Case Study #4 - Exercises

Exercise #1

Assumptions

1. There are a few easy ways that we could do that, but I would like to explore a complete solution without using Collision Detection initially.
2. Assume the following:
    a. We have the direction vector for the object with **`objectDirection.x`** and **`objectDirection.y`** being the x and y elements of the vector.
    b. We have the bounding rectangle for our object as **`rect`**.
    c. We know the size of the screen as **`screenSize`**..
    d. The speed of the object is given by the **`objectSpeed`** variable.
    e. The position of the object is given by the vector **`objectPosition`**.
3. The other information that we need is as follows:
    a. When there is a collision with a boundary of the screen we need to know which edge has been hit so that we can figure out the direction we need to change.
    b. In our solution we only care about whether:
        i. The edge is top/bottom in which case we reverse the y direction
        ii. The edge is left/right in which case we reverse the x direction
4. There are two approaches we can take here:
    a. We can use collision detection but when the collision happens we do not know immediately which “wall” was hit.
    b. We can do our own calculation by simply testing the bounding rectangle of the ball if it has hit any of the boundaries.

5. **NOTE**: it is important to understand that when it comes to Collision Detection in Flame Engine, it is not perfect and you will need to add some extra code to ensure that objects do not slip past your detection in more extreme situations.
    a. If you have a lot of small objects and/or the objects are traveling fast, then there is always a possibility that the collision detection might be too slow to detect the actual collision. For example in our case if you spawn 100 objects and let them just bounce around the screen there is a possibility that an object might slip past detection and fly out of the screen boundary..
    b. For example, to detect if an object has left the screen boundary all you need to do is just test to see if it is outside of the boundary. You do not need to have special collision detection for that and it will be 100% accurate but might be a bit slow since you might detect it some time after that object has left the boundary.
    c. In the case of bouncing an object from the wall that is a slightly different use case. If the Collision Detection does not properly detect the collision then your ball will not bounce. If the collision is then detected later (with your extra code) it

will seem that the ball has disappeared and then magically bounced back. But this is better than losing the ball completely.

i. To solve this issue you would need to set some sort of threshold and then detect that the ball has gone past that boundary by testing for the ball's direction in relation to the boundary (i.e. is it moving away from the boundary or towards it?)
ii. So to do this you will need to calculate if the ball is moving away from the screen or towards the screen.

Solution:

1. We can do the following simple test:

```
if ((rect.left <= 0 && objectDirection.x == -1) ||
        (rect.right >= screenSize.x && objectDirection.x == 1)) {
      objectDirection.x = objectDirection.x * -1;
}
```

2. The above both checks the collision detection with the screen boundaries as well as reverses the direction for the x-axis. For the y-direction the idea is the same:

```
if ((rect.top <= 0 && objectDirection.y == -1) ||
        (rect.bottom >= screenSize.y && objectDirection.y == 1)) {
      objectDirection.y = objectDirection.y * -1;
}
```

The above solution will bounce the ball object back at about a 90° angle. The reason why we test the condition against the bounding rectangle is that the bounding rectangle's coordinate data is not relative to the anchor property which can complicate calculations. In this case the bounding rectangle is related to the screen's origin of (0, 0) so the comparison is much simpler.

Also, the reason why we do not use **`onCollision`** notification for the screen edge detection, is because we still need to know which edge we collided with so we might just as well test the whole collision with our own code to quickly determine which part of the screen was hit (i.e. top, bottom, left, or right) which then determines how we change the direction vector.
Consider this to be a follow up exercise for you to try to rewrite this code using actual collision detection from flame:

HINT: use the **`intersectionPoints`** array to determine which edge of the screen the ball has collided with.

You will find the code solution in the directory *"Exercise Solutions\component_004[Collision_Detection]\component_004_001"*

Exercise #2

Assumptions:

1. We are trying to bounce two colliding balls in the opposite direction. We can simply assume at this point that we will just reverse the direction of each ball for simplicity. In real-life you would want to calculate the angles after collision that would be different for each ball, but this exercise deals a bit more with understanding collision detection notification in Flame and it is less about calculating exact angles. So the actual reversal will be done in the same manner as we did in Exercise #1:

```
/// bounce away from the object we collided with
xDirection = xDirection * -1;
yDirection = yDirection * -1;
```

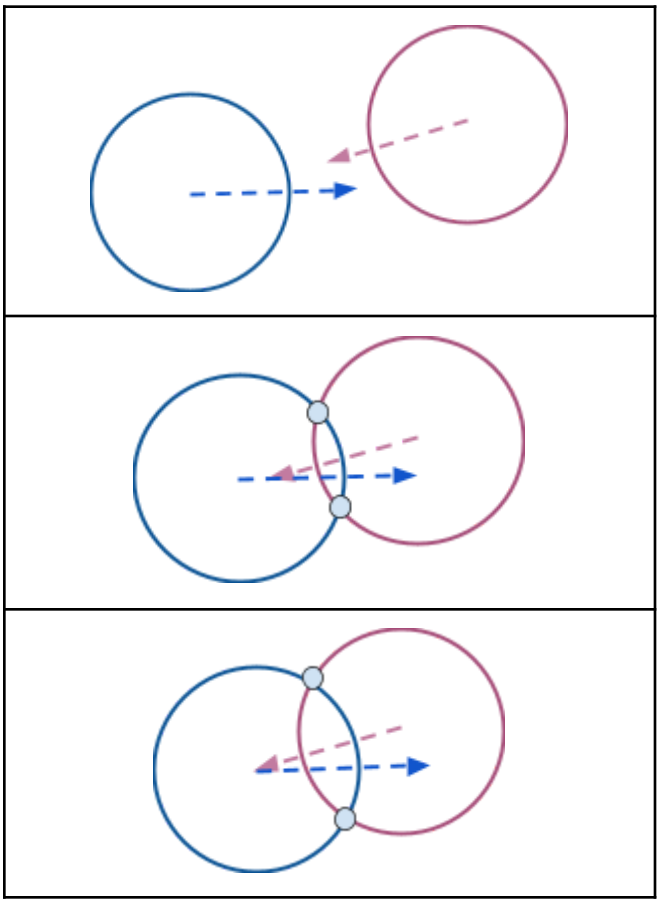
2. To have the balls bounce off each other is harder than dealing with bouncing off the screen edge (i.e. the wall) since even with just 10 balls you could have multiple interactions, so computational complexity starts going up with the number of collisions you will need to track (there are 10 x 9 which is 90 different interactions possible.) and unfortunately it is not as easy to simply just to reverse direction when the collision happens.

3. One interesting problem is that when objects collide, they could (and usually do) have an overlap (which is given by the Set of intersection points in the onCollision method). If that happens and we would like to bounce the objects off each other, we could run into a deadlock problem as follows:

    a. This can lead to a type of a deadlock where the colliding objects could stick together, since as we try to bounce the objects away from each other by simply reversing direction, they could still trigger another collision detection (before they can get away from each other) which would actually make them move back in the original direction thus triggering more collisions and thus we would have something of a deadlock loop.

    b. This comes from the simple fact that a single logical collision is actually a series of collision events, so we cannot just react to all these collisions without context. We need to think of the collision as a collision-start and then at some point a collision-end.

i. For example if we have two balls colliding:



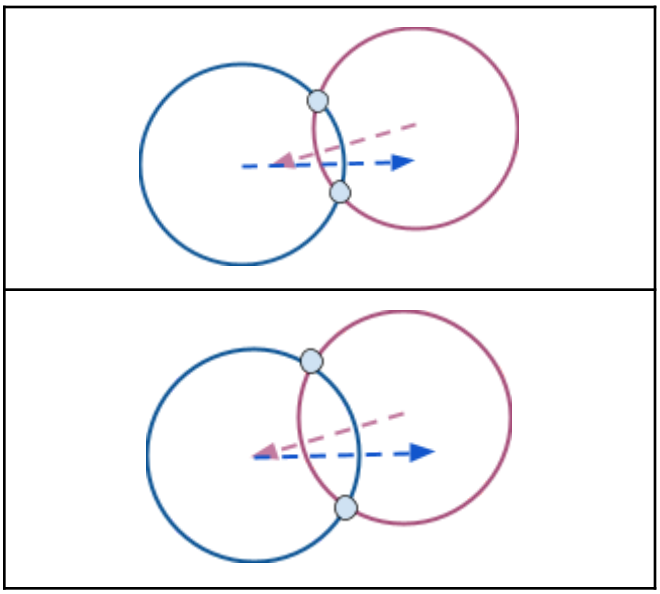
ii. In the case of images 2 and 3 we are seeing the possibility of triggering multiple calls to onCollision method as the objects are moving closer to each other, for what we would like to treat as a single collision event!

iii. So if we want to ensure that only the first such collision detection (i.e. like startt-collision) is taken into consideration we need to think of this as three distinct events:

1. **collision-start:** this is the first collision detection for those two objects and this is when we would bounce the objects.
2. **collision-in-progress**: we would ignore this since we have already bounced the ball.
3. **collision-end:** this is when we are done with the collision and any new collision with the same object would be considered as a brand new bounce.

Solution:

1. One simple solution would be to lock the event and ignore other collisions between the same objects until this one is handled. But how do we do that?
    a. One way to do that is to test if objects are moving away from each other or moving towards each other.
        i. If they are moving towards each other then they have collided and we want to bounce them back.
        ii. If they are moving away from each other then they have bounced back… which means that we can ignore their collisions until they are moving towards each other again.
    b. Another way is to somehow figure out when the actual collision has ended for those two objects. In other words we can think of a collision between two objects as a succession of collision detection events. For example in this situation we have single collision with 2 collision events and once these objects pass through each other we have an end-of-collision:



So how would we code that? We can think of it as follows:

i. When the two objects collide they are at a certain distance from each other.
ii. When we bounce them away (by changing the direction of each ball) they will start moving away from each other, but before this happens we might get a few more events of them moving closer together. In other words since these events are queued we need to wait for the bounce effect to happen while we ignore the rest of the collision events for that collision.
iii. So what we can do is this simple algorithm:
    1. At the start of the collision add the id of the other object that this object collided with to a set of ids of other objects that this object has collided with.
    2. When another collision event happens, we check if we have an ongoing collision with this other object (by looking it up in the set)

    a. If it exists we do not handle this collision since we have already handled it.
    b. Otherwise we bounce the ball and add the id of the other object to the set.
2. At each update() tick for this object we first check if there are any ids of other objects that we have collided with and we test if those objects have moved outside of the boundary of the collision, if YES then we simply remove that other object's id from the set. Here are the code snippets:

The definition of our set for each ball object.

```
  /// The map of ids of the objects we have recently collided with
  Map<String, MyCollidable> collisions = Map<String, MyCollidable>();
```

This is in the onCollision(...) method

```
/// Add the id of the object we have collided with to our set of
/// currently happening collisions.
if (collisions.containsKey(other.hashCode.toString())) {
  /// do nothing, we have handled this collision already
} else {
  collisions[other.hashCode.toString()] = other;
  /// bounce away of the object we collided with
  xDirection = xDirection * -1;
  yDirection = yDirection * -1;
}
```

```
/// check for any unresolved collisions
///
List keys = [];
for (var other in collisions.entries) {
  MyCollidable otherObject = other.value;
  if (distance(otherObject) > size.x) {
    keys.add(other.key);
    print('removing other: ${otherObject.hashCode}');
  }
 }
```

You will find the code solution in the directory *"Exercise Solutions\component_004[Collision_Detection]\component_004_002"*