BRAC UNIVERSITY

Inspiring Excellence

Department of Computer Science and Engineering

Course Code: CSE 423	**Credits: 0.75**
Course Name: Computer Graphics	**Semester: Fall24**

Lab 02

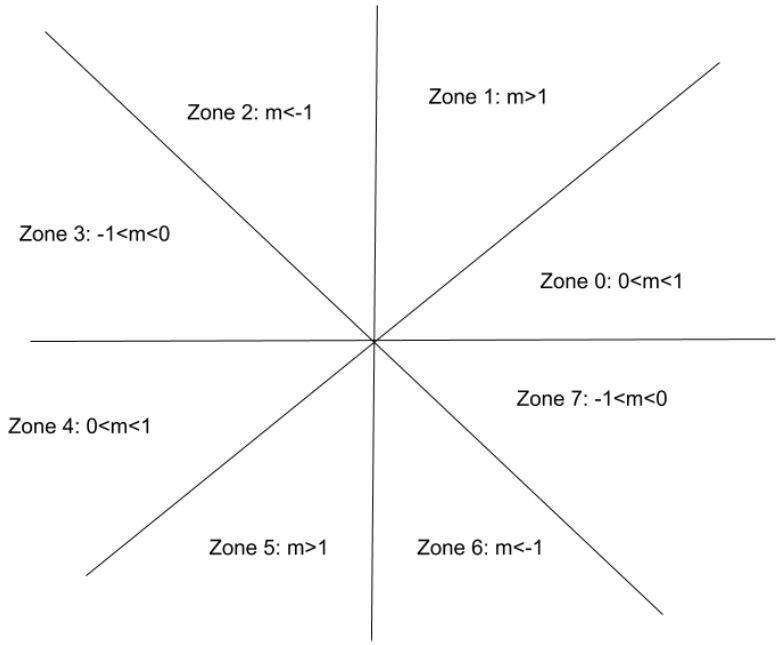
Midpoint Line & Circle Drawing Algorithms

I. Topic Overview:

We will be introducing the midpoint line drawing algorithm in today's class. This algorithm tries to find the best approximation of the next point on the line using some criteria. The benefit of using approximation over exact points is to speed up the whole drawing process - making it particularly suitable for practical implementation. To keep things simple, we would make the following assumptions:

- The line will be drawn from left to right
- For the given coordinates (x1, y1) & (x2, y2): x1<x2 & y1<y2
- The slope of the line is between 0 & 1 (Zone-0), i.e., we will be writing a program to draw a line from the left bottom to the top right.

However, a line can have any slope ($-1 \leq m \leq 1$) and any direction. Therefore, we have to consider a given line can be in any one of the 8 zones in the following figure. We would be implementing the mid-point line drawing algorithm only for Zone 0 but would adjust our program so that it can draw any line in any zone.



We will achieve our goal of using a single method to draw lines by manipulating a special property of lines: **"Eight-way symmetry".** For this purpose,

- First, c**onvert the point in any zone to a point in zone 0**.
- Then we will simply r**un Zone 0's line drawing method**.
- Finally, **convert back the point in Zone 0 to a point in its original zone** before drawing the pixels on the screen.

Next, we will be introducing a similar approach to draw a circle. This algorithm tries to find the best approximation of the next point on the circle using a decision parameter. We would be drawing a part of the circle, let's say only zone 0. Then, we will apply the 8-way symmetry approach to draw the portion of the circle for the rest of the zones.

- Implementation of Midpoint circle drawing algorithm using 8-way symmetry.
- The students should have a clear understanding of the 8-way symmetry approach.
- They should also have a clear idea about all the different zones.

II. Anticipated Challenges and Possible Solutions

a. The students need to carefully convert a point from any zone to a point in zone 0. They also need to convert back the points to their original zone before drawing the pixels. Most of the time, the students aren't careful during conversion & make mistakes!

b. The students should write their algorithm in a way so that the drawing of the line doesn't become dependent on the order of endpoints of a line. For example, it should not be the case that the implementation of the algorithm draws the line properly for (X_1 , Y_1) and (X_2 , Y_2) points but fails when points are given in reverse order (X_2 , Y_2) and (X_1 , Y_1). This should not be the case.

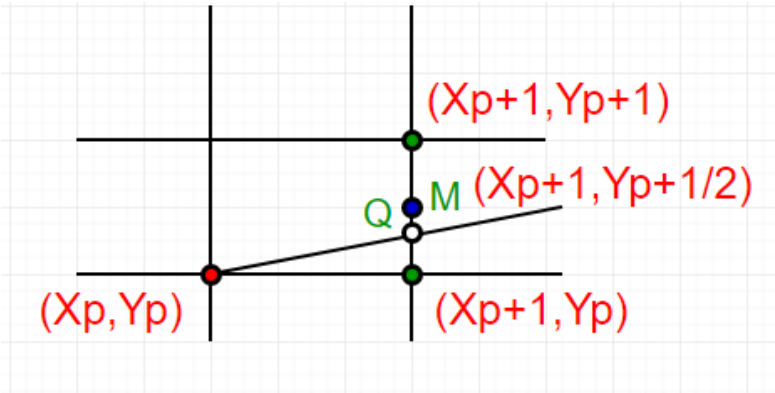
Solutions: Swap!

c. Students should understand why and how we are drawing a point of a particular point to another zone and in which zone the calculations are being done for drawing the circle.

III. Activity Detail

For Midpoint Line Algorithm:

1. The basic idea is as follows: For any given/calculated previous pixel $P(X_p, Y_p)$, there are two candidates for the next pixel closest to the line, $E(X_p+1, Y_p)$ and $NE(X_p+1, Y_p+1)$. In the Mid-Point algorithm we do the following:
   - **A.** Find the mid-point of two possible next points. Middle of $E(X_p+1, Y_p)$ and $NE(X_p+1, Y_p+1)$ is $M(X_{p+1}, Y_p+1/2)$.
   - **B.** If M is above the line, then choose E as the next point.
   - **C.** If M is below the line, then choose NE as the next point.



How to find if a point is above a line or below a line?

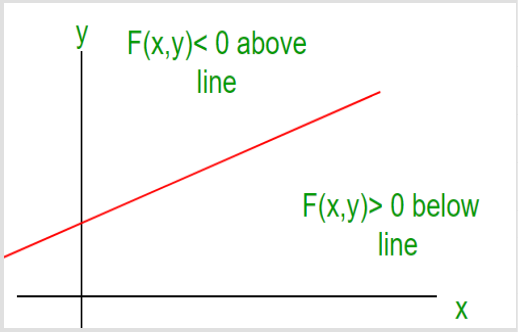
Let us consider a line y = mx + B.

We can re-write the equation as :

y = (dy/dx)x + B or

(dy)x + B(dx) - y(dx) = 0

Let **F(x, y)** = (dy)x - y(dx) + B(dx) -----(1)
Let we are given two end points of a line (under above assumptions)
-> For all points (x,y) on the line,
the solution to F(x, y) is 0.
-> For all points (x,y) above the line,
F(x, y) results in a negative number.
-> And for all points (x,y) below the line,
F(x, y) results in a positive number.



This relationship is used to determine the relative position of M.
The algorithm works as follows:

```
DrawLine(int x1, int y1, int x2, int y2)
{
   int dx, dy, d, incE, incNE, x, y;
   dx = x2 - x1;
   dy = y2 - y1;
   d  = 2*dy - dx;
   incE  = 2*dy;
   incNE = 2*(dy - dx);
   y = y1;
   for (x=x1; x<=x2; x++)

   {
      WritePixel(x, y);
      if (d>0) {
         d = d + incNE;
         y = y + 1;
      } else {
         d = d + incE;
      }
    }
}
```

Discussion on Midpoint Line Algorithm:

First of all, given the coordinates of any two points on the line, the students need to convert them into points of Zone-0. For this purpose, the students need to determine in which zone the given coordinates are.

```
int FindZone(int x1, int y1, int x2, int y2)
{
    if (abs(dx)>=abs(dy)){
            if(dx>0 && dy>0)
            Zone=0;
        //Write conditions for other zones
}
else{
        if(dx>0 && dy>0)
            Zone=1;
        //Write conditions for other zones
}
return Zone ;
}
```

Now, to convert the coordinates of any zone to the coordinates of zone 0: For example in Zone 2: (x<0, y>0 & abs(dy)>abs(dx)) while in Zone 0: (x>0, y>0 & abs(dx)>abs(dy)). To convert a point of zone 2 to a point of zone 0, we need to swap its x & y. Students need to figure out the required conversion for points in other zones.

The basic steps are as follows:

1. Given two points (x1, y1) & (x2,y2)
2. $Zone_1$=findZone(x1, y1) & $Zone_2$=findZone(x2, y2). For simplicity, we assume both endpoints of the line are in the same zone, i.e., $Zone_1 = Zone_2$
3. Convert (x1, y1) from $Zone_1$ to a point of Zone-0, say (x1', y1')
4. Convert (x2, y2) from $Zone_2$ to a point of Zone-0, say (x2', y2')

5. Run mid-point line drawing algorithm for zone 0 using (x1', y1') & (x2', y2') as input co-ordinates.
6. Calculate the intermediate points (x, y).
7. Now before drawing the pixels, convert (x,y) to its original zone.

For Midpoint Circle Algorithm:

A circle is defined as a set of points that are all at a given distance r from a center positioned at (x_c,y_c).

So, the equation of the circle according to the center (x_c, y_c) is,

$$(x - x_c)^2 + (y - y_c)^2 = r^2 \quad \ldots\ldots\ldots\ldots\ldots\ldots 1$$

from equation 1 we get the value of y as below,

$$y = y_c \pm \sqrt{r^2 - (x - x_c)^2} \quad \ldots\ldots\ldots\ldots\ldots\ldots 2$$

As $f(x, y) = 0$ represents a circle, Let,

$$f(x, y) = (x - x_c)^2 + (y - y_c)^2 - r^2 \quad \ldots\ldots\ldots 3$$

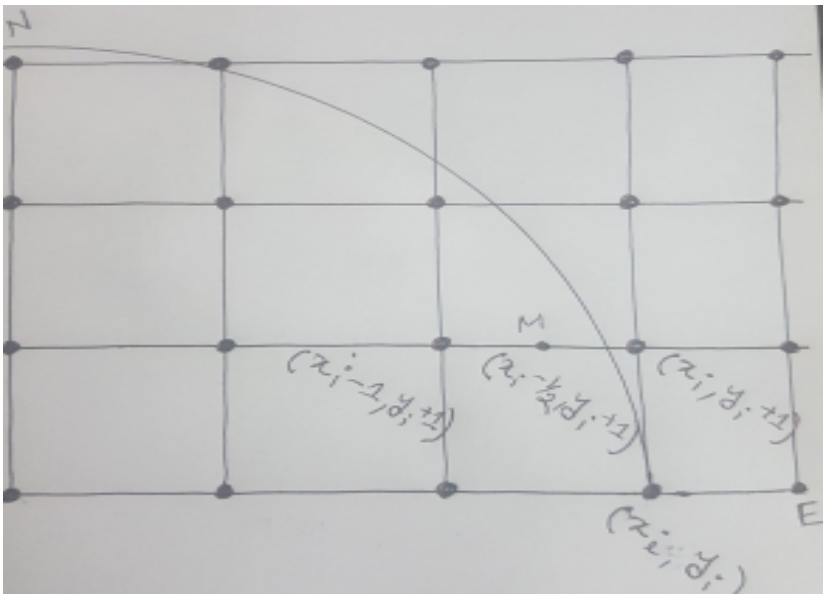
Now, for any point,

$f(x, y) = 0$, the point is on the circle

$f(x, y) > 0$, the point is outside the circle

$f(x, y) < 0$, the point is inside the circle

In the midpoint circle drawing algorithm at each of the i^{th} steps we will be evaluating the circle function to come to a decision. Assuming we have just plotted the i^{th} pixel at (x_i,y_i), we next need to determine whether the pixel at the position (x_i+1,y_i) or the one at (x_i,y_i+1) is closer to the circle.



Our decision parameter d_i at the i^{th} step is the circle function evaluated at the midpoint of these two pixels. The coordinates of the midpoint of these two pixels $_i - {}_2^1 y_i + 1$ is $(x,)$.

Hence, $d(x,)\, x)\, y)$ …….. $4\ _i = f_i - {}_2^1 y_i + 1 = (_i - {}_2^1)^2 + (_i + 1^2 - r^2$

Successive decision parameters are obtained using incremental calculations, thus avoiding a lot of computation at each step. We obtain a recursive expression for the next decision parameter.

$$_{i+1} = f_{i+1} - {}_2^1 y_{i+1} + 1 = (_{i+1} - {}_2^1)^2 + (_i + 1 + 1^2 - r^2$$

Therefore, $d(x,)\, x)\, y)\ d\, x)\, y)$ …….. $5\ _{i+1} = (_{i+1} - {}_2^1)^2 + (_i + 2^2 - r^2$ Now we get by subtracting eqn (4) from eqn (5)

$$\Delta d = d_{i+1} - d_i$$

$$_{i+1} - {}_2^1)^2 - (_i - {}_2^1)^2 + (_i + 2^2 - (_i + 1^2 - r^2 + r^2$$

$$\Delta d = (x)\, x)\, y)\, y)$$

$\Delta d = (x).\ (x)\, y$ …… $6\ _{i+1} + x_i - 1\ _{i+1} - x_i + 2\ _i + 3$ Now if $d =$, then the midpoint of the two possible pixels lies within the circle, $_i < 0$ thus the north pixel is nearer to the theoretical circle.

Hence, x. Substituting this value in eqn (6), we get, $_{i+1} = x_i\, \Delta d = d\, x).(x)\, y$

$_{i+1} - d_i = (_i + x_i - 1\ _i - x_i + 2\ _i + 3$

$$d\, x).0\, y\ _{i+1} - d_i = (_i + x_i - 1 + 2\ _i + 3$$

$$d\,y_{i+1} = d_i + 2\,_i + 3$$

And if we consider d then, the midpoint of the two possible pixels lies $_i > 0$ outside the circle, so the northwest pixel is nearer to the theoretical circle. Therefore, x. Substituting this value in eqn (6), we get, $_{i+1} = x_i - 1\ \Delta d = d\,x$). $(x)\,y_{i+1} - d_i = (_i - 1 + x_i - 1\ _i - 1 - x_i + 2\ _i + 3$

$$d\,2x).(-)\,y_{i+1} - d_i = (_i - 2\,1 + 2\ _i + 3$$

$$d\,x\,y_{i+1} = d_i - 2\ _i + 2 + 2\ _i + 3$$

$$d\,x\,y_{i+1} = d_i - 2\ _i + 2\ _i + 5$$

For the initial decision parameter $x = r$, $y = 0$. So, from eqn (4) we get, $d\,r)\,0)_0 =$

$$(-\ _2 1^2 + (+ 1^2 - r^2$$

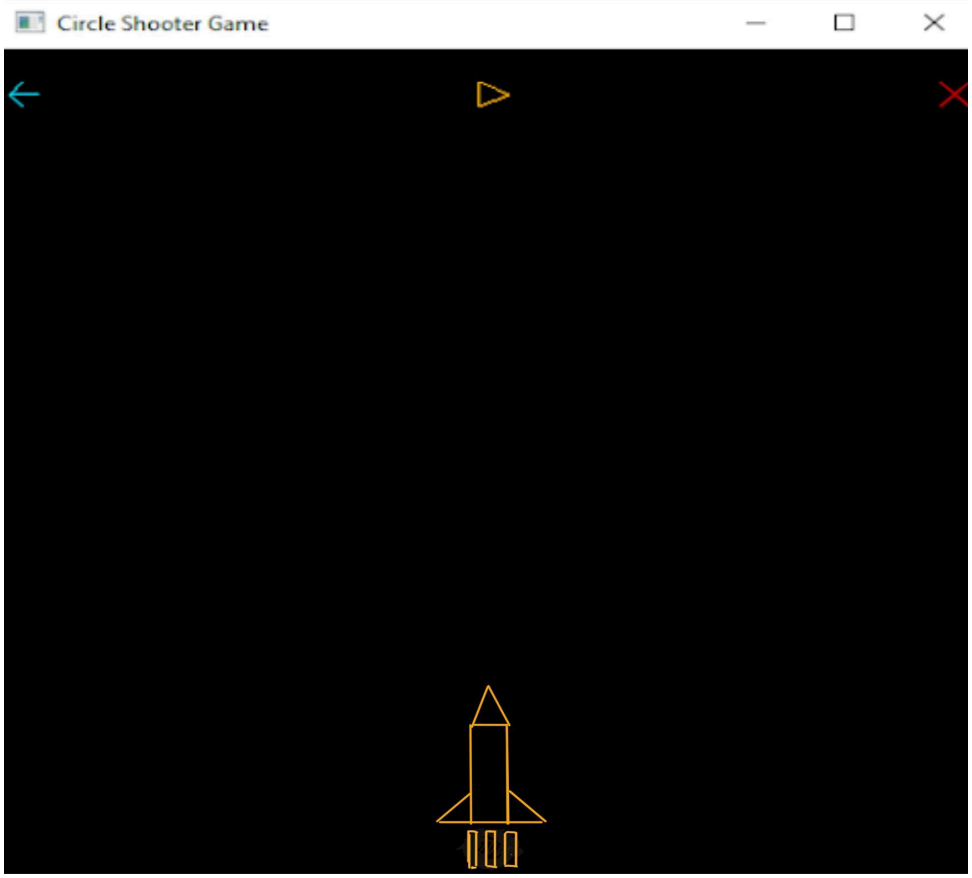
$d_0 = _4^5 - r$

To get the integer value of pixel coordinates, we approximate,

d …….. $7_0 = 1 - r$

IV. Activity Task

In this lab, students will create a simple 2D game called "Shoot The Circles!" from scratch. The objective is to shoot falling circles before they reach the ground. Players score points for every successful hit but must avoid missing three consecutive circles, which ends the game. The lab includes a shooter spaceship at the bottom, controlled with 'a' and 'd' keys for horizontal movement, and the ability to shoot upwards with the spacebar.



Game Rules:

1. **Using the midpoint line and circle drawing algorithms, you must draw everything on screen. With that being said, you're only allowed to use the `GL_POINTS` primitive type.**
2. **Shooter Control:** A shooter spaceship is placed at the bottom of the player's screen. It can be moved horizontally using the 'a' and 'd' keys. The shooter's movement is restricted to staying within the screen boundaries.
3. **Firing Mechanism:** Players can shoot by pressing the **spacebar**, sending a fire projectile (circles) upwards on the screen toward the falling circles.

4. **Falling Circles:** Circles fall vertically from the top of the screen. To score a point, the shooter must hit a falling circle directly. The horizontal position of each falling circle **is** randomized.
5. **Scoring:** Successfully hitting a falling circle increases the player's score by 1 and both the projectile and falling circle will be removed from the screen upon hit. The current score is displayed in the console for tracking.
6. **Game Over:** The game can be over in any of the first two following ways:
    i. If a falling circle touches the spaceship shooter directly, the game will be over immediately.
    ii. If the player misses three falling circles to shoot, i.e. falling circles crossing the bottom boundary before the spaceship shooter manages to vanish them three times, the game will be over.
    iii. Bonus and Optional: The game will also be over if a player misfires, i.e. shoots but fails to hit any falling circle three times, the game will be over.

    In this state, falling circles disappear, shooter movement is disabled, and "Game Over" is displayed in the console along with the final score.
7. **Control Buttons:** Three clickable buttons are positioned at the top of the screen:
    a. Left Arrow (Restart): Clicking this button restarts the game, resetting the score and circle speed. A message like "Starting Over" is shown in the console.
    b. Play/Pause Icon (Amber Colored): This button toggles the game between play and pause states. The icon represents the current state. In the pause state, falling circles freeze, and shooter movement is disabled.
    c. Cross (Red Colored): Clicking this button prints "Goodbye" in the console along with the final score and terminates the game.

8. **Circle Drawing:** The size of the circles is designed to be visible but not overly large.

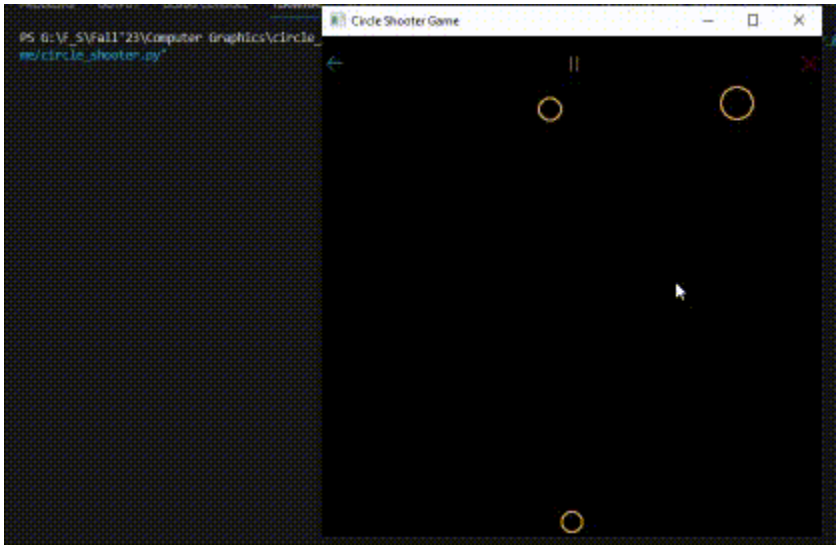
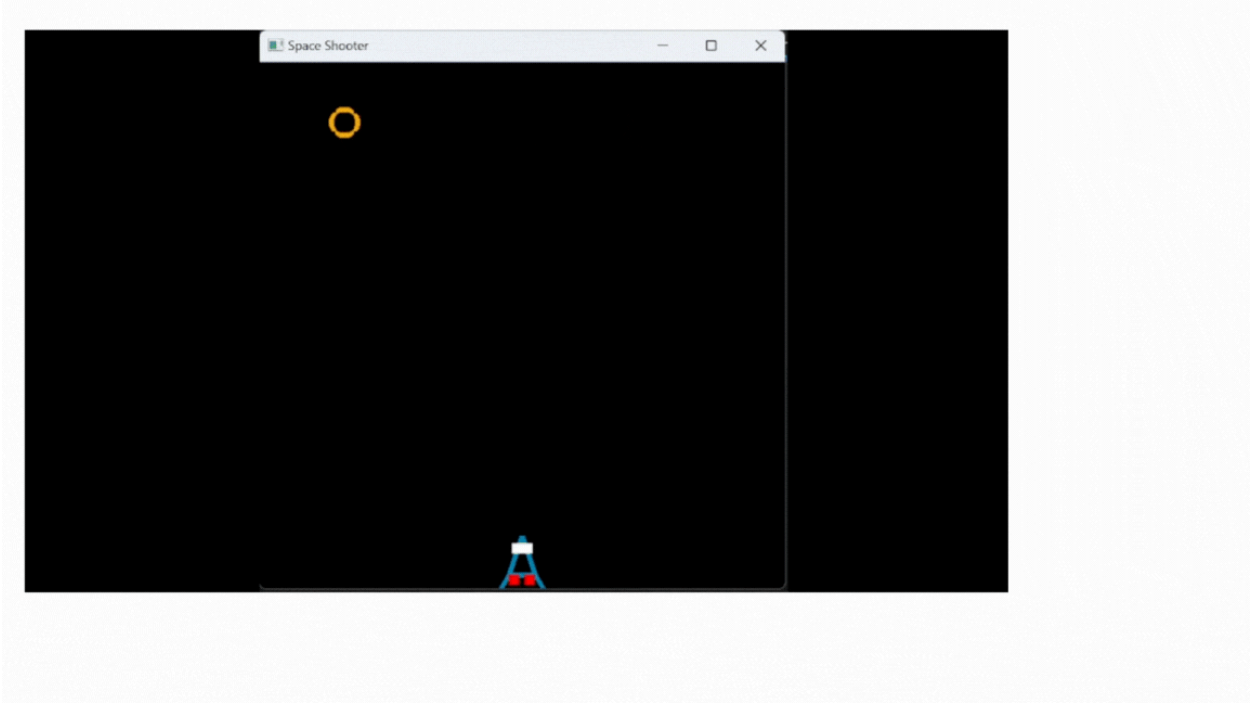


Fig: Shoot the Circles, Basic Gameplay

*There will be a spaceship here shooting, instead of a circle

9. **Unique Falling Circles:** Occasionally, there will be a circle falling whose radius will expand and shrink simultaneously. Hitting this circle will earn more points.



Tips:

- For collision detection, you can rely on the AABB (Axis-Aligned Bounding Box) collision detection algorithm for 2D. Which detects if two rectangles (or boxes) are colliding/intersecting with each other. You can consider the circles and spaceship to have their own respective bounding boxes.

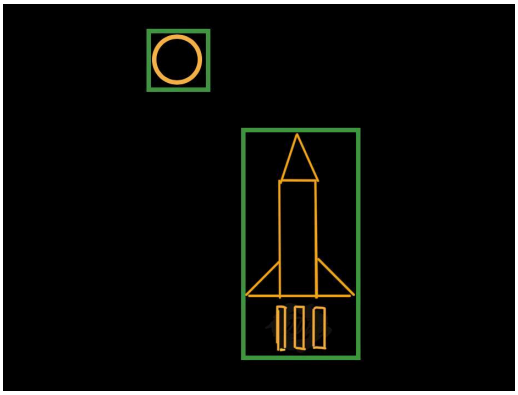


Figure: Objects showing their AABBs (shown as thin green rectangles)

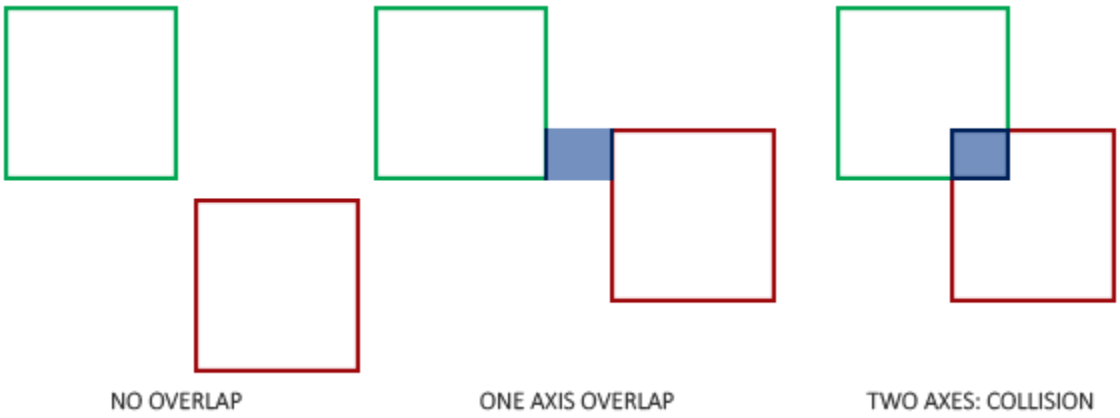


Figure: AABB collision (courtesy of learnopengl.com)

Here is the pseudocode for AABB collision detection:

```
bool hasCollided(AABB box1, AABB box2) {
    return box1.x < box2.x + box2.width  &&
           box1.x + box1.width > box2.x  &&
           box1.y < box2.y + box2.height &&
           box1.y + box1.height > box2.y;
}
```

- For terminating the application from code, you can call the **`glutLeaveMainLoop()`** function. (Try to guess through trial and error why this is preferred over using Python's usual **`exit()`** function).
- You can adjust the circles' acceleration and the spaceship's speed to tune the difficulty of the game.
- Do you know that different computers/devices have different refresh rates? And CPU performance can have an effect on the frame rate as well. So, animated objects might have different speeds depending on the machine. To tackle this, rather than **blindly incrementing/decrementing shape coordinates by a constant value**, it is preferable to calculate displacement/velocity in real-time using the **time elapsed between two frames**, also known as **delta time** (Δt). The knowledge of high school physics formulas will come in handy here. And for getting time as a UNIX epoch, you can simply rely on Python's **`time`** module. Helpful Wikipedia article on delta timing here.