{EPITECH}

BOOTSTRAP MY RADAR

INTRODUCTION TO COLLISIONS

BOOTSTRAP MY RADAR

Preambule

In order to be able to do that bootstrap correctly you should have entirely finished the first initiation bootstrap Bootstrap.
This Bootstrap will help you to :

- ✓ draw CSFML shapes in a window.
- ✓ check collisions between two rectangles.
- ✓ implement an optimised algorithm to check collisions between thousands of entities.

Starting with collisions

The CSFML library provides tools to draw simple shapes like circles, rectangles or any convex shapes.

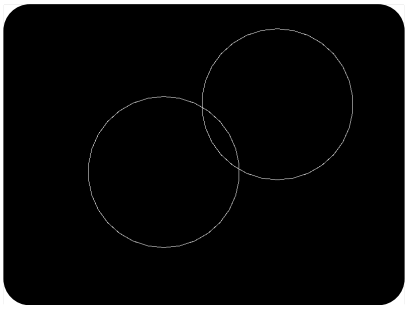
Drawing circles

First, you will draw a circle in the window. It shall have **no fill color** (or a transparent one) and the **outline shall be white with a thickness of 1**.
Write a function which creates a drawable circle and sets its properties as expected. It should have the following prototype:

```
sfCircleShape   *create_circle(sfVector2f position, float radius);
```

Have a look in the sfCircleShape documentation page.
You will find many functions that you need.

Use the so-written function to draw 2 circles in your window.

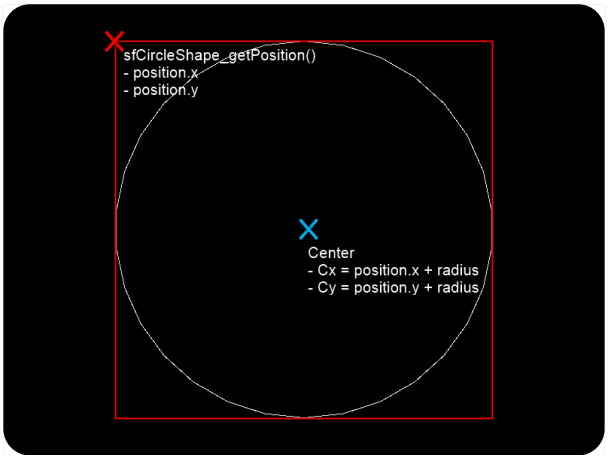


Checking intersection between 2 circles

You will now check whether 2 circles are intersecting with each other or not. If they do, you will change their **fill** color to white.

The intersection between 2 circles is calculated out of the positions of their centers and their radius.

Note that the position of the center is given by adding the radius from the positions retrieved with the function *sfCircleShape_getPosition*.

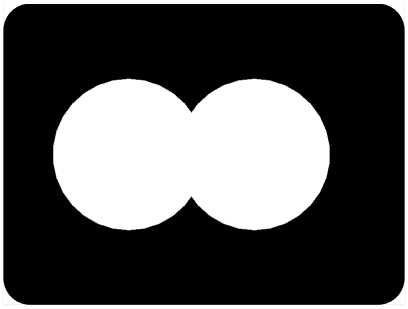


Let's consider 2 circles *C* and *D* with the respective positions of their centers *(Cx, Cy)*, *(Dx, Dy)* and their respective radius *Cr*, *Dr*,
If $(Cx - Dx)^2 + (Cy - Dy)^2 <= (Cr + Dr)^2$, then *C* and *D* intersect with each other.

Write a function which checks the intersection between two circles and returns 1 if they intersect, 0 otherwise:

```
int     is_intersecting_circles(sfCircleShape *c1, sfCircleShape *c2);
```

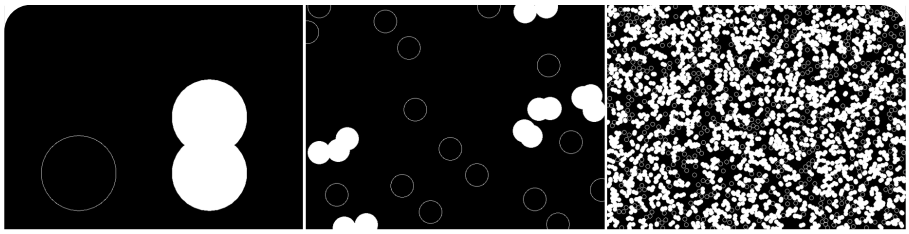
Use this function to fill your circles in white if they intersect/collide. Leave them transparent if they don't.



In this bootstrap, we use circles for collision checking. Note that there are many functions and algorithms for collision calculation.
For instance, the Axis-Aligned Bounding Boxes (abbr. AABB) is one the most famous colliding function between axis-aligned shapes.
Another famous one is the Separated Axis Theorem which allows you to check collisions between two different convex shapes.

Checking collisions with N circles

Repeat the operation with 3, 50, 500 and 5000 circles. Remember to color any colliding circle in white.



3 circles 30 circles 3000 circles

Make them move

Moving circles over the screen implies to calculate the collision between them at each frame.

At each frame, make the circles move randomly between -1, 0 or 1 pixel(s) on either or both the x and y axis.
Repeat the operation with 3, 50, 500 and 5000 circles.

#warn(How does the FPS go as you increase the number of circles ? Why ?)

You can display the FPS on the terminal by calling the function as the first instruction in your game loop:

```
void    print_framerate()
{
  static int            first = 1;
  static sfClock        *clock;
  static int            fps = 0;

  if (first == 1)
    {
      clock = sfClock_create();
      first = 0;
    }
  sfTime elapsed = sfClock_getElapsedTime(clock);
  if (sfTime_asSeconds(elapsed) >= 1)
    {
      printf("%3d FPS\n", fps);
      fflush(stdout);
      fps = 0;
      sfClock_restart(clock);
    }
  else
    fps++;
}
```

Optimising algorithm

A way to optmise the collision calculation is to decrease the number of operations.
Indeed, it is not useful to check the collision between each and every entity.

Dividing screen

Let's consider the windows with 4 parts:

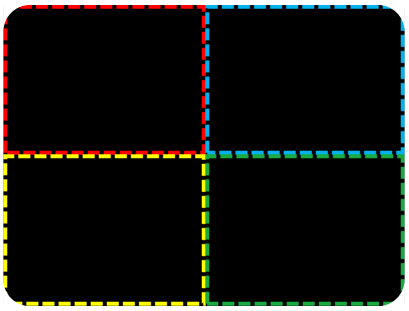
- ✓ top-left
- ✓ top-right
- ✓ bottom-left
- ✓ bottom-right

You can use this structure to represent a part/corner of the window:

```
struct corner
{
  sfIntRect  area;
}
```

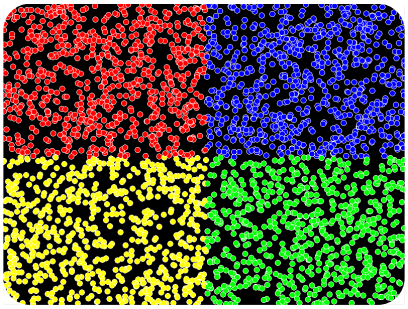
The *area* field represents the rectangle covered by the corner.

Check out the structure sfIntRect in the documentation



Dispatch circles

You can now assign each and every circle to one (and only one) corner of the window, depending on the position of the center.



We need to modify the structure *corner* to save the related circles inside:

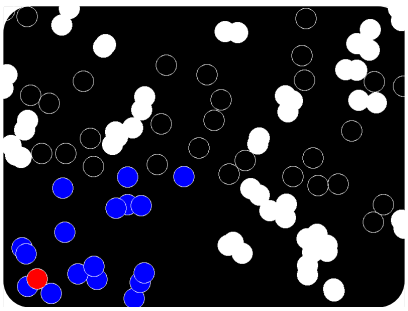
```
struct corner
{
  sfIntRect      area;
  unsigned in    nb_circles:
  sfCircleShape **circles;
}
```

The *circles* field is a NULL-terminated array of *nb_circles* circles.

Submit a query

You are in the main step of the algorithm: retrieving only nearby circles to a given one to limit the collision calculation. As said before, there is no need to check every circle with every other.

You only need to calculate the collision between a circle and the other circles of the same corner.

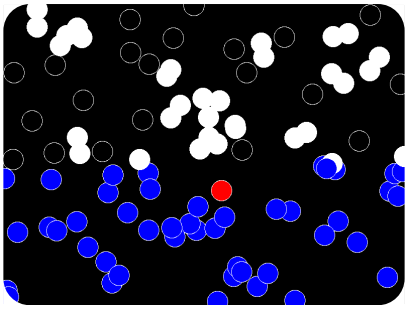


It will work for most cases but for circles that are straddling over two corners. For this specific case, you need to use a *query*.

For every circle C, we define an area in which any colliding circle would have its center inside. This area is called *query* and is defined as followed:

```
sfIntRect query = {Cx - Cr, Cy - Cr, Cr * 2, Cr * 2}
```

Now you have a query, you have to retrieve all circles from any corner whose *area* intersects with the *query*.



The function sfIntRect_intersects allows you to simply check if two sfIntRect are intersecting.

Check collisions

You can test the collision using *is_intersecting_circles* with the limited circles retrieved from the previous step. Keep drawing the circles in white if they collide with other any other.

Run the program and note how the FPS have improved.

You now have a base for a famous collision checking algorithm, call **grid-based collisions**. It can be even more efficient if you keep dividing your current 4 corners into 4 subcorners, and so on.

What is the main drawback of this algorithm ? In which case is it the less efficient ? How to counter it ?

v 2.4