BRAC UNIVERSITY

Inspiring Excellence

Department of Computer Science and Engineering

Course Code: CSE 423	**Credits: 0.75**
Course Name: Computer Graphics	**Semester: Fall23**

Lab 02

Midpoint Line Drawing Algorithm

I. Topic Overview:

The students were introduced to the DDA line drawing algorithm in the theory class. Given the coordinates of any two points on the line: (x1, y1) & (x2, y2), the student's task is to find all the intermediate points required for drawing the line on the computer screen of pixels where every pixel has integer coordinates. The DDA algorithm works fine but is slow due to floating point arithmetic.

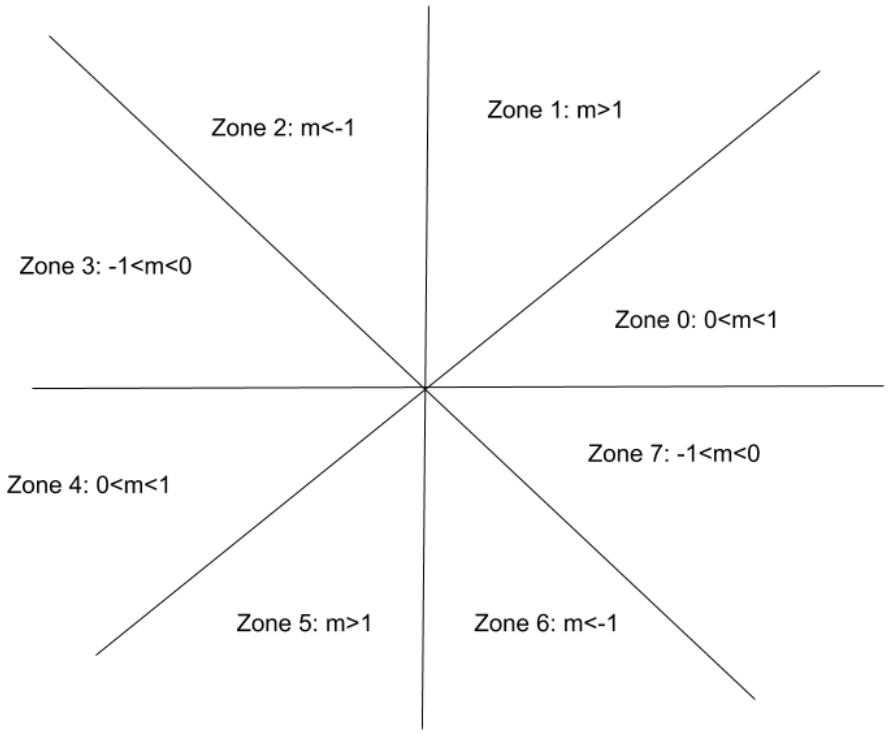
Therefore, we would be introducing a midpoint line drawing algorithm in today's class to speed things up a little bit by using integer arithmetic. This algorithm tries to find the best approximation of the next point on the line using some criteria. The benefit of using approximation over exact points 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
- Slope of the line is between 0 & 1 (Zone-0), i.e., we will be writing a program to draw a line from left bottom to top right.

However, a line can have any slope ($-1 \leq m \leq 1$) and any direction. Therefore, we need to adjust our program so that it can handle every single case of a line. We can consider a given line can be in any one of the 8 zones in the following figure. We would be

implementing mid-point line drawing algorithm only for Zone 0, but would adjust our program so that it can draw any line in any zone.

The students might think that they could implement separate methods to draw lines in 8 different zones. Though the methods are supposed to be quite similar to each other, it's a lot to have 8 separate methods to draw a line!



Couldn't we get rid of all these redundant methods & have only one method that would handle lines in all 8 zones? It would make things more compact, flexible & robust. In today's class we will introduce that idea to the students!

We will achieve our goal of using a single method to draw lines by manipulating a special property of lines: **"Eight way symmetry".** The key idea is that we will use Zone-0's line drawing method to draw ***any line in any zone.*** For this purpose,

- First we need to map any point in any zone to a point in zone 0 [**Convert point in any zone to point in zone 0**].
- Then we will simply use Zone-0's line drawing method to calculate the intermediate points representing the line [**Run Zone 0's line drawing method**].
- Finally, before drawing the pixels on the screen, we need to convert back the points in Zone 0 to its original zone [**Convert back point in Zone -0 to a point in its original zone**].

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!

   Solutions:

   i. Have a sound idea about the coordinates & slopes in different zones.

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 end points of a line. For example, it should not be the case that the implementation of the algorithm draws 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.

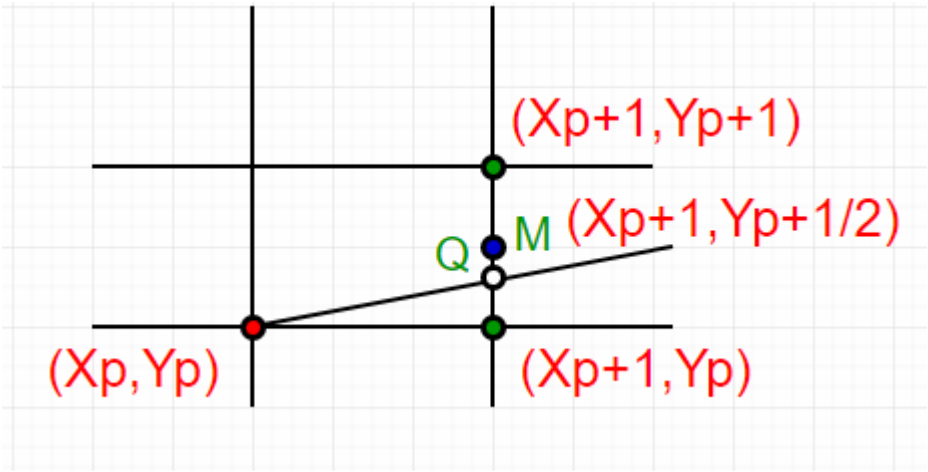
   Solutions:

   i. Swap!

III. Activity Detail

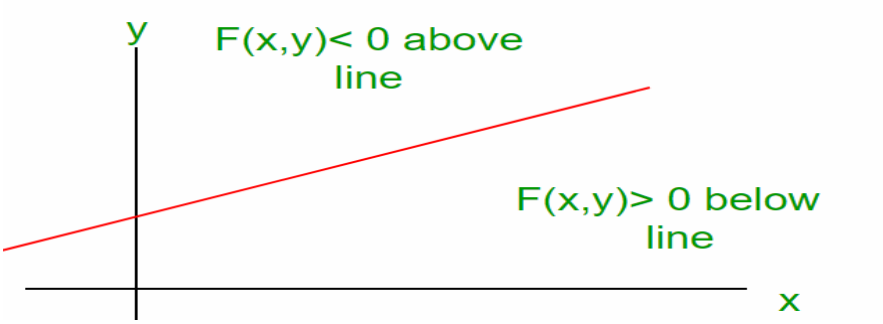
The basic idea is as follows: For any given/calculated previous pixel $P(X_p, Y_p)$, there are two candidates for the next pixel closest to the line, $E(X_p+1, Y_p)$ and $NE(X_p+1, Y_p+1)$ (**E** stands for East and **NE** stands for North-East).
In Mid-Point algorithm we do following.

1. Find middle of two possible next points. Middle of $E(X_p+1, Y_p)$ and $NE(X_p+1, Y_p+1)$ is $M(X_{p+1}, Y_p+1/2)$.
2. If M is above the line, then choose E as next point.
3. If M is below the line, then choose NE as 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) result in a negative number.
-> And for all points (x,y) below the line,
      F(x, y) result in a positive number.
```



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

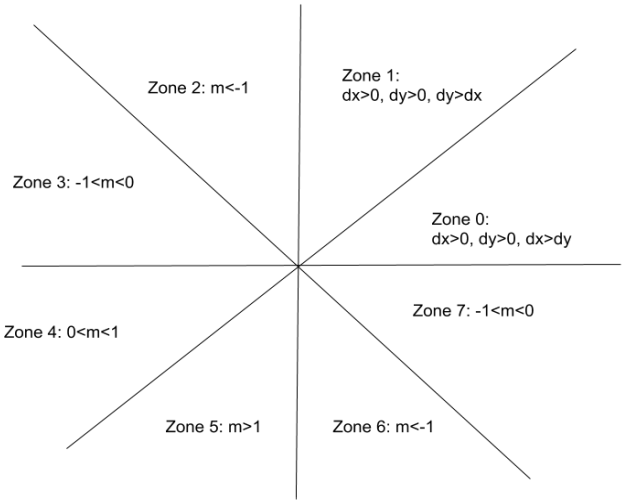
```
DrawLine(int x1, int y1, int x2, int y2)
{
   int dx, dy, d, incE, incNE, x, y;
   dx = x2 - x1;
   dy = y2 - y1;
   d   = 2*dy - dx;
   incE   = 2*dy;
   incNE = 2*(dy - dx);
   y = y1;
   for (x=x1; x<=x2; x++)

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

a. **Hour: 2**

Discussion:

During this period the teachers will discuss how the students can utilize the above mentioned line drawing algorithm for Zone-0 to draw lines in other zones. First of all, given the co-ordinates of any two points on the line, the students need to convert them in points of Zone-0. For this purpose, the students need to determine in which zone the given co-ordinates 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 co-ordinates of any zone to the coordinate 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

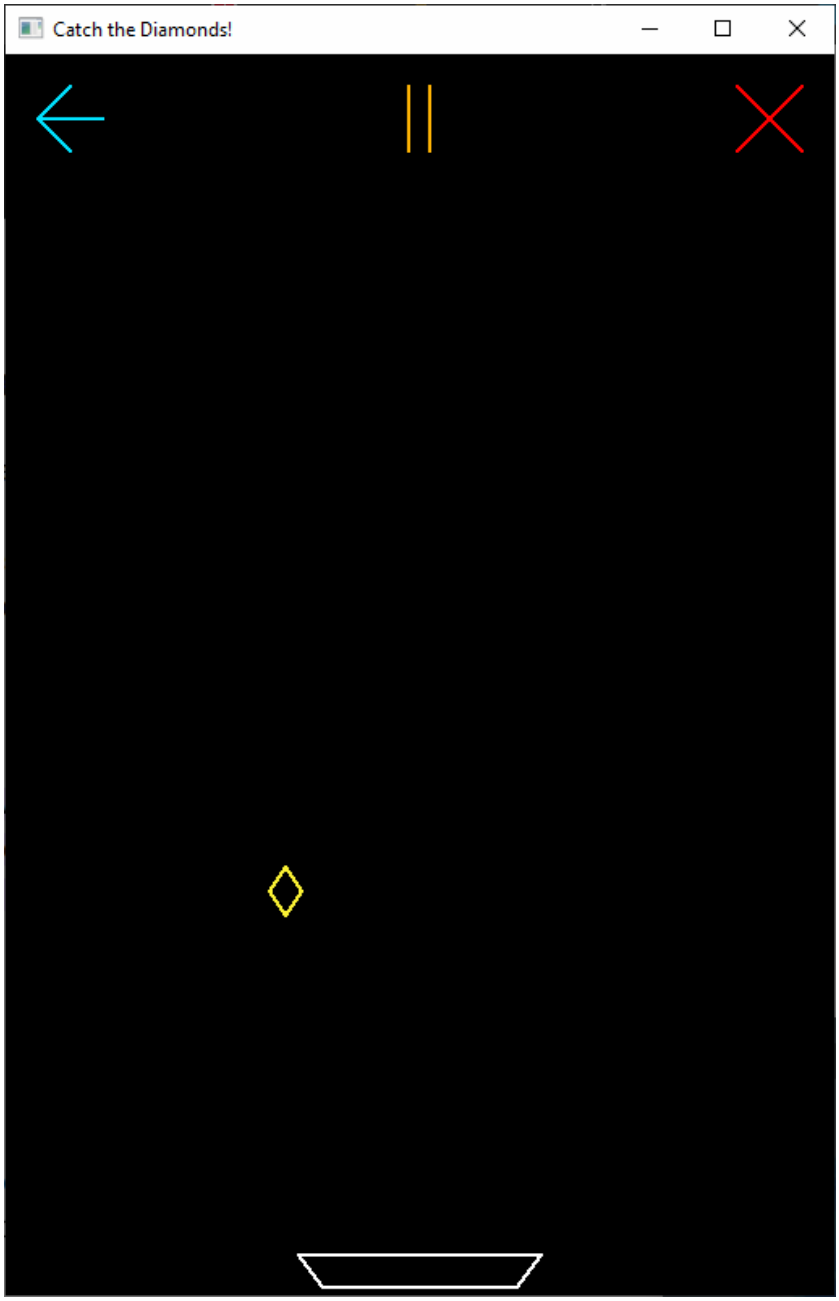
point of zone 2 to a point of zone 0, we need to swap its x & y. Since y>0, the x co-ordinate of the point in Zone 0 will be, x of zone 0=y of zone 2. But since x<0 in zone 2, the y co-ordinate of the point in Zone-0 will be, y of zone 0= -x of zone 2 so that it becomes positive. The students need to figure out the required conversion for points in other zones.

The students will convert the given co-ordinates of the line to co-ordinates of Zone-0. Then they will use Zone-0's midpoint line drawing algorithm to calculate the intermediate points. However, before drawing the pixels the students need to convert them back to their original zone. For example, given any point (x,y) in Zone -0, if we want to convert it back to its original zone, say zone 2, then we need to swap its x & y coordinates again. Since x>0 in zone 0 then the point's y co-ordinate in zone 2 will be y of zone 2=x of zone 0. But since y>0 in zone 0 but x<0 in zone 2, zone 2's x co-ordinate will be x of zone 2= -y of zone 0. So, 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 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.

Activity Task

In this lab, students will implement a simple 2D game called **"Catch the Diamonds!"**. There will be diamonds falling from the top, your goal is to catch them before they hit the ground. The more diamonds you catch, the greater you score. But if you miss any of the diamonds, your game is over. You then have to start over again.



Rules:

- There is a catcher bowl at the bottom of the player screen, which can be moved horizontally using left and right arrow keys, please make sure the catcher doesn't get out of the screen.
- The diamonds fall vertically from the top of the screen. For a diamond to be regarded as "catched", the catcher must be right beneath it at the right moment (meaning the catcher and diamond have to collide with each other). There will be one diamond falling at a time on the screen.
- If a diamond is "catched", your score will increase by 1, and the current score should be printed on the console. After that, a new diamond will start falling from the top. The color of the diamond will be random. The horizontal position of it will be random as well.
- If a diamond is missed, the game will be over. In this state, the falling diamond will vanish, no other diamonds will be falling from the screen, you won't be able to move the catcher, and the catcher will turn red instead of usual white. In the console, "Game Over" should be printed including your last score.
- **The speed of the diamond falling will gradually increase with time to ramp up the difficulty.**
- There will be 3 clickable buttons on the top of the screen (all drawn using midpoint lines):
    - A bright teal colored button on the left in the shape of a left arrow. Clicking this will restart the game (no matter if your game is over or not). Your score and diamond speed will also be reset. A new diamond will also start falling. You can show a text like "Starting Over" in the console. Don't forget to revert back the catcher's color to its usual one.
    - An amber colored button in the middle in the shape of a play or pause icon. Clicking this will toggle your game's playing/paused state. The icon will depend on the state as well. Which is, the pause icon shows when the game is in play state, and the play icon will show when it's the opposite. As you can guess, in the

paused state, the falling diamond will freeze, and you won't be able to move the catcher.

- A red colored button on the right in the shape of a cross. Clicking this will print "Goodbye" along with your score in the console, and terminate the application.

- **You have to draw everything on screen using the midpoint line drawing algorithm. With that being said, you're only allowed to use the `GL_POINTS` primitive type.**
- The size of the diamond won't be too big. It also shouldn't be too small that it's barely noticeable. Same thing for the catcher's length, it shouldn't be too long that it gives the player an unfair advantage. **The shape of the diamond and the catcher should be exactly what is shown in the figure- diamond with four midpoint lines and catcher with another four midpoint lines, maintaining the shape shown in the figure.**
- As the color of the diamonds are random, the colors should be bright enough so that it contrasts with the background.

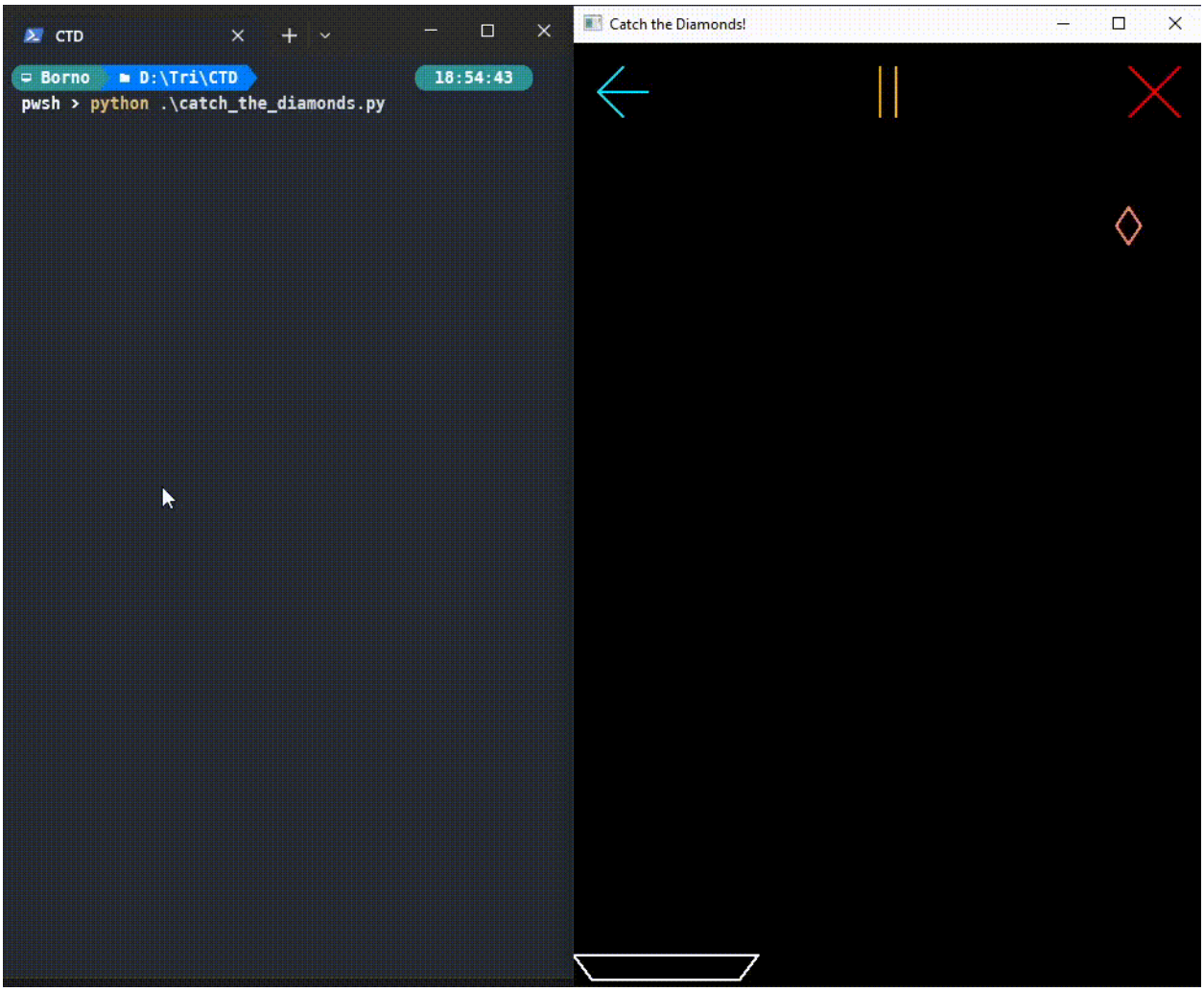


Fig: Catching diamonds, Basic Gameplay

Please Note: The above figure is an animated gif. If it doesn't play, you might want to open it in Google Docs instead of MS Word.

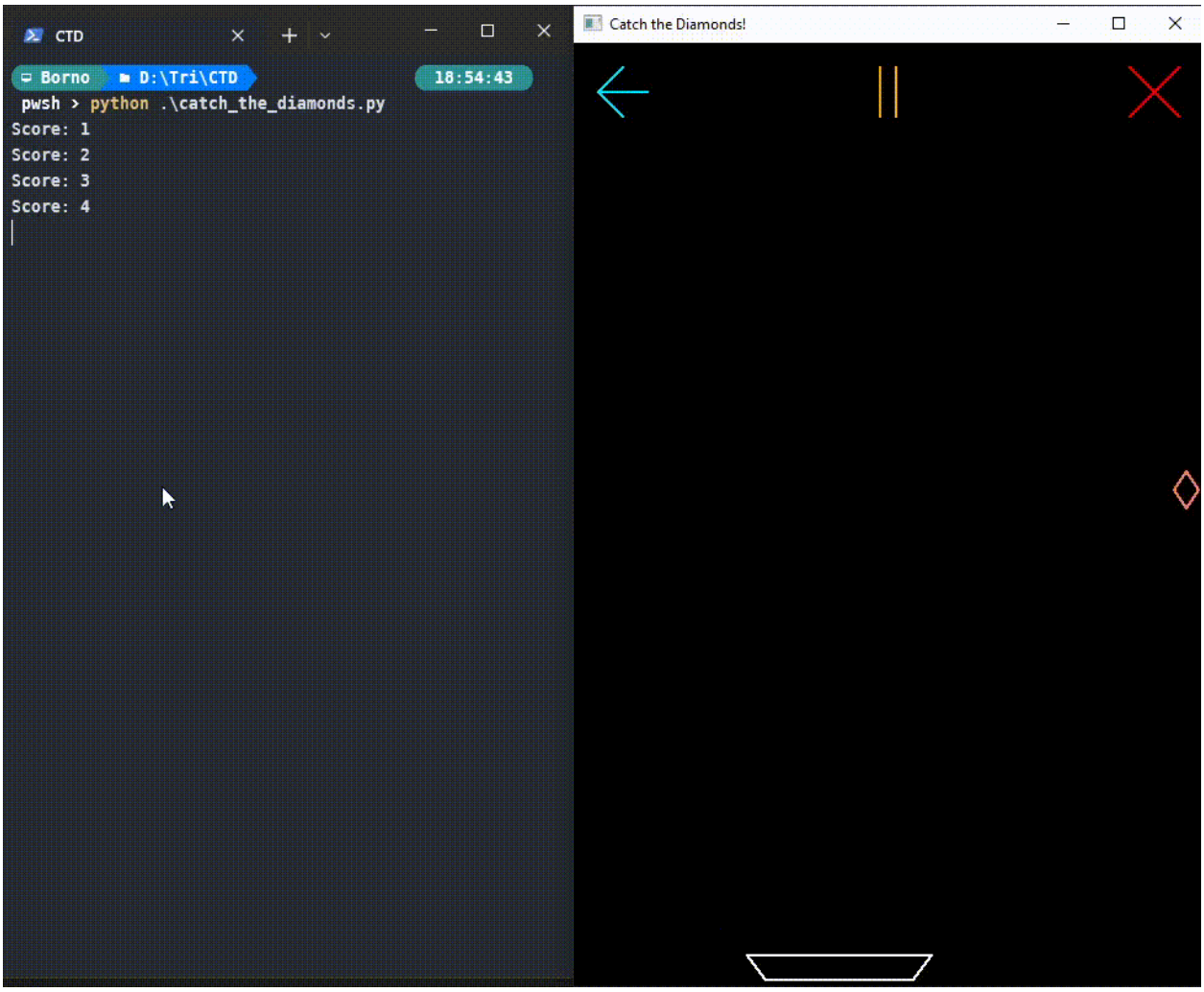


Fig: Game Over

Please Note: The above figure is an animated gif. If it doesn't play, you might want to open it in Google Docs instead of MS Word.

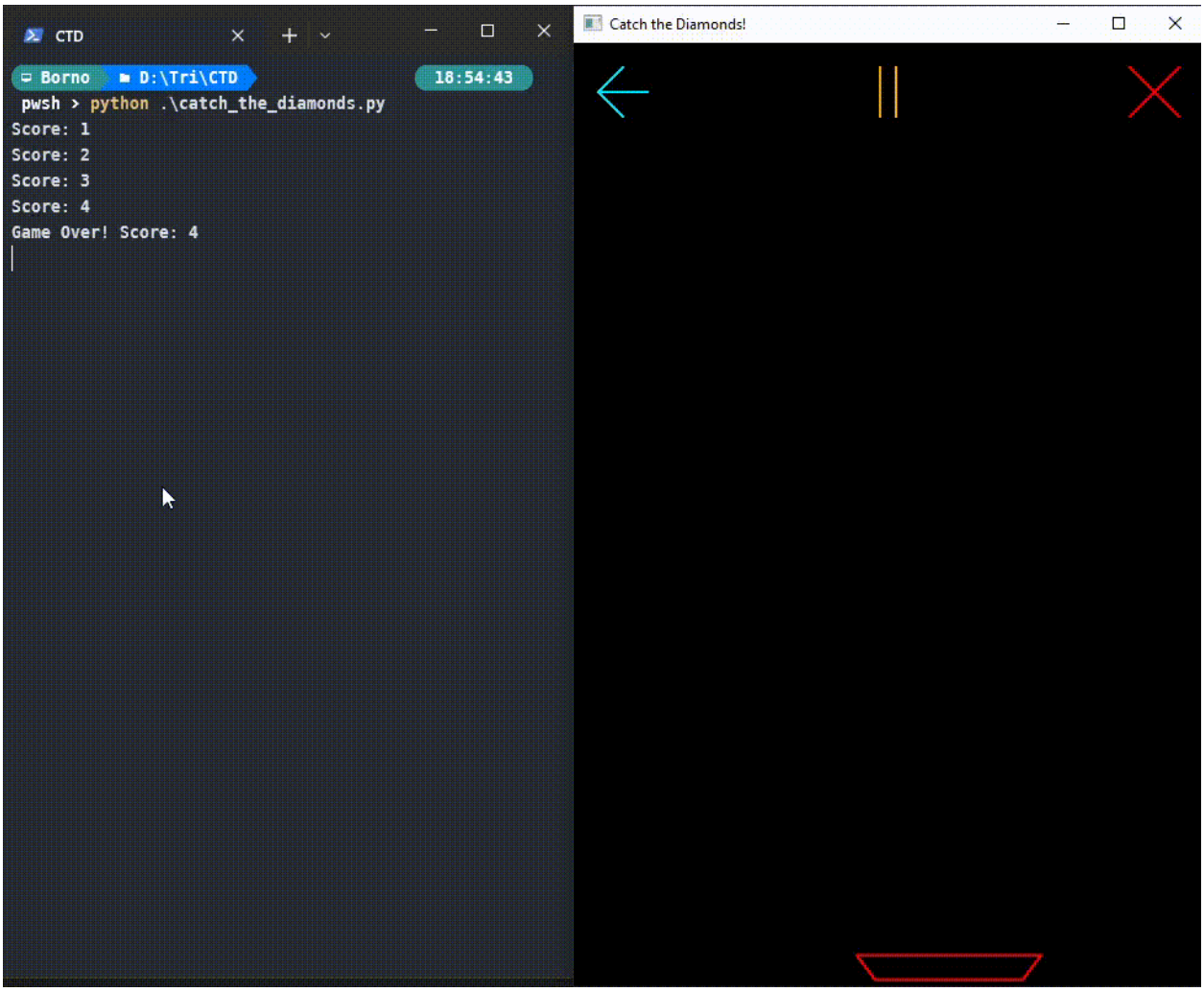


Fig: Starting Again (A.K.A. New Game)

Please Note: The above figure is an animated gif. If it doesn't play, you might want to open it in Google Docs instead of MS Word.

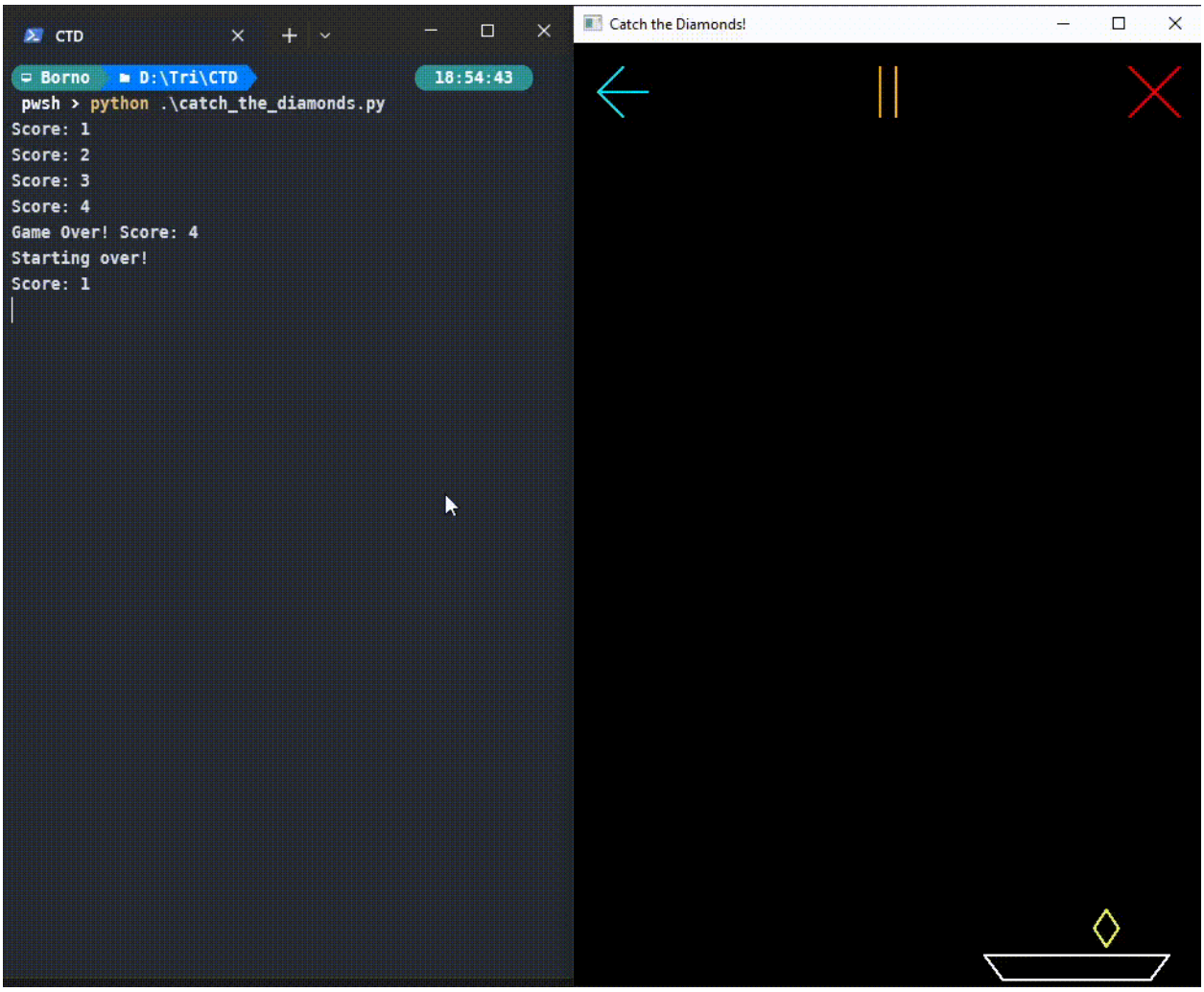


Fig: Play/Pause Toggle

Please Note: The above figure is an animated gif. If it doesn't play, you might want to open it in Google Docs instead of MS Word.

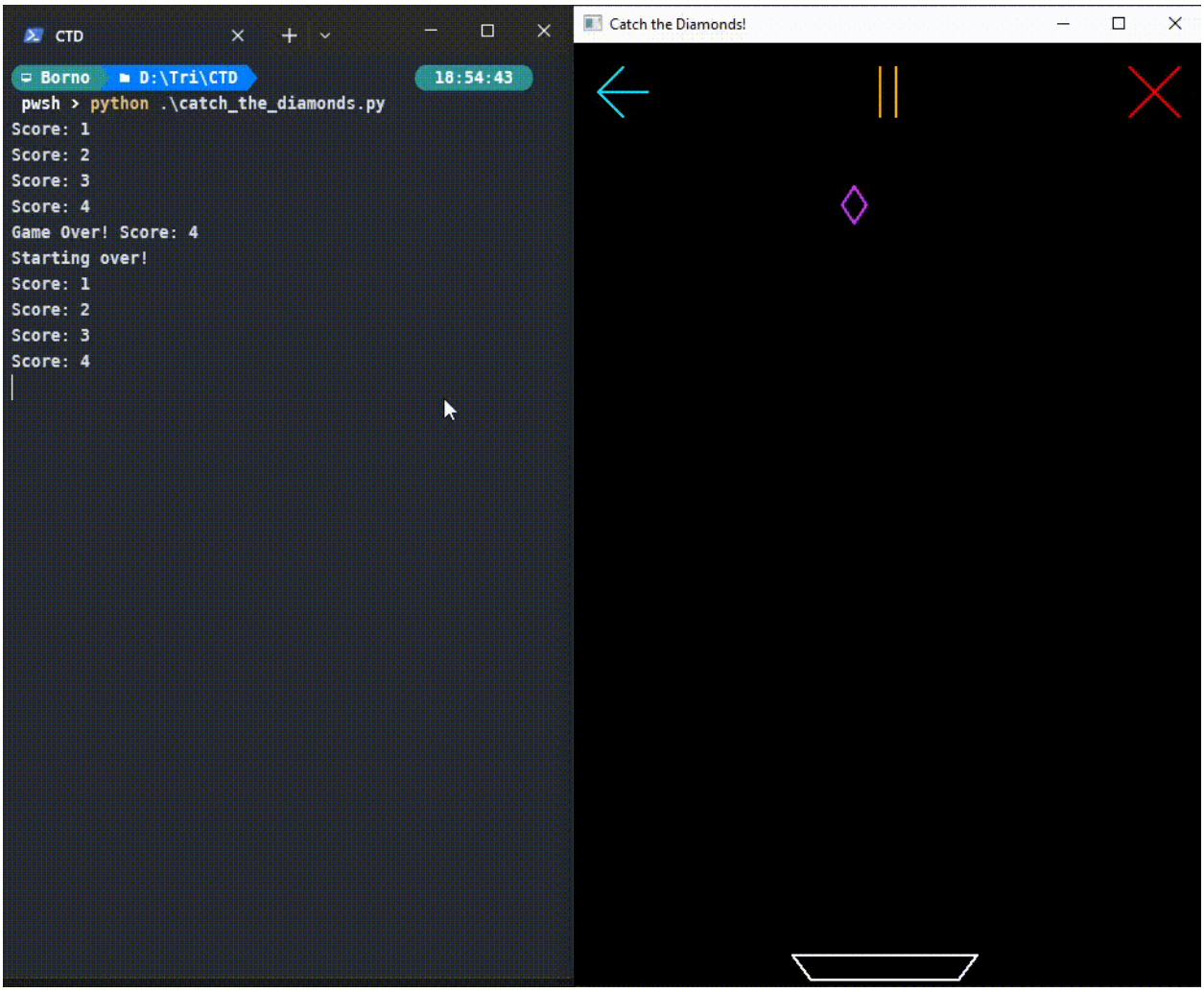


Fig: Terminating Application

Please Note: The above figure is an animated gif. If it doesn't play, you might want to open it in Google Docs instead of MS Word.

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 diamonds and catcher to have their own respective bounding boxes.

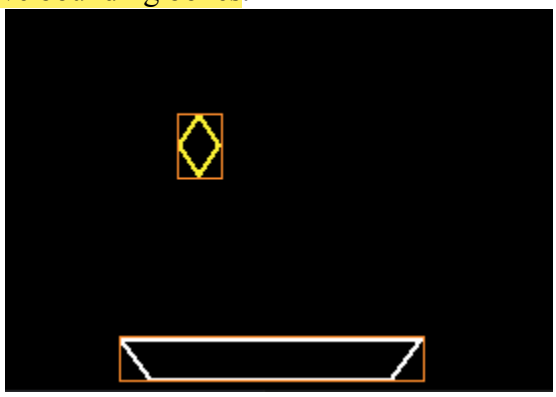


Figure: Objects showing their AABBs (shown as thin orange rectangles)

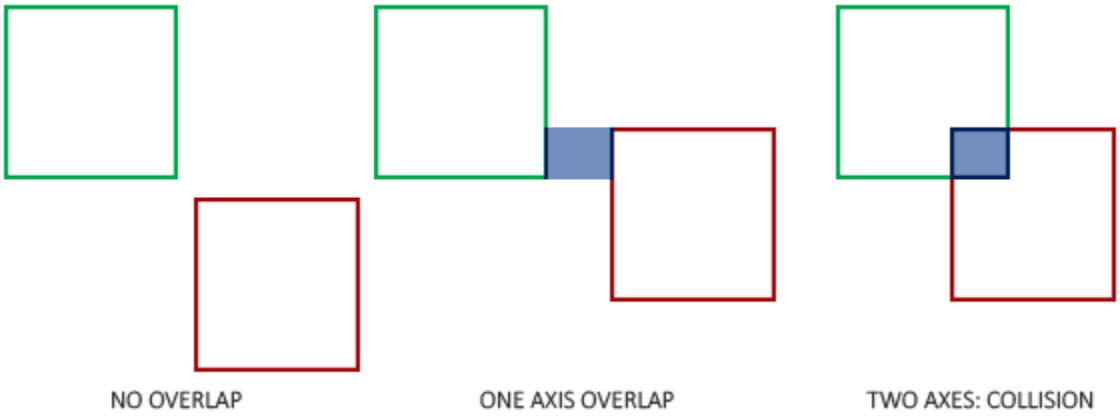


Figure: AABB collision (courtesy of learnopengl.com)

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 diamonds' acceleration and catcher'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 handy here. And for getting time as UNIX epoch, you can simply rely on Python's `time` module. Helpful Wikipedia article on delta timing here.