objects are **infinitely small points**
  - No rotation or size
  - Just **position**
- We're doing this to **simplify presentation**
- But such systems are **frequently used in games**
  - Simulation of fog, **fire**, explosions, etc.



# Particle motion

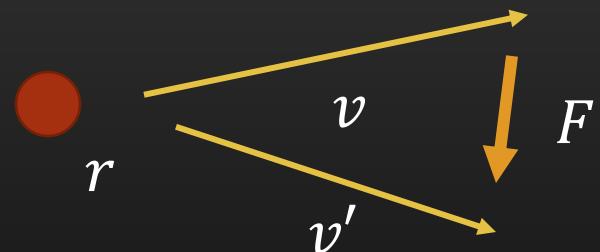
- Newton says things move in a **straight line** at constant speed
- So the **state** of a particle is
  - Its **position**  $r$  (a point)
  - Its **velocity**  $v$  (a vector)
- Its position  $t$  seconds **in the future** will be
$$r + tv$$



# Disturbing the motion

(ignoring mass)

- Anything that can **change** the velocity is called a **force** (a vector)



# Disturbing the motion

(ignoring mass)

- Force is the **rate of change of velocity**

$$\frac{d\mathbf{v}}{dt} = \mathbf{F}$$

- So we can describe particle motions using **integrals**

$$\mathbf{v}(t) = \mathbf{v}(0) + \int_{0_t}^t \mathbf{F}(t) dt$$

$$\mathbf{r}(t) = \mathbf{r}(0) + \int_0^t \mathbf{v}(t) dt$$



# Disturbing the motion (with mass)

- If we **glue three particles together** and subject them to the same force
- Their velocity only changes **a third as much**



# Disturbing the motion (with mass)

- More generally, the effect of a force is **inversely proportional** to the **amount of stuff** its acting on
- The amount of stuff is called the **mass  $m$**



# Disturbing the motion (with mass)

So **now we have**

$$\frac{dv}{dt} = \frac{F}{m}$$

$$v(t) = v(0) + \int_0^t \frac{F(t)}{m} dt$$

$$r(t) = r(0) + \int_0^t v(t) dt$$



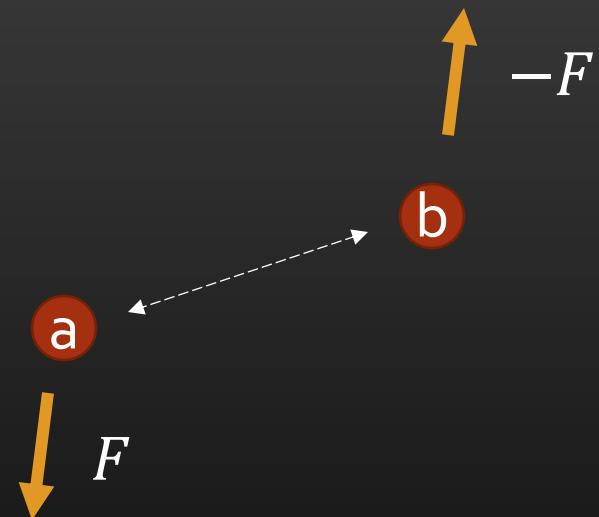
# Demo

# Linear momentum

# Forces come in pairs

So far as we know

- All forces are due to **interactions** between particles
- All interactions are **reciprocal**
  - If **b** applies force  $F$  to **a**
  - Then **a** applies force  $-F$  to **b**
- So in some sense, the “**net force**” is always zero
- How do we think about **what that means?**



# Forces come in pairs

If the two particles have unit mass, then their **average velocity is constant**:

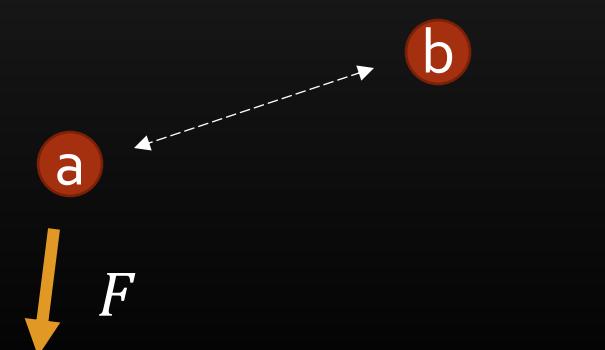
$$\frac{d}{dt}(\nu_a(t) + \nu_b(t)) = \frac{d}{dt}\nu_a(t) + \frac{d}{dt}\nu_b(t)$$

$$= F + (-F)$$

$$= 0$$



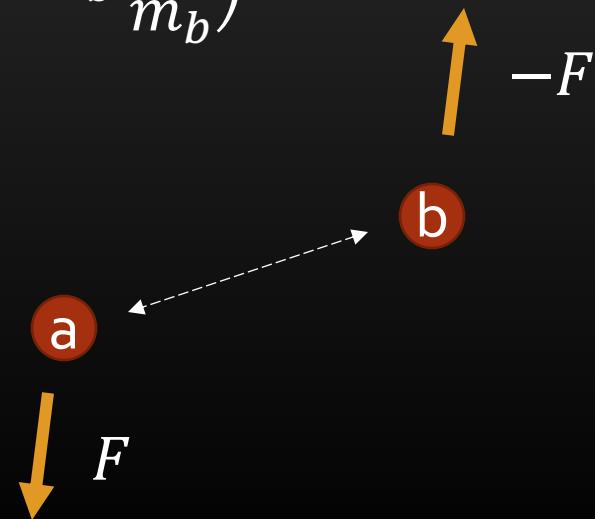
•



# Forces come in pairs

If the particles **don't have unit mass**, then it's more complicated:

$$\begin{aligned}\frac{d}{dt} (m_a v_a(t) + m_b v_b(t)) &= m_a \frac{d}{dt} v_a(t) + m_b \frac{d}{dt} v_b(t) \\&= m_a \frac{F}{m_a} + \left( -m_b \frac{F}{m_b} \right) \\&= F + (-F) \\&= 0\end{aligned}$$



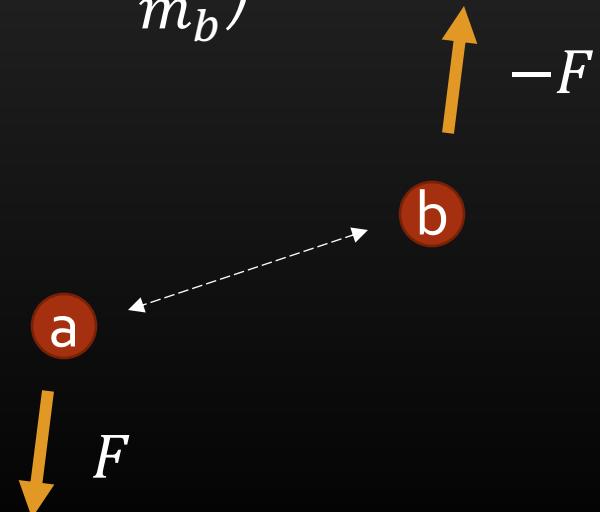
# Forces come in pairs

What's constant isn't the net velocity, but the net velocity  
**weighted by mass**:

$$\frac{d}{dt}(\mathbf{m}_a \mathbf{v}_a(t) + \mathbf{m}_b \mathbf{v}_b(t)) = m_a \frac{d}{dt} v_a(t) + m_b \frac{d}{dt} v_b(t)$$

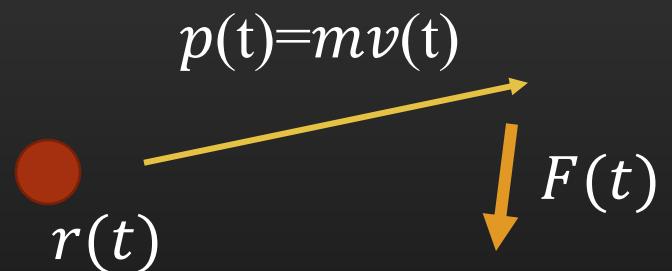
$$= m_a \frac{F}{m_a} + \left( -m_b \frac{F}{m_b} \right)$$

$$= F + (-F)$$
$$= 0$$



# Momentum

- Velocity weighted by mass comes up so often it **has its own name**: momentum
- It's traditionally notated with a  $p$   
 $p(t) = mv(t)$

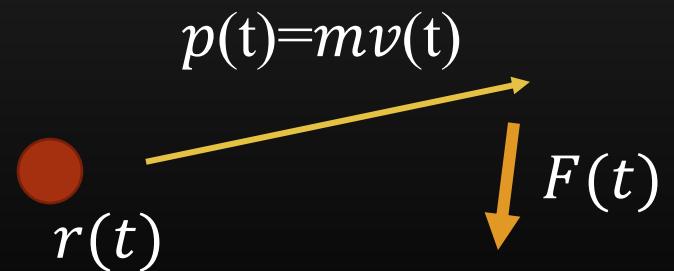


# Momentum

So **now we have** that

$$\frac{dp}{dt} = \frac{d}{dt}(mv) = m \frac{dv}{dt} = m \frac{F}{m} = F$$

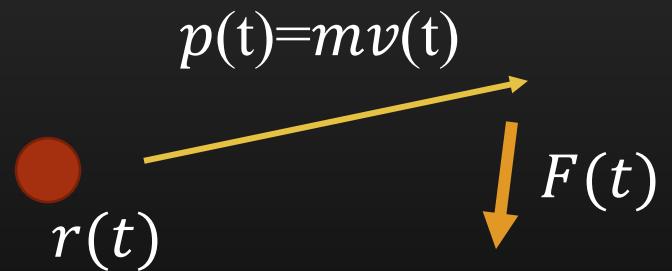
Force is the **time derivative of momentum**



# Momentum

And we can think of the **true state** of a particle as being its **position and momentum**

$$p(t) = p(0) + \int_0^t F(t)dt$$
$$r(t) = r(0) + \int_0^t \frac{p(t)}{m} dt$$

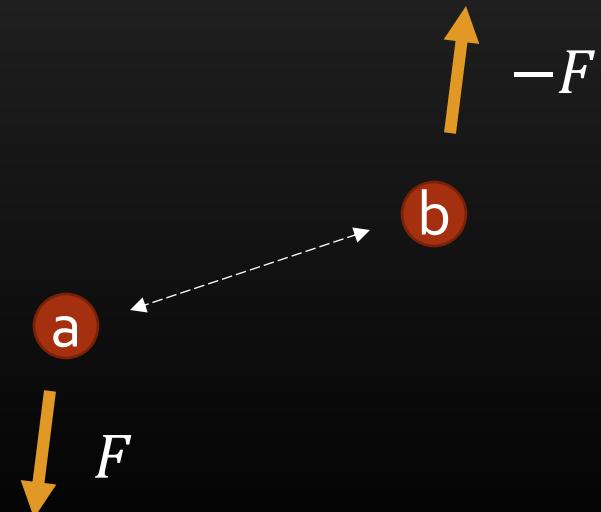


# Forces come in pairs

And so what's really constant isn't net velocity, but **net momentum**:

$$\begin{aligned}\frac{d}{dt}(\mathbf{p}_a(t) + \mathbf{p}_b(t)) &= F + (-F) \\ &= 0\end{aligned}$$

•



# Conservation of momentum

This is referred to as **conservation** of momentum:

The **net momentum** of a system of particles is **constant unless** an **outside force** acts upon them

- Momentum **doesn't change within the system**
- So much as **flow through it**
  - There's the **same net momentum**
  - But it **moves around** the particles

$$\frac{d}{dt} \sum_i p_i(t) = 0$$

# Angular momentum

# Conservation of area

There's a **corollary** of conservation of momentum  
that we might call conservation of **area**

# Conservation of area

- Suppose a particle is undergoing **inertial** motion
  - i.e. **no forces** are applied to it
- It moves in a **straight line** at constant velocity



# Conservation of area

- After **1 second**, it's moved to a new position



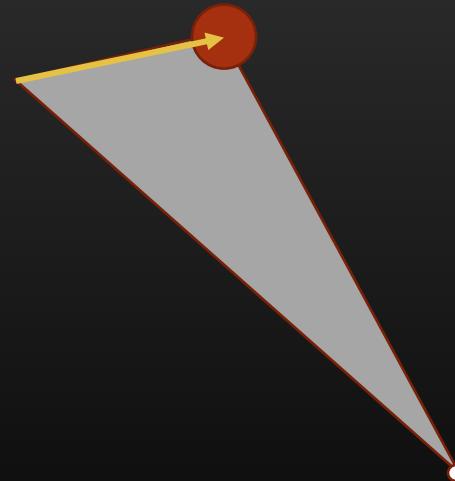
# Conservation of area

- Choose an **arbitrary reference point**



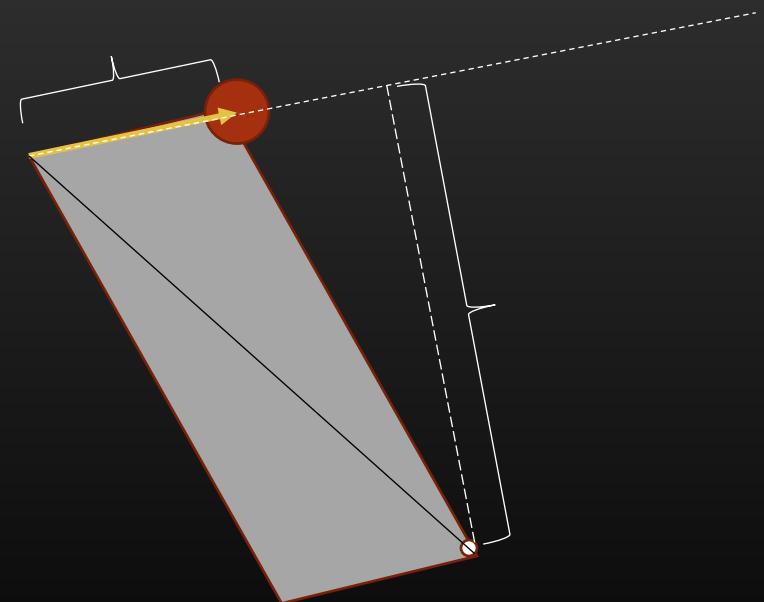
# Conservation of area

- Choose an arbitrary reference point
- And compute the **area** the particle **sweeps** relative to this point



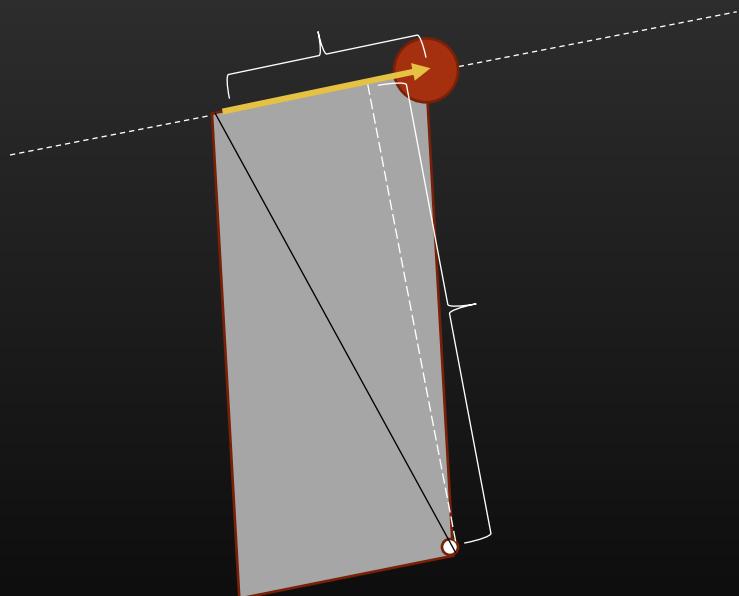
# Conservation of area

- It's half the area of the **parallelogram**
- Which is
  - The **distance traveled**
  - Times the distance from the **reference point to the line**



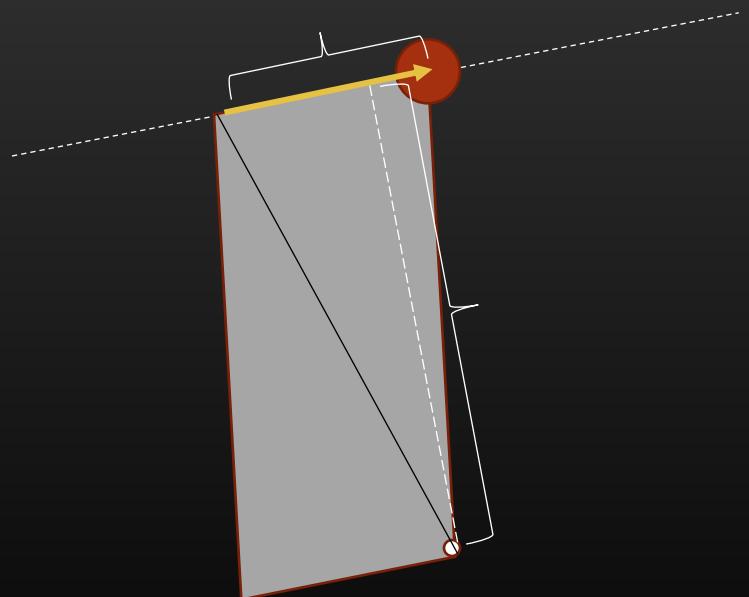
# Conservation of area

- Now suppose it moves for **another second**
- It's a **different triangle**
  - And **different parallelogram**
- But the same
  - **distance traveled**
  - And distance from the **reference point to the line**
- So it's the **same area**



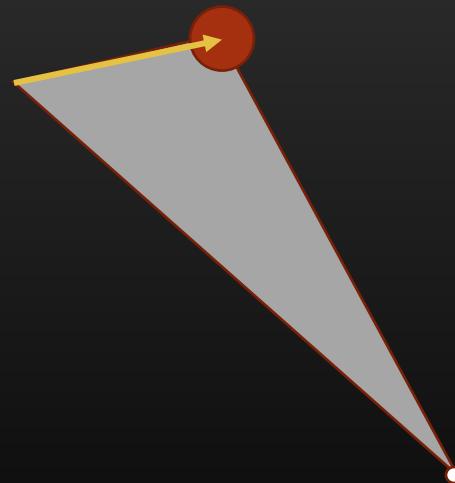
# Conservation of area

- There's **nothing magic** about this
- It's a **consequence** of
  - Objects tending to move at **constant velocity**
  - The **properties of triangles**



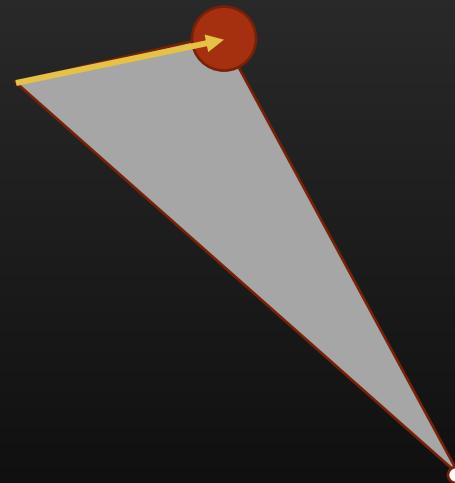
# Conservation of area

- So an **inertial particle**
  - **Sweeps area**
    - Relative to a reference point
  - At a **constant rate**
- This is true for **any arbitrary reference point**



# Conservation of area

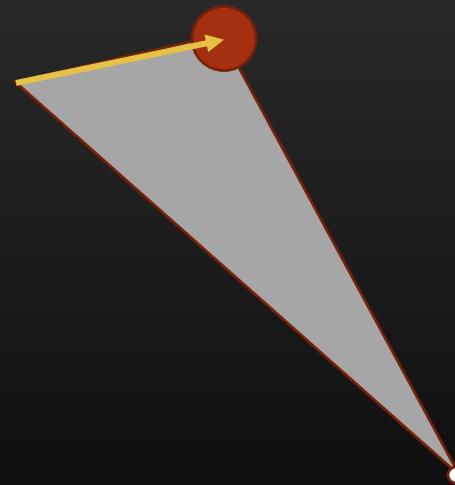
- But it's **not really area** that's conserved
  - Just as it's **not velocity** that's conserved
  - But rather **momentum**
- It's **area times mass**
  - So **mass times the rate of swept area** that's conserved



This is known as ...

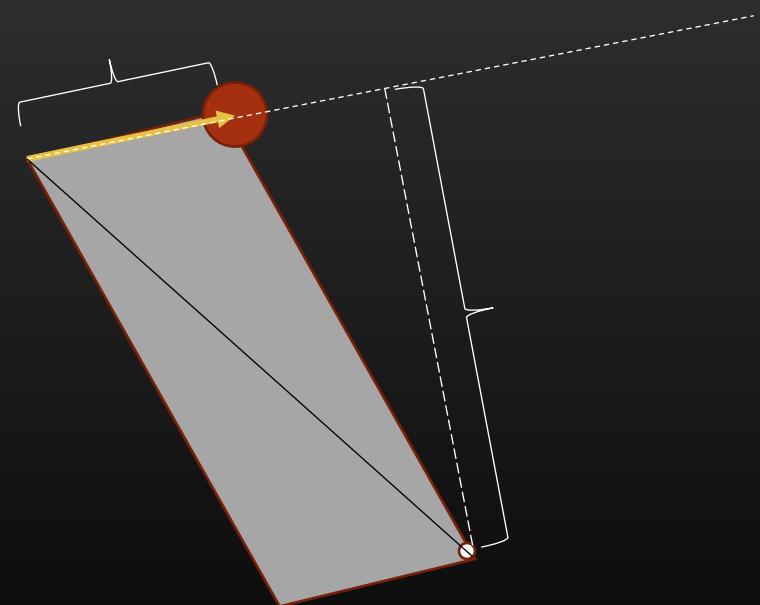
# Conservation of angular momentum

- Angular momentum about some reference point is
  - The rate area is swept about it
  - Times the mass doing the sweeping
- Again, you can choose
  - Any inertial motion
  - Any reference point
- There doesn't have to be rotation per se



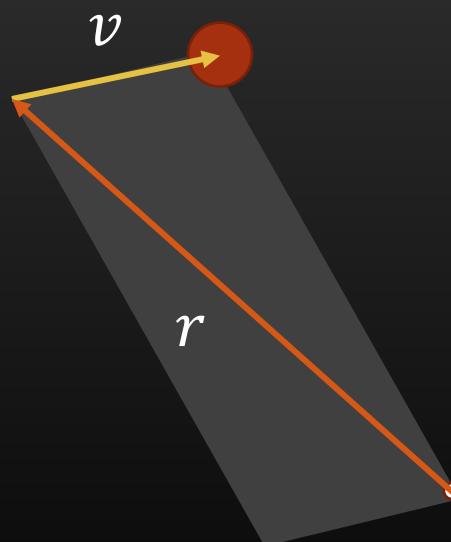
# Computing angular momentum

- The area of the triangle  
is **half the area** of the  
**parallelogram**
- What's an **easy** way to  
compute the **area of**  
**the parallelogram?**



# Computing angular momentum

- **Wedge product!**
- If the object
  - Starts at an **offset  $r$**  from the reference points
  - And moves at a **distance  $v$**
- The area of the **parallelogram** is  $\mathbf{r} \wedge \mathbf{v}$
- And so the **area of the triangle** is
$$\frac{1}{2} m(\mathbf{r} \wedge \mathbf{v}) = \frac{1}{2} (\mathbf{r} \wedge \mathbf{p})$$

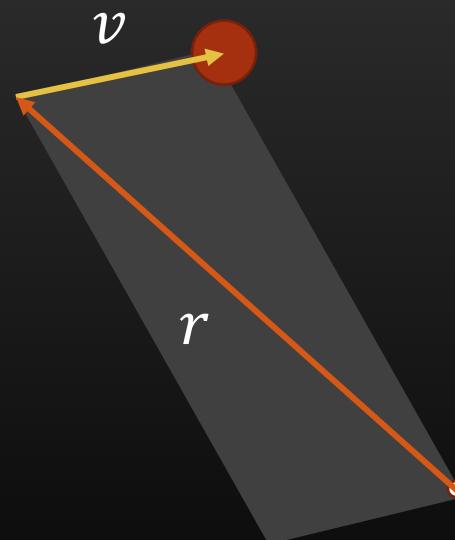


# Computing angular momentum

- We actually define angular momentum in terms of **parallelogram area**

- Saves multiplying by 0.5

- So the **angular momentum** is defined to be  
$$\mathbf{L} = m(\mathbf{r} \wedge \mathbf{v}) = (\mathbf{r} \wedge \mathbf{p})$$



# Conservation of angular momentum

We haven't proven it, but **conservation** of angular momentum holds in the general case:

The **net angular momentum** of a system of particles is **constant unless** an **outside force** acts upon them

- Angular momentum **doesn't change within the system**
- So much as **flow through it**
  - There's the **same net angular momentum**
  - But it **moves around** the particles

$$\frac{d}{dt} \sum_i (r_i(t) \wedge p_i(t)) = 0$$

# Forces

We've described particle motion in terms of forces  
Now all we have to do is describe the forces!

# Gravity

All **masses attract** one another

- In **proportion** to the **masses of both** bodies
- And **inverse proportion** to the **square of the distance** between them

# Gravity

- If two particles  $A$  and  $B$  are separated by a vector  $r = B - A$  ( $B$ 's position minus  $A$ 's)
- The **gravitational force** exerted on  $A$  by  $B$  is  
$$F_{AB} = Gr \frac{m_A m_B}{\|r\|^3}$$
, for some magical constant  $G$ 
  - $G \approx 6.7 \times 10^{-11}$ , if you care
- And the force exerted on  $B$  by  $A$  is  
$$F_{BA} = -F_{AB}$$

**Question:** why are we dividing by  $r^3$  rather than  $r^2$ ?

# Terrestrial gravity

- In games, we generally only care about simulating the attraction of objects **toward the Earth**
  - Or planet of your choosing
- We **ignore**
  - The force of the **object exerts on the Earth**
    - Has negligible effect
  - The **distance** of the object from the Earth
    - Doesn't vary enough to significantly change the force
- And focus on the force the **earth exerts** on the object
  - Which is **effectively a constant**
  - One that you usually just **tune** in the editor until it **looks good**

# Demo

# Throwing a ball

- Let's **ignore wind resistance** (drag) for a moment
- Suppose I **throw a ball** up in the air
  - It starts at rest
  - When it **leaves my hand**, it's at
    - Some height  $h$
    - And some **velocity  $v$**
  - As it rises, **gravity decelerates** it until it comes to rest
  - Then **gravity accelerates** it downward
  - When it **returns** to height  $h$ 
    - Its **velocity is now  $-v$**

# Kinetic energy

- Notice we get the same **magnitude of velocity**
  - But a different direction
- In addition to momentum, we can define a quantity for a particle called **kinetic energy**
  - A **scalar**, not a vector
  - Proportional to **mass** times **square of velocity**
  - Or technically  $\frac{1}{2}mv^2$

# Conservation of energy

- Our ball **starts and ends** with the **same kinetic energy**
- In between, it **disappears**
- And if we let it fall farther, it would **increase**
- But if it comes back to the **same state** (height), it will always be the same
- All **known physical forces** have this property

# Potential energy

- The standard explanation of this is that kinetic energy is **converted** into another kind of energy, **gravitational potential energy**
- The **total energy** (kinetic + potential) is **conserved**
- So like momentum, **energy isn't created or destroyed** in a closed system
  - But rather **flows**
  - Between **particles**
  - And between **forms** of energy

# Energy is not conserved in games

- Working on games makes you understand how **lucky** we are to have conservation of energy
- The **alternatives** are
  - Everything **slows down and stops**
  - Everything **explodes**

*Okay, those aren't the only possibilities, but they're the ones that happen in games*
- Alas, games only **approximate** the laws of physics
  - Numerical **error accumulates**
  - So they always either **leak energy**
  - Or worse, **create it**

# Energy is not conserved in games

- In practice, games almost always err on the side of **leaking energy**
  - Much closer to **Aristotelian mechanics** and everyday life
- But there are times when they can **pump energy**
  - **Boom**
  - We shall see **examples** of this shortly
- Or they can try to **slow down** and compute results more accurately
  - But that can lead to diminishing returns
  - Some physics programmers call this the **well of despair** (seriously)

# Electrostatic force

# Electrostatic force

Like gravity, except

- Mind bogglingly **stronger** (gravity is pathetic)
- Determined by **charge** rather than mass
- Charge is **signed** (can be positive or negative)

# Electrostatic force

- If two particles  $A$  and  $B$  are separated by a vector  $r = B - A$  ( $B$ 's position minus  $A$ 's)
- The **electrostatic force** exerted on  $A$  by  $B$  is

$$F_{AB} = -k_e r \frac{q_A q_B}{\|r\|^3}, \text{ for some magical constant } k_e$$

- $k_e \cong 9 \times 10^9$ , if you care
- So it's approximately **10<sup>20</sup> times stronger than gravity**

- And the force exerted on  $B$  by  $A$  is

$$F_{BA} = -F_{AB}$$

# Demo

# Aside: why was that **unstable**?

- Electrostatic forces are **non-linear**
  - Increase rapidly as distance decreases
- When particles are **far apart**
  - They exert **small forces**
- But the system **updates** the positions in **discrete steps**
  - Effectively assumes **no forces along the way**

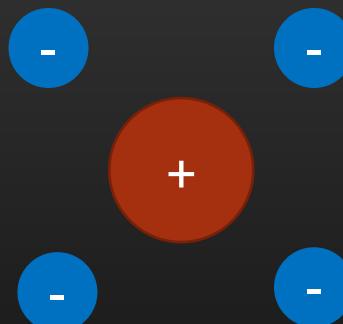
# Aside: why was that **unstable**?

- If a particle is already moving **very fast**
  - Position can **change a lot** from frame to frame
  - It can go from being safely **far from a particle to right on top** of it
    - **Without simulating forces along the way**
  - Now it experiences an **infinite force!**
- In this simulation, some particle will **move fast sooner or later**
- Game physics engines are generally **not written to handle situations like that**

# Collision dynamics

# Atoms

- Negatively charged **electrons**
- And positively charged **nuclei**
  - **Protons** optionally grouped with **neutrons**
- Attract one another to form **atoms**
  - Where the **total charges is zero**
- **Quantum magic** keeps them from touching
  - **Heisenberg effects**, in particular



# Demo

The actual **physics of this demo are totally lame**, since it's only modeling the electrostatic force, and preventing the electrons from collapsing on the nucleus by holding them in place using sticks (in real life, it's due to Heisenberg effects)

# Electrostatic interactions with an atom

- So when a particle (e.g. an electron) interacts with an atom, it feels **forces from both**
  - The **electrons** (repulsion)
  - The **nucleus** (attraction)
- Both these are **weighted by the inverse square** of the distance from the electron or nucleus

$$F = k_e q_e \sum_{p \in Atom} \frac{r_p q_p}{\|r_p\|^3}$$

# Electrostatic interactions between atoms

- Suppose the atom has a radius of 1
  - And its nucleus is a **distance  $d$**  away
  - Then the **nearest electron** has a distance of  $d - 1$
  - And the farthest is  $d + 1$
- Then
  - The **attraction** to the nucleus will be proportional to  $1/d^2$
  - The **repulsion** from the electrons will be no more than  $1/(d - 1)^2$
  - The **net force** is proportional to
$$F \propto \frac{1}{(d - 1)^2} - \frac{1}{d^2}$$
- This is
  - **Tiny** when  $d$  is **large**
  - **Huge** when  $d$  is **small**

# Collision without touching

- When two atoms are **far apart**, they **don't interact**
- As they **draw near**
  - They exert **repulsive forces** on one another
  - Pushing them **apart**
  - Until they're far enough **stop interacting**
- Momentum and energy are **conserved** during the whole interaction
  - Which means we can
    - **Compute** their **ending velocities**
    - **Directly from** their **starting velocities**
  - Without needing to compute the intermediate repulsive forces

# Example: collision dynamics

Let's assume two particles collide in a **one-dimensional world**

- With **masses  $m_1$  and  $m_2$**
- And **velocities  $u_1$  and  $u_2$  before the collision**
- And **velocities  $v_1$  and  $v_2$  after the collision**

Assuming there are no external forces acting on the particles,

- The **conservation** laws give us **two simultaneous equations**
- And we have **two unknowns** ( $v_1$  and  $v_2$ )
- So we can solve for the unknowns

**Conservation of momentum:**

$$m_1 u_1 + m_2 u_2 = m_1 v_1 + m_2 v_2$$

**Conservation of energy:**

$$m_1 u_1^2 + m_2 u_2^2 = m_1 v_1^2 + m_2 v_2^2$$

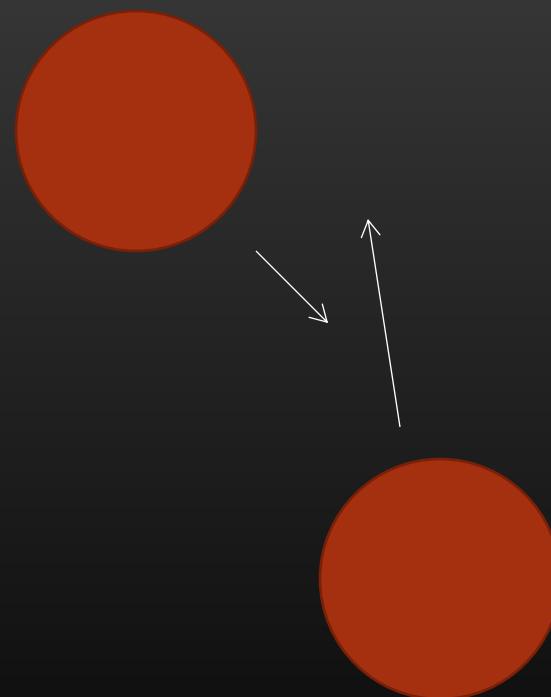
So through **the miracle of algebra**:

$$v_1 = \frac{u_1(m_1 - m_2) + 2m_2 u_2}{m_1 + m_2}$$

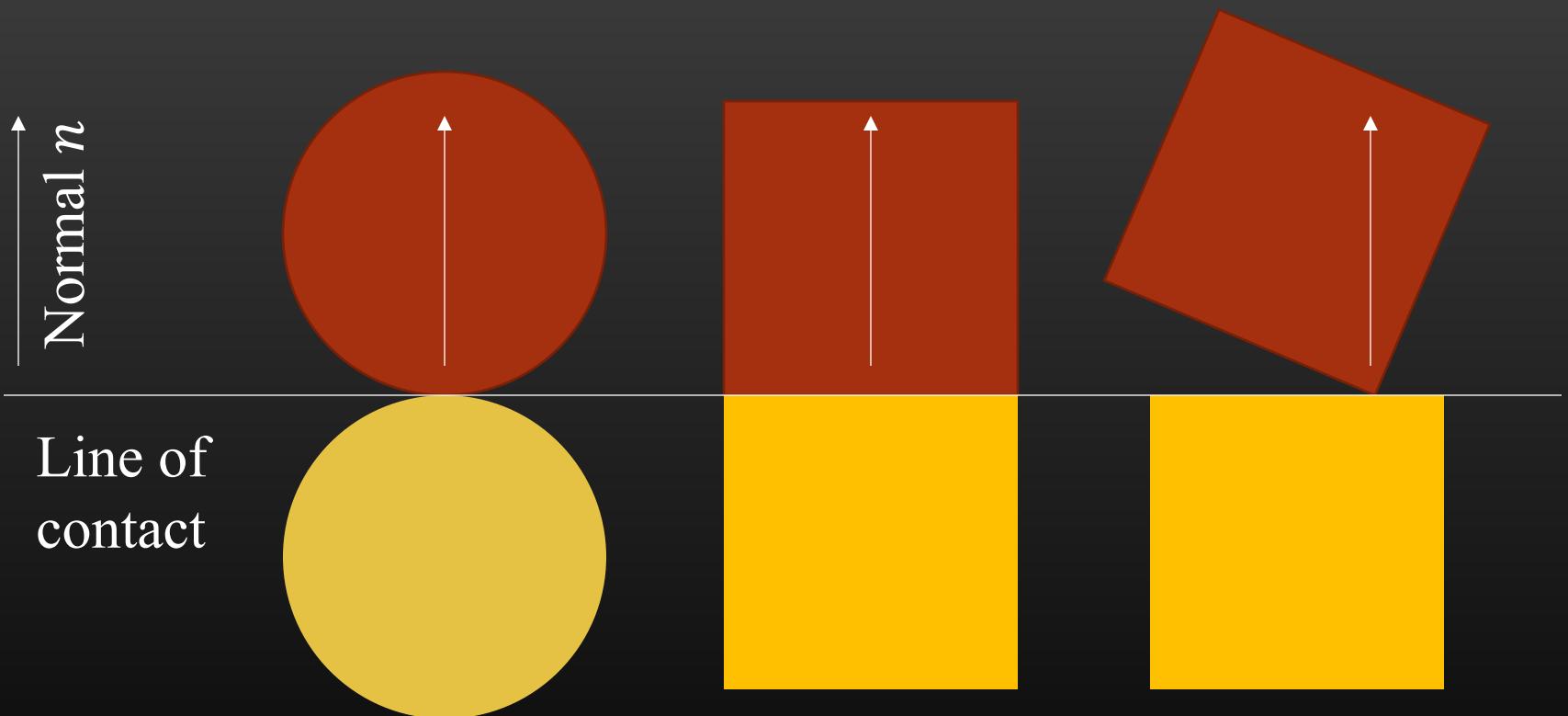
$v_2$  = same, with subscripts reversed

# Elastic collision of billiard balls

- We can model billiard balls as **big point particles** that
  - Mass and position, but not orientation
- Actual collision involves **forces exerted between atoms** over time
- We can **fudge it** by using **conservation laws**



# Contact geometry



# Elastic collision of billiard balls

When the balls hit, they'll have **two components** to their velocities

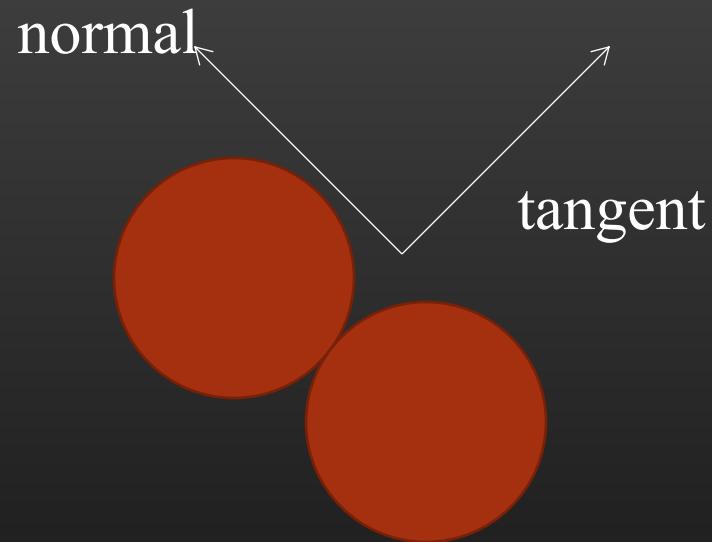
- **Tangential**

- This is the balls sliding along the line of contact
- This part of the velocity doesn't change

- **Normal**

- This is the balls compressing and separating from one another
- This component changes

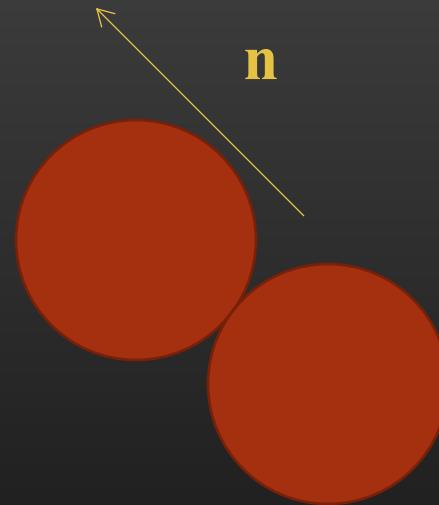
- So we can **reduce** it to a **1D collision**



# Elastic collision of billiard balls

Let  $\mathbf{n}$  be the **unit normal** in the direction of the balls collision:

$$\mathbf{n} = \frac{\mathbf{x}_1 - \mathbf{x}_2}{\|\mathbf{x}_1 - \mathbf{x}_2\|}$$



Then the **normal velocities** of the balls at the time of collision are:

$$u_1 = \mathbf{n} \cdot \mathbf{v}_1$$

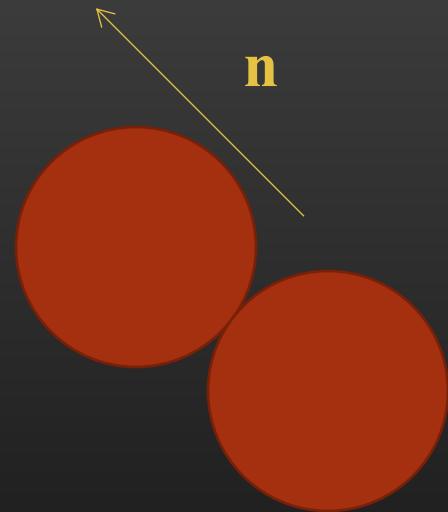
$$u_2 = \mathbf{n} \cdot \mathbf{v}_2$$

# Elastic collision of billiard balls

Using our handy-dandy formula  
for the **post-collision velocities**  
in 1D:

$$v_1 = \frac{u_1(m_1 - m_2) + 2m_2 u_2}{m_1 + m_2}$$

$v_2$  = same, with subscripts reversed



We can compute the new **2D velocities** by subtracting off the old normal component and adding in the new one:

$$\mathbf{v}'_1 = \mathbf{v}_1 + (v_1 - u_1)\mathbf{n}$$

$$\mathbf{v}'_2 = \mathbf{v}_2 + (v_2 - u_2)\mathbf{n}$$

# Demo

# Other forces

# Spring forces

- Like elastic collisions, spring forces are **ultimately due to atomic interactions**
- Springs follow the rule:

$$\mathbf{F} = -k\mathbf{x}$$

$$\frac{dv}{dt} = \frac{-kx}{m}$$

where

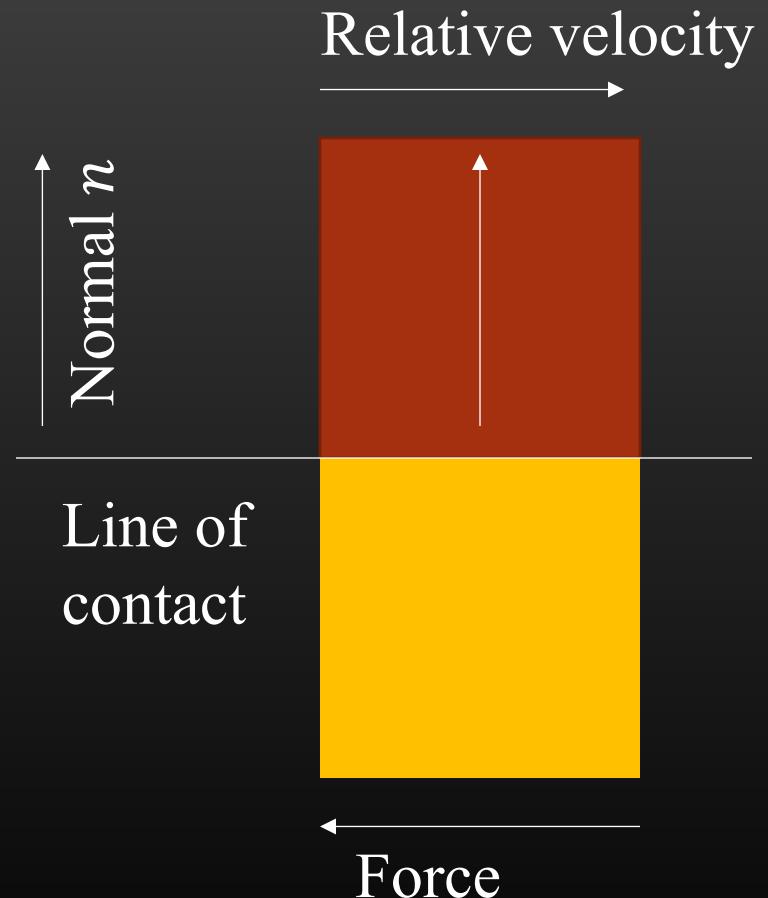
- $x$  is the **displacement** of the spring from its preferred position
- $k$  is the **stiffness** of the spring
- $m$  is the **mass** of the particle being moved by the spring

# Dissipative forces

# Friction

Friction forces are always

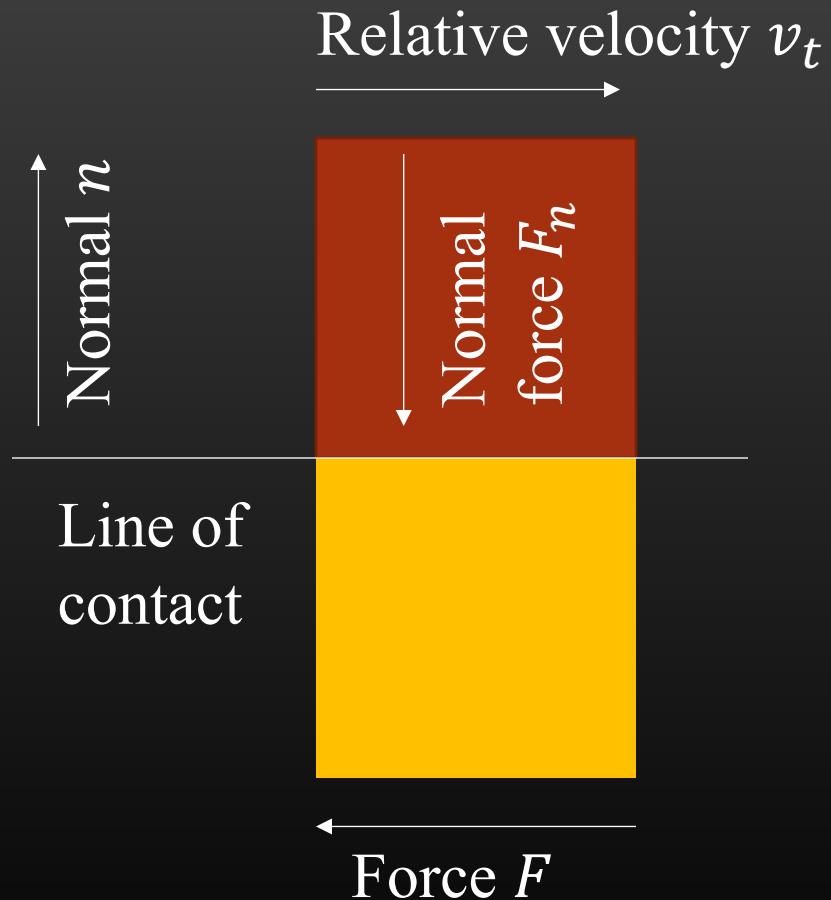
- Along the **line of contact**
- Opposite to the relative motion
- Proportional to the **force along the normal**  
(how hard the surfaces are **pressed together**)



# Friction

$$F = -\mu \|F_n\| \frac{v_t}{\|v_t\|}$$

- $\mu$  is the **coefficient of friction**, and depends on the **materials** in use
- It's more **complicated** than this in practice
  - Both in the physics
  - And in the implementation
    - Need to make sure friction does cause **reversals** in direction
- But this is the basic idea

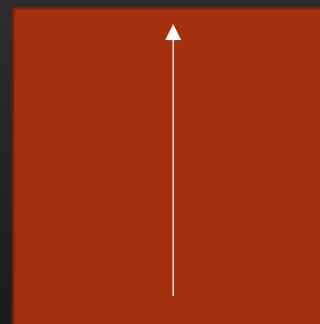


# Fluid forces

Two cases

- Viscous damping
  - Viscous medium
  - Low velocity
  - Linear in velocity
  - $F \cong -cv$
- Drag
  - Non-viscous medium
  - High velocity
  - Quadratic in velocity
  - $F \cong -c\|v\|v$

Relative velocity  $v$



# Dissipative forces in games

Dissipative forces are **used a lot in games**

- More **realistic**
- Needed for numerical **stability**
  - Which brings up the issue of ...

# Numerical simulation

# Physics simulation

To simulate a physics system, we

- **Discretize** time into a series of finite time **steps**
- **Numerically integrate** the dynamics equations to solve for the new position at each step
- Resolve any **constraints** among the bodies
  - **Collision resolution**  
Two objects can't occupy the same space at the same time
  - **Joints**  
Doors have to stay attached to their hinges

# Problems with physics solvers

Physics simulators (“solvers”) have to make **approximations**

- **Discrete** time
  - Things happen in steps
  - Assumes things are constant between steps
- **Numerical** integration
  - Assumes simplified forms for functions (e.g. truncated Taylor series)
- **Floating-point**
  - Numbers only approximated

These lead to two important failure modes

- **Inaccuracy**

- The solver gives us a slightly different answer than the real world
- We mostly don’t care about that for game applications

- **Instability**

- Velocities and positions quickly go to infinity

# Euler integration

- Simplest possible integrator
  - Replace integrals with **sums**
- **Easy** and obvious to implement
- Unfortunately, **error** is **quadratic**
- Basic **loop**:

For each particle

Recompute net force F

$$a = F/m$$

$$v = v + a \Delta t$$

$$x = x + v \Delta t$$

$$\int dt \rightarrow \sum \Delta t$$

$$\mathbf{x}_t = \mathbf{x}_{t-1} + \mathbf{v}_t \Delta t + O(\Delta t^2)$$

$$= \mathbf{x}_0 + \sum_{i=0}^t \mathbf{v}_i \Delta t + O(\Delta t^2)$$

$$\mathbf{v}_t = \mathbf{v}_{t-1} + \mathbf{a}_t \Delta t + O(\Delta t^2)$$

$$= \mathbf{v}_0 + \sum_{i=0}^t \mathbf{a}_i \Delta t + O(\Delta t^2)$$

# Instability of the Euler integrator

Consider simulating the following **1D motion**:

$$\frac{dx}{dt} = -kx$$

Question: what happens for **large values of k**?

Here's the **Euler integrated** version:

$$\begin{aligned}x_t &= x_{t-1} - kx_{t-1}\Delta t \\&= x_{t-1}(1 - k\Delta t)\end{aligned}$$

# Instability of the Euler integrator

- Consider simulating **the following equation:**

$$\frac{dx}{dt} = -kx$$

- This **converges to 0**

Here's the **Euler integrated** version:

$$\begin{aligned}x_t &= x_{t-1} - kx_{t-1}\Delta t \\&= x_{t-1}(1 - k\Delta t)\end{aligned}$$

Question: what happens for **large values of k**?

If  $k$  is **small**, then  
 $|1 - k\Delta t| < 1$

So

$$|x_t| < |x_{t-1}|$$

So  $x$  **converges** to 0, like we'd expect

# Instability of the Euler integrator

- Consider simulating **the following equation:**

$$\frac{dx}{dt} = -kx$$

- This **converges to 0**

Here's the **Euler integrated** version:

$$\begin{aligned}x_t &= x_{t-1} - kx_{t-1}\Delta t \\&= x_{t-1}(1 - k\Delta t)\end{aligned}$$

Question: what happens for **large values of  $k$** ?

But if  $k > 2/\Delta t$ , then  
 $1 - k\Delta t < -1$

So

$$|x_t| > |x_{t-1}|$$

So

$$\lim_{t \rightarrow \infty} |x_t| = \infty$$

**Boom**

**Instability**  
is the bane of the physics  
programmer's existence

# Mass/spring system w/Euler integrator

**Springs** follow the rule:

$$\mathbf{F} = -k\mathbf{x}$$

$$\mathbf{a} = \frac{-k\mathbf{x}}{m}$$

where

- $x$  is the **displacement** of the spring from its preferred position
- $k$  is the **stiffness** of the spring
- $m$  is the **mass** of the particle being moved by the spring

**Euler** integrator:

$$\mathbf{x}_t = \mathbf{x}_{t-1} + \mathbf{v}_t \Delta t$$

$$\mathbf{v}_t = \mathbf{v}_{t-1} + \frac{-k\mathbf{x}_{t-1}}{m}$$

Pseudocode:

do forever:

$$\mathbf{v} = \mathbf{v} - (k\mathbf{x}/m) \Delta t$$

$$\mathbf{x} = \mathbf{x} + \mathbf{v} \Delta t$$

# Verlet integration

# Coping with instability

- Note that the mass/spring system looks a lot like the equation that made the Euler integrator **blow up**
- So not surprisingly, it also blows up if **k is too big**
  - It's too big when  $k > 1/\Delta t$
- How do we deal with this?
- Use a **smaller k**
  - Might be fine
  - Or might make things look wobbly
- Use a **smaller time-step**
  - Works, but requires more CPU time
- Use a **different integrator**
  - E.g. Runge-Kutta
  - Better, but still doesn't make the problem go away entirely

# Verlet integration

- Basic idea: compute position in next frame from **position in current and previous frames**:

$$\begin{aligned}\mathbf{p}(t + \Delta t) &\cong \mathbf{p}(t) + (\mathbf{v}(t) + \mathbf{a}(t)\Delta t)\Delta t \\ &\cong \mathbf{p}(t) + \frac{\mathbf{p}(t) - \mathbf{p}(t - \Delta t)}{\Delta t} \Delta t + \mathbf{a}(t)\Delta t\Delta t \\ &= \mathbf{p}(t) + (\mathbf{p}(t) - \mathbf{p}(t - \Delta t)) + \mathbf{a}(t)\Delta t^2 \\ &= 2\mathbf{p}(t) - \mathbf{p}(t - \Delta t) + \mathbf{a}(t)\Delta t^2\end{aligned}$$

- Fancier version with viscous **damping**:

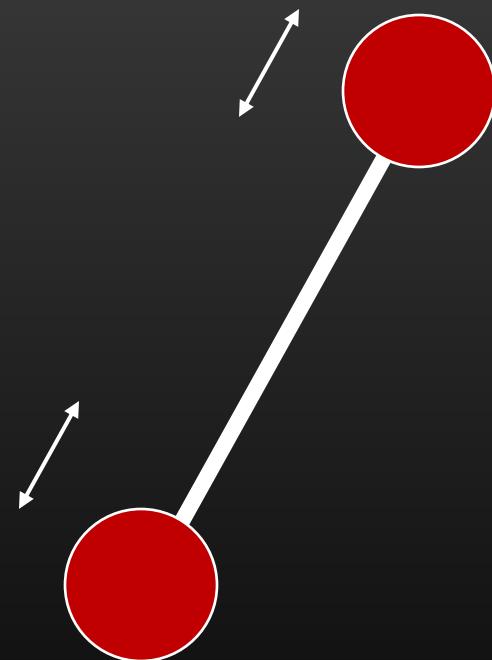
$$\begin{aligned}\mathbf{p}(t + \Delta t) &= (2 - d)\mathbf{p}(t) - (1 - d)\mathbf{p}(t - \Delta t) \\ &\quad + \mathbf{a}(t)\Delta t^2\end{aligned}$$

- Fast, easy to implement, and **stable**: error is  $O(\Delta t^4)$

# Constraint satisfaction

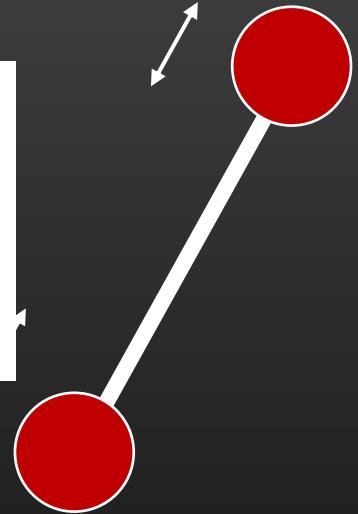
# constraint satisfaction

- Verlet integration is also good for **constraint satisfaction**
  - Just find the **closest position** that satisfies the constraint
  - And **move** it there
- **Distance** constraints
  - View the distance constraint as a **spring**
  - Solve for the **equilibrium position**
  - Move both particles into those positions



# enforcing a distance constraint

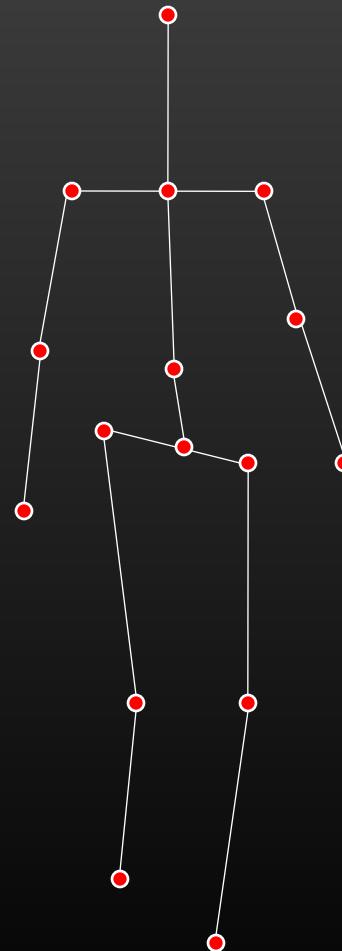
$$\mathbf{p}_i(t) = \mathbf{p}_i(t) - m_i^{-1} \frac{\|\mathbf{o}\| - d}{2\|\mathbf{o}\|(m_i^{-1} + m_j^{-1})} \mathbf{o}$$
$$\mathbf{p}_j(t) = \mathbf{p}_j(t) + m_j^{-1} \frac{\|\mathbf{o}\| - d}{2\|\mathbf{o}\|(m_i^{-1} + m_j^{-1})} \mathbf{o}$$



- Compute **difference** between **desired distance**  $d$  and length of offset  $\mathbf{o}$
- **Move particles that distance** along  $\mathbf{o}$
- Weight relative motions by particle masses

# Ragdoll physics in Hitman (2001)

- People modeled as **particles** (at joints) + **massless rods** (bones)
  - Effectively molecules
- **Verlet** integration
- Good enough to nicely **simulate dead bodies!**



# Extended bodies

Objects that **aren't points**

# Using lots of particles

- Ultimately real extended bodies are just **atoms**
- We can **simulate** an extended object by
  - Using a lot of **particles**
  - Connecting them using **springs** to hold them together
  - Best to use **Verlet** as the integrator
- Used for
  - **Soft body** physics (jelly, goo, etc.)
  - **Cloth** simulation

# Demo

# Soft body physics



# Particles are problematic for rigid bodies

- We could simulate **rigid bodies** with particles
  - Just make the springs **stiffer**
- However, that has **issues**
  - **Expensive**
  - Need **really stiff** springs
  - And that means **boom**

# Demo

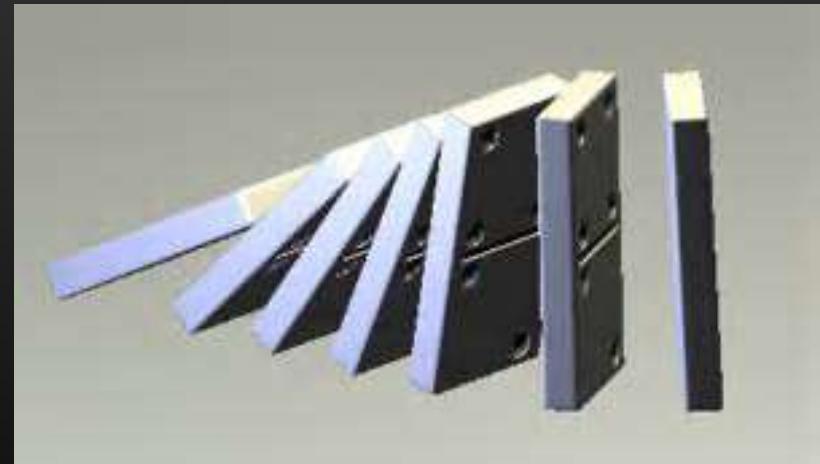
# Rigid body dynamics

- Rigid bodies add **rotational degrees of freedom** to the object's state
- The angular version of mass is a matrix called the **moment of inertia**
- So we need to track
  - **Linear** position, mass, and momentum
  - **Angular** position, mass, and momentum

# Rigid bodies

We can make things more **efficient** and **stable** by

- Representing the body as **one object**
- Now **state** is more complicated
  - **Position**
  - Velocity / **momentum**
  - **Rotation**
  - Rotational velocity / **angular momentum**



# Motion of the center of mass

- We pretend the rigid body is a **big particle**
- Whose position is the **center of mass**

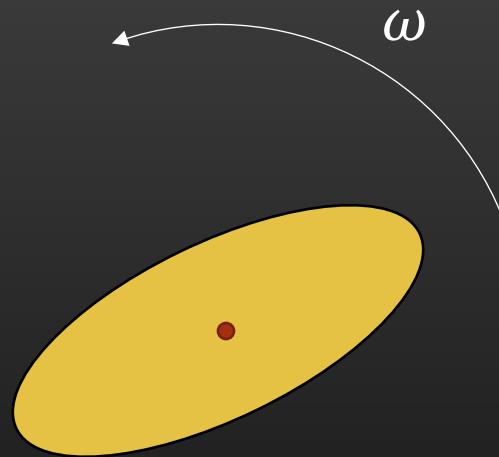
$$\frac{1}{m} \int \rho(r) dr$$

- Which moves based on the **net momentum** of all the mass in the object



# Rotation

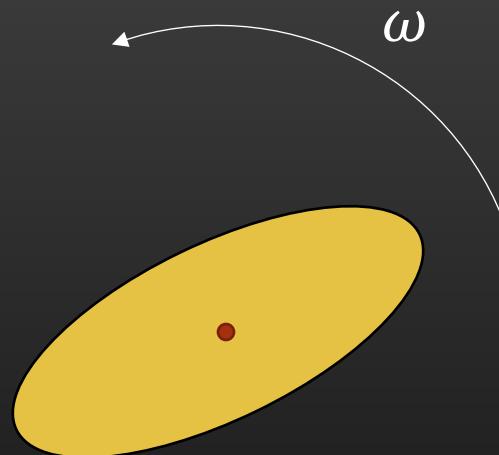
- The other **degree of freedom** of the object's pose is its
  - **Rotation**
  - **About the center** of mass
- Which changes based on its **angular velocity**



# Angular momentum

As with positional velocity,

- It's sometimes better to think of angular velocity in terms of **angular momentum**
- Specifically the net angular momentum **about the center of mass**



# Angular momentum

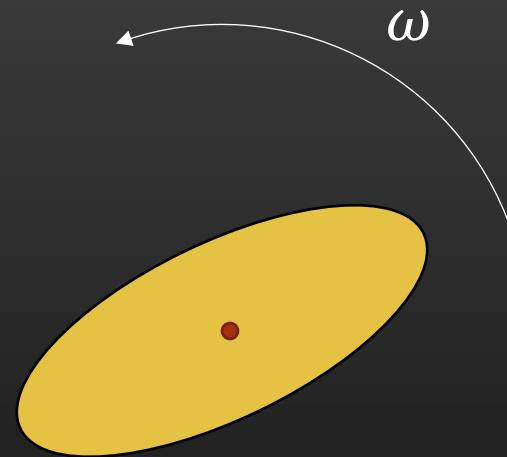
- The **net angular momentum** is just the sum of the momenta of the atoms

$$L = \int r^2 \rho(r) \omega dr$$

- We typically **factor** this into  **$\omega$** , which changes, and  **$I$**  which is fixed by the object

$$L = I\omega,$$

$$I = \int r^2 \rho(r) dr$$

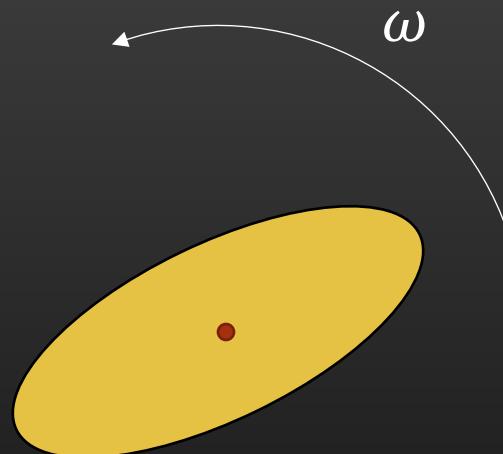


# Angular momentum

$$L = I\omega,$$

$$I = \int r^2 \rho(r) dr$$

- $I$  is known as the **moment of inertia**, as is the rotational equivalent of **mass**



# Torque

- Torque is the **rotational equivalent of force**

$$\tau = \frac{dL}{dt} = \frac{d(I\omega)}{dt} = I \frac{d\omega}{dt}$$
$$\frac{d\omega}{dt} = \tau/I$$

# Applying a force to a rigid body

- Effect depends on **where** its applied and **what direction** it is
  - Change **linear** momentum
  - Change **angular** momentum
- **Common cases** are
  - Apply **force to center** of mass
    - Changes just **linear** momentum
  - Apply **torque about center** of mass
    - Changes just **angular** momentum
  - **Collision** at a contact point
    - Changes both, based on the contact point and momentum of the colliding object

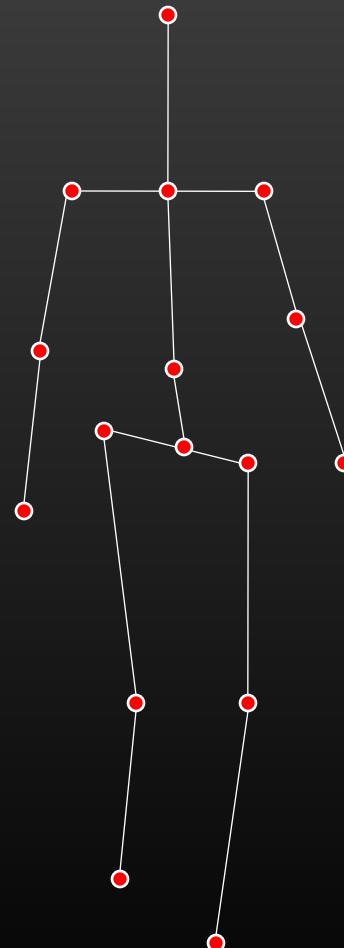
# Articulated bodies

# Constraints in real life

- In real life, **objects aren't rigid**
- They're springy, they're just **very stiff strings**
  - Elastic collision of objects really consists of the **compression and release** of both objects
  - Just so **fast** we can't see it
- Most constraints in real life **aren't really constraints** at all
  - They're **elastic forces** between interacting objects
- **Door hinges** work because the different parts of the hinge are pressing on one another

# Ragdoll physics in Hitman (2001)

- People modeled as **particles** (at joints) + **massless rods** (bones)
  - Effectively molecules
- **Verlet** integration
- Good enough to nicely **simulate dead bodies!**



# Penalty methods in constraint simulation

- So most constraints can be simulated by putting **springs between objects**
- Problem:
  - If the spring is stiff, the system is **unstable**
  - If the spring isn't stiff, the constraint is only **weakly satisfied**
- Note: real physics is also unstable this way, it's just has a **much smaller  $\Delta t$**

Example: **stacking crates**

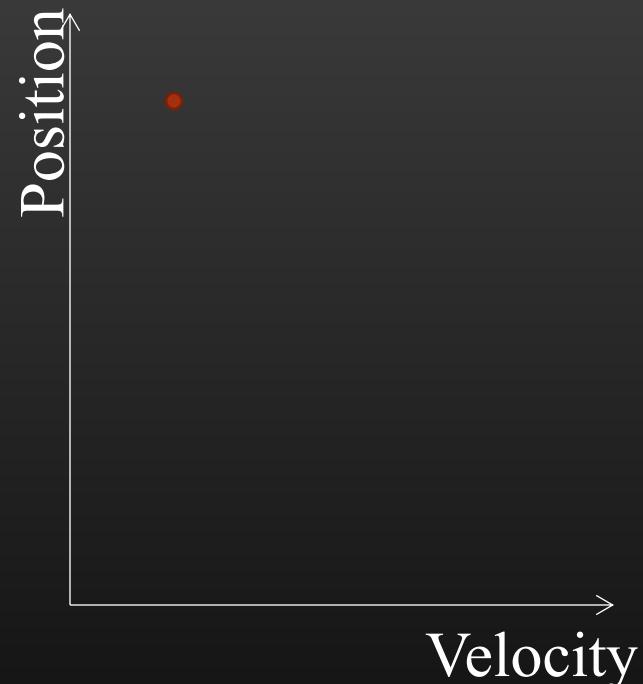
- Put a spring between them that **pushes them apart**

“Physics can simulate anything, so long as it’s **jello**”

*Chris Hecker*

# State space (or “phase space”)

- Each particle or body has a **state**
  - Position
  - Velocity
  - Orientation & angular velocity (for non-point rigid bodies)
- The states of all particles collectively form the state of the **complete system**
- The space of possible states for the system is its **state space** or **phase space**



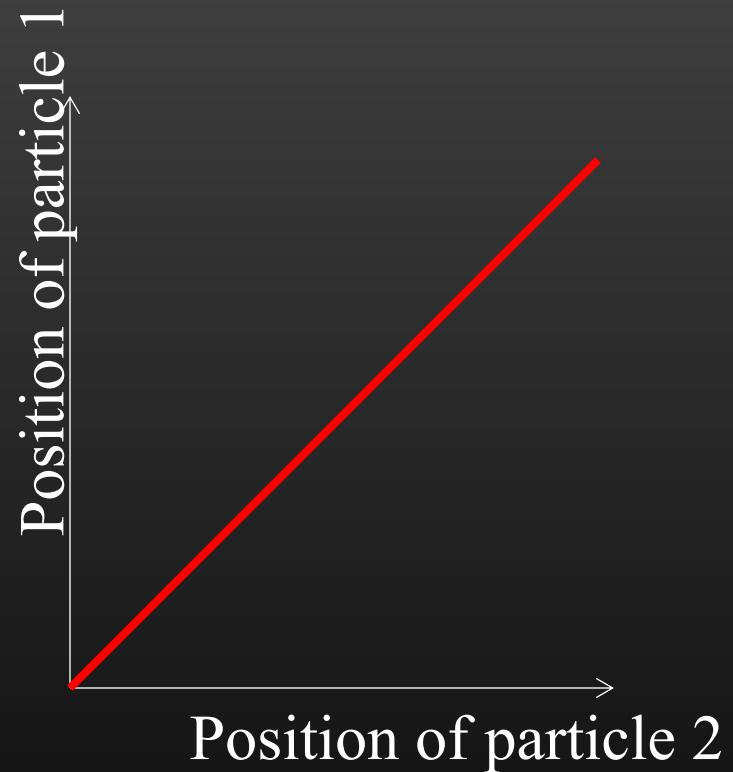
Phase space for a single particle

# Constraint resolution

**Constraints** place restrictions on the possible states the system can be in

- **Collision** constraints
  - Objects can't interpenetrate
- **Joints**
  - Objects have to be connected at one end

After integrating the state, the simulator needs to **resolve** the constraints to make sure the system is in a valid state

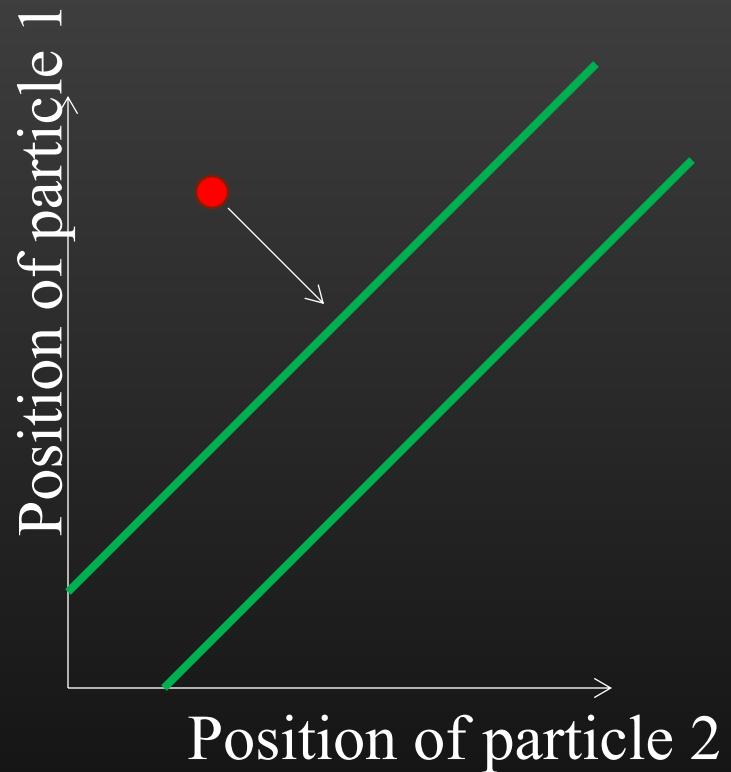


**Constraint manifold** for collision constraint on two 1D particles

System can be in any state not on the red line

# Projection methods

- Projection methods “**project**” the invalid state onto the constraint manifold
- What does that mean?
  - Check the state
  - Is it invalid?
  - If so, find the “**closest**” state that is valid
  - Use that as the state



Constraint manifold (green lines)  
for two 1D particles constrained  
to be a fixed distance apart

Projection method moves an **invalid state** to the **nearest valid state**

# Game analysis 1: Elements and **mechanics**

# Thinking formally and analytically about games

- What do games in a given genre have in **common**?
- How do the pieces of a game **fit together**?
- What's **important** to the gameplay of a given game?
  - What's **central** versus added bells and whistles?
- What can I **change** in a game without **breaking** it?
- How can I **move ideas** from one game to another?

# The elements of a game

A simplified model

- **Players**

- One? Two? Many?
- All the same? Different roles

- **Resources**

- What do the players work with?
- RPGS: gold, experience, ammo, weapons, key items
- Chess: pieces, positions
- Poker: cards, your hand, chips, pot

- **Rules**

- What can the player do?
- What can't players do?
- What are players forced to do?
- What is done to players?

- **Objectives**

- What is the player working toward/away from?
- What are the player's values in the game?

# Mechanics

# Game mechanics

- Common term in game design
- Used loosely but means **something like**
  - **Combination** of **interlocking** game elements
  - That work together as a **unit**
  - To **engage** the player
  - These are often organized into interlocking groups, called **systems**

# Example: power-ups

- We might talk about the **mushrooms** in Mario Kart as being a mechanic
- But we actually don't mean **just** the mushroom, we mean the mushroom plus
  - The rule that mushrooms make you **move faster**
  - The rule that you can **choose** when to "fire it"
  - The fact that item boxes are **scarce**
  - The fact that power-ups last for a **limited time**

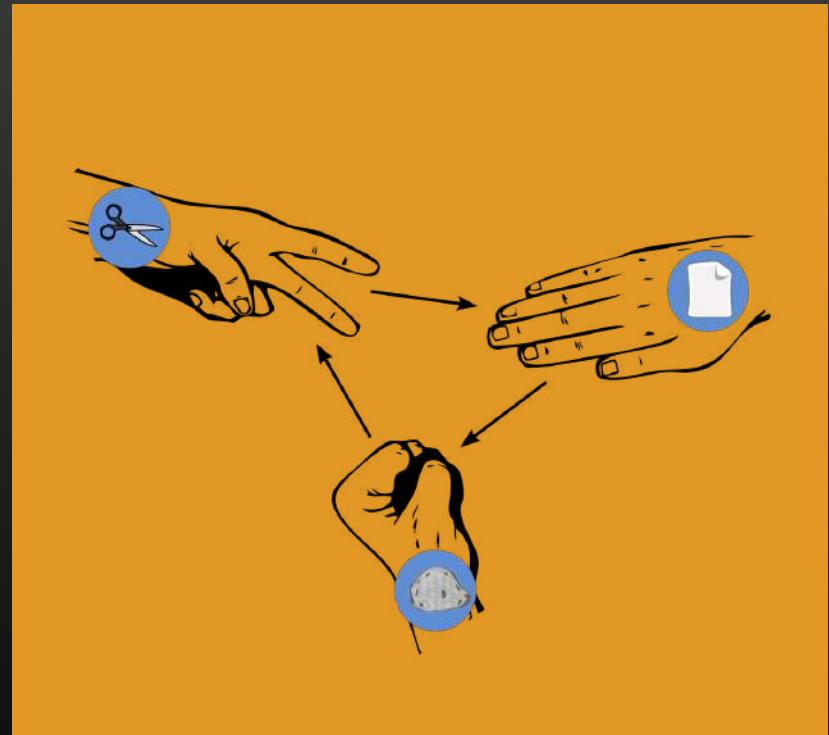
# What happens when we change these?

- Its **appearance** (e.g. a mushroom)
- The rule that it makes you **move faster**
- The rule that you can **choose** when to “fire it”
- The fact that item boxes are **scarce**
- The fact that it lasts for a **limited time**

Rock, paper, scissors

# Example: rock, paper scissors

- Three moves
- Each definitively **wins** or **loses** against each other move
- No move **dominates**
  - Each is better than one other and worse than the other



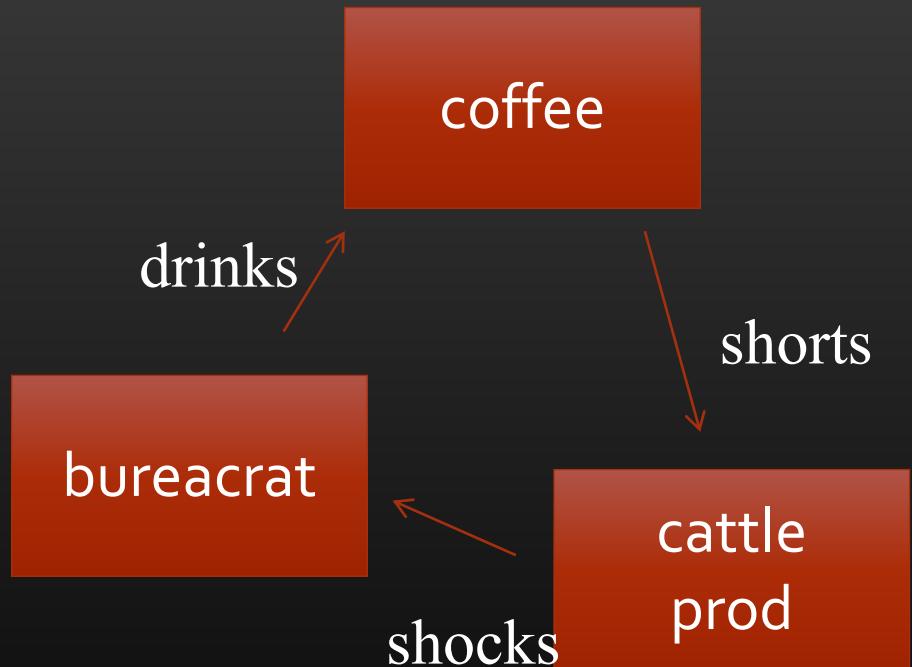
R-P-S is about  
**choice without a  
dominant choice**

# Theme and variation

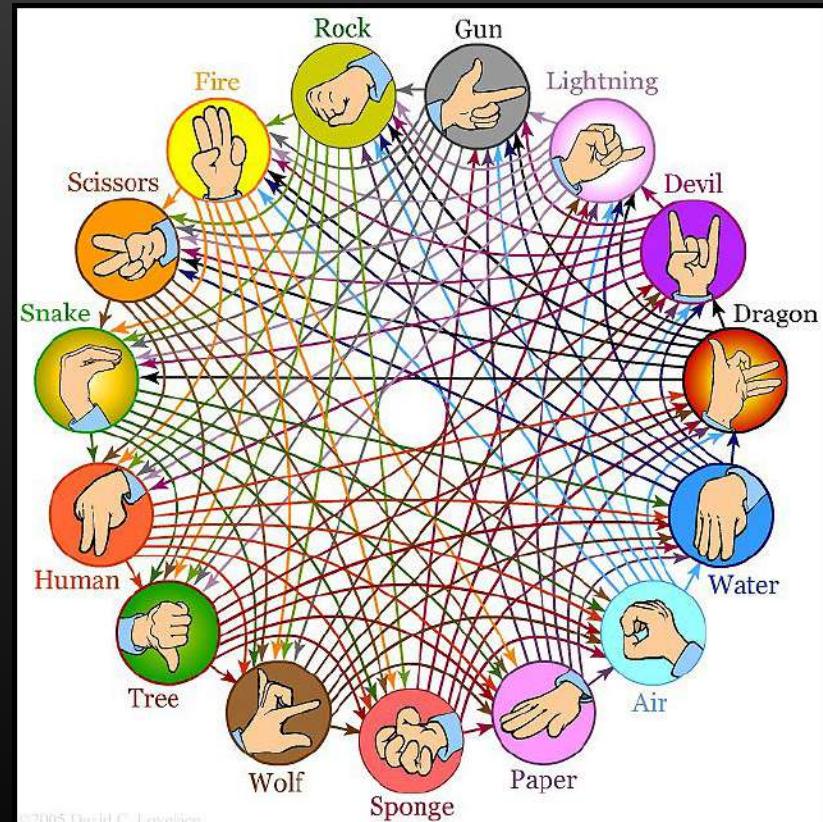
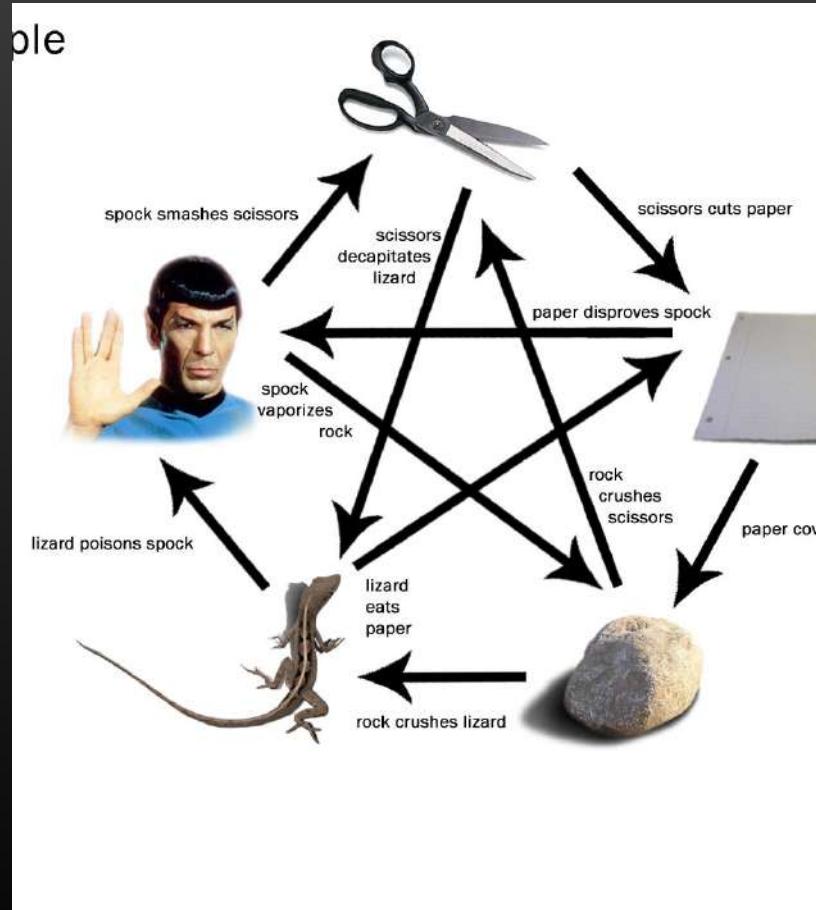
- We can understand the formal properties of a mechanic by imagining what we can **change** without **breaking** it
  - Making it non-fun or just making it feel different

# Changing labels

- Three moves
- Each definitively **wins** or **loses** against each other move
- No move **dominates**
  - Each is better than one other and worse than the other



# Changing count



# Changing count

- Two moves
- Bureaucrat **dominates**
  - Nobody would play coffee

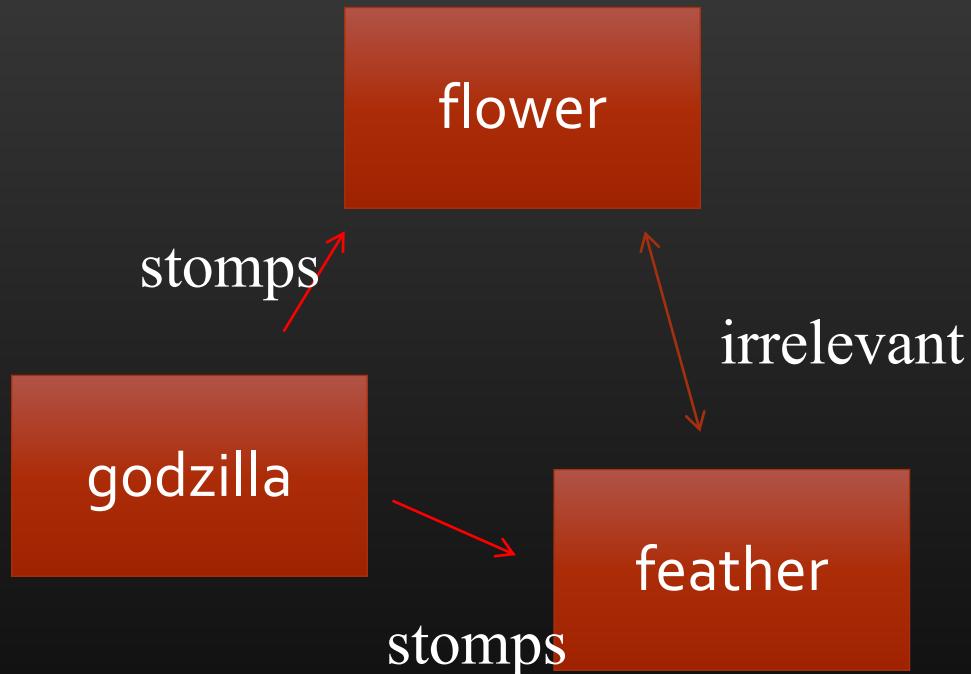
coffee

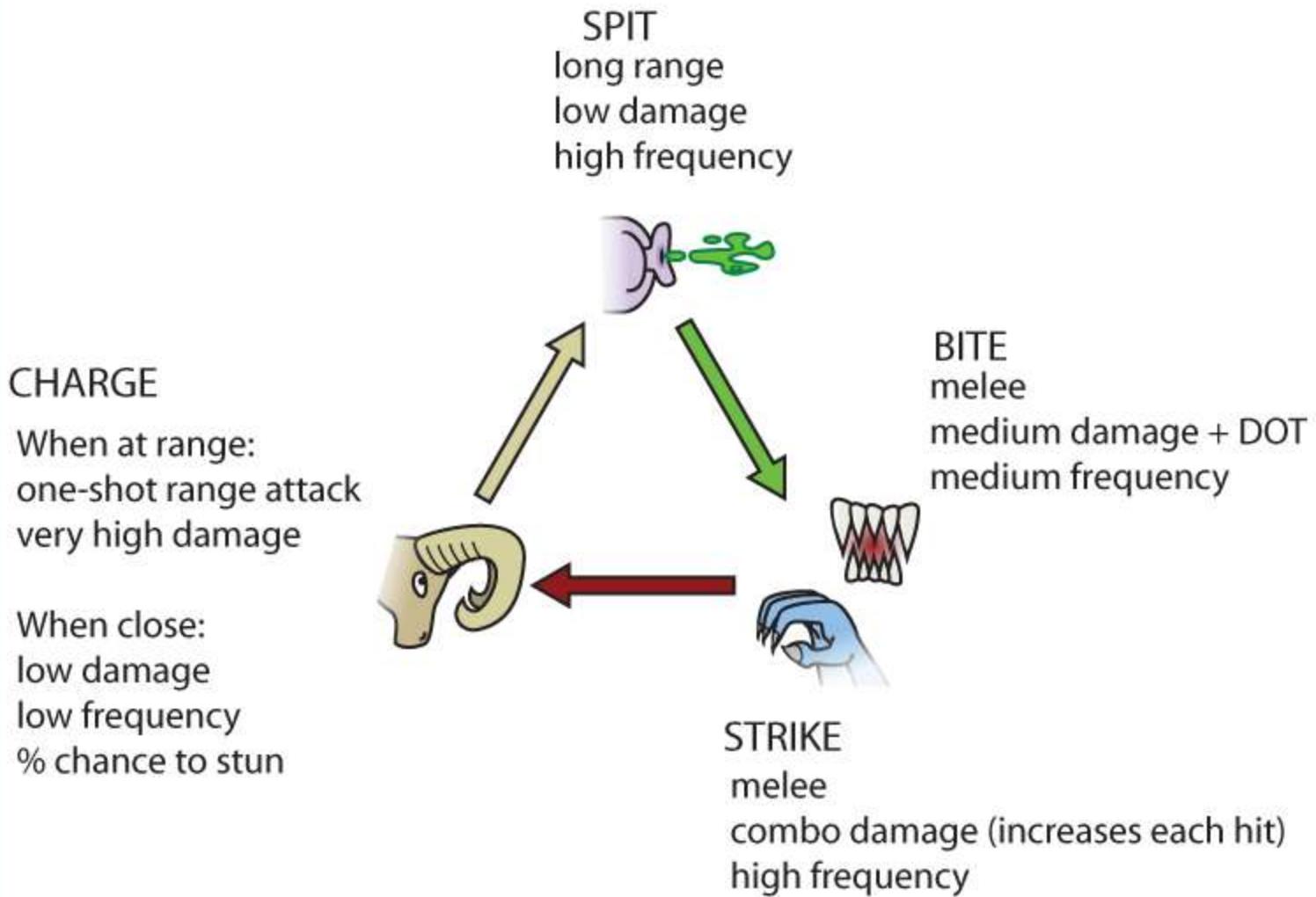
drinks ↗

bureaucrat

# Changing arrows

- Can't change **direction** of arrows without introducing dominant or pessimal strategy





# Temptation mechanics

# Example: temptation mechanics

E.g. **blackjack**

- Resources
  - Cards
  - Your hand
- Objective
  - Get sum of cards in your hand closer to 21 than the dealer **without going over**
- Rules
  - Can keep asking for more cards
  - Automatic loss if you go over 21

# Changing materials

E.g. **blackjack**

- Resources
  - ~~Cards~~ dice
  - Your hand
- Objective
  - Get sum of cards in your hand closer to 21 than the dealer **without going over**
- Rules
  - Can keep asking for more cards
  - Automatic loss if you go over 21

# Changing goal

E.g. **blackjack**

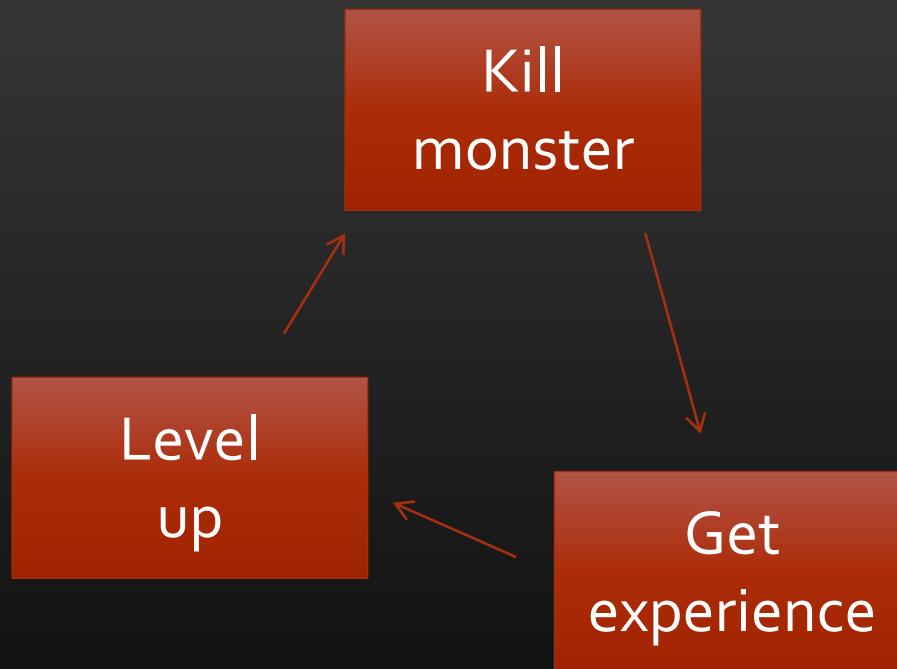
- Resources
  - Cards
  - Your hand
- Objective
  - Get sum of cards in your hand closer to ~~21~~ 31 than the dealer **without going over**
- Rules
  - Can keep asking for more cards
  - Automatic loss if you go over 31

# Changing it a lot

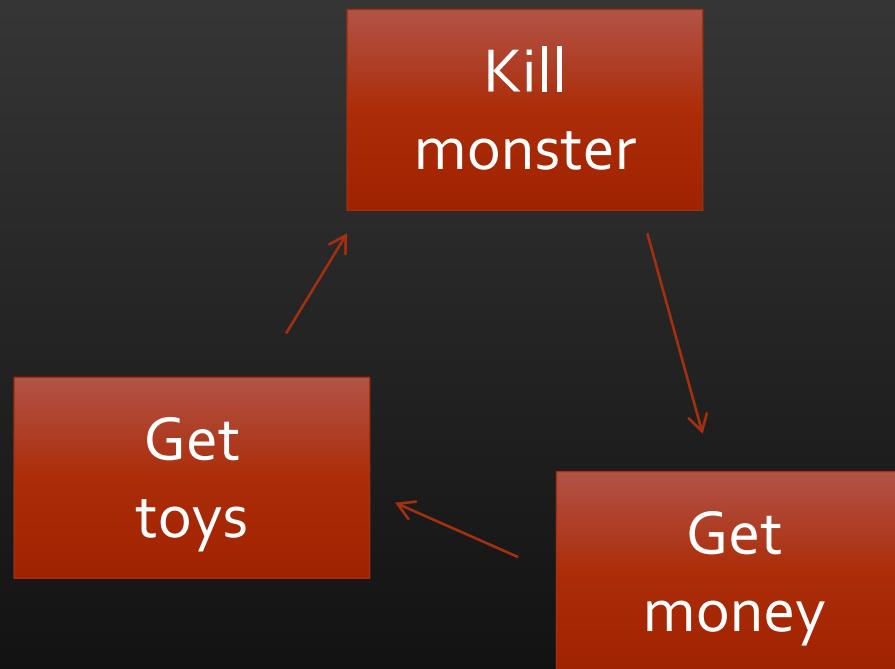
## Sabacc

- Cards have **signed values** (-17 to +15)
- Multiple players competing with one another
- Can draw as many cards as you want
- Winner is player with **absolute value** closest to **23** without going over

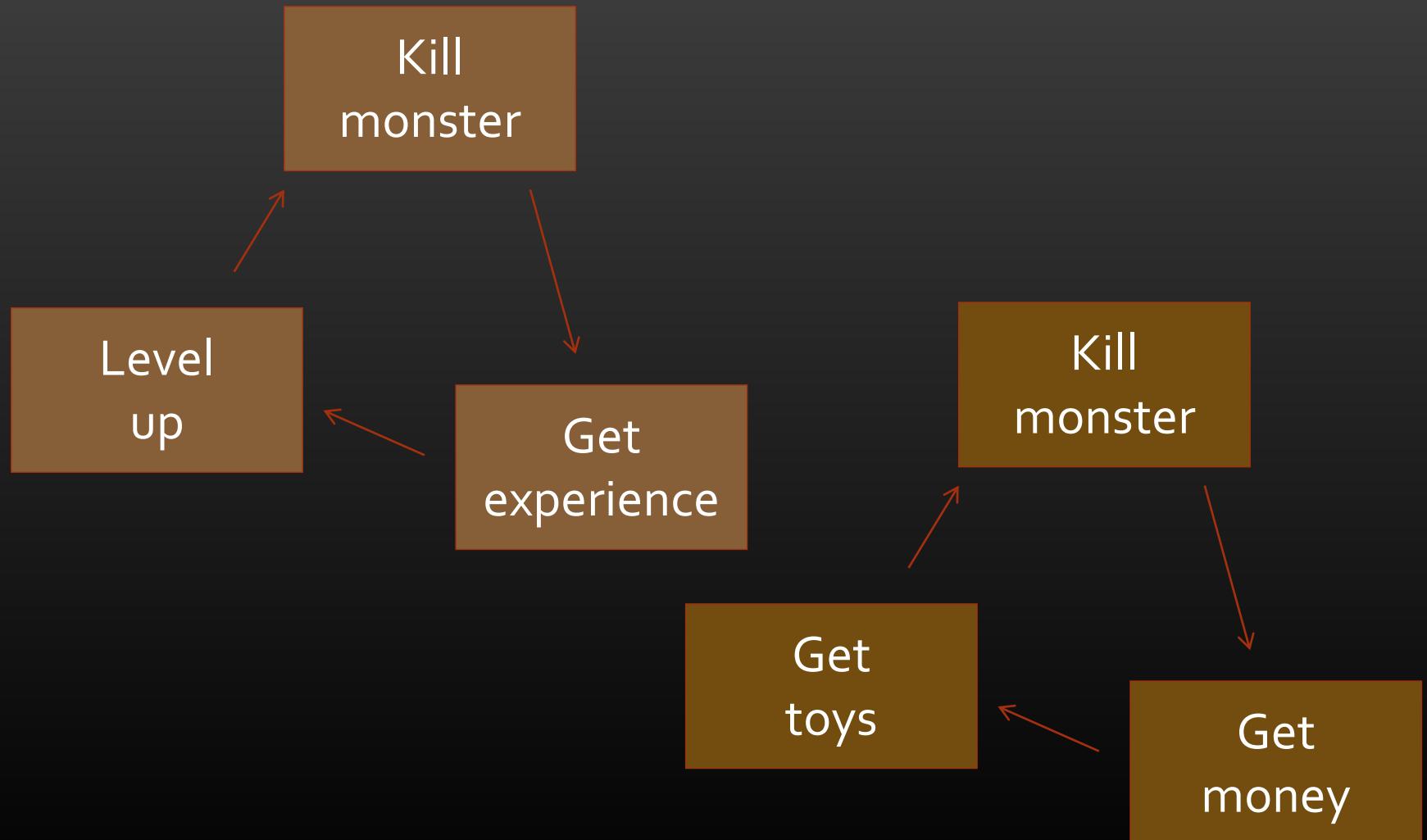
# Example: leveling up



# Changing resource



# Running them together



# Example: timing puzzles



# Example: platforming



# Core mechanics

# Core mechanics of a game

- The **central** mechanics that everything revolves around
- If you change them, it's a **fundamentally different** game

# Core mechanics and prototyping

- **Prototyping** is about trying to find the core mechanics of your game and perfect them
- **Paper simulation** making a paper game (board game, card game, ...) that captures the core mechanics of a digital game
- You're much better off if you can do paper simulations of your game as part of the development process

# Simulating Tony Hawk



# Paper simulation of Tony Hawk



Stone Lebrande: Paper simulations of digital games, GDC 2009

# Tony Hawk == Blackjack



# Game analysis 2: **Dynamics**

# Mechanics

- Combinations of **elements**
  - Resources, rules, objectives, players
- That interact to **work as a unit**
- Short-term effects (e.g. 1 turn)

# Dynamics

- **Long term patterns** of play
  - That result from the mechanics
- **Choreographed** dynamics
  - **Escalation** of challenge
  - **Pedagogy** (introducing one mechanic at a time)
- **Emergent** dynamics
  - **Not specifically intended** by the designers

# Emergent dynamics

- **Camping** at spawn points
- Everyone **picking on the weak** player
- Everyone **picking on the leader**
- **Grinding** / gold farming

# Positive feedback systems

- **Self-reinforcing** pattern of gameplay
  - The more it happens, the more it **accelerates**
- Examples
  - **Leveling** up
  - **Acquisition** in Monopoly
  - Players focusing on particular skills in an RPG
  - **Level differences** between players
    - E.g. South Park's *Make Love, Not Warcraft*



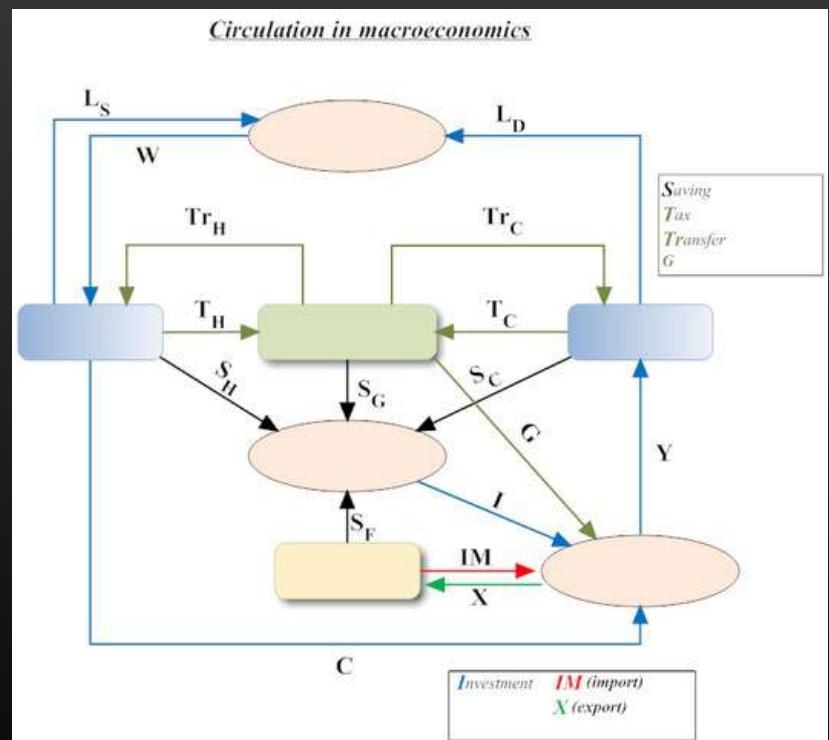
# Negative feedback systems

- Self-interfering patterns of gameplay
  - More there is, the faster it shrinks
- Examples
  - Leader/loser in Bullshit
  - Dynamic difficulty systems (monsters scaled to player)



# Economies

- Games are often complex **networks of feedback loops**
- Act like **macroeconomic systems**



# Resource dynamics

# Example: resources in RPGs

- **Killing monsters** give you
  - **Gold**
  - Experience
- **Gold** gives you
  - **Weapons**
  - Ammo
  - Armor
  - Health/mana/whatever
- **Weapons**, etc. let you **kill more monsters**



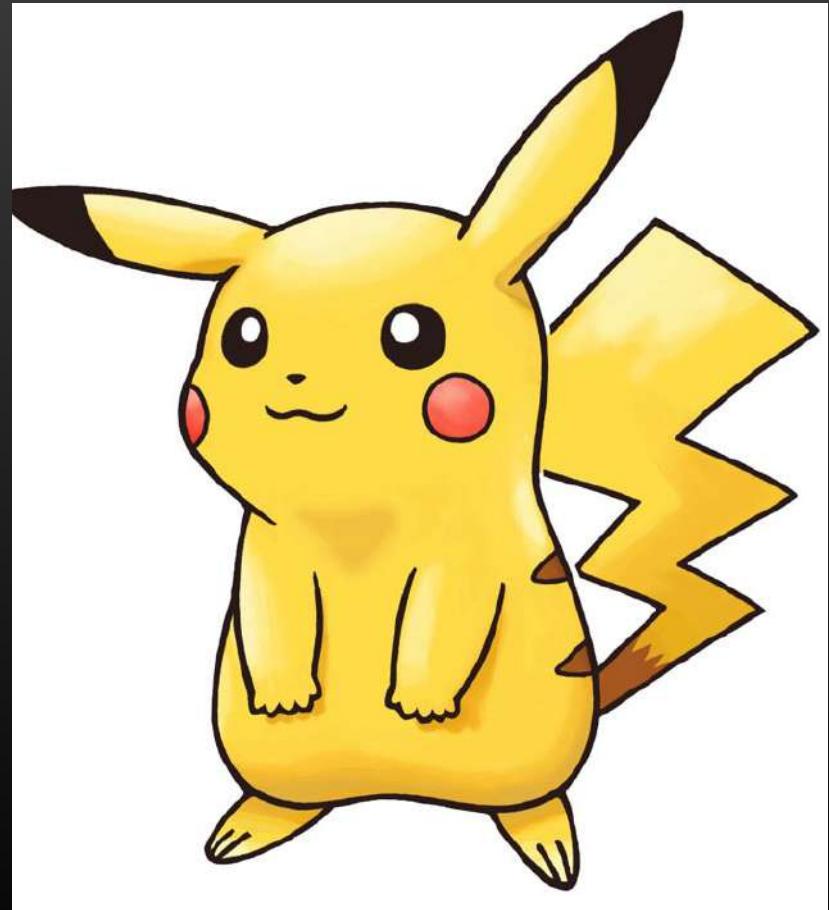
# Example: resources in RPGs

- Monsters have **cash value**
- But cash has “**weapon value**”
  - So **monsters have weapon value**
  - **Cash has monster value**
  - In some sense they’re all **interconvertible**
- In **tuning** your game, you need to get the “**exchange rates**” just right
  - Or the **feedback loops** behave badly



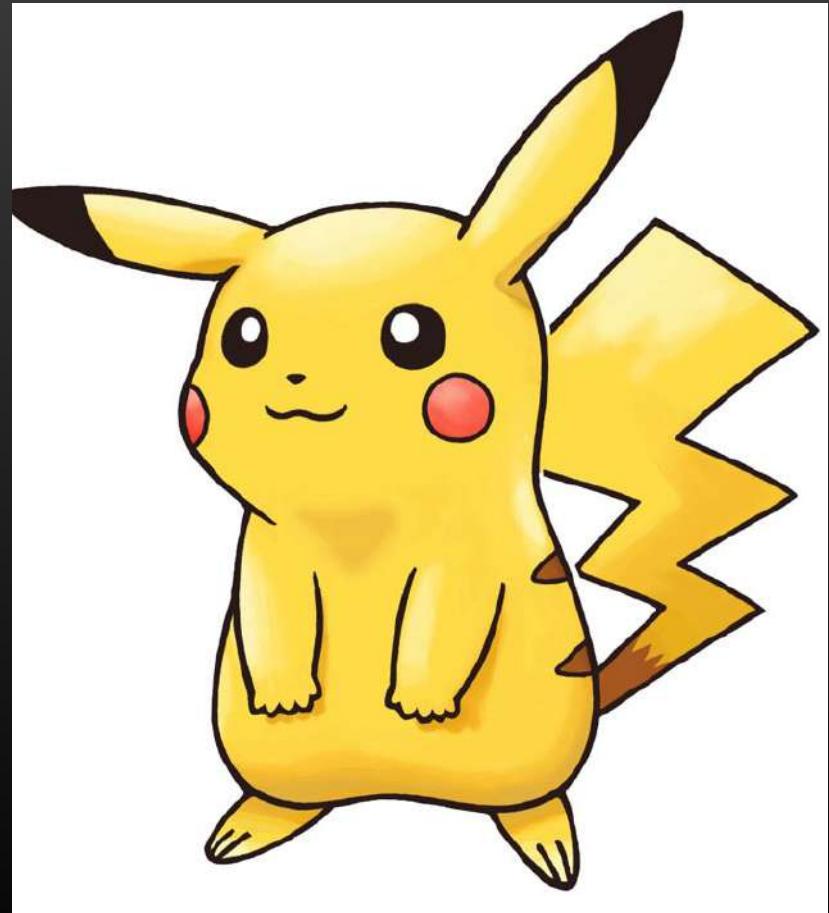
# Example: Pokemon Hunter

- Suppose
  - Pikachu takes 10 bullets to kill on average
  - And a Pikachu pelt is worth 50 GP
  - And bullets are 10GP each
- What does this tell you?



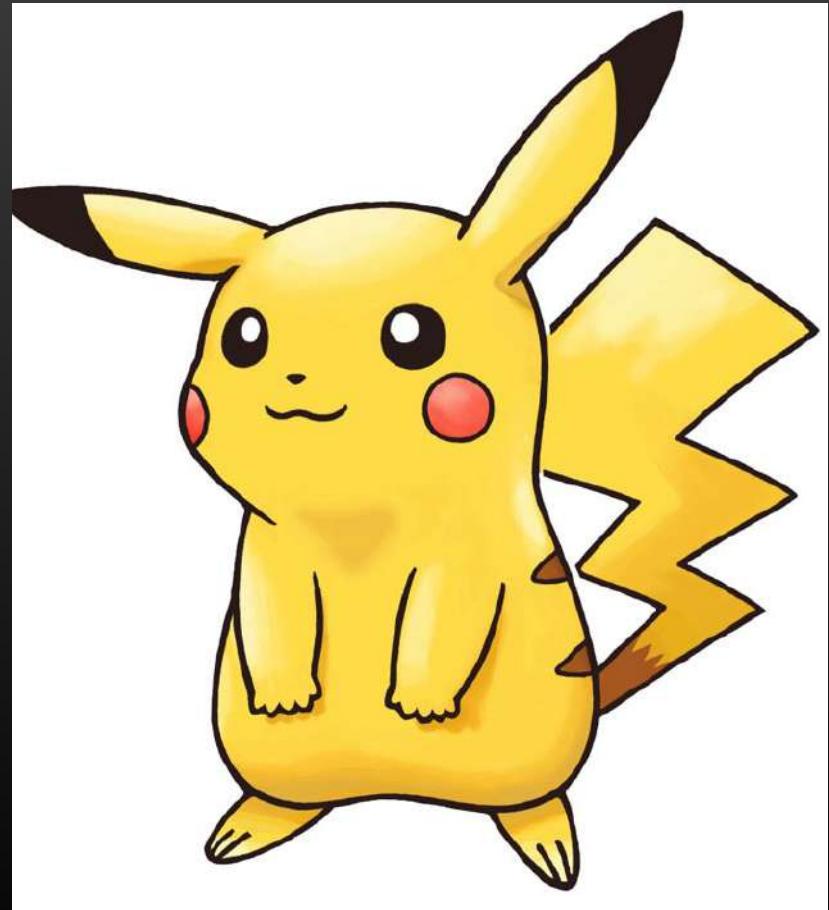
# Don't mess with the Pikachu

- It's **not worth it**
- You'll use 100GP worth of bullets
- To get back 50GP



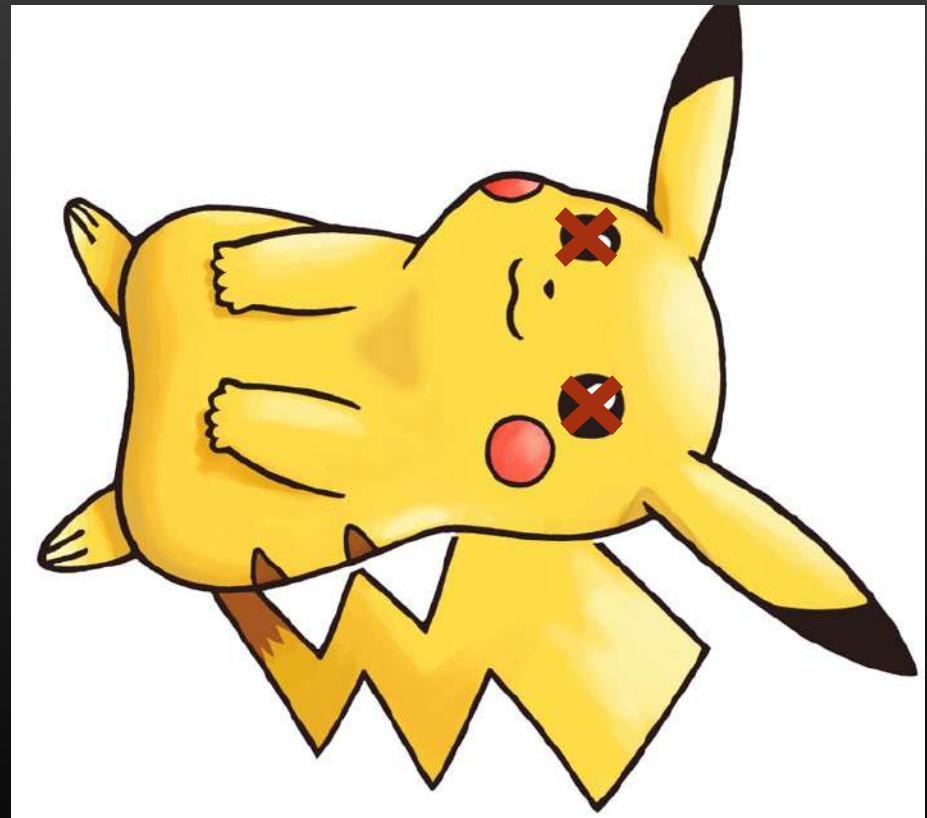
# Simple example

- Suppose
  - Pikachu takes **1 bullet** to kill on average
  - And a Pikachu pelt is worth 50 GP
  - And bullets are 10GP each
- What does this tell you?



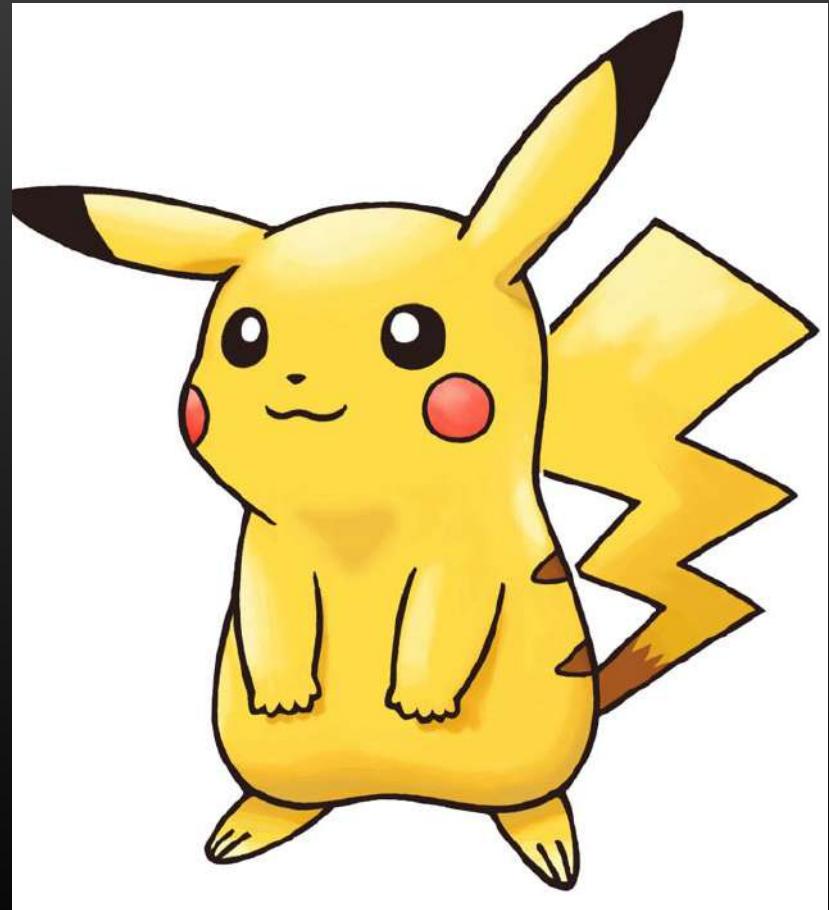
# Massive Pikachu depopulation

- **Easy** to kill
- **Pays** well
- If
  - There's an **infinite population** of them
- Then
  - you've given people an **infinite source of money** they can mine forever
- **Money is free!**



# Well, almost

- What you've actually done now is to make **player time** the scarce resource
- This has two different, possibly **unintended consequences**



# Punishing newcomers

- If it's a multiplayer game, it **won't be fun** for
  - **Newcomers**
  - Players with **limited free time**



# Gold farming

- Now people start **paying other people to play** for them
- So they can level up without having to put in the required time



Note: Some many of these slides  
are lovingly stolen from Mark Leblanc

# Game analysis 3: **Aesthetics**

# Mechanics

- Combinations of **elements**
  - Resources, rules, objectives, players
- That interact to **work as a unit**
- **Short-term effects** (e.g. 1 turn)

# Dynamics

- **Long term** patterns of play
  - That result from the mechanics
- **Choreographed** dynamics
  - Escalation of challenge
  - Pedagogy (introducing one mechanic at a time)
- **Emergent** dynamics
  - Not specifically designed in by rules

# Aesthetics

- Mental **experience created** within the player by the game
  - Especially emotional experience
- **MDA** model of **Hunicke, Leblanc, and Zubek**
  - Mechanics create dynamics
  - Dynamics create aesthetic responses

Leblanc's 8 kinds of fun

# Clarifying Our Aesthetics

- **Charades** is “fun”
- **Counter-Strike** is “fun”
- **Final Fantasy** is “fun”

Fun is **not a useful word**

# Understanding Aesthetics

- We need to **get past** words like “fun” and “gameplay.”
- What **kinds of “fun”** are there?
- How will we **recognize a particular kind** of “fun” when we see it?

# Eight Kinds of “Fun”

# Eight Kinds of “Fun”

- Sensation *Game as art object*

# Eight Kinds of “Fun”

- Sensation
- Fantasy *Game as make-believe*

# Eight Kinds of “Fun”

- Sensation
- Fantasy
- Narrative *Game as unfolding story*

# Eight Kinds of “Fun”

- Sensation
- Fantasy
- Narrative
- Challenge *Game as obstacle course*

# Eight Kinds of “Fun”

- Sensation
- Fantasy
- Narrative
- Challenge
- Fellowship

*Game as social framework*

# Eight Kinds of “Fun”

- Sensation
- Fantasy
- Narrative
- Challenge
- Fellowship
- Discovery

*Game as uncharted territory*

# Eight Kinds of “Fun”

- Sensation
- Fantasy
- Narrative
- Challenge
- Fellowship
- Discovery
- Expression

*Game as soap box*

# Eight Kinds of “Fun”

- Sensation
- Fantasy
- Narrative
- Challenge
- Fellowship
- Discovery
- Expression
- Submission

*Game as mindless pastime*

# Clarifying Our Aesthetics

**Charades** is

Fellowship, Expression, Challenge

**Counter-Strike** is

Challenge, Sensation, Competition, Fantasy

**Final Fantasy** is

Fantasy, Narrative, Expression, Discovery, Challenge,  
Masochism

*Each game pursues multiple aesthetics.*

*Again, there is no Game Unified Theory.*

# Aesthetic goals

# Clarifying Our Goals

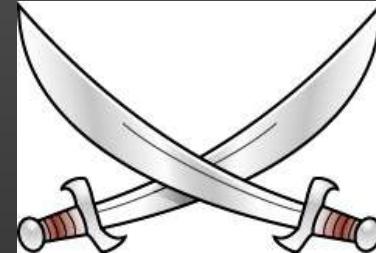
- As **designers**, we can choose certain **aesthetics as goals** for our game design.
- But we need **more than a one-word definition** of our goals.

# Formulating an “Aesthetic Model”

- For each **aesthetic goal**:
  - Write a formal **definition**
  - List criteria for **success**
  - List **modes of failure**
- Serves as an “**aesthetic compass**”
- These are often **reusable**

*Some examples...*

# Goal: Competition



- Definition: A game is **competitive** if players are **emotionally invested in defeating each other**.
- Success:
  - Players are **adversaries**
  - Players **want to win**
- Failure:
  - Players feel that they **can't win**
  - Players can't measure their **progress**

# Goal: Pirate Fantasy



- Definition: A pirate fantasy
  - **Conforms to the genre conventions** of pirate movies
  - **Permits the player** to engage in certain kinds of **anti-social pirate behavior**.

# Goal: Pirate Fantasy



## Success

- **Role play**
- **Empowerment**
- **Independence**
- Greed
- Treachery
- Prey upon Weak

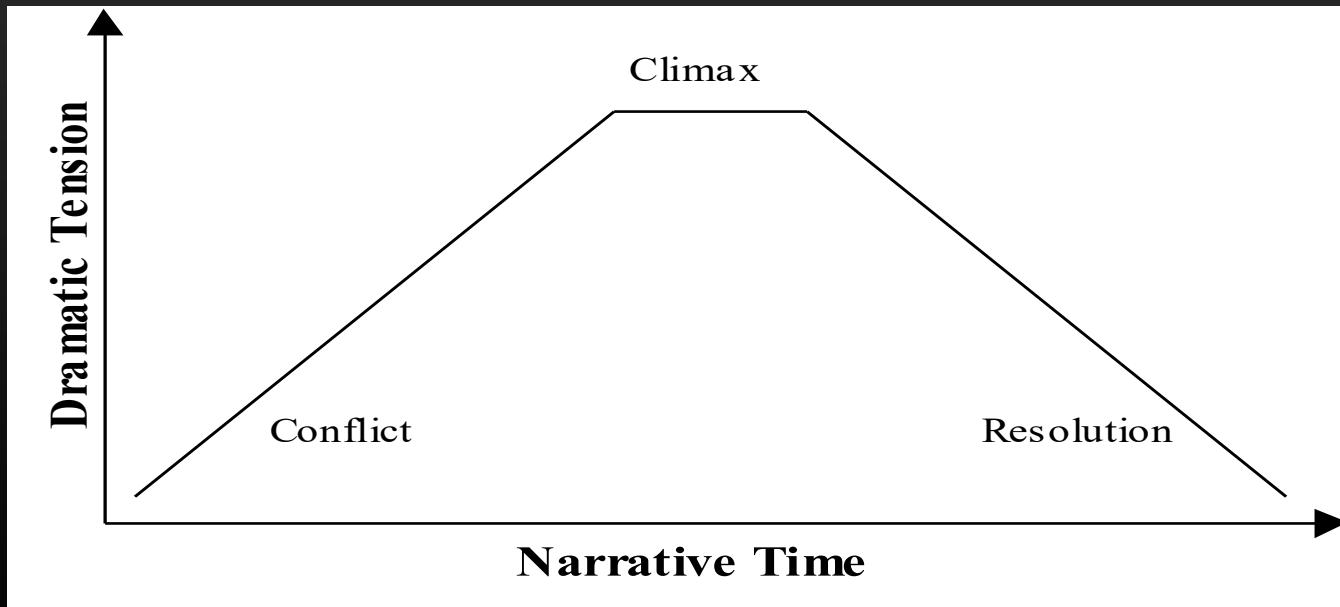
## Failure

- **Vulnerability**
- Compassion
- Generosity

# Goal: Drama



- Definition: A game is **dramatic** if:
- Its central conflict creates dramatic **tension**
- The dramatic tension builds towards a **climax**



# Goal: Drama



- Success:
  - A sense of **uncertainty**
  - A sense of **inevitability**
  - Tension increases towards a **climax**
  
- Failure:
  - The conflict's **outcome is obvious** (no uncertainty)
  - No **sense of forward progress** (no inevitability)
  - Player **doesn't care** how the conflict resolves.

# Designing for aesthetics

# How do we get from...

- Cards
- Chips
- Rules

# To...

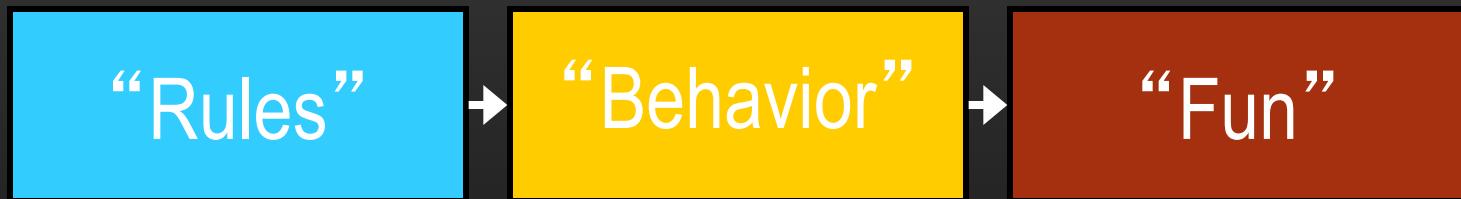
- Cards
- Chips
- Rules
- Intrigue
- Challenge
- Drama

# What's missing?

“Rules”

“Fun”

# The causal link...

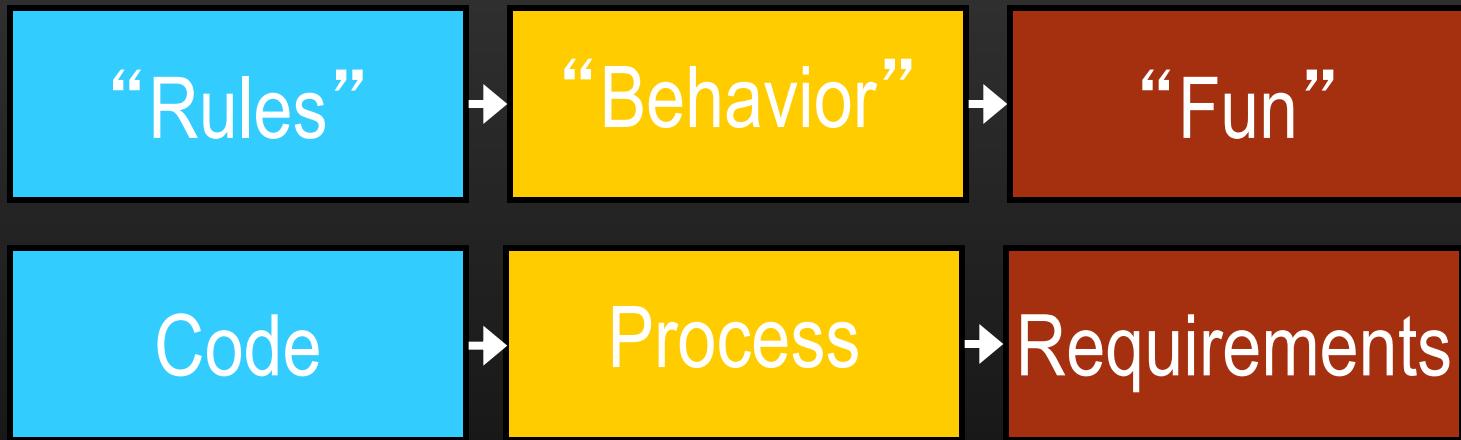


*This is what sets games apart...*

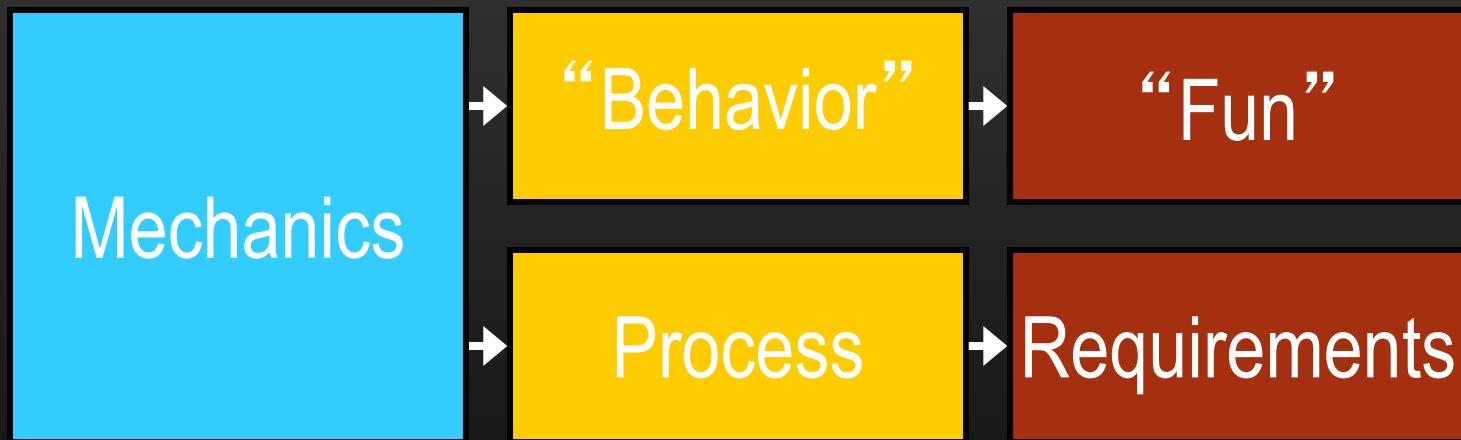
# Games As Software



# Games As Software



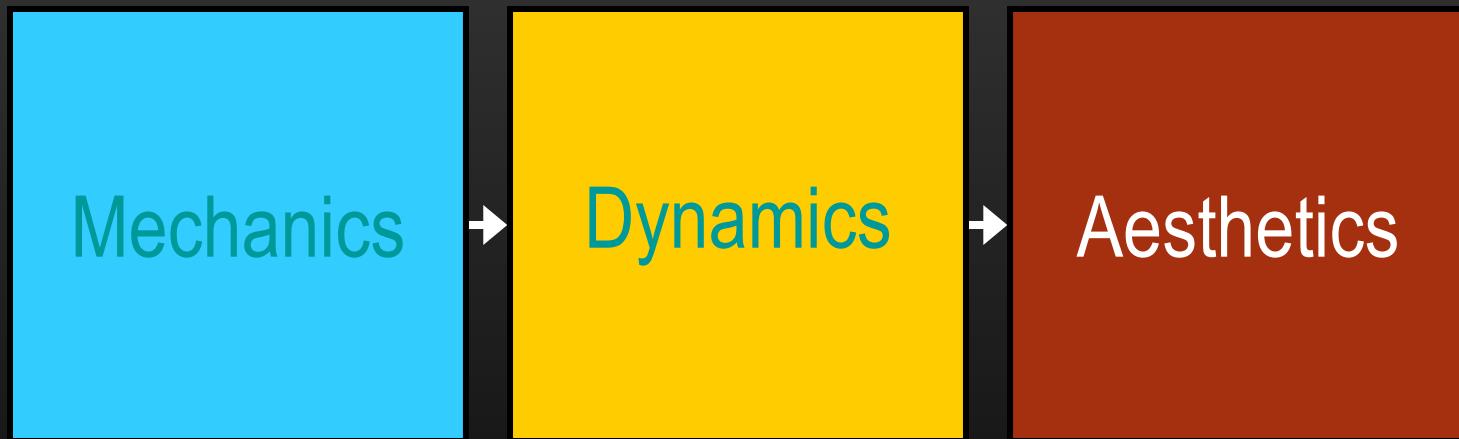
# A Design Vocabulary



# A Design Vocabulary



# A Design Vocabulary



# The MDA Framework



# Definitions

- **Mechanics**

The **rules and concepts** that formally specify the **game-as-system**

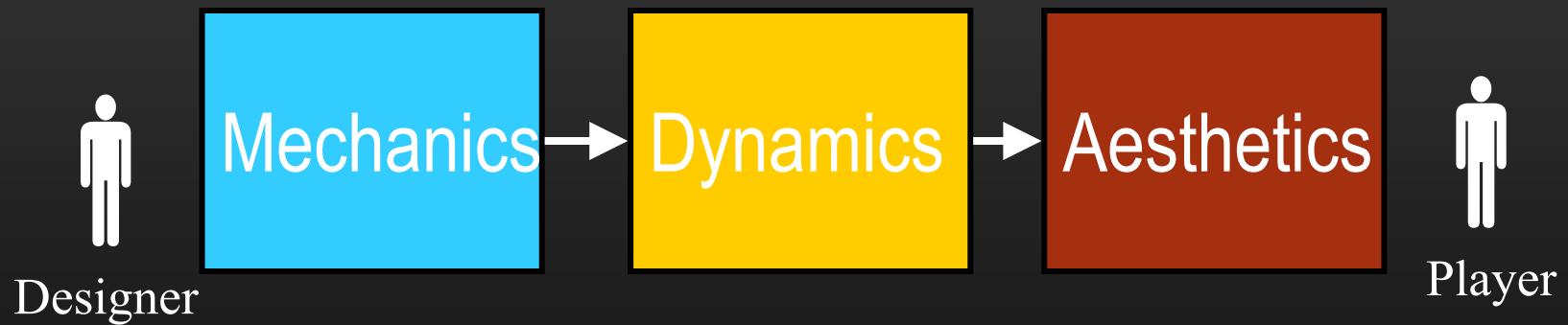
- **Dynamics**

The **run-time behavior** of the game-as-system.

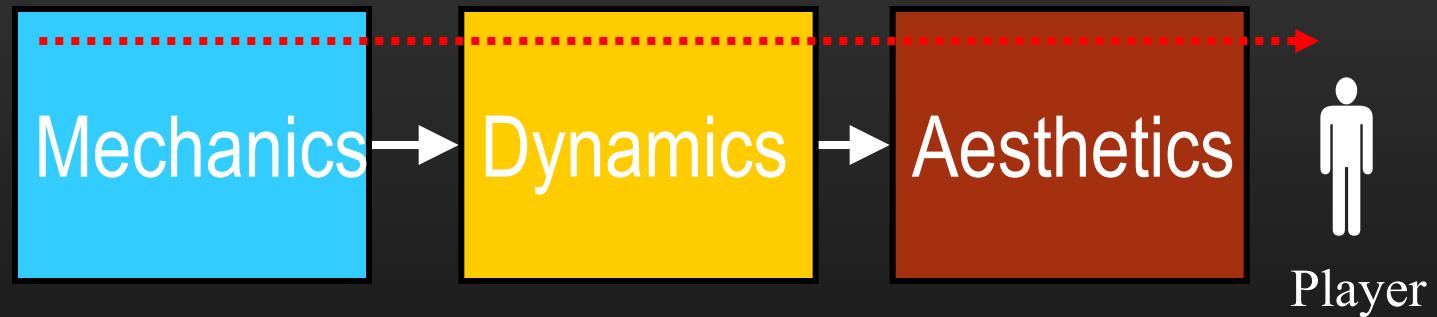
- **Aesthetics**

The **desirable emotional responses** evoked by the game dynamics.

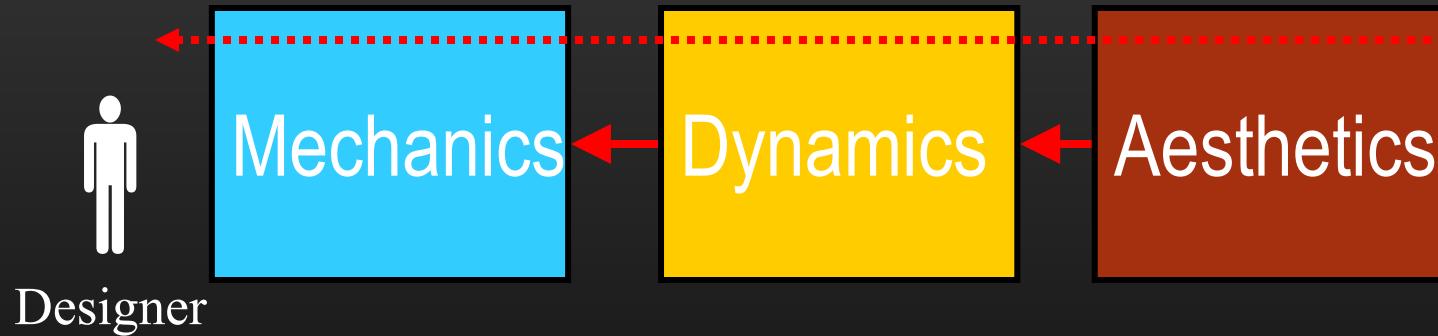
# The Designer/Player Relationship



# The Player's Perspective



# The Designer's Perspective



# Reading

- Read the **MDA paper** (on Canvas, under readings)

# Collision detection

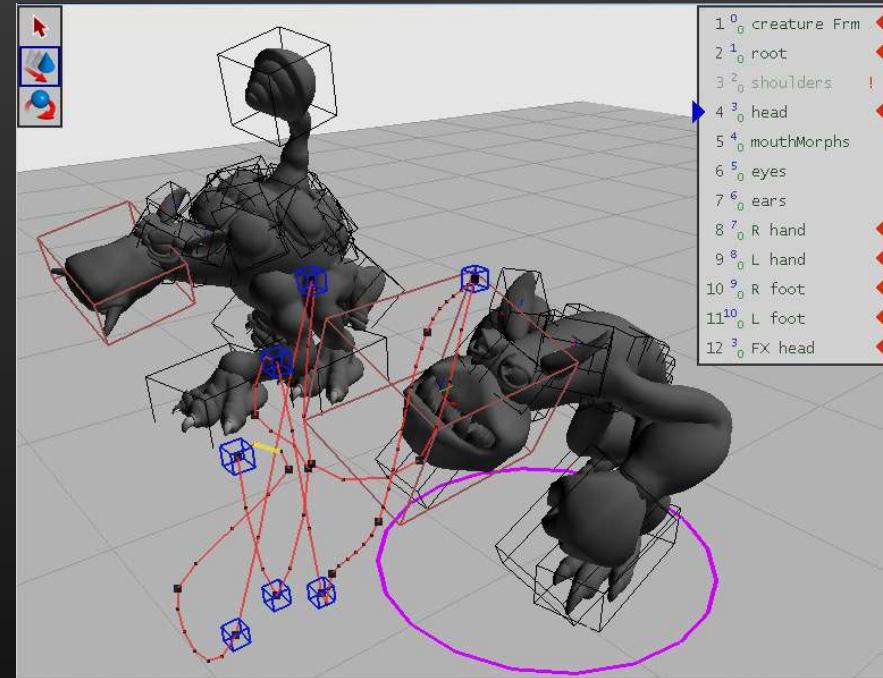
# Collision detection

- Performed on each physics update
- Usually just an **intersection test**
  - Check if objects **overlap** after they've been moved
- But there are fancier versions such as  
**“continuous” collision detection**
  - We'll talk about these later

# Collision geometry vs. rendering geometry

Collision detection has **its own separate shape representation**

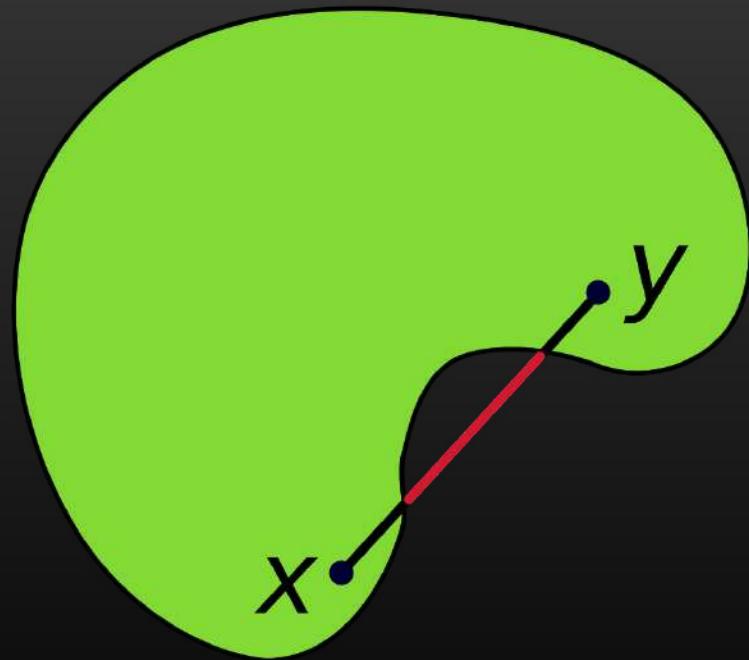
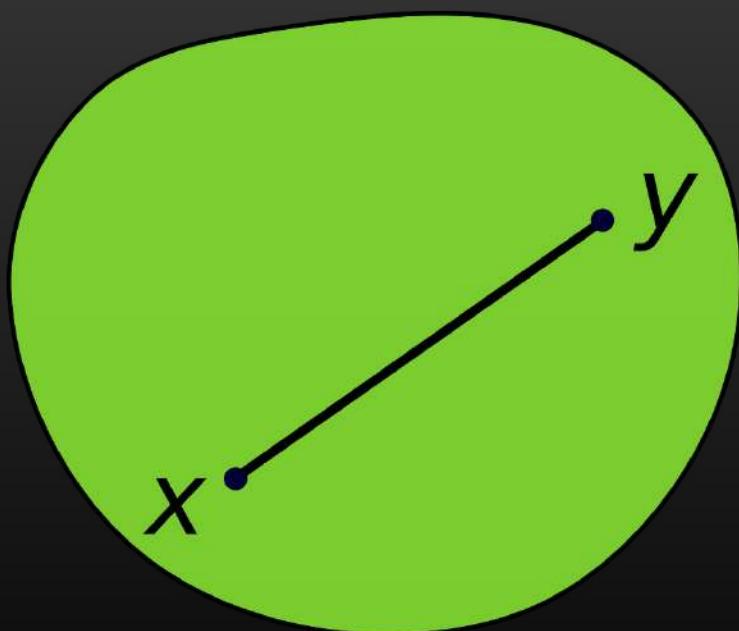
- Collision detection is
  - Hard
    - Need to **simplify**
    - Preprocess for **hierarchies** or other optimizations that rendering doesn't care about
  - Different
    - **Volumes**, not surfaces
    - Divide into **convex** parts
  - Wired into physics and gameplay
    - May connect specific collision regions to specific event handlers
- Common approaches
  - **Approximate** using **standard shape primitives** (spheres, boxes)
  - Convex polyhedra



Source: Hecker et al., SIGGRAPH 2008

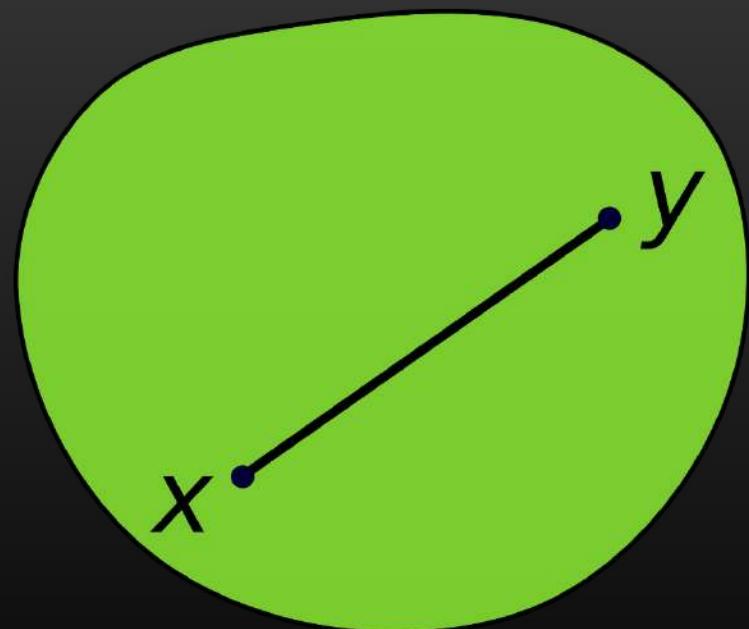
# The separating axis theorem

# Convexity and concavity



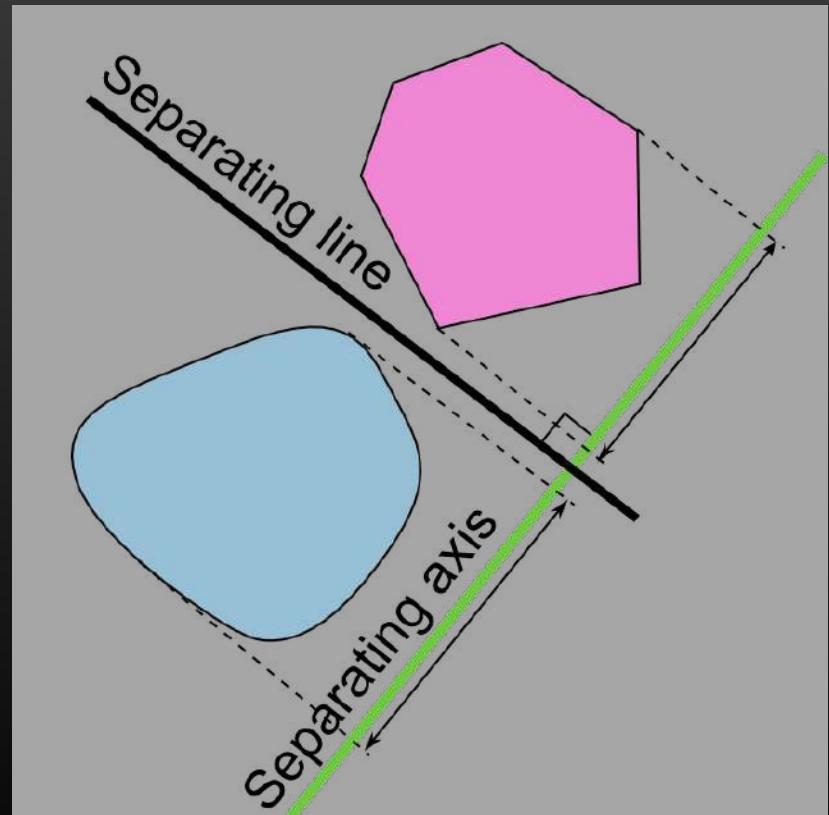
# Convex shapes

- A **shape is convex** if
  - Line **segments**
  - Joining two **points in the shape**
  - Always lie entirely **within the shape**
- All the basic collision tests **only work on convex shapes** :-)



# Separating axis theorem

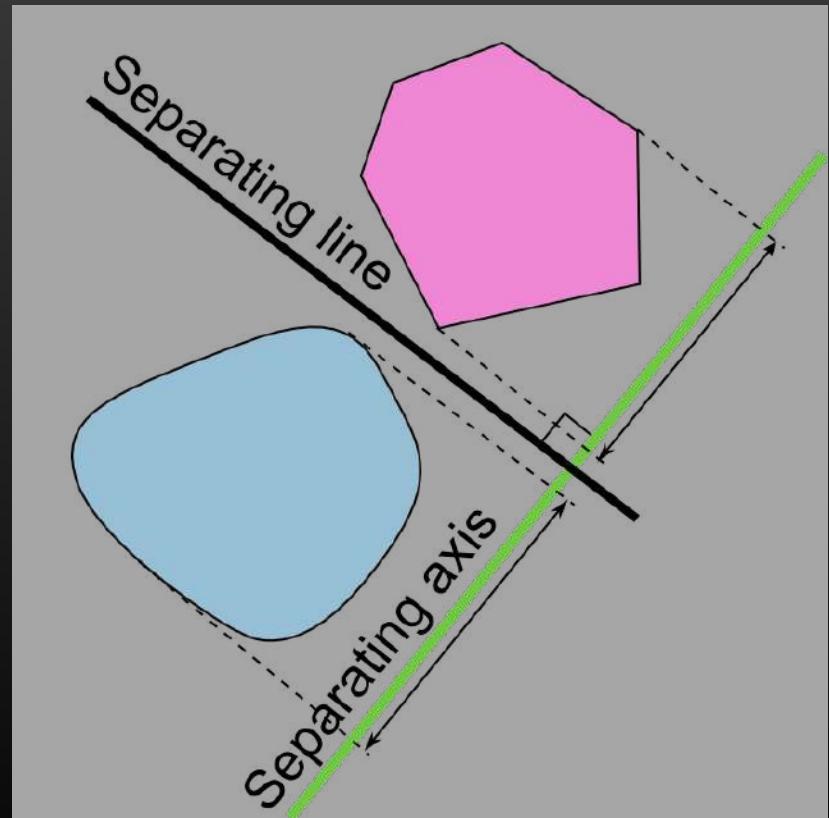
- If two **convex shapes** are **non-intersecting**
- There exists an **axis that separates them**
  - There is a **separating line/plane**
  - **Perpendicular** to the axis
  - That separates them
    - One object on **one side**
    - Other object on the **other side**



# Collision detection

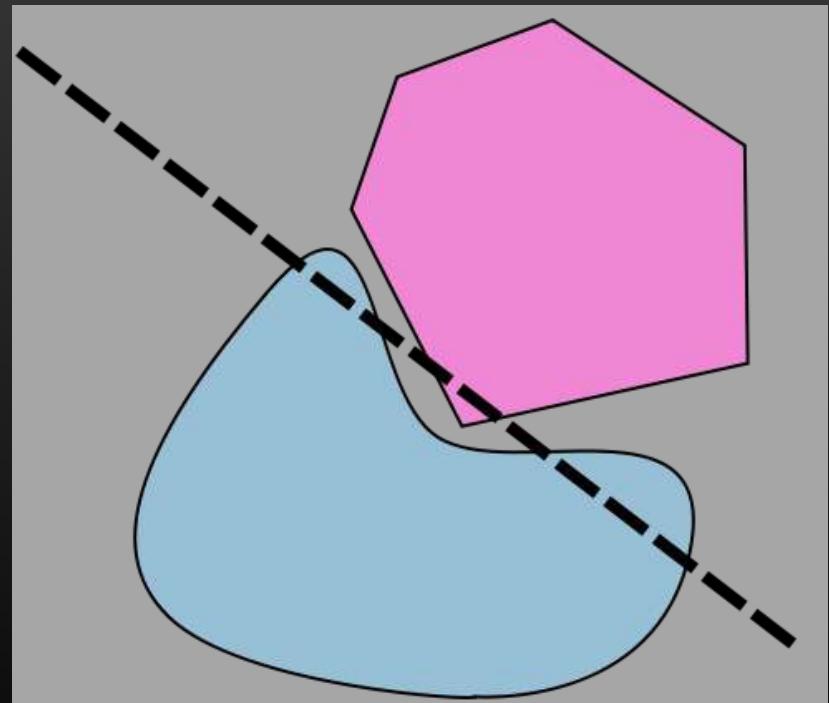
Simple collision  
detection algorithms

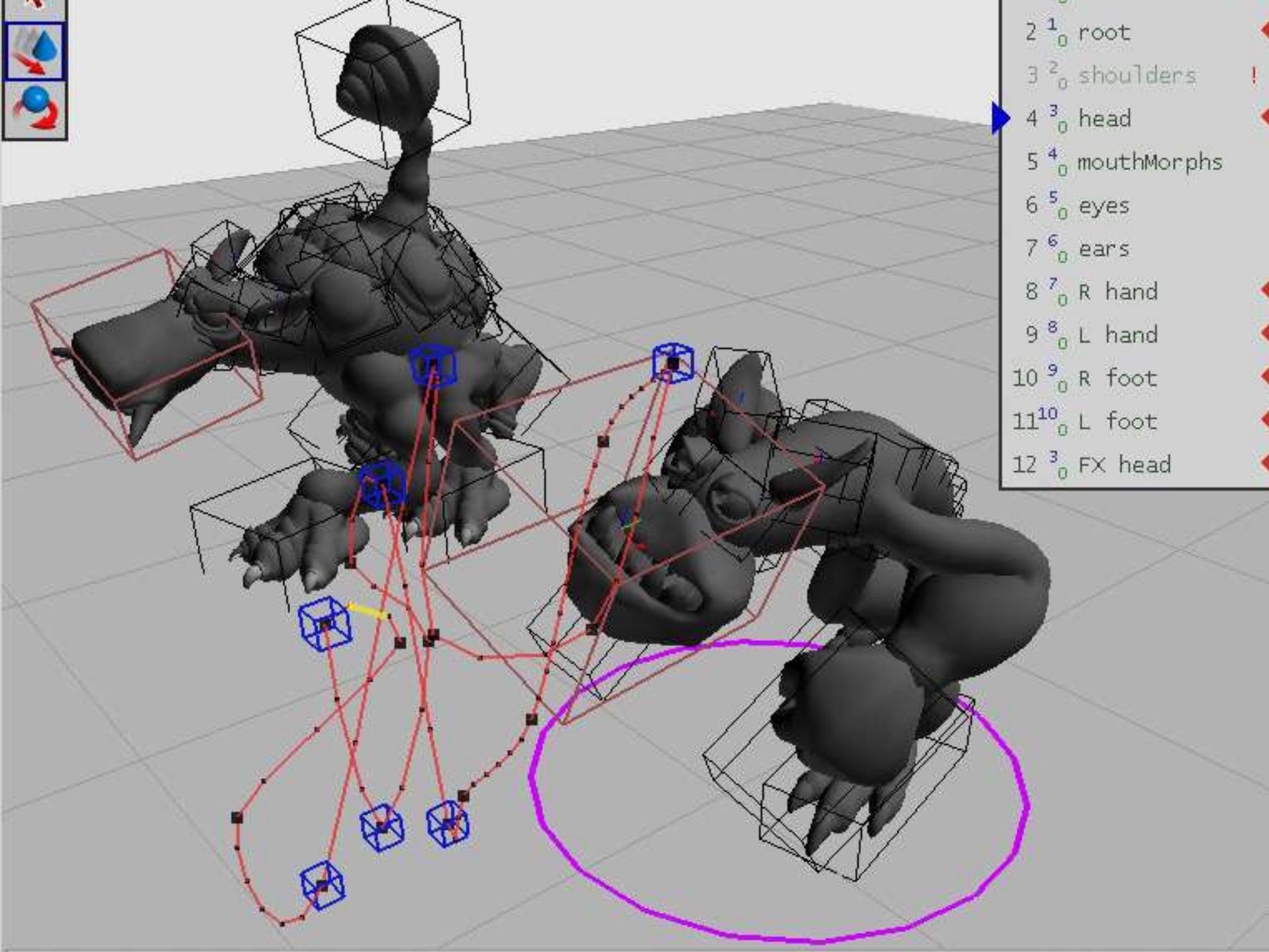
- **Search** for a separating axis
- Declare **collision if they can't find one**



# Separating axis theorem doesn't hold for concave objects

- If one of the objects is concave,
  - There **might not be a separating axis**
  - Even for **non-intersecting** objects
- So in practice, we end up
  - **Breaking up** concave objects
  - Into multiple **convex regions**





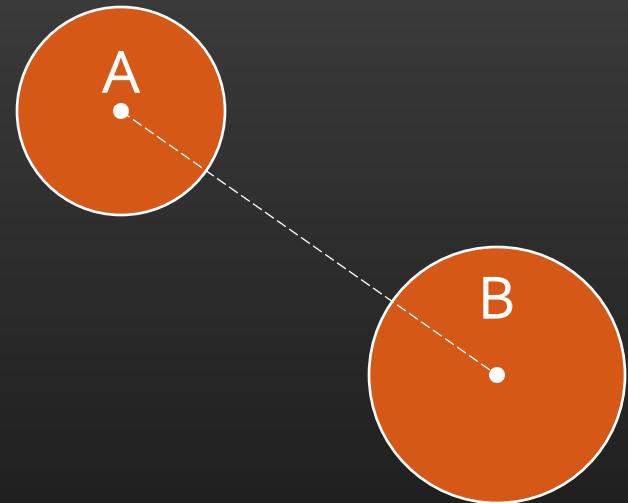
# Primitive shapes that are easy to test for collision

- The easiest shapes to test for collision are the ones where you **know in advance** what their possible **separating axes** would have to be
- These tend to be very **symmetrical** shapes
  - **Circles/Spheres**
  - Circle/Sphere-swept volumes (**SSVs**)
  - **Boxes**

# Sphere-swept volumes

# Sphere/sphere intersection test

- Spheres are the **fastest**, simplest case for collision testing
- If there's a **separating axis**, it will be the one **joining their centers**
- So you need only
  - Find the **distance between their centers**
  - Compare it to the **sum of their radii**
  - More efficient to compare **squared distances**
    - Don't have to do square root



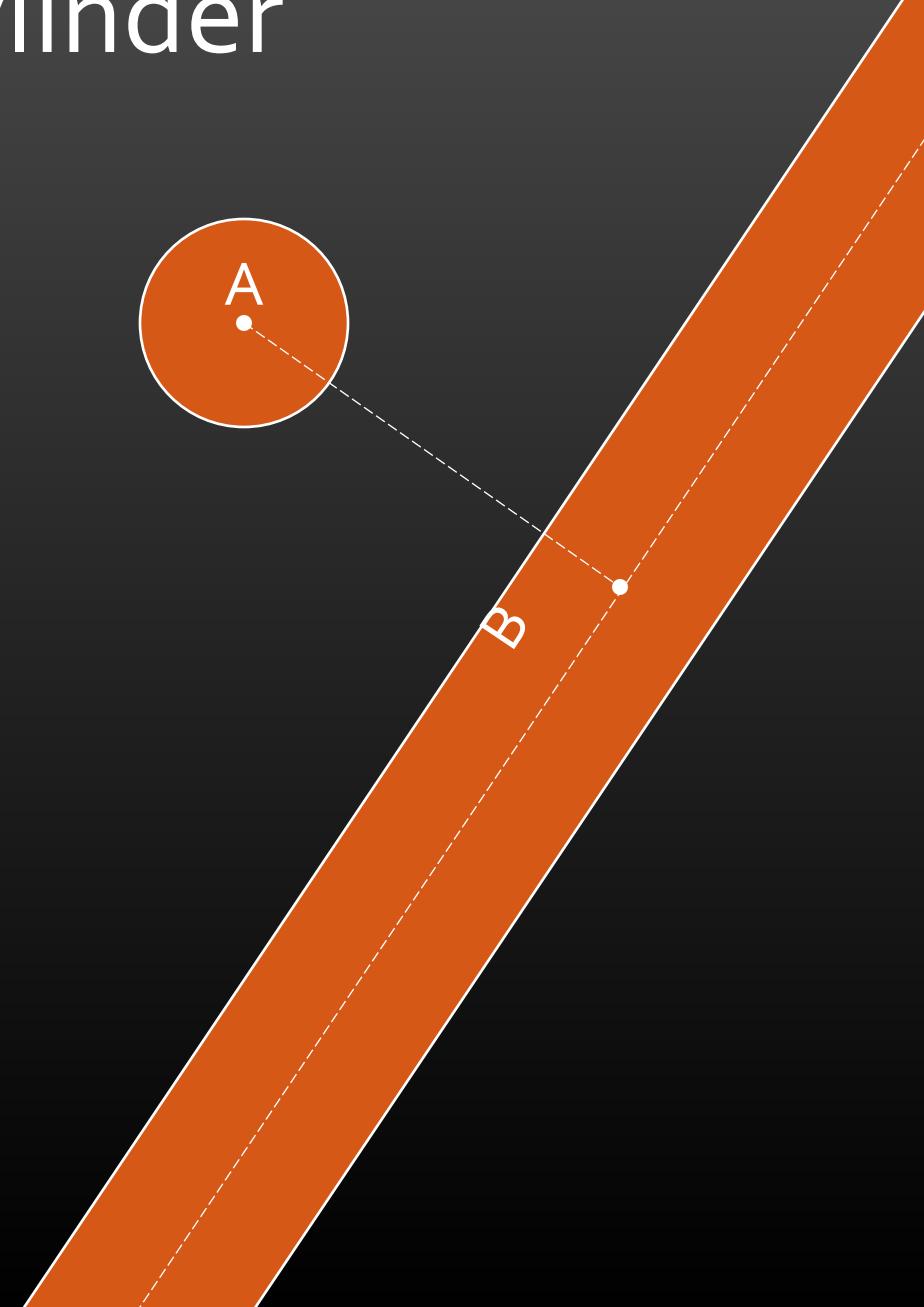
$$\|A - B\| < A_r + B_r$$
$$\|A - B\|^2 < (A_r + B_r)^2$$

Where:

- $A, B$  are the spheres' locations
- $A_r, B_r$  are their radii

# Sphere/infinite cylinder intersection test

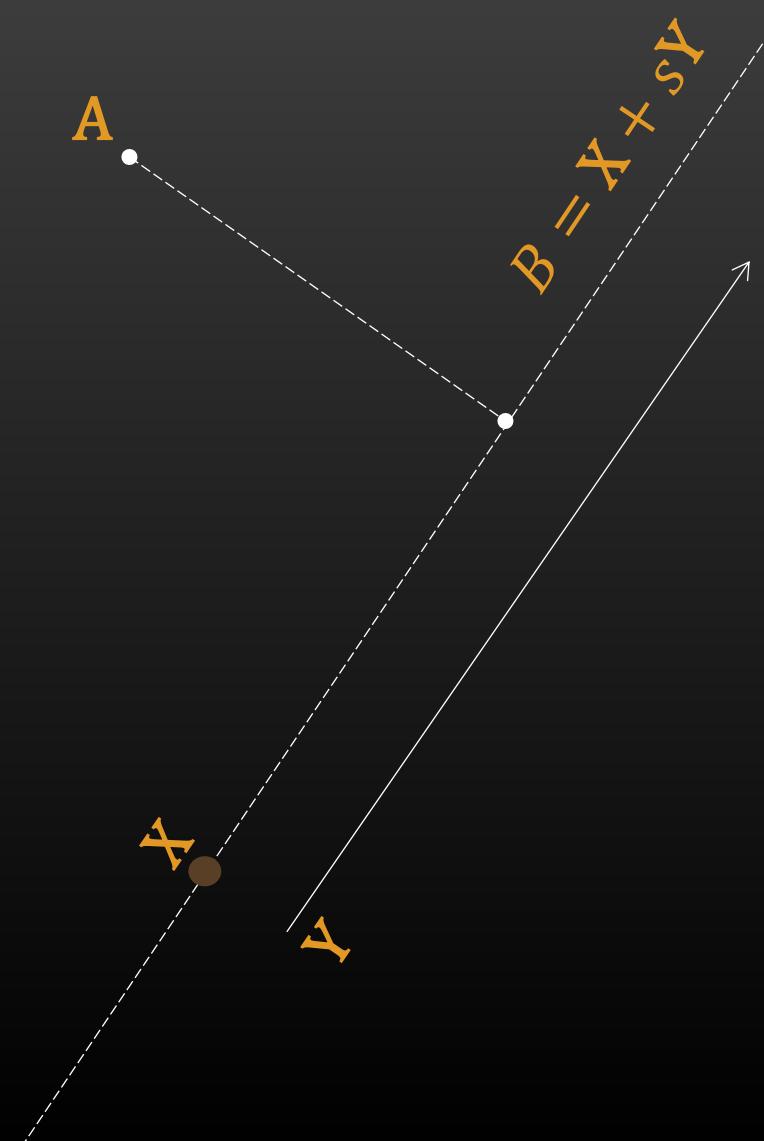
- The next easiest case is testing the intersection of a sphere with an **infinitely long cylinder**
- Again, **reduce intersection to distance**
  - From **center point** of sphere
  - To **center line** of cylinder



# Point/line distance test

Want to measure **distance between point A and line B**

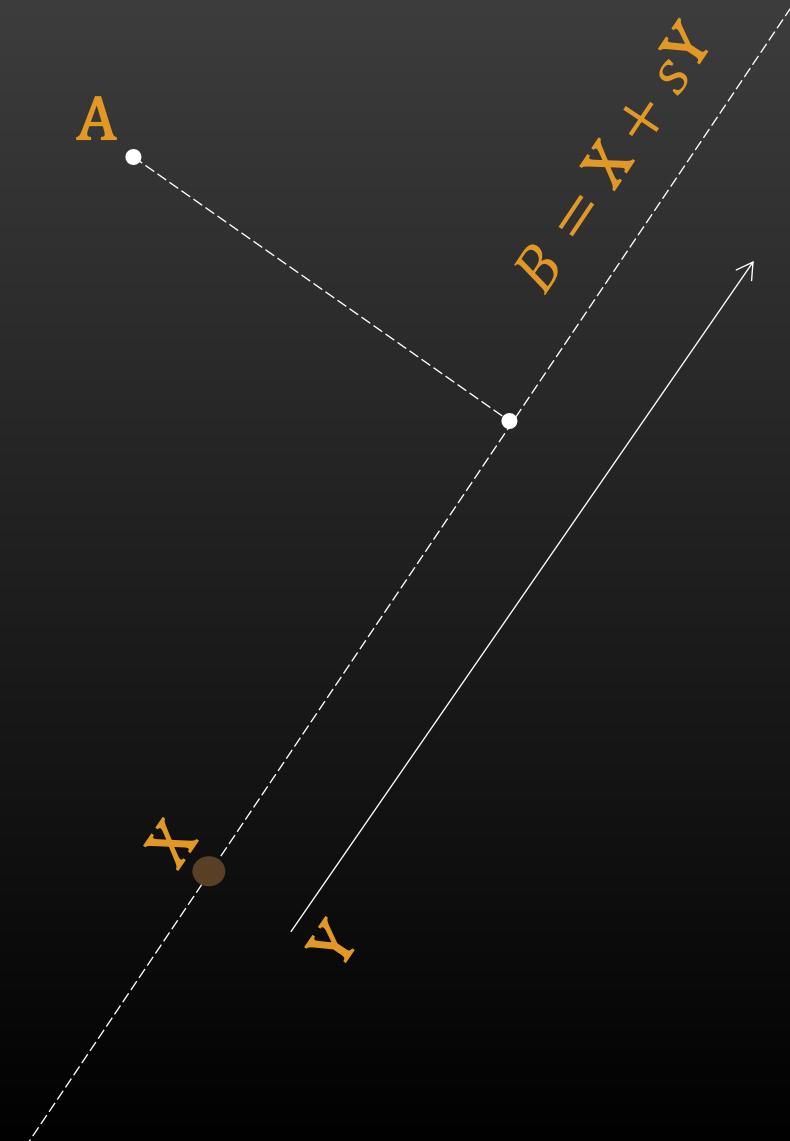
- When B is specified in **parametric** form
  - Point (X)
  - Plus any multiple of a direction (Y)



# Point/line distance test

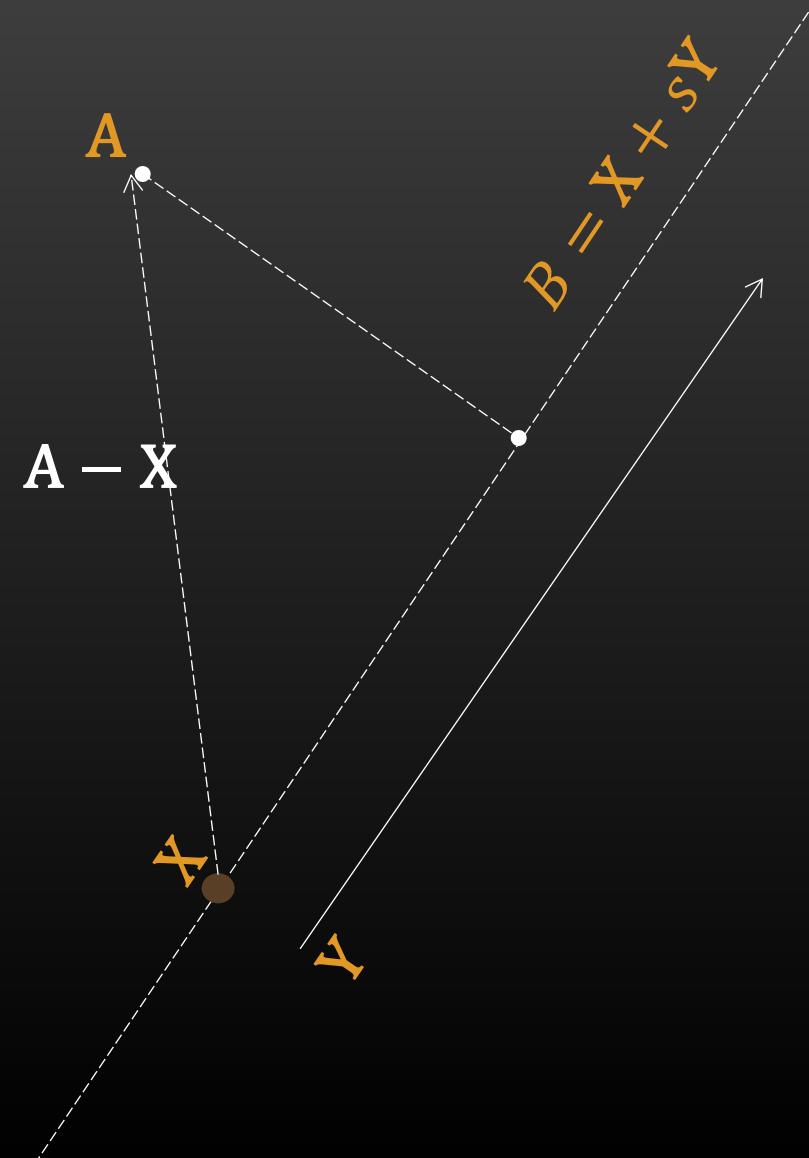
This is essentially the same as asking

- What's the **distance between A and X**,
- **Ignoring** any distance along the **Y axis**



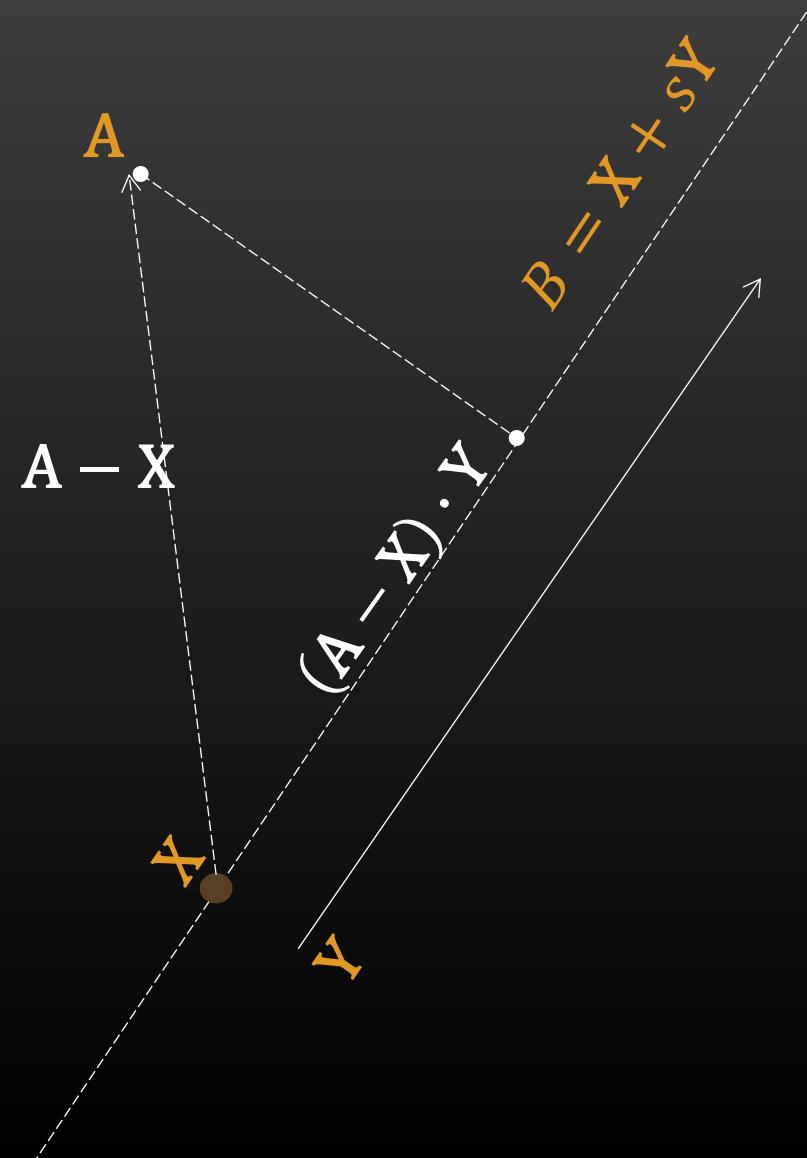
# Point/line distance test

- Can't use distance from A to X
- Because it's got a bunch of **Y offset** we don't care about



# Point/line distance test

- But it's easy to **compute the offset**
  - Its **magnitude** is  $(A - X) \cdot Y$
  - And so the **offset vector** itself is  $((A - X) \cdot Y) Y$

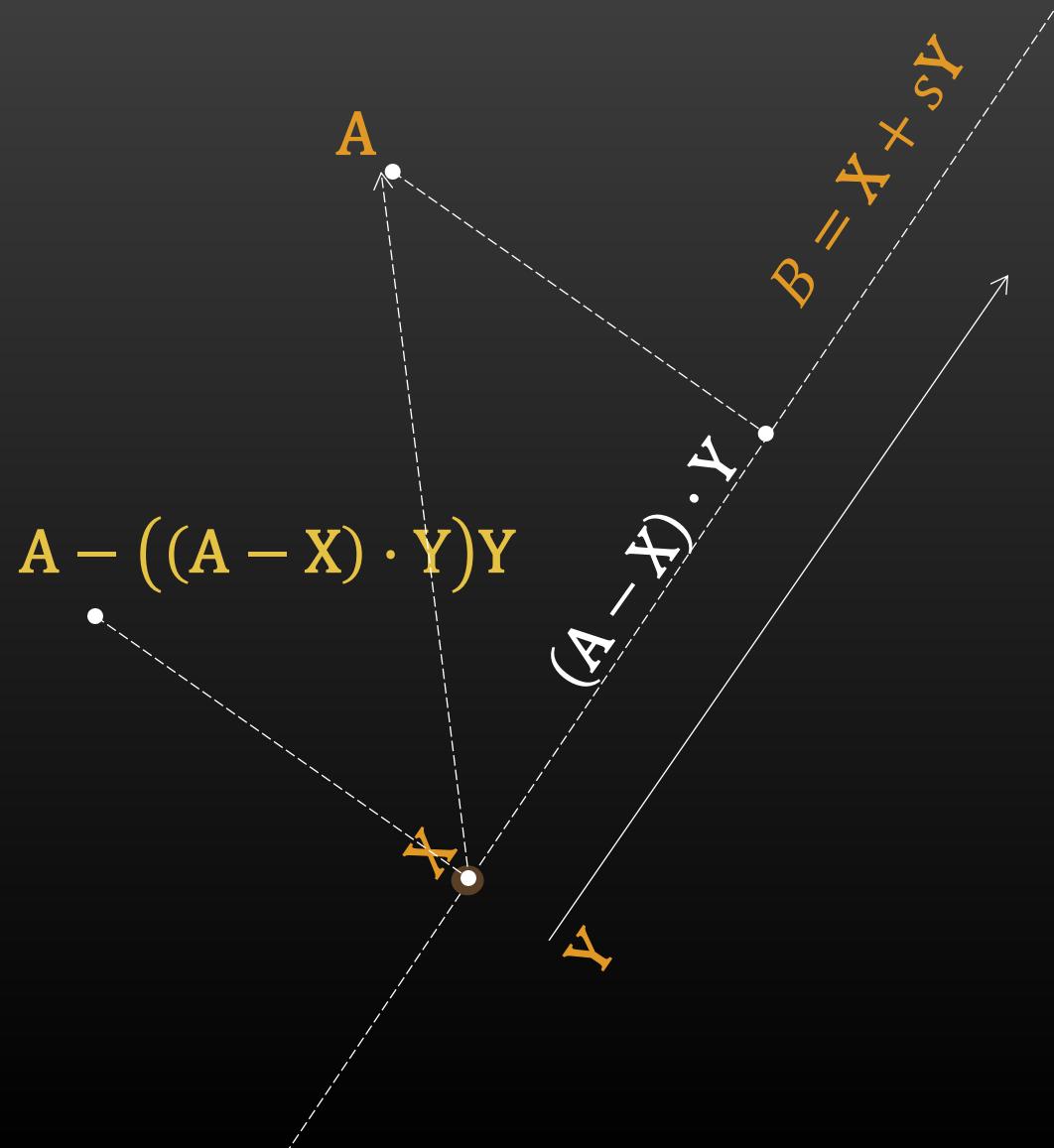


# Point/line distance test

- So if we **shift A** by that offset
- Then we've **removed the Y offset** from A
- And we can just take its distance from X

Or:

$$\|A - ((A - X) \cdot Y)Y - X\|$$



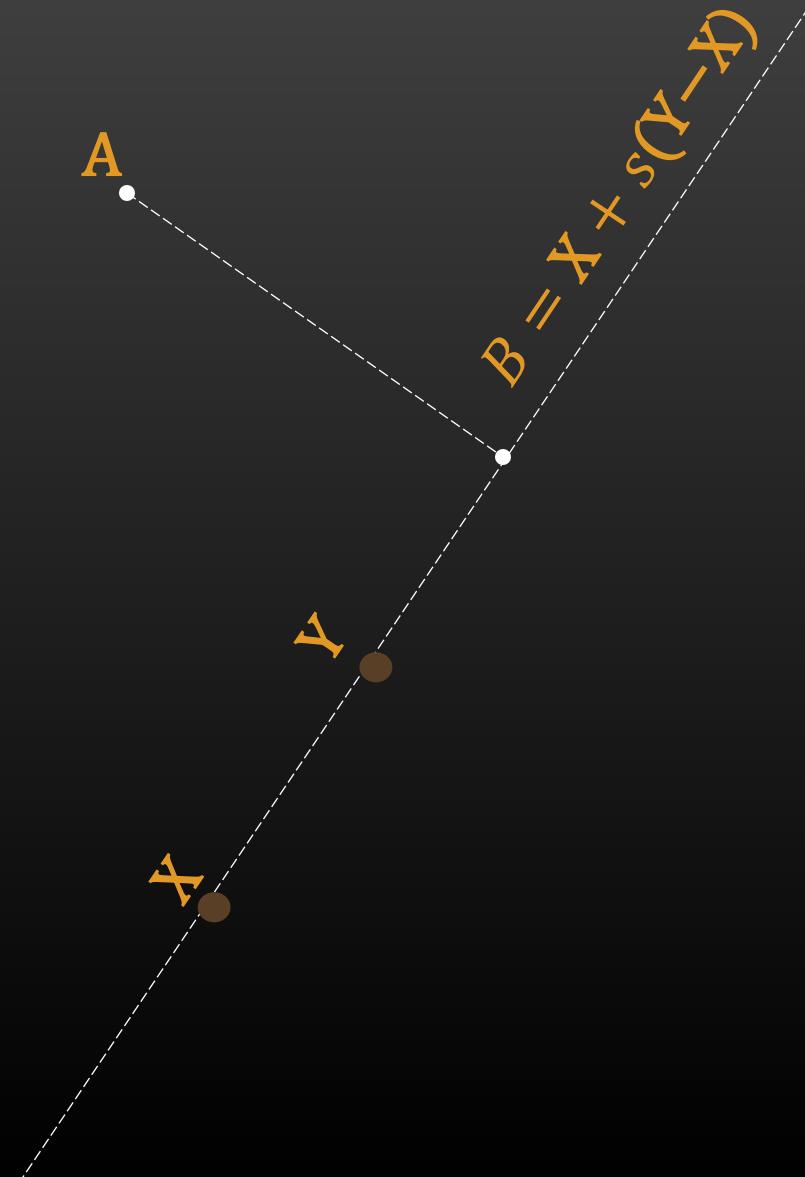
# Alternate parameterization

- It's often easier to **parameterize** the line by **two points** that it goes through (here **X** and **Y**)
- We can then
  - Write **distance** between **A** and a point on the line as a **function of s**
  - Take the **derivative** w.r.t. s
  - Find the s that **minimizes** it
  - Plug it in to get the distance
- When you do that, you get the **minimal distance is at:**

$$s = \frac{(\mathbf{A} - \mathbf{X}) \cdot (\mathbf{A} - \mathbf{Y})}{\|\mathbf{X} - \mathbf{Y}\|^2}$$

$$\text{distance} = \frac{\|(\mathbf{A} - \mathbf{X}) \wedge (\mathbf{A} - \mathbf{Y})\|}{\|\mathbf{X} - \mathbf{Y}\|}$$

(you can also derive this result from the area of the parallelogram XYA)

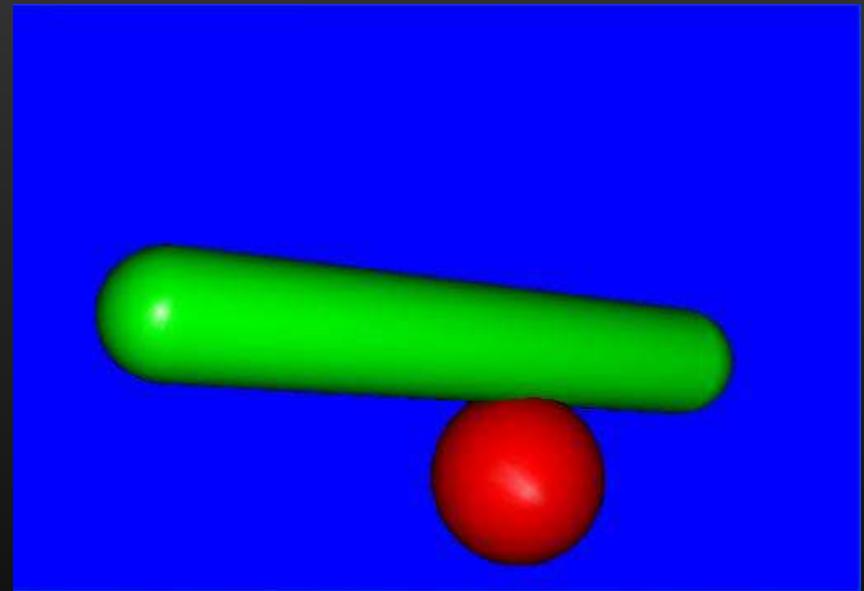


# Sphere-swept volumes

- We can think of an infinite cylinder as
  - All the points you get
  - When you **move a sphere**
  - **Across the line** that forms the spine of the cylinder
- We call that **sweeping** the sphere across the line
- And we call the infinite cylinder a **sphere-swept volume**
- **SSVs** make good collision geometry because
  - You can reduce **intersection tests** on SSVs
  - To **distances tests** on simpler objects that **generate** them (lines, lines, polygons)

# Standard SSVs

- **Sphere**
  - Sphere swept along a single point
- **Capsule**
  - Sphere swept along a line segment
  - Cylinder with hemispherical end caps
- **Lozenge**
  - Sphere swept along a box
  - Box with rounded edges and corners



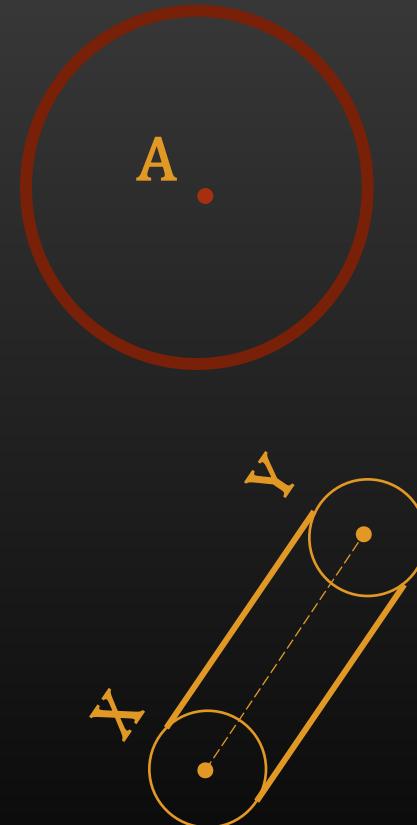
Sphere/capsule  
intersection

# Sphere/capsule intersection

Let:

- A be the **center** of the sphere
- X, Y be the **endpoints** of the spine of the capsule
- **Spine** (line segment) is:  
$$X + s(Y - X)$$

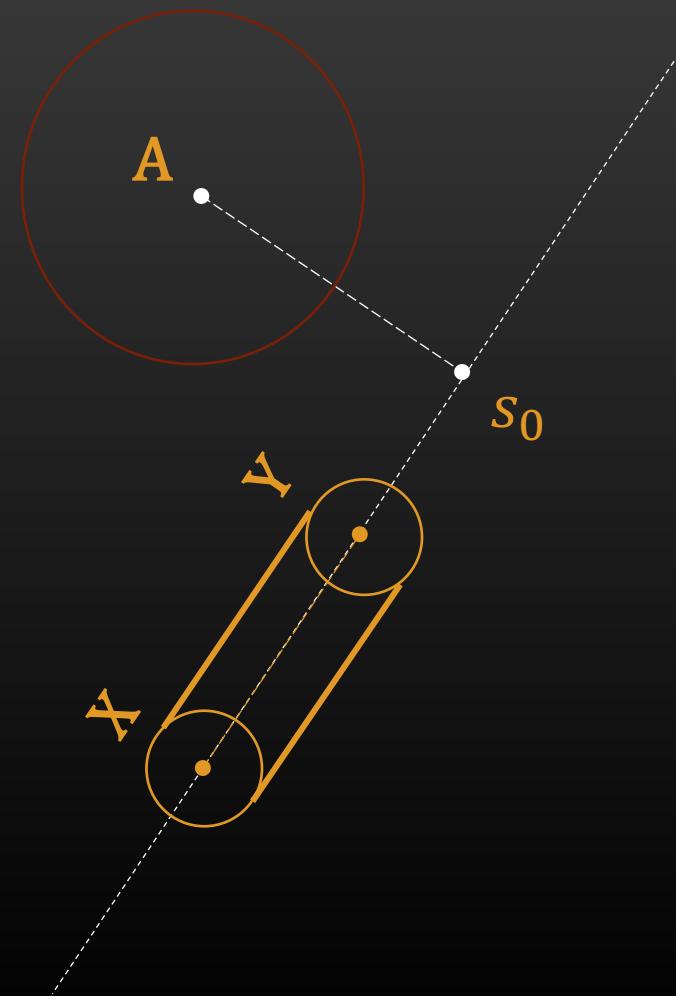
For all  $0 \leq s \leq 1$



# Sphere/capsule intersection

**Closest point** on the line forming the capsule's spine to A is at  $s_0$ :

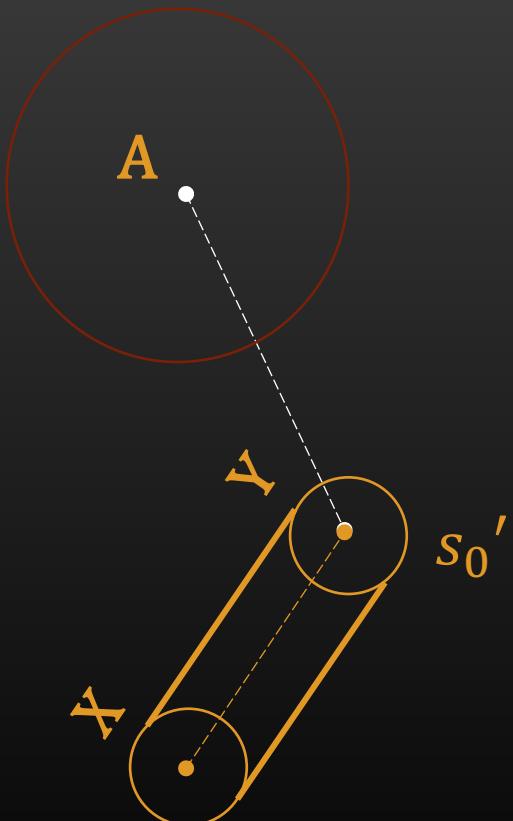
$$s_0 = \frac{(\mathbf{A} - \mathbf{X}) \cdot (\mathbf{A} - \mathbf{Y})}{\|\mathbf{X} - \mathbf{Y}\|^2}$$



# Sphere/capsule intersection

But that point **might be off the spine**,  
so we **limit** it to  $[0,1]$ :

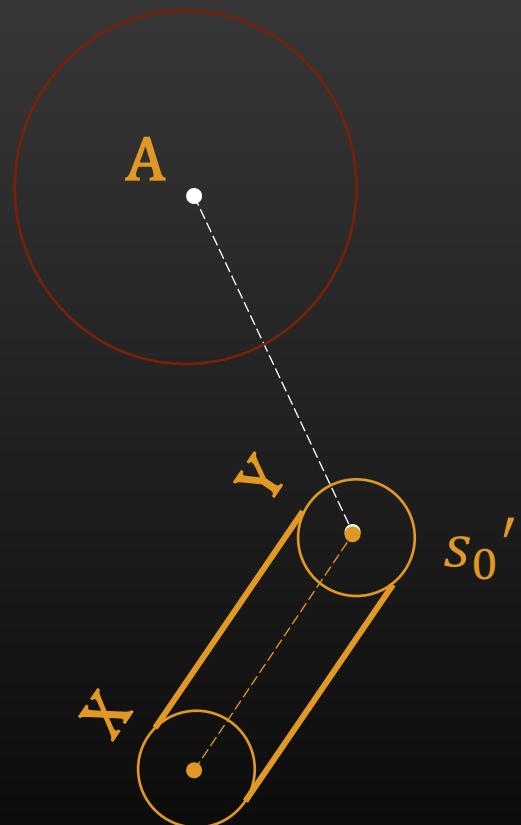
$$s'_0 = \min \left( 1, \max(0, s) \right)$$
$$= \min \left( 1, \max \left( 0, \frac{(\mathbf{A} - \mathbf{X}) \cdot (\mathbf{A} - \mathbf{Y})}{\|\mathbf{X} - \mathbf{Y}\|^2} \right) \right)$$



# Sphere/capsule intersection

Plug that back into the line equation to get the actual **point of closest approach**:

$$X + s_0' Y$$



# Sphere/capsule intersection

And subtract from A to get the **distance**:

$$\|A - (X + s_0' Y)\|$$

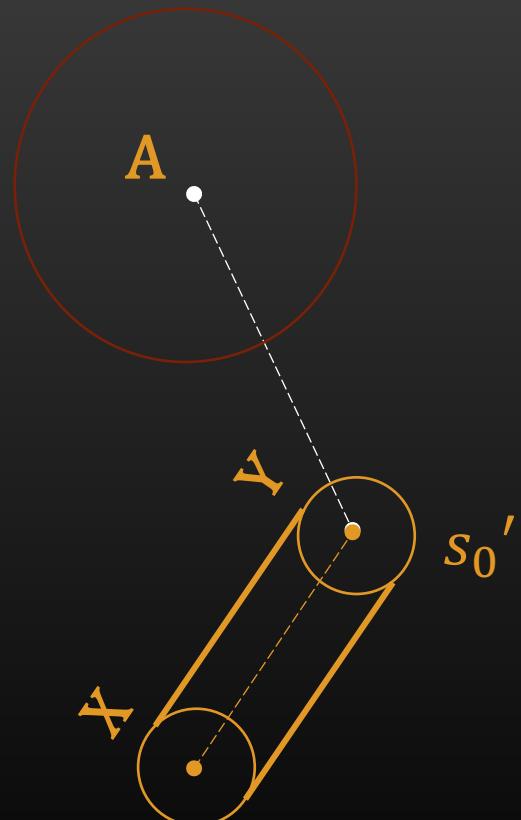
And test it for **intersection**:

$$\|A - (X + s_0' Y)\| < A_r + B_r$$

Where  $A_r, B_r$  are the radii of the sphere and capsule, respectively

And of course, it's faster to **compare squares**:

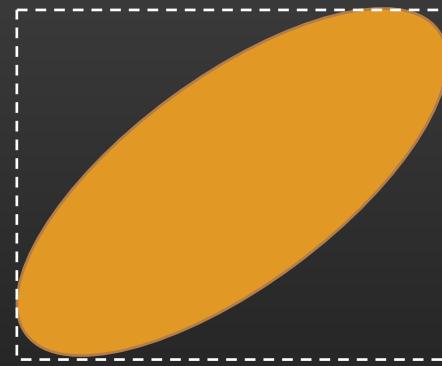
$$\|A - (X + s_0' Y)\|^2 < (A_r + B_r)^2$$



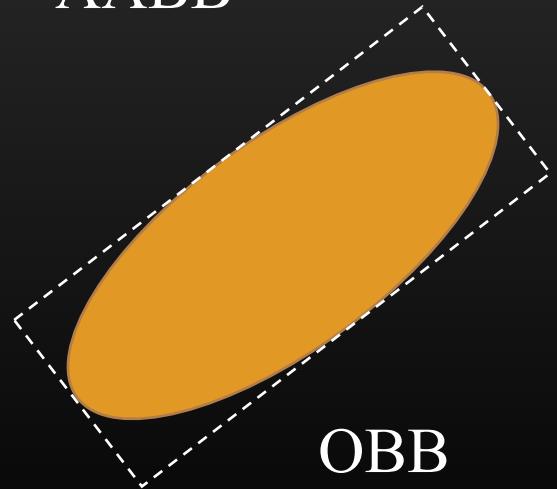
# Bounding boxes

# Bounding boxes

- **Smallest box** (aka 3D rectangle, rectangular prism) that **fully encloses** the object
- Typically either
  - **Axis-aligned** bounding box (AABB)
    - Box aligned with X, Y, Z axes of the coordinate system being used
  - **Oriented** bounding box
    - Box (somehow) aligned with the object's natural axes



AABB

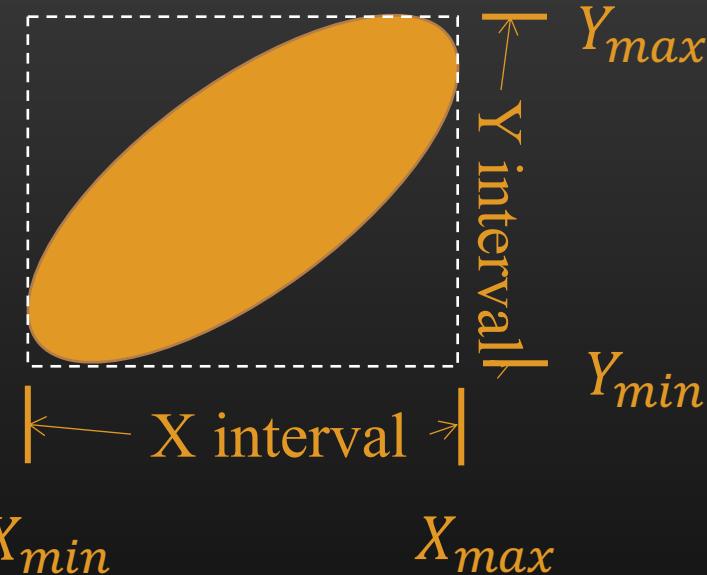


OBB

# Bounding boxes

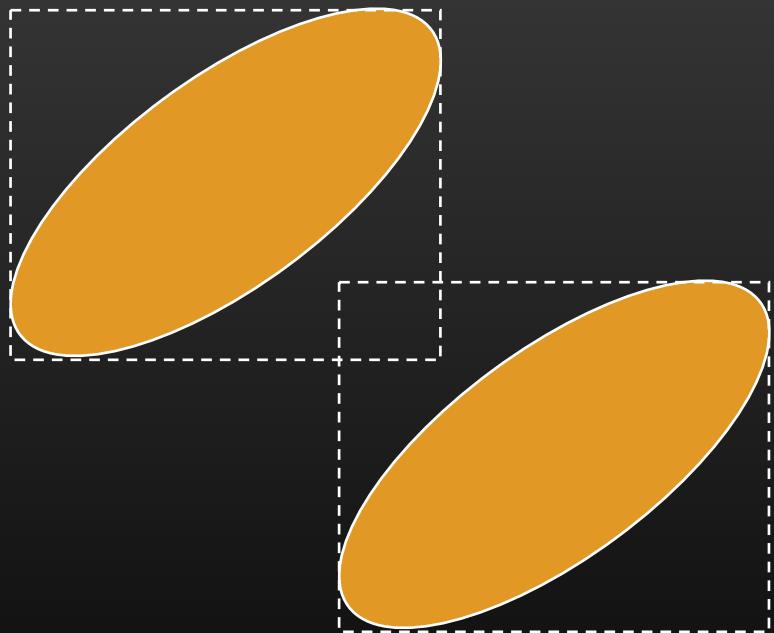
- Bounding boxes can be represented as **intervals** along the axes of the box
- So it's common to represent them in terms of **min** and **max values along each axis**

```
class AABB {  
    Vector3 min, max;  
}
```



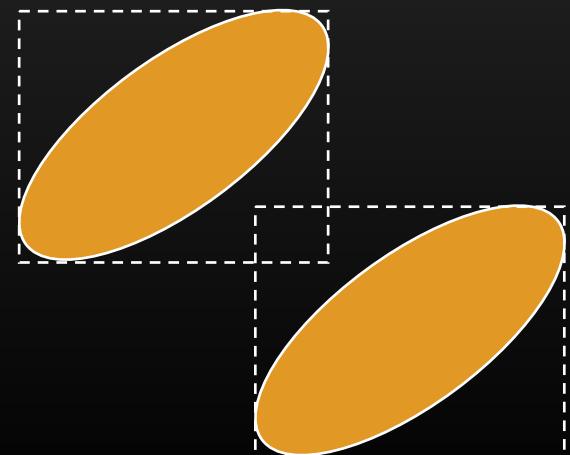
# Intersection test for AABBs

- AABBs are **almost as easy** to intersect as spheres
  - **Separating axis** will be one of the **axes of the box**, if it exists
  - Check **intervals along each axis** for overlap
  - Boxes intersect if **all intervals overlap**
- Intervals A, B disjoint if
  - $A_{min} > B_{max}$ , or
  - $B_{min} > A_{max}$



# Intersection test for AABBs

```
bool Intersects(AABB a, AABB b)
{
    return !(a.min.x > b.max.x || a.max.x < b.min.x
            || a.min.y > b.max.y || a.max.y < b.min.y
            || a.min.z > b.max.z || a.max.z < b.min.z);
}
```



Hard stuff

# Intersecting convex polyhedra

## Separating axis theorem

- If two **convex shapes** are non-intersecting
- Then there must be **some axis** along which they're disjoint

## Special case

- If the shapes are **polygons/polyhedra**,
- The **normals** of at least one of its faces must be a separating axis

## Bad algorithm

```
Intersect(A, B) {  
    for each face f of A {  
        Let n be f's normal  
        Let  $n_o = f \cdot n$   
        for each vertex v of B {  
            if ( $v \cdot n < n_o$ )  
                // v is behind f  
                // so n isn't a separating axis  
                Skip to next face of A  
        }  
        return false  
    }  
    ... do it again with B's faces ...  
    return true  
}
```

## Better algorithm: GJK

- But way too complicated for today

# But, wait! There are more headaches!

## Contact point determination

- Physics doesn't just want to know that two objects hit
- It needs to know
  - **Where** they first hit
  - **When** they first hit
- This is a whole other world of hurt

## Static vs. dynamic collision detection

- Games usually simulate in **discrete frames**
- What if an object **moves so fast** that it goes
  - From in front of an object
  - To behind it
  - In one frame?
- It **should collide** with the object, but
  - They don't intersect in the first frame
  - And they don't intersect in the second frame either
- Instead, it **teleports** through the other object
- Handling this requires much hairier processing called **dynamic** or **continuous collision detection**

# Broad-phase collision detection

# Broad-phase detection

- Intersection testing is expensive
- Testing **every pair** of objects is  $O(n^2)$ 
  - And  $n$  may be big
- Can easily eat the whole CPU
- Want to
  - **Rule out** most pairs
  - Without explicitly testing them for intersection
- So we usually divide collision detection into
  - **Broad-phase**: prune the space of pairs to test
  - **Narrow-phase**: detailed intersection testing and contact point determination

# Collision layers

- One simple thing to do is to **break game objects into groups** ("layers")
- Specify in advance that **certain groups don't collide** with other groups
- Unity lets you do this
  - Set object's layer in the **collider**
  - Set what layers collide with what others in the **project settings**

		Default	TransparentFX	Ignore Raycast	Water	Layer1	Layer2	Layer3
		Default	✓	✓	✓	✓	✓	✓
Default	Default	✓	✓	✓	✓	✓	✓	✓
	TransparentFX	✓	✓	✓	✓	✓	✓	✓
TransparentFX	Default	✓	✓	✓	✓	✓	✓	✓
	TransparentFX	✓	✓	✓	✓	✓	✓	✓
Ignore Raycast	Default	✓	✓	✓	✓	✓	✓	✓
	TransparentFX	✓	✓	✓	✓	✓	✓	✓
Ignore Raycast	Default	✓	✓	✓	✓	✓	✓	✓
	TransparentFX	✓	✓	✓	✓	✓	✓	✓
Water	Default	✓	✓	✓	✓	✓	✓	✓
	TransparentFX	✓	✓	✓	✓	✓	✓	✓
Water	Default	✓	✓	✓	✓	✓	✓	✓
	TransparentFX	✓	✓	✓	✓	✓	✓	✓
Layer1	Default	✓	✓	✓	✓	✓	✓	✓
	TransparentFX	✓	✓	✓	✓	✓	✓	✓
Layer1	Default	✓	✓	✓	✓	✓	✓	✓
	TransparentFX	✓	✓	✓	✓	✓	✓	✓
Layer2	Default	✓	✓	✓	✓	✓	✓	✓
	TransparentFX	✓	✓	✓	✓	✓	✓	✓
Layer2	Default	✓	✓	✓	✓	✓	✓	✓
	TransparentFX	✓	✓	✓	✓	✓	✓	✓
Layer3	Default	✓	✓	✓	✓	✓	✓	✓
	TransparentFX	✓	✓	✓	✓	✓	✓	✓

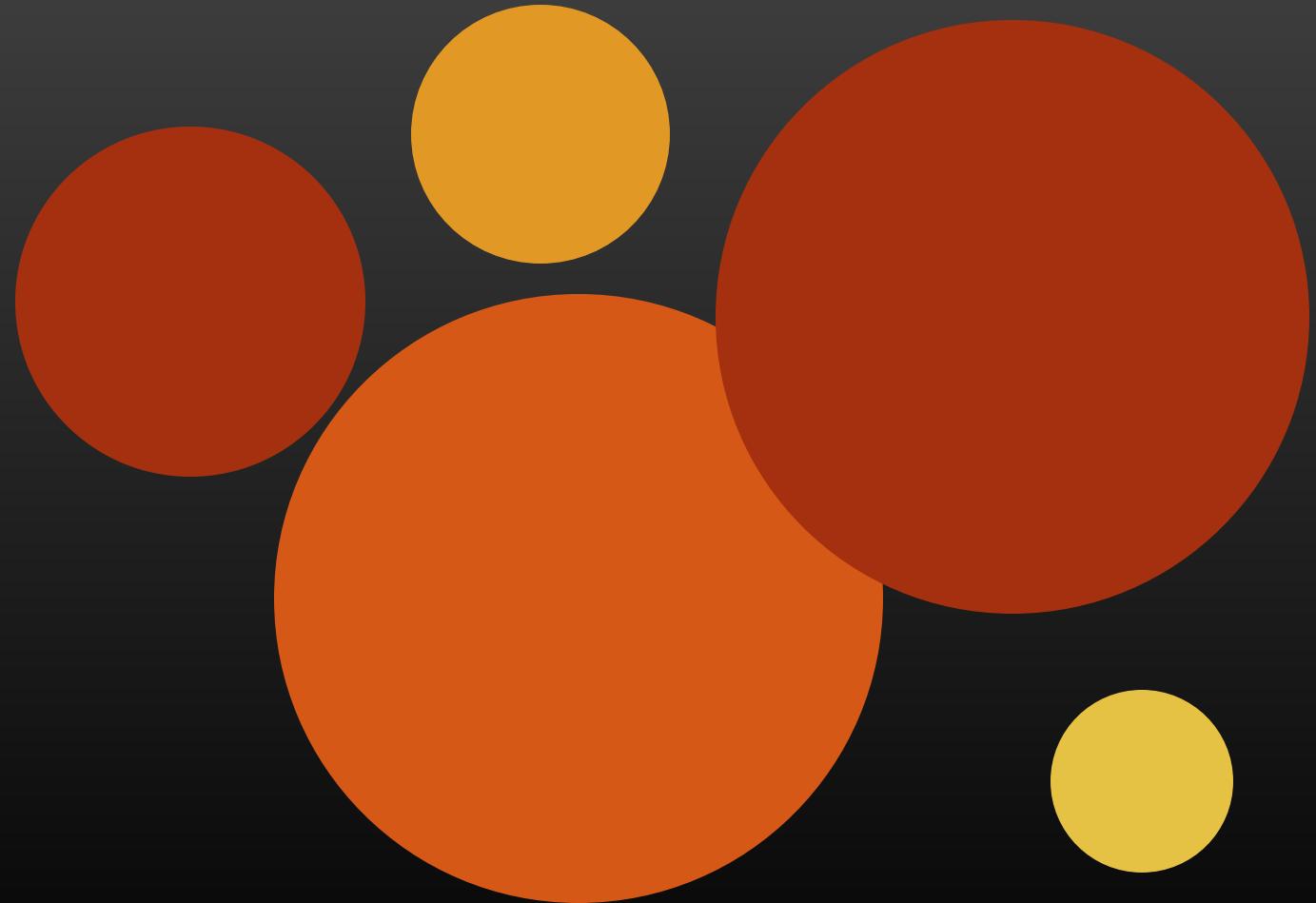
# Broad-phase algorithms

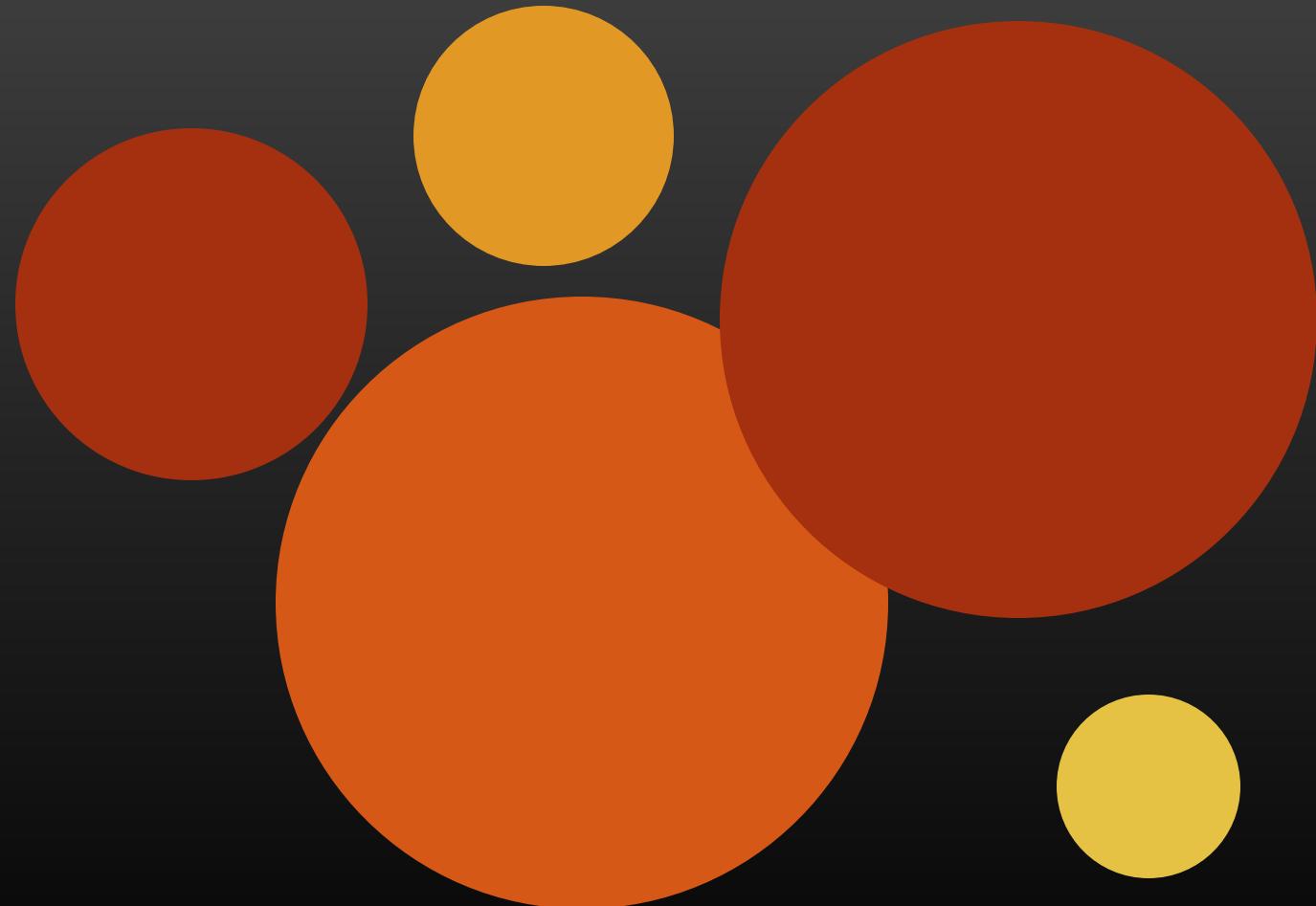
## Spatial search trees

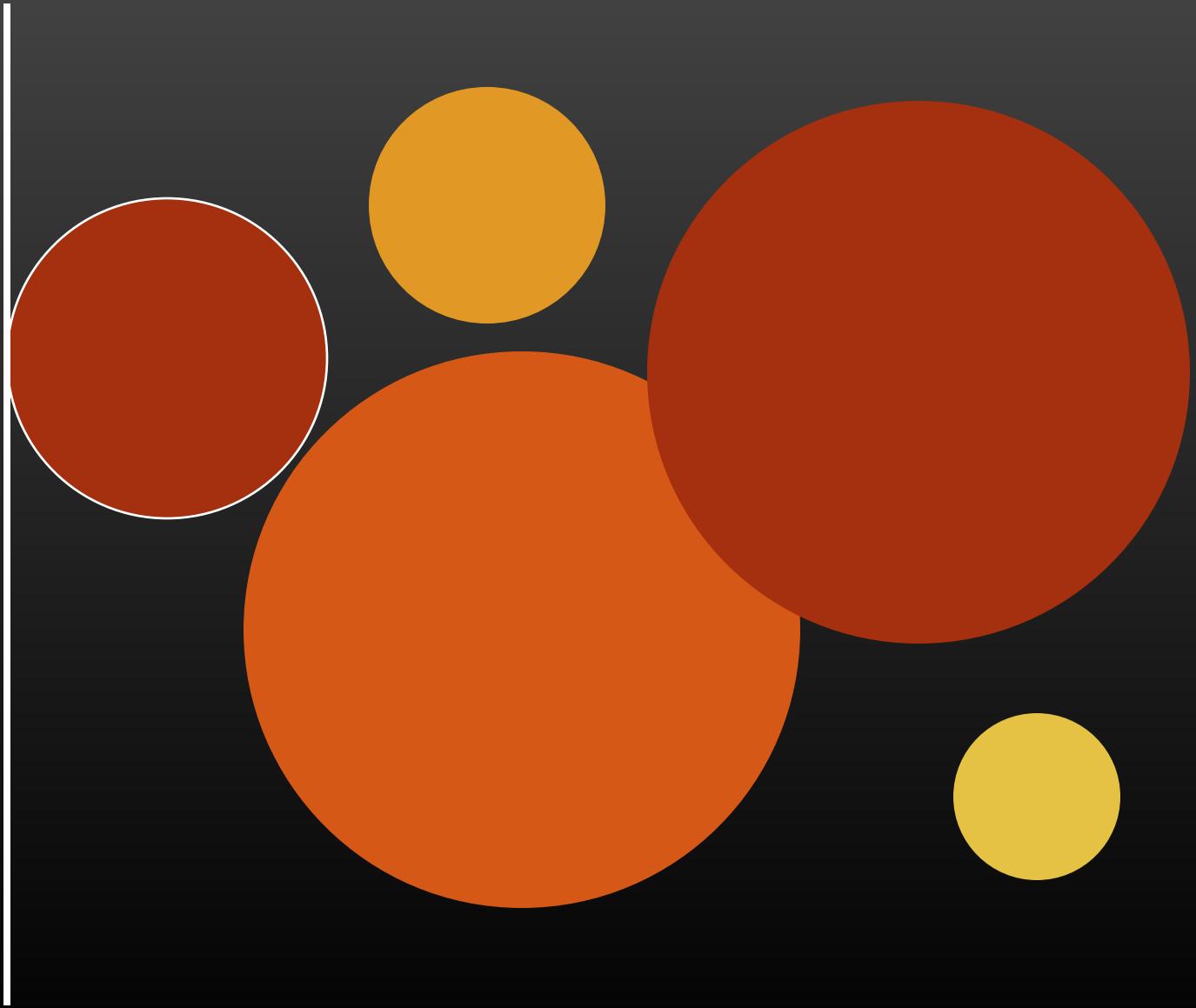
- **BSP trees, kd-trees**
  - Each tree node divides space into two halves
- **Quadtrees, octtrees**
  - Each node divides space into 4 or 8 cells
- **Bounding volume hierarchies**
  - Arranges space as a hierarchy of e.g. AABBs within AABBs
- **$O(n \log n)$**  if the trees are relatively balanced

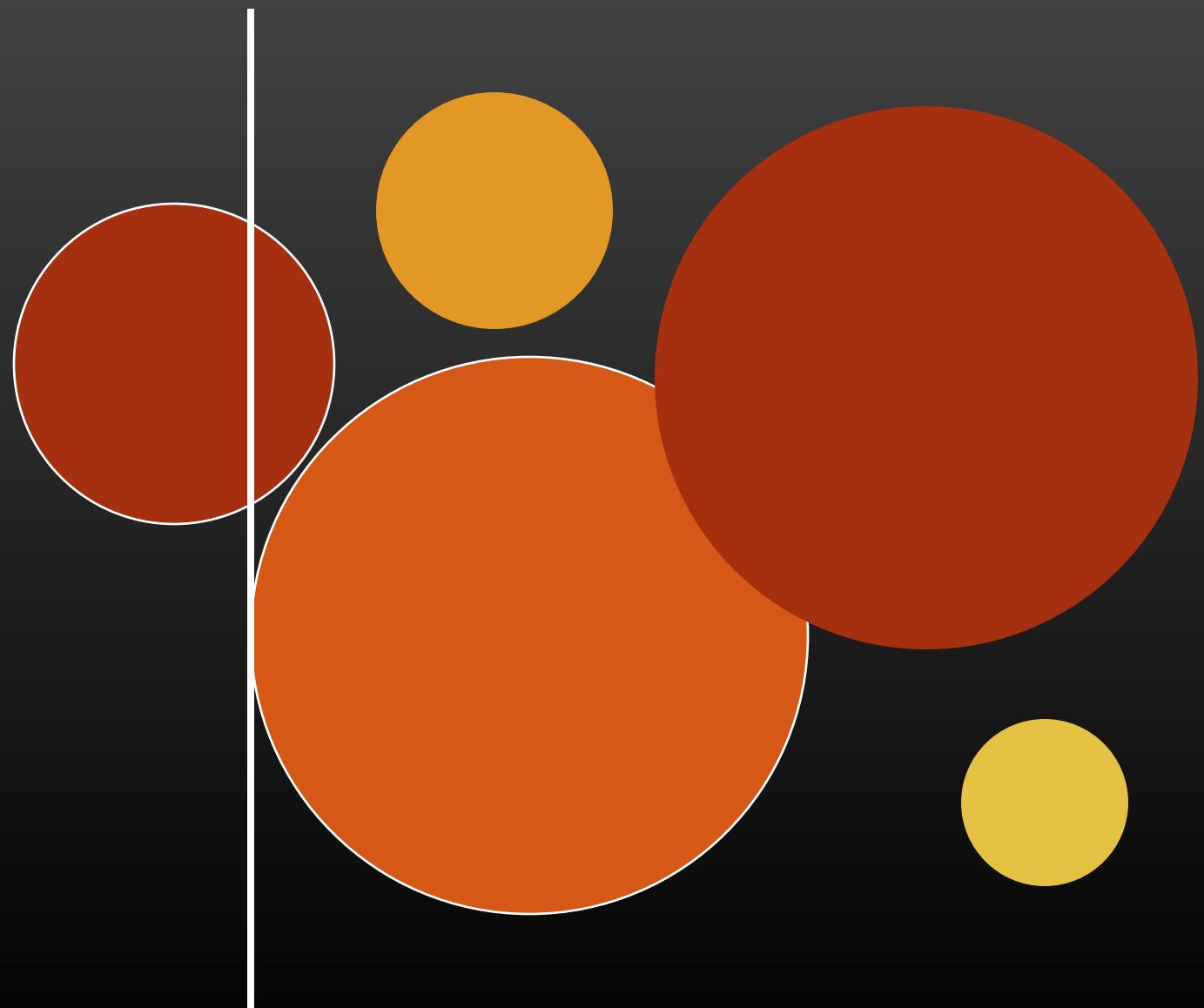
## Sweep and prune

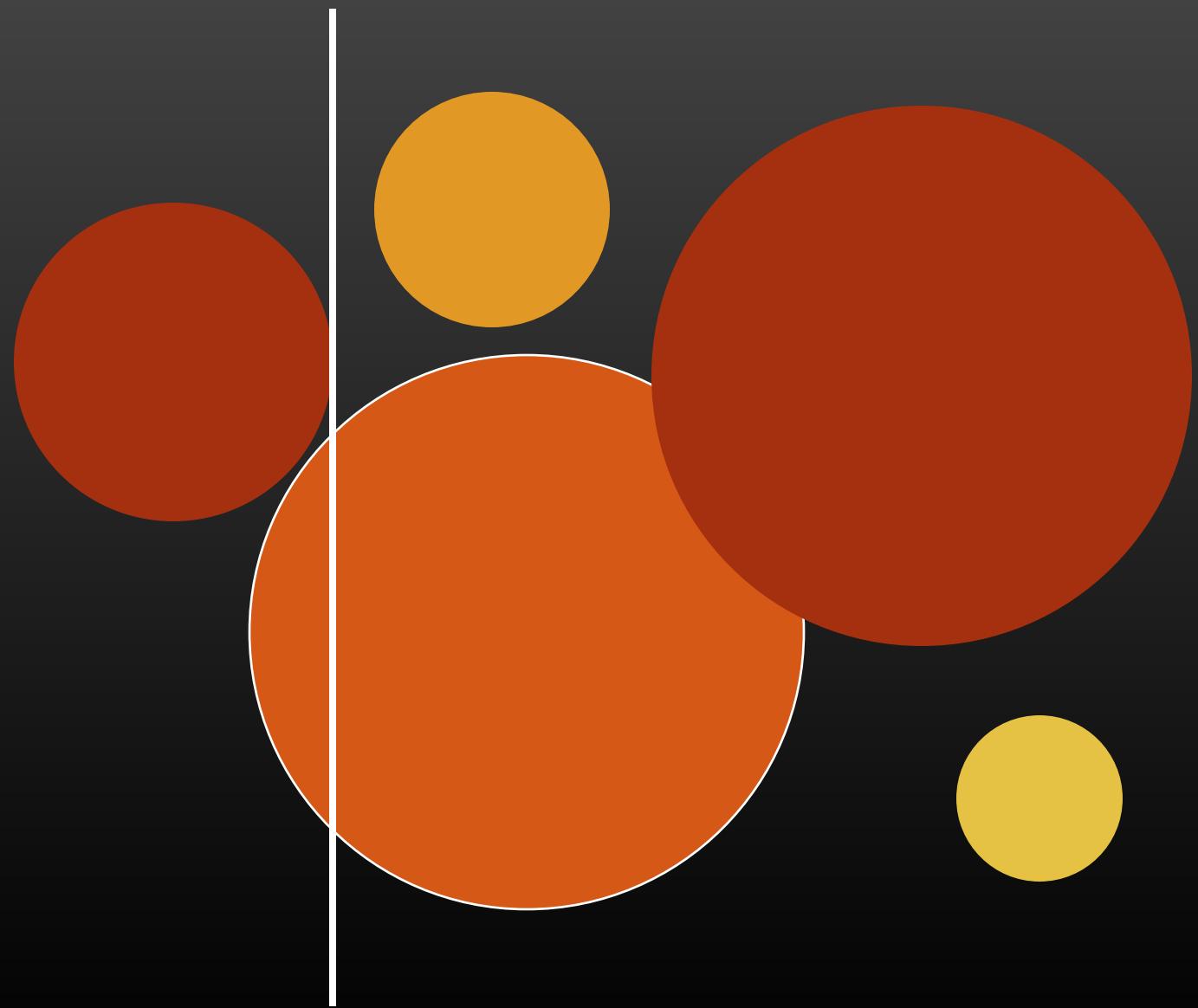
- **Sort** all objects along **one axis**
  - $O(n \log n)$
- **Sweep** an imaginary plane from one end of the axis to the other
  - **Track objects intersecting the plane** as it sweeps
  - $O(n)$
- Each time you encounter a **new object, test it** against all objects currently intersecting the plane
  - $O(n)$  worst case for each object
- So still  $O(n^2)$  in the worst case, but hopefully much faster in the **average case**

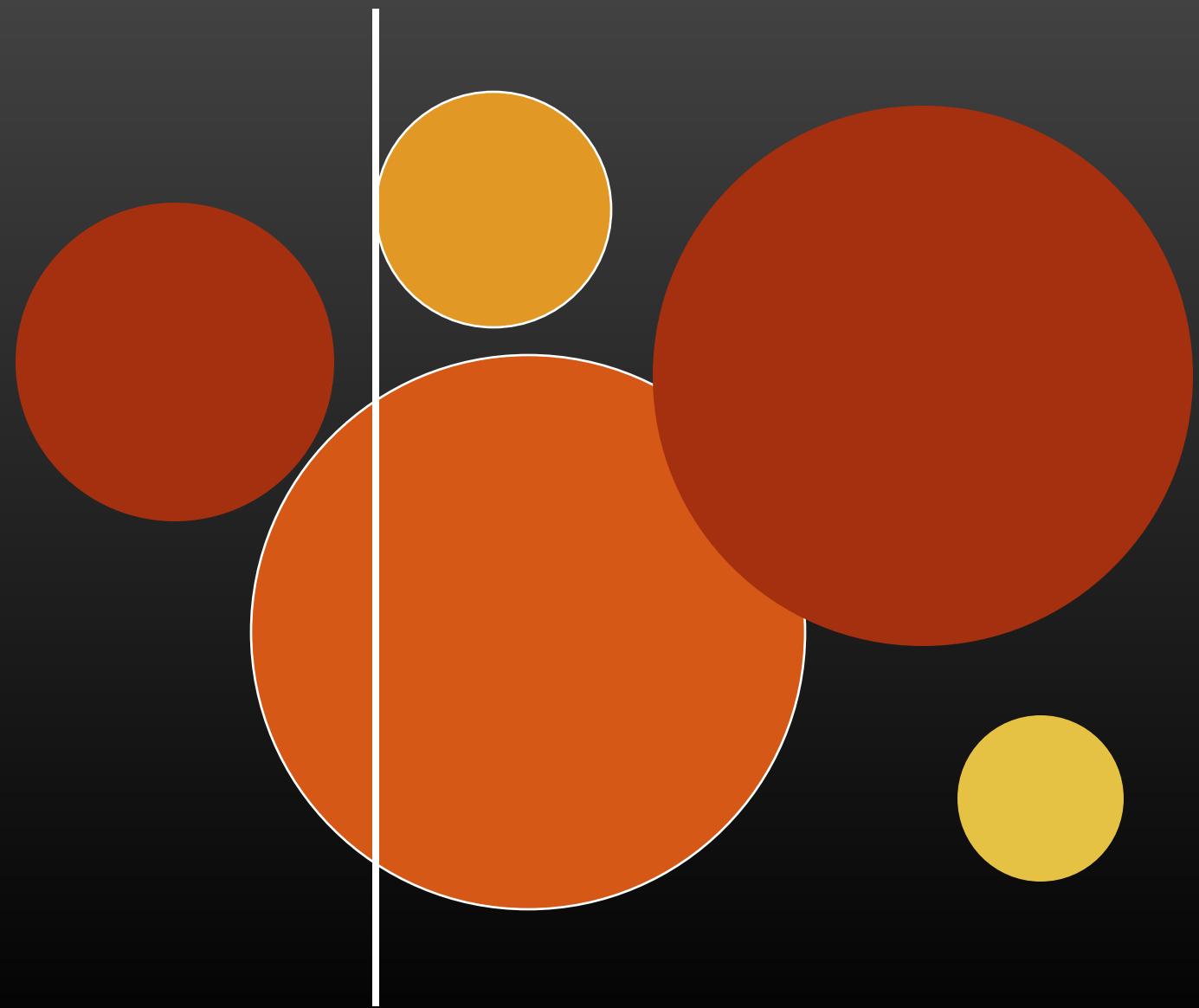


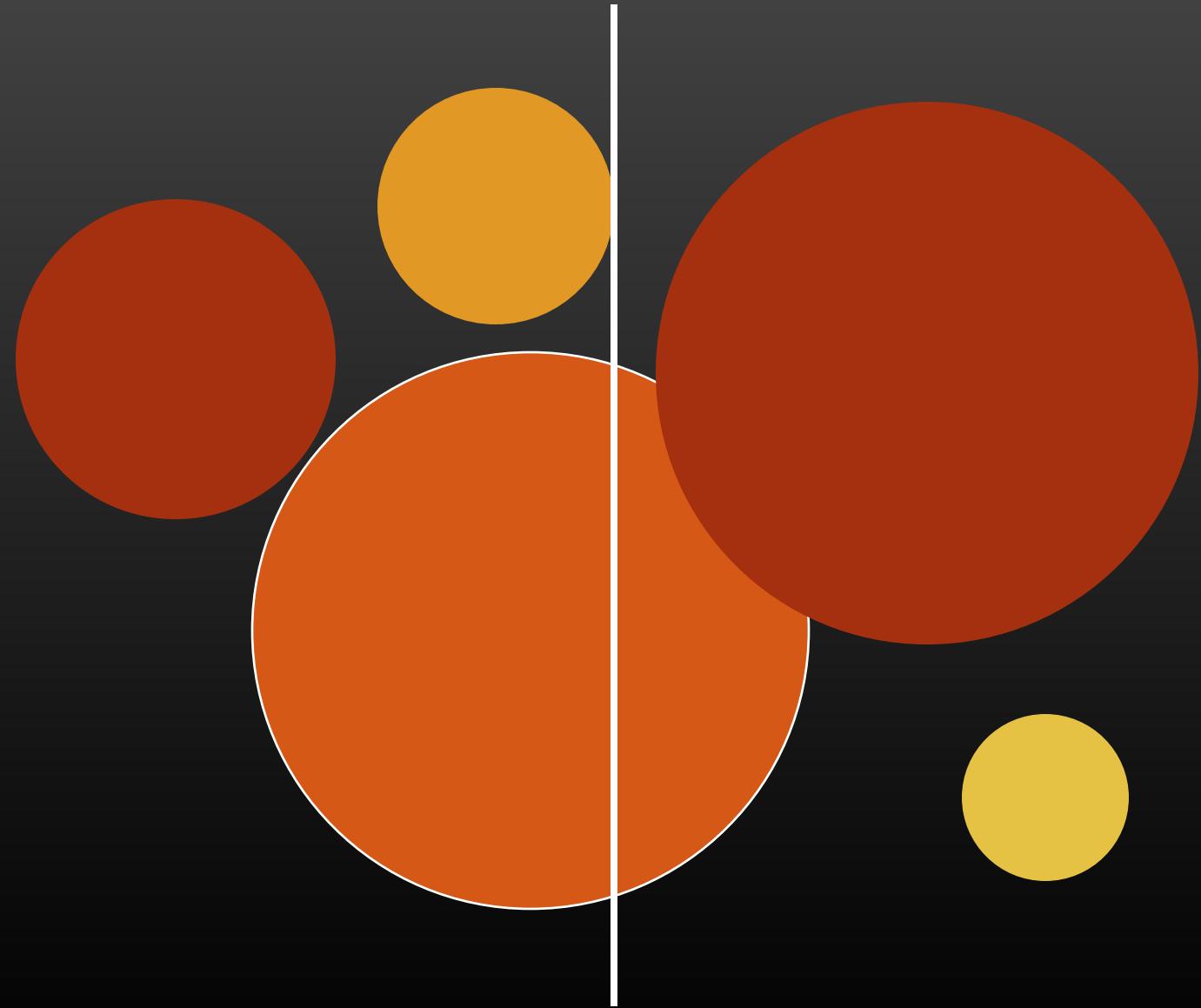


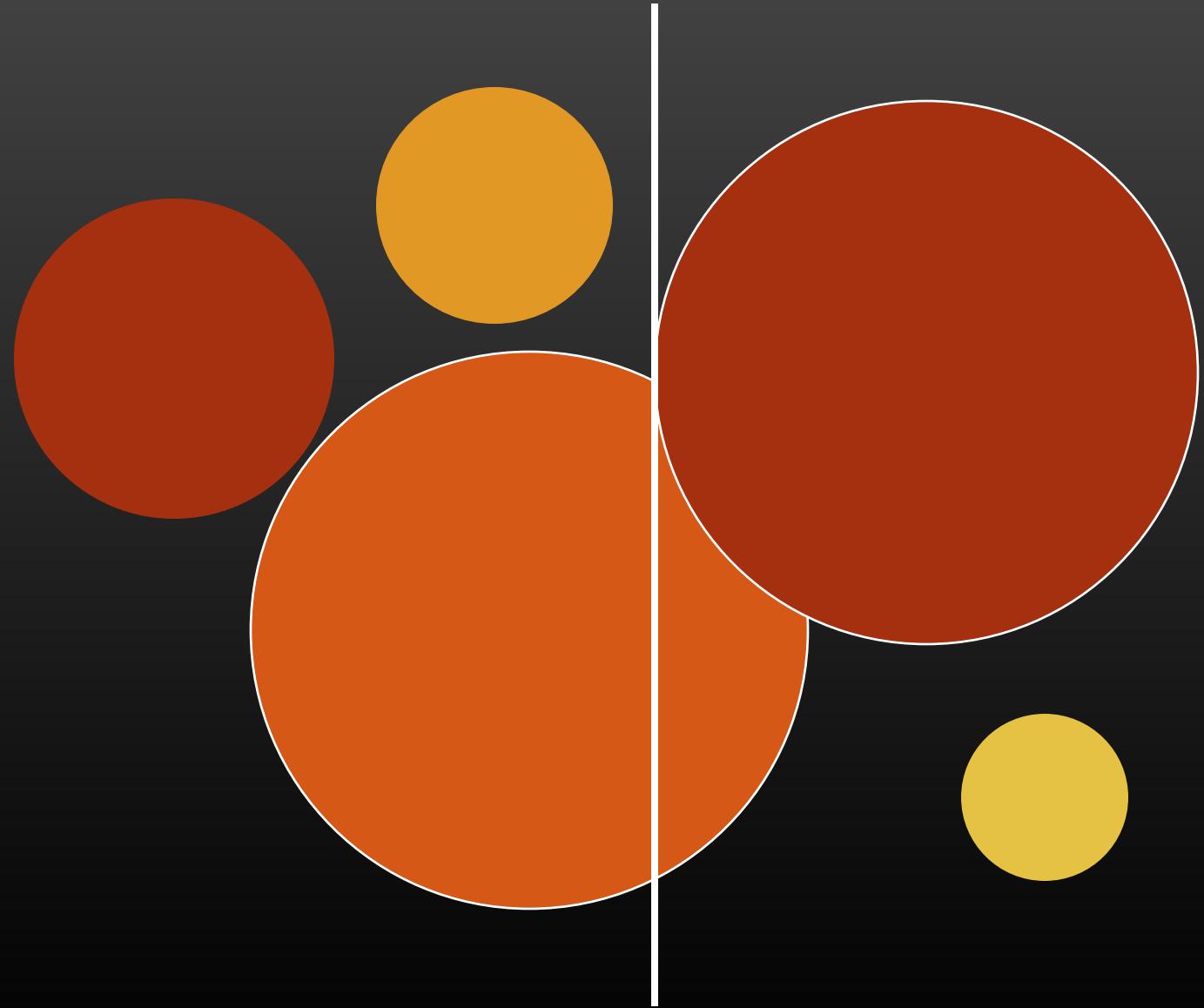




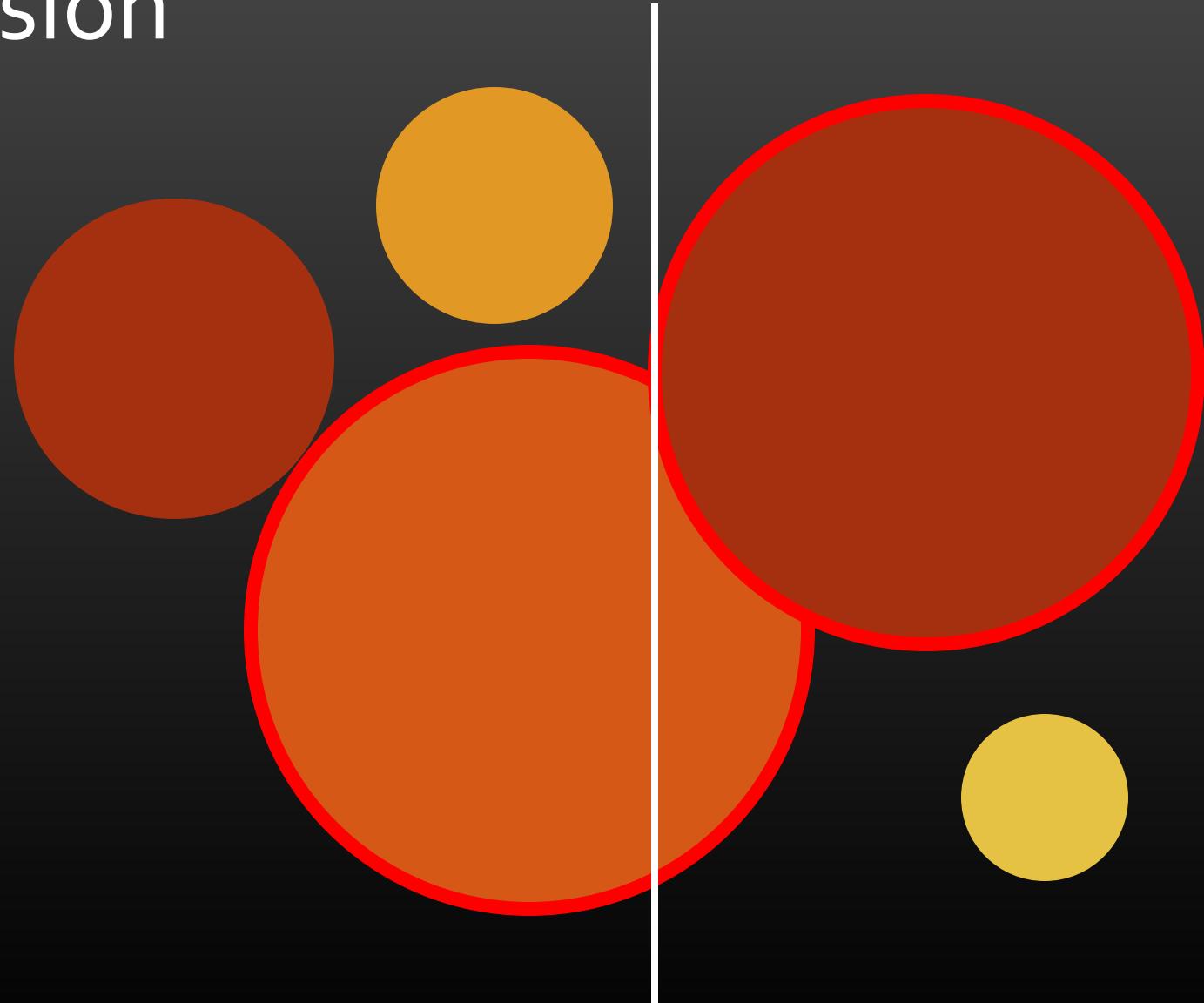


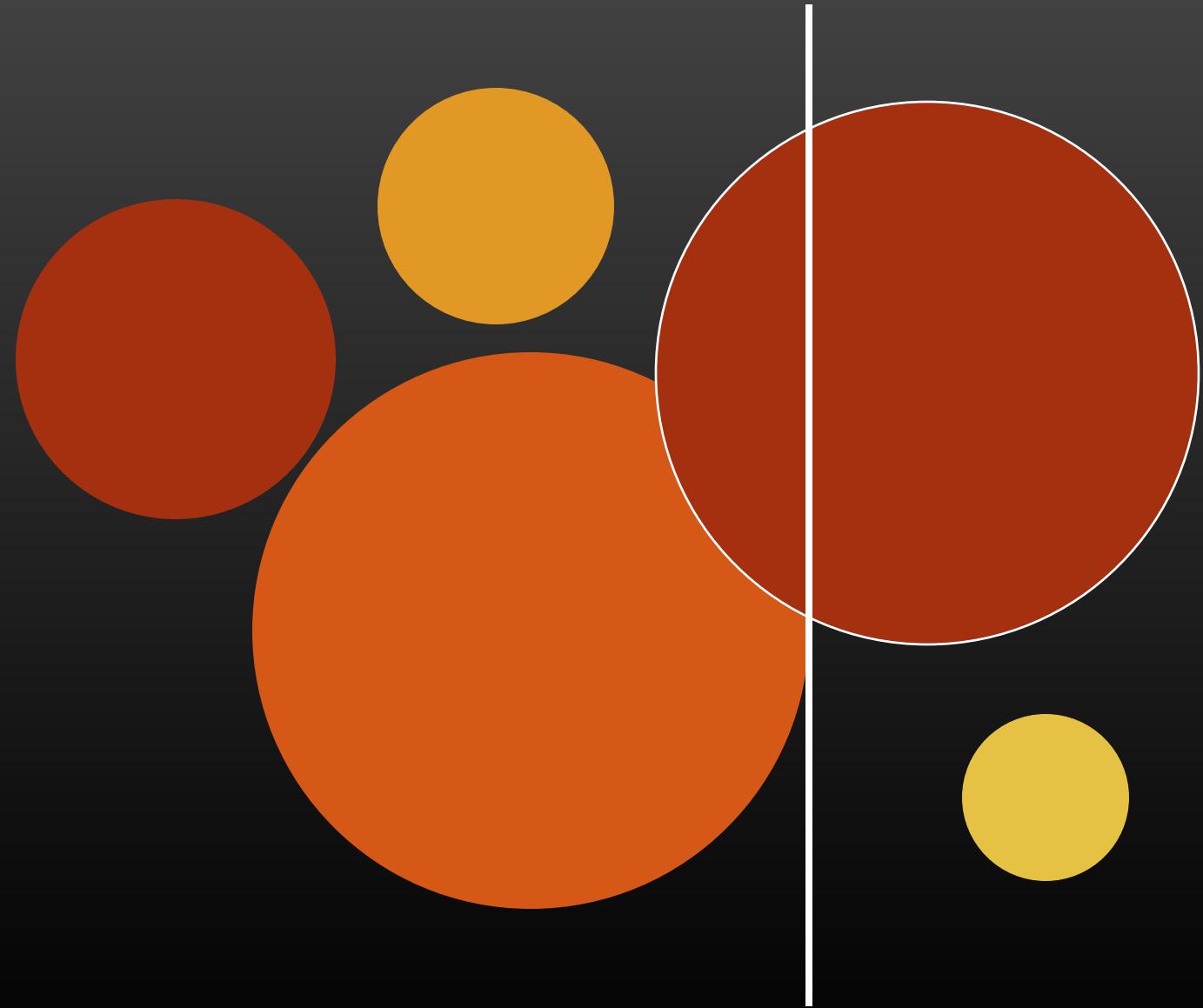


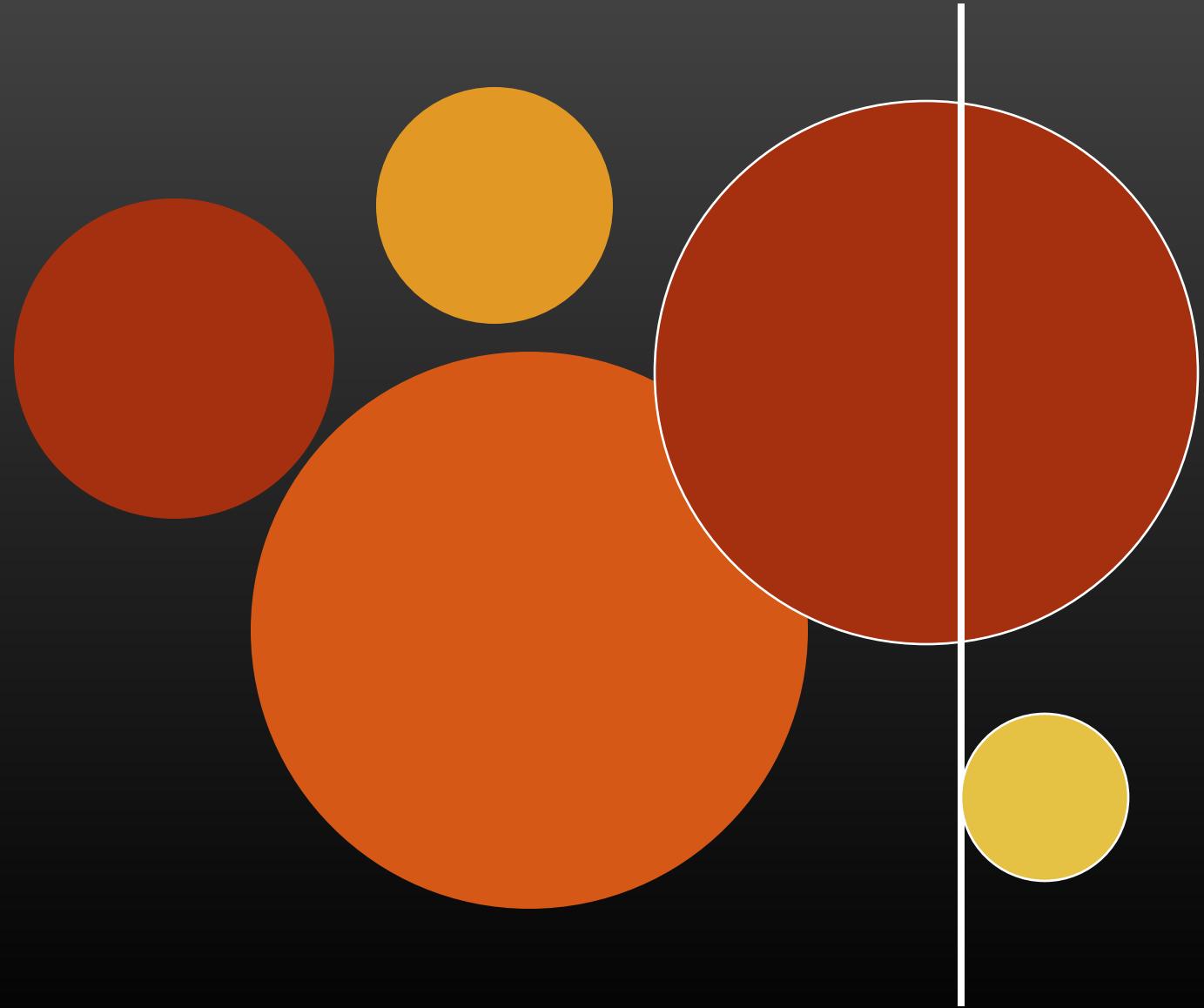


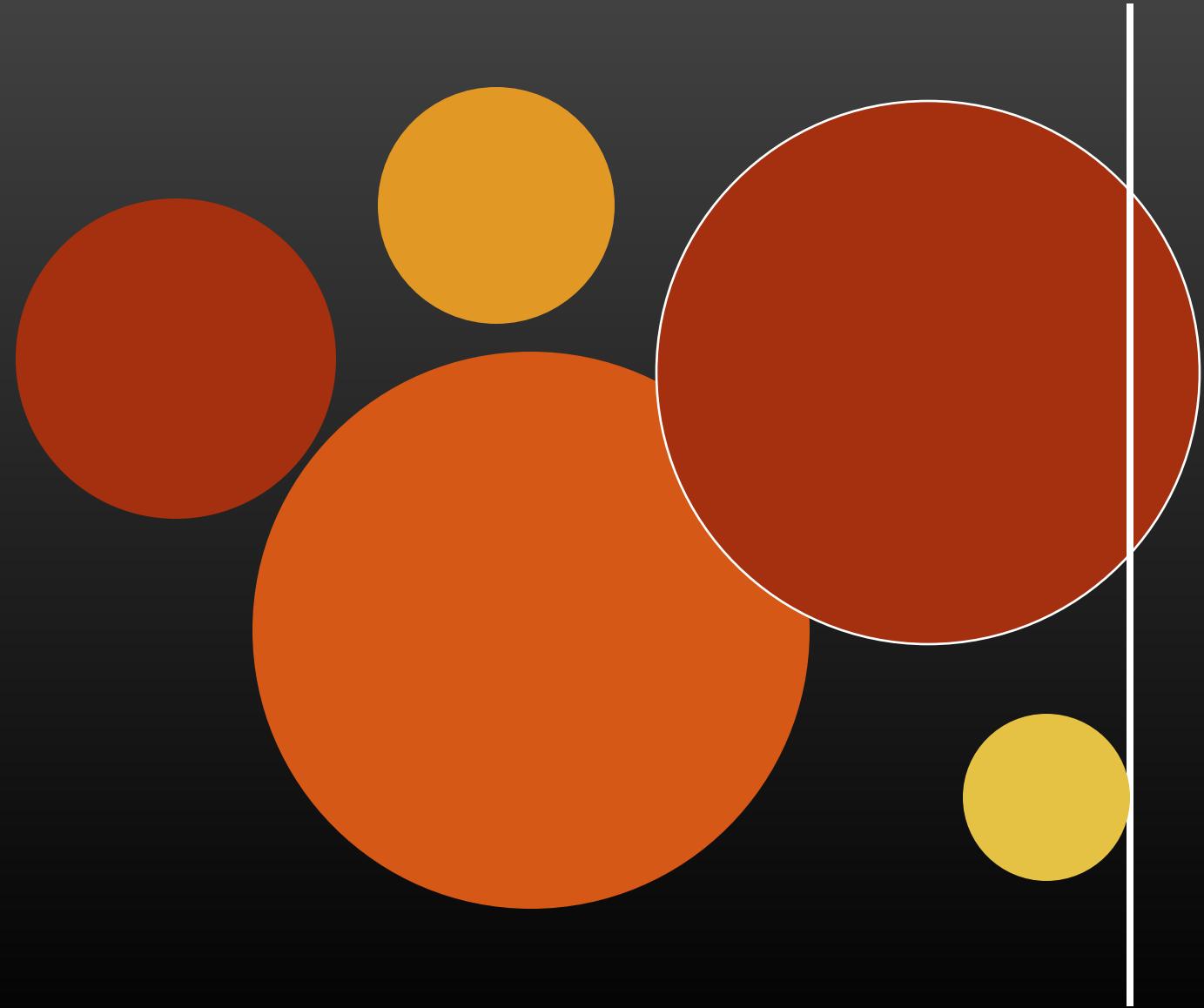


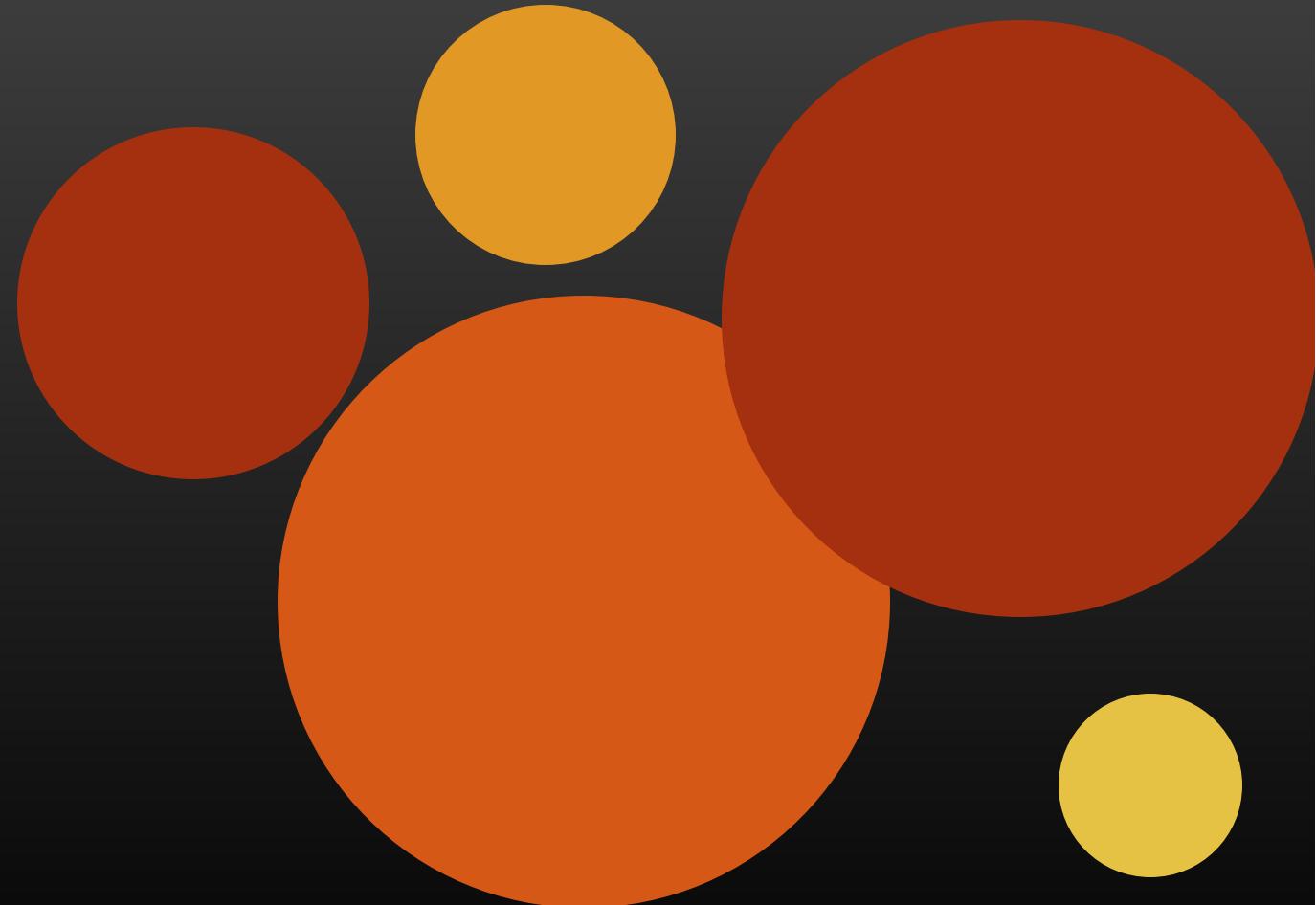
# Collision



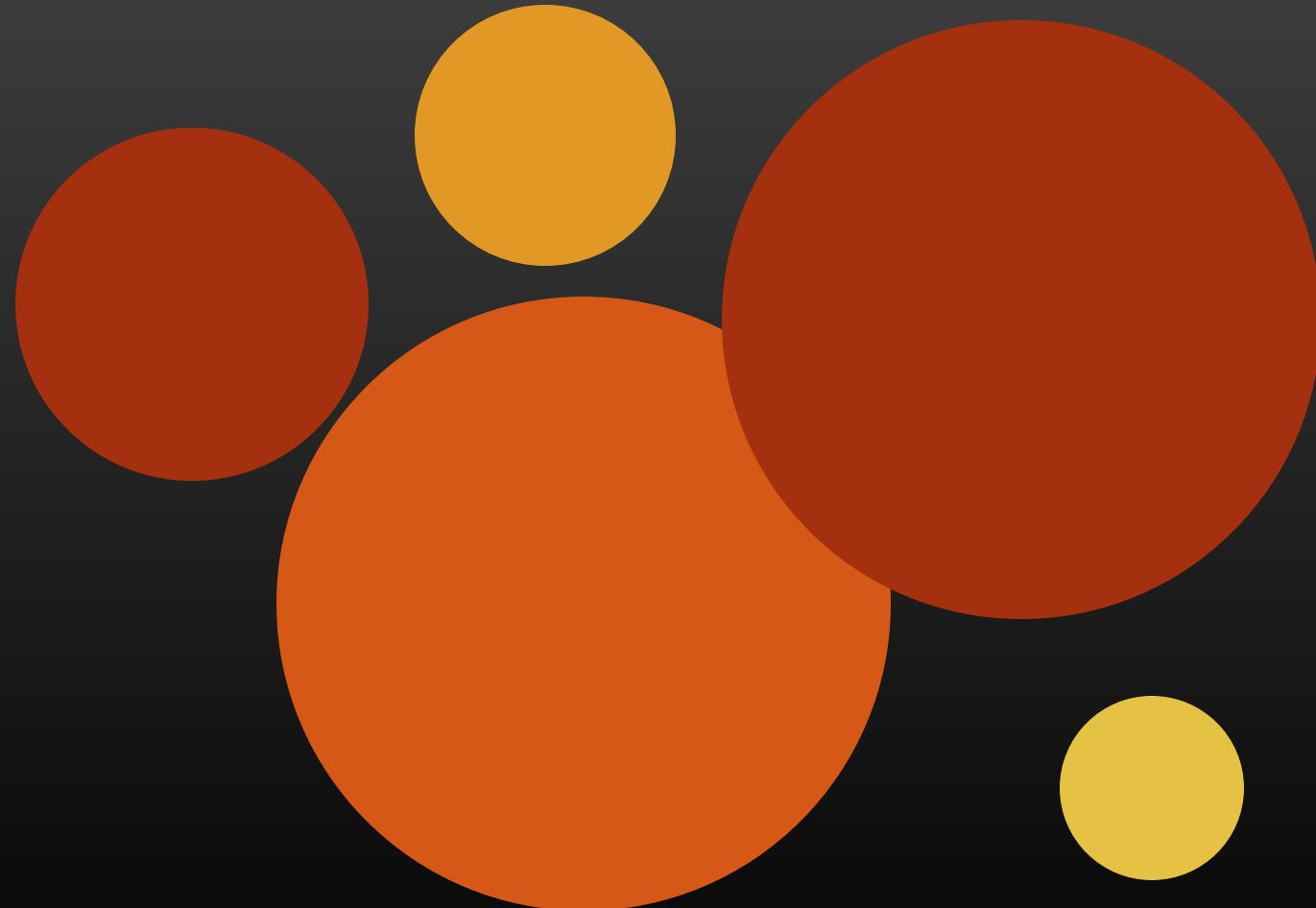








Done



# Reading

- Read Gregory Ch. 12.1-12.3

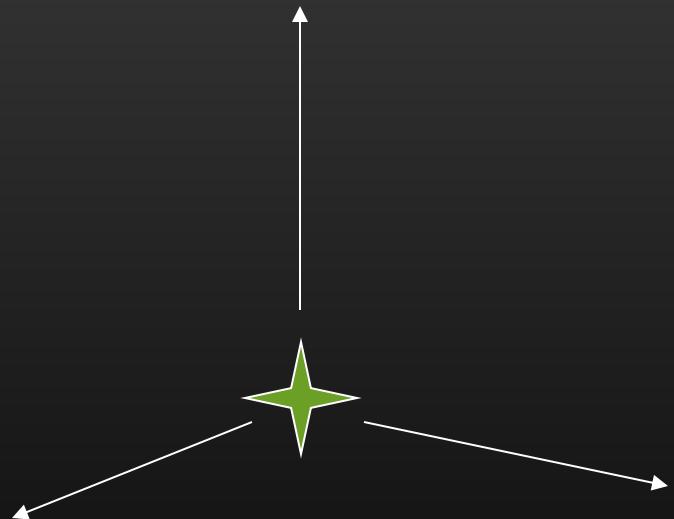
Modeling 3D space

# 3D

3D is the **same as 2D** except:

- We choose **3 axes for basis**
- And represent positions and vectors with
  - **3 coordinates**
  - Or **4 in projective coordinates**
- **Planes have 3 degrees of freedom**
  - They had zero in 2D

(Also, it's harder to draw convincingly)



# 3D in projective coordinates

Points represented as

$$\begin{bmatrix} wx \\ wy \\ wz \\ w \end{bmatrix}$$

Usually just:

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Kluge: vectors represented as

$$\begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix}$$

This way, standard addition does the right thing, if the point's **w coordinate is 1**

Otherwise, we use a **translation matrix**

# Normalizing projective coordinates

- To **convert** a point from projective coordinates back to Euclidean coordinates
- We **normalize** (divide) by the  $w$  component
- Note that this means that
  - **Increasing  $w$  scales the point down**
  - Decreasing  $w$  scales it up

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \Rightarrow \begin{bmatrix} x/w \\ y/w \\ z/w \end{bmatrix}$$

# Linear transformations on projective coordinates

**Rotation** about Z axis by  $\theta$  radians:

$$\begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We'll talk about other rotations later

**Translation** by  $[t_x \ t_z \ t_z]^T$ :

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

# Linear transformations on projective coordinates

**Uniform scaling** by factor  $s$ :

$$\begin{bmatrix} s & 0 & 0 & 0 \\ 0 & s & 0 & 0 \\ 0 & 0 & s & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

**Non-uniform scaling** by factors  $s_x, s_y, s_z$ :

$$\begin{bmatrix} s_z & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

# A kluge for scaling

- Remember we **divide by  $w$**  when converting to normal coordinates
  - So we can also scale uniformly by **reducing  $w$**
  - We don't usually do this because we like to keep  **$w = 1$**  when possible
  - But this will be **important later**
- Uniform scaling** by factor  $s$ :

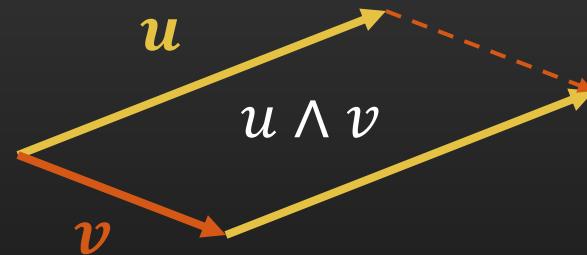
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1/s \end{bmatrix}$$

# The 3D wedge product

# The exterior product (aka wedge product)

- **Oriented area** created by **sweeping** one vector along another
- Oriented means it has both
  - A **magnitude**
  - A **direction**

We'll relate this to circular areas and angles next week



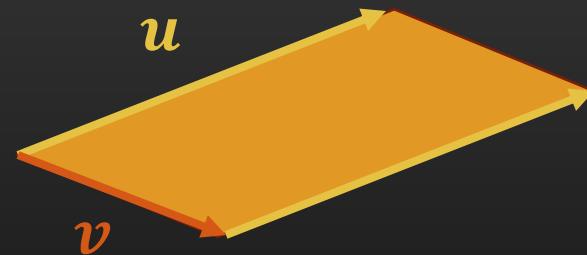
# The exterior product (aka wedge product)

- Start with a vector  $\mathbf{u}$



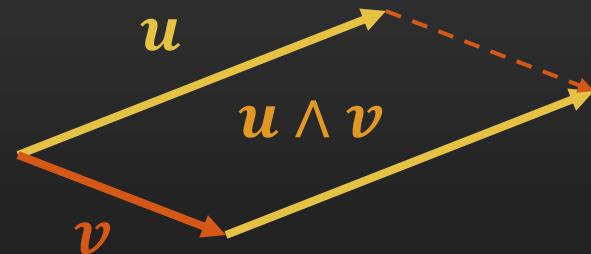
# The exterior product (aka wedge product)

- Start with a vector  $\mathbf{u}$
- Sweep it along another vector  $\mathbf{v}$



# The exterior product (aka wedge product)

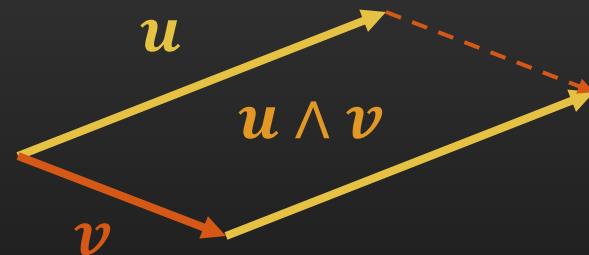
- Start with a vector  $\mathbf{u}$
- Sweep it along another vector  $\mathbf{v}$
- Resulting **area** is the **wedge product**  $\mathbf{u} \wedge \mathbf{v}$



# The exterior product in 2D

In 2D, the area is the **determinant** of the matrix formed by the two vectors:

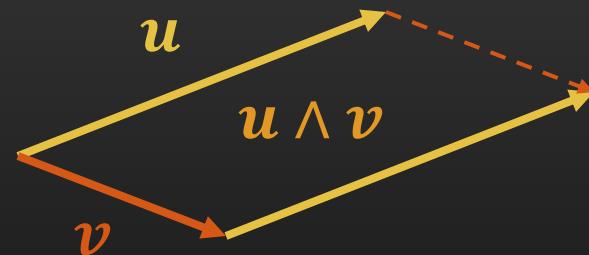
$$\begin{aligned} u \wedge v &= \left\| \begin{bmatrix} u_1 & u_2 \\ v_1 & v_2 \end{bmatrix} \right\| \\ &= \mathbf{u}_1 \mathbf{v}_2 - \mathbf{u}_2 \mathbf{v}_1 \end{aligned}$$



We'll talk about the 3D case later in the quarter

# The exterior product in 3D

- In 3D, you can have planes in **different orientations**
- So the exterior product has **3 degrees of freedom**
  - 2 dof of **orientation**
  - 1 dof of **magnitude**



# The exterior product in 3D

The exterior product in 3D  
is computed as

- The **2D product**
- Of **each pair** of basis vector
  - **XY** plane
  - **YZ** plane
  - **ZX** plane

$$\begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix} \wedge \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = \begin{bmatrix} u_y v_z - u_z v_y \\ u_z v_x - u_x v_z \\ u_x v_y - u_y v_x \end{bmatrix}$$

# The exterior product in 3D

- This will be familiar to you as the **cross product**
- The wedge product is a **generalization** of the cross product to n dimensions
- The result technically **isn't a vector**
  - It's a “bivector”
  - But we can mostly **ignore this**
  - See unit 7 for more on bivectors

$$\begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix} \wedge \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = \begin{bmatrix} u_y v_z - u_z v_y \\ u_z v_x - u_x v_z \\ u_x v_y - u_y v_x \end{bmatrix}$$

# Projection

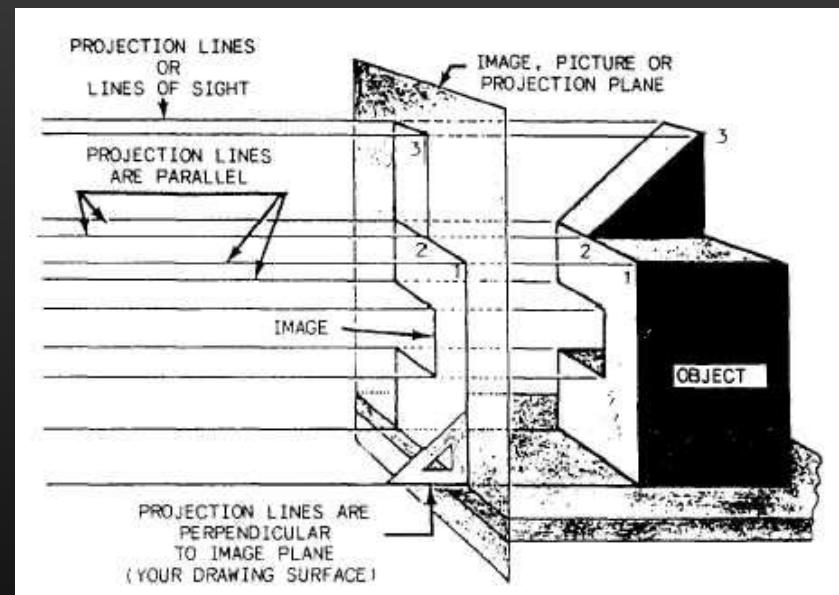
# Projection

The camera projects

- From **3D positions**
  - In **camera coordinates**
  - Represented by **4-vectors**
- To **2D screen coordinates**
  - Which we'll still represent as **4-vectors**
  - Because that turns out to be most convenient

# Orthographic projection

- Essentially means **throwing away the z component**
  - Or rather, setting it to 0
- Often incorporate a **scale factor** to help fit scene onto the screen



# Orthographic projection in games



# Orthographic projection

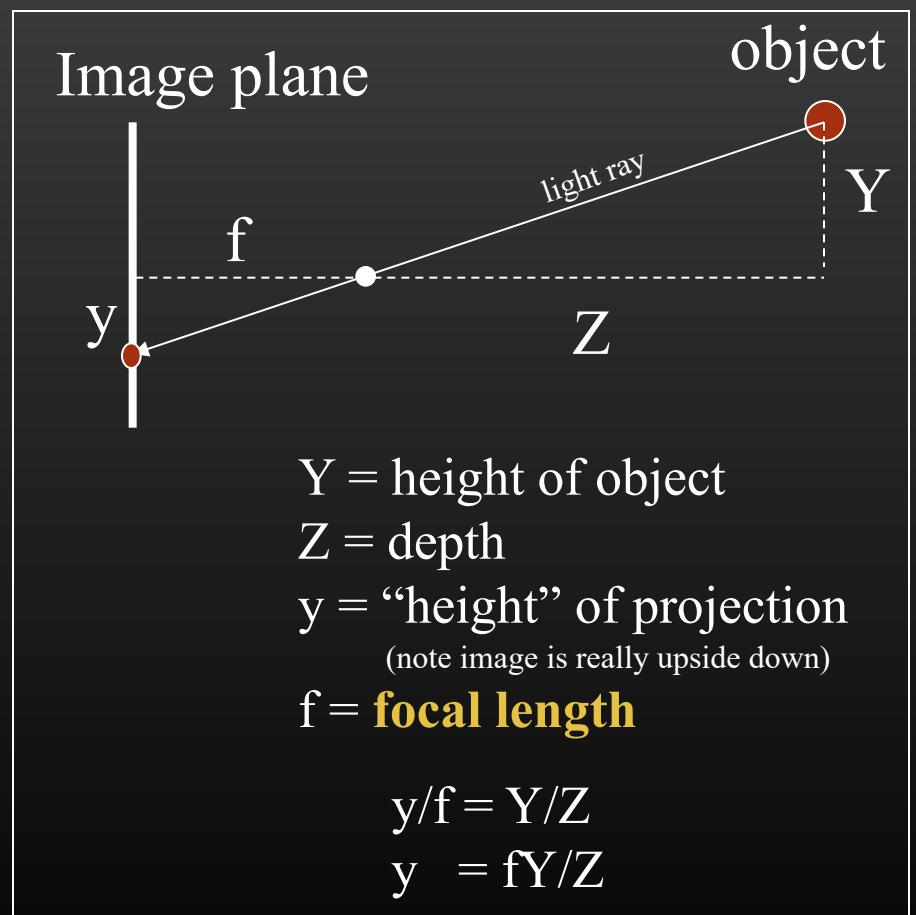
- Computing image plane position is just a matter of going from **3D to 2D**
- That's a **linear** transform
- So we can do it with another  $4 \times 4$  matrix and just **ignore the z coordinate**

Orthographic projection with **scale factor**  $s$ :

$$\begin{bmatrix} s & 0 & 0 & 0 \\ 0 & s & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

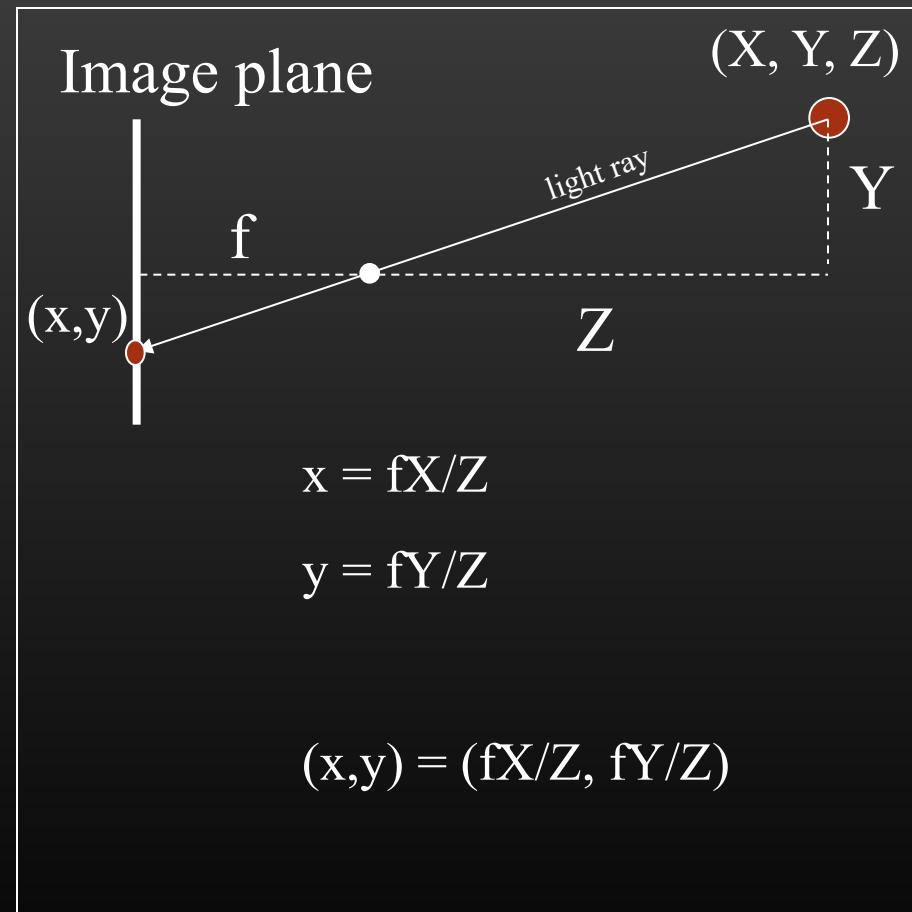
# Perspective projection

- A **real camera** uses perspective projection
  - Objects are **scaled based on distance**
- In terms of coordinates, we **divide by z**
- Usually also incorporate a **scale factor** called the **focal length**



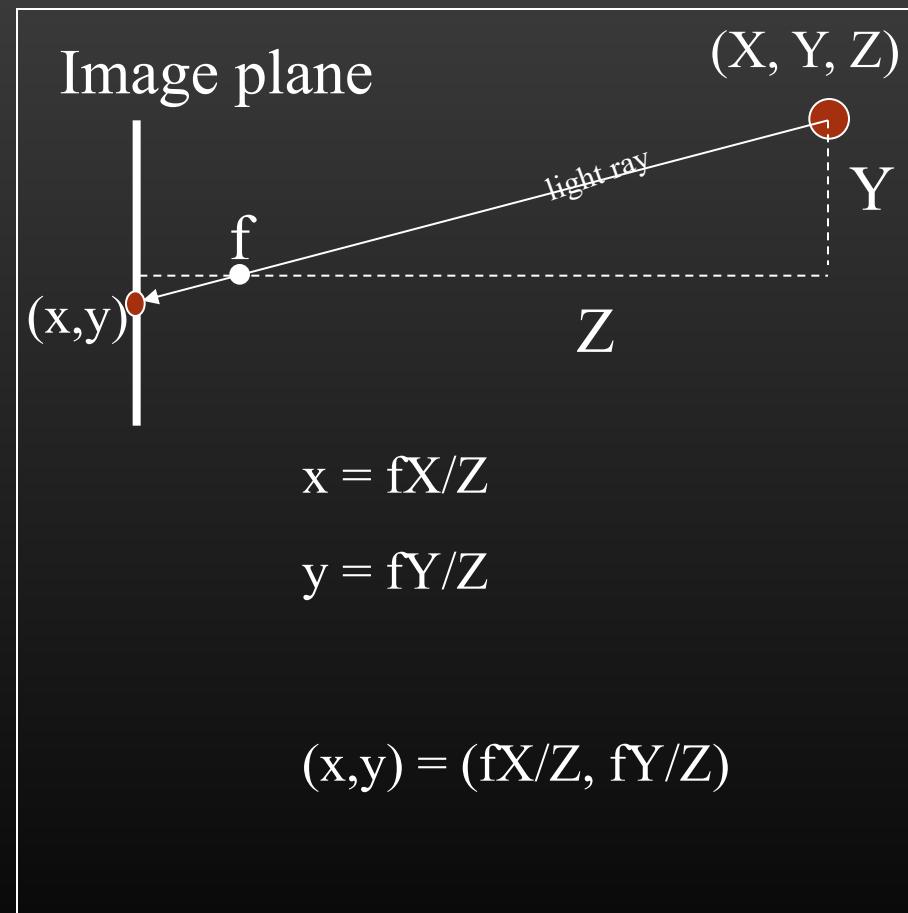
# Perspective projection

- A **real camera** uses perspective projection
  - Objects are **scaled based on distance**
- In terms of coordinates, we **divide by z**
- Usually also incorporate a **scale factor** called the **focal length**



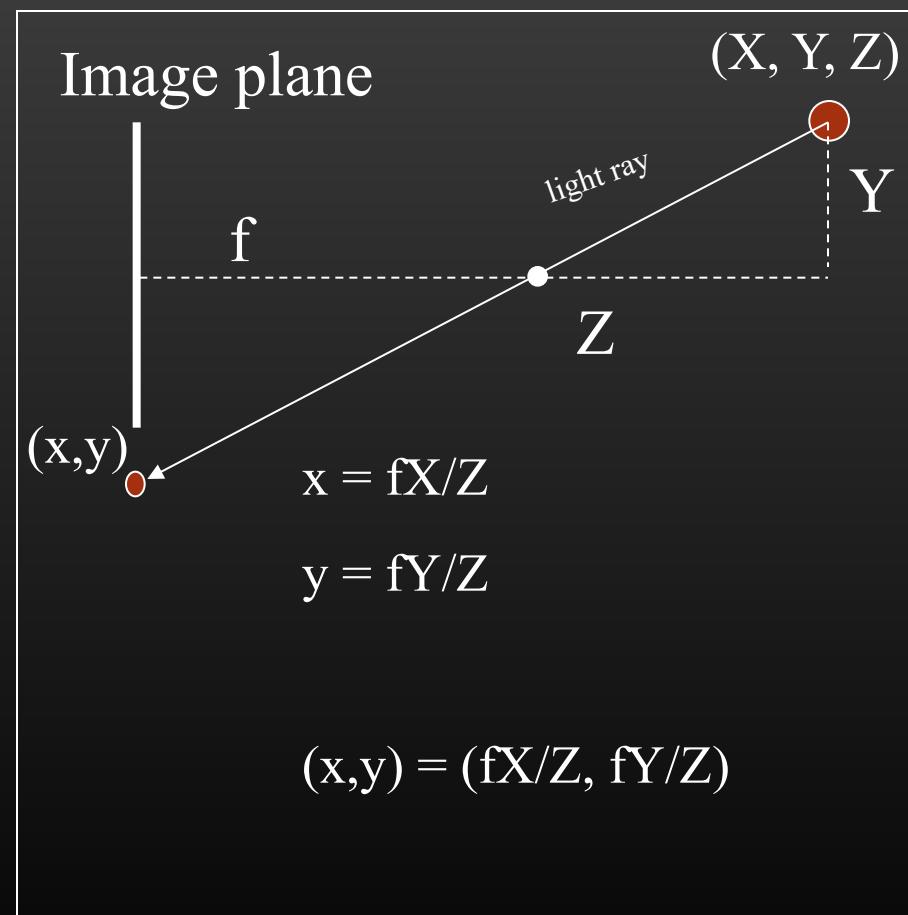
# Perspective projection

- Decreasing the focal length makes the image smaller
  - But also increases your **field of view**
  - Lenses with short focal lengths are therefore called **wide-angle** lenses



# Perspective projection

- Increasing the focal length makes the image bigger
  - But decreases your field of view
  - Lenses with large focal lengths are called **telephoto** lenses



# Perspective projection

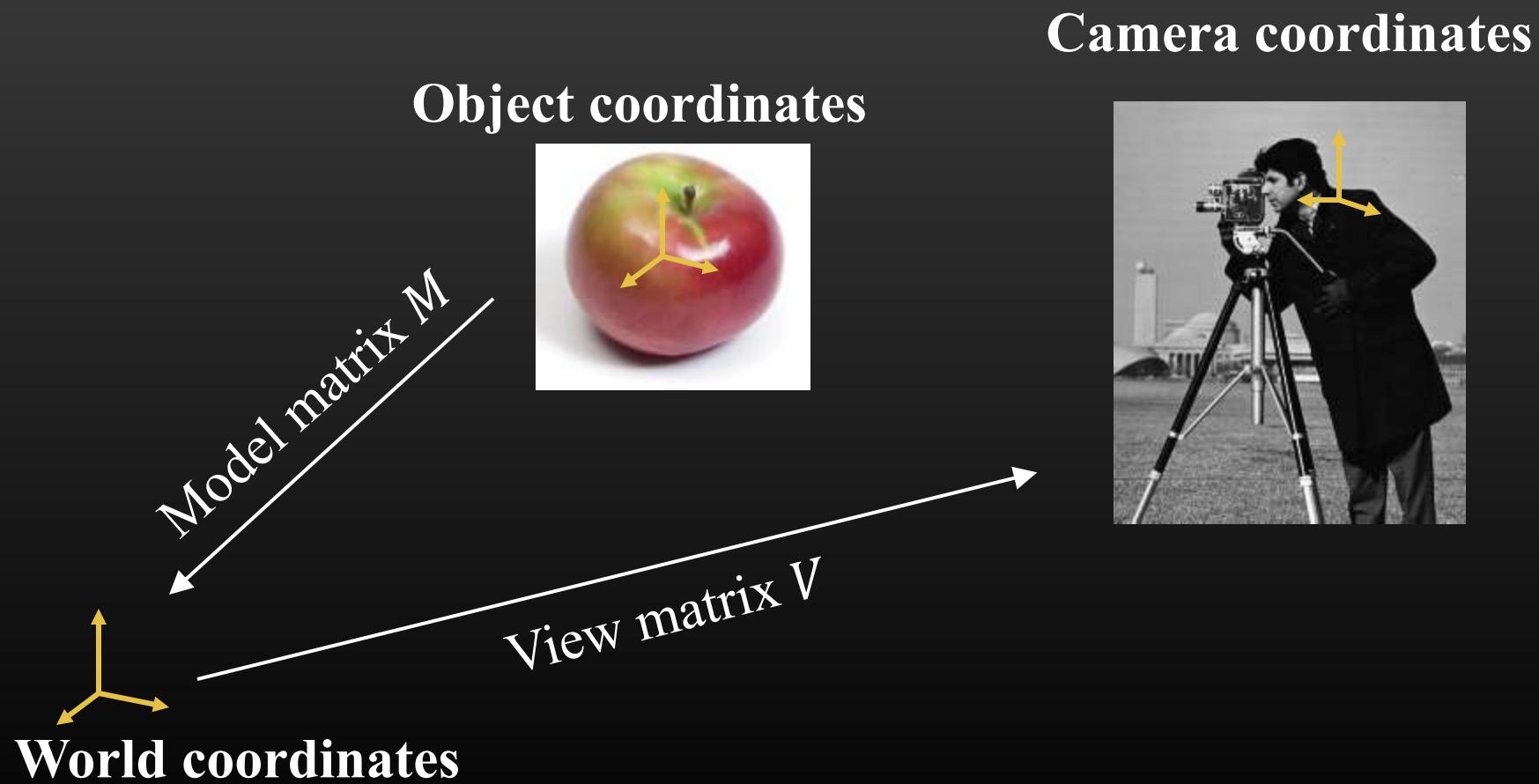
- Perspective requires **dividing by Z**
  - So a variable scaling
  - That depends on one of the coordinates
- That's very much **not linear**
- But we can use our **scaling trick** for projective coordinates
  - **Multiply w**
  - To shrink everything else
- To implement perspective
  - **Make w be z**

- Perspective projection
  - $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
  - Perspective projection with focal length  $f$ :

$$\cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{f} & 0 \end{bmatrix}$$

# Chained coordinate systems

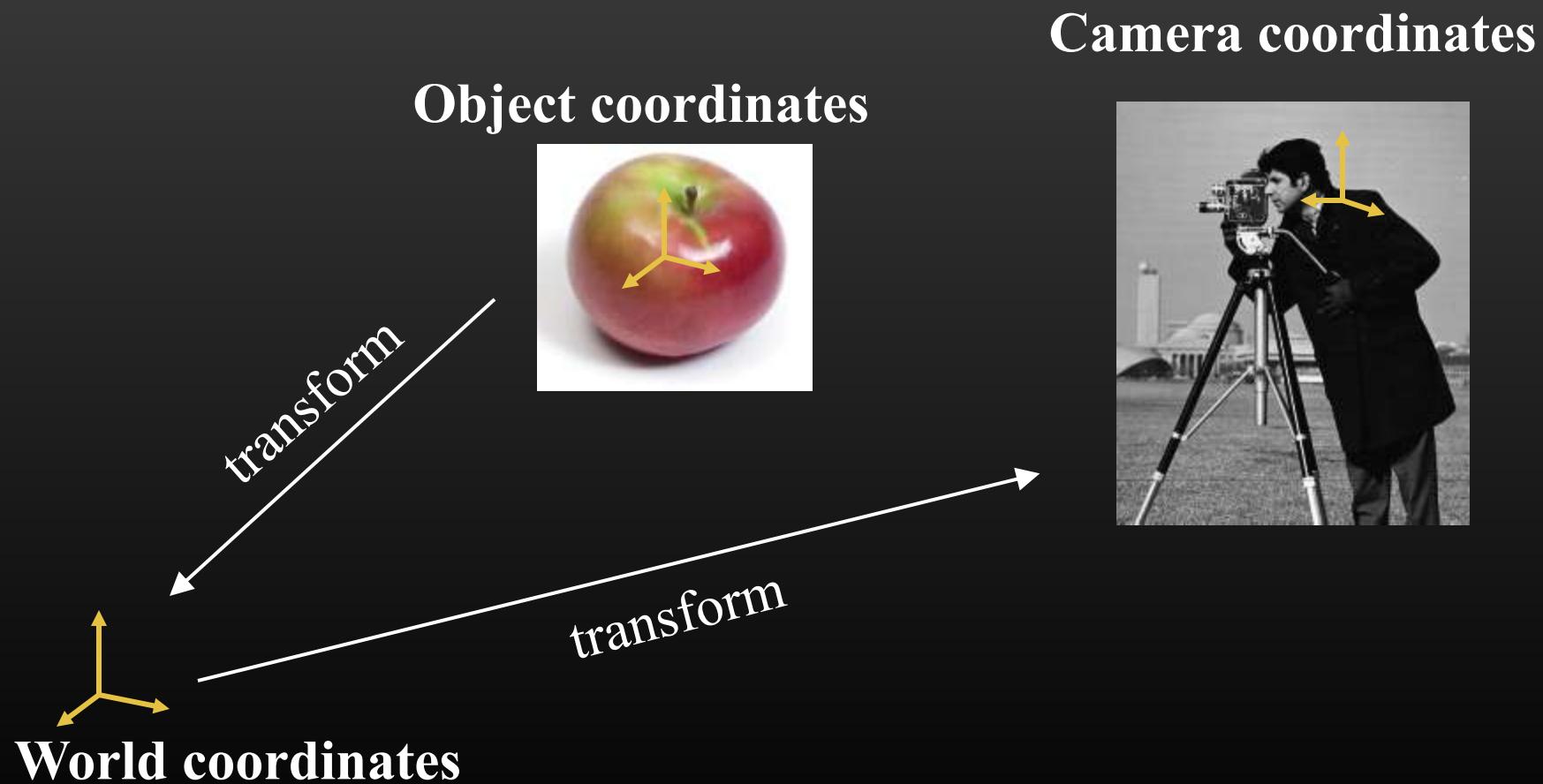
# Viewing an object in different coordinate systems



# Rendering objects

- 3D models are specified in **object-local coordinates**
  - The model is positioned in space by specifying a coordinate **transform from local to world coordinates**
  - This is called the object's **model matrix  $M$** 
    - Aka the world matrix of the object
  - World coordinates of a point  $p$  in the model are given by  $\mathbf{Mp}$
- But we render from the **camera's viewpoint**
  - Which is specified by a transform from **world to camera coordinates**
  - Called the **view matrix  $V$**
  - So camera coordinates of a point  $p$  in the model are given by  $\mathbf{Vm}p$

# Viewing an object in different coordinate systems



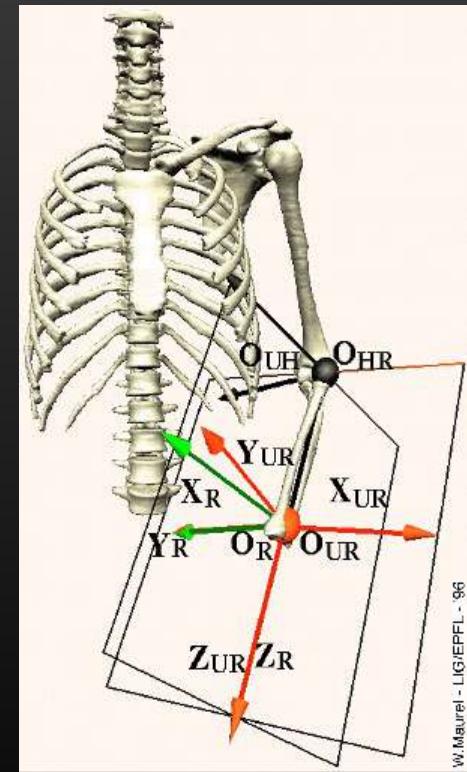
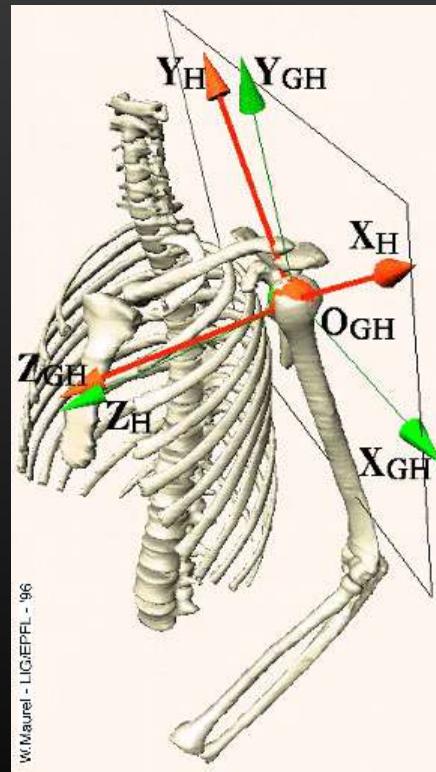
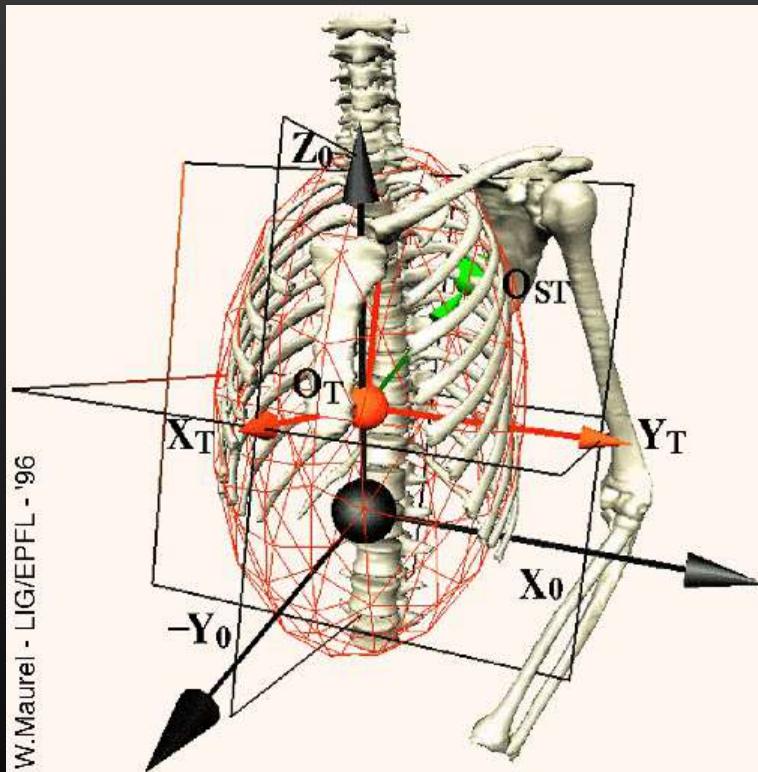
# The model/view/projection matrix

- So what your GPU is really using when it converts from model coordinates to screen coordinates is the **product** of
  - The **Model** matrix  $M$
  - The **View** matrix  $V$
  - The **Projection** matrix  $P$
- So the screen coordinates of a point  $p$  are  $(PVM)p$
- We can **compute the  $PVM$  matrix once** and multiply all object points by it
  - Saving work

# Well, almost

- As you know, game objects are **really trees**
  - Each with its own transform from **local coordinates to parent's coordinates**
  - You can **move an object relative to its parent** by changing just its matrix
  - That will also move any of the **local object's children**
- So in real life, the model matrix M is really a **product of a series of matrices**
  - Starting with the matrix of the local object
  - Then the matrix of its parent
  - Then the matrix of its grandparent
  - Etc.
- Again, all these can be **premultiplied once**

# Chained coordinate systems



# Rotation in 3D

# Question

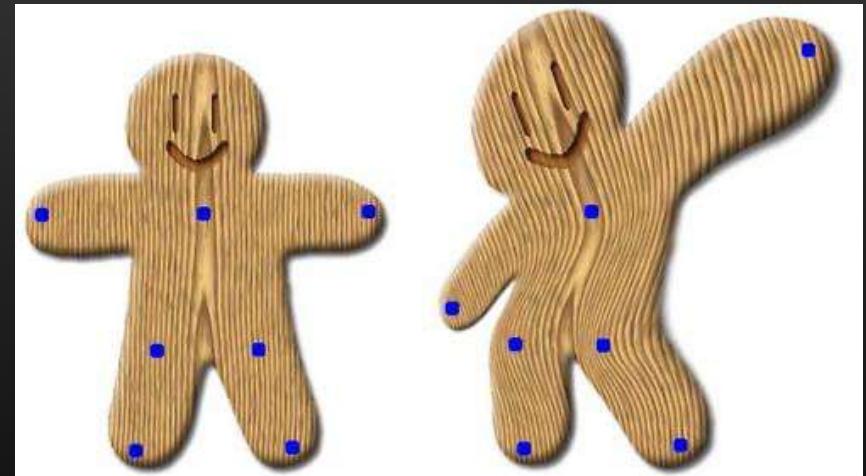
- How do we represent the **position** of an object?

It depends  
on what you mean by “position”

**Any movement** you can make to an object ought to  
count as **changing its position**  
(short of cutting, burning, smashing, etc.)

# There are a lot of ways you can move an object

- **Translate** it
- **Rotate** it
- **Stretch** it
- Squish it
- **Bend** it
- Tie it up in a knot



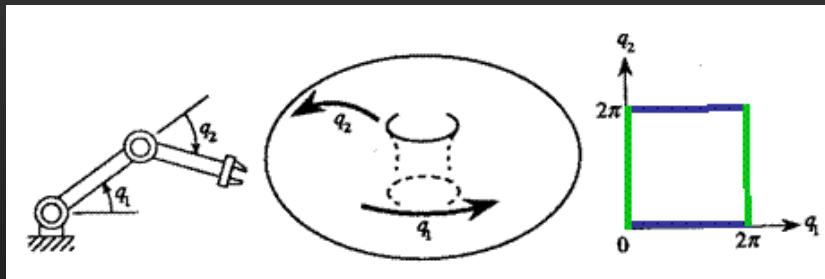
source: [http://www.cs.cmu.edu/afs/andrew/scs/cs/15-463/f07/proj\\_final/www/mdouglal/](http://www.cs.cmu.edu/afs/andrew/scs/cs/15-463/f07/proj_final/www/mdouglal/)

How do you represent  
the position of an object?

How do you represent  
the ~~position~~ of an object?  
**pose**

How do you represent  
the ~~position~~ of an object?  
**configuration**

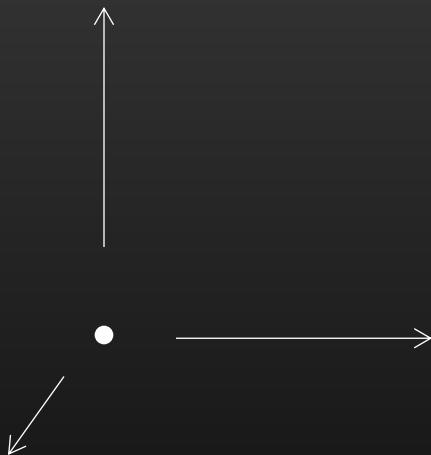
# Configuration space



source: [http://robotics.stanford.edu/~latombe/cs326/2009/class3/class3\\_files/image004.gif](http://robotics.stanford.edu/~latombe/cs326/2009/class3/class3_files/image004.gif)

- An object's configuration space is the **set of possible configurations** you can move it into
- You can **understand** the configuration space of an object
  - By understanding the **motions** (configuration changes)
  - That **you can put it through**

# Configuration space



Different objects have  
**different configuration  
spaces**

- A (point) **particle** only has **3 degrees of freedom** in its configuration
- All you can do is **translate** it

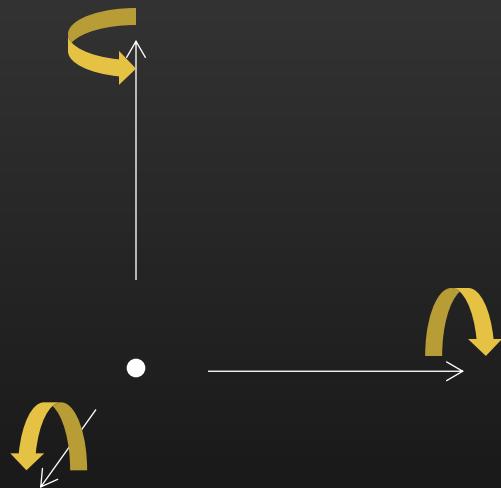
# Configuration space



- A **liquid**, however, has an **infinite** number of degrees of freedom
- Because you can pour it into a container of any possible shape

source: <http://blog.lib.umn.edu/marqu154/architecture/water-drop-1b.jpg>

# Configuration space



- **Rigid bodies** have a much simpler configuration space

- All you can do is
  - **Translate** them and
  - **Rotate** them

How do we represent the  
**configuration** of a **rigid body**?

# First idea: position + direction

We can represent an object in terms of its

- **Position**
  - E.g. position of its **center of mass**
- **Direction**
  - E.g. what direction its **facing**



# First idea: position + direction

- This is a really convenient and **intuitive** representation



# First idea: position + direction

- This is a really convenient and intuitive representation
- Unfortunately, it **doesn't work**
  - It doesn't capture one of the **rotational degrees of freedom**



Stephen is **facing the same direction** as on the last slide, but is in a **different configuration**

# Better idea

- Choose some **reference configuration**



# Better idea

- Choose some reference configuration
- **Pose = translation + rotation** required to change reference configuration to current configuration
- Great! How do we **represent** translations and rotations?



# Representing translation

- How do we represent **translation**?
- Easy! It's just a **vector**
  - Or a **translation matrix** if you prefer



# Representing rotation

Lots of ways

- Rotation **axis** plus **angle** of rotation
- Single **rotation vector**
  - Direction = axis
  - Magnitude = angle
- Rotation **matrix**
- **Euler angles**
- Yaw, pitch, roll



these all suck

Sadly,

part 1:  
angles, axes,  
and matrices

# Rotation vectors

# Axis + angle

- **Euler's rotation theorem**

(Roughly): in 3-space, any series of rotations is equivalent to a **single rotation** about some single axis

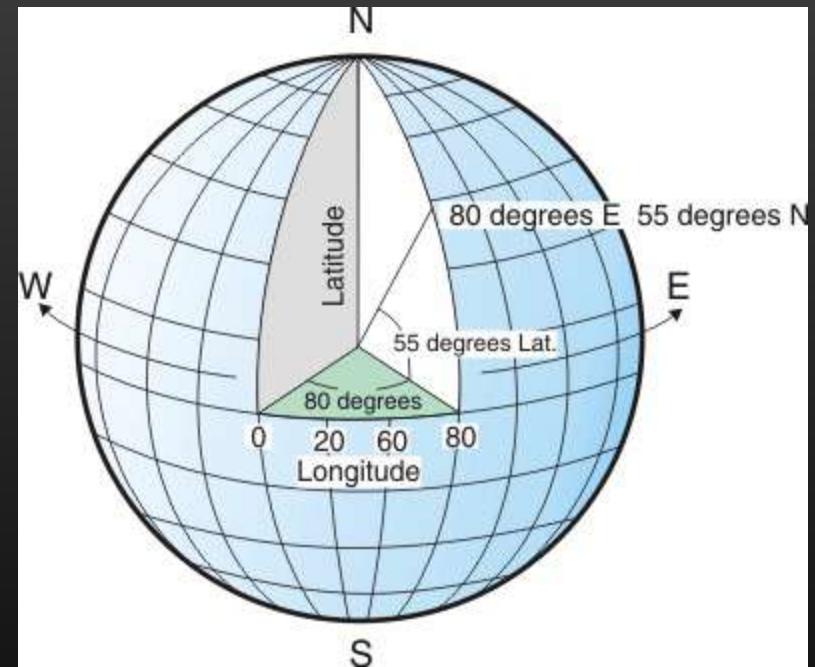
- This gives us an easy way to **represent any rotation** or orientation
  - Figure out the **equivalent axis/angle**
  - Represent the **axis** (somehow)
  - Represent the **angle** (e.g. in radians) to rotate about the axis



# Representing the axis

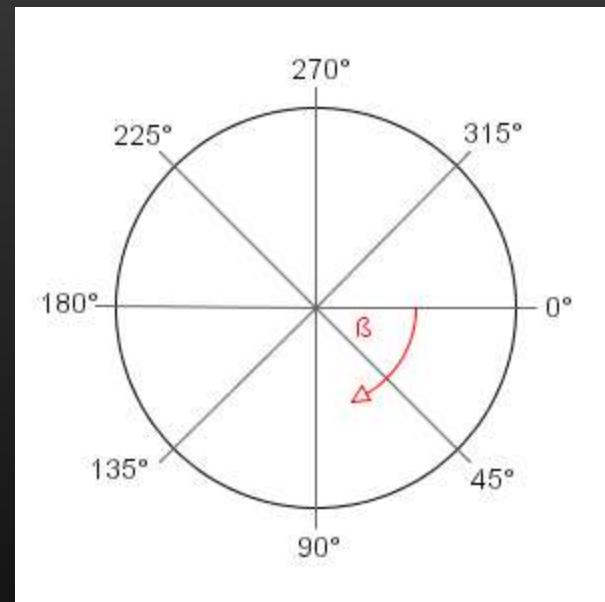
**Representing direction** is a pain

- In 3-space, a direction has **2 degrees** of freedom
- So you'd think you could just use **two numbers**
- They'd have to be **angles**
  - E.g. **longitude** and **latitude**



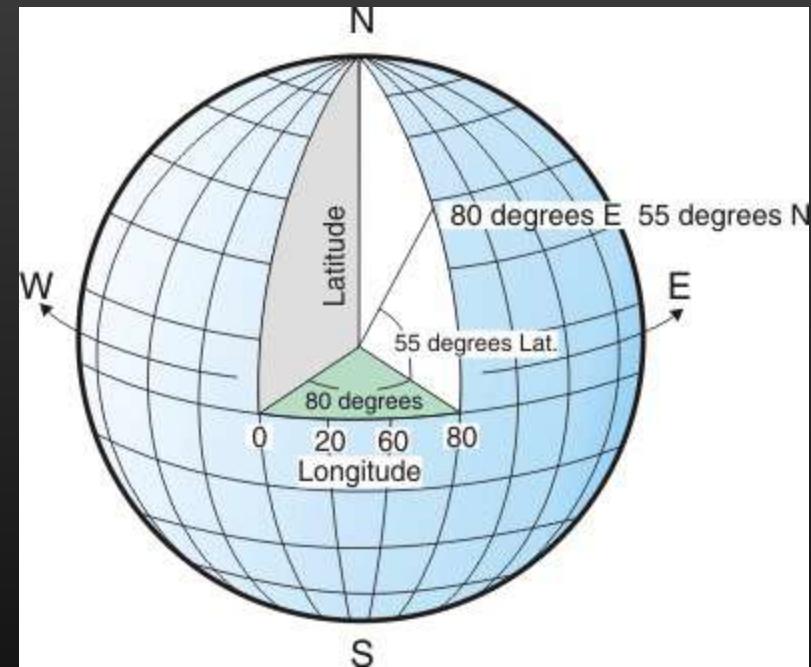
# Problems with angles

- Numbers and angles aren't **topologically equivalent**
  - Angles are positions on a **circle**
  - Real numbers are positions on a **line**
- So you either put up with
  - **Discontinuities** (wrapping around from 360 to 0), or
  - **Duplication** (0, 360, and 720 all represent the same angle)
- Either way, it makes life more **difficult** than you'd like



# Problems with angles

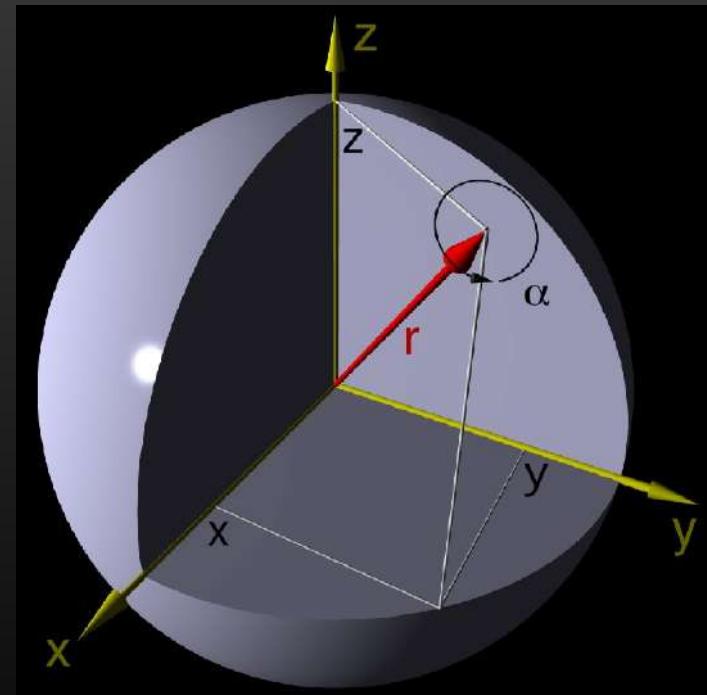
- It's **worse in 3-space**
  - Use **two angles**
  - But they **interact** in strange ways
- What's the **longitude** of the **north pole**?
  - **No longitude?**
  - **Every** longitude?



# Projective geometry to the rescue

The **standard trick** is to

- **Ignore** the fact that a direction in  $n$ -space only has  $n - 1$  degrees of freedom
- Represent it with  **$n$  numbers** anyway
- Usually, a **unit vector** in  $n$ -space



# How do we represent axis + angle?

- Explicit **axis and angle**

- Axis = unit vector
  - Angle = scalar

- **Rotation vector**

- Unit vector times angle
  - Direction = axis
  - Magnitude = angle



# Using axis-angle representations

$$\mathbf{v}_{rot} = \mathbf{v} \cos \theta + (\boldsymbol{\omega} \times \mathbf{v}) \sin \theta + \boldsymbol{\omega}(\boldsymbol{\omega} \cdot \mathbf{v})(1 - \cos \theta)$$

**Rodrigues' rotation formula** lets us efficiently

- Rotate **a point  $\mathbf{v}$**
- About the **origin**
- By a specified **angle  $\theta$**
- About an axis specified by a **unit vector  $\boldsymbol{\omega}$**

(At least for some definition of “efficiently”)

# Pros and cons

## Pros

- Relatively **simple**
- **Compact**  
(uses little storage)
- **Efficient** to compute

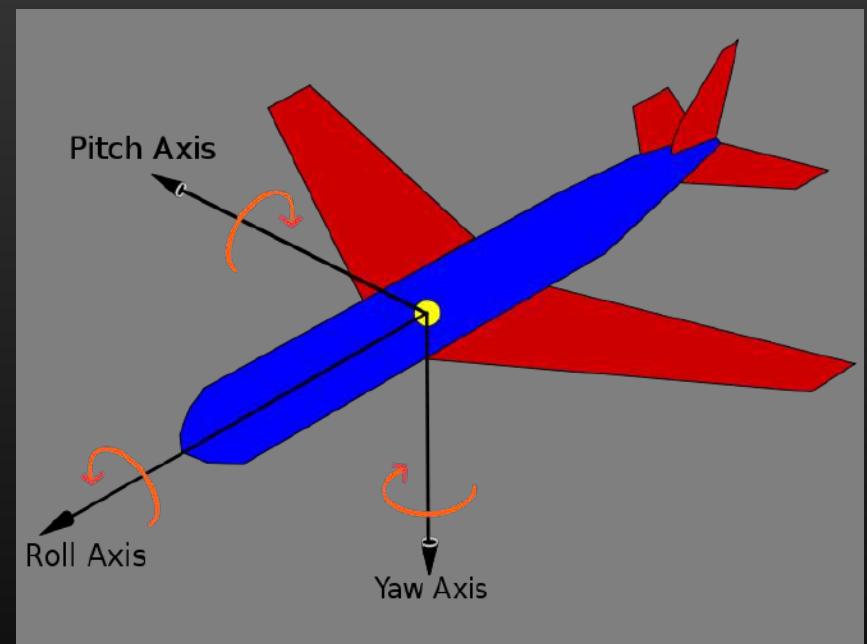
## Cons

- Hard to **compose** rotations
  - What's the axis-angle representation of the result of
    - First applying one rotation
    - And then another?
- Can't **interpolate** easily
  - What's the axis-angle pair that's **halfway between** two other axis-angle pairs?

# Angle-based representations

# Yaw, pitch, and roll

- It's also common to describe rotations in terms of **standard object-centered axes**
  - Up/down axis (yaw)
  - Left/right axis (pitch)
  - Front/back axis (roll)
- Any rotation can be described as a **combination** of these rotations



# Pros and cons

## Pros

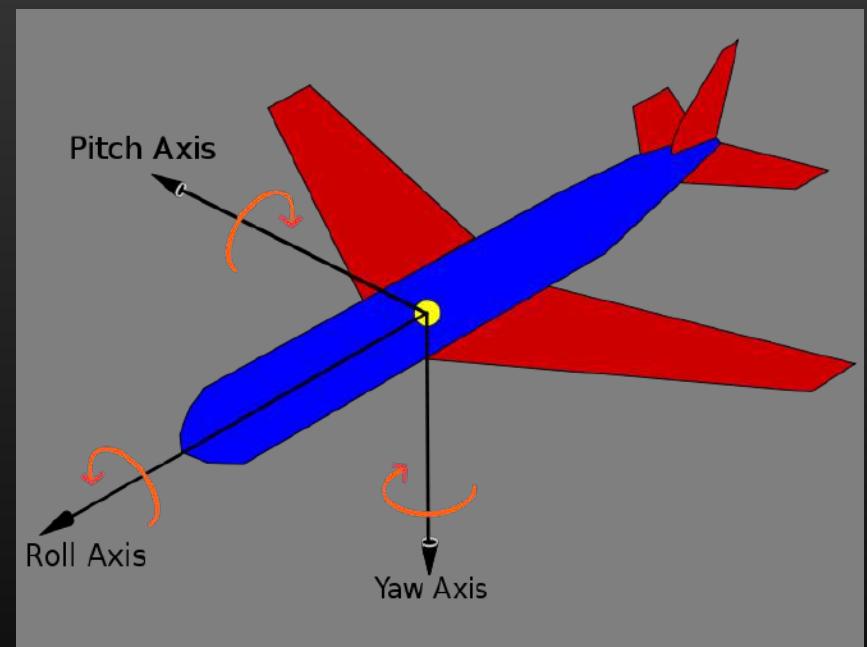
- Easy to understand
- Compact
- Fairly easy to compute

## Cons

- Not simple to **determine Y-P-R angles** given a rotation
- **Not commutative**
  - Roll  $\phi$ , yaw  $\theta$
  - Is different from yaw  $\theta$ , roll  $\phi$
- Still **hard to compose**
- Still **can't interpolate**

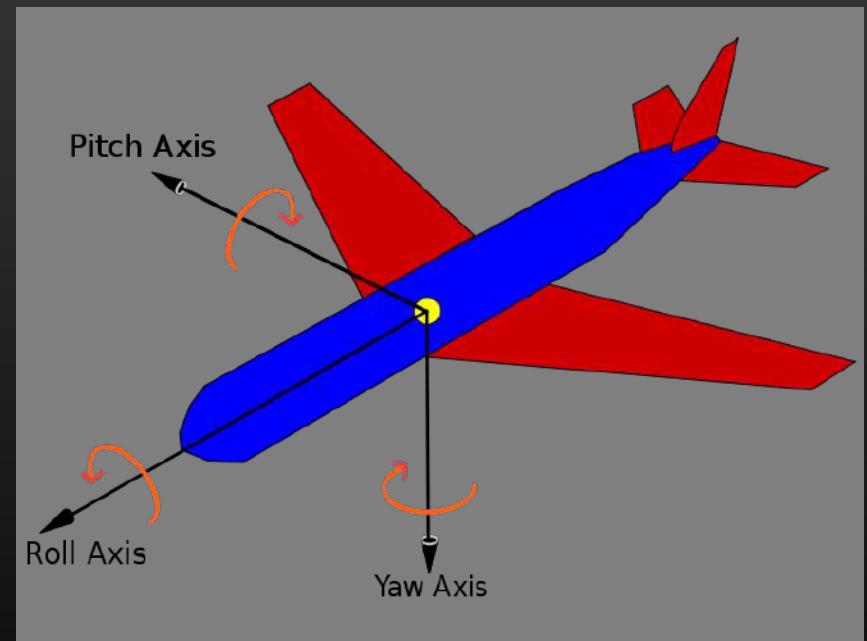
# Another problem

- Then we have to choose some **order** for the rotations
  - E.g. yaw first, then pitch, then roll
- But each rotation **rotates the axes** of subsequent rotation(s)
- What happens if the **second angle** (pitch) is **90 degrees**?



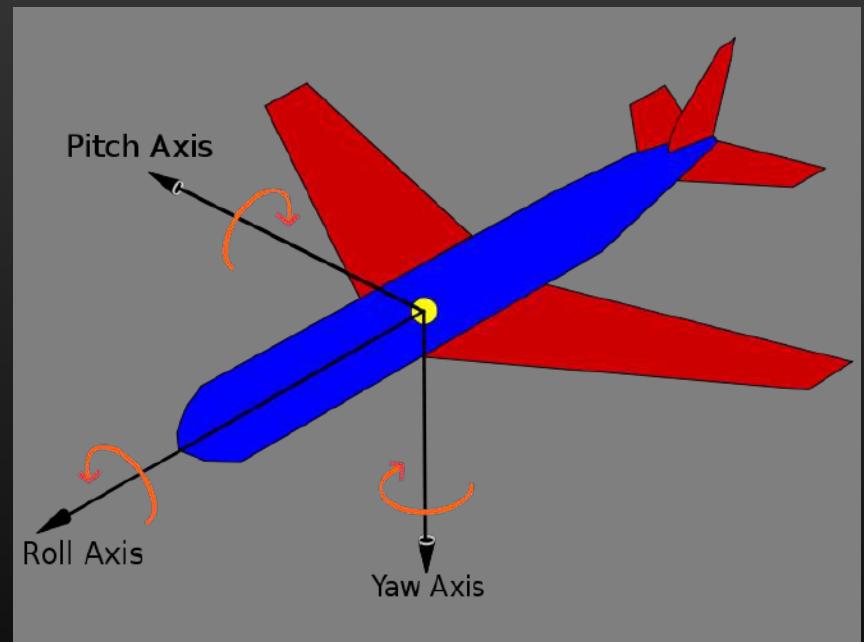
# Gimbal lock

- If the **second angle** is 90 degrees
  - Then it rotates the **last axis** to be **the same as the first axis**
- From there, you can only represent rotations along **two axes**
  - Because two of your three **axes are really the same**
- This is called **gimbal lock**



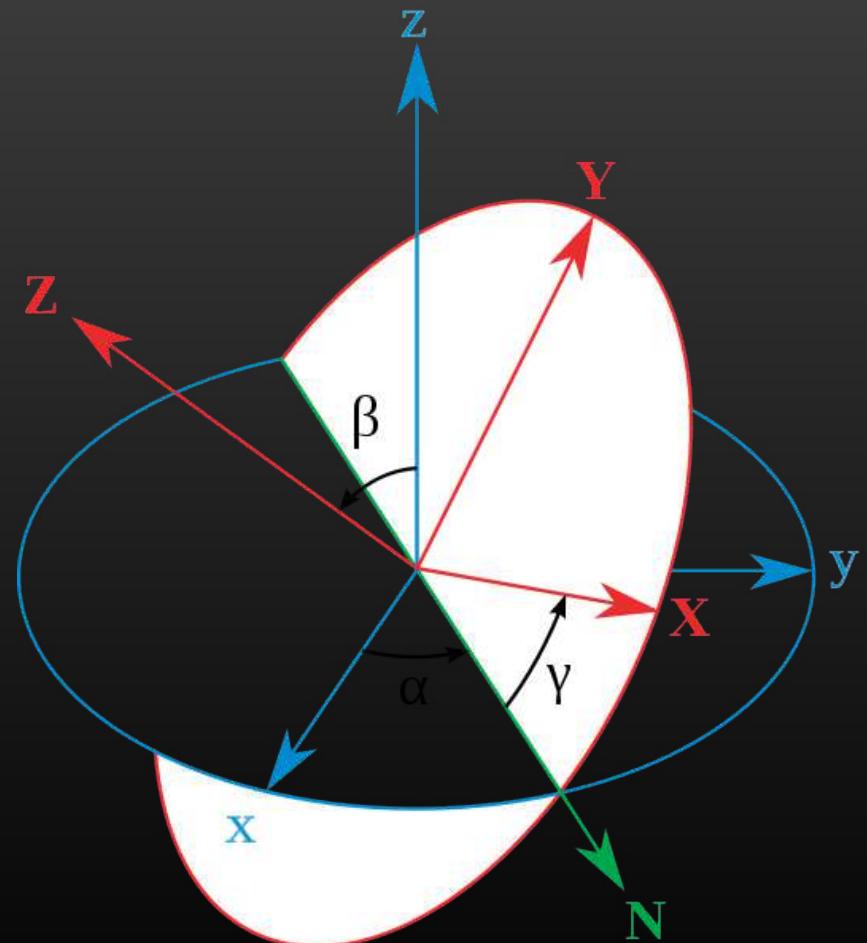
# Gimbal lock

- Is gimbal lock a **problem**?
- It depends on your **application**
- For an **FPS** (with no physics),
  - Representing aim in terms of **yaw** and **pitch** (aka longitude and latitude) makes perfect sense
  - Don't even need roll
- But it's **bad for physics** calculations that involve **arbitrary tumbling** of objects



# Euler angles

- Often confused with yaw/pitch/roll
  - What Unity calls Euler angles are actually YPR
- Let
  - The **original axes** be **xyz**
  - The **new axes** after rotation be **XYZ**
  - **N** be the **intersection** of the XY and xy planes
- The **Euler angles** are
  - $\alpha$ , the angle between the old **x axis and N**
  - $\beta$ , the angle between the **old z and new Z axes**
  - $\gamma$ , the angle between **N and the new X axis**



# Pros and cons

## Pros

- Compact
- Angles **aren't order-dependent**

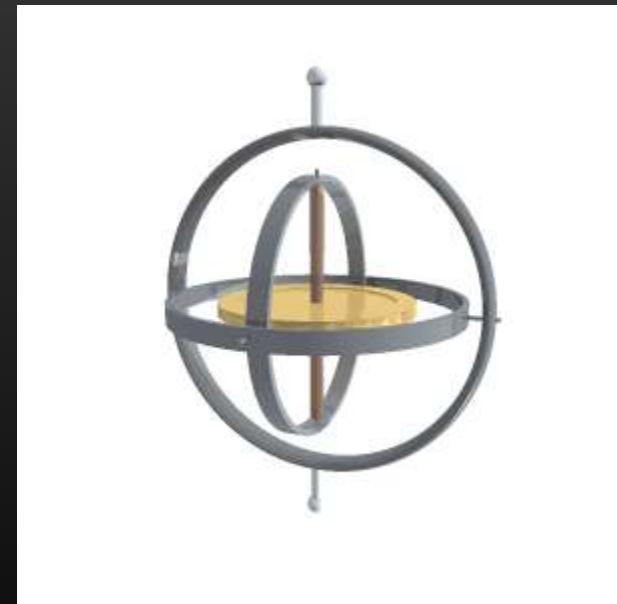
## Cons

- **Not composable**
- **Not interpolatable**
- **Not intuitive**
- **Gimbal lock**

# Gimbal lock

# Euler angles and gimbals

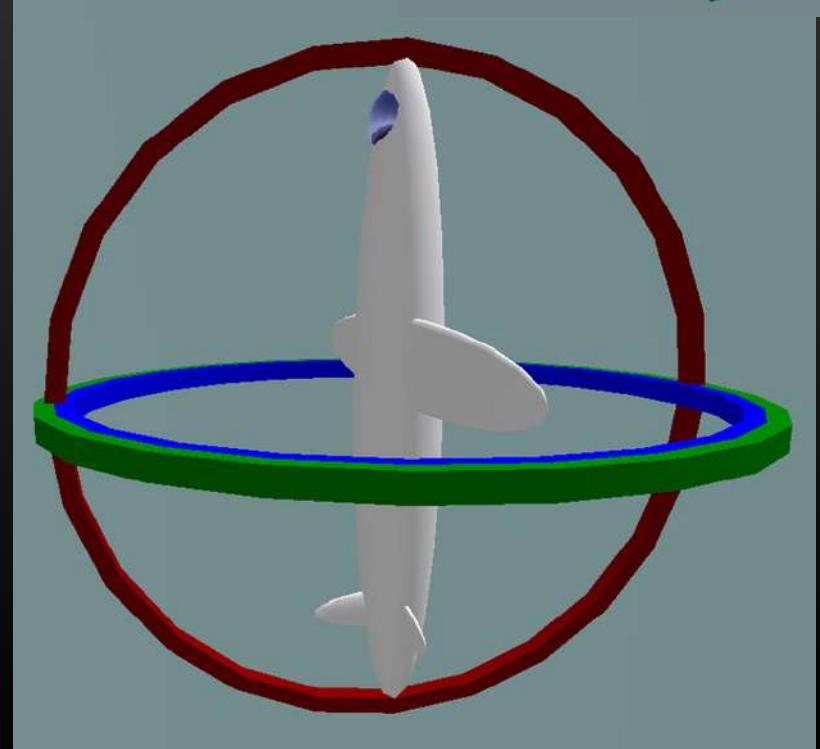
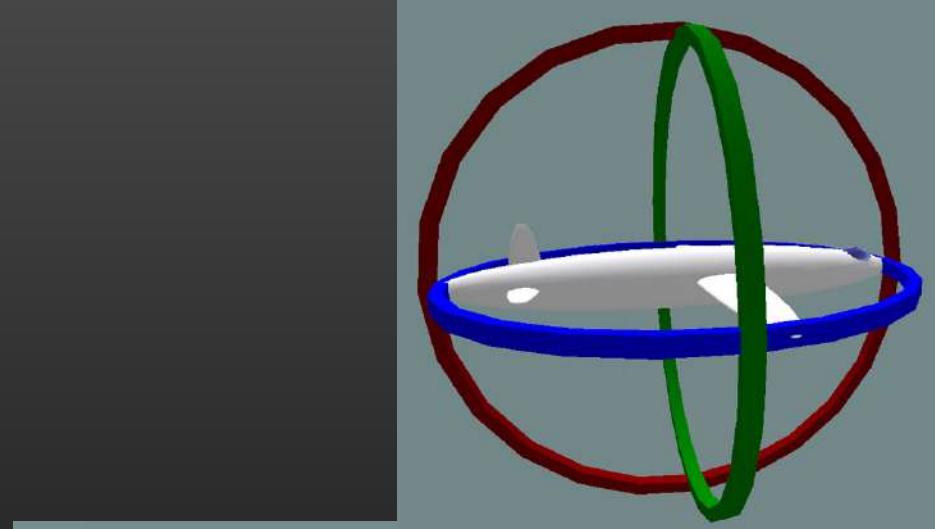
- **Gimball** mechanisms give us a good way of **visualizing Euler angles**
- The **angles between the rings** are essentially Euler angles
- What happens **when one of them rotates a full 90 degrees?**



# Gimbal lock

If **one of the rings**

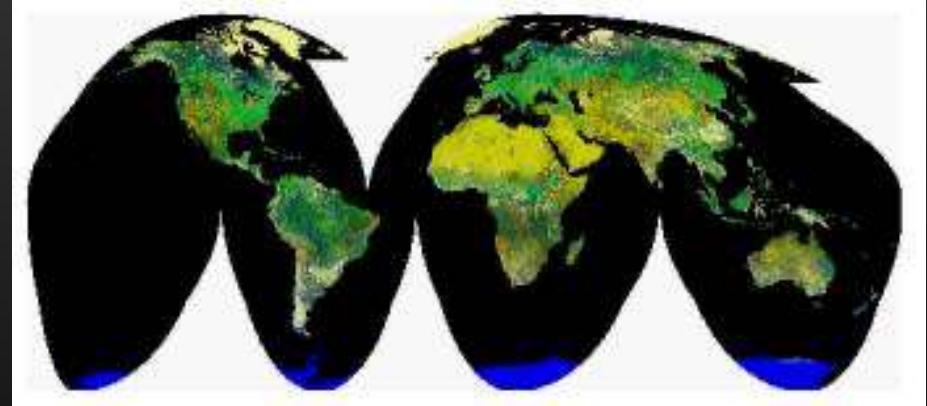
- Rotates enough
- To become **parallel** to one of the other rings
- Then it **locks** the mechanism in one direction
- Until the rings come **out of alignment**



# Discontinuity

Ultimately, the **problem** is that

- There can be **no mapping**
- From any set of **3 numbers**
- To **rotations**
- That doesn't have **discontinuities**



Basically the **higher dimension version** of the problem with mapping the globe in a plane

# Rotation matrices

# Rotation matrices

- Rodriguez's formula is **linear** in  $\mathbf{v}$

$$\mathbf{v}_{rot} = \mathbf{v} \cos \theta + (\boldsymbol{\omega} \times \mathbf{v}) \sin \theta + \boldsymbol{\omega}(\boldsymbol{\omega} \cdot \mathbf{v})(1 - \cos \theta)$$

- That means it has to be equivalent to a matrix multiplication

# Cross-product matrix

- A **cross product is equivalent to a matrix multiplication**

$$\boldsymbol{\omega} \times \boldsymbol{v} = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix} \boldsymbol{v} = [\boldsymbol{\omega}]_{\times} \boldsymbol{v}$$

- We can use this to build the matrix form of a rotation

# Rotation matrices

- Rodriguez's formula is **linear** in  $\mathbf{v}$
- So for any  $\boldsymbol{\omega}, \theta$  we can find a **matrix** that rotates vectors around  $\boldsymbol{\omega}$  by  $\theta$ :

$$\begin{aligned}\mathbf{v}_{rot} &= \mathbf{v} \cos \theta + (\boldsymbol{\omega} \times \mathbf{v}) \sin \theta + \boldsymbol{\omega}(\boldsymbol{\omega} \cdot \mathbf{v})(1 - \cos \theta) \\ M_{\boldsymbol{\omega}, \theta} &= I \cos \theta + [\boldsymbol{\omega}]_x \sin \theta + (1 - \cos \theta)[\boldsymbol{\omega}]_x^2\end{aligned}$$

# Pros and cons

## Pros

- No gimbal lock
- **Easy to compose** rotations
  - Product of two rotation matrices is a rotation matrix (yay!)
- Easy to **compose with other operations**, if using homogeneous coordinates
  - One matrix can translate and rotate
  - Or even translate, rotate, scale, and project
- Easy to derive from any of the other representations (e.g. Euler angles)

## Cons

- More **memory**
  - 9 floats rather than 3 or 4
  - 16 floats, if you're using homogeneous coordinates
- **Still can't interpolate**
  - The **average of two rotation matrices** isn't necessarily even a rotation matrix!
  - You can **orthonormalize** the matrix using the Gram-Schmidt method
  - But it still doesn't produce satisfying looking results

# Euler-Rodriguez form

# Component form of rotation matrices

- When we multiply this out, we get:

$$M_{\omega,\theta} = I \cos \theta + [\boldsymbol{\omega}]_x \sin \theta + (1 - \cos \theta)[\boldsymbol{\omega}]_x^2$$
$$= \begin{bmatrix} \cos \theta + \omega_x^2(1 - \cos \theta) & \omega_x \omega_y(1 - \cos \theta) - \omega_z \sin \theta & \omega_x \omega_z(1 - \cos \theta) + \omega_y \sin \theta \\ \omega_y \omega_x(1 - \cos \theta) + \omega_z \sin \theta & \cos \theta + \omega_y^2(1 - \cos \theta) & \omega_y \omega_z(1 - \cos \theta) - \omega_x \sin \theta \\ \omega_z \omega_x(1 - \cos \theta) - \omega_y \sin \theta & \omega_z \omega_y(1 - \cos \theta) + \omega_x \sin \theta & \cos \theta + \omega_z^2(1 - \cos \theta) \end{bmatrix}$$

- That's a **mess**, but everybody's **graphics library** has subroutines for making rotation matrices for you
- It's also ripe for a lot of simplification
  - Precompute  $\sin \theta, \cos \theta, 1 - \cos \theta$

# Euler-Rodriguez rotation formula

- Through the liberal application of **trig identities** that the rest of us learned in high school and forgot, **Euler simplified the matrix to this form:**

$$M_{\omega, \theta} = \begin{bmatrix} a^2 + b^2 - c^2 - d^2 & 2(bc - ad) & 2(bc + ac) \\ 2(bc + ad) & a^2 + c^2 - b^2 - d^2 & 2(cd - ab) \\ 2(bd - ac) & 2(cd + ab) & a^2 + d^2 - b^2 - c^2 \end{bmatrix}$$

- Where

- $a = \cos \frac{\theta}{2}$
- $b = \omega_x \sin \frac{\theta}{2}$
- $c = \omega_y \sin \frac{\theta}{2}$
- $d = \omega_z \sin \frac{\theta}{2}$
- $a^2 + b^2 + c^2 + d^2 = 1$

# Composition in Euler-Rodriguez form

- This gives us a way of **representing a rotation with just four numbers**:  $a, b, c$ , and  $d$
- Moreover, they proved that the **composition of two rotations** represented as  $a_1, b_1, c_1, d_1$  and  $a_2, b_2, c_2, d_2$  has the parameters:
  - $a = a_1a_2 - b_1b_2 - c_1c_2 - d_1d_2$
  - $b = a_1b_2 + b_1a_2 - c_1d_2 + d_1c_2$
  - $c = a_1c_2 + c_1a_2 - d_1b_2 + b_1d_2$
  - $d = a_1d_2 + d_1a_2 - b_1c_2 + c_1b_2$(yuk)
- This is good preparation for talking about **quaternions**

part 2:  
**imaginary  
representations**

# Development of number systems

- Imaginary numbers were developed in Europe
- Kind of shocking, since the West was backward mathematically at that time
- Took an extra 1500+ years to accept negative numbers
  - Indians and Chinese developed negative numbers independently somewhere in the 200BC-400AD range
  - Spread from India to the Islamic world in the 8<sup>th</sup> century
  - Spread from there to Europe
  - Largely accepted in 17<sup>th</sup> century
  - Although still resisted by many mathematicians in the 18<sup>th</sup> century



# Solution to quadratic equations

- The quadratic formula
  - Includes the **square root of a negative** number
  - When asking for the roots of a parabola
  - That **don't actually intersect Y=0**
- Two possible interpretations
  - a) Those equations **don't have roots** and so negative numbers **don't have square roots**
  - b) They do but they're a weird different kind of number
- At first, (a) won

$$ax^2 + bx + c = 0$$

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

# What about cubics?

Cubics are more complicated

- Also have **closed-form solutions** for their roots
  - But they're **messy**
- **Cardano** published formulas for the different cases in 1545
- His formula for

$$x^3 = px + q$$

Turned out to be problematic

(Note: it's in this weird form rather than  $x^3 + px + q = 0$  because people still weren't sure they believed in negative numbers, so they wanted p and q greater than zero)

$$x^3 = px + q$$
$$x = \sqrt[3]{\frac{1}{2}q + w}$$

where:

$$w = \sqrt{\left(\frac{1}{2}q\right)^2 - \left(\frac{1}{3}p\right)^2}$$

(Trust your ancestors that they did the proof right)

# What about cubics?

Here's the **problem**. The polynomial:

$$x^3 = 15x + 4$$

has the solution:

$$x = 4$$

But **Cardano's formula** gives:

$$x = \sqrt[3]{2 + \sqrt{-121}} + \sqrt[3]{2 - \sqrt{-121}}$$

ouch

# Embracing the imaginary

**Bombelli** showed (1572) that if you assume

$$\sqrt[3]{2 + \sqrt{-121}} = a + bi$$

For some  $a, b$ , then

$$\sqrt[3]{2 - \sqrt{-121}} = a - bi$$

So:

$$\begin{aligned}x &= 4 = a + bi + a - bi = 2a \\a &= 2\end{aligned}$$

- So **imaginary numbers are needed** to solve even for some non-imaginary roots

- And indeed, if we plug in  $a = 2, b = 1$ , we get
$$\begin{aligned}(a + bi)^3 &= (2 + i)^3 \\&= 2 + 11i \\&= 2 + 11\sqrt{-1} \\&= 2 + \sqrt{-121} \\(a - bi)^3 &= 2 - \sqrt{-121}\end{aligned}$$

# Complex numbers

Assume

- $i = \sqrt{-1}$  exists
- Arithmetic works on it normally
  - Associativity
  - Commutativity
  - Distributivity

Then

$$\begin{aligned}(a + bi) + (c + di) &= (a + c) + (b + d)i \\(a + bi)(c + di) &= (ac) + bci + adi + bdi^2 \\&= (\mathbf{ac} - \mathbf{bd}) + (bc + ad)i\end{aligned}$$

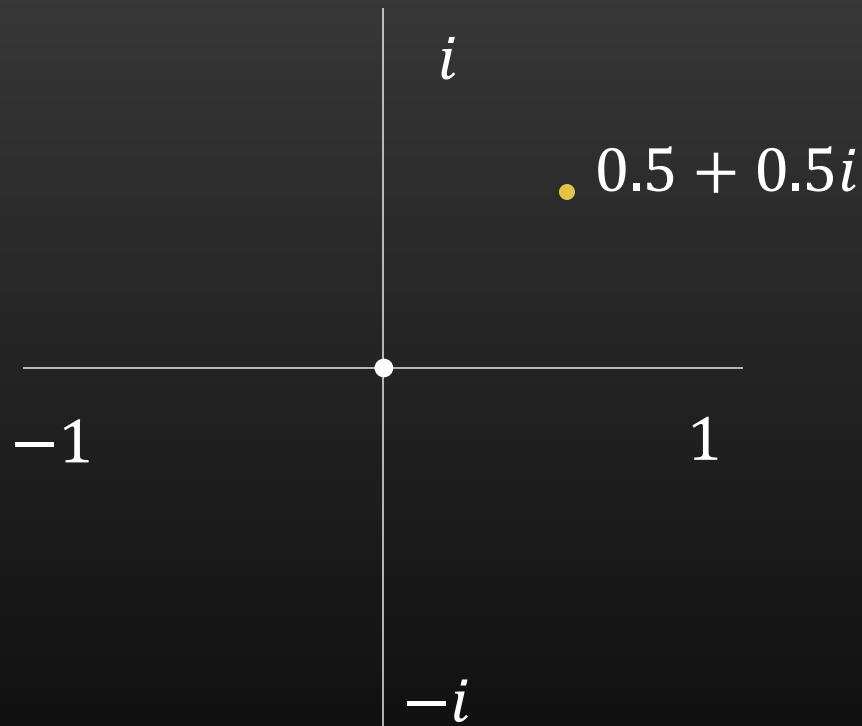
# Complex numbers

In general

- Any series of mathematical operations
  - On real and/or complex numbers
- Can be represented as  
 $a + bi$

# The complex plane

- We can **visualize** complex numbers as a 2D space
- With the axes
  - $\pm 1$
  - And  $\pm i$



# Geometry of complex numbers

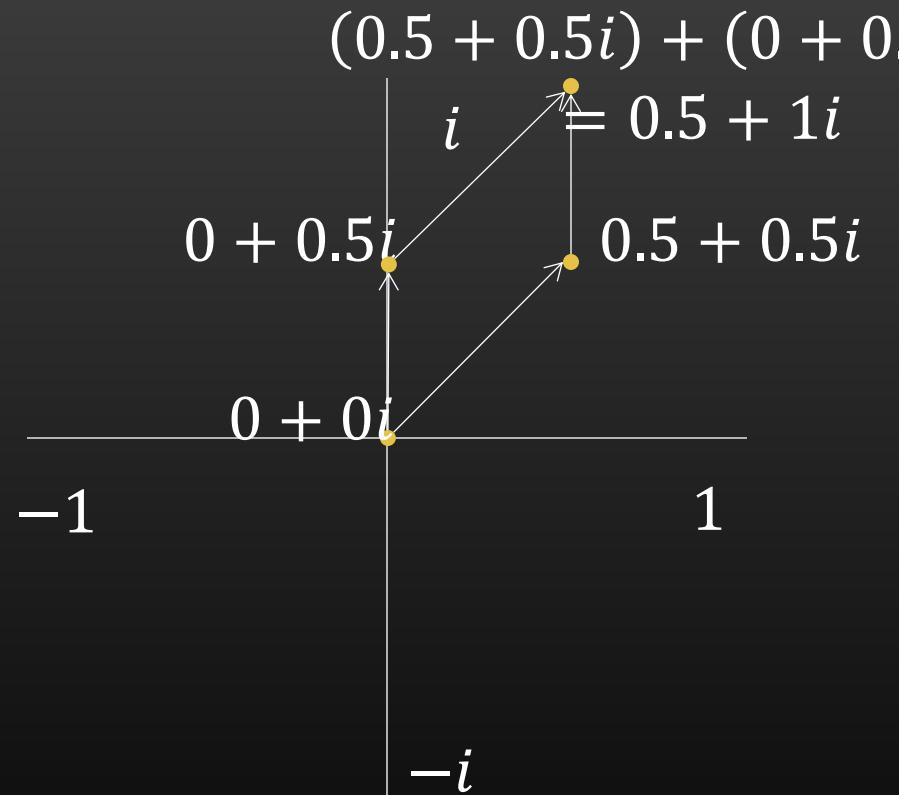
- Then **addition is vector addition**

Complex numbers:

$$(a + bi) + (c + di) \\ = (a + c) + (b + d)i$$

Vectors:

$$(a, b) + (c, d) \\ = (a + c, b + d)$$

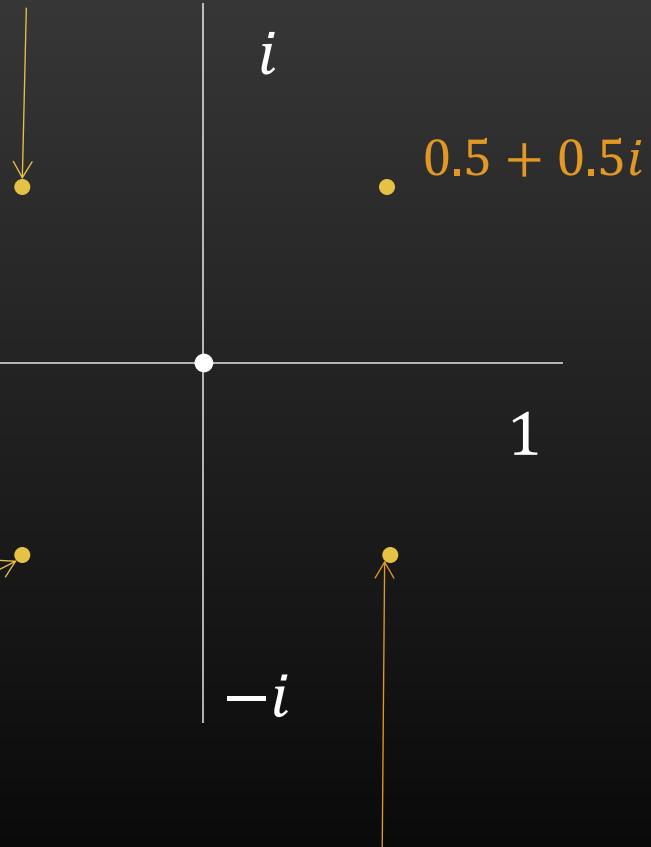


# Multiplication as rotation

- People quickly realized that **multiplying by  $i$  rotated** a number by 90 degrees
- And, in fact:

- Multiplying by  $\sqrt{i} = \frac{1+i}{\sqrt{2}}$  rotates by 45 degrees
- Multiplying by  $\sqrt[3]{i}$  rotates by 30 degrees

$$i(0.5 + 0.5i) = 0.5i^2 + 0.5i \\ = -0.5 + 0.5i$$



$$i^2(0.5 + 0.5i) = -1(0.5 + 0.5i) \\ = -0.5 - 0.5i$$

$$i^3(0.5 + 0.5i) = -i(0.5 + 0.5i) \\ = 0.5 - 0.5i$$

# Polar coordinates

We can **represent** numbers as

- A **magnitude** (distance from the origin)

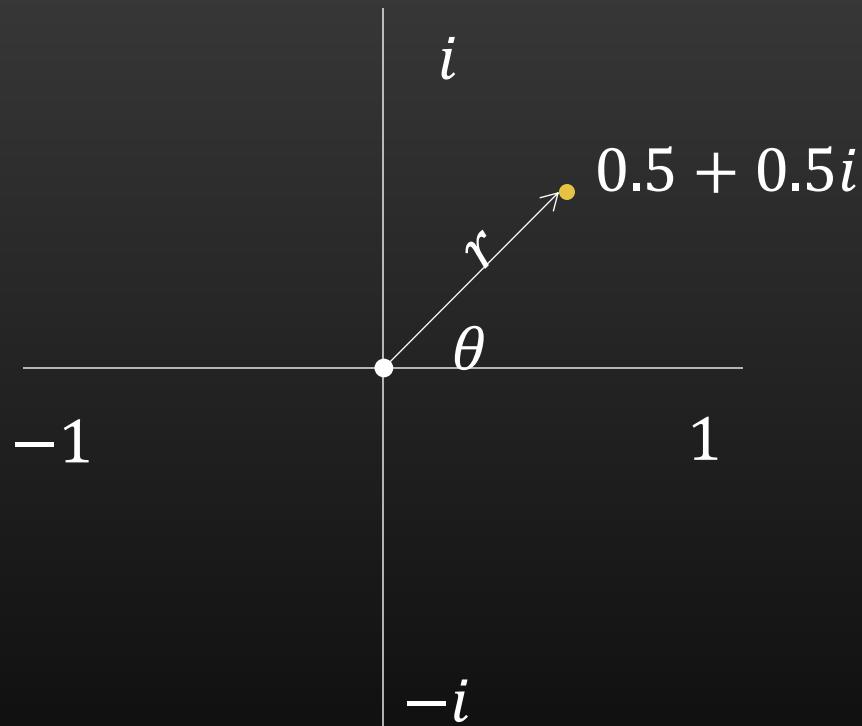
$$|a + bi| = \sqrt{a^2 + b^2}$$

- An **angle** from the real axis

$$\theta(a + bi) = \text{atan2}(a, b)$$

Then **multiplication**

- Adds angles
- Multiplies magnitudes



# Unit complex numbers

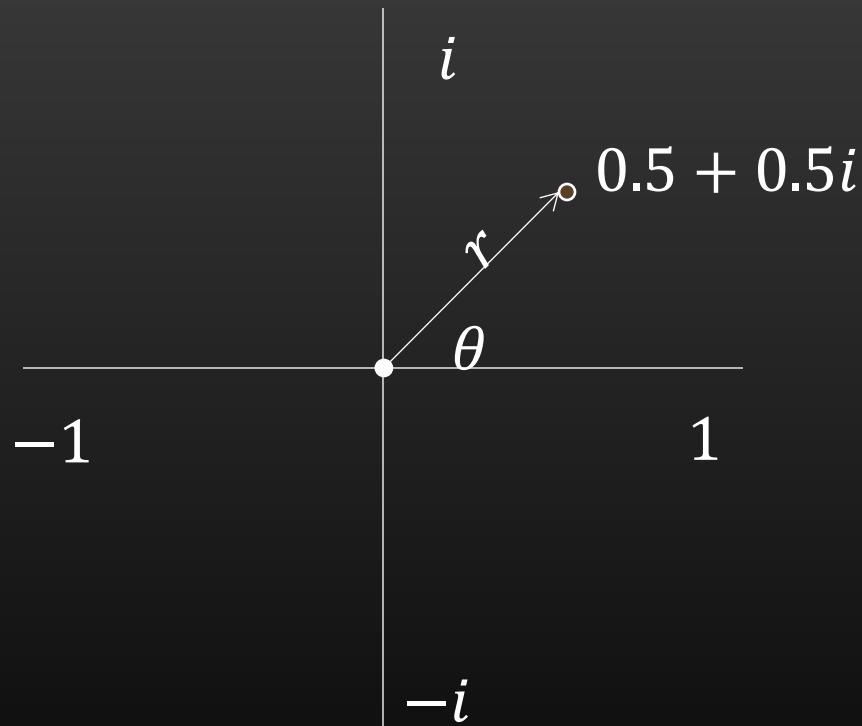
# Representing rotations as unit magnitude numbers

So we can **represent any rotation** about the origin as

- A **multiplication**
- By number with a **magnitude of 1**

Awesome

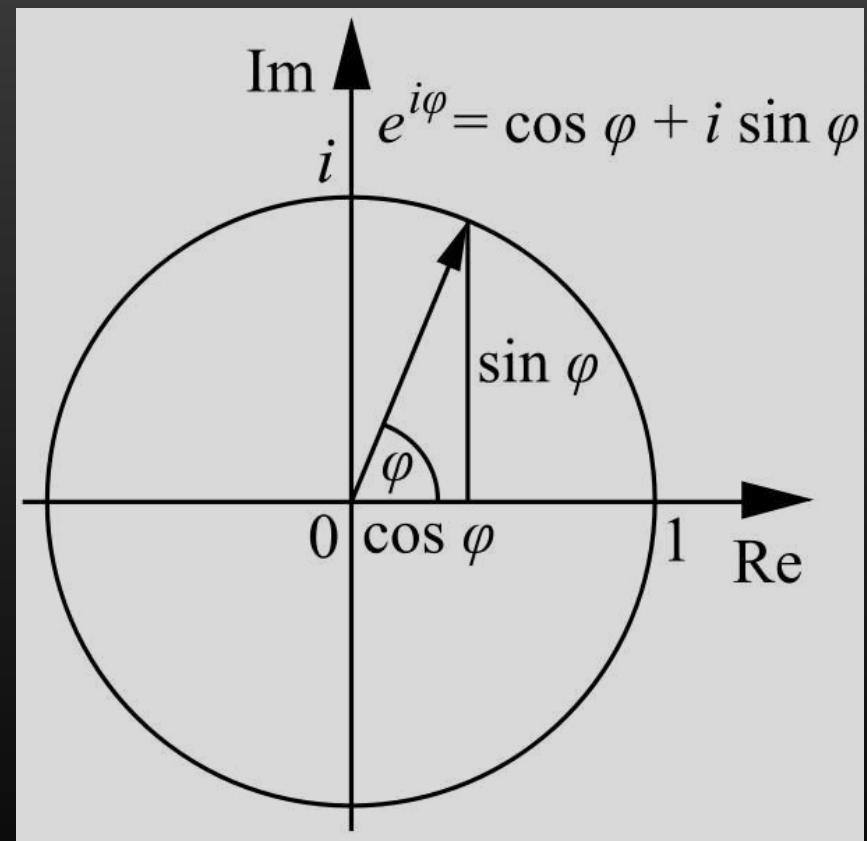
Uh, **where** do we get these magic unit magnitude numbers?



# Euler's formula

Euler showed that

- When raising a real number to a **complex power  $a + bi$** 
  - The **magnitude** was scaled by  **$a$**
  - The **angle** was rotated by  **$b$**
- We can get **unit numbers with a given angle** by raising  $e$  to a complex power
  - $e^{i\phi}$  is a number that rotates by  **$\phi$  radians**



# Why is Euler's formula true?

**Taylor series** for  $e^x$ :

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

Taylor series for  $\sin$  and  $\cos$ :

$$\begin{aligned}\sin x &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots \\ \cos x &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots\end{aligned}$$

Just plug in the values:

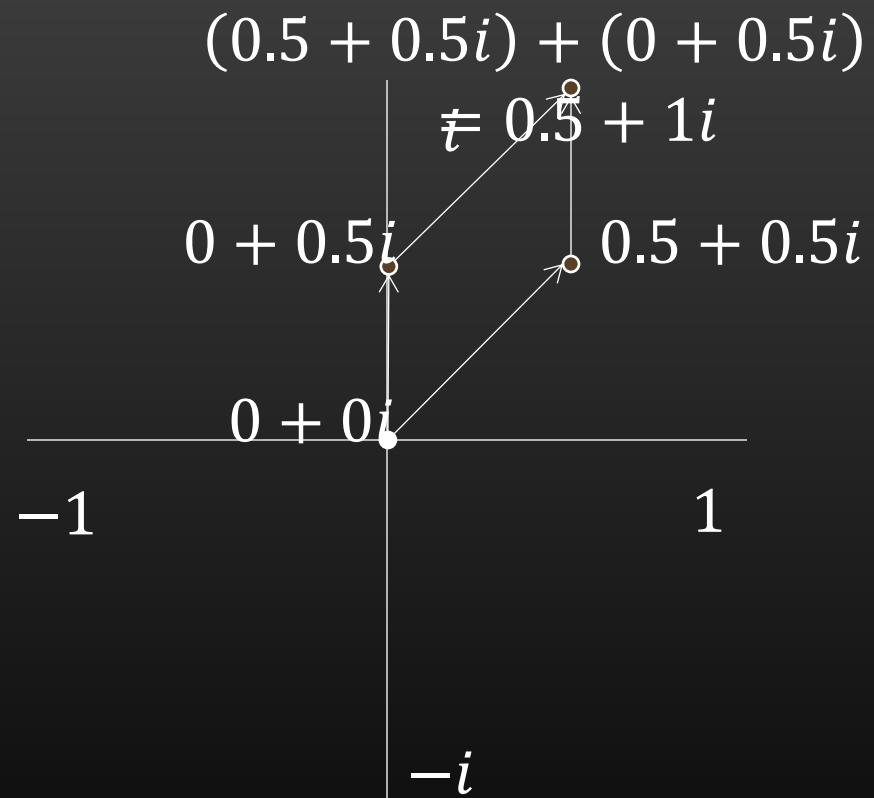
$$\begin{aligned}e^{i\phi} &= 1 + i\phi + \frac{(i\phi)^2}{2!} + \frac{(i\phi)^3}{3!} + \dots \\ &= 1 + i\phi + \frac{i^2 \phi^2}{2!} + \frac{i^3 \phi^3}{3!} + \dots \\ &= 1 + i\phi - \frac{\phi^2}{2!} - i \frac{\phi^3}{3!} + \dots \\ &= \left(1 - \frac{\phi^2}{2!} + \frac{\phi^4}{4!} + \dots\right) \\ &\quad + i \left(\phi - \frac{\phi^3}{3!} + \frac{\phi^5}{5!} + \dots\right) \\ &= \cos \phi + i \sin \phi\end{aligned}$$

# Complex numbers and geometry

So **complex numbers** let us perform

- **Translations** using addition
- **Scaling** using multiplication by a real
- And **rotation** using multiplication by a complex number

... but **only in 2D** ...



# What about 3D?

- The race was on to represent 3-space using complex numbers
- One obvious approach was to assume -1 had two square roots
- But this had a number of problems

# Quaternions

# Quaternions

- Eventually Hamilton developed **quaternions**
- Based on the assumption that  $-1$  has **3 linearly independent square roots**,  $i, j$ , and  $k$
- So quaternions are **linear combinations** of the **reals** and these **basis elements**:

$$q = a + bi + cj + dk$$

- Quaternion **addition** is what you'd expect
$$(a_1 + b_1 i + c_1 j + d_1 k) + (a_2 + b_2 i + c_2 j + d_2 k) \\ = (a_1 + a_2) + (b_1 + b_2)i + (c_1 + c_2)j + (d_1 + d_2)k$$

# Quaternion multiplication

Quaternion **multiplication is complicated**

- Multiplication of **real components** behaves normally
- Multiplication of the **imaginary components** is **anticommutative**

$$AB = -BA$$

Quaternion axiom:

$$i^2 = j^2 = k^2 = ijk = -1$$

From this you can prove that:

- $ij = k$
- $jk = i$
- $ki = j$

And by anticommutativity:

- $ji = -(ij) = -k$
- $kj = -(jk) = -i$
- $ik = -(ki) = -j$

# Quaternion multiplication

This **table**, plus the normal rules of **associativity and distributivity** are enough to let us compute the product of any two quaternions

<b>x</b>	<b>1</b>	<b>i</b>	<b>j</b>	<b>k</b>
<b>1</b>	1	<i>i</i>	<i>j</i>	<i>k</i>
<i>i</i>	<i>i</i>	-1	<i>k</i>	- <i>j</i>
<i>j</i>	<i>j</i>	- <i>k</i>	-1	<i>i</i>
<b>k</b>	<i>k</i>	<i>j</i>	- <i>i</i>	-1

# Quaternion multiplication

This gives us the **Hamiltonian product** of two quaternions:

$$\begin{aligned}(a_1 + b_1i + c_1j + d_1k)(a_2 + b_2i + c_2j + d_2k) \\&= a_1a_2 + a_1b_2i + a_1c_2j + a_1d_2k \\&\quad + b_1a_2i + b_1b_2i^2 + b_1c_2ij + b_1d_2ik \\&\quad + c_1a_2j + c_1b_2ji + c_1c_2j^2 + c_1d_2jk \\&\quad + d_1a_2k + d_1b_2ki + d_1c_2kj + d_1d_2k^2 \\&= \mathbf{a}_1\mathbf{a}_2 - \mathbf{b}_1\mathbf{b}_2 - \mathbf{c}_1\mathbf{c}_2 - \mathbf{d}_1\mathbf{d}_2 \\&\quad + (\mathbf{a}_1\mathbf{b}_2 + \mathbf{b}_1\mathbf{a}_2 + \mathbf{c}_1\mathbf{d}_2 - \mathbf{d}_1\mathbf{c}_2)\mathbf{i} \\&\quad + (\mathbf{a}_1\mathbf{c}_2 - \mathbf{b}_1\mathbf{d}_2 + \mathbf{c}_1\mathbf{a}_2 + \mathbf{d}_1\mathbf{b}_2)\mathbf{j} \\&\quad + (\mathbf{a}_1\mathbf{d}_2 + \mathbf{b}_1\mathbf{c}_2 - \mathbf{c}_1\mathbf{b}_2 + \mathbf{d}_1\mathbf{a}_2)\mathbf{k}\end{aligned}$$

(yuk)

# Composition in Euler-Rodriguez form

- This gives us a way of **representing a rotation with just four numbers**:  $a, b, c$ , and  $d$
- Moreover, they proved that the **composition of two rotations** represented as  $a_1, b_1, c_1, d_1$  and  $a_2, b_2, c_2, d_2$  has the parameters:
  - $a = a_1a_2 - b_1b_2 - c_1c_2 - d_1d_2$
  - $b = a_1b_2 + b_1a_2 - c_1d_2 + d_1c_2$
  - $c = a_1c_2 + c_1a_2 - d_1b_2 + b_1d_2$
  - $d = a_1d_2 + d_1a_2 - b_1c_2 + c_1b_2$(yuk)
- Quaternion multiplication **just is** the composition rule for the Euler-Rodriguez form

# Vectors as quaternions

# Geometric interpretation

Consider two **purely imaginary** quaternions (real part is zero):

- $p = b_1i + c_1j + d_1k$ ,
- $q = b_2i + c_2j + d_2k$

Then we have that:

$$\begin{aligned} pq &= (b_1i + c_1j + d_1k)(b_2i + c_2j + d_2k) \\ &= -\mathbf{b}_1\mathbf{b}_2 - \mathbf{c}_1\mathbf{c}_2 - \mathbf{d}_1\mathbf{d}_2 \\ &\quad + (\mathbf{c}_1\mathbf{d}_2 - \mathbf{d}_1\mathbf{c}_2)\mathbf{i} + (\mathbf{d}_1\mathbf{b}_2 - \mathbf{b}_1\mathbf{d}_2)\mathbf{j} + (\mathbf{b}_1\mathbf{c}_2 - \mathbf{c}_1\mathbf{b}_2)\mathbf{k} \end{aligned}$$

Look **familiar**?

# Geometric interpretation

What if we think of  $p$  and  $q$  as **vectors**?

- $\vec{p}_v = (b_1, c_1, d_1)$
- $\vec{q}_v = (b_2, c_2, d_2)$

Then we have that:

$$\begin{aligned} pq &= (b_1i + c_1j + d_1k)(b_2i + c_2j + d_2k) \\ &= -\mathbf{b}_1\mathbf{b}_2 - \mathbf{c}_1\mathbf{c}_2 - \mathbf{d}_1\mathbf{d}_2 + (\mathbf{c}_1\mathbf{d}_2 - \mathbf{d}_1\mathbf{c}_2)\mathbf{i} \\ &\quad + (\mathbf{d}_1\mathbf{b}_2 - \mathbf{b}_1\mathbf{d}_2)\mathbf{j} + (\mathbf{b}_1\mathbf{c}_2 - \mathbf{c}_1\mathbf{b}_2)\mathbf{k} \\ &\approx -(\vec{p}_v \cdot \vec{q}_v) + \vec{p}_v \wedge \vec{q}_v \end{aligned}$$

# Quaternions and 3-space

We can think quaternion:

$$q = a + bi + cj + dk$$

as a **weird hybrid number-vector**

- A **scalar** (real) part  $q_s$  (in this case,  $a$ )
- Plus an **imaginary vector**  $\vec{q}_v$  (in this case,  $(b, c, d)$ )

So it's already kind of like an **axis-angle** representation

Given two quaternions:

- $p = p_s + \vec{p}_v$
- $q = q_s + \vec{q}_v$

We can compute their product in **vector form** as:

$$\begin{aligned} pq = & p_s q_s - \vec{p}_v \cdot \vec{q}_v + \\ & p_s \vec{q}_v + q_s \vec{p}_v + \vec{p}_v \times \vec{q}_v \end{aligned}$$

(This will eventually turn out to be useful...)

# Rotation using quaternions

# Three last annoying definitions

Let

$$\begin{aligned} q &= a + bi + cj + dk \\ &= q_s + \vec{q}_v \end{aligned}$$

be a quaternion

The **conjugate** of  $q$  is defined as:

$$\begin{aligned} q^* &= a - bi - cj - dk \\ &= -\frac{1}{2}(q + iqi + jqj + kqk) \end{aligned}$$

The **magnitude** of  $q$  is defined as:

$$\begin{aligned} \|q\| &= \sqrt{qq^*} = \sqrt{a^2 + b^2 + c^2 + d^2} \\ &= \sqrt{q_s^2 + \vec{q}_v^2} \end{aligned}$$

The **inverse** (reciprocal) of  $q$  is then just:

$$q^{-1} = \frac{q^*}{\|q\|^2}$$

So that:

$$qq^{-1} = q^{-1}q = 1$$

# Polar decomposition

- A **unit quaternion**, like a unit vector is a quaternion with a magnitude of 1
- Any quaternion can be made into a unit quaternion by **dividing by its magnitude**
- Any quaternion can be represented in **polar form** as the product of its magnitude and unit quaternion

$$q = \|q\|U_q$$

$$U_q = \frac{q}{\|q\|}$$

# Unit quaternions as rotations

- Unit quaternions correspond to rotations in 3-space
  - Just as unit complex numbers correspond to rotations in 2-space
  - The rotation of a vector by a unit quaternion is given by:  
$$\text{rotate}(q, \vec{v}) = q\vec{v}q^{-1} = q\vec{v}q^*$$
  - If you multiply this out, you get the matrix form of Euler-Rodriguez
- The unit quaternion:  
$$q_{\vec{\omega}, \theta} = \cos \frac{\theta}{2} + \sin \frac{\theta}{2} \vec{\omega}$$
Performs a rotation of:
    - $\theta$  radians
    - About axis  $\vec{\omega}$
  - Multiplication of unit quaternions corresponds to composition of rotations
    - $pq$  does the equivalent of
    - First rotating by  $q$ ,
    - Then rotating by  $p$

# Why is this good?

- **No gimbal lock**
- The **extra degree of freedom** in quaternions
  - Let's us represent 3D rotations **without discontinuities**
  - Just as the extra degree in complex numbers lets us represent 2D rotations
  - Because they're kind of **projective**
- **Compositionality**
  - Multiplication = composition
    - $pq$  does  $q$  then  $p$
  - Exponentiation = iteration
    - $q^n$  does  $q$ ,  $n$  times
    - $\sqrt{q} = q^{\frac{1}{2}}$  does  $q$  halfway
- **Interpolation**
  - $q^t$ ,  $0 \leq t \leq 1$ , does  $q$ ,  **$t$  percent of the way**

Slerp

# Spherical linear interpolation

- Suppose we want to do **animation**
- We want our object
  - To **start at rotation  $q_0$**  at time  $t = 0$
  - **End at rotation  $q_1$**  at time  $t = 1$
- What's an **expression for the rotation** at time  $t \in [0,1]$ ?
- Between  $q_0$  and  $q_1$  it's going to go through some rotation
- What rotation?
  - The rotation that when multiplied by  $q_0$  gives us  $q_1$
  - So it's  $q_1 q_0^{-1}$ 
    - Since  $(q_1 q_0^{-1})q_0 = q_1(q_0^{-1}q_0) = q_1 1 = q_1$
- So at time  $t$ , we want to
  - Start with  $q_0$
  - Then do  $t\%$  of  $q_1 q_0^{-1}$ 
    - Which would be  $(q_1 q_0^{-1})^t$
- So the correct rotation is
$$\text{Slerp}(q_0, q_1, t) = (q_1 q_0^{-1})^t q_0$$
- “**Slerp**” means “**Spherical Linear intERPolation**”

**computing  $(q_1 q_0^{-1})^t$**  is  
a pain

Actually, you don't compute Slerp this way in practice

# Finding powers of quaternions

- Let's start by understanding how to raise  $e$  to a quaternion power

- Let  $q = q_s + \vec{q}_v$

- We have that

$$e^q = e^{q_s + \vec{q}_v} = e^{q_s} e^{\vec{q}_v}$$

- Great! What's  $e^{\vec{q}_v}$ ?

- Take the **Taylor series**

$$e^{\vec{q}_v} = \sum_{n=0}^{\infty} \frac{(\vec{q}_v)^n}{n!}$$

- The algebra is a lot trickier than with Euler's formula, but we end up with a similar result:

$$e^{\vec{q}_v}$$

$$= \cos\|\vec{q}_v\| + \frac{\vec{q}_v}{\|\vec{q}_v\|} \sin\|\vec{q}_v\|$$

- Again, it's a **cosine real part** and a **sine times an imaginary part**

# Finding powers of quaternions

By similar analysis, we can show that

$$\begin{aligned}\ln q &= \ln q_s + \vec{q}_v \\ &= \ln \|q\| + \frac{\vec{q}_v}{\|\vec{q}_v\|} \cos^{-1} \frac{q_s}{\|\vec{q}_v\|}\end{aligned}$$

And so now we can use

$$q^t = e^{t \ln q}$$

To compute powers of quaternions

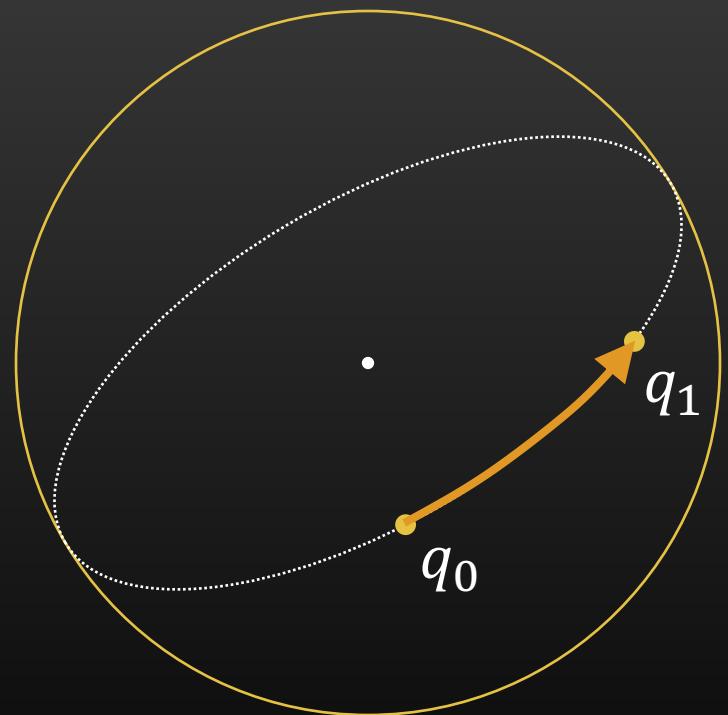
(**yuk!**)

# How it really works

- Slerp's **primary features** are that
  - It moves you along a **minimum torque path**
    - The path that would require the least possible energy
    - Trust the physicists that this is true
  - It moves through it at a **constant angular speed**
    - So it doesn't speed up and slow down in surprising ways
- Let's think about **what that path is** in real life

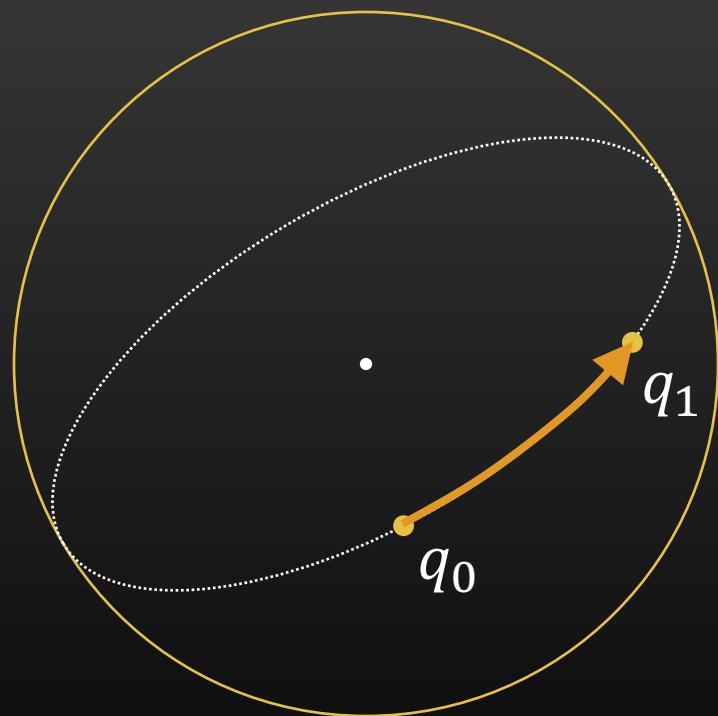
# How it really works

- Suppose we rotate a **single point on the unit sphere** from orientation  $q_0$  to  $q_1$
- If it's on the minimum torque path, it moves along a **great circle**
  - Aka a geodesic
  - Trust me
- And circles lie in **planes**



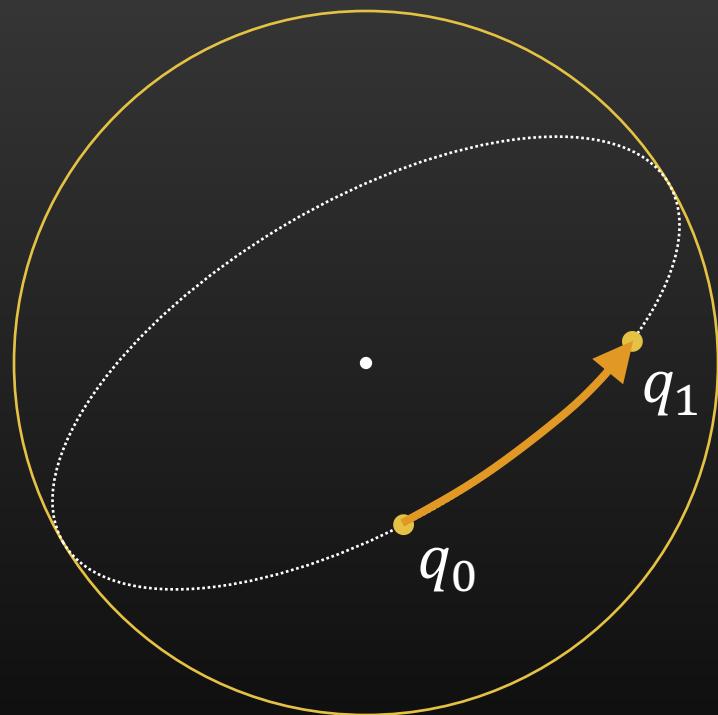
# How it really works

- We're representing rotations using **unit** quaternions
$$a^2 + b^2 + c^2 + d^2 = 1$$
- Think of the unit quaternion as a point in a **4-dimensional space** $(a, b, c, d)$
- Then the unit quaternions lie on a **hypersphere** in that 4-dimensional space



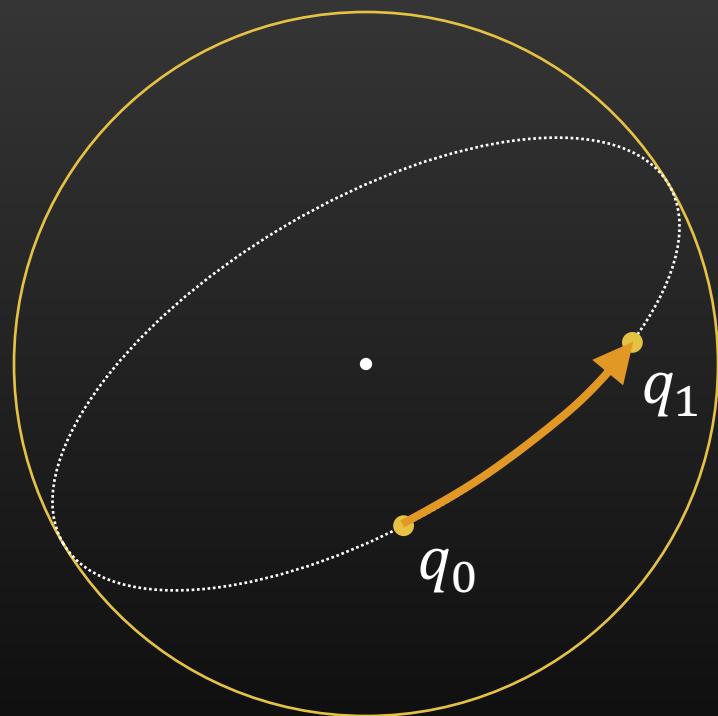
# How it really works

- It can be proven that
  - The path Slerp takes between two points also lies on a **great circle** of the hypersphere
  - It traces it at **constant angular velocity**
- That means it moves through a **plane**



# Why do we care?

- Any point on a plane can be represented as a **linear combination** of two vectors in the plane
- We have two vectors in the plane:  $q_0$  and  $q_1$
- So we know  $\text{Slerp}(q_0, q_1, t)$  is always a **linear combination** of  $q_0$  and  $q_1$



# Think about it in the plane

- Since the Slerp value is co-planar with  $q_0$  and  $q_1$ , we have that:

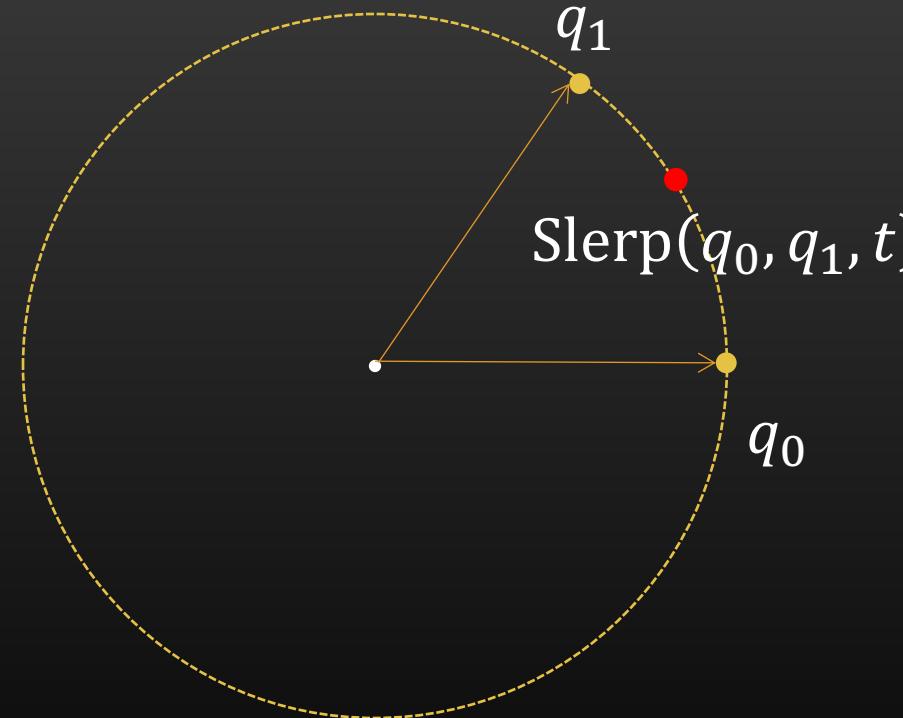
$$\text{Slerp}(q_0, q_1, t) = aq_0 + bq_1$$

For some  $a$  and  $b$

- We also know that it's a unit quaternion so:

$$\|aq_0 + bq_1\| = 1$$

- How do we compute points on a unit circle?

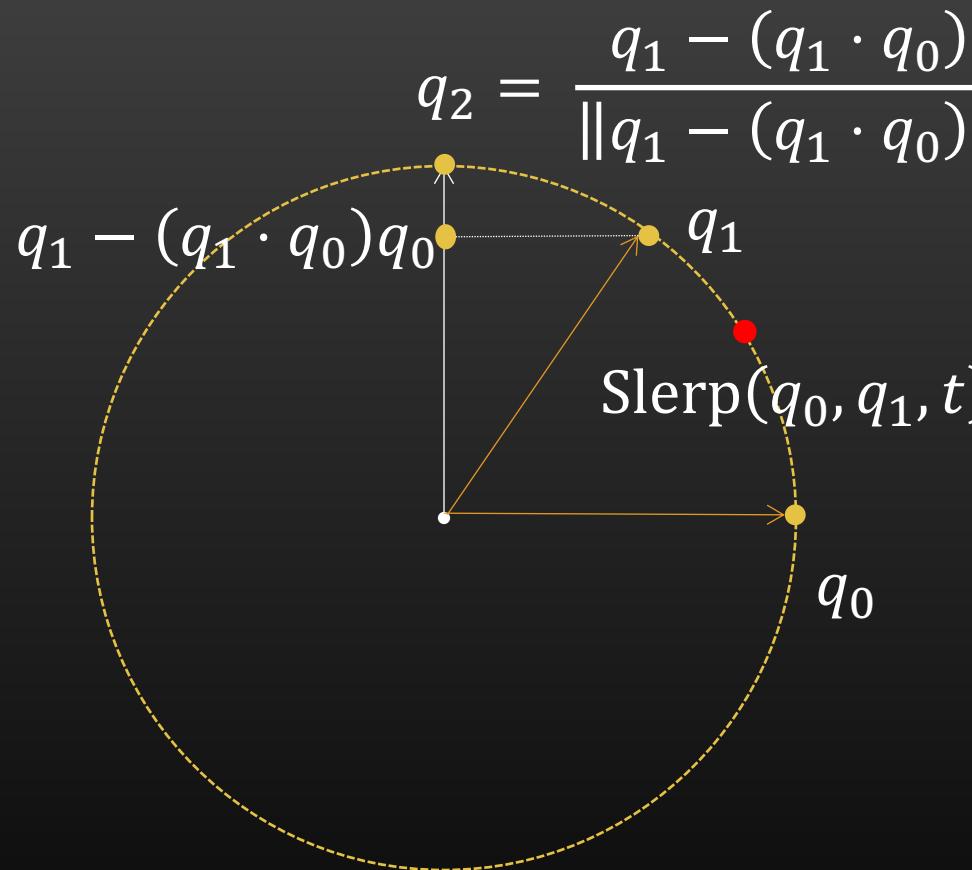


# Blow's construction

- Jonathan Blow (author of *Braid* and *The Witness*, among other great games) has an easy to understand construction for this
- Start out by using Gram-Schmidt to construct a new quaternion:

$$q_2 = \frac{q_1 - (q_1 \cdot q_0)q_0}{\|q_1 - (q_1 \cdot q_0)q_0\|}$$

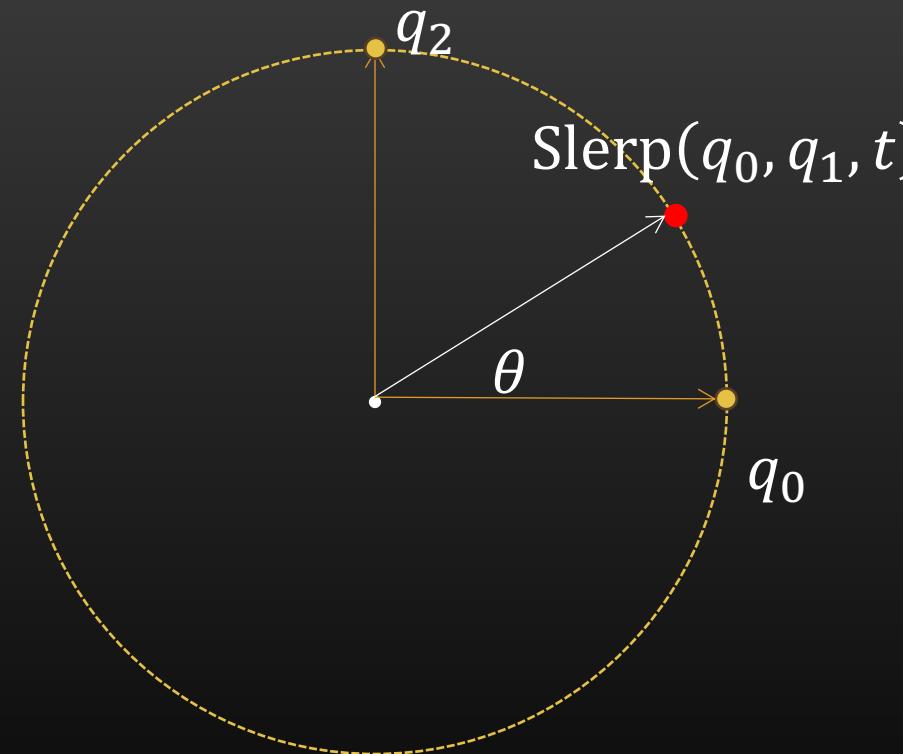
that's in the plane, but orthogonal to  $q_0$



# Blow's construction

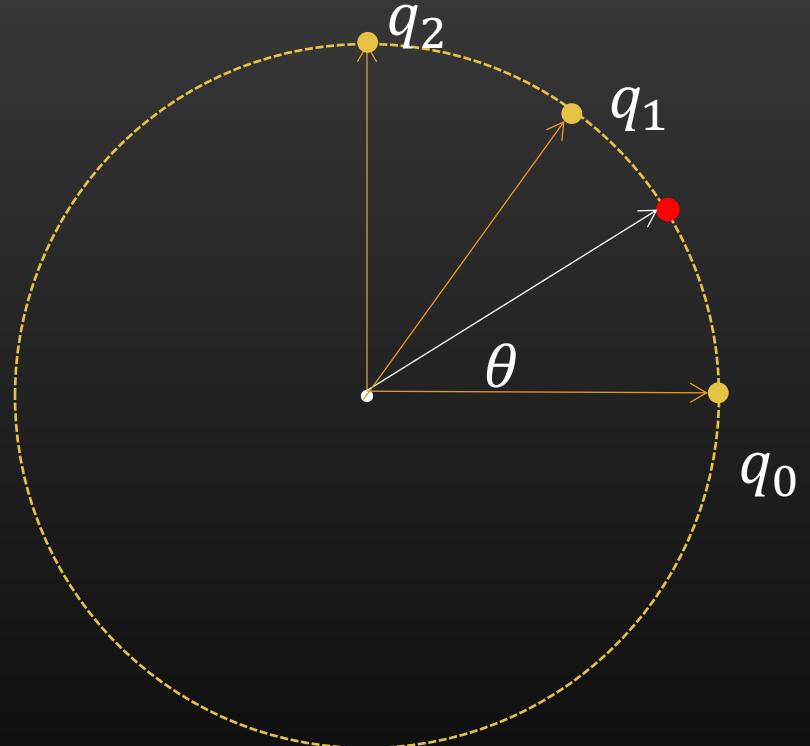
- Now we can construct any point on the circle with sines and cosines

$$\text{Slerp}(q_0, q_1, t) = q_0 \cos \theta + q_2 \sin \theta$$



# Blow's construction

- How do we find  $\theta$ ?
  - We start at  $o$
  - We go up whatever angle  $q_1$  makes with  $q_0$
- How do we know what angle *that* is?
  - We know its cosine is its dot product with  $q_1$
  - So the angle of  $q_0$  with  $q_1$  is  $\cos^{-1}(q_0 \cdot q_1)$
- So  $\theta = t \cos^{-1}(q_0 \cdot q_1)$



# Alternate implementation of slerp

**Jonathan Blow's formula**

$$\text{Slerp}(q_0, q_1, t) = \mathbf{q}_0 \cos t\theta + \mathbf{q}_2 \sin t\theta$$

where:

- $q_2 = \frac{q_1 - (q_0 \cdot q_1)q_0}{\|q_1 - (q_0 \cdot q_1)q_0\|}$
- $\theta = \cos^{-1}(q_0 \cdot q_1)$

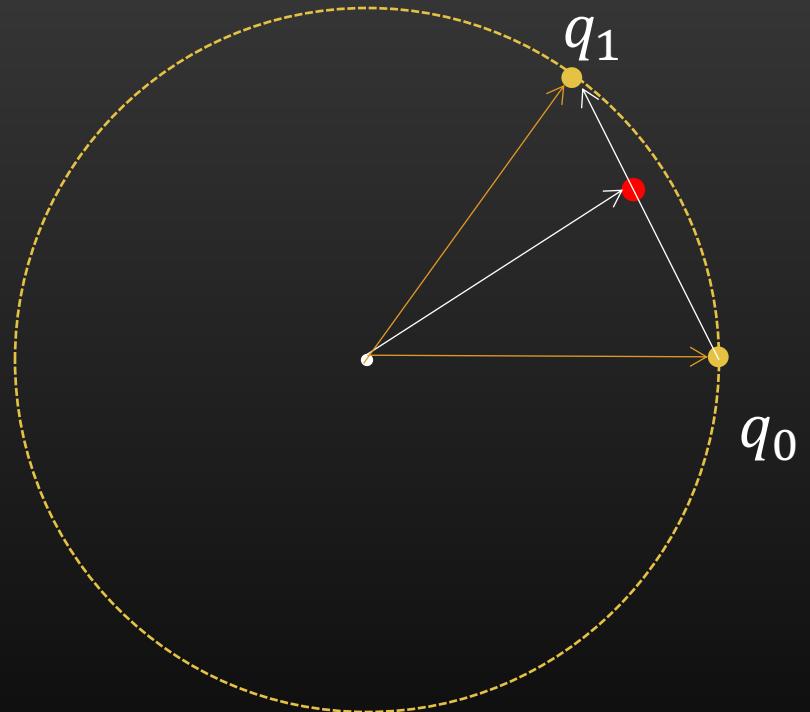
You can also use the **Shoemake/Davis** formula from Shoemake's original paper on using quaternions for animation:

$$\text{Slerp}(q_0, q_1, t) = \frac{\sin((1-t)\theta)}{\sin \theta} \mathbf{q}_0 + \frac{\sin t\theta}{\sin \theta} \mathbf{q}_1$$

Which is more efficient, but has a more complicated derivation

# Linear interpolation

- That's still kind of a **pain**
- What if we just did a **linear interpolation?**  
$$tq_1 + (1 - t)q_0$$
- Then we get a point that **isn't quite** on the circle
  - And isn't a unit quaternion, so it isn't a valid rotation

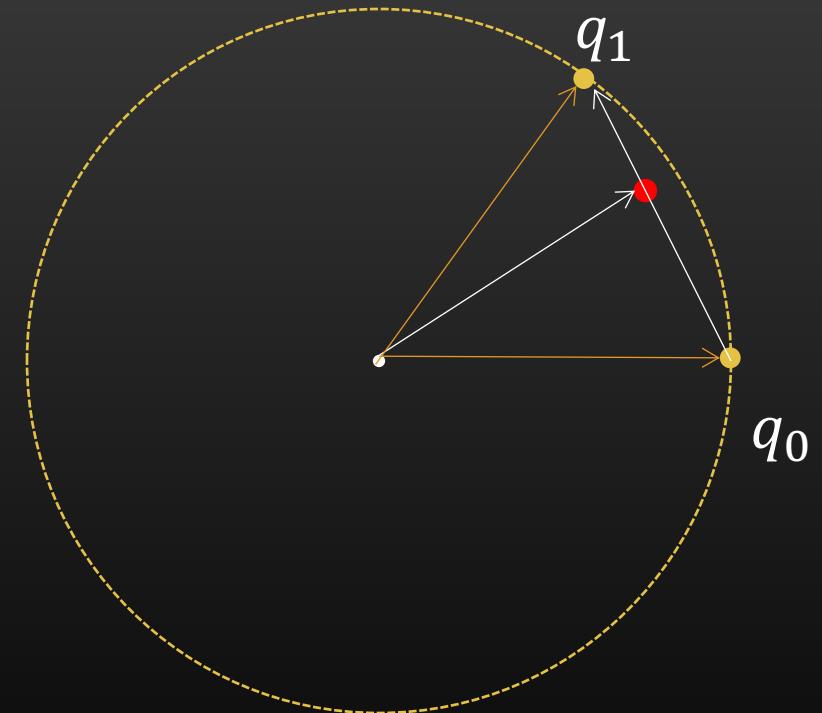


# Normalized linear interpolation

- But if we just **normalize** its magnitude:

$$\frac{tq_1 + (1 - t)q_0}{\|tq_1 + (1 - t)q_0\|}$$

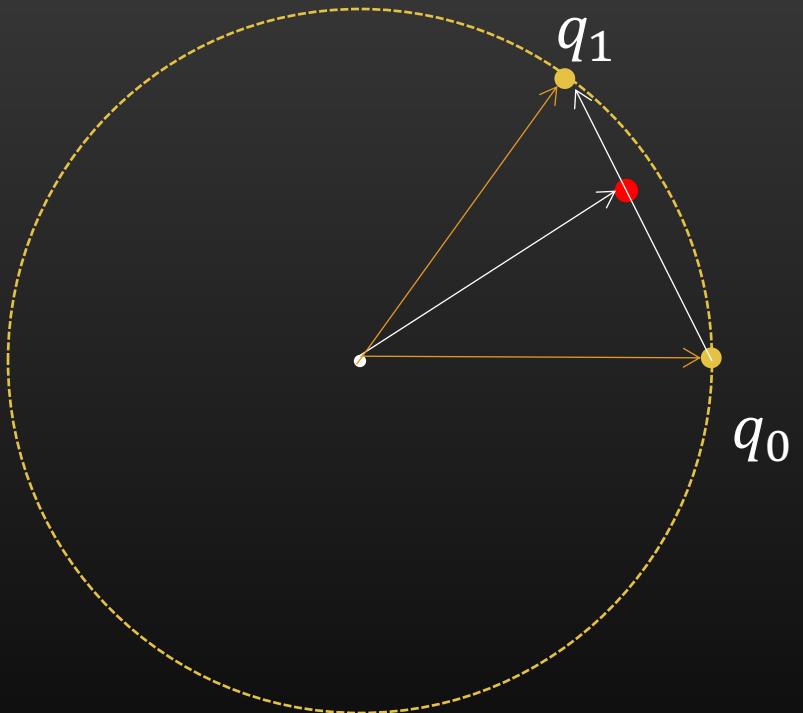
- Then we're **back on the circle**
  - And back to being a valid rotation
- This is called **normalized linear interpolation** or "nlerp"



# Nlerp

$$\text{nlerp}(q_0, q_1, t) = \frac{tq_1 + (1 - t)q_0}{\|tq_1 + (1 - t)q_0\|}$$

- Nlerp traces the **same path** as slerp
  - The only difference is it's **not quite constant speed**
- But it's **faster to compute**
  - And the variation in angular velocity is **minor** if the difference between  $q_0$  and  $q_1$  is small anyway
- So Blow has argued for using nlerp instead of slerp



# Further reading

- Gregory, chapter on math
- Jonathan Blow, Hacking Quaternions
- Jonathan Blow, Understanding Slerp, Then Not Using It
- Ken Shoemake, “Animating Rotation with Quaternion Curves”, Computer Graphics, Volume 19, Number 3, 1985
  - Original SIGGRAPH paper on quaternions for animation
  - Appendices include formulae for converting between quaternions, rotation matrices, and Euler angles

Modeling **shape**

# How do we represent the shape of an object?



# Eeeew

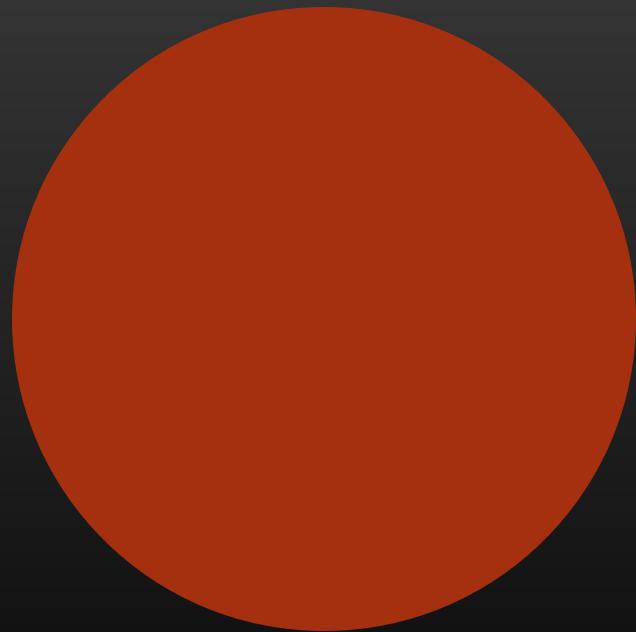
- We don't really need to represent all the details of an object
- Just **approximate** the shape
- Enough that it looks right when displayed



# What do we mean by shape?



# The interior of the shape?



# The boundary of the shape?



# Representation of shape

- In computer graphics, we *usually* use **boundaries**
  - **Curves** in 2D
  - **Surfaces** in 3D
- A curve or surface is an **infinite set** of points
- This is a problem, since our computers don't have infinite memory



# Approximating a curve

# Approximating a curve



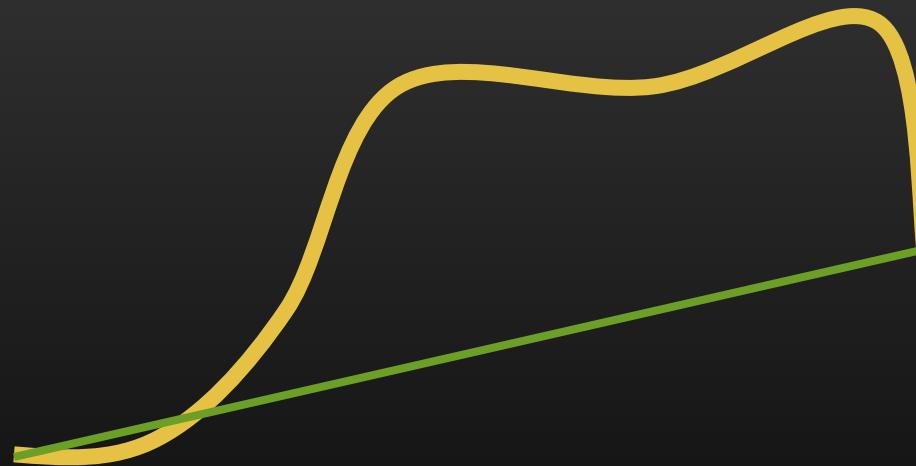
# Theorem

- For any function  $f: \mathbb{R}^n \rightarrow \mathbb{R}$ , and any input  $x_0$  for  $f$ , there is **at most one linear function**  $Df_{x_0}$  for which:
  - Let  $g(x + \Delta x) = f(x_0) + Df_{x_0}(\Delta x)$
  - $|f(x_0 + \Delta x) - g(x_0 + \Delta x)| = o(\Delta x^2)$
- Translation: there's at **most one linear function** that approximates a given function at a point with **better than linear error**

# Taylor polynomials

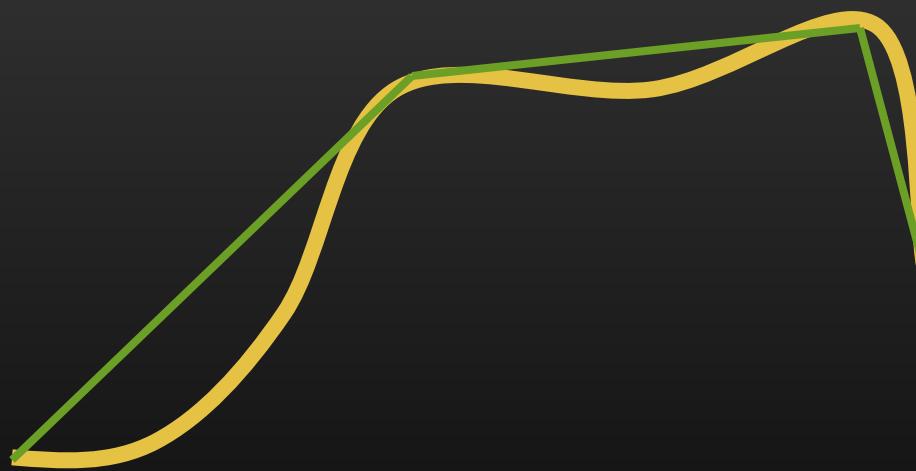
- And of course, you can do that with fancier polynomials
- A Taylor polynomial is just a **truncated Taylor series**:
  - $f(x_0 + \Delta x) \approx f(x_0) + f'(x_0)\Delta x + \frac{1}{n!}f^n(x_0)\Delta x^n$
- An nth-order Taylor polynomial approximates a function with an **error of  $o(\Delta x^{n+1})$**
- So, again, we can think of derivatives as being about how polynomials approximate functions

# Linear approximation



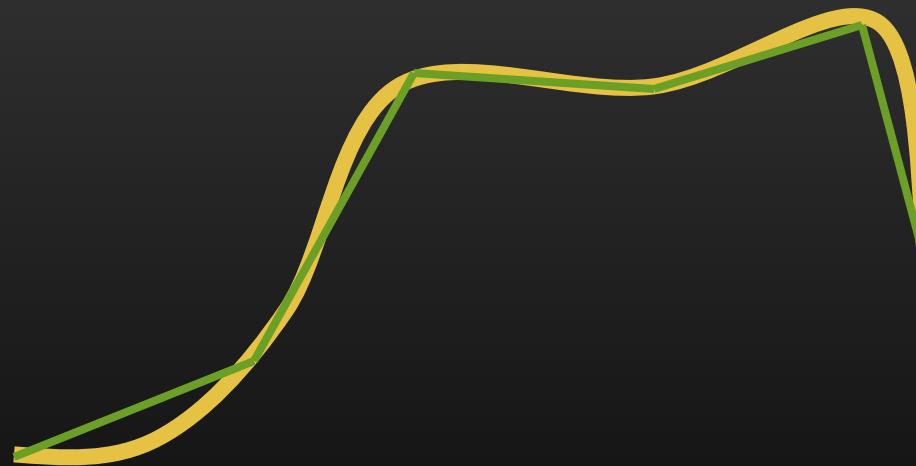
We can use a line as a (bad) approximation to a curve

# Piecewise-linear approximation



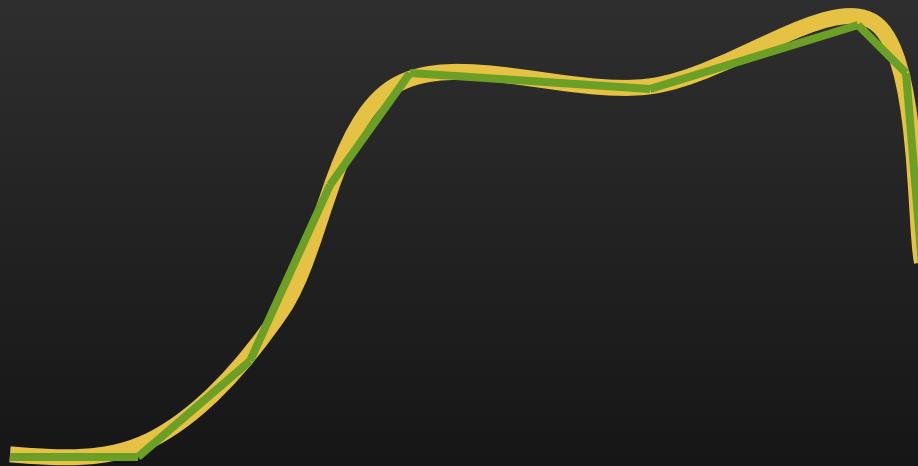
We can use several lines to make a better approximation

# Approximating a curve



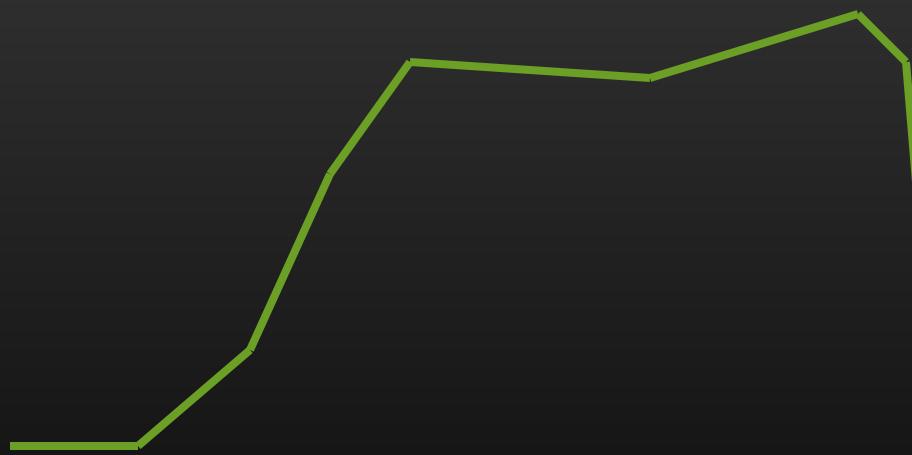
By adding more lines, we can make it as accurate as we like

# Approximating a curve



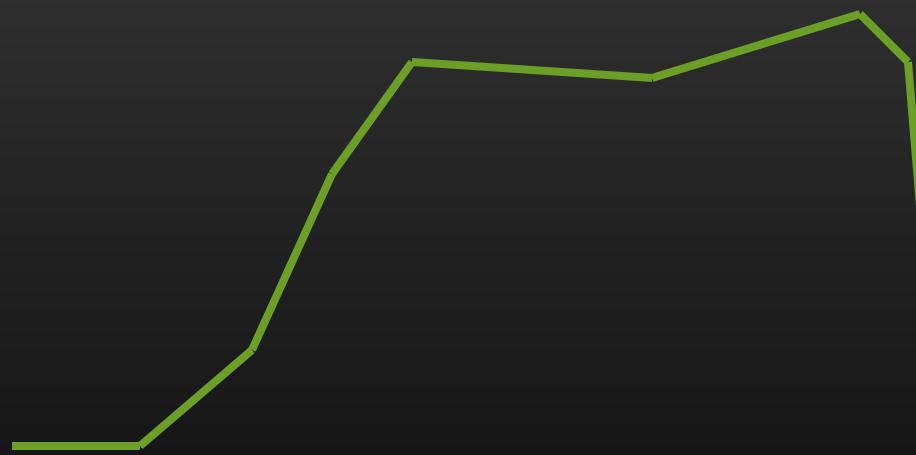
By adding more lines, we can make it as accurate as we like

# Polylines



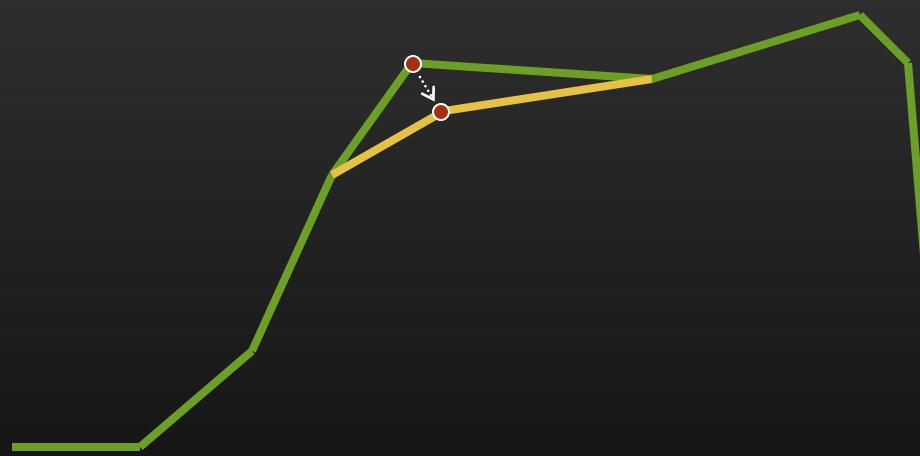
We end up with a bunch of line segments that share endpoints

# Vertices and edges



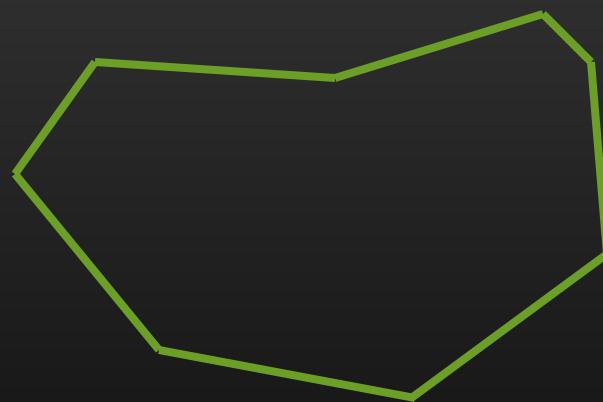
The endpoints are called **vertices** and the line segments **edges**

# Vertices and edges



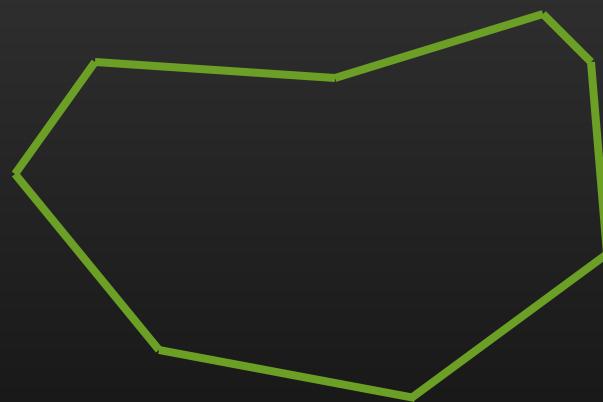
We can change the shape by moving a vertex

# Closed curves and polygons



If the curve joins up with itself, we say it's a **closed** curve

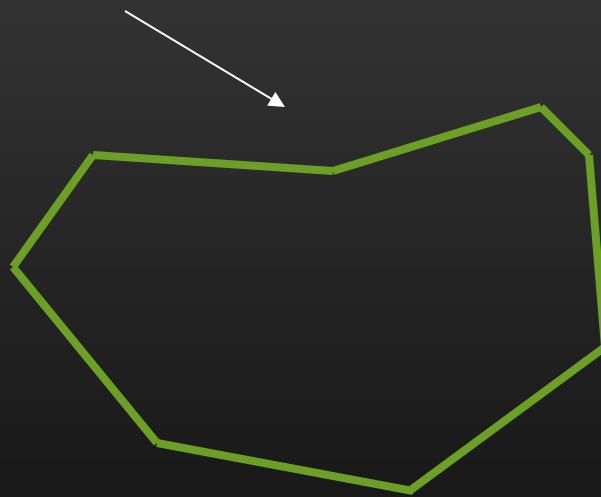
# Closed curves and polygons



A closed curve made out of lines is called a **polygon**

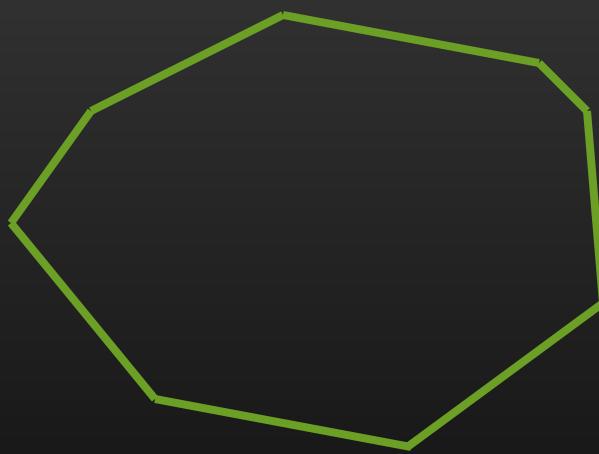
# Concave polygons

concavity



A polygon with an indentation is called a **concave** polygon

# Convex polygons



A polygon with no concavities is called **convex**

# Approximating a surface

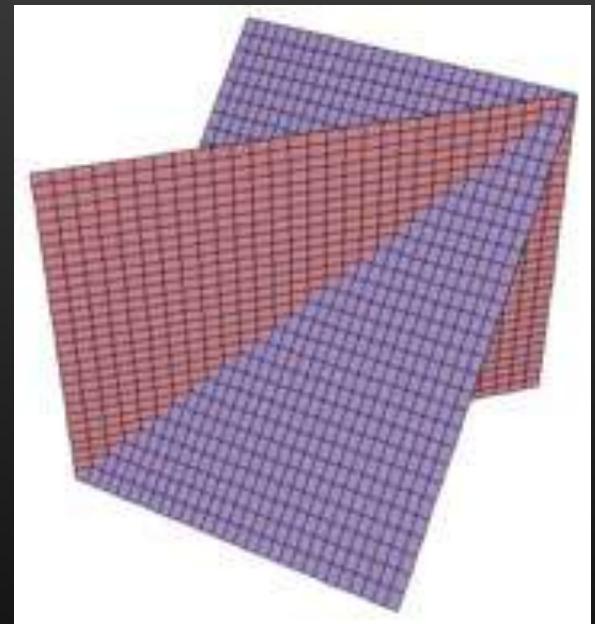
# Representing surfaces

- To represent 3D objects,  
we need to represent  
their **surfaces**



# Planar approximation

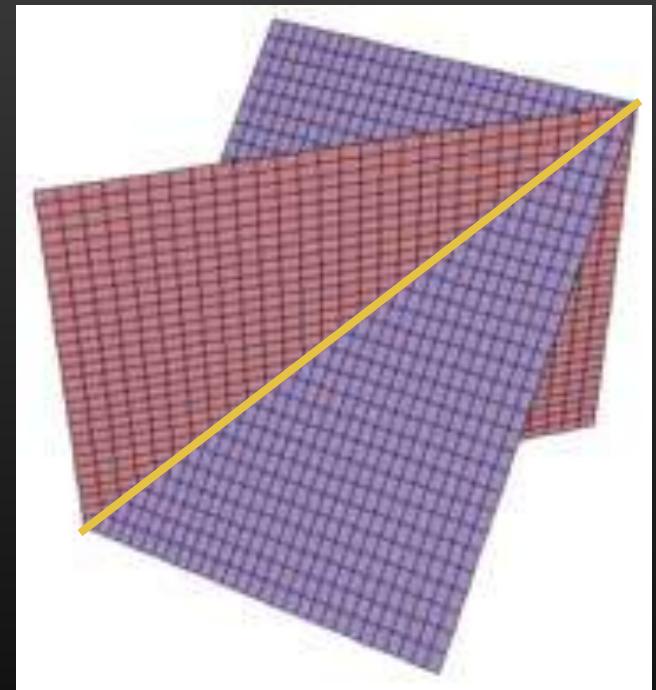
- We approximated **2D shapes** with groups of **line segments** (aka polygons)
- What would the **3D equivalent** be?
  - The 3D analog of a line is a **plane**
  - So we approximate surfaces with groups of **planar patches**
- What would those patches look like?



# Planar approximation

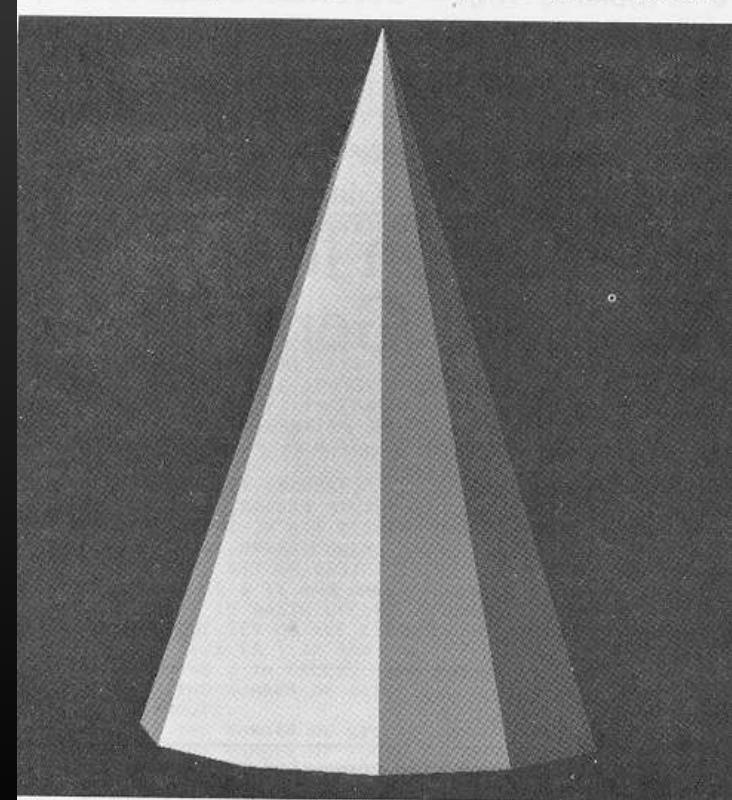
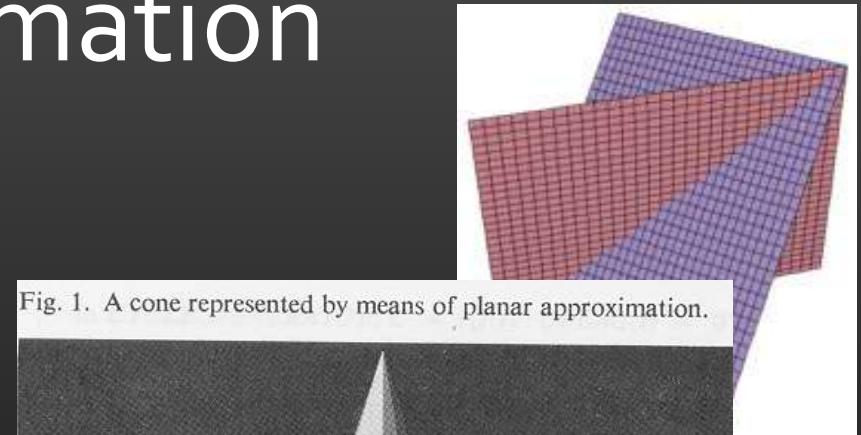
What would those patches look like?

- **Two planes** intersect in a **line**
- So **many planes** intersect in many lines
- Gee, this sounds familiar ...



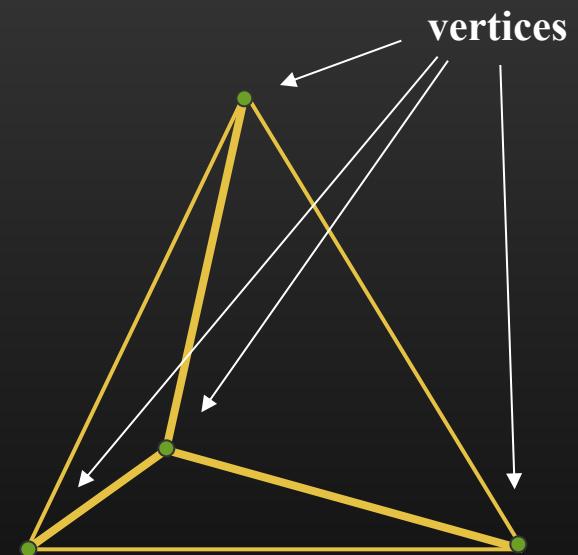
# Polyhedral approximation

- Whenever 4 or more planar patches intersect, they form a **polygon**
- Shapes modeled with **planar patches** form groups of **polygons**



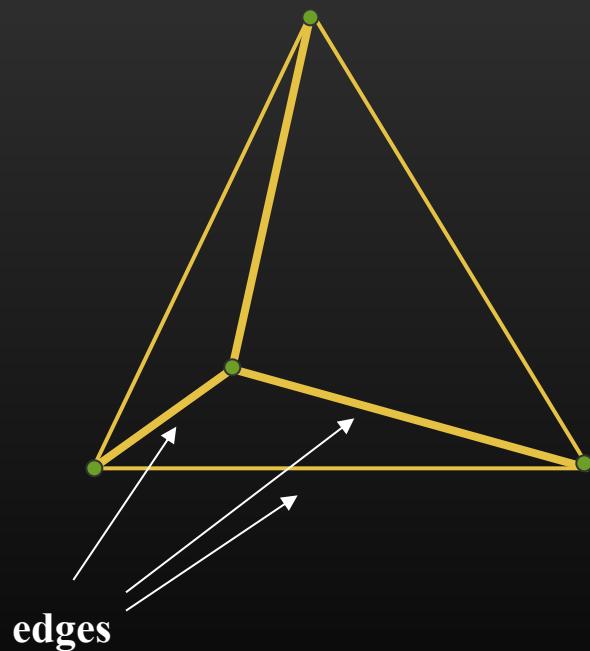
# Vertices, edges, and faces

- In 3D
  - The corners of the polygons are still called **vertices**
  - But the lines between them are called edges
  - And the polygons themselves are called faces
- Every edge is part of two faces
- Every vertex is part of at least two lines



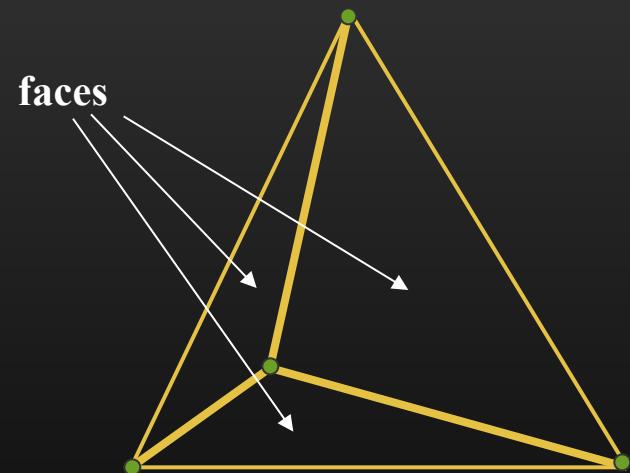
# Vertices, edges, and faces

- In 3D
  - The corners of the polygons are still called **vertices**
  - But the lines between them are called **edges**
  - And the polygons themselves are called faces
- Every edge is part of two faces
- Every vertex is part of at least two edges



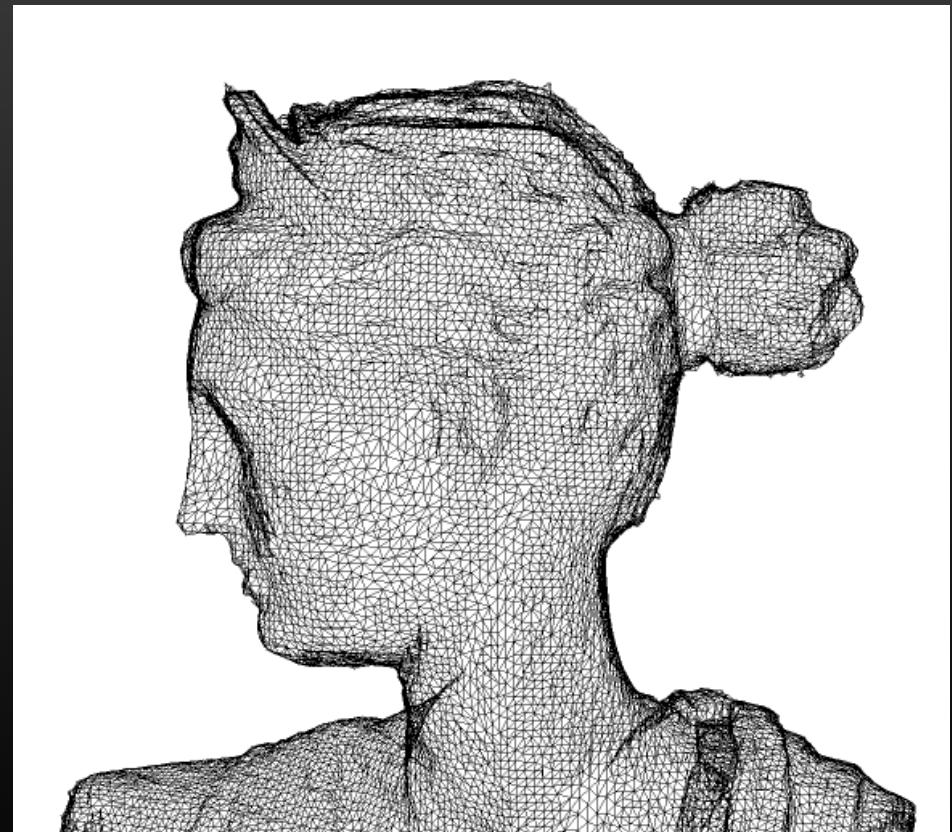
# Vertices, edges, and faces

- In 3D
  - The corners of the polygons are still called **vertices**
  - But the lines between them are called **edges**
  - And the polygons themselves are called **faces**
- Every edge is part of two faces
- Every vertex is part of at least two edges

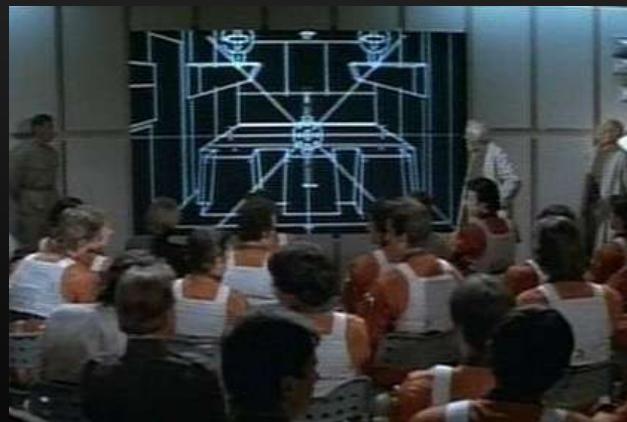
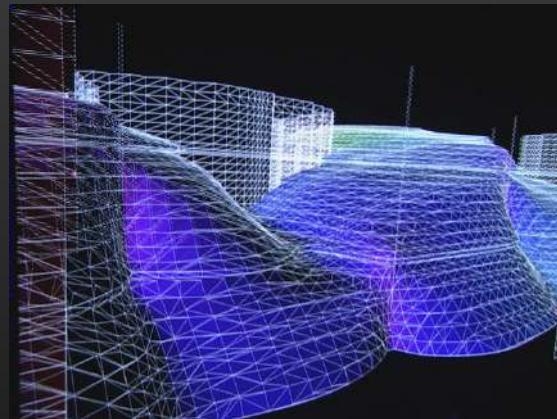
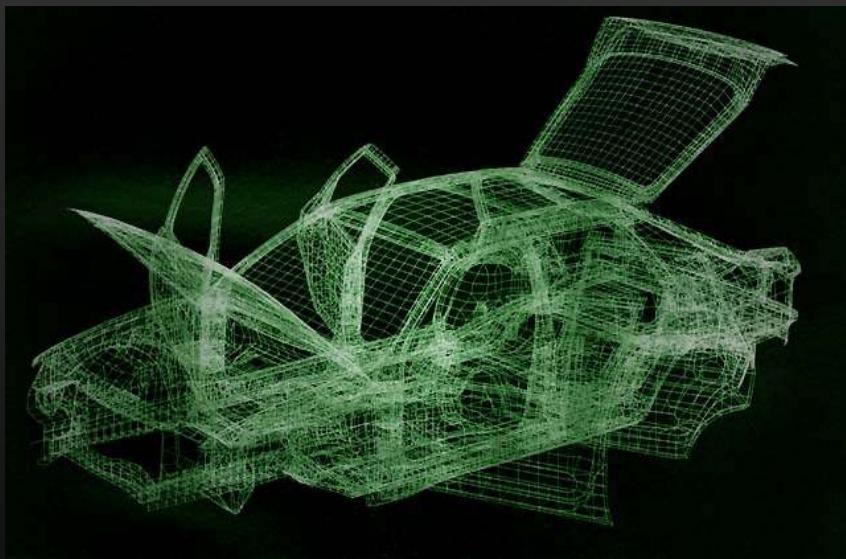


# Polygonal meshes

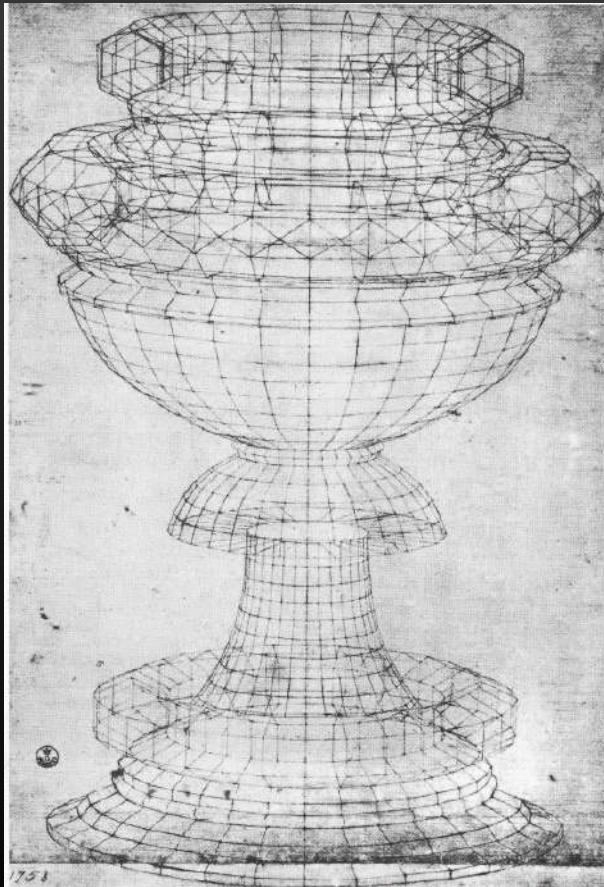
- When large groups of polygons form a surface, we call it a **mesh**
- Mesh polygons share a set of **common vertices**
- **All GPUs** use this surface representation



# Polygonal meshes



# Meshes have a long tradition

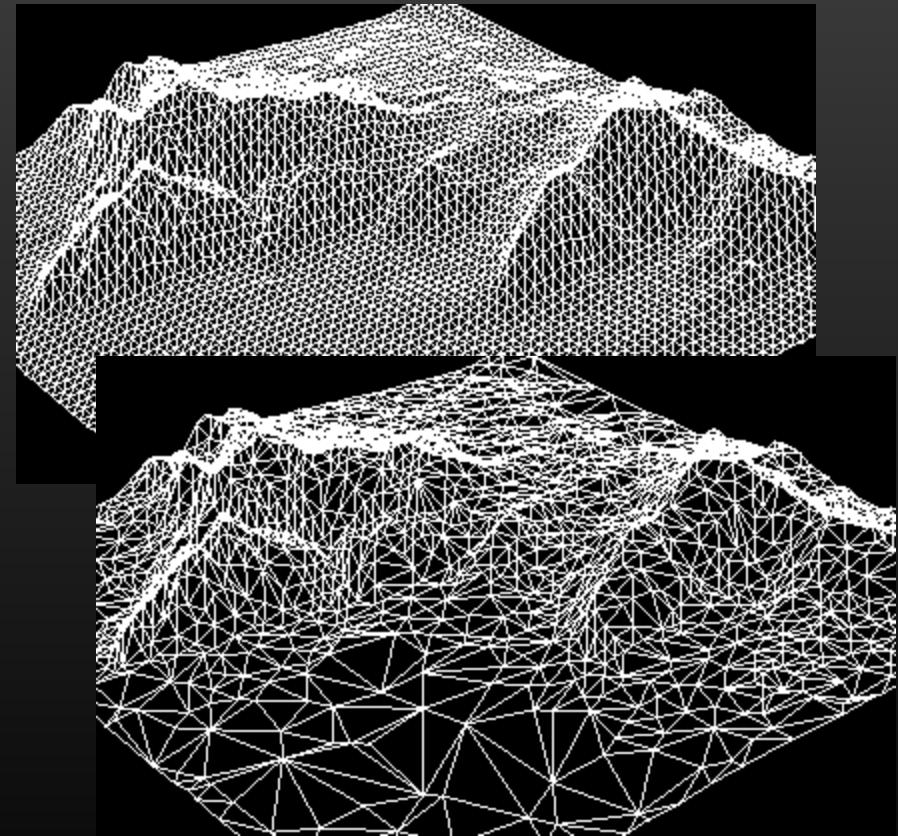


*Perspective Study of a Chalice*  
Paolo Uccello 1430-1440.

# Triangle meshes

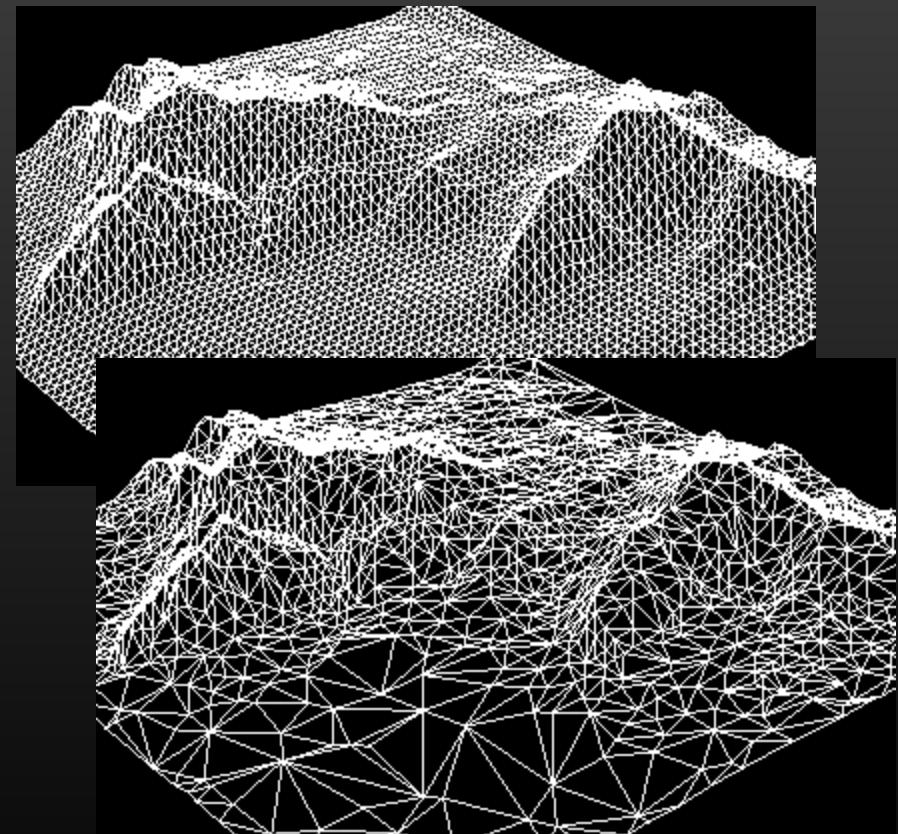
# Triangle meshes

- **3D modeling programs** usually represent surfaces as sets of arbitrary polygons
- Most **graphics cards**
  - Provide hardware for drawing **triangles only**
  - But any polygon can be decomposed into a **set of triangles**



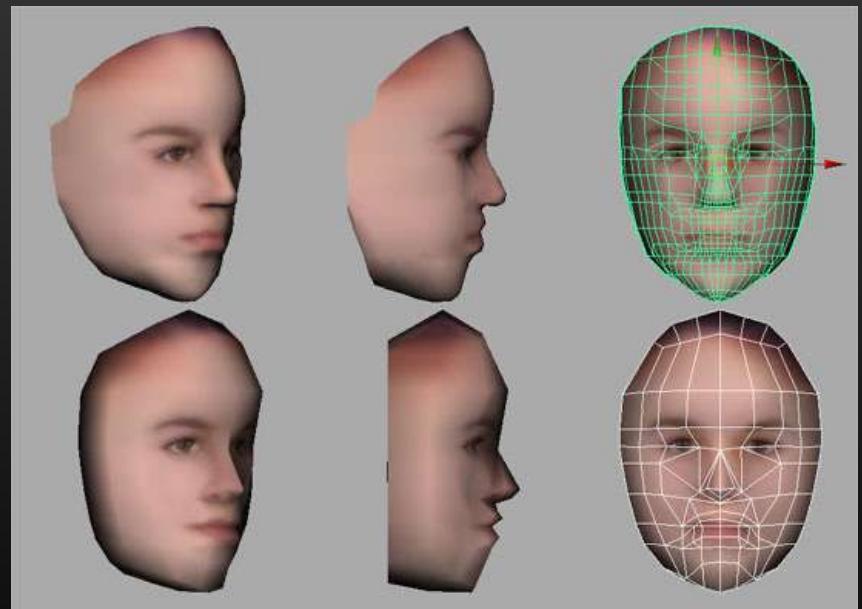
# Triangle meshes

- **Polygon meshes** are usually **converted** to triangle meshes for final rendering
- Many artists model **directly using the triangle mesh** so they have as much control as possible



# Resolution and polygon count

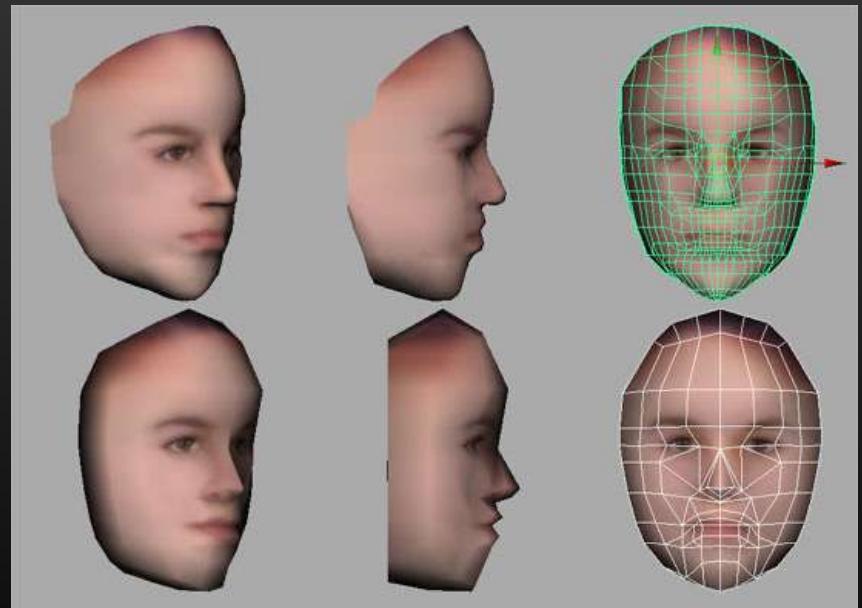
- Graphics card speed is often measured in **triangles per second**
- A **1M tri/sec** card **sounds fast**
- But it can **only render**
  - A **10,000 triangle** scene at **100 fps**
  - A **100,000 triangle** scene at **10 fps**



Source: [http://cache-www.intel.com/cd/00/00/01/45/14535\\_continuous\\_detail\\_levels\\_f2.jpg](http://cache-www.intel.com/cd/00/00/01/45/14535_continuous_detail_levels_f2.jpg)

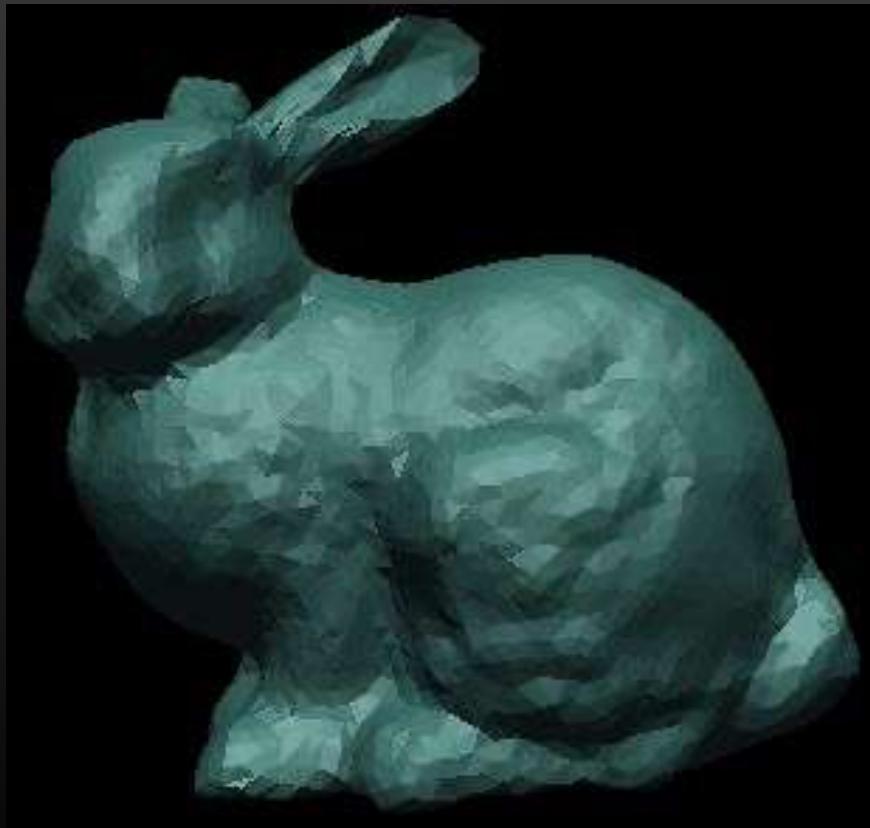
# Resolution and polygon count

- So models trade off **speed for accuracy**
  - More polys **More realistic** still frames
  - Fewer polys **Higher frame rate**
- Many artists in the game industry specialize in **low poly count** modeling



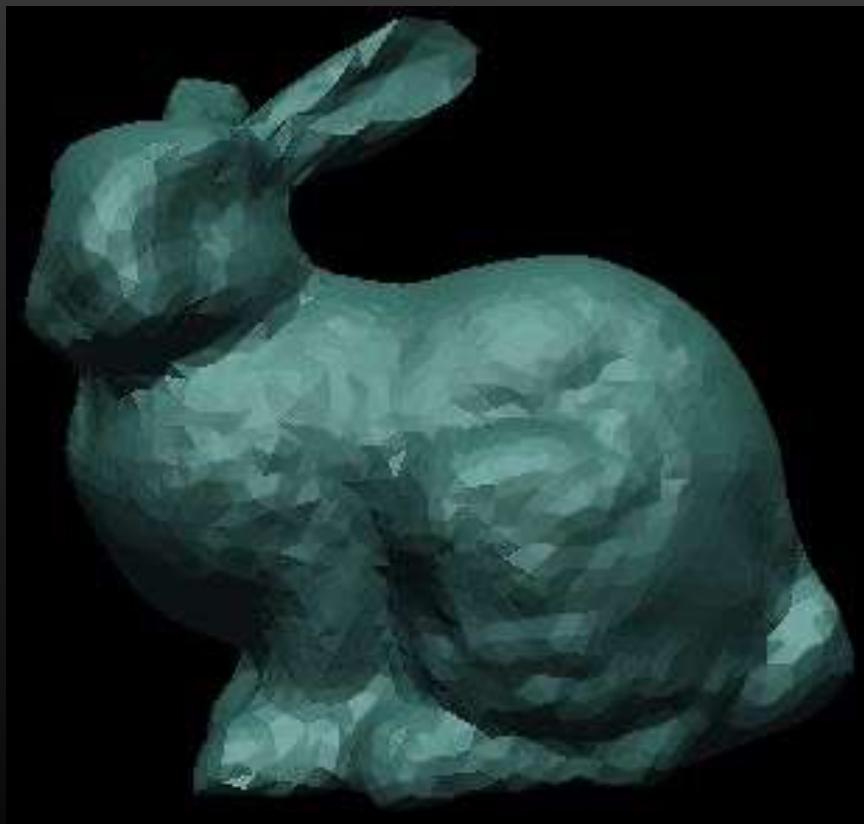
Source: [http://cache-www.intel.com/cd/00/00/01/45/14535\\_continuous\\_detail\\_levels\\_f2.jpg](http://cache-www.intel.com/cd/00/00/01/45/14535_continuous_detail_levels_f2.jpg)

# 10000 triangles

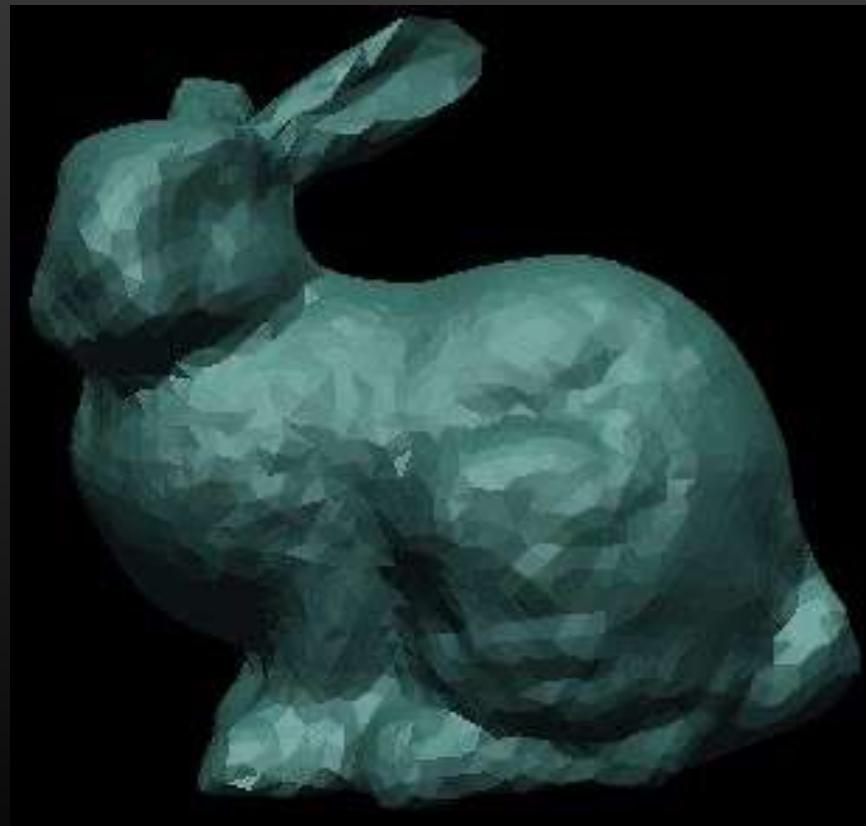


Source: <http://www.cs.utah.edu/~wyman/classes/wavelets/proj4/>

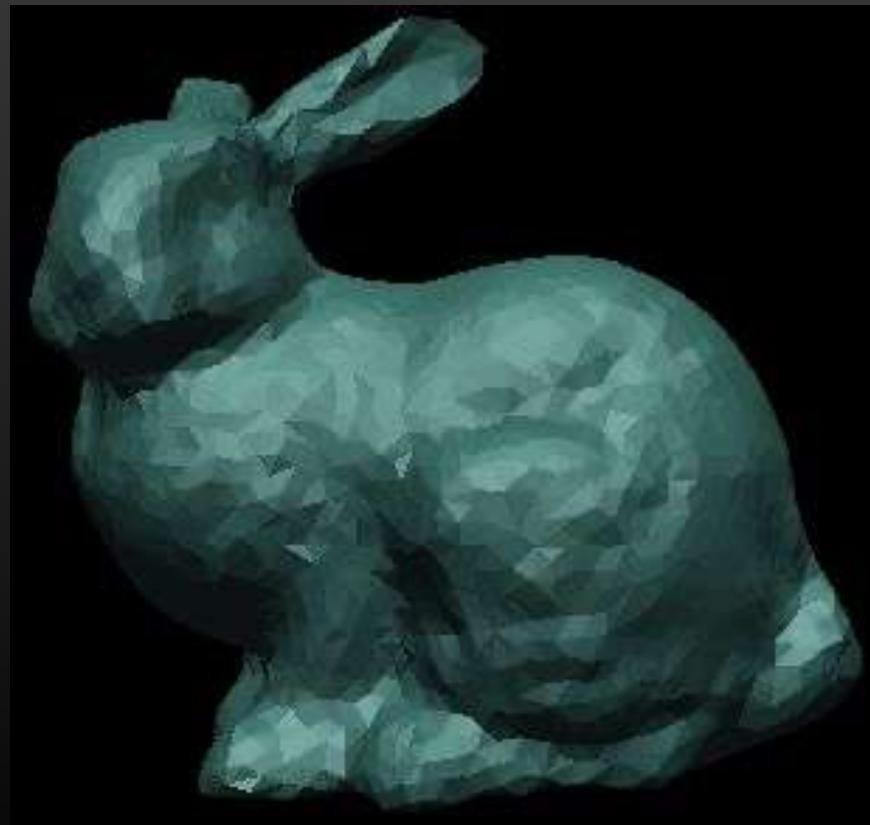
9000 triangles



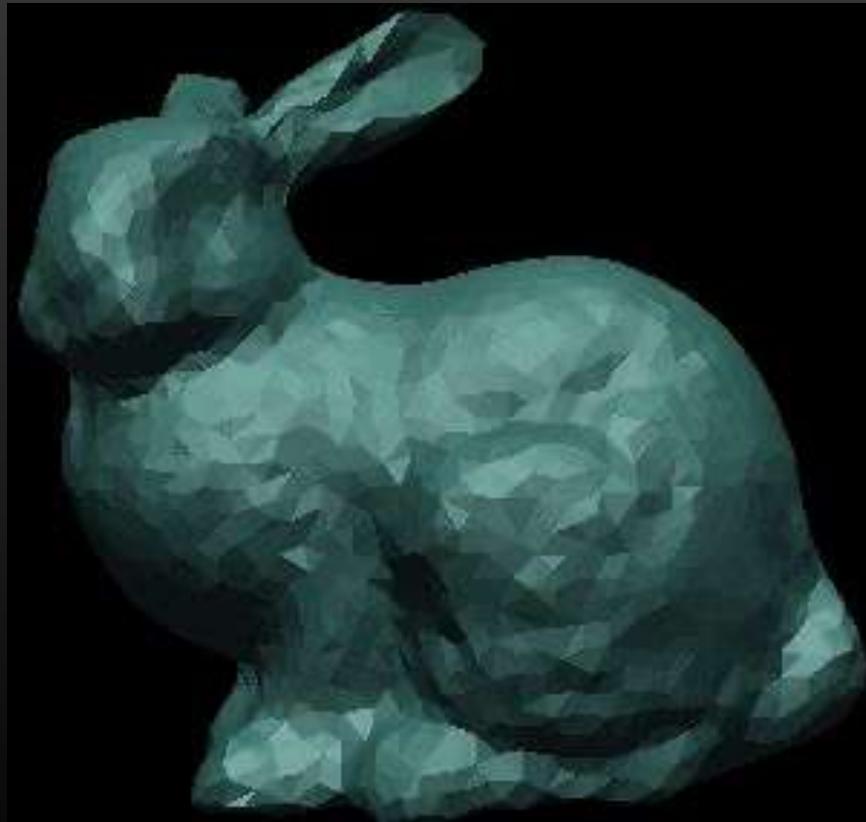
# 8000 triangles



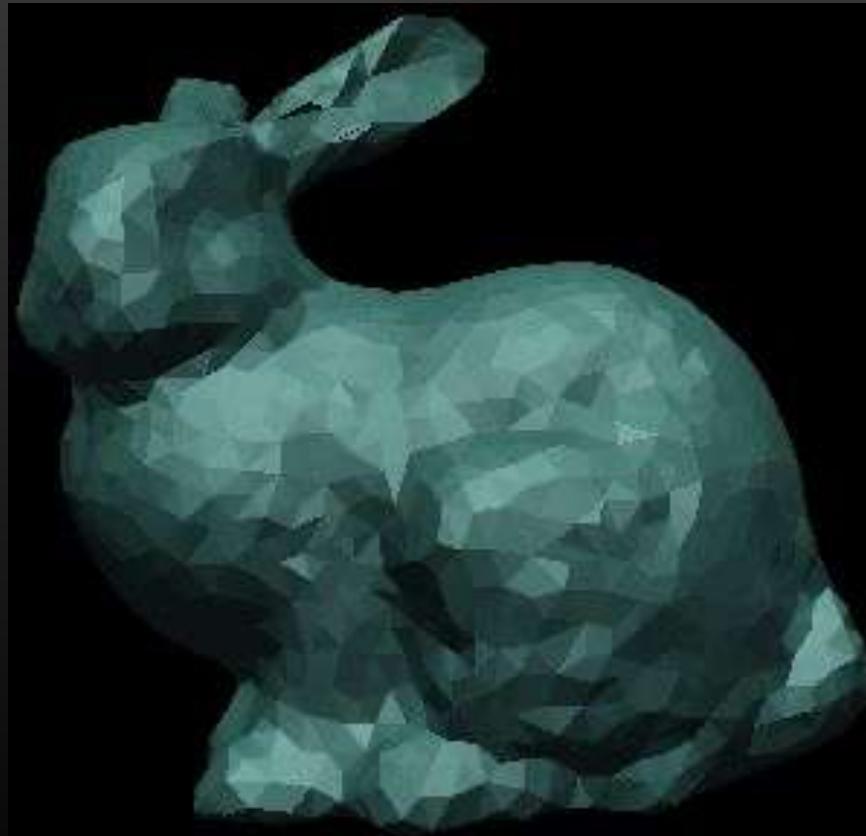
7000 triangles



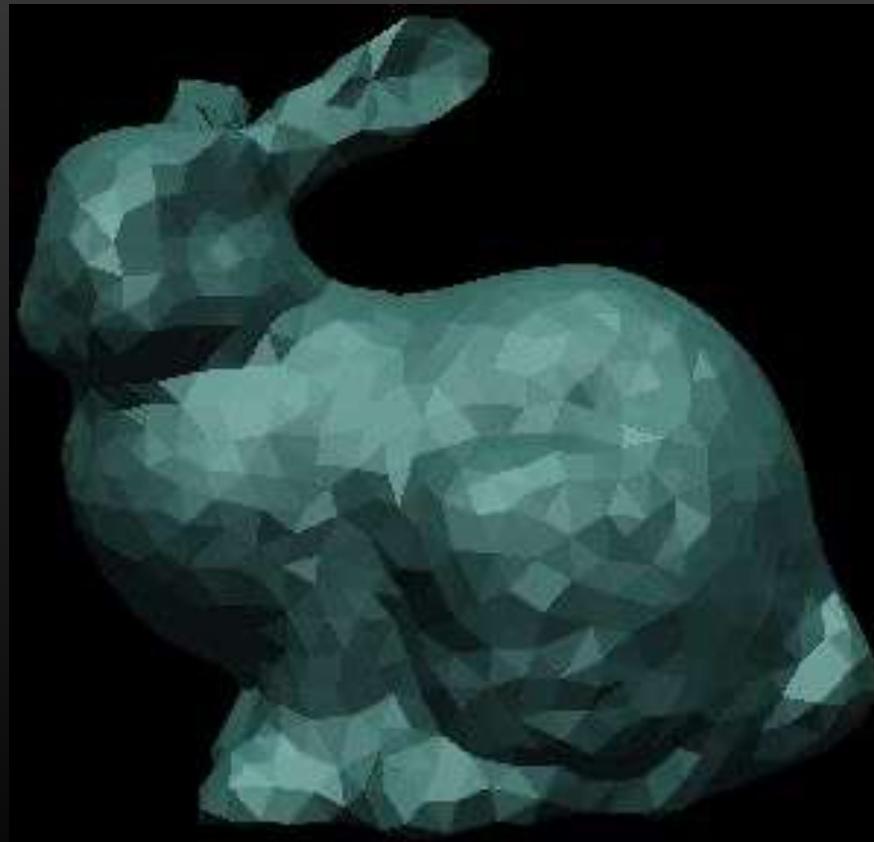
# 6000 triangles



# 4000 triangles



# 3000 triangles



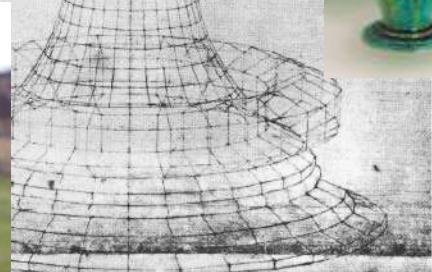
# Constructing meshes

# Polygon modeling techniques

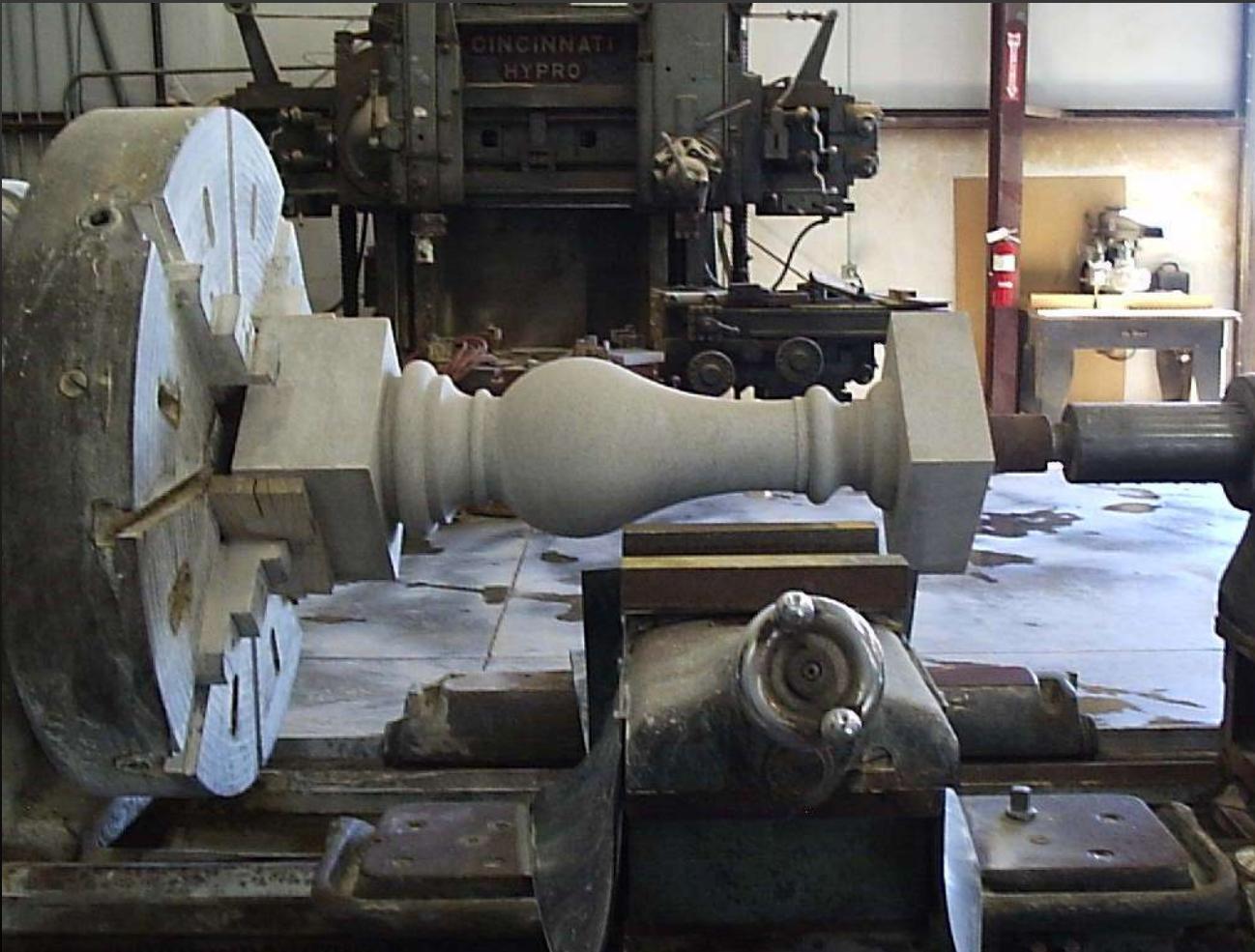
- 3D **scanning**
  - **Detailed**
  - But requires a lot of **cleanup** by a skilled artist
- Make a **primitive shape**
  - Most modeling programs can make spheres, cones, cubes, etc. automatically
- **Deform** an existing shape
  - Rotate, translate, **scale**
  - **Stretch** (scale in one direction)
  - **Move the vertices** around
  - **Add/remove** vertices  
(Modeling tool will split/merge polygons accordingly)

# Solids of revolution

- Solids of revolution are **common in the real world**
- Partly because we have machines that make them **cheaply** (lathes)



# Turning an object on a lathe



# An electronic lathe: Making a wine glass in Maya

- Start by drawing the **curve for the profile** of the object



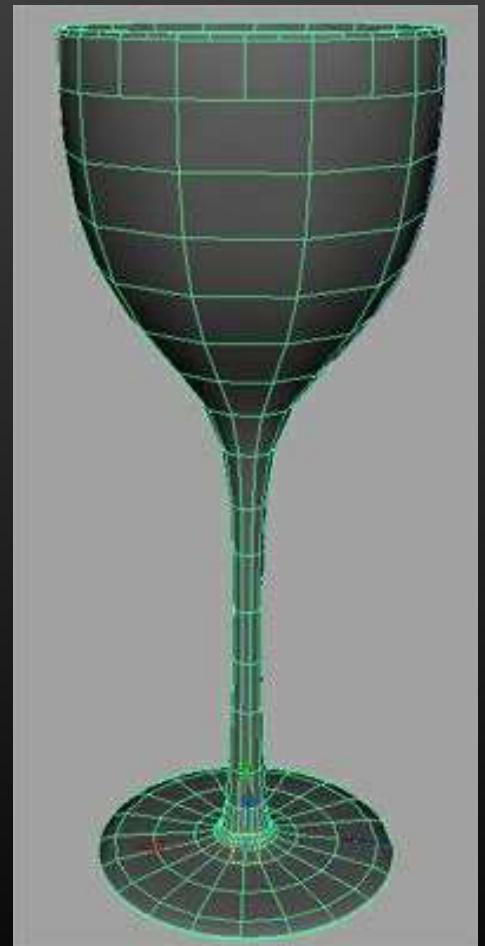
# Making a wine glass

- Start by drawing the **curve for the profile** of the object
- Select the **revolve** operation



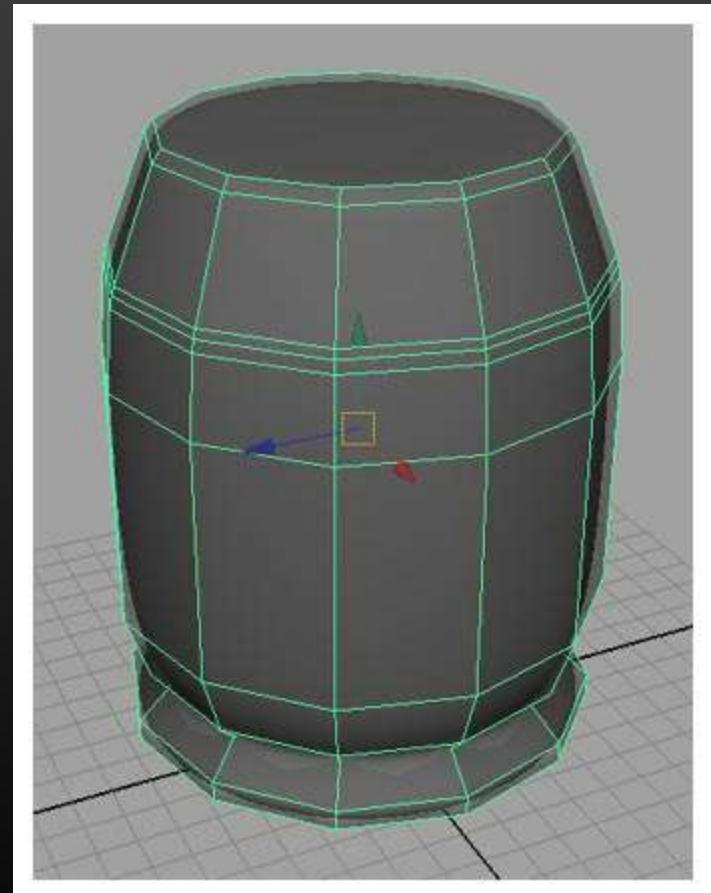
# Making a wine glass

- Start by drawing the **curve for the profile** of the object
- Select the **revolve** operation
- Voila!



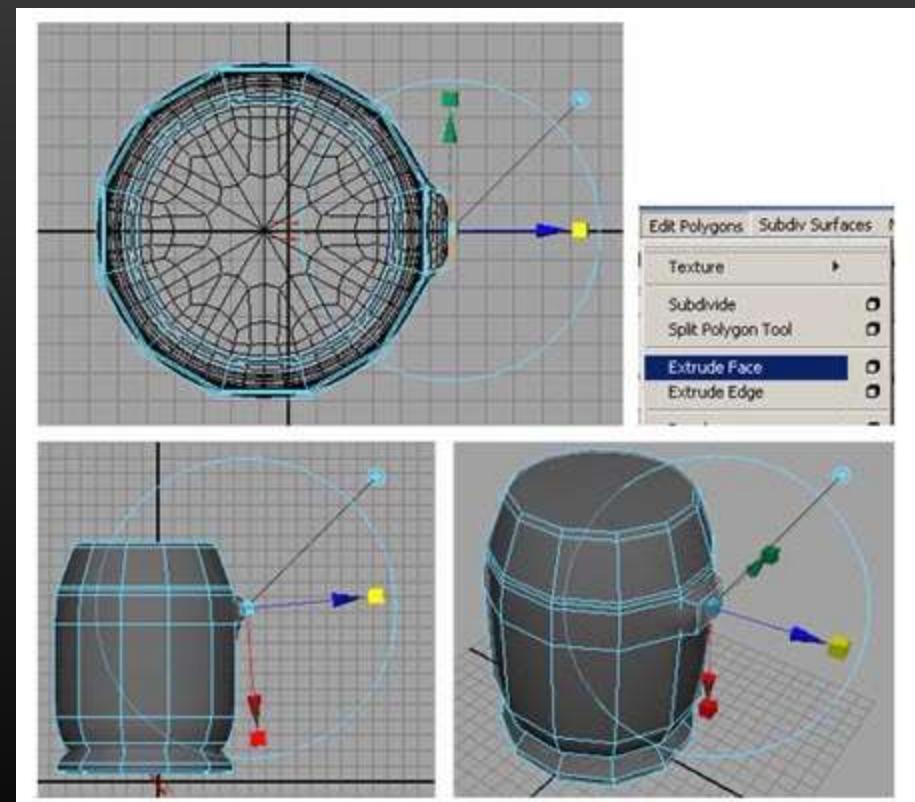
# Extruding an existing face

- Most modelers will let you “**pull**” a face out **from the surface**
- It will **automatically create new polygons** to rejoin the face to the surface

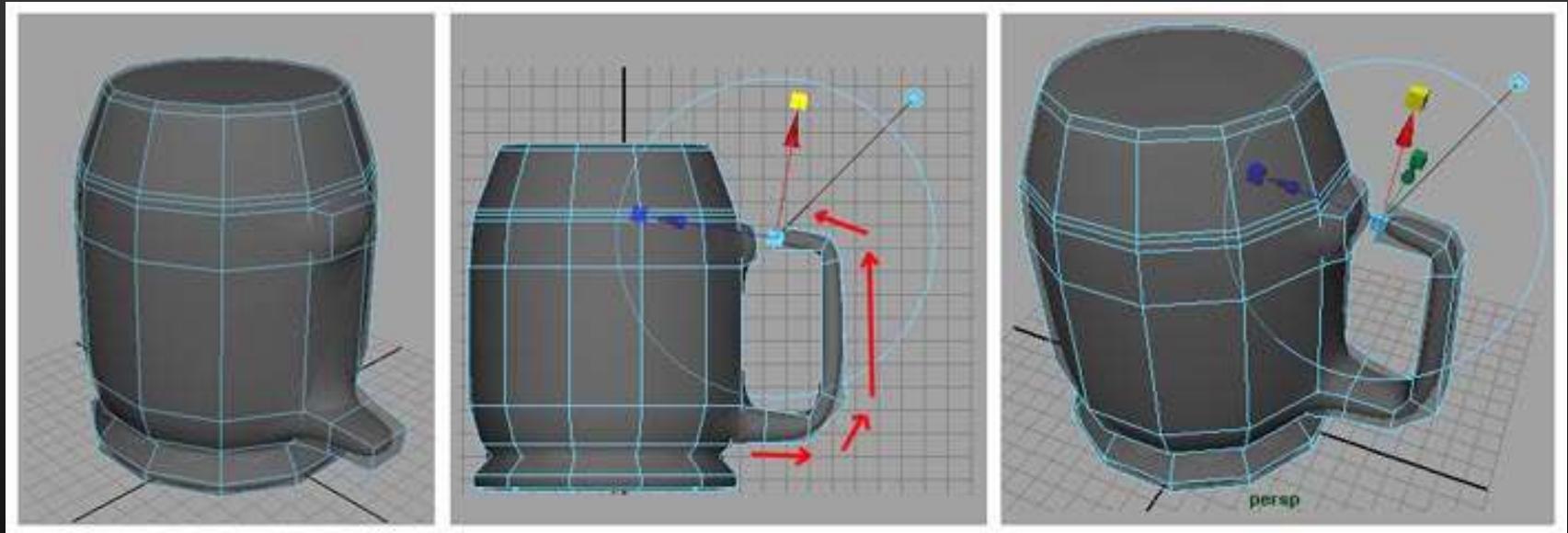


# Extruding an existing face

- Most modelers will let you “**pull**” a face out **from the surface**
- It will **automatically create new polygons** to rejoin the face to the surface



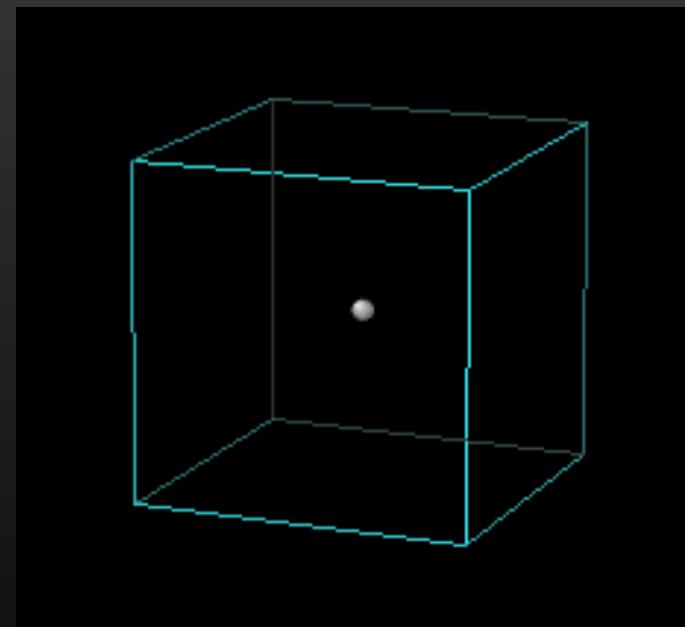
# Extruding an existing face



# Making a T-shirt

# Example: Making a T-Shirt

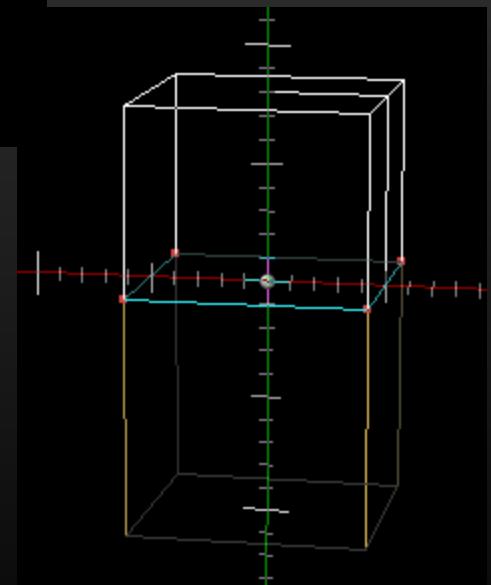
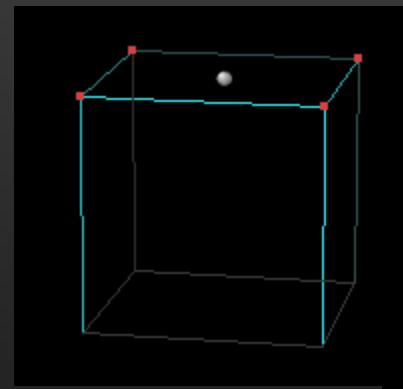
- Start with a **cube**



Source: [http://zobal.free.fr/tuts/acm/en\\_acm2.htm](http://zobal.free.fr/tuts/acm/en_acm2.htm)

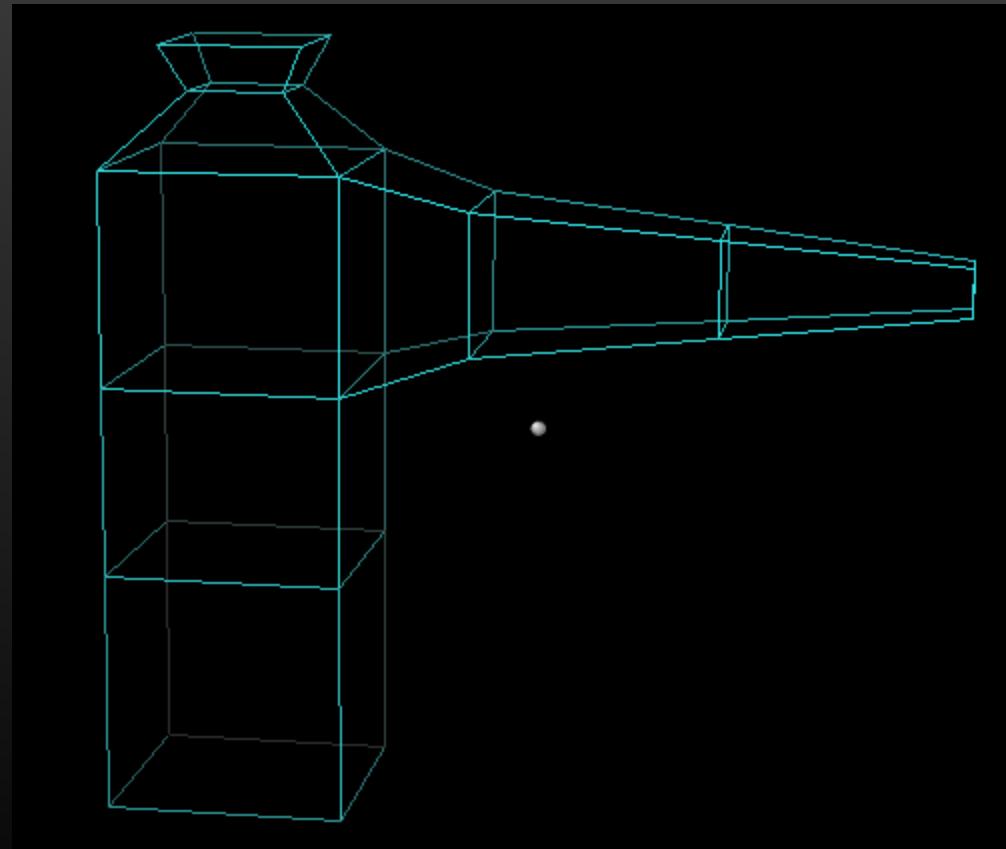
# Face extrusion

- **Extrude** the top face



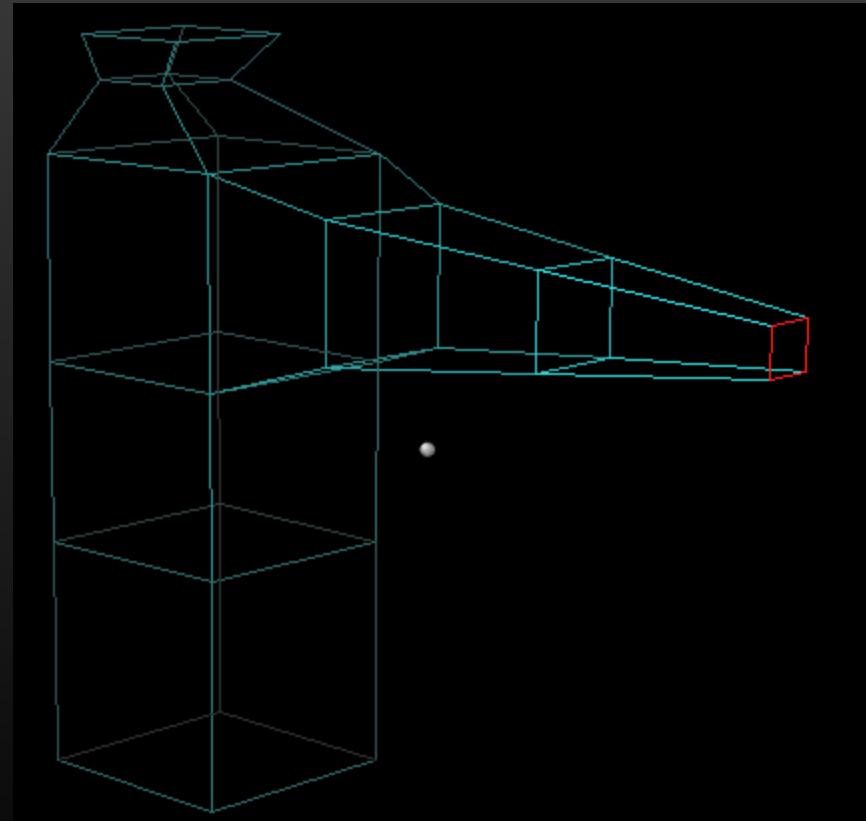
# Face extrusion

- **Pull sides** to make
  - The **neck** (pull in)
  - An **arm** (pull out)



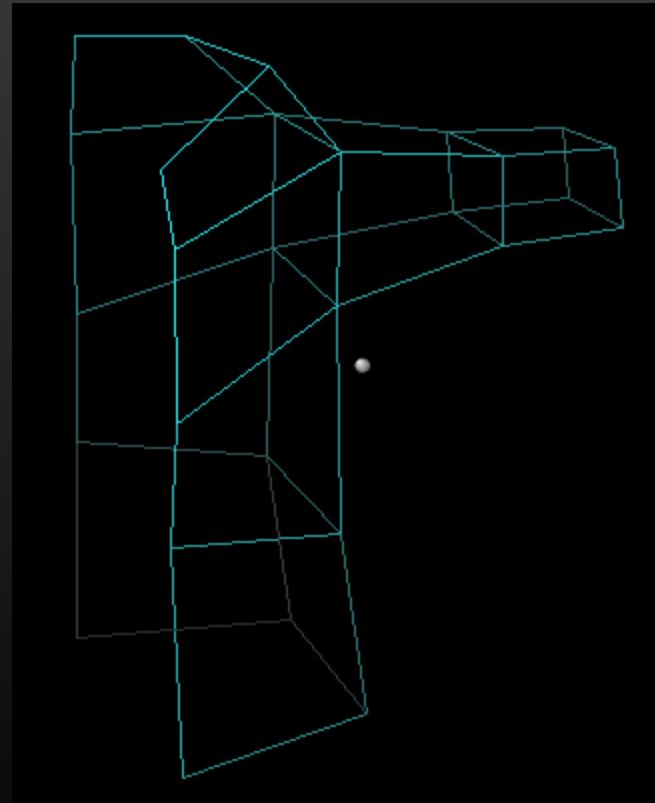
# Face deletion

- Right now, the arm is **closed at the end**
- **Remove the face** on the end to make an arm hole



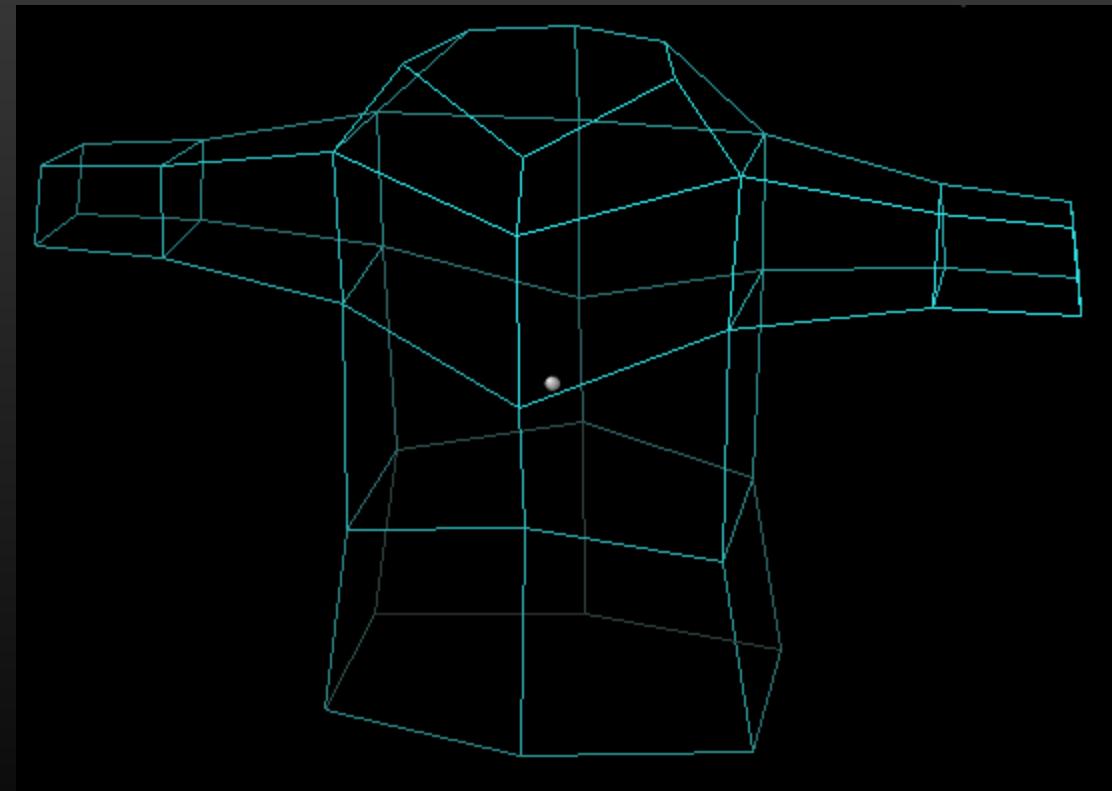
# Removing one side

- Remove
  - All the faces on the **left side**
  - The **bottom face**
- And generally **tweak the vertices** until you like them



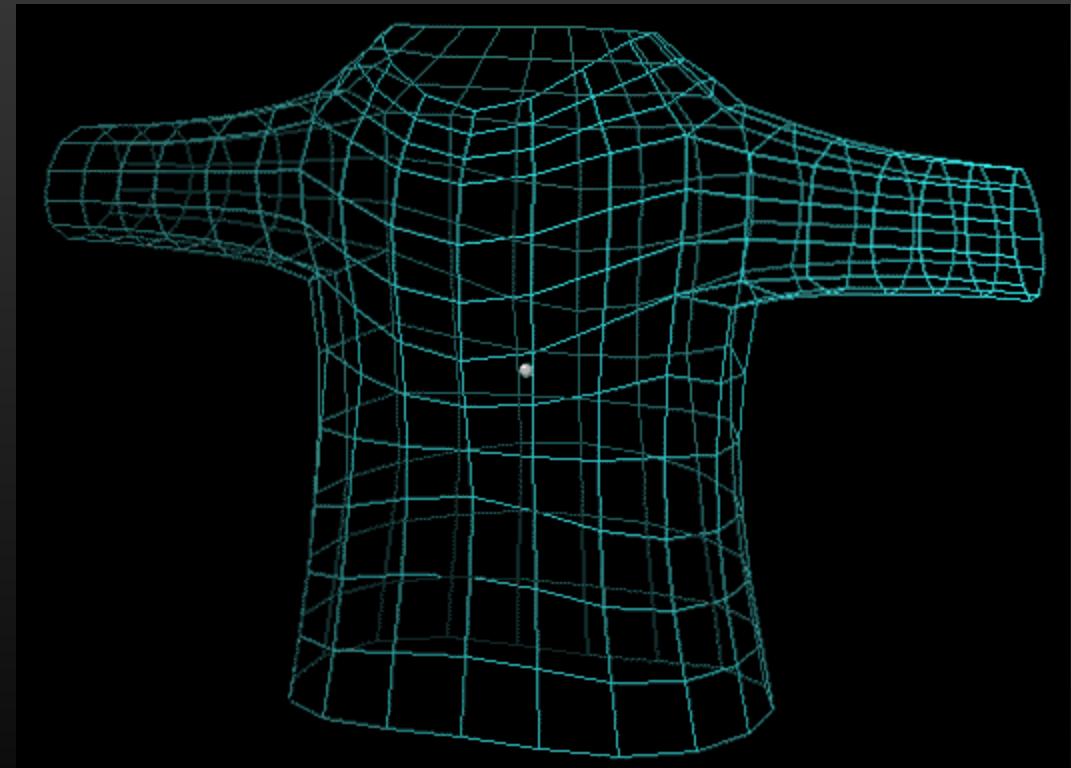
# Forming the whole thing

- Now **clone** the mesh
- **Flip** it over
- And **join** it to the old mesh



# Smoothing

- Most modeling tools have a **smoothing** function
  - Although it's **expensive**
  - In terms of polys
- After smoothing, **tweak** to taste



# Rendering

- The previous images were **wireframes**
  - They just show the vertices and edges
- **Rendering** is the process of producing a final image from the polygon mesh



Modeling  
surface material

# Rendering

- The previous slide deck showed **wireframes**
  - They just show the vertices and edges
- **Rendering** is the process of producing a final image from the polygon mesh



# Rendering

Rendering is the process of modeling

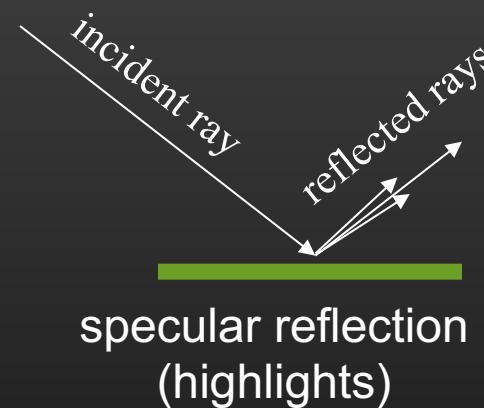
- The transmission of **light**
- From a **light source**
- To the **camera**
- As it **interacts with surfaces** along the way



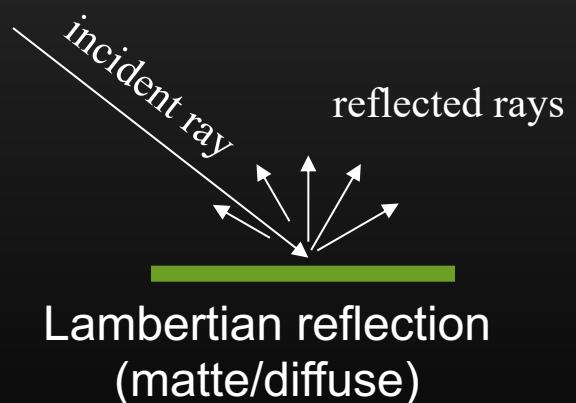
# Surface reflection

# Surface reflection

- Surface reflectance is very **complicated**
- There are **two main models** of reflectance
  - **Specular** (glossy) surfaces bounce it directly off
  - **Lambertian** (matte, aka **diffuse**) bounce it evenly in all directions



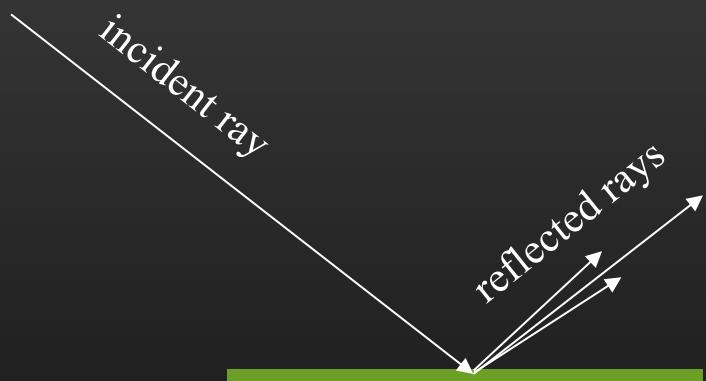
specular reflection  
(highlights)



reflected rays  
Lambertian reflection  
(matte/diffuse)

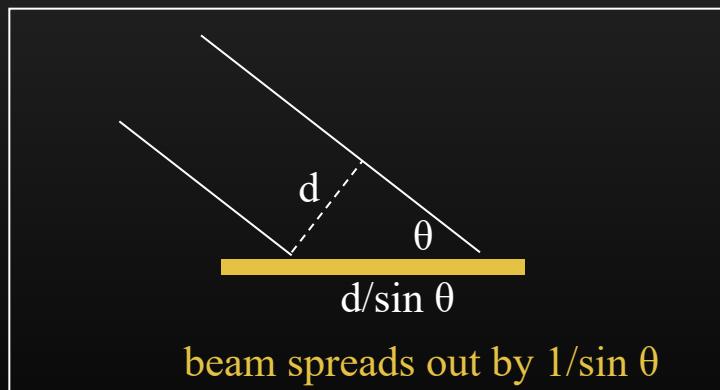
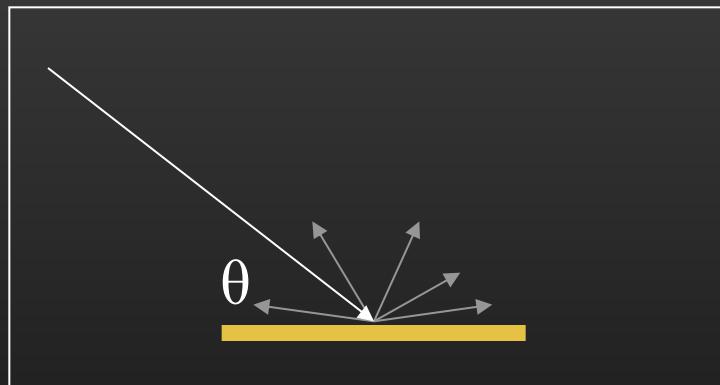
# Specular reflection

- **Mirror-like reflection**
  - Mirrors are near-perfect specular reflectors
- **Incident** and **reflected** rays have (almost) the **same angle**
  - In practice, there's some **scattering**
- All **wavelengths** are (usually) reflected equally
  - So reflection has the **color of the light**



# Lambertian reflection

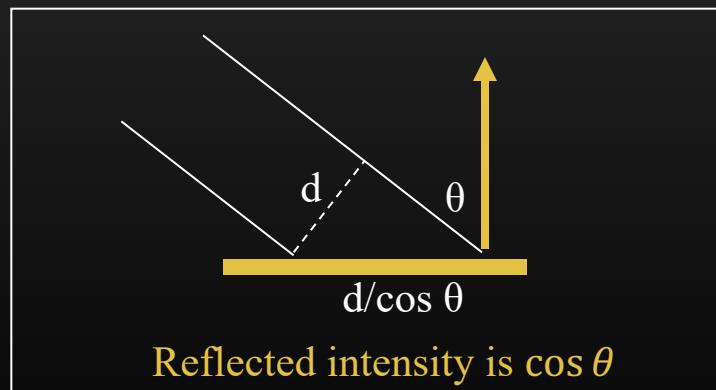
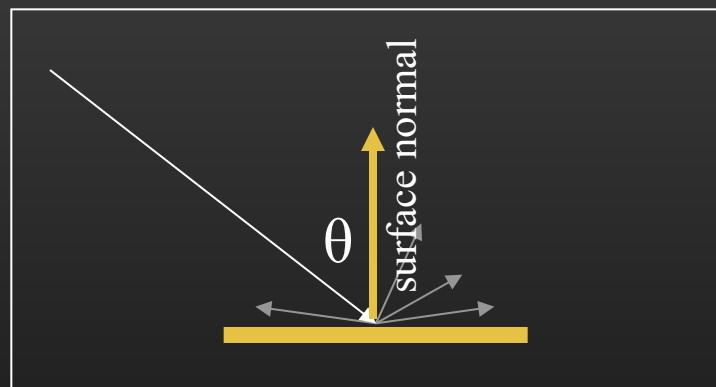
- Lambertian/**diffuse**/matte reflection
  - Perfect **non-glossy paint**
  - Light reflected equally in all directions
- Brightness depends on **illumination angle**
  - When light hits at an angle, it's **spread out** over a wider area ( $1/\sin \theta$  times wider)
  - So the **intensity** of the light coming out is dimmed by  $\sin \theta$
- Not all **wavelengths** are reflected equally
  - The reflection has the **color of the surface** (as well as the color of the light)



# Surface normals

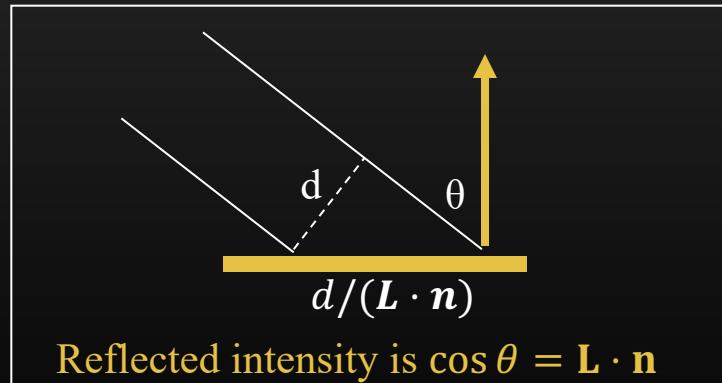
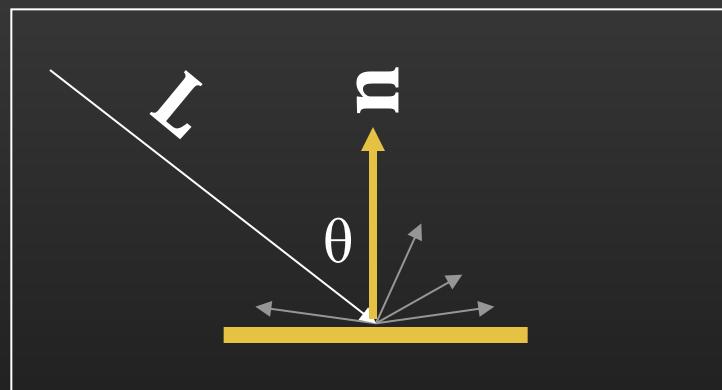
In graphics, we measure the angle differently

- Angle between the **incident light** and
- The **surface normal**
  - A line sticking straight out at **right angles** to the surface



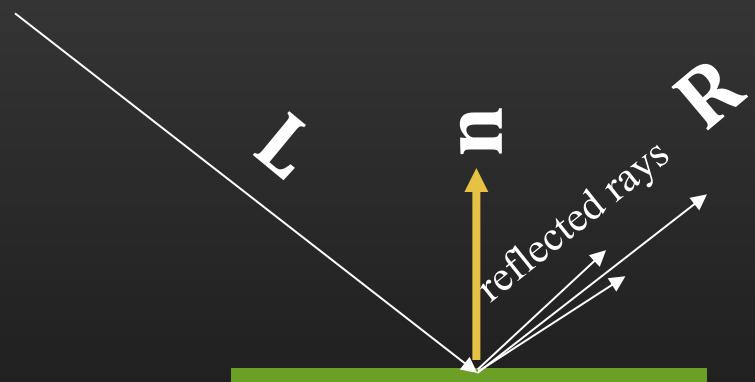
# Surface normals

- This means the dimming factor is  $\cos \theta$ 
  - Because we measured  $\theta$  differently
- And now we can write it as a **dot product** of the unit vectors for
  - The direction of the **incident light**
  - The **surface normal**



# Computing specular reflection

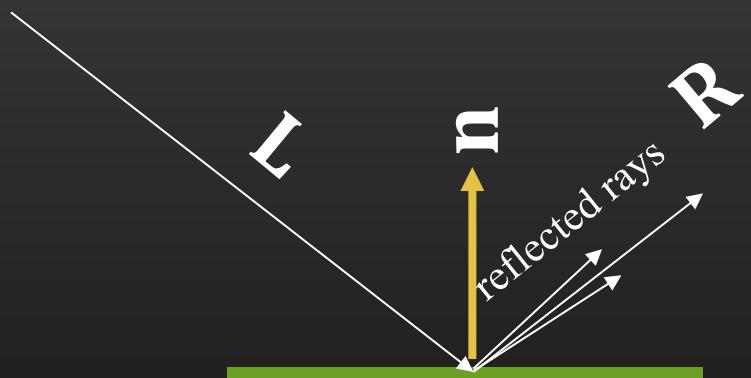
- To compute the specular reflection, we need to compute the **vector for the reflected ray**



# Computing specular reflection

Here's the idea:

- $L$  can be decomposed into **two components**
  - A **normal** component (parallel to  $n$ )
    - $L_n = n(L \cdot n)$
  - A **tangential** component
    - Everything else
    - $L_T = L - L_n$
- $R$  is  $L$  with its **normal component reversed**



# Computing specular reflection

So we have:

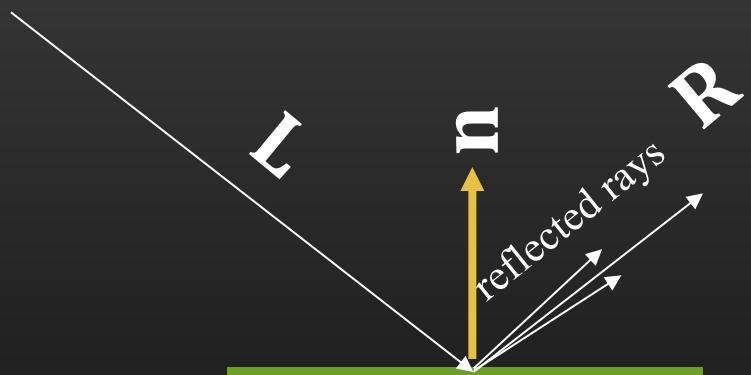
$$\mathbf{L} = \mathbf{L}_T + \mathbf{L}_n$$

$$\mathbf{R} = \mathbf{L}_T - \mathbf{L}_n$$

$$= (\mathbf{L} - \mathbf{L}_n) - \mathbf{L}_n$$

$$= \mathbf{L} - 2\mathbf{L}_n$$

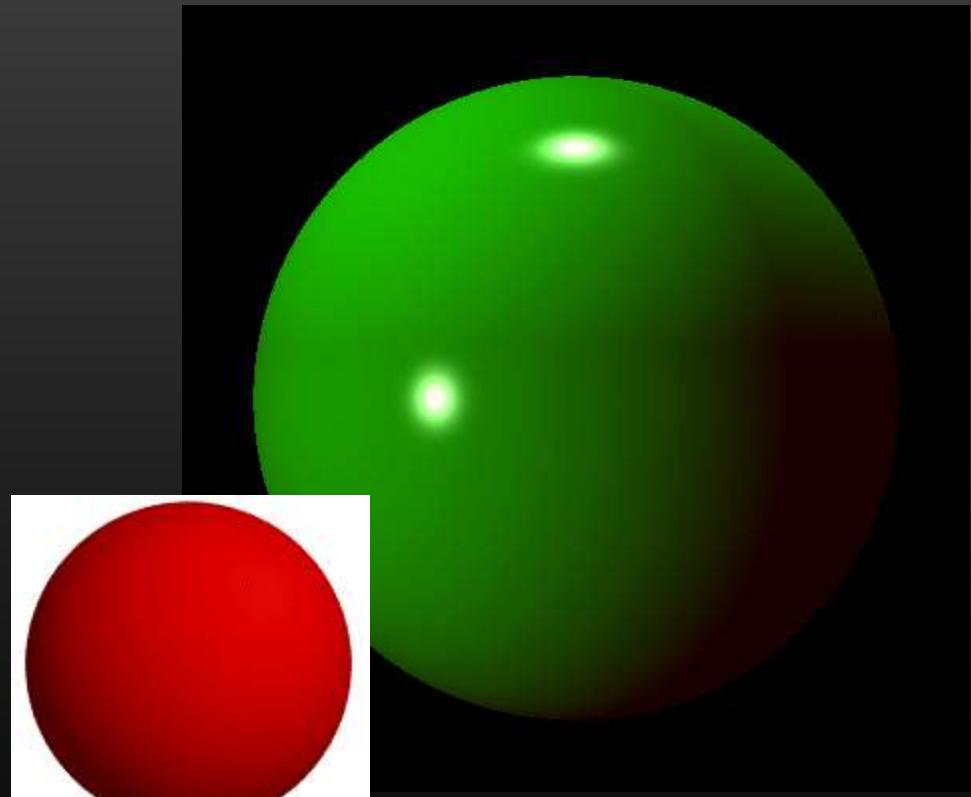
$$= \mathbf{L} - 2(\mathbf{n}(\mathbf{L} \cdot \mathbf{n}))$$



Note: this is a different derivation from the book, which has L pointing in the opposite direction

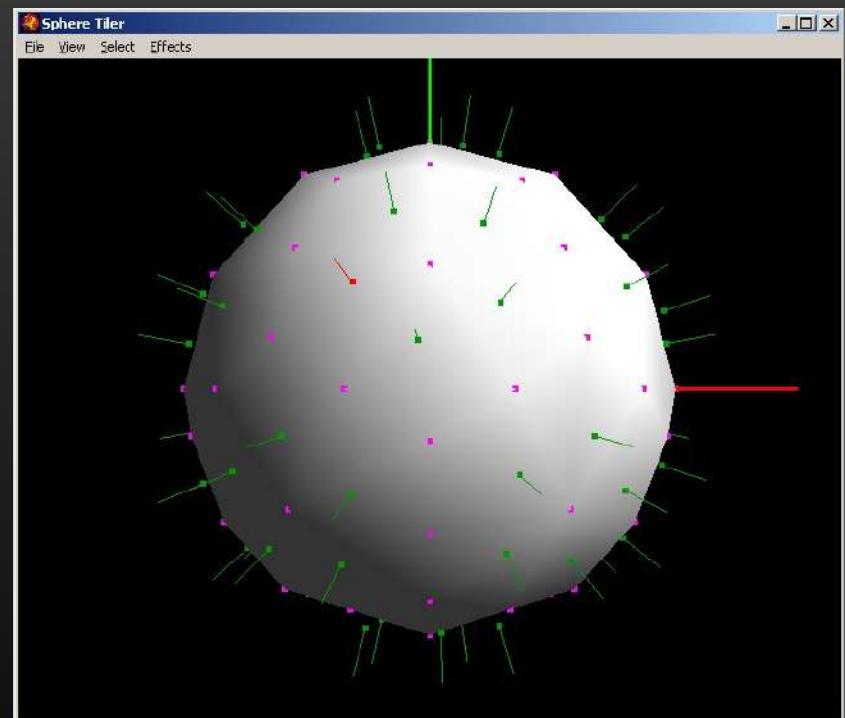
# Surface normals

# Specular and diffuse reflection



# Normals in 3D

- **Each point on a surface** has a normal
  - The **normal varies** over the surface
- All points in a **polygon** technically have the **same normal**



# “Flat shading”

- All points on a face have the **same normal**
  - So they really **look flat**
  - And the **boundaries** are very noticeable



# Normal interpolation

- By **interpolating** the normal **within a triangle**,
- We get a **smoother** look for a given **poly count**
- **Each pixel** in a face recalculates its normal
  - Using an **average** of the normals of **each vertex**
  - **Weighted by distance** from the pixel
  - Closer vertices exert more influence



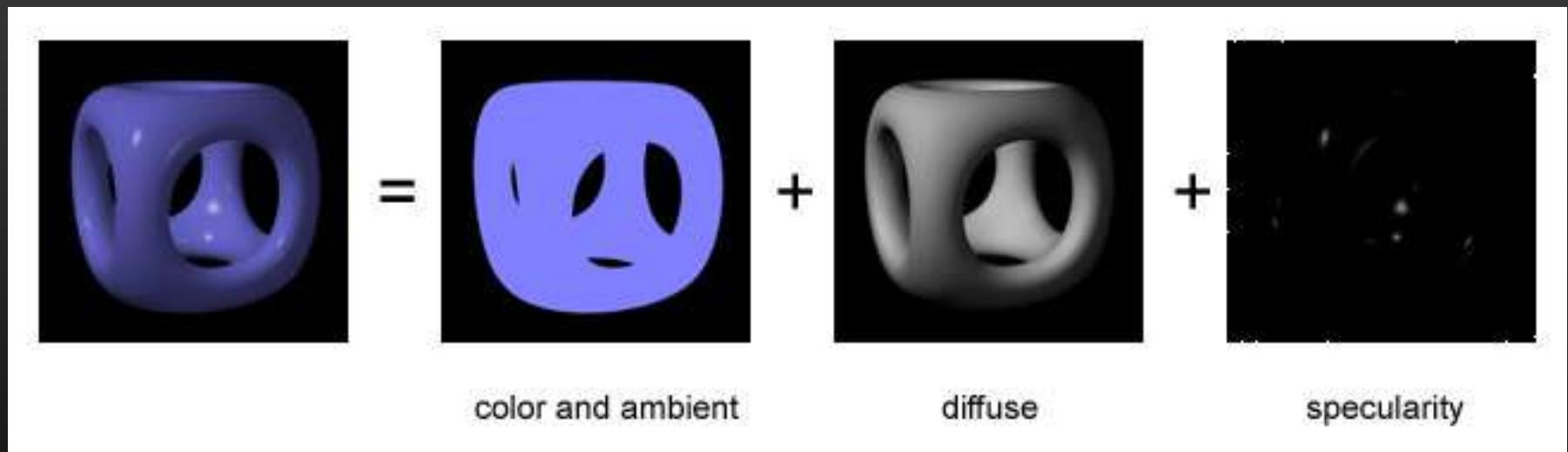
# Phong shading

# Phong shading

- **Common** lighting model
- **Sum** of
  - **Specular** and **diffuse** components
    - With **normal interpolation**
  - “**Ambient**” component
    - **Constant** added to all pixels
    - Acts like a **fill light**



# Phong shading

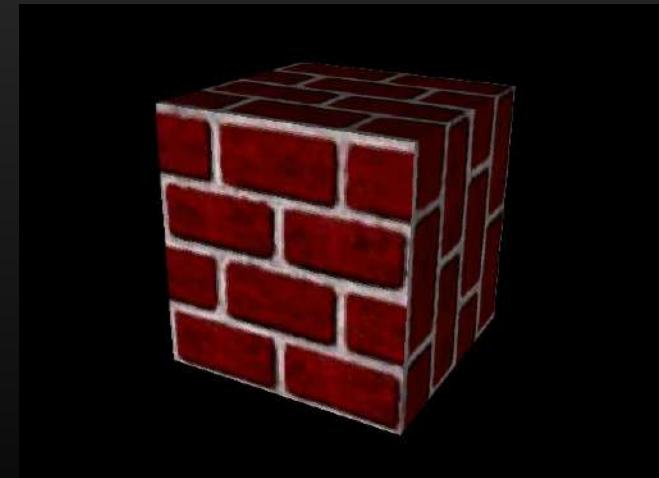
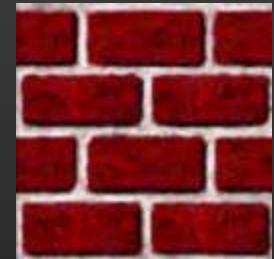
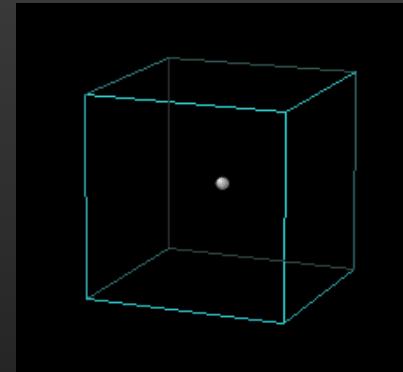


Source: wikipedia

# Texture mapping

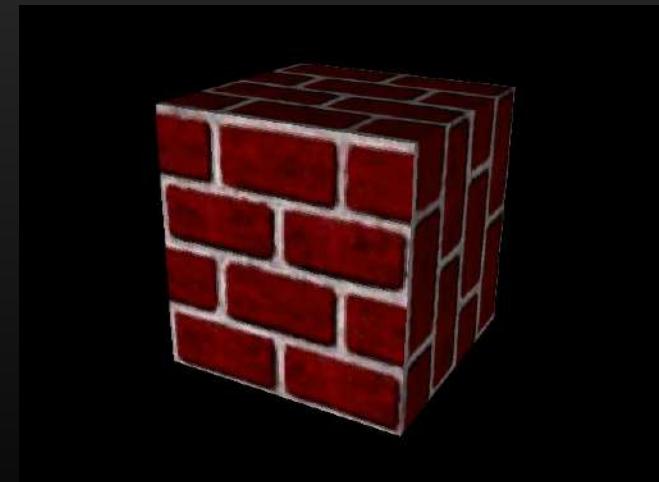
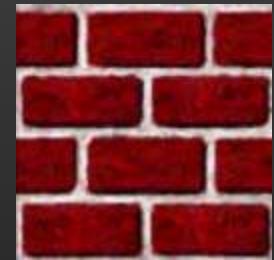
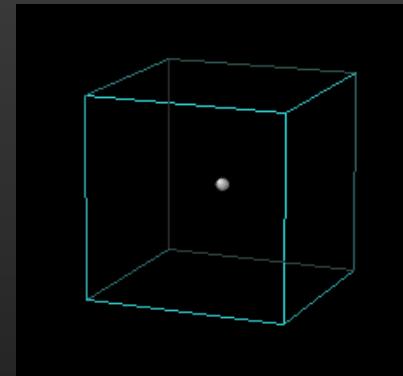
# Texture mapping

- Individually **coloring each polygon** would be tedious
- It would also take **many triangles** to make a realistic object



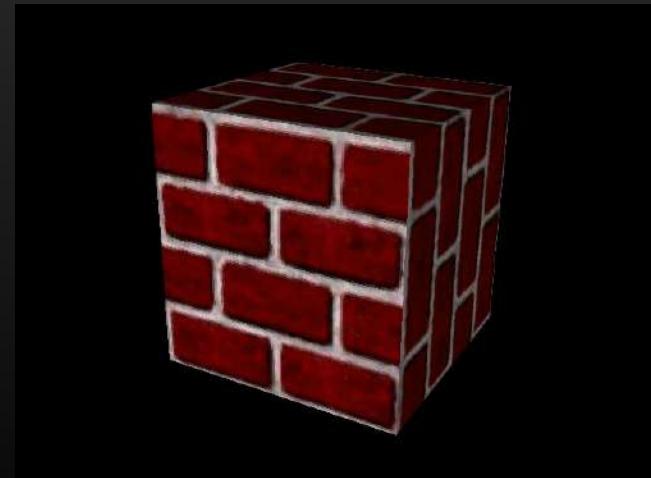
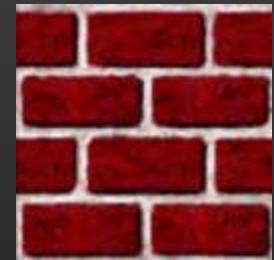
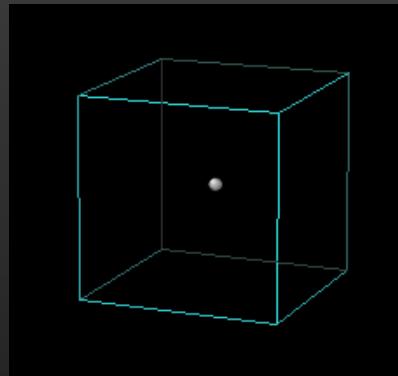
# Texture mapping

- Graphics hardware has the ability to **paint triangles** with sections of a **bitmap image**
  - Called **texture maps** or just **textures**
- Artist tags vertices with **texture coordinates**
  - Specify a **position within the texture**
  - Pixels in the interior of the triangle are painted with the pixels between the pixels of the vertices



# Texture mapping

- Texture is often used to make objects **look complicated**
- When they really have **very few polys**
  - There are no actual depressions between the bricks in this image
- Most modern games **wouldn't be practical** without it



# Diffuse texture map

- The **most common** way to incorporate a texture map
- Value read from texture map is used as the **diffuse color** for the pixel



# Specularity map

- But then you have a **fixed specularity** for the whole object
- If you want the object to be **selectively shiny**
- You need a separate **map for specularity**



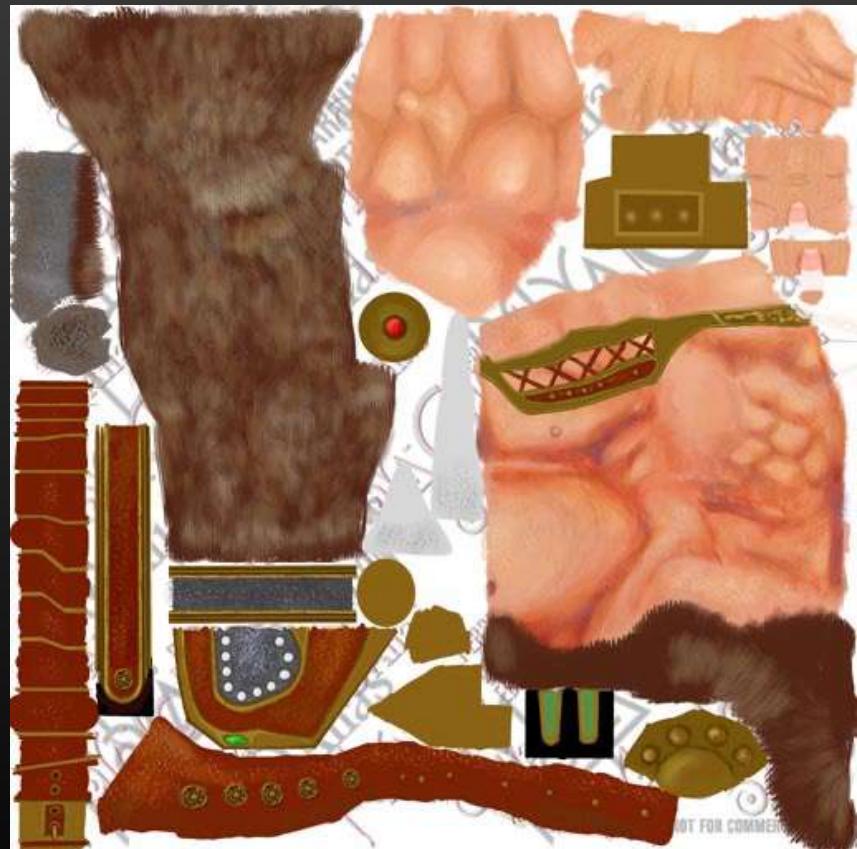
# Texture map representation

- Maps are represented as raw **bitmap images**
  - Possibly with some limited **compression**
- Many graphics cards restrict dimensions to be a **power of 2**
  - Prevents them from having to do a **multiply/divide** operation



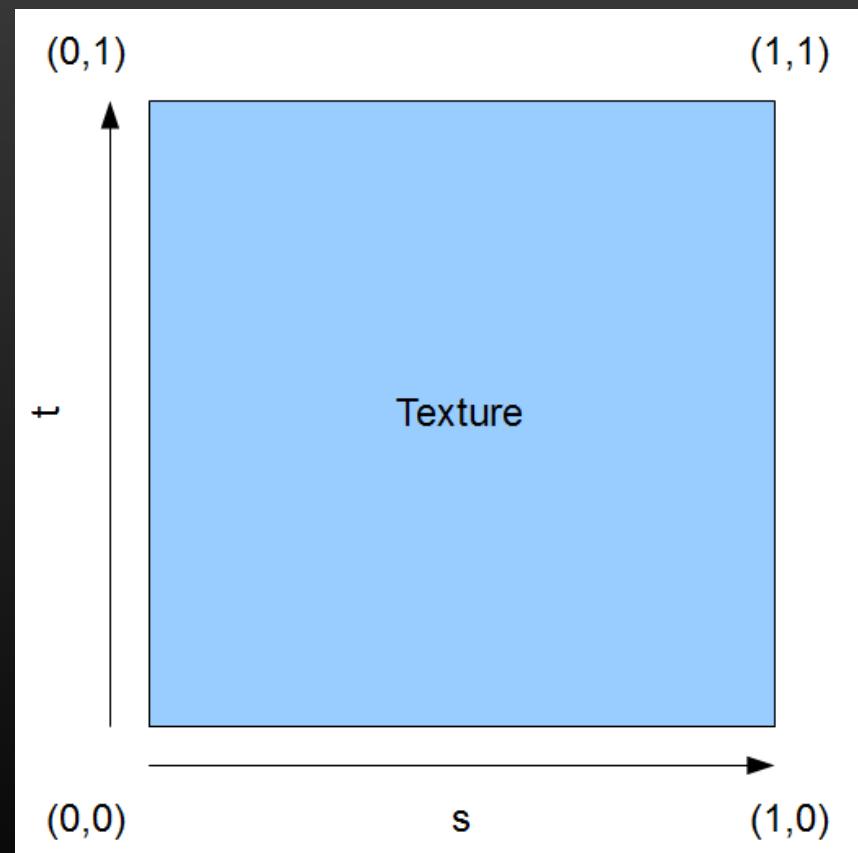
# Packed textures

- Textures for different objects are often placed in the **same texture**
  - To save **memory**
  - To save **time** (changing texture maps takes time)
- Automated tools (**texture packers**) exist to help do this

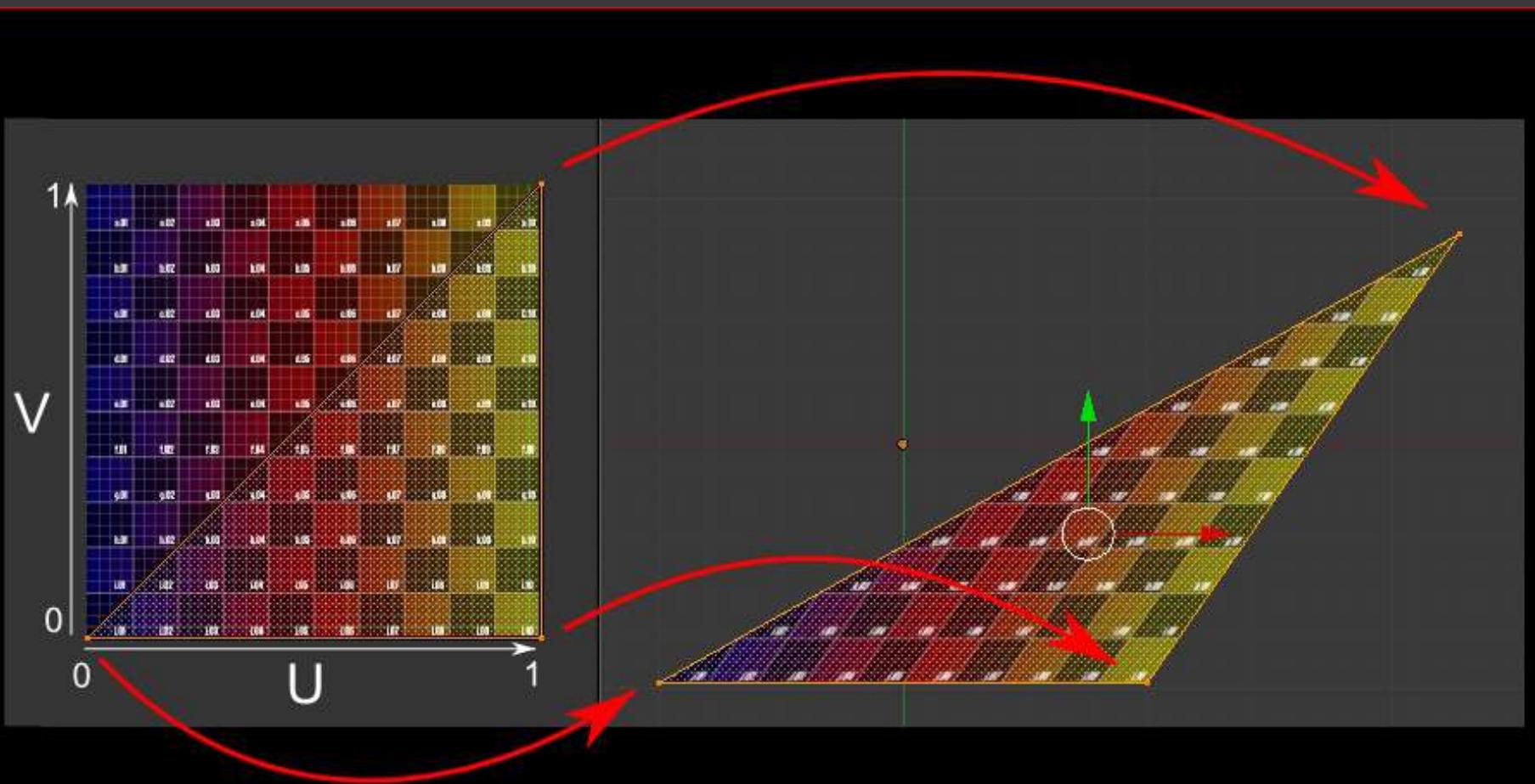


# Texture coordinates

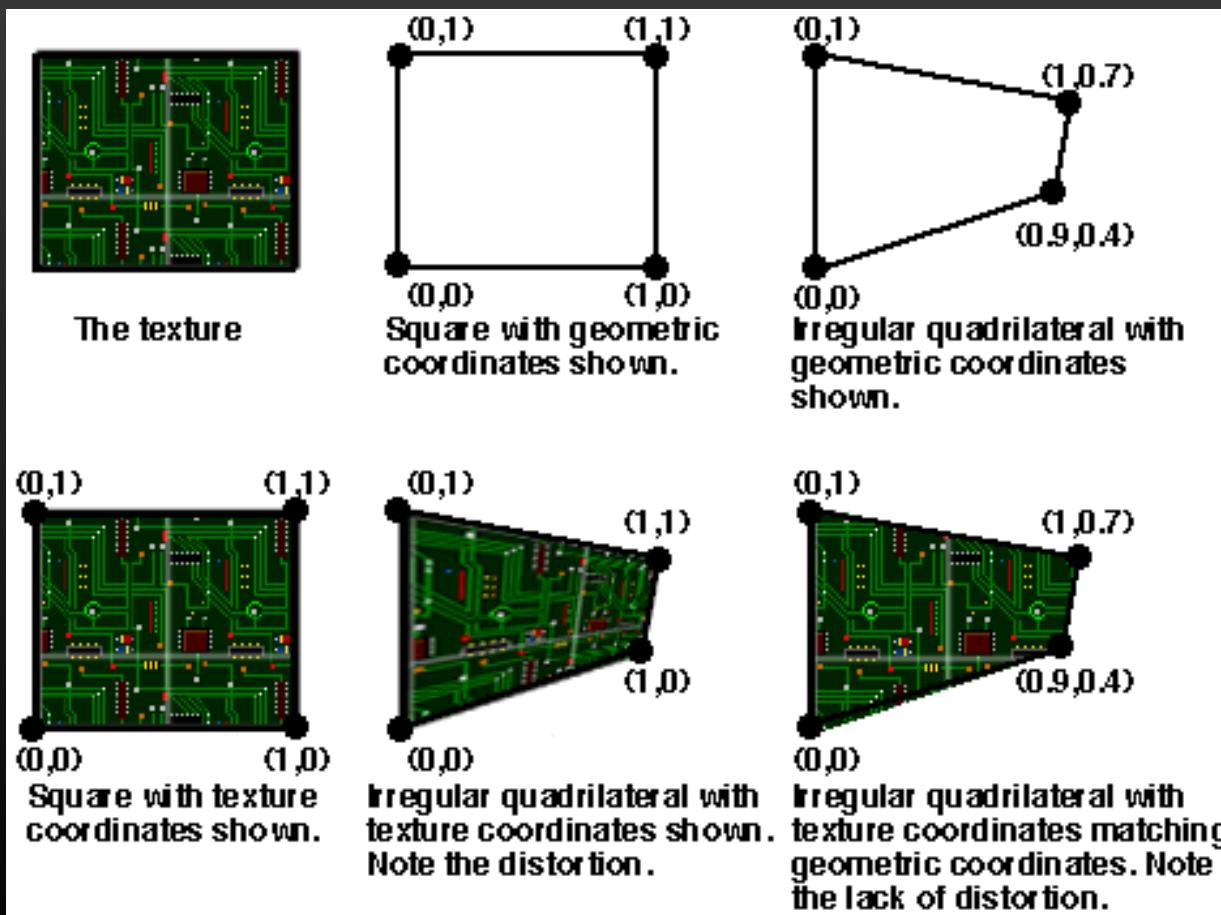
- Each vertex is labeled with its **texture coordinates**
  - **Location in map** it corresponds to
- Pixels in a triangle **interpolate** their texture coordinates from those of their vertices



# Texture-mapped Triangle rendering



# Texture-mapped poly rendering



# The curse of topology

- Texture maps are **flat**
- Most 3D **objects aren't**
- To **map** a flat texture to a curved 3D object we have to
  - **Stretch** some parts
  - **Squish** others
  - **Cut holes** (sometimes)

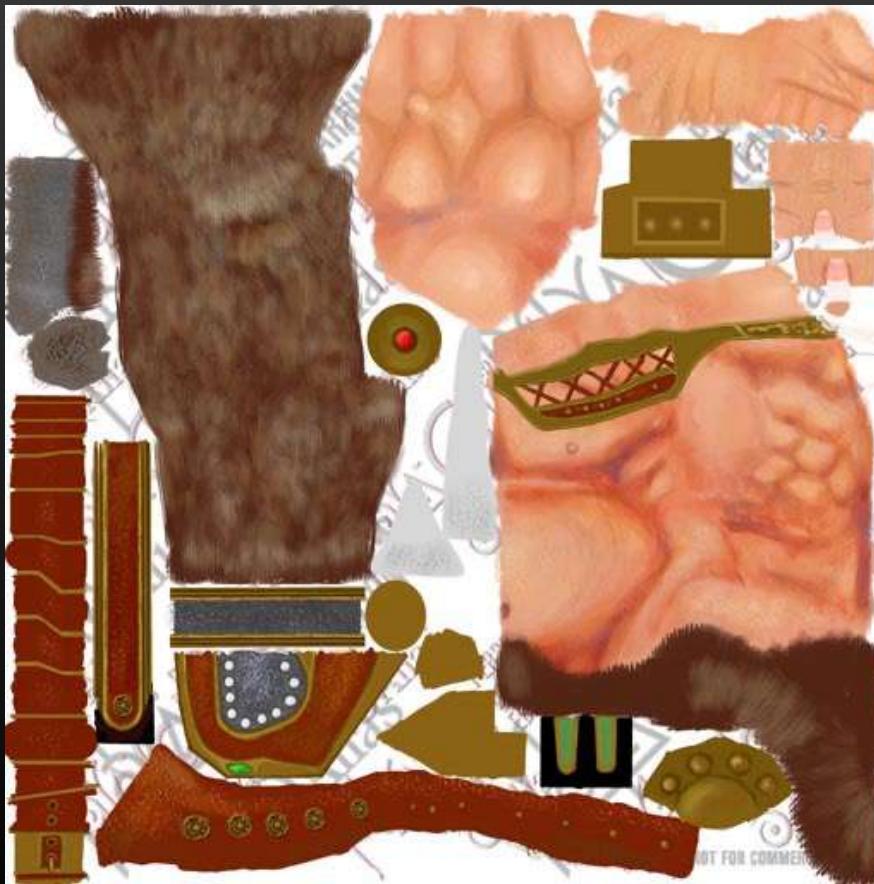


# Mesh

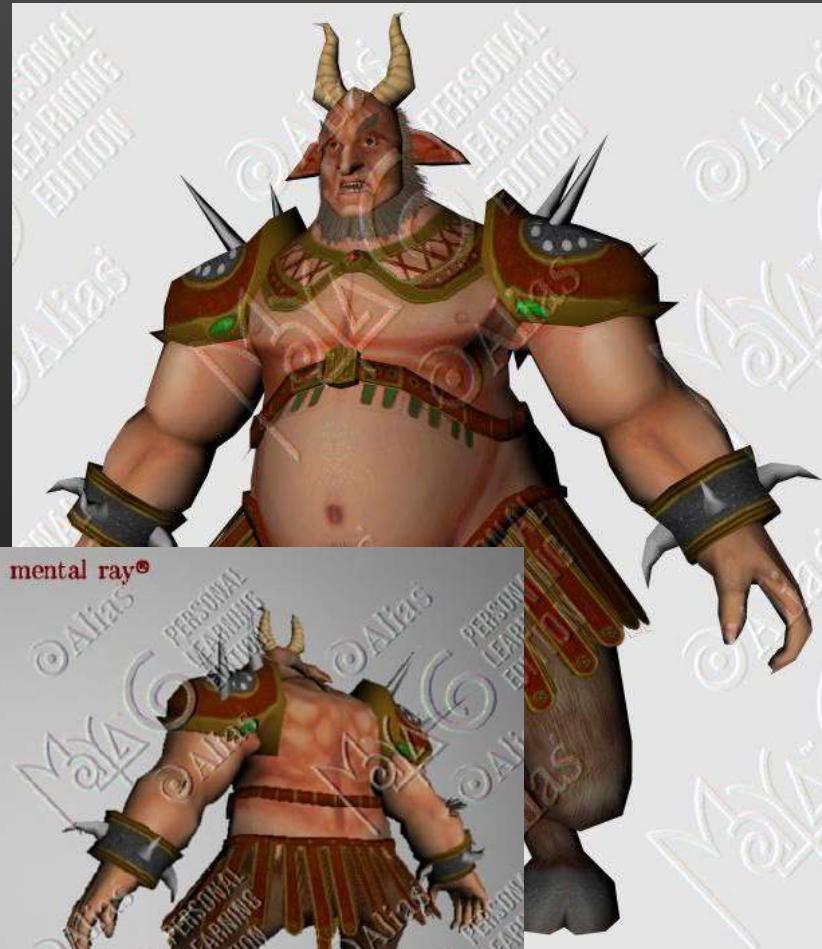


Source: [http://www.randomstuff.org.uk/~laura/Gallery\\_digital\\_art/silenum.html](http://www.randomstuff.org.uk/~laura/Gallery_digital_art/silenum.html)

# Texture maps



# Mapped mesh



# With and without texture



Other stuff

# Normal mapping

- Normal mapping is another way to make a **low poly model** look like a high poly model
- Provide a **texture map for normals**
  - Lets the artist “paint” **changing normals over a single poly**
  - Normals respond to **lighting** as if they were part of the “real” geometry



# Information in a 3D model

## Geometry

- **Vertices** of the mesh
- **Triangles** formed by the vertices
- Vertices are optionally **tagged** with
  - Normals
  - Color information
  - **Texture** coordinates
  - Other **exotic** information (e.g. bone weights)

## Material information

- **Diffuse** color
- **Specular** color
- **Specular** hardness
- **Ambient** color
- **Texture** map
- Other **exotic** stuff
  - Bump map
  - Specularity map

# Transparency

- So far, we've assumed materials are **opaque**
  - Topmost surface **occludes** other surfaces
- Transparent rendering makes everything much **more complicated**
  - Have to **render all surfaces** at a point
  - And **combine** their effects



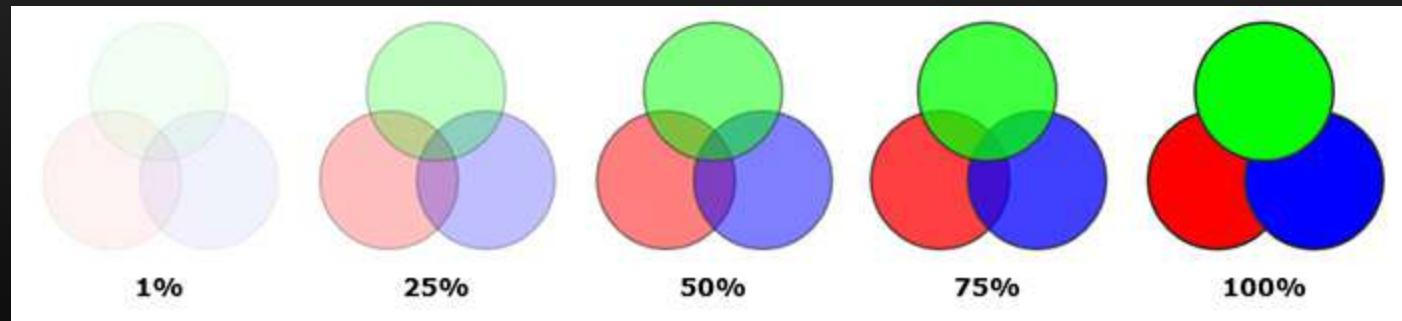
(c) Copyright David Lanier 3D 2006, all rights reserved  
[www.dl3d.com](http://www.dl3d.com)

# Alpha blending

$$O_R = I_\alpha I_R + (1 - I_\alpha) O_R$$

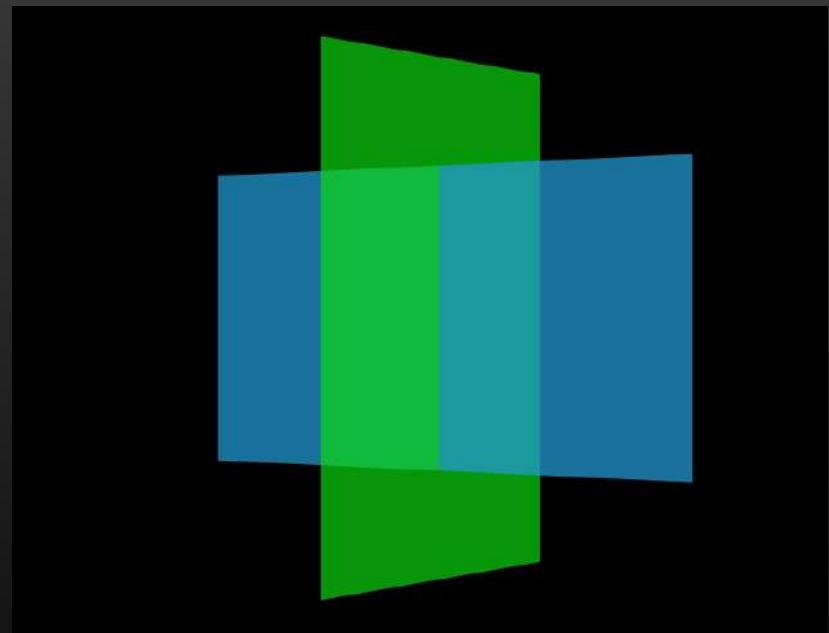
$$O_G = I_\alpha I_G + (1 - I_\alpha) O_G$$

$$O_B = I_\alpha I_B + (1 - I_\alpha) O_B$$

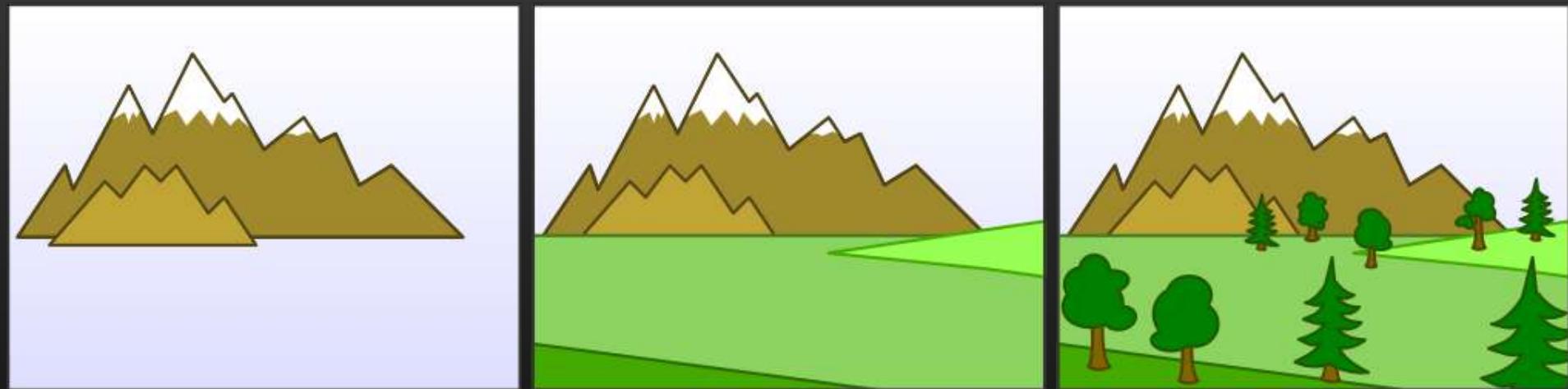


# Depth-ordering

- The problem with alpha blending is that it's **not commutative**
  - Color for **A in front of B**
  - Is different from color for **B in front of A**
- So we not only have to **render all** the surfaces
- We have to render them **back to front**

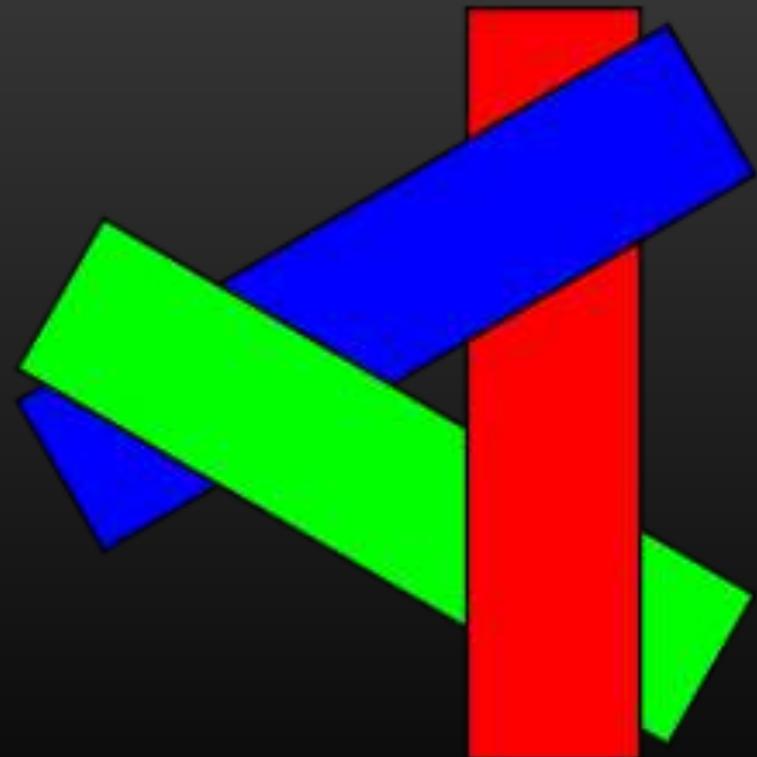


# Painter's algorithm



# Depth-sorting polygons

- In the film industry, this is done by **sorting individual polygons** by depth
  - The problem is, you have situations where a polys **overlap in depth**
  - So you may have to **split polys**
- This just **isn't practical** for games



# Depth-ordering

The typical way this is handled in games is to

- Render all **opaque surfaces first**
- Then render **transparent surfaces**
- Hope for the best
  - Tell your artists to make sure **no transparent surfaces overlap**

