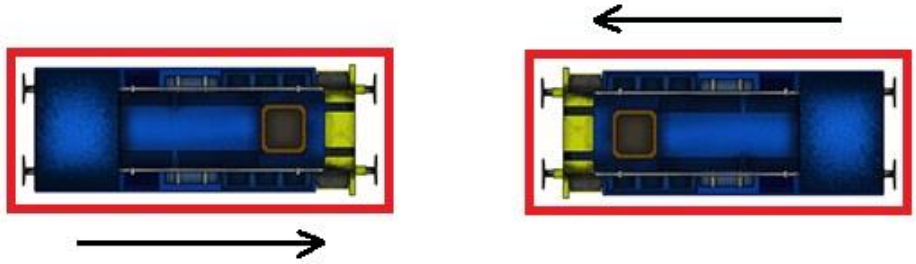
Physics Tutorial 4: Collision Detection



New Concepts

- Broad Phase and Narrow Phase
- Convex and Concave Shapes
- Generic Collision Detection Algorithms
- Separating Axis Theorem

The Broad Phase and the Narrow Phase

Broad Phase Premise

- Vast majority of our objects will not be colliding – common sense
- Vast majority of our objects are of a size and location which means it isn't even remotely possible they're colliding
- What we need to do is find a quick way of sorting through collision-capable objects in our environment, to ensure that as many of the can't-collides as possible are pruned before we get to the narrow phase can-collide checks

Broad Phase Premise

- Somewhat analogous to frustum culling in graphics
- We'd NEVER use frustum culling for physics
- Because whether or not objects collide can be really, really important ('falling through the floor' important) whether or not we're looking at them
- And, if we were making a game based on the Dr Who episode "Blink", doubly so

Broad Phase Premise

- What would happen if we didn't perform collision checks until objects were in view?
- Consider:
  - A load of objects are overlapping by a considerable margin (not the small margins we deal with normally)
  - We turn and bring them into view
  - Collision response kicks in
  - The world EXPLODES
- Reason for this is our constraint-based system – it reacts based on the idea that a change has occurred over a single time step. It will assume that this massive overlap is the function of a single time step

Broad Phase Premise

- Simple but effective approach is to compare bounding boxes, spheres or capsules around objects
- These checks are cheap (more on that later!)
- They cull collision pairs that definitely can't happen

- While the checks are cheap per pair, relative to a complicated narrow phase check, they're still being applied to a N^2 problem, where N is the number of collisionable objects in our environment.
- It is far more efficient to group nearby agents together somehow, to ensure that objects on the left hand side of a game map aren't compared against objects opposite (i.e., $k \times n^2$, where $kn = N$)

BSP Trees and World Space Partitioning

- If we know objects in our environment are going to be reasonably ‘normally’ distributed – an even distribution throughout the game world – we can use a fixed partitioning of our world space.

- This is also appropriate for certain checks, even if we’re employing more advanced techniques for other, mobile objects (if, for example, our environment has lots of stationary but complex colliders)

- Fixed world space partitioning is trivially easy to implement, so long as you keep in mind that objects might be occupying more than one region

BSP Trees and World Space Partitioning

- Problem with fixed world space partitioning is that if the environment is highly dynamic it can be worse than doing nothing
- Because all of our entities might end up in one defined ‘region’, all we’re doing is making things MORE expensive, not less, because we’re adding a sorting algorithm which doesn’t reduce our problem at all.
- There has to be a better way…

BSP Trees and World Space Partitioning

- All we are saying, is give Octrees a chance…
- A more versatile approach, and one favoured in industry, is binary search tree based partitioning
- Recursively subdivides the world based on occupancy of regions
- So no region contains more than a set maximum number of entities, however those entities may be divided throughout the environment

BSP Trees and World Space Partitioning

- Not a perfect solution
- Can be tricky to implement
- Can be too recursive, if objects are very strongly clustered
- Scale can be an issue, with bigger objects occupying several regions at once if present in an environment with many smaller objects
- All that said, very powerful performance enhancement to our physics system

Sort and Sweep

- Essentially this algorithm can be bolted on to any broad phase checks
- Sorts the entities along one axis, based on the positions of their bounding volumes
- Works from one end of the axis to the other, dropping items from comparisons as we move beyond their furthest point along the axis.
- See the handout for more details

Broad Phase as a Hierarchy

- Once we've sorted our environment to minimise the number of bounding volume checks (those bounding box/sphere/capsule based checks we mentioned earlier), we need to actually check those.

- Object pairs that are still possibly colliding after –those-checks, are passed on to the narrow phase

- In that sense, the broad phase can be considered somewhat hierarchical. At the highest level, we're sorting objects by spatial location. At the lowest, we're performing cheap checks on over-estimated volumes which allow us to prune the list still further.
- The algorithms from yesterday can help here!

Narrow Phase as a Hierarchy

- The same in part is true of narrow phase – there's some blurrioscity. We can use equations which are capable of providing narrowphase collision resolution data as broadphase culling checks.
- We can perform, if we want, simple collision checks not dissimilar to their broad phase counterparts – in many cases, these will provide enough useful data to resolve collisions believably
- For objects where that really isn't feasible/appropriate, we have more advanced checks which can be made (and will be the subject of today's lecture)

Narrow Phase Checking

A Note on Collision/Interface Detection and SAT

- At this point, having covered broad phase collision checks, and some simple algorithms which might handle trivial objects at narrow phase, we are moving into the area of narrow phase proper

- We're reminded of how we're defining collision:
- If there exists a point on the surface of object A which lies within the volume defined by object B, objects A and B have interfaced/collided

A Note on Collision/Interface Detection and SAT

- There are, quite literally, hundreds of algorithms which address this problem.
- It's one of those cases in computing where there's not really a single 'right answer'
- The most popular algorithms for interface detection in real-time are generally accepted to be the Lin-Canny algorithm (basis of I-Collide) and the GJK algorithm.

A Note on Collision/Interface Detection and SAT

- The obvious question at this point is “Why learn SAT, then, instead of Lin-Canny/GJK?”
- The answer is two-fold.
- First, SAT is as much as anything a learning tool which permits the programmer to visualise the problem, and to visualise each step they take in solving it.
- As you’ll see today, the process we go through in solving a problem through SAT is intuitive (if anything in geometry is intuitive)

A Note on Collision/Interface Detection and SAT

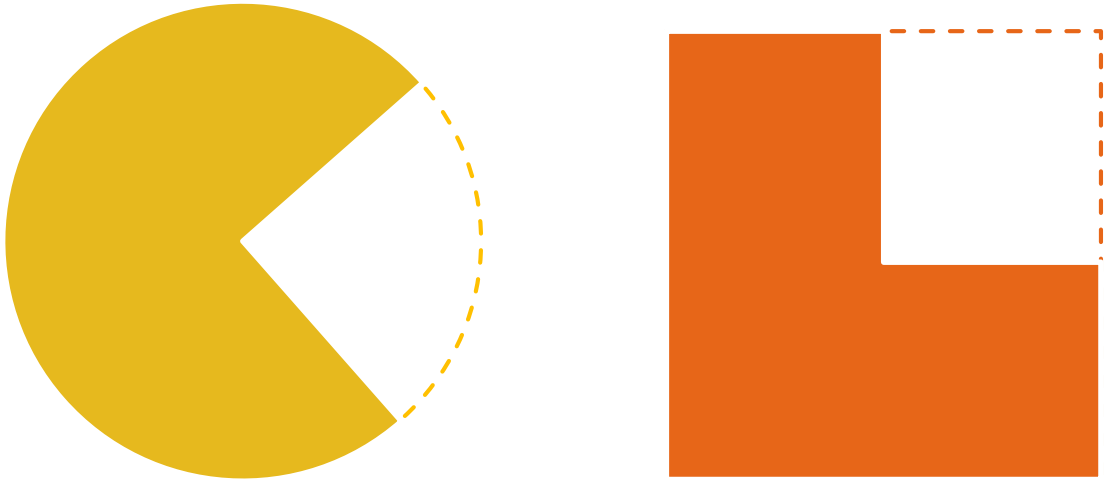
- It's important that the process be intuitive because we're using the algorithm **as** a learning tool.
- If the process were more abstract, it'd be harder to learn anything meaningful about what the geometry is actually doing.

- The other reason is that it's more straightforward for you to implement than many other available algorithms. You only have two and a half weeks to get the coursework done – you don't want to spend all of that time encoding a physics system without any clue if it's going to work correctly at the end of the day

Convex and Concave Shapes

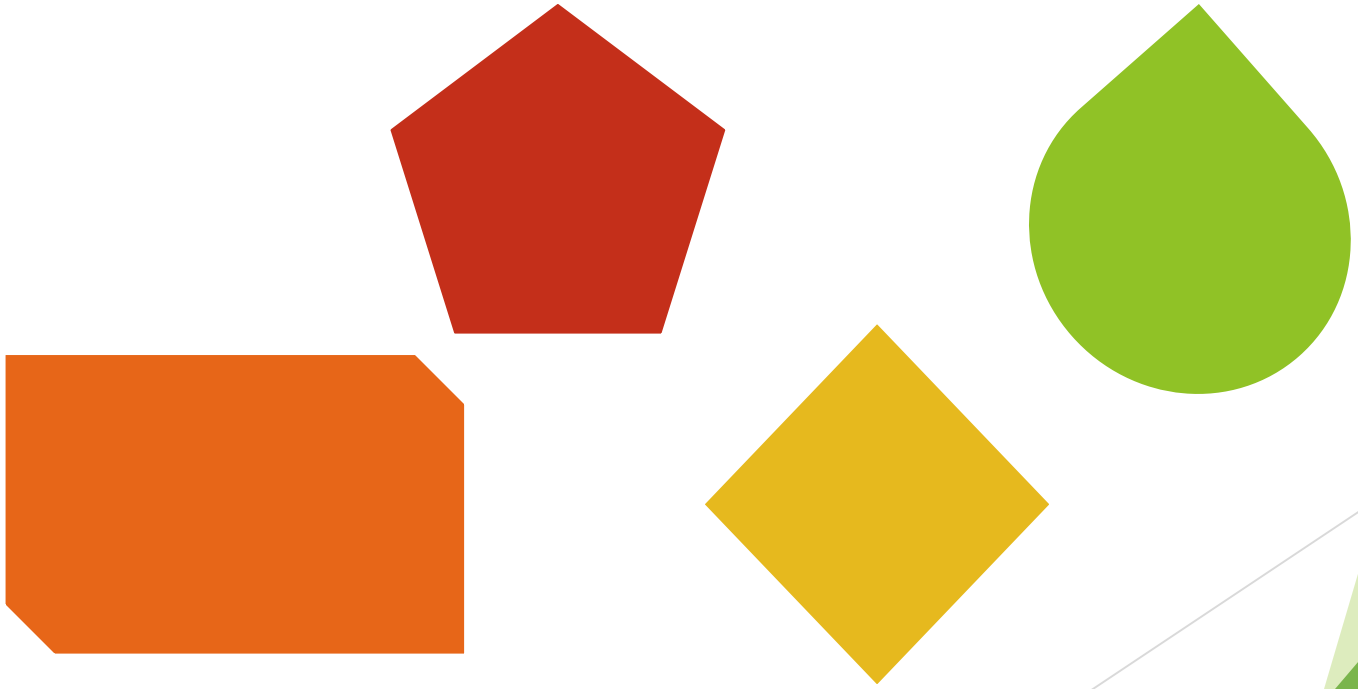
Concave

- You should recall from high school that a concave shape is one which has a hollow, e.g.



Convex

- By contrast, a convex shape is any shape which is not concave – i.e., a shape which does not have a hollow:



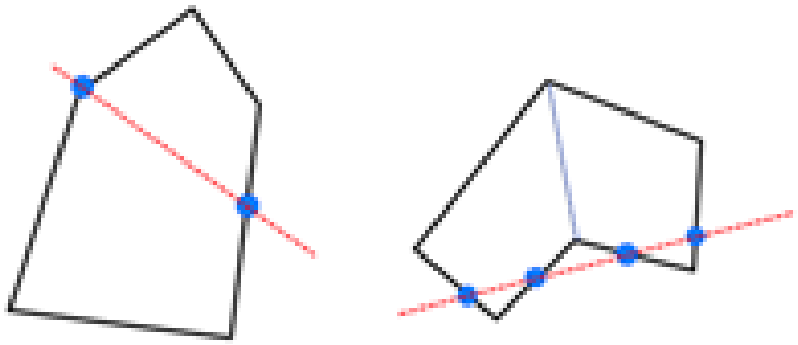
Identifying the Difference

- Visually, it's obvious when an object is concave or convex.
- Algorithmically, should we need to identify which of the two an object is, we can employ a simple rule:
- **If a line is drawn through the shape, the shape is convex if the line has two points of intersection; if it has more, it is concave**
- **Note:** The line cannot be a tangent to the object, or it will only have one point of intersection irrespective of the shape of the object.

Identifying the Difference

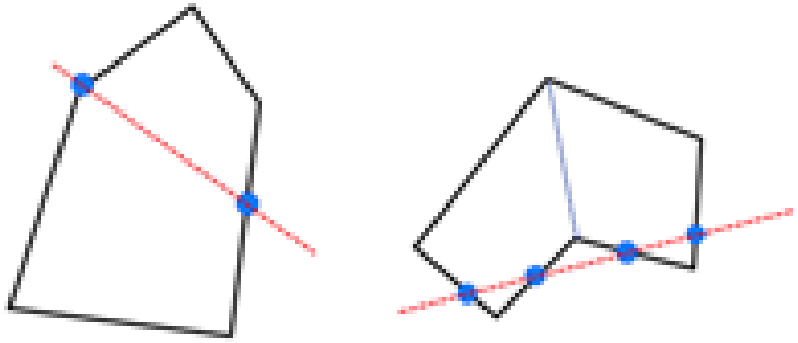
- A SAT-based approach cannot account for collisions between concave objects without significant extensions/additions

- BUT all concave objects can be broken down into a number of convex objects – consider the right-hand object below:



Identifying the Difference

- This process of decomposition into convex objects can be automated (for those looking for a real challenge in their coursework), but the process for doing so is very daunting to implement
- For this reason, the tutorial series assumes that any concave objects in your environment are manually broken down into constituent convex elements

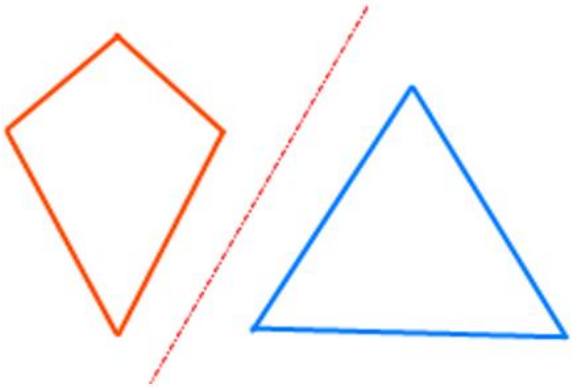


Separating Axis Theorem in 2D

Separating Axis Theorem: Premise

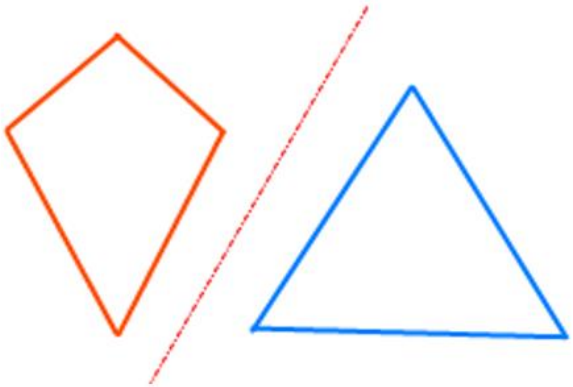
- Separating Axis Theorem (SAT) states that:

- If two convex objects are **NOT** colliding, then a line (or plane, in 3D space) can be drawn between them which does not intersect with either



Separating Axis Theorem: Premise

- The corollary to this, and the principle our collision detection is based on, is that if we can find a single case where we can draw a line/plane between two shapes without intersecting either of them, we can prove that the two shapes do not collide



Separating Axis Theorem: Practise

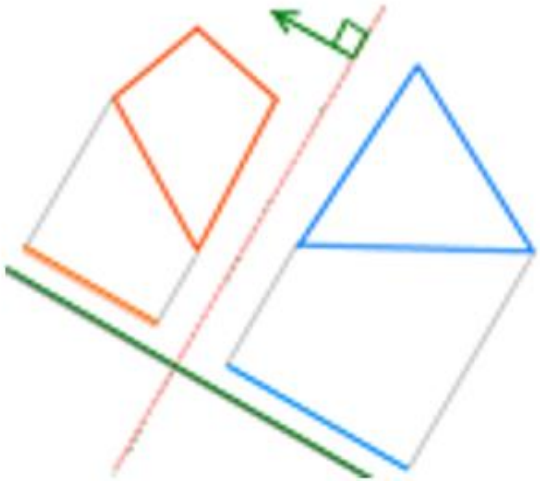
- Let's envision our objects as collection of lines/planes – not hard, since that's all anything is in a video game
- Following on from our talk about spheres intersecting with a plane in the last tutorial, we should be able to see how we can use the same check to help here
- We can achieve our intended effect by:
  - Defining the 'axes' we wish to test
  - Projecting all points of each shape along each tested axis

Separating Axis Theorem: Practise

- Projecting a point gives us a single value describing the distance of that point along the axis being tested.
- The maximum and minimum of those values defines how much of that axis is occupied by our object – and that allows us to determine if, in that axis, two objects overlap.
- Let's look at an example

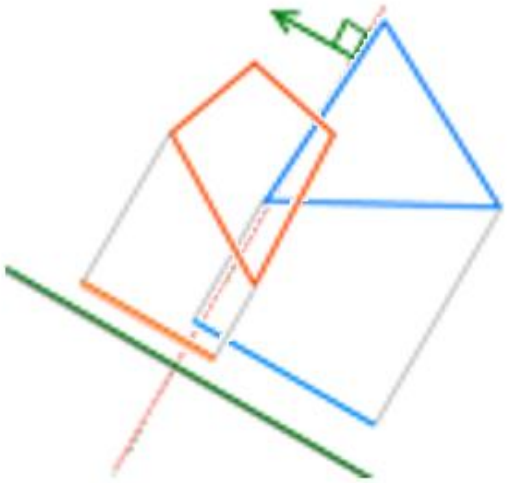
Separating Axis Theorem: Practise

- Visualise the axis we're projecting against as the green line (the normal of the red line, projected infinitely)
- Notice the blue and orange lines of projection here – that's the maximum projection of the points of each object along that axis
- There's a gap – they don't overlap in this axis, ergo they haven't collided



Separating Axis Theorem: Practise

- We can see in this slightly altered image that were the objects actually colliding, there WOULD be overlap of their projections on that normal



- This also helps is get a better understanding of why we're projecting onto the normal of the test axis in order to determine if the objects overlap, which can be counter-intuitive

Separating Axis Theorem: Practise

- The code to find the minimum and maximum projections along a vector is provided in SphereCollisionShape and CuboidCollisionShape

- We should pay attention to the Cuboid case during today's practicals, and devote some time to understanding what the code is doing, as cuboids are the generic convex polyhedron

Separating Axis Theorem: Practise

- Projection of point a along axis b is given by the formula

$$proj_a b = \frac{a \cdot b}{|b|} \frac{b}{|b|}$$

- Normalising b lets us substitute the equation down to

$$proj_a b = (a \cdot \hat{b})\hat{b}$$

- The distance of the point along the axis is the first term, so

$$distance = (a \cdot \hat{b})$$

Potential Separating Axes

Potential Separating Axes: The Problem

- We've now determined how SAT tells us when two objects haven't collided
- But we've not addressed a core issue with SAT – and, indeed, the algorithm's main weakness
- How on earth do we know what axes to test against?
- We can't consider every axis, there's potentially an infinite number of them…

Potential Separating Axes: The Solution

- Thankfully, there aren't really an infinite number of axes to check – because we're dealing with polyhedral objects. Spheres notionally have an infinite number of axes to check (more on that in the next tutorial), but a twenty-sided die (for example) only has a finite number we need to consider

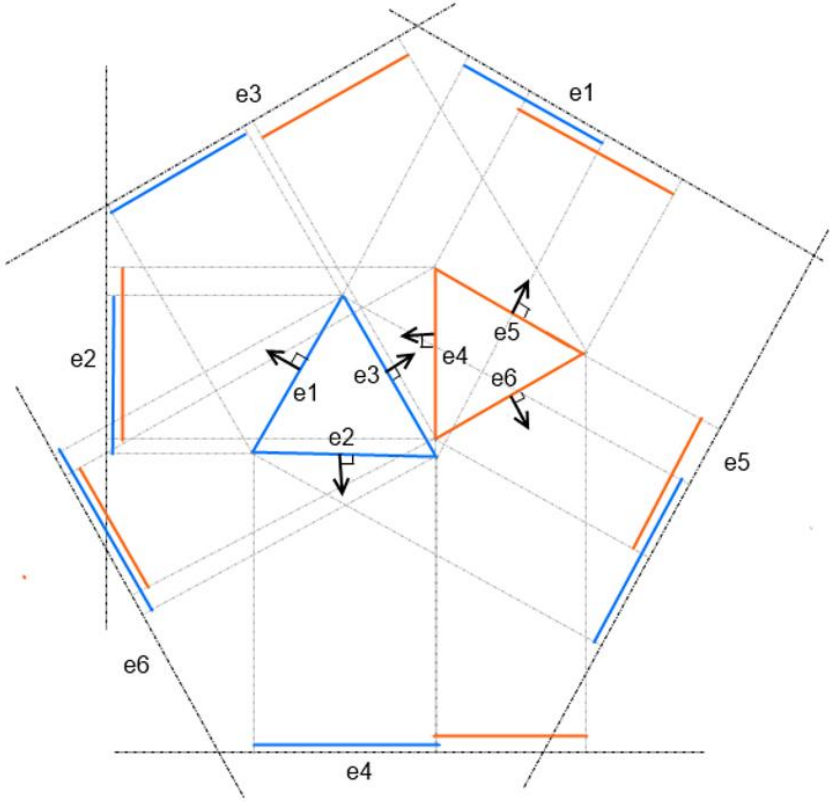
- We assume every object to be made up of a series of lines or faces – just like most objects in our game will be.

Potential Separating Axes: The Solution

- This means that the number of collision points is limited to each flat face. (Important fact!)
- As a result, we can take the normal of the faces as potential collision axes.
- This can be a little counterintuitive, so let's consider an example…

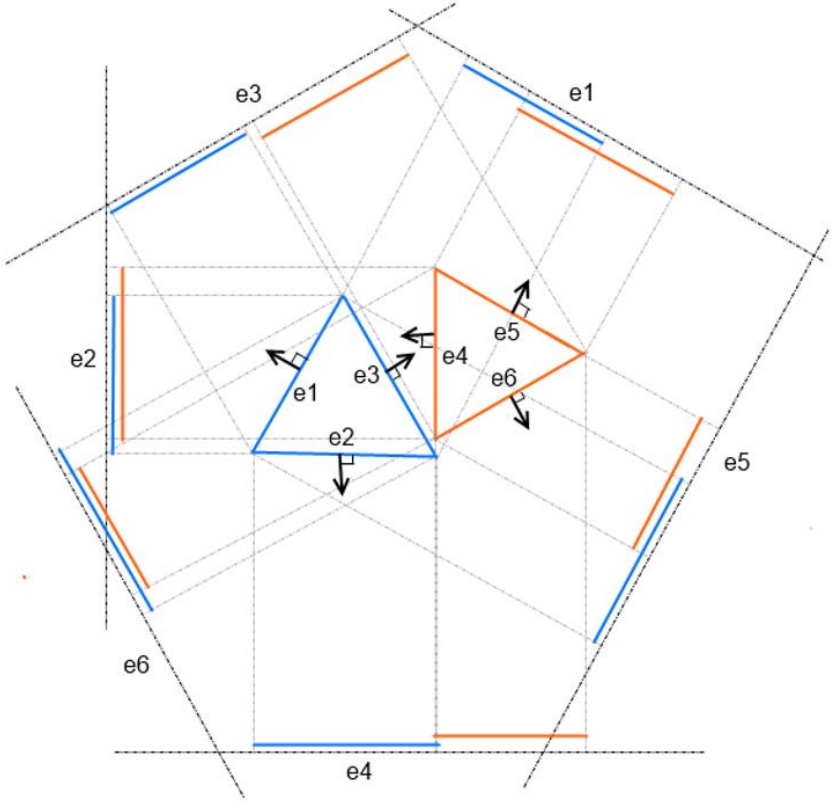
Potential Separating Axes: The Solution

- The figure below shows a check between two triangles



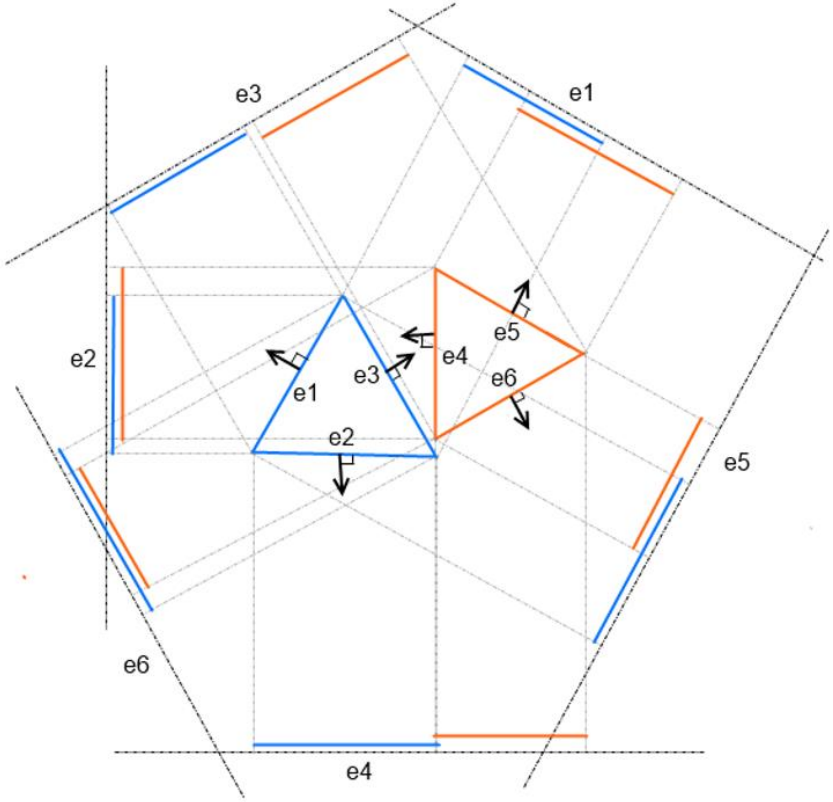
Potential Separating Axes: The Solution

- Let's consider the axes in turn, as generated by the faces



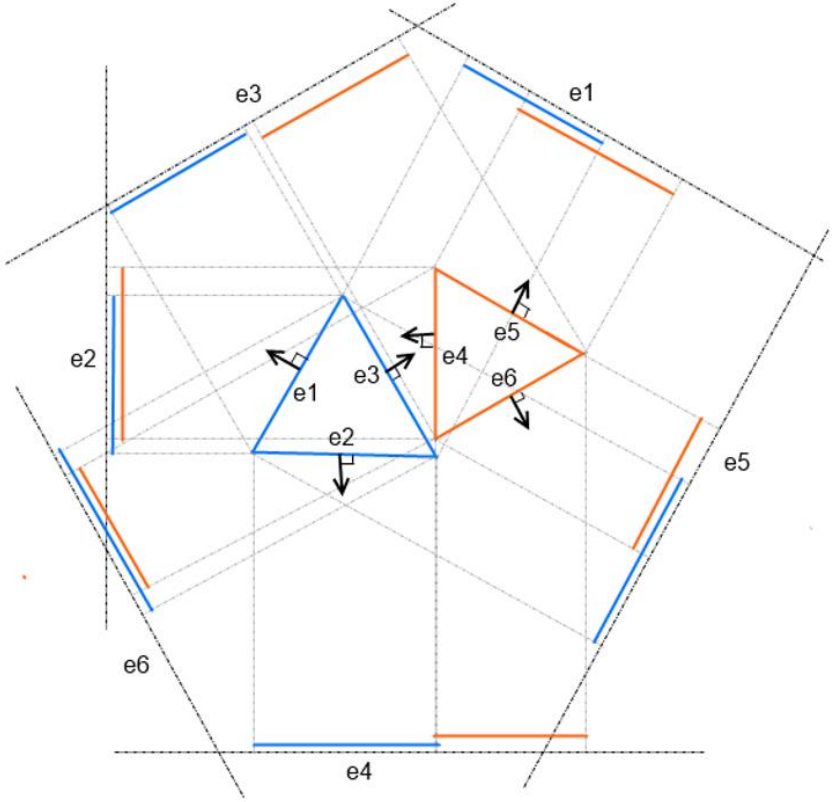
Potential Separating Axes: The Solution

- We'll see there are six axes to check, one for each face



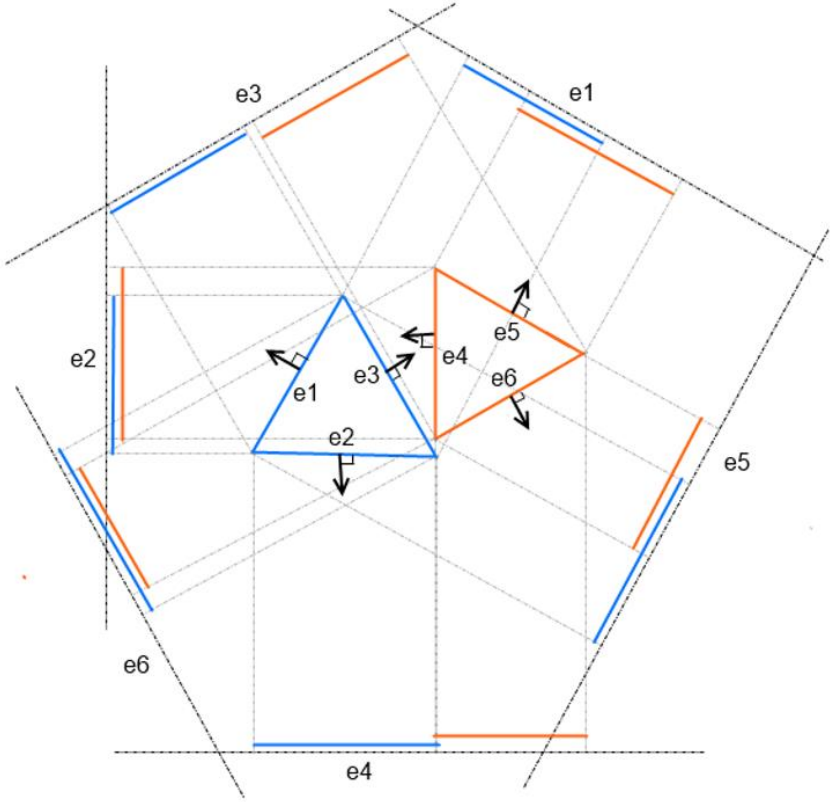
Potential Separating Axes: The Solution

- The two objects intersect on all axes except one – e3



Potential Separating Axes: The Solution

- As such, there has been no collision



Potential Separating Axes: Optimisations

- Just like AABB, our SAT axis check can break out if any of its conditions aren't met – e.g., the example before could have broken out at e3, because that proved no collision
- Parallel Axes don't need considering (common sense). How many axes would nee considering for the d20?

SAT Key Feature Summary

Key Features

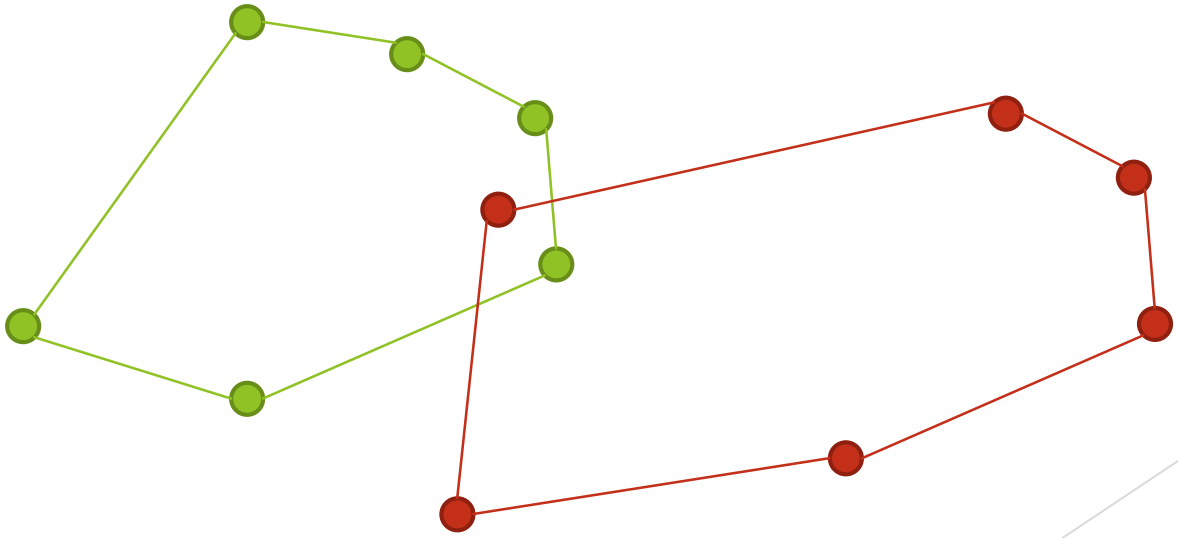
- If there exists an axis in which two objects do NOT overlap, we can prove they do not collide
- If no axis exists in which they do not overlap, we can safely assume that they have collided (interfaced)
- The number of possible axes to check is the same as the number of faces of the objects, summed
- Parallel axes can be ignored
- As all conditions must be met for confirmation of an axis, check can break out when any one condition isn't met without checking the remainder

Generic Interface Detection

The problem is simple – why are the solutions complex?

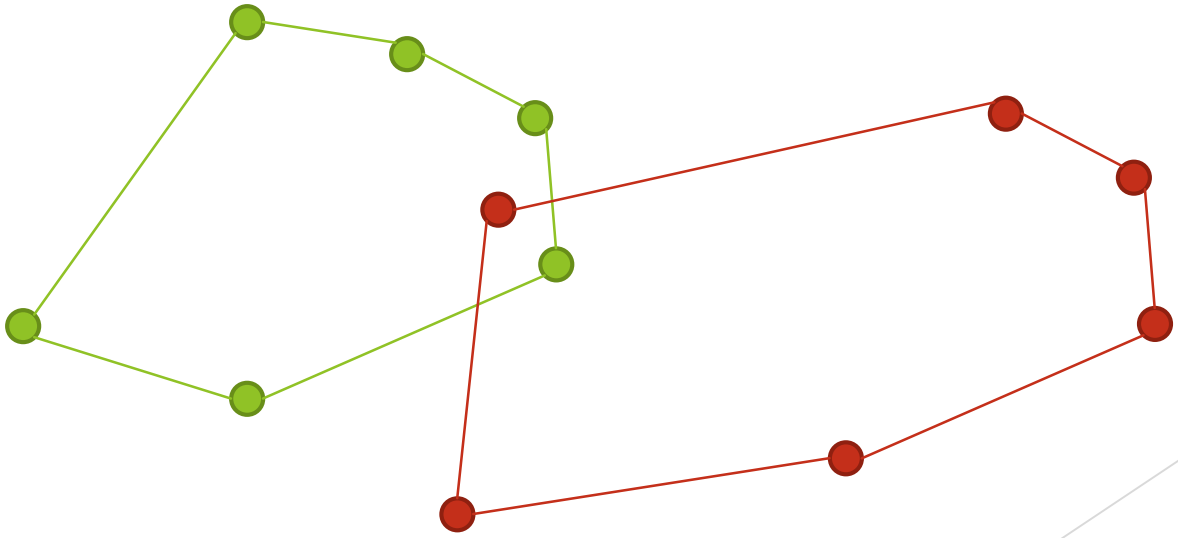
- Let's pause for a moment and think about the problem we are trying to solve

- Envision the scenario below:



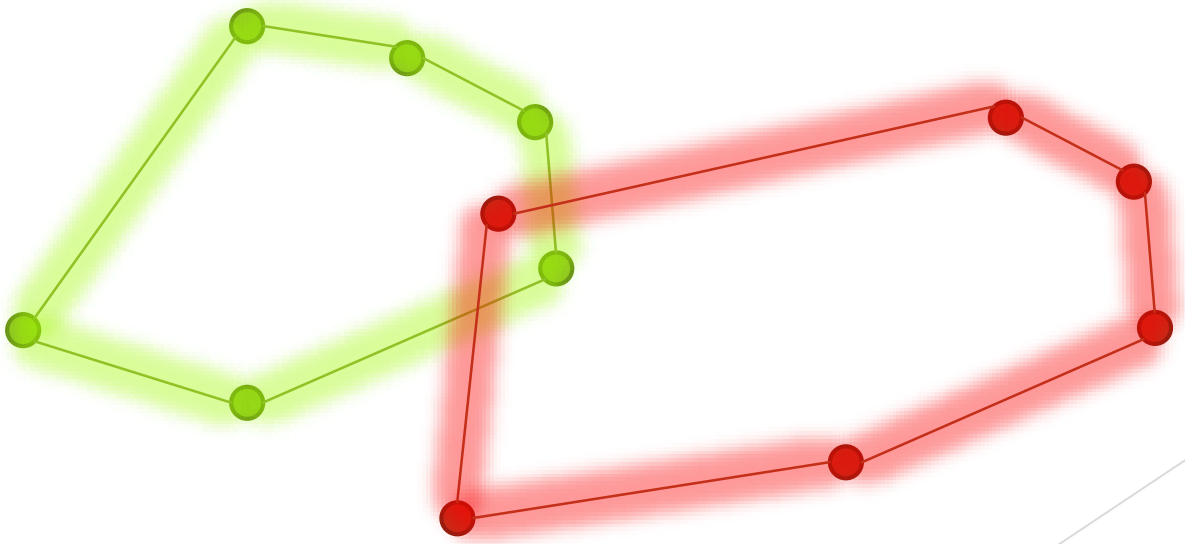
The problem is simple – why are the solutions complex?

- Just going off what we knew yesterday of determining whether a point lies beneath a plane, there's an obvious approach to solving this scenario.
- We'll explore this approach over the course of the next few slides



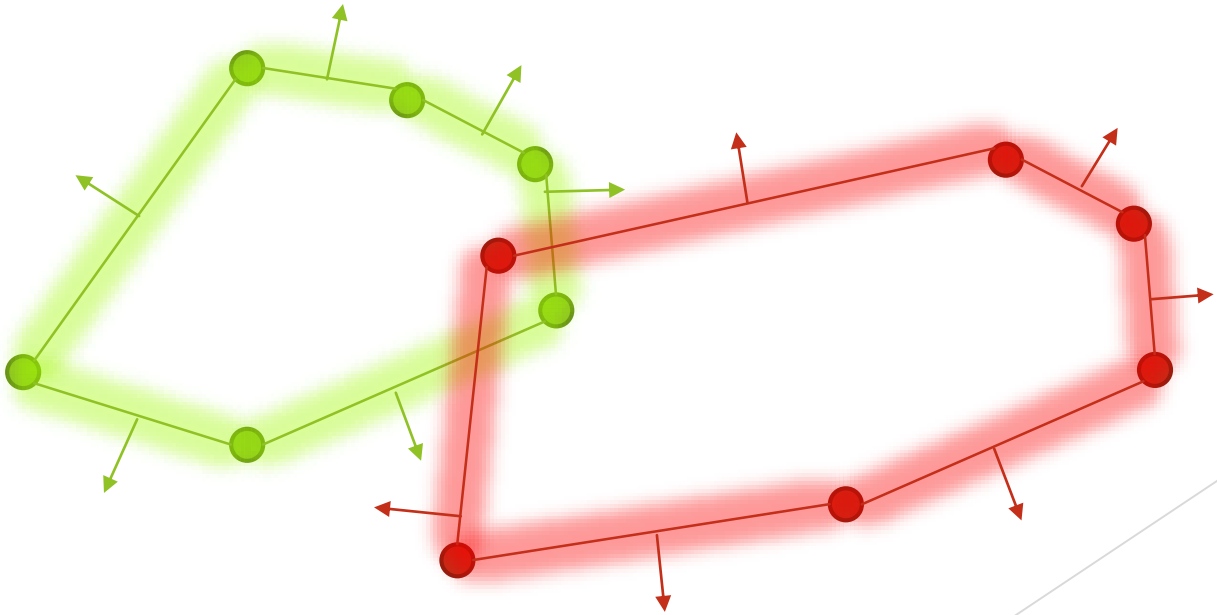
The problem is simple – why are the solutions complex?

- So, what we need to do first is define the lines (planes, in three dimensions) that represent our objects.
- We do that, in this example, twelve times



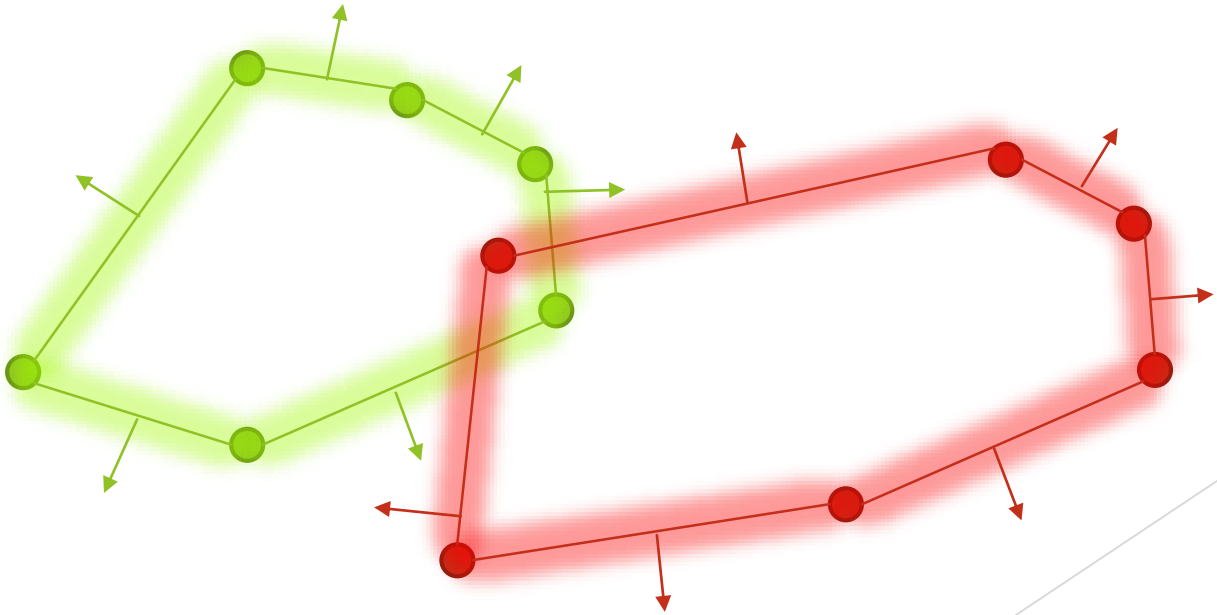
The problem is simple – why are the solutions complex?

- By defining these planes, we obtain their normal
- And now we have the planes' normals, we're ready to actually perform our check.



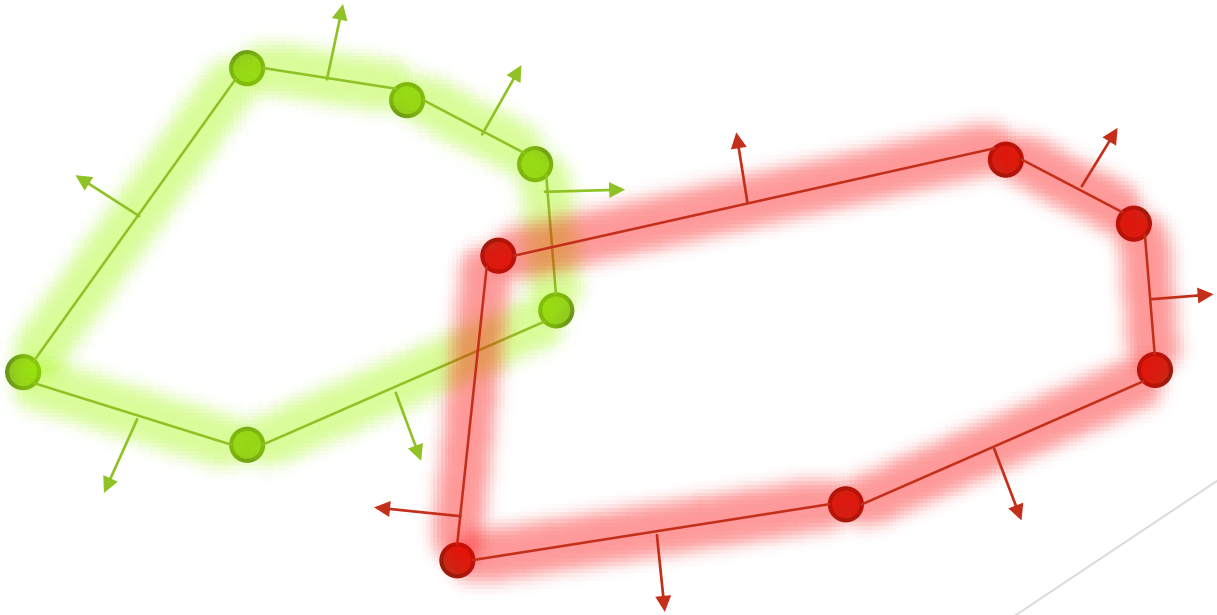
The problem is simple – why are the solutions complex?

- We can now compare every point in the green object against every plane defining the red object
- If a point exists inside **every** plane in the red object, there is an interface



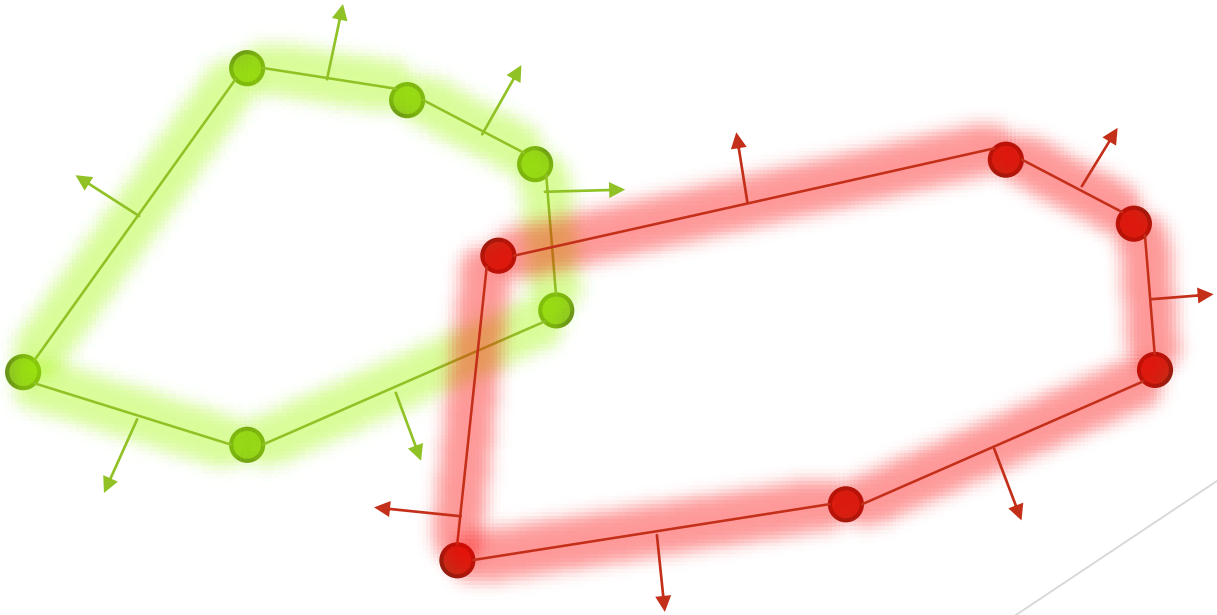
The problem is simple – why are the solutions complex?

- By using this N-squared approach, we guarantee that, for any object, we can determine whether or not there's an interface
- But this is **lacking the accurate collision normal** – for collision response – which can be extracted using SAT



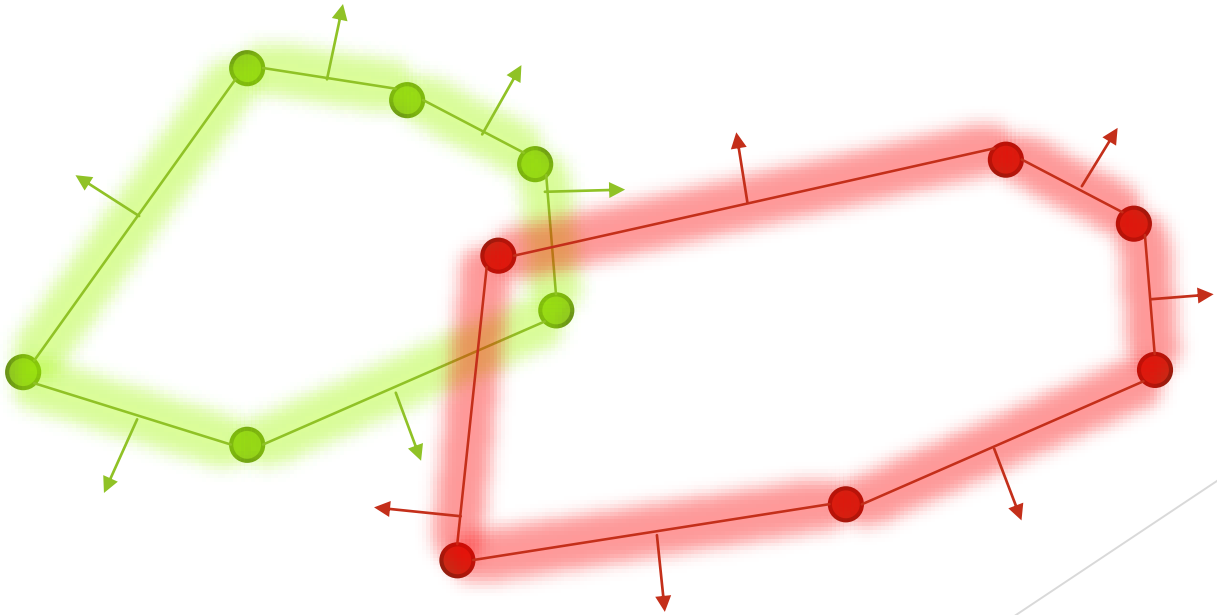
The problem is simple – why are the solutions complex?

- That is the nature of SAT – and, indeed, any algorithm, is a trade-off. We perform a more computationally expensive operation in order to obtain a more valuable result – in this case, data we can meaningfully employ in the final stage of our physics update.



The problem is simple – why are the solutions complex?

- This should reiterate the importance of elements like the broad phase culling, and the reason we use simple hulls to represent even complex objects in our environment



Edge Cases

What are edge cases?

- Edge cases are scenarios in which an algorithm can't be applied to solve without some external reasoning.
- A simple example would be the definition of a chess board.
- You can define the connections of a chess board, for the majority of squares, as being +1 (right), -1 (left), +8 (up), -8 (down)
- But that doesn't take into account the board's edges

Are edge cases a problem?

- Not really.
- They happen all the time.
- Sometimes, the solution is external reasoning about the scenario (i.e., If square == 1, then… If square == 9 || 17 || 25…)
- Sometimes, the algorithm itself can be extended to account for edge cases
- Where possible, employ algorithms with allowances for edge cases already built-in

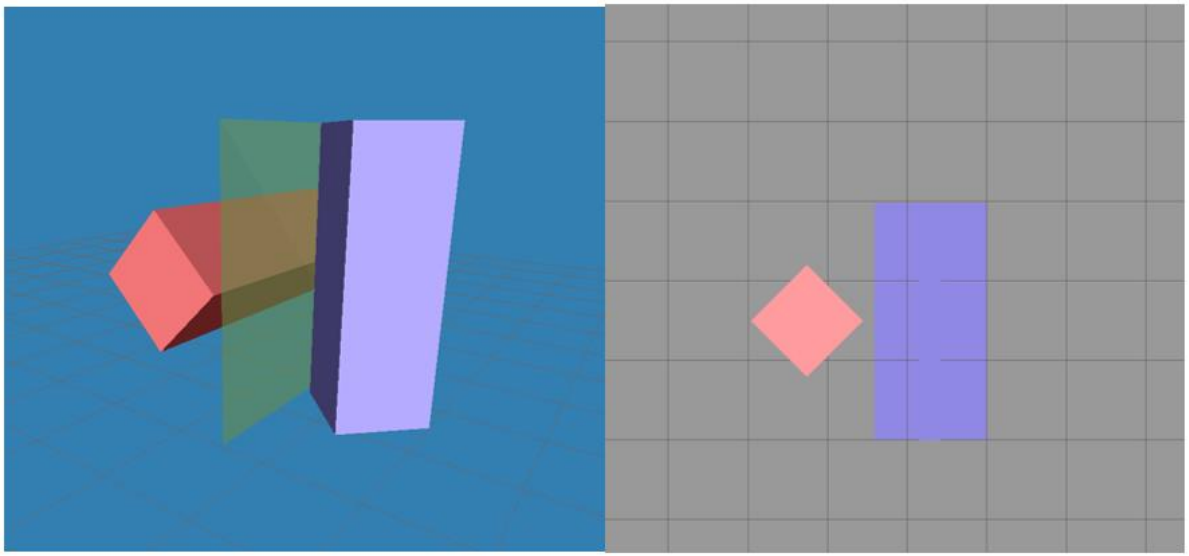
SAT in 3D: Edge Cases

In SAT's case, the Edge Cases are literally Edge Cases

- Edge-Edge Collisions
- Spheres and other curved surfaces

Edge-Edge Collisions

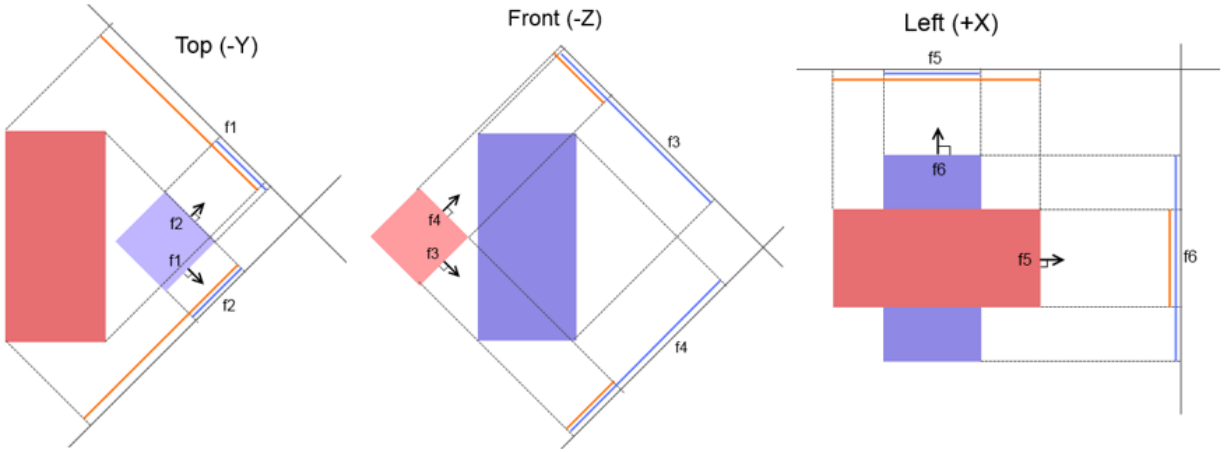
- Consider the scenario in three dimensions, shown below.



- Our SAT algorithm as explored in the last tutorial will give a false positive on this check. Here's why…

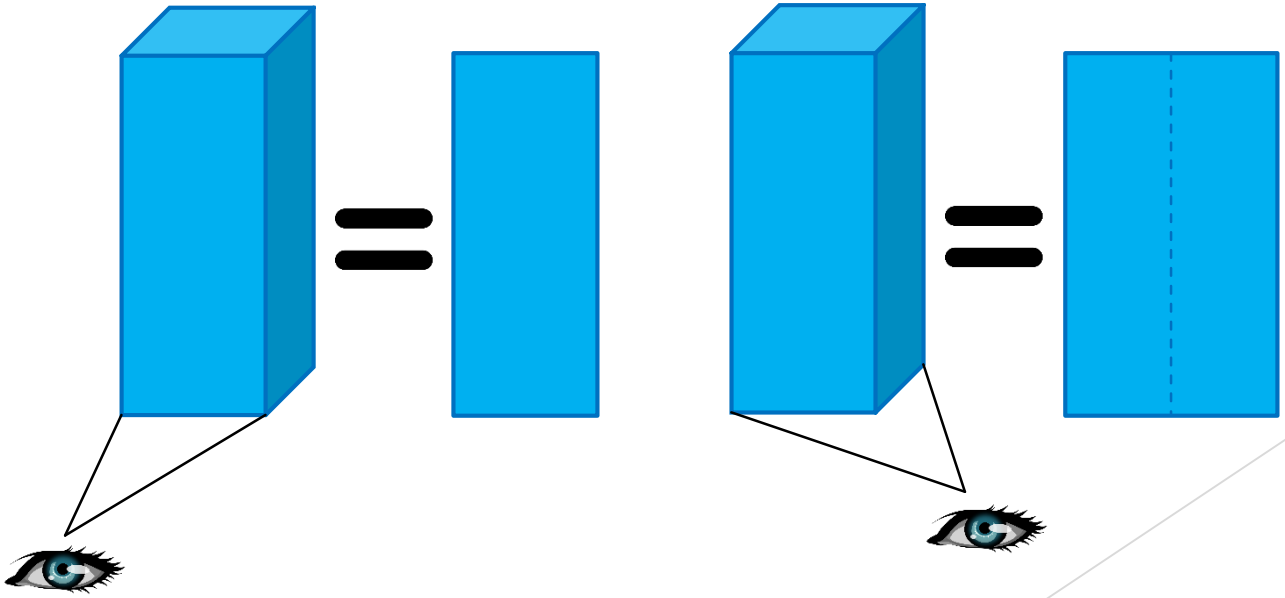
Edge-Edge Collisions

- Note only six checks, because parallel faces
- Looking through them, we can clearly see that all of our face normal give a positive result for a collision: SAT believes these objects have collided, and we can clearly see they haven't.



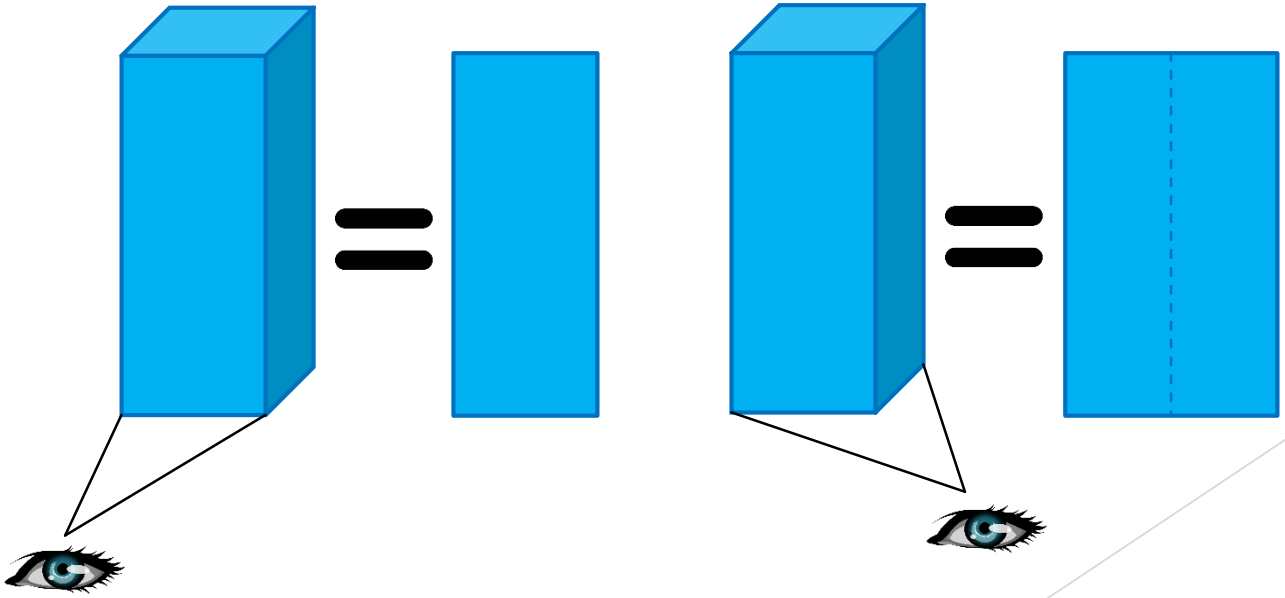
Edge-Edge Collisions

- This is due to the fact we're extending logic originally applied in two dimensions into three dimensions
- Looking at it another way, it happens because, if we view our cuboids as flat objects, they actually have two surface areas, e.g.



Edge-Edge Collisions

- Our approach only accounts for the left-hand case. We need to think a little abstractly in order to account for this problem



Edge-Edge Collisions

- The easiest way to extend our algorithm to account for the problem is to take every edge of both objects, and cross the permutations to produce additional axes
- You'll remember from graphics that the cross product of two non-parallel vectors results in a vector which is orthogonal (perpendicular) to both
- As such, considering Cartesian axes, $x \times y = z$

Edge-Edge Collisions

- Taking the cross-product of an edge on Object 1 and an edge on Object 2 will generate a new axis which is perpendicular to both of these edges, and thus define a new plane in which the two objects might not overlap

- You'll note in the framework that we've added a check which prevents permutations being considered more than once – that's because we've now got quite a few more planes to compare against – removing duplicates saves us processing time – if we didn't, the above example would give us 64 new planes to consider

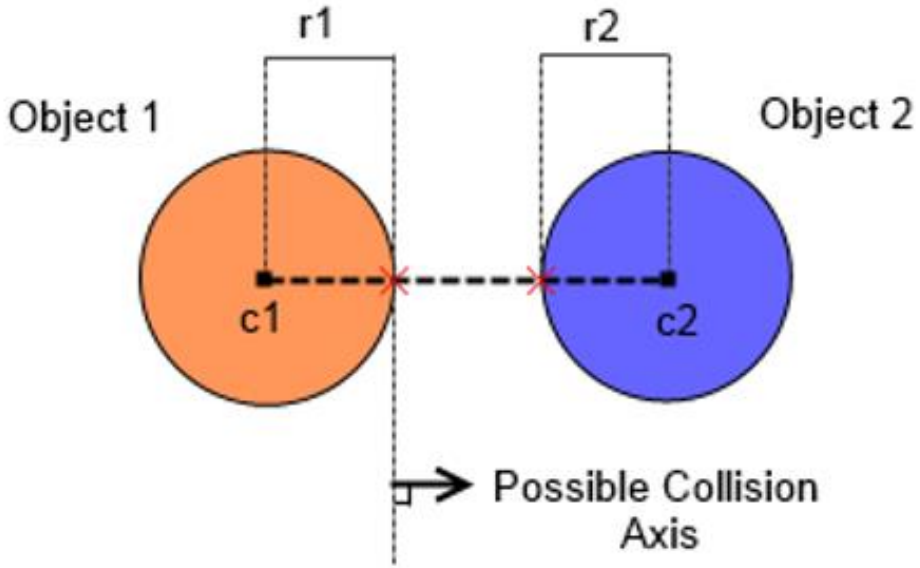
- This addresses our first edge case

Spheres and Curved Surfaces

- As discussed yesterday, spheres are the only object which guarantee a single point of contact with another convex hull
- That's fine, except spheres also have an infinite number of faces.
- Since our SAT-based system is based on face count, how are we meant to resolve this?

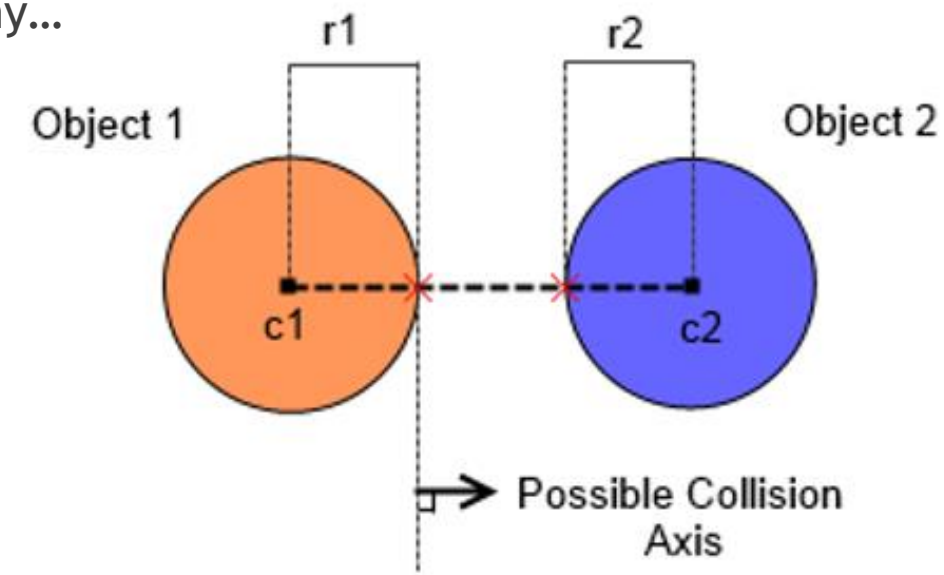
Spheres and Curved Surfaces

- Consider the scenario below



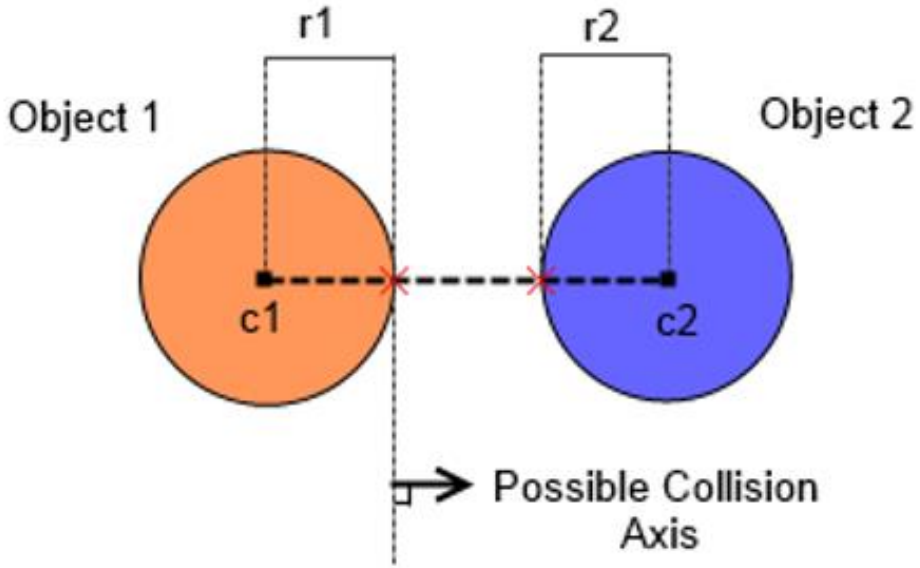
Spheres and Curved Surfaces

- In the sphere-sphere case, determining the ‘face’ we ought to be checking is actually fairly trivial – it’s the vector between the centres of the spheres – we can see why…



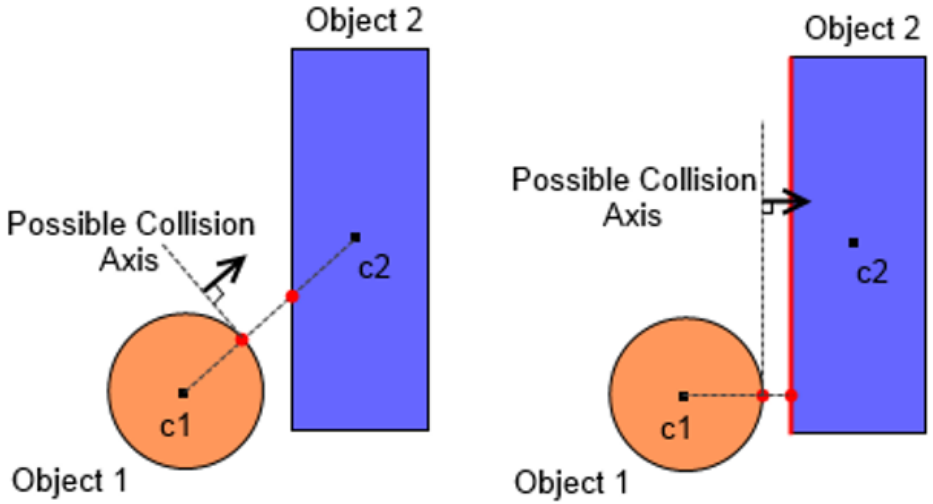
Spheres and Curved Surfaces

- This isn't always the case for sphere-polygon, however – let's have a look at an example of that situation.



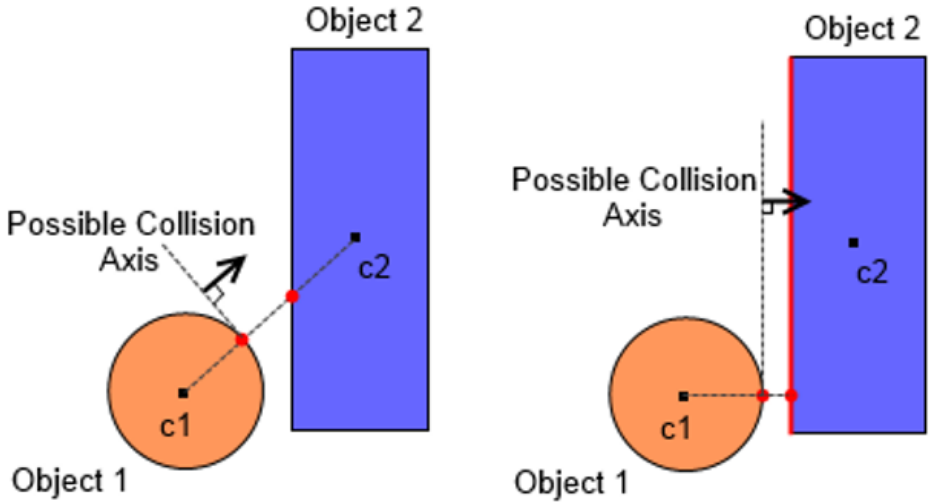
Spheres and Curved Surfaces

- Here, we need to determine the closest point of Object 2 to the sphere. We iterate through each face of the polygon, to deduce which point of the polygon is closest to the sphere based on the normal of that face



Spheres and Curved Surfaces

- The code to perform this check is provided for you to integrate into your framework



Summary

- Discussed Broadphase approaches and the differences between Broad and Narrow phases
- Discussed SAT in the context of interface detection
- Outlined the way SAT works
- Highlighted ways to optimise an SAT-based check
- Discussed collision detection algorithms in a general sense
- Highlighted the importance of algorithmic edge cases
- Addressed key edge cases which apply to SAT in three dimensions

Implementation

- Check the Tasks handout for today
- Have some fun with object representation