Firstly… Let's add a Spawner!

- We didn't quite get round to this last week due to an early finish….
- I've got some sample code for a very basic spawner here:

  github.com/JamesAdey/zompy/blob/week3/zombie_spawner.py

- You should first copy the code into a new file called *`zombie_spawner.py`*
- Then edit your *`zompy_engine.py`* file and add code to create the spawner.

Creative Computing. Week 3

(cool stuff to do on a computer)

If you don't have the project yet...

- I've hosted the basic code on my GitHub

- Go to: ***github.com/jamesadey/zompy/tree/week3***
  - *This is the project containing finished code from week 2, ready for week 3.*
- Download all the files
  - *There should be a button to **download/clone***
- Open zompy_launcher.py in IDLE
  - Python version 3.x please!
- Run this file, and the game should start…

How far have we got...

- Our zombies can move...
- And they can be shot...
- And we have spawners!
- What next...

- Items and collision detection!

Collision Detection.

(woo! more theory!)

What is Collision Detection?

- Simply put, collision detection is the process of determining if two objects would overlap each other.
- What happens next depends on the context it's being used in.
  - Collision detection is most commonly used to stop objects intersecting.
  - The same system can be used as a trigger, checking if an object has entered a certain area but without affecting it's motion.
- However… The process of checking if 2 objects intersect is often expensive for complex shapes, so simple primitive shapes are used to approximate the complex areas.
- Primitive shapes are simple shapes that have nice properties, so can be used in very fast and optimised algorithms for intersection checking.
  - Examples include: Circles, rectangles, cubes, spheres and capsules

Our system is simple.

- Firstly, all our objects are circles.
  - This makes the maths considerably easier.
- We're not going to implement full collisions that block movement.
  - This is a complex problem and is beyond the scope of this project.
- Instead we're going to use these circles as triggers, performing actions when an overlap occurs.
- *(Box collision is similarly quick but I'm not covering that in this project yet)*

What will our engine do?

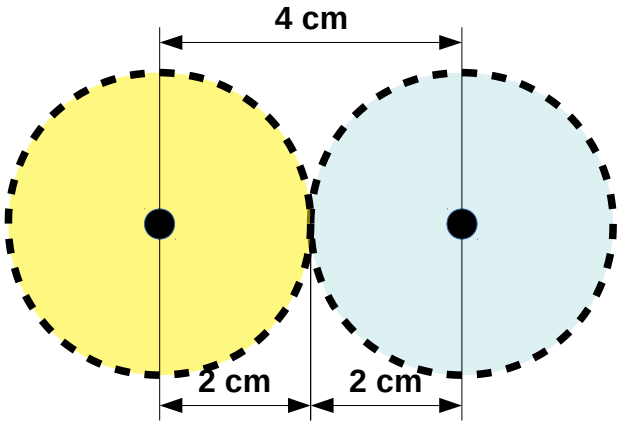
- Maintain a list of all the items that exist in the world.
- Check collisions every frame.
- Every frame:
  - For each item in the game, check if it overlaps with the player:
    - If it does, then notify the player and the item that they have been involved in a collision

(This system could be extended to check against any number of objects, not just our single player)

Circle-circle collisions.

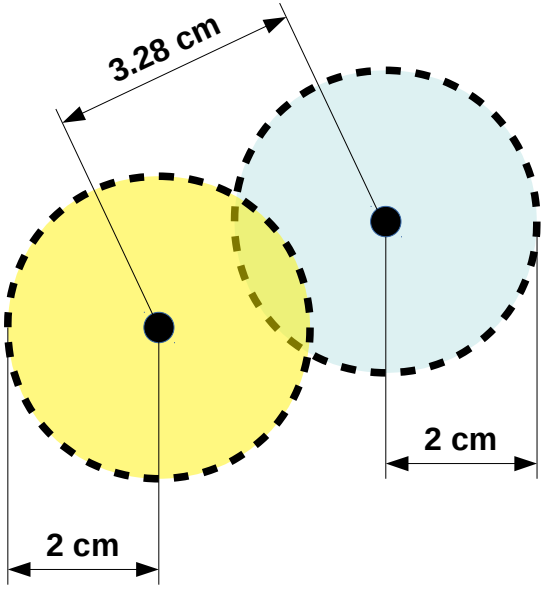
- Two circles collide if the distance between their centres is less than the sum of their radii.

Not colliding



Maybe colliding?
(depends on implementation)

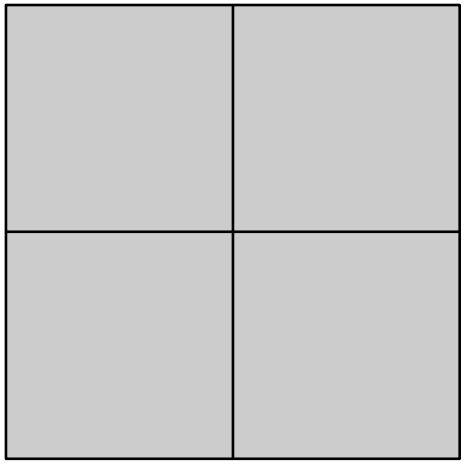
Definitely colliding



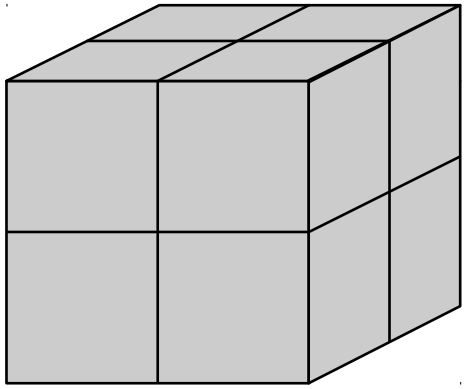
How larger game engines work

- Unfortunately, a naiive solution doesn't scale to having lots of objects colliding.
- This is because there are just too many objects to test each one against another
  - With 20 objects and 10 items that could intersect, we're already at 200 calculations per frame.
- Furthermore, a lot of time is wasted checking objects that were never going to collide in the first place. Both of these issues can be overcome with a technique known as spatial partitioning.
- Spatial Partitioning is the process of splitting the game space into smaller areas. Then objects are only checked against other objects in nearby areas. Instead of checking across all, objects in the world.

Spatial partitioning with Quadtrees & Octrees



- Quadtrees is often used in 2D space. Splitting squares into 4 smaller squares.



- Octrees are often used in 3D space. Splitting cubes into 8 smaller cubes.
- Each sub-area is then split again and again until a small enough area is covered.

The Fun Part.

(actually coding stuff)

Starting with an example!

- I've provided you with an item manager and an example item pickup to get you started.

  github.com/JamesAdey/zompy/blob/week3/item_manager.py

  github.com/JamesAdey/zompy/blob/week3/example_item.py

- To integrate these with the game:
  - We're going to need to add an *on_collision* function into `player.py`
    - *(This should be the same as the one in* `example_item.py`*)*
  - We're also going to need to create the item manager and items at the start of the game. We can do this just as we have done with the other objects in `zompy_engine.py`

Creating an ammo box!

- The sample item doesn't do anything… Time to create one that does.

- We could do with an ammunition system in the game, currently our player can shoot infinitely. So…
- 1) Create a variable in our player for our current ammo.
- 2) Then modify the shooting code to only fire if we have ammo left!
- 3) Then create a function in the player to add ammo.

  (We can call this from the ammo box item to add ammo to our player)
- 4) Create a new class for the ammo box using `example_item.py` as a template.
- 5) Finally, make it so that when the ammo box collides with the player, it adds ammo.

Extensions...

- Item spawners!
  - Try combining what you've learned from zombie spawners with the items from today, and use it to create items randomly across the map.
- Player upgrades!
  - Try creating another item that upgrades our player when it is picked up.
    - Could it increase their speed?
    - Could it increase their health?

Finally… A roadmap.

- This is currently week 3 of the club, how are you finding it so far?
- Next week is half term…

Finally… A roadmap.

- This is currently week 3 of the club, how are you finding it so far?
- Next week is half term…
- We have the basics of a game so far, and at least 5 weeks left… So here is what else I want to cover:
  - Pathfinding
  - Levels/Maps
  - Scoring
  - Special Zombies