Collision Detection

GAME 300

Objectives

- Review 2D collision detection.
- Examine ways to implement 3D collision detection.
- Understand limitations and issues with collision detection.

Collision Basics

- Collision Detection is, as the name implies, the detection of two objects colliding with one another.
  - This typically requires some amount of force or movement applied to at least one of the objects.
  - Often measured in Velocity.

Collision information

- collision typically requires a series of variables to decide whether contact has been made.
- What type of Variables do you think might be needed?
  - Location of object
  - Bounds of object
  - Velocity of object
  - Location, bounds and velocity of second object
  - Time
  - Physical Materials… etc…

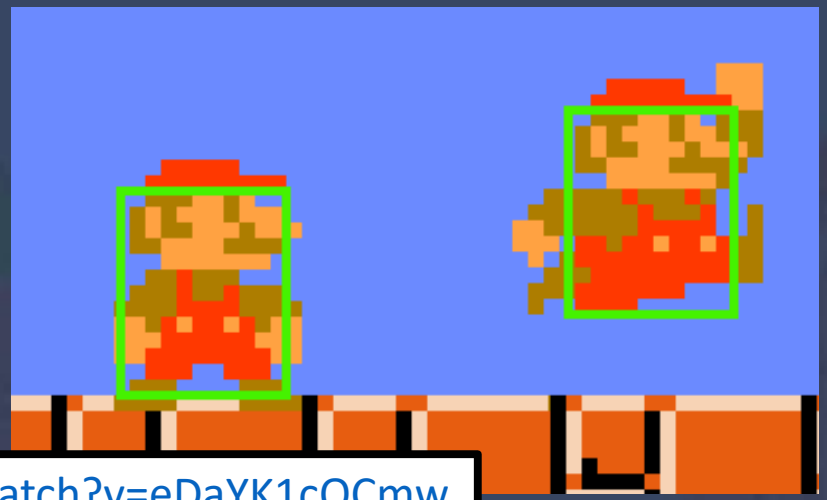
2D Collision: Positions

- Positional coordinates allow us to understand the current location of an object.
  - In terms of 2D this is often a screen coordinate in relation to the pixels of the screen.
    - X, Y
- Positions alone can’t be used to determine whether a collision has occurred.
  - (Unless you are testing for collision on a single pixel)

- GameObject2D:
  - Vec2(floats) position;

2D Collision: Bounds

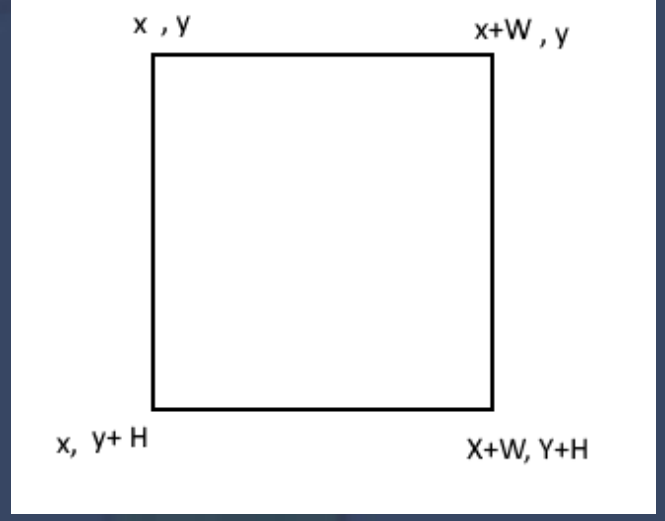
- Bounds of an object allow us to visualize and track the width and height of an object.
  - Not all pixels of a 2D image might require collision detection.
  - This will often result in user frustration if the game acts like a collision has occurred when visually it has not.
  - It's often better to require more contact within the object for collision detection than less.
- Bounds are added to an objects positional data to define the area where contact can occur.
  - This is often known as the bounding box.
- GameObject2D:
  - Vec2(floats) position;
  - Vec2 bounds; // width and height



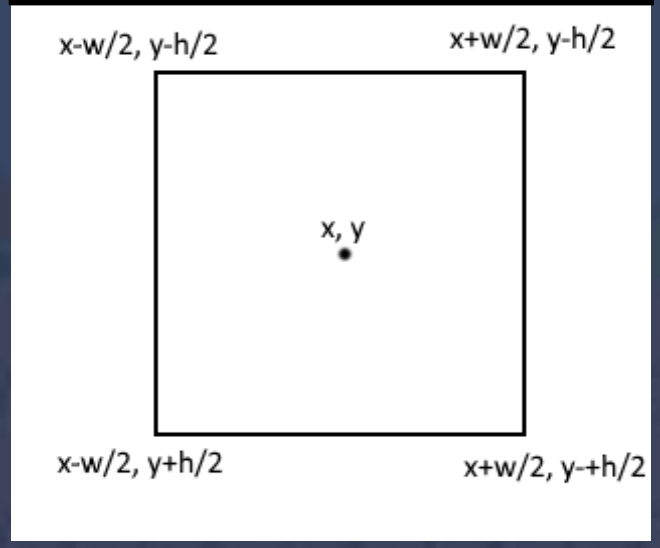
https://www.youtube.com/watch?v=eDaYK1cOCmw

2D Collision: bounding Box Calculations

Top Left Origin



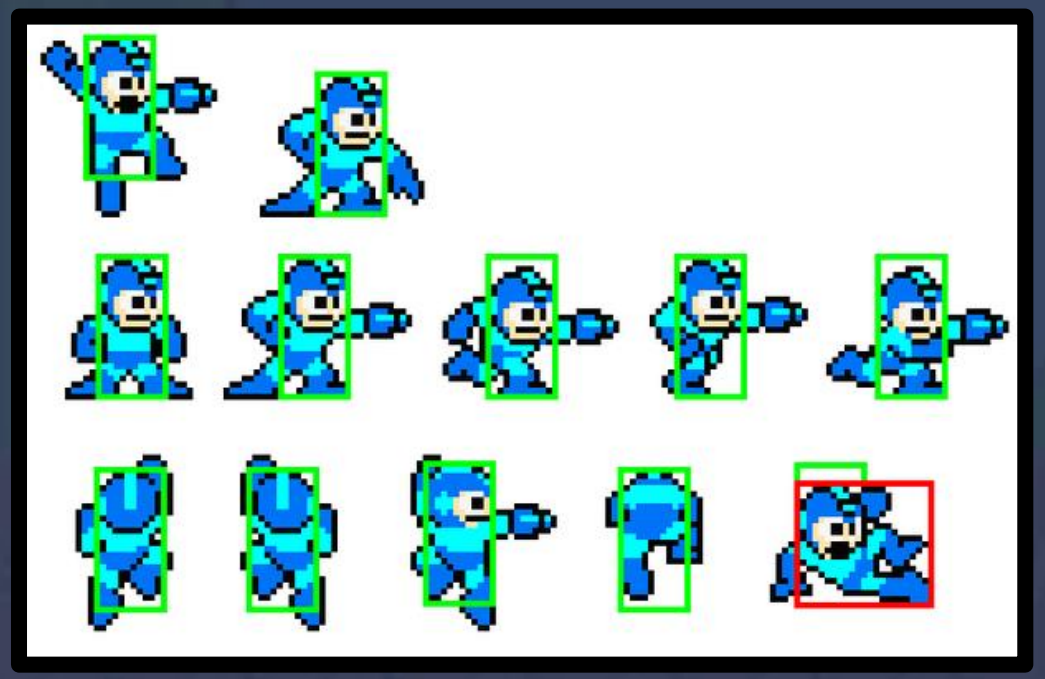
Center Origin



- GameObject2D:
  - Vec2(floats) position;
  - Vec2 bounds; // width and height
- To create the bounding box, we define the area by taking our starting position and adding in the bounding box.
- It's important to consider the point of origin.
- Sometimes a point of origin is at the center of the object.
- This changes the way we have to calculate our bounding box

Colliders

- Bounding boxes are a type of collider.
  - This allows us to detect our collision, but they don't always need to be defined as a rectangular shape.
  - A radius could be used surrounding a central origin point to determine a circular collider.



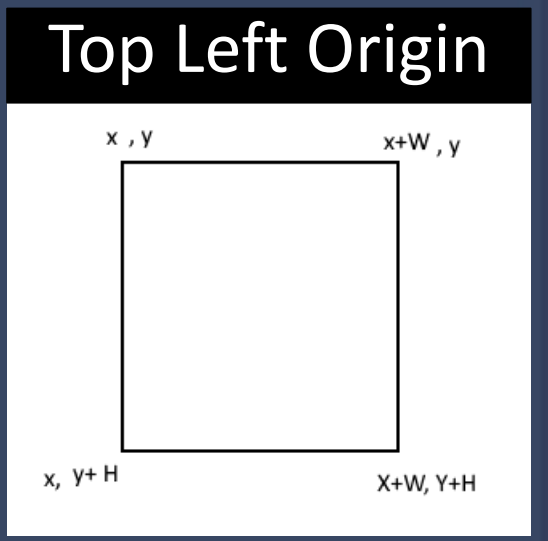
- For more precision, multiple hitboxes or colliders can be associated with a single game object.
- Also note that for 2D hit detection of sprites, the animation may alter the dimensions of the character and the dimensions of your object may need to adjust based on this at specific frames.

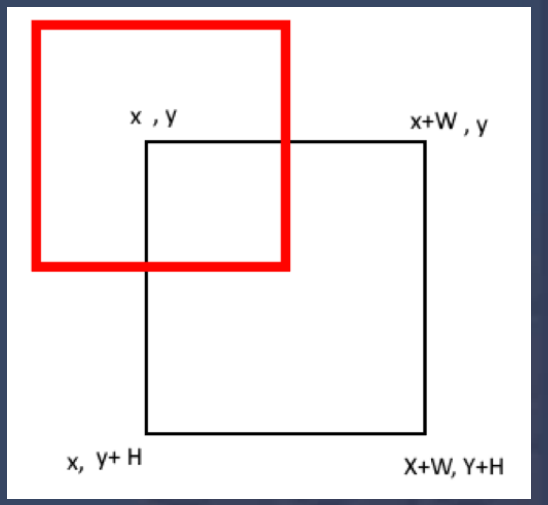
2D Collision: bounding Box Calculations

- Using the example shown, we could detect if an single position is within the area by checking all four corners.
  - This would require 4 separate if statements condition checks:

```
if( XtoCheck > X && YtoCheck > Y // we are after the origin,
&&  XtoCheck < X+W && YToCheck < Y+H) // we are before the end
```

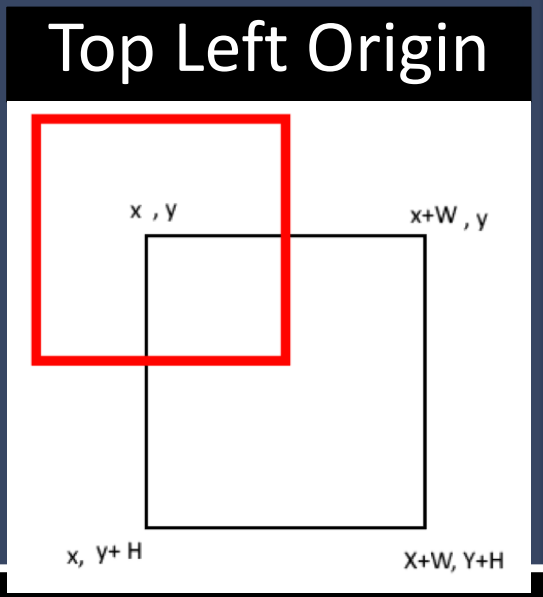
  - This works for a single point… however our collisions are often not performed using a single point.
  - We must also factor in the width and height of the object coming in contact.





2D Collision: bounding Box Calculations

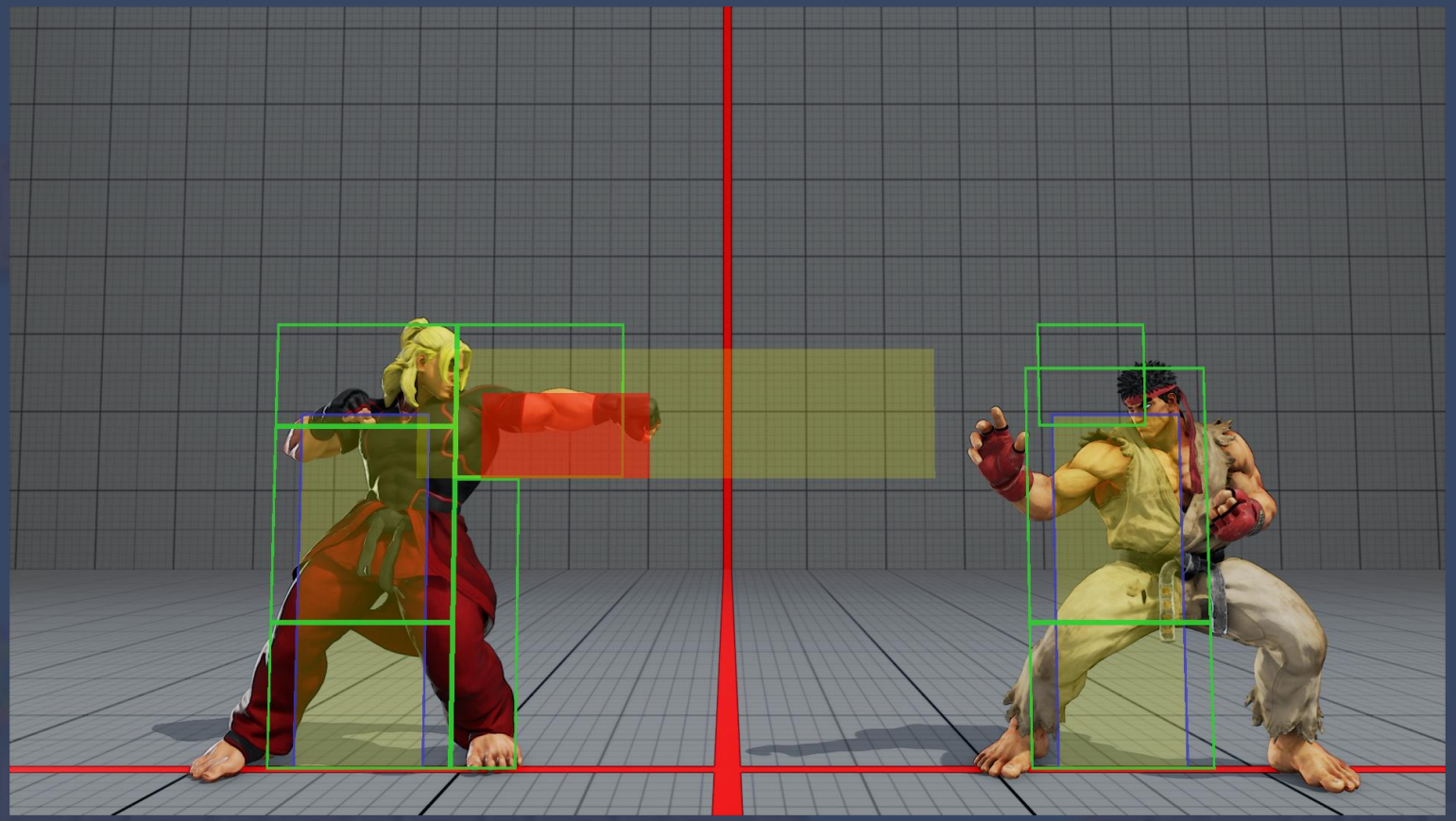
- The following check below now tests all four corners within the complete area of the other bounding box:




```
if( (XtoCheck > X && YtoCheck > Y &&  XtoCheck < X+W && YToCheck < Y+H) ||
(XtoCheck > X && YtoCheck+HtoCheck > Y &&  XtoCheck < X+W && YToCheck +HtoCheck < Y+H) ||
(XtoCheck + WtoCheck > X && YtoCheck > Y &&  XtoCheck + WtoCheck < X+W && YToCheck < Y+H) ||
(XtoCheck + WtoCheck > X && YtoCheck+ HtoCheck > Y &&  XtoCheck + WtoCheck < X+W && YToCheck+ HtoCheck < Y+H) )
{
        return true;
}
```

Hitbox vs Hurtbox

- Sometimes an object may have more than one collider for different purposes.
- A hitbox is typically a collider box used to determine damage (shown here in red).
- A hurtbox is a collider box where the character will receive damage.



Cost of Collision Detection

- Collision detection is expensive.
- We often are needing to check multiple collider boxes per object against many other collider boxes.
- Optimizing the way we perform our calculations and checks as much as possible is essential to maintaining a proper frame rate for your game.
- Many games do not perform collision detection every frame, instead they check collision intermittently.
  - The frequency is often a tweakable variable inside of game engines.

2D Collision: Calculations Optimized

```
if( (XtoCheck > X && YtoCheck > Y &&  XtoCheck < X+W && YToCheck < Y+H) ||
(XtoCheck > X && YtoCheck+HtoCheck > Y &&  XtoCheck < X+W && YToCheck +HtoCheck < Y+H) ||
(XtoCheck + WtoCheck > X && YtoCheck > Y &&  XtoCheck + WtoCheck < X+W && YToCheck < Y+H) ||
(XtoCheck + WtoCheck > X && YtoCheck+ HtoCheck > Y &&  XtoCheck + WtoCheck < X+W && YToCheck+ HtoCheck < Y+H) )
{
        return true;
}
```

- Instead of checking for area containment, check for exterior:

```
if( XtoCheck + WtoCheck < X ||  // far left of box a compared to far right of box b
        YtoCheck + HtoCheck < Y || // top of box a compared to bottom of box b
        XtoCheck > X+W || // far right of box a compared to far left of box b
        YtoCheck > Y+H ) // bottom of box a compared to top of box b
{
        return false;
}
```

3D Collision Detection

- When we perform 3D collision detection we are adding in 2 more values to check for:
  - Z position and Depth / Length.
  - The following assumes the 0 point is one corner of the object:

```
if( XtoCheck + WtoCheck < X ||  // far left of box a compared to far right of box b
      YtoCheck + HtoCheck < Y || // top of box a compared to bottom of box b
      YtoCheck + LtoCheck < Z  || // back of box a compared to front of box b
      XtoCheck > X+W || // far right of box a compared to far left of box b
      YtoCheck > Y+H  || // bottom of box a compared to top of box b
      ZtoCheck > Z+L  || // front of box a compared to back of box b
{
      return false;
}
```

  - NOTE: In the event that the object has a center origin, you would need to subtract and add the width and height /2.
    - Unless the height and width represent a half dimension.

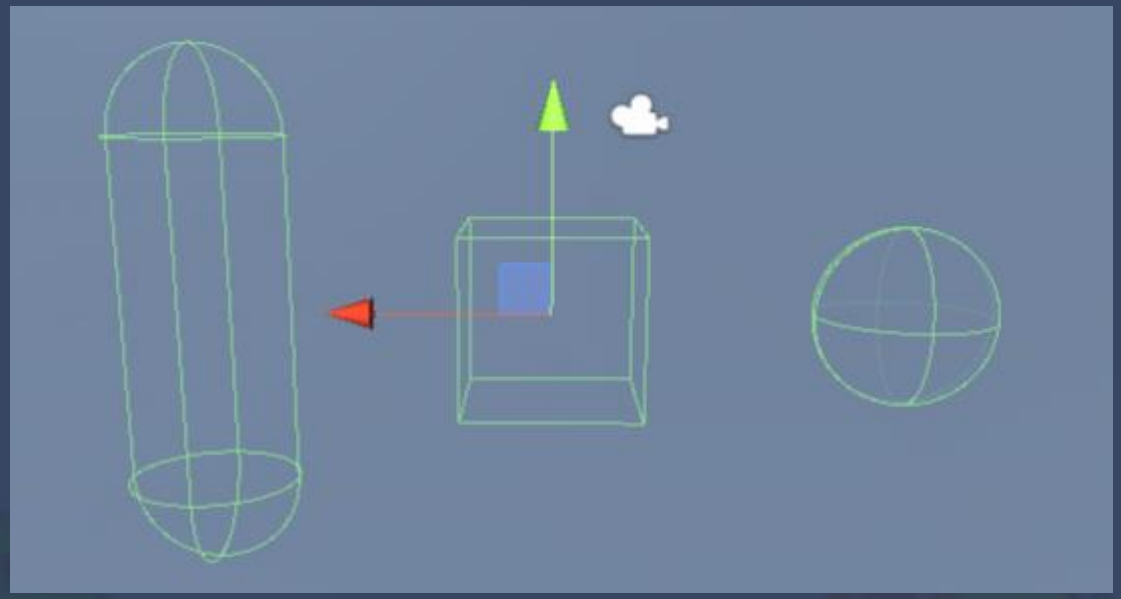
3D Collision Detection

- Assuming an equal dimension we can calculate if our object is out of range of the collider using the following style of hit detection:

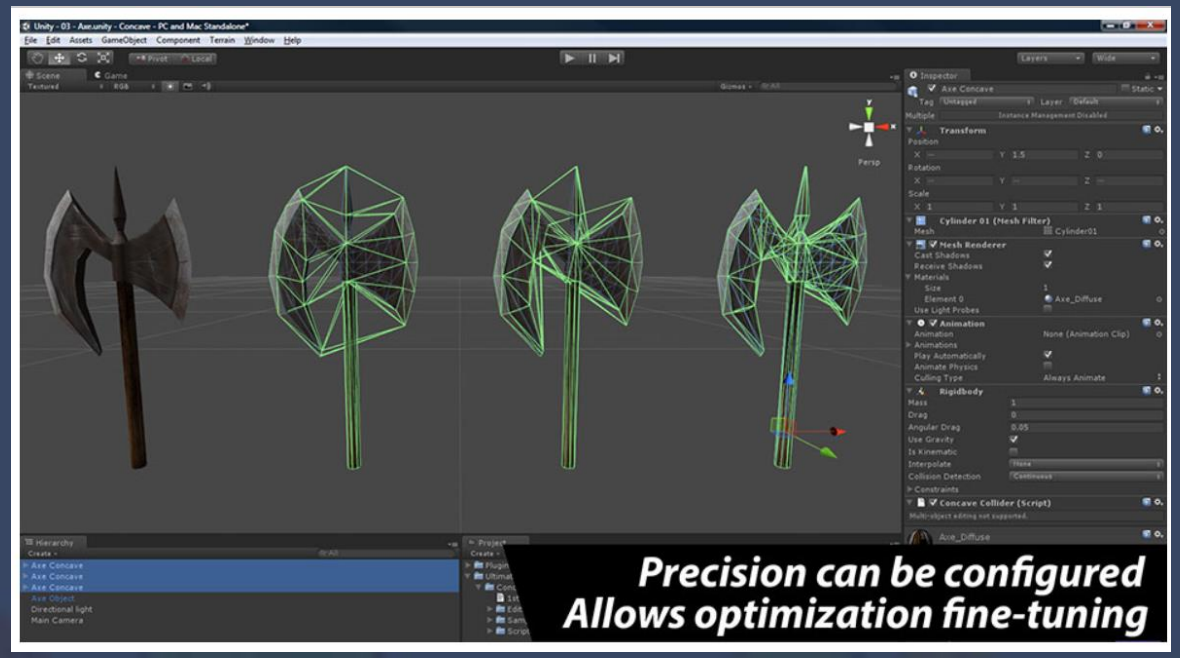
```
if (Position.x - dimensions.x < Position2.x - Dimensions2.x ||
    Position.x + dimensions.x > Position2.x + Dimensions2.x ||
    Position.y - dimensions.y < Position2.y - Dimensions2.y ||
    Position.y + dimensions.y > Position2.y + Dimensions2.y ||
    Position.z - dimensions.z < Position2.z - Dimensions2.z ||
    Position.z + dimensions.z > Position2.z + Dimensions2.z)
{
    return false;
}
```

Game Engine Colliders

- Game Engines typically come with Collider objects available.
- Most common options are:
  - Box, Sphere, capsule, and mesh colliders.
  - **Box** colliders calculate using the area of the box.
  - **Spheres** use radius in 3 dimensions to determine the area to check.
  - **Capsules** use two spheres extended and all areas in between to determine area to check.
  - **Mesh** Colliders check the area within a mesh.



Mesh Colliders



http://www.ultimategametools.com/products/concave_collider

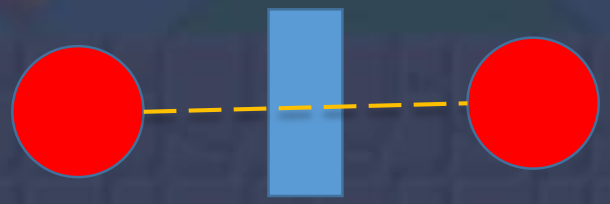
- Mesh colliders are an extreme precision case and should be used sparingly.
  - These are the most complex colliders which makes them the most costly.
  - Often times to save on this cost, we will create a secondary model with reduced vertices to apply collision.
    - This is sometimes handled by the engine itself.
    - The less vertices the less calculations.
  - Often times they are converted to convex outlines of all the mesh instead of having concave points within the object.
    - Concave points will cause spikes of new calculations and possibilities for bugs.

BUGS

- When we intermittently check the location of objects for collision, we run into a problem.
  - A common bug for collision detection is for items to completely pass through an object, or get stuck within an object.
  - This is typically because the object or character moving passed by the object at a velocity so quickly it was never detected within the area of the object.

History

- To resolve the pass through bug, we have a couple of options:
  - Limit the speed of our objects so that they cannot pass through before the next collision calculation.
  - Increase the frequency of checks for collision.
  - Keep a HISTORY of the locations of the object.
    - For this solution typically only the previous position is required.
    - You can draw an imaginary line between the previous point and the new point.
      - If the line were to intersect through the collision box then we know that a collision occurred between frames and handle it appropriately.



Reactions

- Once we've detected a collision, we need to think about what should occur.
  - Destroy the object
  - Particle Effects
  - Increase speed
  - Physics calculations:
    - Force feedbackImpact
    - Reflection of velocity
    - What the physical metarials are of each object
    - Any other physics calculations that need to be applied
  - Physics calculations applied after impact can be yet another variable for bugs. If we perform the calculations in succession too quickly our objects could get stuck together repeatedly triggering the collision processing.
    - It is often customary to leave a buffer window after collision is detected where the objects are not processing collision detection on each other.

Summary

- Collision detection is important when dealing with 3D objects within a virtual world.
- Incorrect collider placement or application can result in catastrophic gameplay bugs for your players.
- Cost of collision detection can be high so be careful on how you process it and what object you apply it to.