T0wer D3f3ence

Synopsis

Et billede, der indeholder skærmbillede, diagram, Grafiksoftware, grøn

Automatisk genereret beskrivelse

Dato: 15-09-2023

Ma’ad Ali Mohamud Abukar

Alex Daunoras Nielsen

Jonas Maul Hyldgaard

# Abstract

I denne opgave har vi skrevet om vores Tower Defence spil. I opgaven kommer vi blandt andet ind på, hvad selve spillet går ud på, og hvordan man spiller det samt med, at vi går i dybden med vores kode og forklarer hvad hvert kodestykke gør, og hvordan det gør det. Derudover finder man også vores krav til vores spil og til sidst er der en konklusion, vi ser på, om vores produkt, har opfyldte de krav, vi har sat.

# Problemformulering

Vi ønsker at lave et spil som kræver fokus, logisk tænkning og engagement af dets spillere. Spillet vil kunne bruges til at styrke spillerens evner indenfor de specifikke områder. Derudover vil spillet kunne bruges som inspiration til andre, for at de kan lave deres egne spil.

# Funktionsbeskrivelse

Et billede, der indeholder skærmbillede, diagram, Grafiksoftware, grøn

Automatisk genereret beskrivelse

Spillet er et ”Tower defence”-spil og det går ud på at man som bruger, ikke må lade fjenderne på skærmen komme hen til slutningen af stien. Som bruger er det muligt at købe og placere tårne ude fra menuen til højre. Tårnene kan så placeres på banen og de skader fjender som kommer ind i deres radius. Man kan starte den næste runde ved at klikke på den grønne knap nede til højre i menuen. Man får penge hvis man dræber fjender og man mister penge hvis man køber et tårn. Man mister også liv hvis man lader en fjende komme hen til slutningen af stien.

**Krav**

|  |
| --- |
| Der skal være en sti som nogle fjender skal kunne følge til slutningen |
| Der skal være tårne som skal kunne placeres ved at brugeren trækker dets ikon ude til højre ind på banen og de skal kun kunne placeres de steder hvor det er meningen |
| Tårne skal kunne skyde på den forreste fjende i deres radius og fjenderne skal tage skade af dette |
| Der skal være liv som skal ændre sig hvis en fjende kommer til slutningen af stien  Der skal være penge som skal ændre sig hvis en fjende dør eller man køber et tårn |
| Der skal være mere end 1 slags fjende og 1 slags tårn |
| Man skal kunne se tårnenes rækkevidde når man placerer dem og man skal kunne se hvert tårns pris |
| Der skal være flere runder som bliver sværere og sværere |

# Programbeskrivelse

Vores kode er spredt ud på flere filer, så den bliver mere overskuelig, men også så det er muligt at være flere der arbejder på samme tid i koden, da man i VSCode ikke kan være flere der skriver i en fil på samme tid. Vores hovedfil hedder Game.py. I Game.py og mange andre af vores filer bruger vi PyGame til at lave en skærm, håndtere sprites og inputs mm. Meget af koden i Game.py kører i et while running: loop, som gentager 60 gange i sekundet. Vi bruger pygame.time.Clock().tick(60) til at holde styr på vores framerate.

Et sted i vores main loop har vi et stykke kode der ser sådan ud:

#Tegner alt der skal tegnes

        Draw.DrawScreen(screen)

        eg.enemies.draw(screen)

        for enemy in eg.enemies:

            enemy.DrawHealth(screen)

        eg.towers.draw(screen)

        screen.blit(playButton.image, playButton.rect)

        DrawText(str(pv.health) + " <3", "gray100", 1050, 650, screen)

        DrawText(str(pv.money) + " $", "gray100", 1050, 680, screen)

        SmallText("100$", (255,255,255), 1040, 50, screen)

        SmallText("200$", (255,255,255), 1040, 150, screen)

        SmallText("300$", (255,255,255), 1040, 250, screen)

        SmallText("400$", (255,255,255), 1040, 350, screen)

Det er en samling af næsten alle funktioner der tegner sprites og figurer i vores kode. Til at starte med kaldes DrawScreen() fra Draw.py-filen, så baggrunden er på plads. Så tegnes alle tårne, fjender og deres liv. Tårne og fjender ligger i spritegrupper i filen EveryGroup.py kaldet eg. Derefter placeres næste runde-knappen, og alt tekst på skærmen (penge, liv og priser på tårne). Vores DrawText() og SmallText()-funktioner er næsten ens. Den eneste forskel er at de bruger en anden font. eg.towers.draw() og eg.enemies.draw() er en indbygget funktion i pygame til spritegrupper. DrawScreen()-funktionen kan findes i Draw.py-filen. Den ser sådan ud:

roadSegments = [

    #Linjer

    pygame.Rect(0, 50 - width/2, 400, width),

    pygame.Rect(400 - width/2, 50, width, 100),

    pygame.Rect(400, 150 - width/2, 100, width),

    pygame.Rect(500 - width/2, 50, width, 100),

    # (...)

    # Hjørner

    pygame.Rect(400 - width/2, 50 - width/2, width, width),

    pygame.Rect(400 - width/2, 150 - width/2, width, width),

    pygame.Rect(500 - width/2, 50 - width/2, width, width),

    # (...)

    ]

def DrawScreen(screen):

    screen.fill((76, 187, 23))

    for roadSegment in roadSegments:

        pygame.draw.rect(screen, roadColor, roadSegment)

    #Menu

    pygame.draw.rect(screen, menuColor, pygame.Rect(1000, 0, 400, 700))

Her har vi fjernet en del af vejsegmenterne for at spare plads i eksemplet. RoadSegments er en liste over en masse rektangler som vores vej består af. Grunden til at vi har valgt at bruge hjørner i stedet for at gøre ”linjerne” længere, er at det skaber mere fleksibilitet i vores kode. Selvom det ville se ud på samme måde hvis linjerne var længere, og overlappede hinanden i hjørnerne, ville man ikke kunne ændre bredden af vejen uden at skulle ændre en masse koordinater. Hvis man bruger hjørner, kan man gøre vejen tykkere og tyndere uden problemer. Vi har valgt at hardcode placeringerne og længderne af vejsegmenterne fordi det ville kræve alt for meget arbejde og tid, og alt for mange variable ikke at gøre det. Vores DrawScreen-funktion tegner hele baggrunden ved først at lave grønt græs, så tegner den alle vejsegmenterne, og til sidst tegner den menuen hvor man kan købe tårne

Vores DrawText()-funktion skal bruge en font til teksten. Den defineres sådan gennem pygame:

textFont = pygame.font.SysFont("Comic Sans", 24, bold = True)

DrawText() ser sådan ud:

    def DrawText(text, textCol, x, y, screen):

        textSurface = textFont.render(text, True, textCol)

        textRect = textSurface.get\_rect(center = (x, y))

        screen.blit(textSurface, textRect)

DrawText() har 5 inputs som den bruger til at tegne tekst. Den laver en surface med tekst, der har den rigtige font, på, og placerer den i de angivne koordinater. Derefter blitter den surfacen på skærmen.

I vores classes.py-fil har vi bla. en klasse der hedder enemy. Den fungerer som parent-klasse til de forskellige fjendetyper. Den ser sådan ud:

class enemy(pygame.sprite.Sprite):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        self.width = 40

        self.height = 40

        self.totalDistance = 0

        pygame.draw.rect(self.image, (200, 200, 200),

pygame.Rect(0, 50, self.width, self.height))

        self.rect = self.image.get\_rect()

enemy-klassen nedarver som alle vores andre sprite-klasser pygame.sprite.Sprite, hvilket gør at vi kan arbejde med dem som sprites i PyGame. I \_\_init\_\_()-funktionen kalder vi super().\_\_init\_\_() for at få alle funktionaliteter fra pygame.sprite.Sprite. Vi giver den længde og bredde så alle fjender har samme størrelse medmindre vi eksplicit skriver at en fjende ikke skal. Vi tegner spriten og giver den en rect. Bemærk at self.image ikke er defineret endnu. Det bliver den i child-klasserne før super().\_\_init\_\_() bliver kaldt. self.totalDistance bliver brugt når fjenden skal bevæge sig. Det sker i move()-metoden som enemy-klassen også har. Den ser sådan ud:

def Move(self):

        if self.totalDistance >= 0 and self.totalDistance < 400:

            self.rect.y = 50 - self.height/2

            self.rect.x = self.totalDistance - self.width/2

        if self.totalDistance >= 400 and self.totalDistance < 500:

            self.rect.y = self.totalDistance - 400 + 50 - self.height/2

            self.rect.x = 400 - self.width/2

        if self.totalDistance >= 500 and self.totalDistance < 600:

            self.rect.y = 150 - self.height/2

            self.rect.x = self.totalDistance - 500 + 400 - self.width/2

        # (...)

Her vil den enkelte sprite placere sig i det stykke på banen hvor den skal være alt efter dens totalDistance-variabel. totalDistance måles i pixels. Hvis spriten har bevæget sig mellem fx 400 og 500 pixels, ligger den på et lodret vejstykke hvis x-værdi er 400. y-værdien af spriten bestemmes i dette tilfælde ved at tage dens totalDistance minus ved hvilken totalDistance vejen starter (400). På den måde er y indtil videre 0 hvis totaldistance er 400. så lægges 50 til, som er den y-værdi hvor vejen starter. På den måde er y 50, hvis den er i starten af vejstykket. Til sidst trækkes spritens halve højde fra for at centrere. Grunden til at vi fastlåser spritens placering via totaldistance-variablen, er at vi er sikre på at spriten altid er på banen. Havde vi haft en funktion der fik spriten til at flytte til højre så længe den var på et bestemt vejstykke, ville vi kunne komme ud for en situation hvor spriten mangler 1 pixel for at gennemføre vejstykket, men dens speed-variabel siger at den skal flytte 10 pixels pr. frame. Så ville den have rykket et stykke uden for banen, hvilket ikke ville se pænt ud. Tårne kan også bruge variablen når de skal skyde på den fjende der er længst fremme på banen.

Enemy-klassen har også en DrawHealth-funktion der ser sådan ud:

def DrawHealth(self, screen):

        pygame.draw.rect(screen, (0, 255, 0), pygame.Rect(self.rect.x, self.rect.y - 20, 40, 10))

        pygame.draw.rect(screen, (255, 0, 0), pygame.Rect(self.rect.x, self.rect.y - 20, 40 - 40 \* (self.health/self.maxhealth), 10))

DrawHealth() tager kun et input, og det er hvilken skærm den skal tegne livindikatoren på. Den fungerer ved at der bliver lavet to firkanter. Først en grøn firkant over spriten. Derefter en rød firkant ovenpå den grønne, som bliver bredere, jo mere skade spriten har taget. På den måde vil livindikatoren blive mere rød hvis spriten har lidt liv, og mere grøn hvis den har meget.

Enemy-klassen er en parent-class for BasicEnemy() og StrongerEnemy(). Den eneste forskel mellem de to er forskellig liv, billede, fart og lignende. Altså ingen funktionel forskel. BasicEnemy() ser sådan ud:

class BasicEnemy(enemy):

    def \_\_init\_\_(self):

        self.image = pygame.image.load("Sprites/BasicEnemy.png")

        self.maxhealth = 3

        self.health = self.maxhealth

        self.moveSpeed = 3

        self.moneydrop = 10

        super().\_\_init\_\_()

        self.rect.x = 0

        self.rect.y = 50

        self.damage = 15

I BasicEnemy() definerer vi først de attributter som ikke kræver at klassen er en sprite. Derefter kører vi super().\_\_init\_\_(), som bl.a. gør BasicEnemy til en sprite. Bagefter definerer vi dens rect.x og rect.y, som begge er afhængige af dens sprite-status.

Helt grundlæggende for et ”Tower defence” spil, er tårnene. Vi har valgt at definere tårnene ved brug af klasser og subklasser.

class Tower(pygame.sprite.Sprite):

    def \_\_init\_\_(self, x, y):

        super().\_\_init\_\_()

        self.cost = 100

        self.damage = 0

        self.placed = False

        self.speed = 0

        self.lastshot = 0

        self.x = x

        self.y = y

        self.rect = self.image.get\_rect(center=(x, y))

Klassen “Tower” er den grundlæggende klasse som alle andre tårne benytter sig af for at nedarve de mest basale attributter. Disse attributter inkluderer blandt andet pris, skade, positionen og hvornår tårnet sidst har skudt.

Tower-klassen har to metoder: Shoot() og DrawRange(). Shoot-funktionen bruges til at finde et mål, og skyde på den. Den ser sådan ud:

def Shoot(self):

        self.furthestDistance = 0

        self.target = None

        for enemy in eg.enemies:

            if ((enemy.rect.x - enemy.width/2) - (self.rect.x - self.image.get\_width()/2))\*\*2 + ((enemy.rect.y - enemy.width/2) - (self.rect.y - self.image.get\_height()/2))\*\*2 <= self.range\*\*2 and enemy.totalDistance > self.furthestDistance:

                self.furthestDistance = enemy.totalDistance

                self.target = enemy

        if self.target != None:

            self.target.health -= self.damage

Til at starte med, defineres at der ikke er noget target, og at furthestDistance er 0. Det bruges i næste for-loop, hvor der for hver enemy tjekkes om (fjende\_x - tårnx)2 + (fjende\_y - tårn\_y)2 er mindre end rækkevidden i anden. Altså om fjenden er inden for rækkevidde.  
Grunden til at vi tjekker og ikke , er fordi hvert tårn tjekker hver fjende hver frame, hvilket er mange kvadratrødder, som vil gøre spillet langsommere. Ud over at tjekke om fjenden er indenfor rækkevidde, tjekkes der også om fjendens totalDistance er større end den hidtil største totalDistance fundet. På den måde vælges kun den fjende der er længst fremme på banen. Hvis der til sidst er blevet valgt en fjende, vil den fjende tage så meget skade som tårnet gør. Hvis vi ikke tjekkede om der rent faktisk var blevet valgt en fjende, ville der ske en fejl hvis tårnet ikke havde noget at skyde på.

DrawRange()-funktionen tegner en halvgennemsigtig cirkel omkring tårnet, der illustrerer dens rækkevidde.

def DrawRange(self, screen):

        transparency = 128

        circle\_surface = pygame.Surface((self.range \* 2, self.range \* 2), pygame.SRCALPHA)

        pygame.draw.circle(circle\_surface, (100, 100, 100, transparency), (self.range, self.range), self.range)

        screen.blit(circle\_surface, (self.rect.x - self.range + self.image.get\_width()/2, self.rect.y - self.range + self.image.get\_height()/2))

Her laves en surface hvor cirklen skal være. Herefter tegnes en cirkel med radius størrelse med tårnets range. Til sidst blittes surfacen på skærmen, så cirklen er synlig.

class ArcherTower(Tower):

    def \_\_init\_\_(self, x, y, default\_x, default\_y):

        self.image = pygame.image.load(r"Sprites/ArcherTower.png")

        super().\_\_init\_\_(x, y)

        self.damage = 1

        self.range = 150

        self.cost = 100

        self.speed = 1200

        self.default\_x = default\_x - self.image.get\_width()/2

        self.default\_y = default\_y - self.image.get\_height()/2

Her ses et eksempel på et tårn. Tårnet har de samme attributter som grundklassen ”Tower” havde. I denne klasse som hedder ”ArcherTower”, er attributterne blevet tildelt en anden værdi. For eksempel har den fået dedikeret en skade, en rækkevidde og en x- og y-værdi som er defineret ud fra halvdelen af tårnets bredde og højde, så dens x- og y-værdi er altså i midten af tårnet. Dette tårn er også gennem super().\_\_init\_\_() blevet en sprite, så den får en masse pygame-funktioner.

Det er disse forskelle mellem grundklassen ”Tower” og subklassen ”ArcherTower” som også gør sig gældende for alle de andre tårnklasser. IceTower() har dog en lille ekstra forskel i shoot()-metoden, hvilket er at den fjende der skydes på får mindsket sin fart med 10% hver gang den skydes.

if event.type == pygame.MOUSEBUTTONDOWN:

                #Hvis der klikkes på et tårn, skal der tages fat i tårnet

                for tower in eg.towers:

                    if tower.rect.collidepoint(event.pos) and tower.placed == False and pv.money >= tower.cost and dragging == False:

                        pv.money -= tower.cost

                        dragging = True

                        chosenTower = tower

Dette stykke kode ligger i game.py, og håndterer placeringen af tårne. Der bliver først tjekket efter om venstreklik på musen er trykket ned og bagefter tjekker den om musen er placeret over tårnet, om man har penge nok til at købe tårnet, om man allerede har placeret tårnet og om man allerede er i gang med at trække i et tårn.

Hvis man får lov til at købe tårnet, mister man det antal penge som tårnet kostede, dragging bliver sat til True og det tårn som man er i gang med at trække i, bliver defineret som ”chosenTower”.

Hvis man er i gang med at trække i et tårn og man slipper venstrekilk igen, så sker følgende:

if event.type == pygame.MOUSEBUTTONUP and dragging:

                #Tjekker om det valgte tårn rører ved vej eller andre tårne.

                touchingSomething = False

                towersTouched = -1

                for roadSegment in Draw.roadSegments:

                    if chosenTower.rect.colliderect(roadSegment):

                        touchingSomething = True

                for tower in eg.towers:

                    if chosenTower.rect.colliderect(tower):

                        towersTouched += 1

                if towersTouched > 0:

                    touchingSomething = True

                if touchingSomething == False:

                    if mouseX <= 980 and not chosenTower.placed:

                        dragging = False

                        chosenTower.placed = True

                    else:

                        #Tilbage til standardposition

                        chosenTower.rect.topleft = (chosenTower.default\_x, chosenTower.default\_y)

                        pv.money += chosenTower.cost

                        dragging = False

                else:

                    #Tilbage til standardposition

                    chosenTower.rect.topleft = (chosenTower.default\_x, chosenTower.default\_y)

                    pv.money += chosenTower.cost

                    dragging = False

Det første der sker, er at variablen ”touchingSomething” bliver sat til False, da tårnet går ud fra at det ikke rører ved noget. Derefter bliver variablen ”towersTouched” sat til -1, da den tæller sig selv med, når den tjekker om den rører ved et tårn. På den måde vil towersTouched som standard være 0. Herefter tjekkes der om det tårn man er i gang med at placere, rører ved et stykke af stien. Hvis tårnet kolliderer med stien, bliver variablen ” touchingSomething” sat til True og man kan ikke placere tårnet. Bagefter bliver der tjekket om det tårn man vil placere, rører ved et andet tårn, hvis det gør, bliver ”towersTouched” en større, derefter tjekkes der om ”towersTouched” er mere end 0, hvis ja, så bliver ”touchingSomething” sat lig med True og tårnet kan ikke placeres.

Efter dette tjekkes der om ”touchingSomething” er lig med False. Hvis ja, tjekkes der om musens placering er ovre i menuen. Hvis musen er i menuen og man slipper venstreklik, så returnerer tårnet til dets startposition i menuen og man får pengene tilbage som man brugte på at købe tårnet. Hvis musen ikke er i menuen og tårnet ikke rører noget, så bliver tårnet placeret og ”dragging” bliver sat til False og ”chosenTower.placed” bliver sat til True, hvilket betyder at tårnet nu er placeret og man kan ikke rykke med det igen.

Det sidste stykke af koden kører hvis man slipper tårnet, enten over stien eller et andet tårn, da vil det blive returneret til dets startposition i menuen, ”dragging” vil blive sat til False og man vil få pengene tilbage som man brugte på at købe tårnet.

Siden vores spil er et Tower defence-spil, vil der være forskellige runder med forskellige fjender. Derfor har vi inkluderet en startknap, som brugeren skal trykke på, for at den næste runder vil begynde.

class PlayButton(pygame.sprite.Sprite):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        self.image = pygame.image.load("Sprites/PlayButton.png")

        self.rect = self.image.get\_rect()

        self.rect.center = (1200, 660)

Playbutton-klassen er en child-klasse, som nedarver fra Pygame.sprite-klassen. Dermed har den en Parent-child relation med sprite-klassen. Klassen har også attributter som self.image, der gør at klassen vil få et billede af en startknap. self.rect.center er hvor den vil placeres i spillet.

Koden der tjekker om man trykker på startknappen er:

if event.type == pygame.MOUSEBUTTONDOWN:

               if playButton.rect.collidepoint(event.pos) and pv.roundRunning == False and pv.currentRound < len(waves) - 1:

                    pv.currentRound += 1

                    pv.currentEnemySpawn = 0

Først tjekker vi om musen rører knappen, om der er en runde der kører, og om der er flere runder tilbage. Areal i denne tilfælde er så, indenfor startknap. Hvis en ny runde må starte gør den rundenummeret 1 større, og nulstiller currentEnemySpawn. Den måde rundesystemet virker er gennem Spawn()-funktion. Den kører en gang hver frame, og spawner en fjende hvis det er tid til det. Den er baseret på en liste ved navn waves der ser sådan ud:

waves = [[0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 1, 1, 0, 0],  
[1, 1, 0, 0, 0, 1, 1, 1], [1, 1, 1, 1, 1, 0, 0, 1], [1, 1, 1, 1, 1, 1, 1, 1]]

Listen består af fem lister der hver især repræsenterer en runde i spillet. Hver runde består af otte tal der repræsenter hvilke fjender der er i runden, og deres rækkefølge. Et nul betyder en svag fjende og et ettal betyder en stærk fjende.

Spawn() ligger i SpawnEnemy.py-filen, og ser sådan ud:

def Spawn(round, roundList, enemiesList):

    #Er alle fjender allerede spawnet, og kører der allerede en runde?

    if pv.currentEnemySpawn < len(roundList[round]):

        pv.roundRunning = True

        #Finder ud af hvilken fjendetype der skal spawnes

        if roundList[round][pv.currentEnemySpawn] == 0:

            enemy = Classes.BasicEnemy()

        else: #round\_list[round][currentSpawn] == 1:

            enemy = Classes.StrongerEnemy()

        currentTime = pygame.time.get\_ticks()

        if currentTime - pv.enemySpawnDelay > pv.lastEnemySpawnTime:

            pv.lastEnemySpawnTime = pygame.time.get\_ticks()

            enemiesList.add(enemy)

            pv.currentEnemySpawn += 1

            if pv.currentEnemySpawn < len(roundList):

                #Spawn delay for hver enemy type

                if roundList[round][pv.currentEnemySpawn + 1] == 0:

                    pv.enemySpawnDelay = random.randint(700, 1300)

                elif roundList[round][pv.currentEnemySpawn + 1] == 1:

                    pv.enemySpawnDelay = random.randint(1100, 1900)

    else:

        pv.roundRunning = False

Den måde Spawn() fungerer er at den først tjekker om variablen currentEnemySpawn er mindre end antallet af fjender i nuværende runde. currentEnemySpawn holder styr på hvilket nr. fjende det er tid til at spawne. currentEnemySpawn ligger i filen PublicVariables.py, kaldet pv. Grunden til at vi bruger en anden fil til at holde variablen, er at den både skal kunne redigeres i Game.py og SpawnEnemy.py. Havde vi defineret variablen i Game.py, ville vi være nødt til at have en cirkulær afhængighed af Game.py og SpawnEnemy.py, da de ville være nødt til at importere hinanden. Havde vi gemt variablen i SpawnEnemy.py, ville dens værdi ikke blive gemt mellem frames. Derfor brugte vi PublicVariables.py. Hvis currentEnemySpawn er større end antallet af fjender i runden, og der altså ikke er flere fjender at spawne, sætter den roundRunning til False. Det gør det muligt at starte en ny runde. Hvis der stadig skal spawnes fjender, tjekker den om den fjende den er nået til at spawne i runden er en stærk eller svag fjende. I begge tilfælde bindes den nye fjende til enemy-variablen. Bagefter tjekker den om der er gået nok tid siden sidste fjende spawnede. Hvis der er det. Definerer den lastEnemySpawnTime til den nuværende tid, og tilføjer enemy til den sprite-gruppe den skal være i. Efterfølgende burde have tjekket for om den var den sidste fjende i rækken, men vi skrev forkert. I if-sætningen burde der have stået

if pv.currentEnemySpawn < len(roundList[round]):

I stedet star der

if pv.currentEnemySpawn < len(roundList):

Dette er ikke en fejl der ødelægger spillet, men det vil skabe en adfærd der ikke er tiltænkt. Den burde have tjekket om der var flere fjender at spawne, og hvis der er var det, ville den gøre spawnDelay til en tilfældig værdi i et interval baseret på fjendetypen, så der ville være en større pause efter en stærk fjende, er spawnet.

# Test

Mens vi arbejdede med programmet og skrev koden, testede vi løbende. Det gjorde vi for at sikre os, at hvad end vi lige havde lavet fungerede. Det betød også at vi nemmere kunne gå videre med koden, fordi vi vidste at alt fungerede som det skulle, og at hvis der skulle opstå nogle fejl, så var det sandsynligvis pga. den nye kode som vi lige havde skrevet.

|  |  |  |
| --- | --- | --- |
| **Situation** | **Forventet Resultat** | **Faktiske Resultat** |
| Trække i et tårn-ikon og slippe det over banen | Tårnet følger med musen og bliver placeret på banen | Tårnet følger med musen og bliver placeret på banen |
| Trække i et tårn-ikon og slippe det over stien, over et andet tårn eller i menuen | Tårnet følger med musen og bliver ikke placeret, men returnerer til sin oprindelige position i menuen | Tårnet følger med musen og bliver ikke placeret, men returnerer til sin oprindelige position i menuen |
| Klikke på den grønne knap i menuen for at starte den næste runde | En ny runde starter hvis der ikke er fjender som er i gang med at spawne og der er flere runder tilbage | En ny runde starter hvis der ikke er fjender som er i gang med at spawne og der er flere runder tilbage |
| En fjende spawner | Den følger stien helt til slutningen | Den følger stien helt til slutningen |
| Brugeren køber et tårn | Man mister penge alt efter hvor meget tårnet kostede | Man mister penge alt efter hvor meget tårnet kostede |
| En fjende dør | Man får penge alt efter hvilken fjende der døde | Man får penge alt efter hvilken fjende der døde |
| En fjende når helt hen til slutningen af stien | Man mister liv alt efter hvilken fjende der nåede hen til slutningen | Man mister liv alt efter hvilken fjende der nåede hen til slutningen |
| En fjende kommer ind i et tårns rækkevidde | Fjenden tager skade alt efter hvilket tårns rækkevidde som det kom ind i | Fjenden tager skade alt efter hvilket tårns rækkevidde som det kom ind i |
| Mere end 1 fjende kommer ind i et tårns rækkevidde | Tårnet gør kun skade til den første fjende i den rækkevidde | Tårnet gør kun skade til den første fjende i den rækkevidde |
| Brugeren trækker i et tårn | Man kan se tårnets rækkevidde | Man kan se tårnets rækkevidde |

# Konklusion

I starten af projektet ønskede vi at lave et spil som krævede fokus, logisk tænkning og engagement af dets spillere. VI lavede et Tower Defence spil, som vi mener er i stand til at inddrage dets spillere på de måder som vi ønskede det. Derudover lavede vi også en række krav for vores spil, som vi mener at vi opfyldte.

Vores Tower Defence spil har en sti som fjender kan følge helt til slutningen af banen. Spilleren er i stand til at placere tårne på banen, men kun de steder hvor det giver mening at spilleren ville kunne placere dem. Spilleren kan trække i et af tårnenes ikoner ude i menuen og slippe ikonet igen over banen og så vil tårnet blive placeret. Tårnene kan skyde på den første fjende i deres radius, der er dog ingen visuelle projektiler, men man kan se at fjenden tager skade af det alligevel. Som spiller har man et antal liv som bliver reduceret hvis en fjende kommer til slutningen af stien. Hvis man dræber en fjende eller køber et tårn, får man penge ved det ene og mister penge ved det andet. I spillet er der i alt 2 forskellige fjender og 4 forskellige tårne. Når man som spiller trækker et tårn ude fra menuen, så kan man se dets radius som en semi-gennemsigtig cirkel omkring det og man kan altid se alle tårnenes priser. Når man starter en ny runde, er den sværere end den sidste runde som man gennemførte.

Alt i alt fik vi lavet et spil som vi mener opfylder alle krav som vi stillede for det og det engagerer spilleren ligesom vi ville have det.