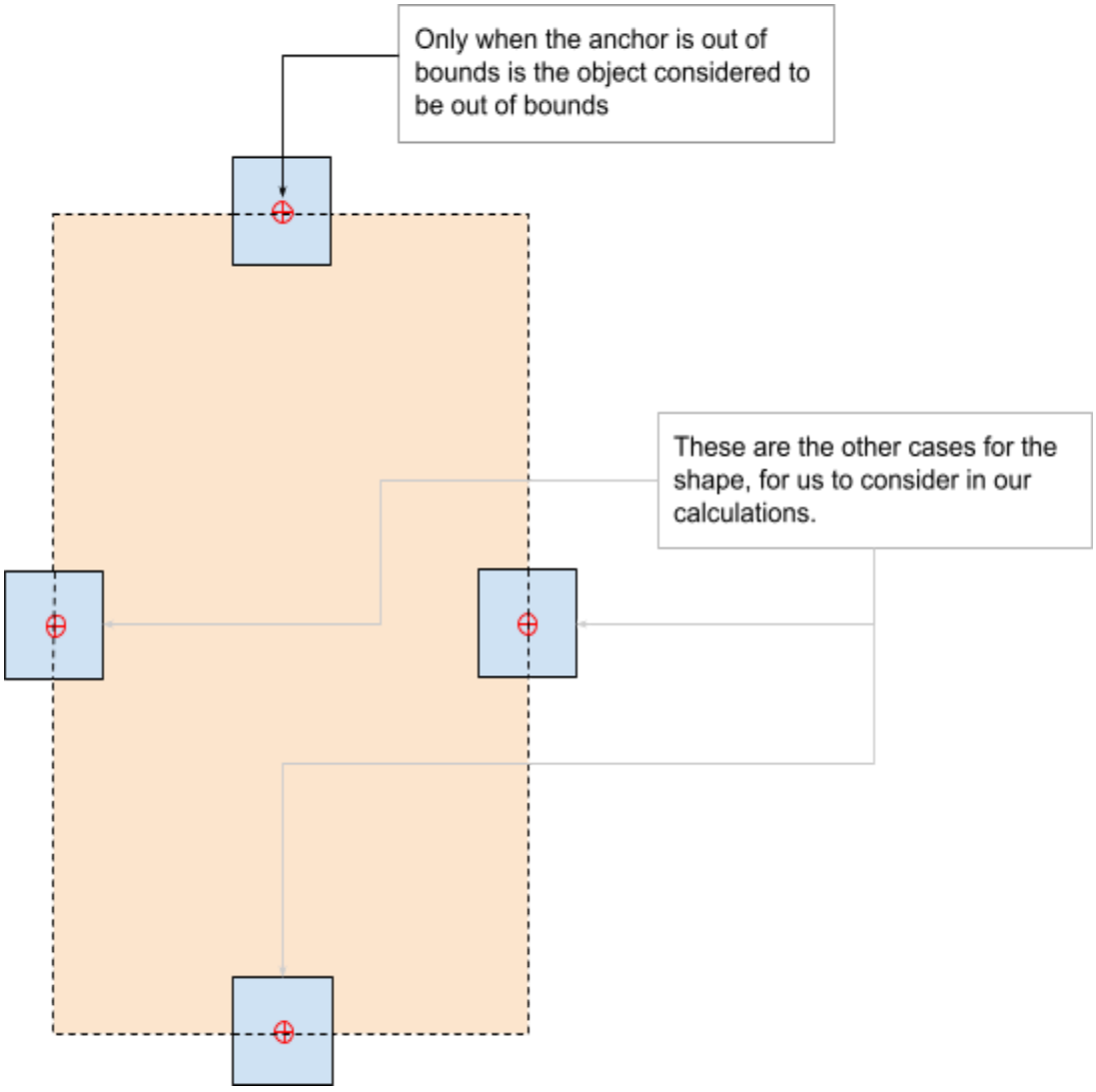
Case Study #2 - Exercise Solutions

Exercise #1

Assumptions:

1. We can assume right now that the `position` member of our Bullet is good enough to determine the “collision” between the shape and the screen boundary. You could think about how the anchor property affects the boundary condition. Currently (by default) the anchor is in the center of the shape, which means that the Bullet moves about half-way past the screen boundary before our code would detect it as being out of bounds if we are strictly comparing the object’s position value.



Solution:

1. To remove a shape when it has crossed the screen boundary we do as follows:
    a. To detect that the shape has crossed the boundary we check its position and compare it to the min and max coordinates of the screen:

```
if(shape.position.x > size.x
      || shape.position.x < 0
      || shape.position.y < 0
      || shape.position.y > size.y)  {
            // then we are out of bounds and should
            // remove the shape.
      }
```

You will find the code solution in the directory
"Exercise Solutions\component_002[Joystick]\component_002_002"

2. There is a better solution. But it would require that you compensate for the anchor's position in the object. The simple solution is that you work with the object's size and consider the 4 main possibilities for collisions with the boundaries (**note**: assumption here is for anchor to be in the middle of the object - math will be different if the anchor is in a different configuration):
    a. Upper edge collision: here we need to compensate for the height/2 since half of the shape will already be out of bounds before we detect it. So here we use the test as follows:

```
if(shape.position.y < shape.size.y/2)
```

    b. Lower edge collision: here we need to compensate for the height/2 as well since half of the shape will already be out of bounds before we detect it. So here we use the test as follows:

```
if(shape.position.y > size.y - shape.size.y/2)
```

    c. Left edge collision: similar idea except now we work with eth width. The test will be as follows:

```
if(shape.position.x < shape.size.x/2)
```

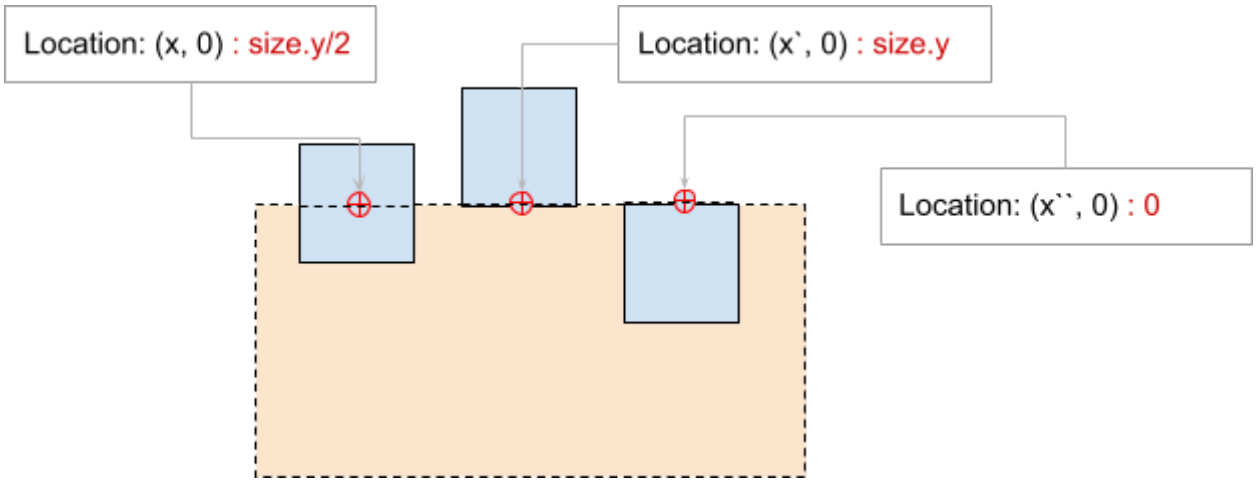
    d. Right edge collision: same concept as in height calculation:

```
if(shape.position.x > size.x - shape.size.x/2)
```

In fact, to fully compensate for a rotating object will be more difficult since you would have to compensate not only for the anchor, but also for the anchor's rotated location to truly create an accurate bounds detection and then proper wrap-around. This is beyond this course but I do encourage you to think about it and create a library method for such a functionality.

Consider a few hints:

1. Below, you see just a few of the possible y-locations for a given object with relation to the boundary at the top. Note that the anchor location changes the actual relationship of the object's position to the boundary.
    a. For example the middle square (x`, 0) will not really trigger any boundary detection until that anchor (which is at the bottom of the object) hits the 0-location. This means that we will be a whole size.y too late in detecting the object crossing the boundary.
    b. So you would need to compensate for each of those possibilities as shown in red font.



2. It becomes even more complex when the object is rotated since its anchor will rotate with it thus completely changing the compensation needed
    a. For example the middle square (x`, 0) if you rotated it by 180° it would actually become the square shape on the right so you would not need to compensate in that situation.

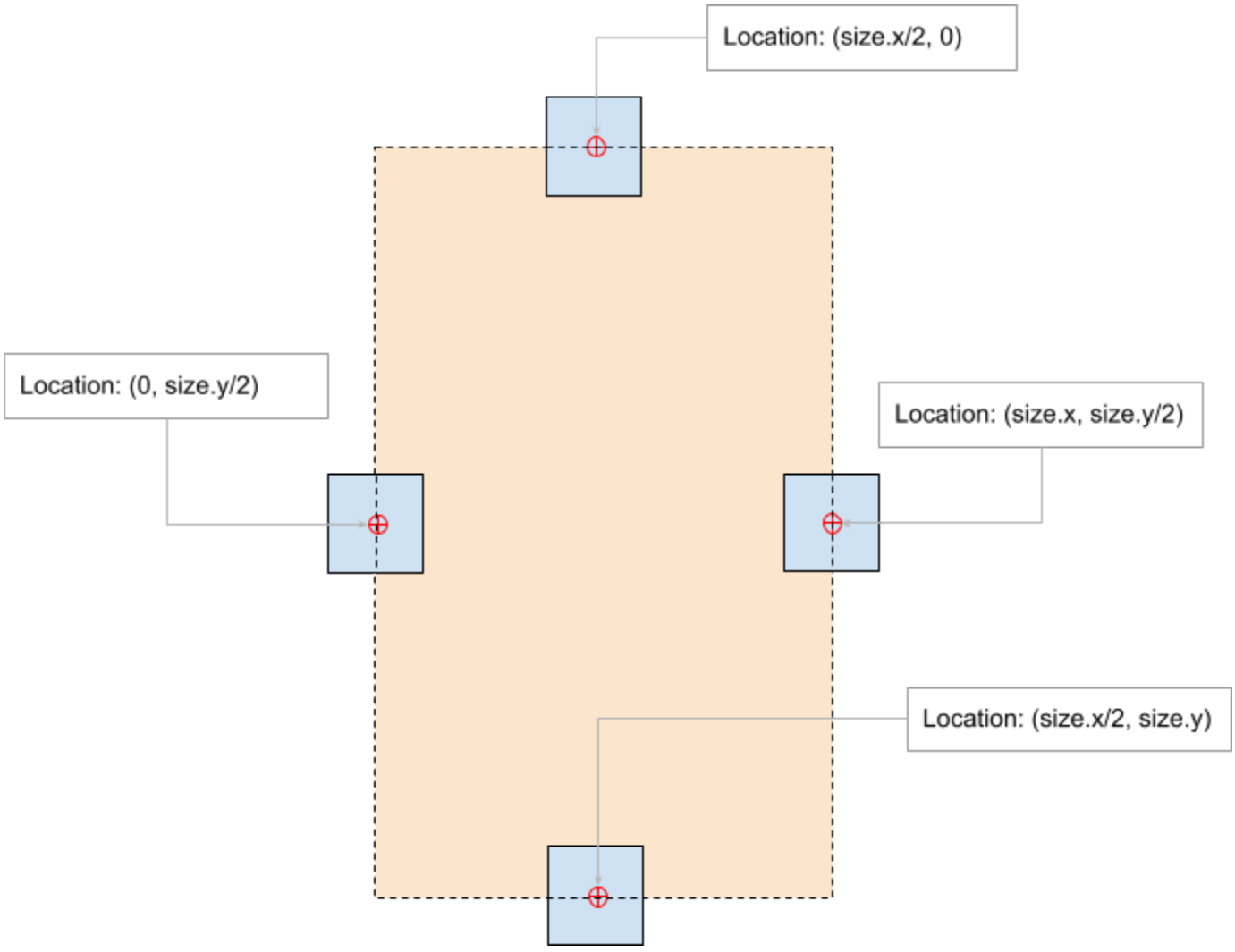
You will find the code solution in the directory

"Exercise Solutions\component_002[Joystick]\component_002_002"

Exercise #2

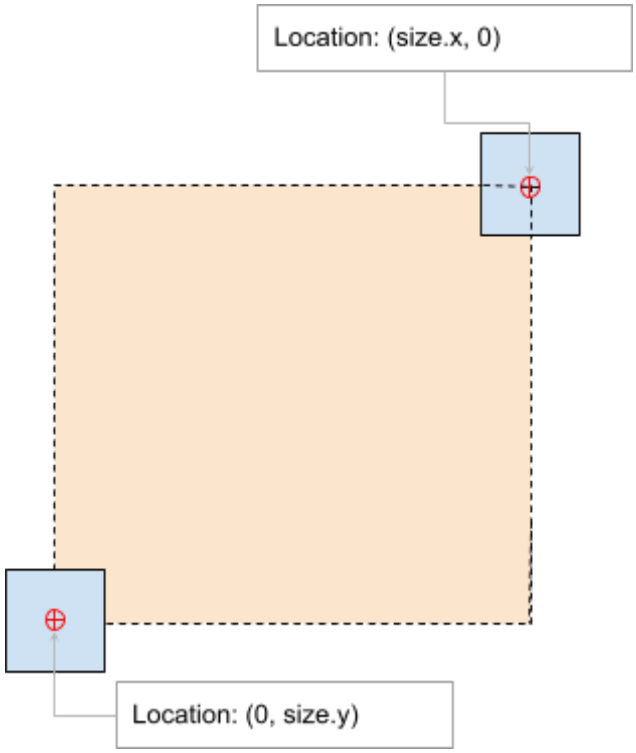
Assumptions:

1. This is really a matter of translation.
    a. We are assuming that the anchor here will be in the center. Calculations will be different for different anchor locations.
    b. We can assume that when we detect the ship to be out of bounds we simply translate the location (i.e. position) to just “wrap around” to the opposite side. Here are the basic translations to consider:



What about corner cases?

1. Corner cases are not really special here. Consider a simple case of a corner case as follows:



NOTE: what needs to be done is that the translation must consider both x and y when it is doing the wrapping around. The corner case is a case when both x and y are translated at the same time.

Solution:

1. It seems pretty obvious that to wrap around the object's position all we need to do is this:
    a. if(out-of-bounds at the top) set position.y to be size.y
    b. if(out-of-bounds at the bottom) set position.y to be 0
    c. if(out-of-bounds on the left) set position.x to be size.x
    d. if(out-of-bounds on the right) set position.x to be 0
2. We also need to be sure that we try each case before we fully update the position to be changed. Think of the corner case where both x and y have to be translated for the same position instance
3. Be careful when making the out-of-bounds comparison so that you can differentiate between just having wrapped around vs. actually hitting that edge as a collision. This is why in the comparisons in exercise #1 We do not have the <= or >= comparisons but rather < or > comparisons for the out-of-bounds.

You will find the code solution in the directory
"Exercise Solutions\component_002[Joystick]\component_002_002"