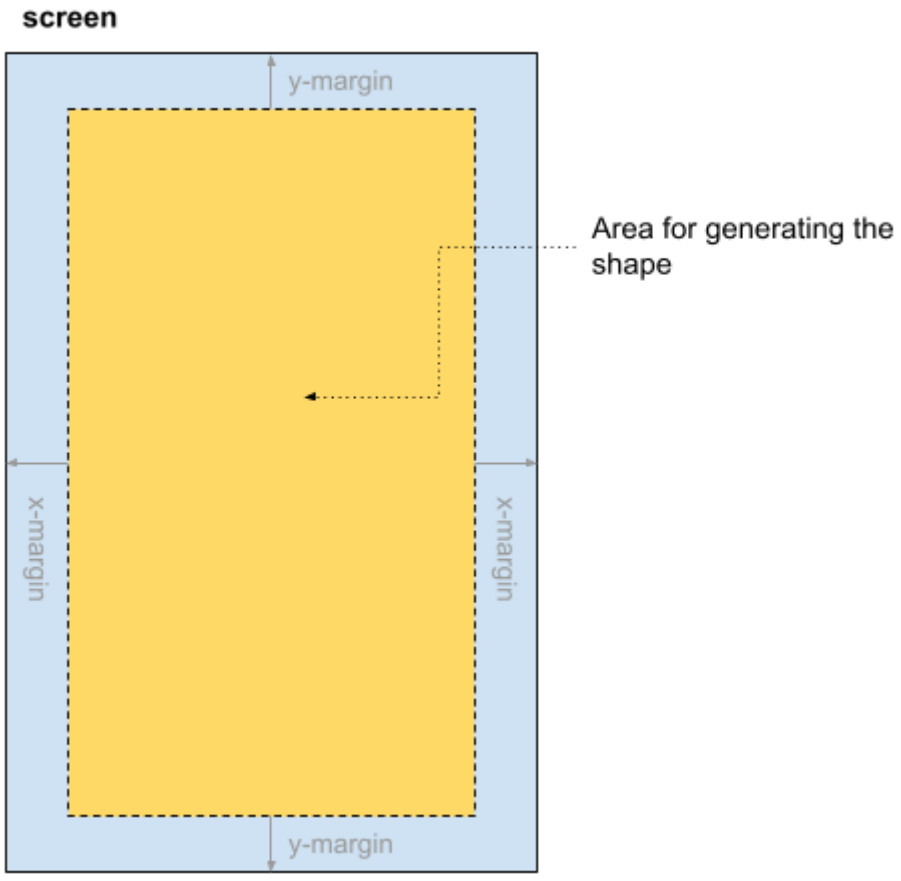
Case Study #1 - Exercises

Exercise #1

Assumptions

1. We need to create the following variables:
    a. **randomPosition** - the randomly generated initial position of the new object.
    b. **randomDirection** - the randomly generated velocity vector for the shape. This means we are looking for just the direction and we will generate this vector as a normalized (i.e. length of 1) vector
    c. **randomSpeed** - the scalar speed for our velocity vector.
    d. **randomVelocity** - will be just the multiplication of direction and speed.
    e. **xMargin** - the margin on the x side for the screen box where we will generate the shape.
    f. **yMargin** - same as above but in the y direction.



2. The bounding box for all this will be the screen size which we get through FlameGame.size vector.

Solution:

1. To generate random position without margins we simply do this:

```
new Vector2(
      Random.nextInt(0,size.x),
      Random.nextInt(0,size.y));
```

2. To generate random position with margins taken into account we do this:

```
new Vector2(
      Random.nextInt(size.x - 2*xMargin) + xMargin,
      Random.nextInt(size.y - 2*yMargin) + yMargin)
```

3. To generate random direction vector, we need to create a vector which is in the range of (-1.0, -1.0) to (1.0, 1.0) which is basically a generation of
   a. x in the range of (-1.0 to 1.0)
   b. y in the range of (-1.0 to 1.0)
   c. One simple way would be as follows:

      `(Random.nextInt(3) -1)*Random.nextDouble()` for both x and y

      NOTE: the reason for the above is that `nextInt()` generates random numbers in the range of 0..max with max being exclusive so `nextInt(3)` gives us the range of 0..2, and by subtracting 1 we are translating that range to -1..1

   d. Once you generate the random vector you then normalize with the normalized() method - this will give us a direction vector with a size of 1
4. To generate the random speed we simply assume some arbitrary range of say (25, 100) and just create a random number in that range:

```
Random.nextInt(25,100)
```

5. Now to generate the actual velocity vector (i.e. direction and speed) we do this:

```
randomVelocity = randomDirection*randomSpeed;
```

   One special case is when we generate a direction vector of (0,0) which is just no length. Normalizing this vector will do nothing so you will have basically 0 velocity. To avoid that we can simply create a while loop case as follows:

```
while(directionVector == Vector2.zero()) {
      // generate another direction vector
}
```

You will find the code solution in the directory *"Exercise Solutions\component_001[Square Shape]\component_001_001"*

Exercise #2

Assumptions:

1. We can assume right now that the position member of our Shape (Square) 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.

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.
      }
```

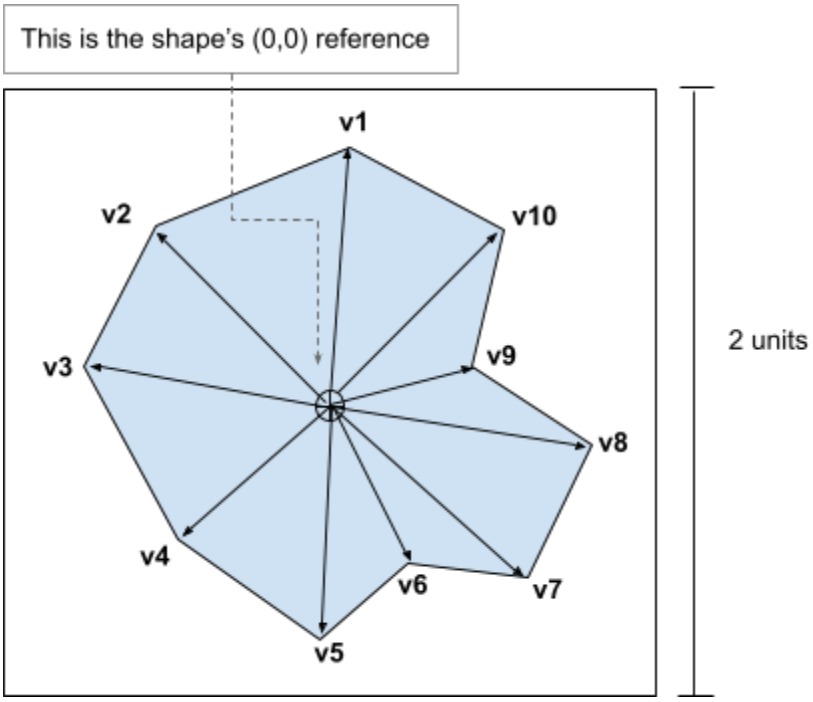
2. The main question here is who removes the out-of-bounds object? Should it be the game itself or should it be the object in question?
    a. We will go with the more efficient solution in which the actual object will remove itself from the game if it is out of bounds. This seems counterintuitive and logically it seems that a better solution would be for the game to make sure that it removes the objects out of bounds.
    b. Architecturally there are a few solutions to this:
        i. The object itself tests its own position against the screen boundaries and if it is out-of-bounds it removes itself from the game, but this means that the objects would have to be aware of the Game instance which is not an optimal solution.
        ii. The game tests every object in its tree of objects to see if any objects of interest are out-of-bounds. This is rather inefficient since it would have to test this for each object on every update() call.
        iii. The Object could test

You will find the code solution in the directory *“Exercise Solutions\component_001[Square Shape]\component_001_002”*

Exercise #3

Assumptions:

1. Here is how we could generate one of the asteroids from the Asteroids Game using a polygon. Assume the following shape:



NOTE: in this image the polygon is made up of 10 vertices (or points) that can be simply modeled as an array of Vector2 points. A couple important points for Flame engine:

1. Note that the vertices are listed **counter-clockwise**. This is the order in which you should provide them to the PolygonComponent.
2. All vertices are in the range of [-1.0, 1.0] with reference to the origin point which is in the middle of the shape. This is like a template which can then be scaled by using the size property of the component.
3. In the code example below (i.e. the solution) we present an 11 vertex 'asteroid' which is very close to the one above. Please consult the source code to see it in action.

Solution:

In this solution we create a polygon for an asteroid shape that is very similar to the above example, with 11 vertices and with all the vertices being in the range [-1.0, 1.0].
All these vertices are with reference to the middle of the bounding-square of the shape and are provided in counter-clockwise direction.

```
    final vertices = ([
    Vector2(0.2, 0.8),    // v1
    Vector2(-0.6, 0.6),   // v2
    Vector2(-0.8, 0.2),   // v3
    Vector2(-0.6, -0.4),  // v4
    Vector2(-0.4, -0.8),  // v5
    Vector2(0.0, -1.0),   // v6
    Vector2(0.4, -0.6),   // v7
    Vector2(0.8, -0.8),   // v8
    Vector2(1.0, 0.0),    // v9
    Vector2(0.4, 0.2),    // v10
    Vector2(0.7, 0.6),    // v11
  ]);


final asteroid = PolygonComponent(
  normalizedVertices: vertices,  // the vertices
  size: Vector2(100, 100),       // the actual size of the shape
  position: Vector2.all(500),    // the position on the screen
 )
```

You will find the code solution in the directory *"Exercise Solutions\component_001[Square Shape]\component_001_002"*

Exercise #4

Assumptions:

1. We will list the following member variables as well as functions we will need:

member	description	required
`currentLife`	The current life of the component. Range is 0..100 will need a getter and a setter. Access to this will be provided through special functions for modification of the values.	N
`currentColor`	The current color for the component. Starts with a healthyColor by default and then changes to either healthyColor or warningColor depending on the currentLife and the warningThreshold.	N
`warningThreshold`	Configurable threshold for when the component changes the color of the bar to warning or healthy color. 25 by default. So if currentLife < warningThreshold then the color will be set to the warning Color. Set in the constructor.	N
`warningColor`	The color used for when the warningThreshold has been passed. Red by default. Set in the constructor.	N
`healthyColor`	The color used for when the warningThreshold has **not** been passed. Green by default. Set in the constructor.	N
`size`	The size of the rectangle that represents the lifeBar. Can be initialized by the parent. IF not specified it will default to parentSize.x in width by 5 pixels in height.	N
`parentSize`	The size of the bounding rectangle of the parent to which the LifeBar will be attached.	Y
`placement`	A bit of customization that allows for the user to decide where the bar will be placed relative to the parent's top edge (i.e. left, center, or right) It is left by default.	N
`barOffset`	The user will be able to specify how much distance (vertical) there is between the life bar and the parent object. In pixels. It is 2px by default.	

2. This component will extend the **`PositionComponent`** in Flame.
3. We are assuming that the parent component will control the following:
    a. It will initialize the LifeBar with the specific size and location (relative to itself) that it needs

b. It will then add this instance to itself for composability.
c. It will then update the life data for the LifeBar as something happens to the parent object. In our case, to test it, we will simply reduce the life of the objects by tapping on it.

Solution:

1. See the provided code and the provided UML Diagram.

You will find the code solution in the directory *“Exercise Solutions\component_001[Square Shape]\component_001_003”*