Maliak Green
COSC 490
Dr.Cone

# A.I. Final Paper: Coin Chaser

For my final project, I have decided to create a game where a monster agent chases the player that is controlled by the user. The player needs to collect all of the "coins" (yellow square objects) that appear on the screen. If the player can manage do to this, while avoiding the monster, then the game is over and they win. The first decision factory only had one class that made all the decisions for the previous maze agent to respond to. I have modified this to include two separate classes: one for the user, and one for the agent; changing the original to this model came with numerous new challenges. For one, I know had to find a way to get the player object to move based on the user's input. I solved this problem along with numerous others by referring to the pygame documentation page.

I found that I could detect user input by using the "key" object within Pygame's class (pygame.key). This object allowed me to control the up, down, left, and right keys. I used these four keys to pick up on the user's movement: making changes on to the GUI after every press. But with this new feature, I also had to paint to the screen after every press; which at first was trouble trying to work it in the main while-loop. After finally solving this problem, I enhanced the press-and-response system I created. I made it so that the user could hold down a key and get constant update to the GUI as opposed to repeatedly pressing the same key to get the desired output. I felt this was more practical and saved myself and potential players a lot of work.

For the monster agent's class, my main problem for the longest was coming up with a method for this agent to chase the user. I then had to learn about how to calculate new x and y coordinates based on the user's x and y coordinates. With a conditional internally built in, the monster's rect object is changed in place based on the player's decision. But then, after implementing this, without re-painting, the monster would basically leave a trail of rects

everywhere showing where it last was before the position update. Graphically, this is not pleasing to see and can cause confusion. Even though you can see multiple monsters on the screen, only one of them is carrying the up-to-date information given by its rect object. After trial-and-error time and time again, this was no longer the problem. I finally was able to get the monster to successfully close in on the user's position. Now the next step was to get the game to end when both the monster and player rect collided.

After viewing the pygame documentation page, I found a method that would solve my problem: rect.colliderect(). When used, this method determines if two rect objects within pygame collide. The only thing though was that originally, my decision factory file containing both the monster and player class did not have internal rects. I was painting rects to the screen based on the x and y coordinates from each class in main. I then had to make an internal rect for both classes. After doing this, it also seemed more practical to follow the OOP approach entirely and give both of the classes an internal draw function. After doing so, I could simply call player.draw() instead of painting hardcode to the screen in main. I then used this approach within my user class for detecting collision:

```
def detect_collision(self, monster, game_run):
    if self.rect.colliderect(monster) or monster.rect.colliderect(self):
        game_run = False
        return game_run
```

In the above code, self would be the user. This check determines if the two object rects have collided. As far as the user playing is concerned, the monster got you. So, after I implemented this feature to end the game, my next objective was to implement the coin object along with the wall object that I created. These two objects were my major enhancement to the original decision factory assignment. At first, my goal was to randomly print to the screen **N** coin objects specified by the user. The user would decide how many coins they wanted to have in the game run.

Maliak Green
COSC 490
Dr.Cone

Printing the coins to the screen at first was not very difficult, all I needed was to make sure they did not exceed the boundaries of the screen rect. This task became difficult when I wanted to also print to the screen my wall objects. In some runs that I tested, I would have coins being printed to the screen with the same position as my generated walls. This is because for my coin generation, I was using this method:

```
(x, y) = random.randrange(0, screen.rect.width)
coin.position = (x,y)
screen.blit(coin, coin.rect)
```

This code would run into a problem if any walls also happened to have the same randomly generated position. Even if they did not have the same exact position as any coin object, because these have rects, just spawning within 20 pixels of each-other would cause a graphical issue. To solve this problem, I created designated ranges for the coins to spawn on the screen along with the walls. This also goes for the user and monster. Before adding these ranges, I would have certain edge-cases where:

- as soon as the game would load, the monster would be within the user's rect; triggering the collide function and immediately ending the game.
- Coins would spawn within wall rects making them impossible to obtain to end the game winning.
- Walls would encompass an area on the screen and coins would spawn within this enclosed area with no way for the user to collect them.

After creating designated pixel ranges for each different type of object I made, this ensured the user would never see a coin that was unobtainable, and the user would have enough distance away from the monster to start the game.

These ranges solved my problem, but it made most of my game runs repetitive. My ultimate goal for spawning **N** coins was to have them spawn more evenly across the game screen. With these fixed ranges I incorporated, the user could expect to see the coins in almost the same areas every time. One solution io tried to implement was a for-loop that would loop through all of the positions of the generated walls. The tested tuple would be the coin object's position that was awaiting validation. If there were no positions of the walls that equaled the coin's position, I would then spawn the coin the code below shows my attempt:

```
for wall_pos in interior_walls_pos:
    while coin_pos == wall_pos:
        x = random.randrange(10, 380)
        y = random.randrange(10, 380)
        coin_pos = (x, y)
```

After implementing this, I got a more spread out and random coin spawn. However, I still had a problem with the coin objects spawning within the rects of the walls. The other problem I did not take into account was that I not only had to check for the positions of the walls, but how many pixels away they were from coin objects. For instance, Wall1 could have a position of (30, 180) and coin2 could have a position of (33, 181), but because these objects have rects that are bigger than the difference in positions, they will spawn within each other. I never figured out how to implement this correctly, so I commented out this solution and just kept the pre-defined ranges I implemented.

So, after I finally had my initial set up for the game coded, there were still some features I wanted to add in. The first feature was having a display at the bottom of the screen to show how many coins the player has gotten and how many coins the player had left to get. I knew what I wanted to say (i.e. coins gotten: x20) but I needed to find a way to print this message to the game

window. After reading on the documentation page, I found that I could turn the string into a pygame.font. after I converted the string to type font, I would then render the font onto the game screen by using screen.blit(); this feature was very straightforward and I encountered little difficulty. Another feature I wanted to add in my game were images. Up until this point, I had just been using squares for agent representation. I now wanted to have actual player textures incorporated into my game. To do this, I used pygame.image.load() to bring in the images that I wanted to use.

```
self.image = pygame.image.load("monster.png").convert()
self.rect = pygame.Rect( self.x, self.y, m_dim, m_dim  )
self.scaledMonster = pygame.transform.scale(self.image, (int(m_dim), int(m_dim)))
```

The code above is from my monster class in the decision factory file. The first two lines are how I set up the image and coordinates for the monster. The third line addressed a problem I was having early on: the images needed scaling. Originally, with no transform, the images would appear on the screen but very unproportionable. At one point, my user image took up most of the screen and I could not see the monster image. With the transform method, I was able to scale the image to be the size of the rects the objects had.  Doing this, I now had the images scaled proportionally and the images were contained within the game screen without going out of bounds.

The last features I mentioned above are really what make me feel like this is a game. Seeing a monster image move around and try to chase me reminds me of certain popular games like *pacman*; along with the coin objects I have that pay homage to the original coin-eater. I still feel that there were many problems I did not finish solving such as the randomly spawned coins,

but overall, I feel that I enhanced the decision factory to incorporate A.I decision making, while having the feel of a video game. My original goal was accomplished; I now wish to continue trying to make games in pygame. I learned a lot from the documentation page, and I feel if I were to create a second game as a personal project, it would be better; previous goals that I did not accomplish would be handled in this next game I would create. I look forward to continuing to make projects in pygame that will help me in the future for my programming portfolio.