# Algorithms:

## 1 get animation algorithm:

This algorithm is designed to load an animation by taking in the animation name and the amount of frames for each image of an animation. This algorithm assumes that each animation e.g. running is in a specified path: “rootAnimPath”, this is the path to a list of folders. Each of these folders contain the animations (images). The “animationName” is a string that will point to one of these folders by name within “rootAnimPath”. Hence the animations of running will be a list of images which is located at here: {rootAnimPath}/running. Each image within this path will be named specifically: {animationName}{animation image number} e.g. the first image of the running animation will be running0 the next will be running1. This is the easiest way of approaching this as software for drawing pixel art tend to save animations as gifs, but due to Pygame only taking in the png file format, software that supports conversion into to png will save made animations the way I described above e.g. running0.png, running1.png … running{n}.png where n is the amount of images in an animation minus 1. frameDurations will be an argument for this function as I know in advance how many images are in each animation, I use the array framedurations to store the number of frames each image should be allocated. This is useful for creating impact by making certain posses last longer. This also makes it easier to see what’s going on as if the player was given one frame for each image then it would be impossible to understand the movements. Furthermore, it slows down the animation to make it match the speed of the game.

## Pseudocode:

|  |  |  |
| --- | --- | --- |
| Given variable | animationName | FrameDurations |
| Type | string | Arr[int] |

animationFrameData <- empty arr

n = 0

FOREACH frame IN frameDurations DO:

animationFrameID <- animationName + string(n)

imageLocation <- rootAnimPath + “/” + animName + “/” + animationFrameId + “.png”

animationImage <- loadImage(imageLocation)

animationFrames[animationFrameID] <- animationImage

FOR I = 0 TO frame:

animationFrameData.append(animationFrameId)

ENDFOR

n++

ENDFOR

animationDatabase[animationName] = animationFrameData

## the Code:

01 def getAnimation(self,animName,frameDurations):

02        animationFrameData = []

03        n = 0

04        for frame in frameDurations:

05            animationFrameId = animName + str(n)

06            imgLoc=f"{self.rootAnimPath}/{animName}/{animationFrameId}.png"

07            animationImage = pygame.image.load(imgLoc).convert()

08            animationImage.set\_colorkey((0,0,0))

09            self.animationFrames[animationFrameId] = animationImage.copy()

10            for i in range(frame):

11                animationFrameData.append(animationFrameId)

12            n += 1

13       self.animationDatabase[animName] = animationFrameData

07 – pygame.image.load().covert() is a function that will load an image and return it on a pygame surface with a specified path. This is useful as I can easily display the surface on the screen with the pygame.blit method. I use the .convert() as this will convert the surface returned to the same pixel format as the one I use for the screen.

08 – animationImage.set\_colorkey((0,0,0)) set\_colorkey((0,0,0)) method takes all the colors that are given as an argument in this case black (R:0,G;0,B;0) and makes them transparent

## 2 Bezier curve sword slashes update method:

to create a clear sense of a sword slash I used the Bezier curve parametric equation:

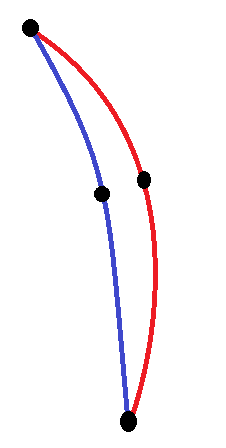
*Where a,b and c are coordinates and 0 < t < 1*

*In code form:*

x = (1-t)\*\*2\*self.start[0] + 2\*(1-t)\*t\*control[0] + t\*\*2\*self.end[0]

y = (1-t)\*\*2\*self.start[1] + 2\*(1-t)\*t\*control[1] + t\*\*2\*self.end[1]

This works by having 3 set points start, control and end and a t value. putting in t values will output coordinates to create a curve. So to generate the curve you need a range of t values from 0-1. The higher density t value will generate more points which will then generate a better curve. But due to the slow speed of for loops having very small t values will take long to generate all the points to make the curve so picking a good t density will be vital so the program runs smoothly. The arc is created into a slash by combining two Bezier arcs with the same start and end points with different control points to create a slash like effect as seen below:



Curve one and curve 2 is generated separately. They are then combined to form one array of points. This array can be put as an argument for the pygame.polygon function along with another value that fills in the inside to generate a sword like slash.

Start point

End point

Control point 2

Control point 1

While it is good to have just the arc, I need to be able to update it. Updating the arc includes drawing it on the screen, making it slowly fade, revealing from the top down or down to the top depending on the direction of the swing of the sword etc.

## pseudocode:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Given variable | Flip | firstPoints | secondaryPoints | Direction | Time |
| type | bool | Arr[float] | Arr[float] | enum{‘down’ or ‘up’} | int |

IF flip:

x <- X -32

ENDIF

neededFirstPoints <- points up to reveal amount

reverse(neededFirstPoints)

points <- secondaryPoints up to revealAmount + neededFirstPoints

IF direction = ’down’:

draw(False)

ELSE:

draw(true)

ENDIF

alpha <- 255\*Cosine(absoluteValue((self.time\*alphaVelocity)\* 3.14/180))

arcSurface.setAlpha(alpha)

time++

revealAmount <- revealAmount+revealSpeed

IF revealAmount > firstPoints.length()

revealAmount <- firstPoints.length()-1

ENDIF

arcIsDone <- arcSurface.getAlpha() = 0

return arcIsDone:

## Code:

01 def update(self,x,y,screen,scroll):

02 if self.flip:

03 x = x - 32

04 points = self.getPoints()

05 if self.direction == 'down':

06 self.draw(points,x,y,screen,scroll,False)

07 else:

08 self.draw(points,x,y,screen,scroll,True)

09 alpha = 255\*math.cos(abs(math.radians(self.time\*self.alphaVelocity)))

10 self.surf.set\_alpha(alpha)

11 self.time+=1

12 self.revealAmount=self.revealAmount+1+self.revealSpeed

13 if self.revealAmount > len(self.firstPoints):

14          self.revealAmount=len(self.firstPoints)-1

15 return self.surf.get\_alpha() == 0

16 def getPoints(self):

17    neededFirstPoints = self.firstPoints[:self.revealAmount]

18    neededFirstPoints.reverse()

19    points = self.secondaryPoints[:self.revealAmount]+ neededFirstPoints

20    return points

01 update function will run first takes in x y – position to draw curve screen-to draw the curve, scroll- so that it goes with the screen.

09 this line generates an alpha for the curve such that it fades away smoothly with a given speed: ‘alpha velocity’

17, 19: the arr[:n] is a faster way of getting the points up to ‘n’ If I were to use for loops the update would take to long and the game would lag every time I produced a arc.

# 3 world building:

To make the world I used a algorithm that many games in the genre. Due to my game having a side on approach instead of the top down approach it’s hard to come up with algorithms that generate worlds completely from scratch with maths alone that can be navigated completely every single time it’s used. This is mainly due to the idea that gravity takes a huge part and platforms need to be place in specific places so that the player can jump to them. This issue gets worse when trying to find places to put enemies and items if the game includes these. Most games will create specific rooms that can be navigated and then place them in ways that is randomized.

## Pseudocode:

Place a starting room

currentRoomPos = starting room

Rnd <- generate a random integer between 1 and 5

IF Rnd = 1 OR Rnd = 2:

place a room to the left of currentRoomPos

currentRoomPos <- currentRoomPos.Left

ENDIF

IF Rnd = 3 or Rnd = 4:

Place a room to the right of currentRoomPos

currentRoomPos <- currentROomPos.right

ENDIF

IF Rnd = 5:

Place a room underneath current room pos

IF currentRoomPos.RoomType <- RoomType1

currentRoomPos.RoomType <- RoomType2NoTop

ELSE:

IF currentRoomPos.RoomType <- RoomType2:

currentRoomPos.RoomType <- RoomType2Top

ENDIF

ENDIF

curretnRoomPos <- currentRoomPos.down

## Code:

Text

Description automatically generated

# Node placement algorithm:

For the a\* algorithm to work explained below it needs a graph parameter. This graph will be the world but due to the randomized nature of the world the nodes must also be procedurally placed to so that the enemy can find the best path to the player:

To achieve this we need to:

* Place nodes in the correct locations
* Place connections between nodes that the enemy can travel along

## Step one pseudocode:

|  |  |
| --- | --- |
| Given var | Mapdata |
| Type | Arr[,string] |

//add nodes

Nodes <- []

X <- 0

Y <- 0

Id <- 0

FOREACH row IN mapData:

nodeRow = []

FOREACH tile IN row:

IF tile <> 1:

nodeRow.append(null)

ENDIF

IF tile = 1:  
 nodeRow.append(NEWNODE(id, x\*16+8, y\*16))

Id++

ENDIF

x++

x<- 0

y<-1

nodes.append(nodeRow)

//remove unneeded nodes

FOREACH row IN nodes:

FOREACH node IN row:

IF node<> null:

IF aboveNode <> null:

ENDIF row[row.index(node)] = null

ENDIF

## Code:

nodes = []

        x = 0

        y = 0

        id = 0

        for row in mapdata:

            nodeRow = []

            for tile in row:

                if tile !=1:

                    nodeRow.append(None)

                if tile == 1:

                    nodeRow.append(Node(id,x\*16+8,y\*16))

                    id +=1

                x += 1

            x = 0

            y +=1

            nodes.append(nodeRow)

        #get rid of any nodes we don't need such as nodes that have nodes above them

        for row in nodes:

            for node in row:

                if node is not None:

                    if self.getRelativeStateOfNode(row.index(node),nodes.index(row),0,-1,nodes) is not None:

                        row[row.index(node)] = None

the getRelativeStateOfNode is a function I made. It takes an array, an x and y position of the array, and an x and y direction. The point of this function is to output the state of a node within the map. The map in this case will be a 2d array and look something like this:

nodes = [[0,0,0,0,0,0],

Each element represents a 16x16 location in the map hence to find the coordinate of a value inside this array find it’s x and y value in terms of the array where the highlighted 0 is the position (0,0) e.g. the highlighted p value will be at (2,1) and times both values by 16. So (32,16). For each element if the value stored is a 0 then this represents that in that location there is no objects such as a node but a p represents a pointer to a Node object.

[0,0,P,P,0,0],

[P,0,0,0,0,P],

[P,P,P,P,P,P]]

When it Comes to getting rid of unneeded nodes and finding possible connections it would be great to know the state of a node within a specified position relative to that node. This can be done with array logic but can get complicated especially when doing many of them such as in the next part of the node placement algorithm: making connections. Hence I created this function to abstract it so that I can put in the map array described above the position of the focused node inside the array and the translation across the array from that focused node e.g.:

getRelativeStateOfNode(2,1,-1,0,Nodes) -> Outputs: None

as:

the focused element is the highlighted p at (2,1) and I want to get the state of the node to the left of it or the vector (-1i + 0j) which is 0 hence no node. If I were to do:

getRelativeStateOfNode(2,1,1,0, Nodes) -> Outputs: pointer to Node Object:

as the element at (1i+0j) relative top at (2,1) is the P

## PheudoCode for getRelativeStateOfNode:

elementToFindXPos = focusedX + i

elementToFindYPos = focusedY + j

YLen <- mapData.length – 1

XLen <- mapData[elementToFindYpos].length – 1

If Ylen < elementToFindYPos or XLen < elementToFindXPos:

Return None:

Else:

row <- mapData[elementToFindYPos]

element <- row[elementToFindXPos]

return element

ENDIF

## Code:

def getRelativeStateOfNode(self,currentX,currentY,x,y,mapdata):#get the state of a node relative to the current node

    if len(mapdata)-1 < currentY+y or len(mapdata[currentY+y])-1 < currentX+x:

            return None

    else:

            return mapdata[currentY+y][currentX+x]

## Step two pseudocode:

//finding connections:

amountsOfRowsNodes <- nodes.length

FOR rowIndex <- 0 to amountOfRowsNodes DO:

row <- nodes[rowIndex]

amountOfNodesInRow <- row.Length

FOR nodeIndex <- 0 TO amountOfNodesInRow DO:

Node <-row[nodeIndex]

IF node <> Null:

IF (nodeIndex is not the end of the list and (there is a node to the left with no node above it)):

Connect(row[nodeIndex-1],row[nodeIndex])

ENDIF

IF (nodeIndex is not the end of the list and (there is a node to the right with no node above it)):

Connect(row[nodeIndex-1],row[nodeIndex])

ENDIF

IF (there is not a node above row[NodeIndex] or a node 2 places above that):

yConnectionRange <- 7

xConnectionRange <- 2

FOR y=0 TO yConnectionRange DO:

FOR x=-xConnectionRange TO xConnectionRange+1 DO:

IF absolute(x) = 0 OR absolute(x) = 1:

Return to start of last for loop

ENDIF

possibleConnection = node at (xi,yj) from row[nodeIndex]

IF possibleConnection <> Null:

IF (there is no node above the possibleConnection) AND ((x > 0 AND there is no node to the right of row[NodeIndex]) or (x < 0 AND there is no node to the left of row[NodeIndex])):

Connect(row[NodeIndex], possibleConnection)

ENDIF

ENDIF

ENDFOR

ENDFOR

ENDIF

ENDIF

ENDFOR

ENDFOR

## Code:

#connections

        amountOfRowsOfNodes = len(nodes)#get the amount of rows in nodes

        for rowIndex in range(0, amountOfRowsOfNodes):#loop through each row in nodes

            row = nodes[rowIndex]#get the row

            amountOfNodesInRow = len(row)#get the amount of nodes in the row

            for nodeIndex in range(0, amountOfNodesInRow):#loop through each node in the row

                node = row[nodeIndex]#get the node

                if node is not None:

                    '''<---make connections on the same y levels:--->'''

                    if (nodeIndex is not amountOfNodesInRow-1 and self.getRelativeStateOfNode(nodeIndex,rowIndex,1,0,nodes) is not None and self.getRelativeStateOfNode(nodeIndex,rowIndex,0,-1,nodes) is None):

                        node.add\_connection(row[nodeIndex+1],[1,0])

                    if nodeIndex is not 0 and self.getRelativeStateOfNode(nodeIndex,rowIndex,-1,0,nodes) is not None and self.getRelativeStateOfNode(nodeIndex,rowIndex,0,-1,nodes) is None:

                        node.add\_connection(row[nodeIndex-1],[1,0])

                    '''<---make connections on different y levels:--->'''

                    #check if there is a node above the current node by 1 and 2

                    if (self.getRelativeStateOfNode(nodeIndex,rowIndex,0,-1,nodes) is None and#is there no node above by 1

                        self.getRelativeStateOfNode(nodeIndex,rowIndex,0,-2,nodes) is None):#is there no node above by 2

                        #if so look for possible connections to make

                        ConnectionYRange = 7#the range of y values to check for nodes

                        ConnectionXRange = 2#the range of x values to check for nodes

                        for y in range(0,ConnectionYRange):#loop through the y range

                            for x in range(-ConnectionXRange,ConnectionXRange+1):#loop through the x range

                                if abs(x) == 0 or abs(x) == 1:continue

                                possibleConnection = self.getRelativeStateOfNode(nodeIndex,rowIndex,x,y,nodes)#get the node at the current x and y

                                if possibleConnection is not None:#if there is a possible connection

                                    #check for nodes that could obstuct the connection:

                                    if ((x > 0 and self.getRelativeStateOfNode(nodeIndex,rowIndex,x,y-1,nodes) is None and self.getRelativeStateOfNode(nodeIndex,rowIndex,1,0,nodes) is None) or

                                        (x < 0 and self.getRelativeStateOfNode(nodeIndex,rowIndex,-1,0,nodes) is None and self.getRelativeStateOfNode(nodeIndex,rowIndex,x,y-1,nodes) is None)):

                                            #make a connection

                                            node.add\_connection(possibleConnection,[x,y])

# A\* algorithm:

For the enemy to chase the users character I’m using the a\* algorithm. The hardest part of implementation is the speed: this algorithm has to be run many times to simulate chasing and if it’s too slow the game will suffer heavily from lag. Some of the ways I countered this was by using a priority queue. This was useful due to the point in the algorithm where I have to find the lowest f costs I did this by overriding the function \_\_lt\_\_ and \_\_gt\_\_ which are the less than and greater than functions that are run when comparing. This means with these functions I can control how the priority queue compares the nodes and I made it so the priority queue sorts based on the nodes f cost. Thus the lowest f cost will always be the next one to be pulled out of the queue.

The way I managed to code this algorithm was researching pseudocode while it is one specific algorithm there are many ways of coding it I found that at: <https://www.geeksforgeeks.org/a-search-algorithm/> had a good example that I could follow and coded based on this. Here is the pseudocode they provided:

## Pseudocode:

// A\* Search Algorithm

1. Initialize the open list

2. Initialize the closed list

put the starting node on the open

list (you can leave its **f** at zero)

3. while the open list is not empty

a) find the node with the least **f** on

the open list, call it "q"

b) pop q off the open list

c) generate q's 8 successors and set their

parents to q

d) for each successor

i) if successor is the goal, stop search

ii) else, compute both **g** and **h** for successor

successor.**g** = q.**g** + distance between

successor and q

successor.**h** = distance from goal to

successor (This can be done using many

ways, we will discuss three heuristics-

Manhattan, Diagonal and Euclidean

Heuristics)

successor.**f** = successor.**g** + successor.**h**

iii) if a node with the same position as

successor is in the OPEN list which has a

lower **f** than successor, skip this successor

iV) if a node with the same position as

successor is in the CLOSED list which has

a lower **f** than successor, skip this successor

otherwise, add the node to the open list

end (for loop)

e) push q on the closed list

end (while loop)

Code:

    def findPath(self,start,goal):

        #init the open list

        openList = PriorityQueue()#queue

        #init the closed list

        closedList = []

        #put the starting node on the open list

        openList.put(start)

        skipSuccessor = False

        #while the open list is not empty

        while not openList.empty():

            #find the node with the least f on the open list, call it "q" and pop q off the open list

            q = openList.get()

            #generate q's successors and set their parents to q

            successors = []

            gScores = []

            for connection in q.connections:

                successors.append(connection['node'])

                gScores.append(connection['g'])

            for successor in successors:

                #if successor is the goal, stop the search

                if successor.id == goal.id:

                    closedList.append(q)

                    path = []

                    path.append(closedList[0])

                    for i in range(1,len(closedList)):

                        connections = []

                        for connection in path[-1].connections:

                            connections.append(connection['node'])

                        if closedList[i] in connections:

                            path.append(closedList[i])

                    return path

                else:

                    #else, compute both g and h for successor

                    successor.setCosts(g=gScores[successors.index(successor)],end=goal,parent=q)

                for node in openList.queue:

                    if node.id == successor.id and node.f <= successor.f:

                        skipSuccessor = True

                        break

                if skipSuccessor:

                    skipSuccessor = False

                    continue

                for node in closedList:

                    if node.id == successor.id and node.f <= successor.f:

                        skipSuccessor = True

                        break

                if skipSuccessor:

                    skipSuccessor = False

                    continue

                openList.put(successor)

            closedList.append(q)