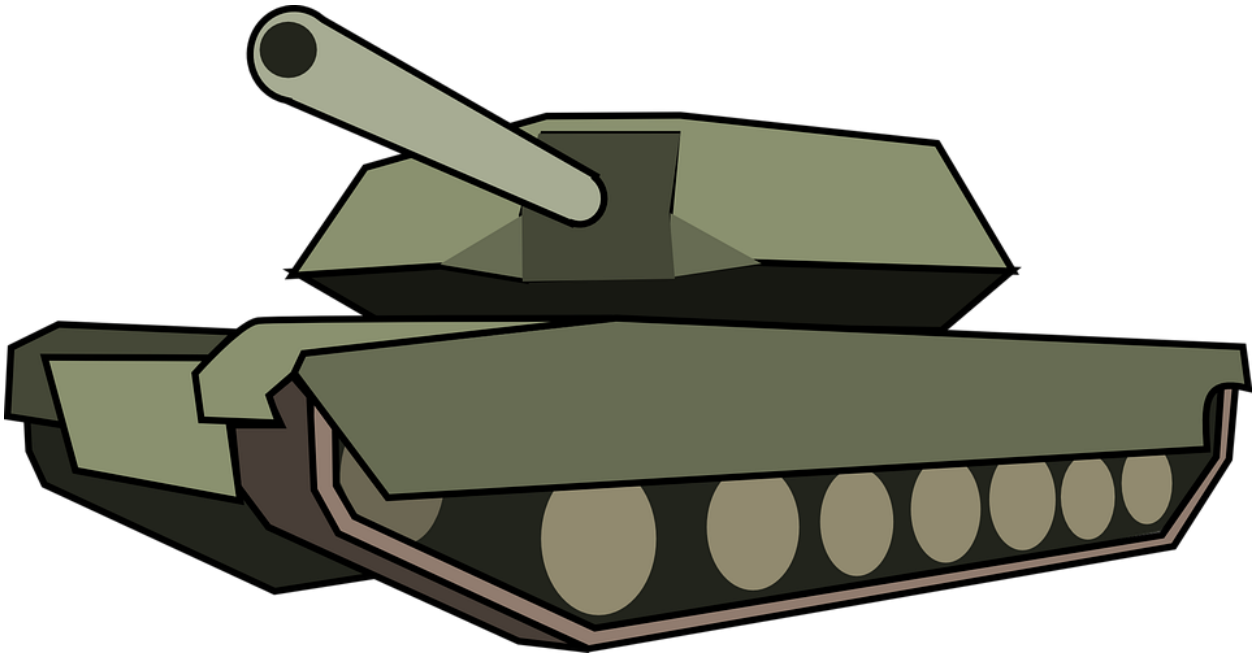


Synopsis af LazerTankz

Eksamensprojekt i Programmering B

Udarbejdet af:
Andreas & Aslak
Klasse: 18xar
HTX
AARHUS TECH TG Midtbyen
Vejleder: Mirsad Kadribasic
Dato for aflevering: 03-06-2020

Indholdsfortegnelse

Resume

Vi har lavet et spil til 2 personer, hvor personerne spiller på sammen tastatur. De kontroller hver sin tank og prøver at skyde hinanden med lasere. Spillet er 2D, set ovenfra, og tanksene kører på en sort labyrint-agtig bane med hvide vægge. Laserne rikochetterer på væggene. Imens man spiller vil der spawne gule firkanter, der symboliserer opgraderinger. En opgradering vil tillade en spiller at skyde igennem mure.

Programmet er lavet med Python, og med meget stor brug af biblioteket "pygame". Vi har valgt pygame frem for biblioteket arcade, da pygame har meget mere funktionalitet og er lavet til at lave spil. Arcade begynder også at køre med meget lav framerate meget nemmere end pygame gør, med mindre mulighed for optimering.

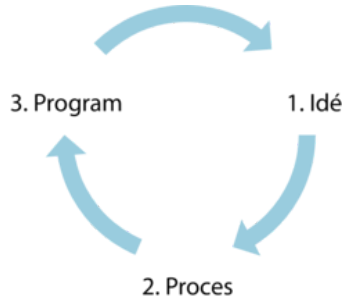
Problemformulering

Vi har i dette eksamensprojekt beslutte at vi ville lave et sjovt spil, som man kunne spille sammen med sine venner. Ud fra de idéer vi havde, valgte vi at lave et 2D skydespil med tanks. For at lave spillet har vi skullet tage følgende ting i betragtning:

- Hvordan skal man kunne spille sammen?
- Hvordan kan man bestemme spillerens bevægelsesretning?
- Hvordan kan man bestemme kollision?
- Hvordan kan man lave tilfældige spaiaonwn-positioner?

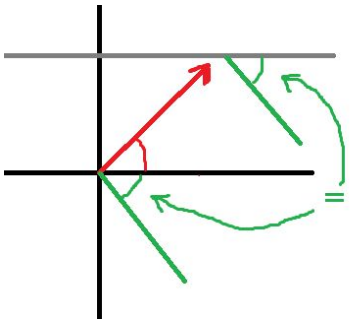
Metoder

I vores program benyttede vi os af nogle forskellige metoder, hvor nogle var mere overordnede end andre. Alt i alt har vi været i en kreativ proces. Denne proces er kendetegnet af, at man først søger efter idéer til et program. Dernæst kommer selve arbejdsprocessen, hvor man udvikler den første prototype af sit program, hvilket fører til den sidste fase, hvor man evaluerer denne prototype og igen får nye idéer, til hvordan man kan forbedre det. Denne kreative proces kaldes også for *agile programmering*, da processen handler om at kunne være adræt og løse eventuelle problemer på farten. Dette står i modsætning til *vandfaldsmetoden*, som i stedet fokuserer på at lave en overordnet plan, som følges helt indtil programmet er færdigt og opfylder planens krav.



I selve arbejdsprocessen brugte vi også metoder, som skulle hjælpe os med at opbygge og strukturere selve koden i vores program. Dette kan ses ved, at vi har gjort meget brug af metoden *Objekt Orienteret Programmering*. Denne metode gjorde det nemlig både meget lettere for os at skrive programmet, og lettere for computeren at køre programmet. Objekt Orienteret Programmering gør os nemlig i stand til at lave klasser, for alle de forskellige dele af programmet som er ens, og dernæst hurtigt skabe så mange nye objekter, som programmet skal bruge. Dette har vi gjort brug af til både vores tanks, projektiler, mure og opgraderinger.

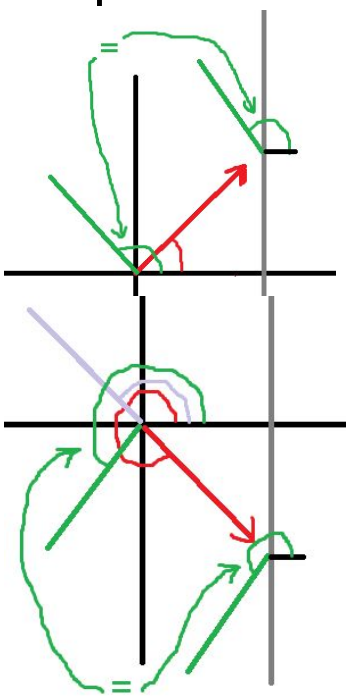
Udover metoderne som påvirker selve programmet og dets struktur, har vi også haft brug for at inddrage nogle matematiske metoder. Disse metoder skulle vi nemlig bruge til at bestemme bevægelsesretningen for vores tanks og projektiler. Begge metoderne blev udledt fra enhedscirklen, da deres bevægelsesretning blev bestemt af deres vinkel med x-aksen. Den første metode vi udledte, brugte vi til at bestemme hvor meget et objekts x- og y-koordinater ændrede sig, da dette ville skabe illusionen om, at den bevægede sig i en diagonal linje. Vi beregnede koordinaterne ved at danne en retvinklet trekant i enhedscirklen, hvor vinklen dannede hypotenusen. x-hastigheden var på den måde cosinus af vinklen ganget med fart-variablen, mens y-hastigheden var lig med sinus af vinklen ganget med den negative fart-variabel. Grunden til at fart-variablen skal være negativ er, at spil vinduets (0, 0)-koordinat ligger i toppen af skærmen. Derfor skal y-værdiens fortegn være "omvendt".



$$\Delta[x, y] = [speed \cdot cos(vinkel), \ -speed \cdot sin(vinkel)]$$

Den anden metode vi brugte, brugte vi til at ændre projektilers retning hvis de kolliderede med en mur, indgangsvinklen blev lig med udgangsvinklen. Hvis muren var horisontal, kunne vi få udgangsvinklen ved blot at trække indgangsvinklen fra 360:

$$vinkel_{ud} = 360 - vinkel_{ind}$$



Hvis muren var vertikal, skulle vi derimod dele enhedscirklen op, så indgangsvinkler i 1. og 2. kvadrant fulgte følgende udtryk:

$$vinkel_{ud} = 180 - vinkel_{ind}$$

mens indgangsvinkler i 3. og 4. kvadrant fulgte dette udtryk:

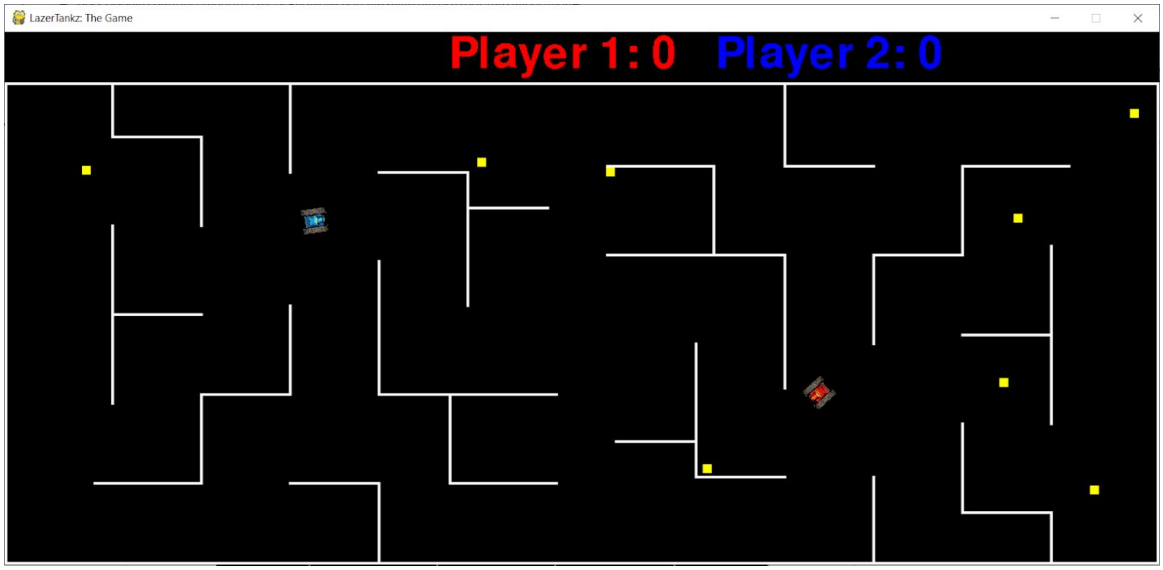
$$vinkel_{ud} = 360 - (vinkel_{ind} - 180)$$

Programbeskrivelse, Funktionalitet, Brugergrænseflader og Udvalgt kode

Vores spil er et 2D skydespil til 2 spillere. Hver spiller styrer en tank, hvor spiller 1 er rød og spiller 2 er blå. Banen ses oppefra og ned, og består af en sort baggrund med hvide mure. Murene står enten lodret eller vandret på skærmen. I toppen af vinduet er der 2 scorer, en til hver spiller, med tilhørende tekst og farve. Hver spiller kan bevæge sig ved hjælp af 4 taster (fremad, bagud, højre og venstre), og skyde med en femte knap. Projektilerne er grønne lasere, som afbøjes på væggene og forsvinder efter et givent stykke tid. Gule, firkantede opgraderinger vil poppe tilfældigt frem på banen. Disse opgraderinger gør, at 3 af spillerens skud kan gå igennem mure. Hvis en spiller kører ind i en opgradering, vil en gul tekst, der siger "UPGRADED!", stå over spillerens tank, indtil spilleren har brugt de 3 skud. Hvis en spiller bliver ramt af en laser og dør, vil spillet genstarte, og den vindende spiller får et point i sin score.

Brugergrænsefladen

Når man starter programmet, og lader det køre i lidt tid, ser spil-vinduet sådan ud:



Noget af det vigtigste omkring et program er, hvordan det tager sig ud. Dette gælder også for vores spil, da brugeren helst ikke skal blive forvirret over det, som bliver vist på skærmen. Derfor har vi skullet tage hensyn til Gestalt-lovene, som er vejledende regler for, hvordan man kan gøre sit GUI så let at forstå som muligt. I vores program har vi gjort brug af Loven om Figur og Baggrund, Loven om Lighed, Loven om Forbundethed,

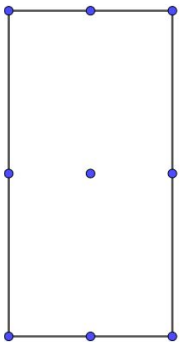
Loven om Nærhed og Loven om Lukkethed. Mange af vores visuelle elementer opfylder også flere af lovene samtidig.
Selve banen er sort, mens murene er hvide, opgraderingerne er gule, og tanksene og score-teksten er blå og rød. Dermed opfyldes Loven om Figur og Baggrund, da de lyse figurer står i skarp kontrast til den rolige mørke baggrund. Figurerne er også forholdsvis små, hvilket gør, at man ikke forveksler dem med baggrunden. Loven om Lighed og Loven om Forbundethed bruges i forhold til at gøre tanksene, deres score-tekst, samt opgraderings-teksten let genkendelige. De 2 tanks' former ligner hinanden, hvilket gør, at man ikke bliver i tvivl om, at de 2 tanks er "forbundet", og det samme med score-teksten. Tanksenes og score-tekstens farver forbinder dem også med hinanden, idet deres farvetema er identisk. Og teksten der popper frem, hvis en tank kører ind i en opgradering, er gul. Dét at skriften er gul, gør, at man let drager den konklusion, at teksten har noget at gøre med den gule opgradering, man lige kørte ind i. Loven om Nærhed hjælper også til denne konklusion, idet teksten dukker op tæt på den tank, der kørte ind i opgraderingen. Både denne lov Loven om Lukkethed hjælper også med at forbinde score-teksten, fordi de 2 score-tekster er tæt ved hinanden og lukket ude fra selve banen.

Biblioteket pygame

For at forstå, hvordan programmet fungerer kode-mæssigt, er man nødt til at forstå, hvordan biblioteket pygame fungerer, da vores programs opsætning og funktionalitet afhænger meget af det. pygame er en gruppe af python moduler, som er designet til at kunne lave spil med. pygame introducerer en masse nye klasser, som vi bruger. En af de vigtigste er Surface.

En Surface er brugt til at repræsentere et billede, men kun display surfacen bliver vist til brugeren. Man kan så bruge Surface-klassens *blit*-metode til at tegne (eller "blitte") én surface på en anden surface. Alt, som vi tegner på display surfacen, blittes til den fra andre surfaces. Til den ende har alle vores objekter *image*-variabler, der indeholder en surface, hvorpå billedet som repræsenterer objektet ligger.

For at kunne vide, hvor på en anden surface et surface skal blittes til, bruges der klassen Rect. En rect er essentielt set bare en gruppe af koordinater, der danner et rektangel (se billedet til højre). Man kan danne et rect fra en surface ved at kalde på surfacens *get_rect*-metode, som afleverer en rect med samme størrelse som surfacen, hvor punkterne har koordinater der passer til, at punktet øverst til venstre ligger på (0,0). Så kan man give rect-objektet nye koordinater ved at opdatere ét af dem, og de andre punkter følger med. En rect kan ikke roteres. Surface-klassens



blit-metode tager en rect som en koordinat-parameter, for at kunne vide, hvor surfacen skal tegnes på den anden surface.

Så er der et par mere avancerede klasser, som vi har gjort brug af: Sprite og Group. Det er meningen, at Sprite-klassen skal bruges som en grund-klasse, som man bygger sine egne klasser ovenpå. Det er også det, som vi har gjort. Alle vores klasser (undtagen en) arver enten direkte eller indirekte Sprite-klassens egenskaber. Group-klassen er så en slags beholder for Sprite-objekter. Det gør det nemmere at udføre den samme handling for hver at sprite-objekterne. Vi har brugt Group-objekterne til at gøre 3 ting nemmere. Den første er, at tegne objekterne på display surfacen. Det bruger Group-objektets *draw*-metode, til at blitte alle Sprite-objekterne den indeholder til en surface, ved at kigge på deres *image*- og *rect*-variabler. Den anden ting er at bruge Group-klassens *update*-metode, som kalder på alle Sprite-objekternes *update* metoder, med de nødvendige parametre. Til sidst, bruger vi også Group-objekterne til at registrere sammenstød mellem sprites ved at bruge pygame.sprites *groupcollide*-metode, som bruger enten rects eller bitmasks til at finde sprites fra 2 grupper der overlapper med hinanden.

Bitmasks er så det sidste vigtig ting fra pygame, som vi forklarer. De er repræsenteret af klassen Mask. Et Mask-objekt oprettes ud fra et billede (surface), og det repræsenterer alle pixlerne på billedet med kun 1 bit hver. Dvs., at hver pixel på billedet enten får værdien 1 eller 0. Om en pixel bliver sat til 1 eller 0, kommer an på, om den har samme farve som surfacens colorkey. Hvis de er ens, bliver pixelen 0. Colorkey'en repræsenterer altså den farve, som skal være "gennemsigtig". På den måde kan man bruge bitmasks til at registrere sammenstød, på en meget mere præcis måde end ved at bruge rects, da man kan sige, at hvis de overlappende pixels allesammen er 0, registreres der ikke sammenstød. Det er meget vigtigt ift. vores program, da vi har roterende tanks. Når et billede roteres, bliver dets rect nemlig meget upræcist, da den skal beholde formen af et ikke-roteret rektangel (se billedet til højre). Det hvide outline repræsenterer rect'en.

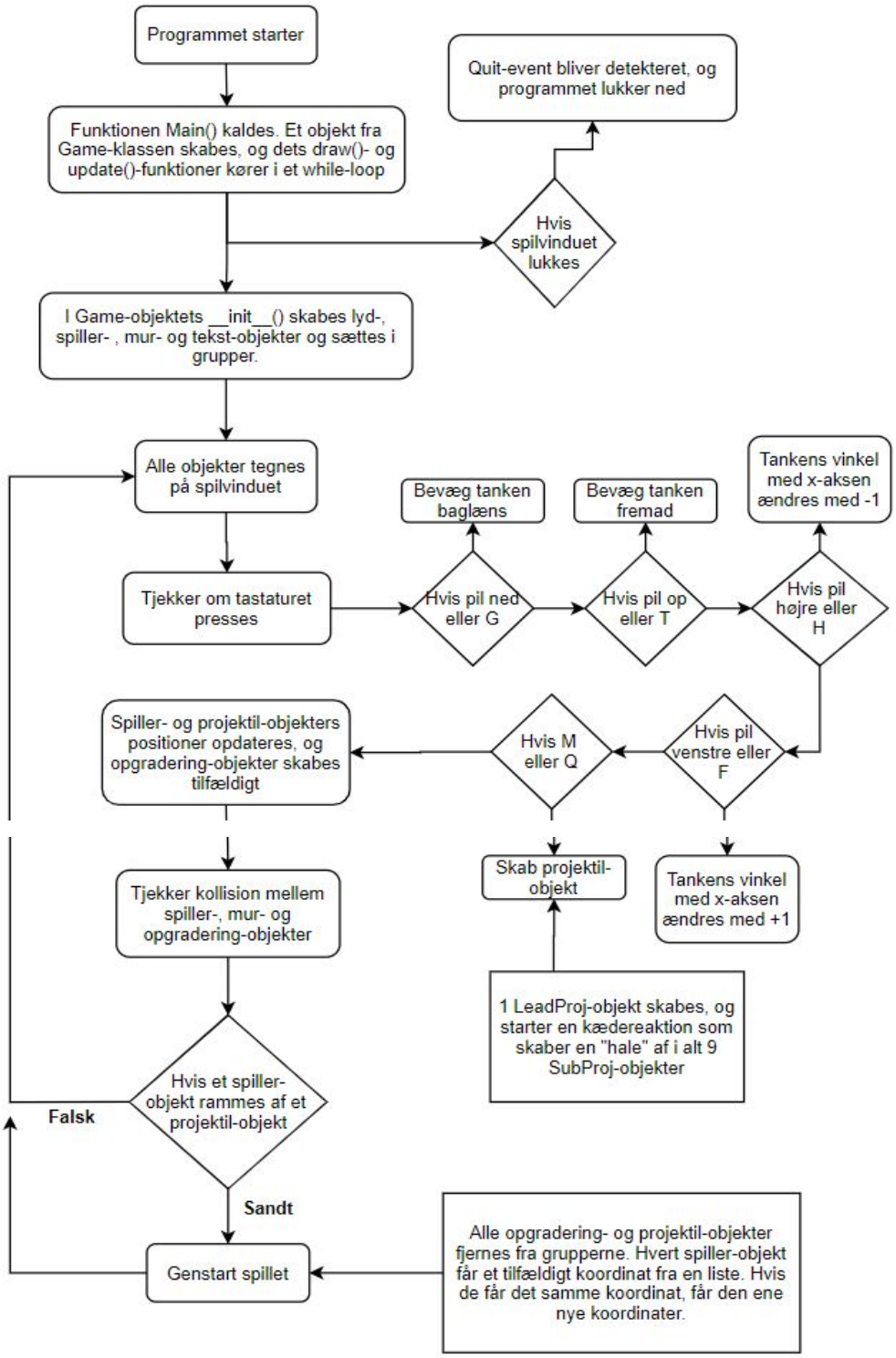
Endnu en ting vi brugte pygame til, var at afspille lydfiler i programmet. Måden vi gjorde dette på, var at skabe et pygame.Sound-objekt til hver lyd, med filstien til hver lydfil i parameteren. Vi fandt disse lydfiler på internettet, eller mere præcist på hjemmesiden *findsounds.com*[1]. Hvert Sound-objekt kan afspilles med dens metode Sound.play().

[1] Link: https://findsounds.com/

Pseudokode

1. Programmet starter og funktionen *Main()* kaldes.
2. Spil-vinduet åbnes. Lyd-, spiller-, mur-, clock- og tekst-objekter skabes og sættes i grupper.
3. Alle grupper og tekst tegnes på spil-vinduet.
    a. Hvis en spiller er blevet opgraderet, skriv "UPGRADED!" over den.
4. *update()*-metoden kaldes.
    a. Clock-objektet kontrollerer frame-raten.
    b. Der tjekkes for tastatur-input.
        i. Hvis "Q" eller "M", skab projektil-objekt i tilhørende liste.
    c. Spillernes og projektilernes positioner opdateres.
    d. Kører tilfældighed for at skabe opgradering-objekt.
    e. Tjekker kollisioner ved hjælp af pygames *collidegroups*-funktion.
        i. Hvis spiller1 kolliderer med mur/spiller2, stop spiller1.
        ii. Hvis spiller bliver ramt af projektil, genstart positioner.
            1. Hvis spillerne spawner på samme lokation, genstart lokationer igen.
        iii. Hvis en spiller kolliderer med en opgradering, bliver spilleren opgraderet.
        iv. Hvis et projektil rammer en mur, skal det rikochettere.
5. Gentag fra step 3

Flowchart


Programmet starter
Funktionen Main() kaldes. Et objekt fra Game-klassen skabes, og dets draw()- og update()-funktioner kører i et while-loop
Hvis spilvinduet lukkes
Quit-event bliver detekteret, og programmet lukker ned
I Game-objektets __init__() skabes lyd-, spiller- , mur- og tekst-objekter og sættes i grupper.
Alle objekter tegnes på spilvinduet
Tjekker om tastaturet presses
Hvis pil ned eller G
Bevæg tanken baglæns
Hvis pil op eller T
Bevæg tanken fremad
Hvis pil højre eller H
Tankens vinkel med x-aksen ændres med -1
Hvis pil venstre eller F
Tankens vinkel med x-aksen ændres med +1
Hvis M eller Q
Skab projektil-objekt
1 LeadProj-objekt skabes, og starter en kædereaktion som skaber en "hale" af i alt 9 SubProj-objekter
Spiller- og projektil-objekters positioner opdateres, og opgradering-objekter skabes tilfældigt
Tjekker kollision mellem spiller-, mur- og opgradering-objekter
Hvis et spiller-objekt rammes af et projektil-objekt
Falsk
Sandt
Genstart spillet
Alle opgradering- og projektil-objekter fjernes fra grupperne. Hvert spiller-objekt får et tilfældigt koordinat fra en liste. Hvis de får det samme koordinat, får den ene nye koordinater.

Class diagram

Navn: Player(pygame.sprite.Sprite)
Variabler: - mapsize (tuple) - speed (int, float) - turnSpeed (int, float) - playerNumb (int) - originalImage (surface) - image (surface) - rect (rect) - coords (tuple) - prevCoords (tuple) - angle (int, float) - prevAngle (int, float) - mask (mask) - upgrade (string) - upgradeCounter (int) - (alle variabler fra Sprite-klassen)
Metoder: - __init__(playerNumb, coords, speed, turnSpeed, image, mapSize), afleverer ingenting - update(keys), afleverer ingenting - rotate(), afleverer ingenting - collide(), afleverer ingenting - reset(self), afleverer ingenting - (alle metoder fra Sprite-klassen)

Navn: Projectile(pygame.sprite.Sprite)
Variabler: - radius (int, float) - color (tuple) - screen (surface) - length (int) - image (surface) - rect (rect) - coords (tuple) - spawned (boolean) - subProj (subProj) - (alle variabler fra Sprite-klassen)
Metoder: - __init__(coords, radius, color, screen, length), afleverer ingenting - update(), afleverer ingenting - (alle metoder fra Sprite-klassen)

Navn: LeadProj(Projectile)
Variabler:

Navn: SubProj(Projectile)
Variabler:

- speed (int, float) - angle (int, float) - mask (mask) - frames (int) - timer (int) - player (player) - projUpgrade (string) - (alle variabler fra Projectile-klassen)
Metoder: - __init__(coords, radius, color, screen, length, speed, angle, player, timer=1100), afleverer ingenting - update(), afleverer ingenting - wallCollide(walls), afleverer ingenting - (alle metoder fra Projectile-klassen)

- proj (leadProj, subProj) - screen (surface) - (alle variabler fra Projectile-klassen)
Metoder: - __init__(coords, radius, color, screen, length, proj), afleverer ingenting - update(), afleverer ingenting

Navn: Wall(pygame.sprite.Sprite)
Variabler: - coords (tuple) - length(int, float) - width(int, float) - orient (boolean) - image (surface) - rect (surface) - mask (mask) - (alle variabler fra Sprite-klassen)
Metoder: - __init__(coords, orient, length=100, width=3, invert=False), afleverer ingenting - (alle metoder fra Sprite-klassen)

Navn: Upgrade(pygame.sprite.Sprite)
Variabler: - color (tuple) - image (surface) - mask (mask) - rect (rect) - (alle variabler fra Sprite-klassen)
Metoder: - __init__(color, coords), afleverer ingenting - (alle metoder fra Sprite-klassen)

Navn: Game
Variabler: - size (tuple) - laserSound (sound) - laserBounceSounde (sound) - explosionSound (sound) - laserLength (int) - cooldown (int) - lastShot (int) - screen (surface) - player1 (player) - player2 (player) - players (group) - lasers (group - upgrades (group) - walls (group) - p1score (int) - p2score (int) - scoreFont (font) - upgradeFont (font) - clock (clock)
Metoder: - __init__(size, laserlength=10, cooldown=500), afleverer ingenting - draw(), afleverer ingenting - update(), afleverer ingenting - collideGroups(group1, group2, resultType), afleverer ingenting - reset(), afleverer ingenting - fixPlayerSpawn(), afleverer ingenting - fixUpgradeSpawn(group1, group2), afleverer ingenting - spawnUpgrade(prob), afleverer ingenting

Class beskrivelse

Player

Til at starte med arver Player-klassen pygames *Sprite*-klasse. Grunden til dette er, at Player-klassen så kan benytte sig af pygames *Group*-klasse.

Player-klassen repræsenterer tanken, som en af spillerne kontrollerer, og dens *update(keys)*-metode bliver derfor næsten kun brugt til at bevæge den i en bestemt retning, alt efter hvilke keys bliver passeret i metoden. *Keys* er et dictionary, som bliver passeret i funktionen fra Game-klassens *update*-metode. *Key*s' keys er konstanter fra pygame, der repræsenterer alle knapperne på tastaturet. Hvert element i det dictionary er så en boolean, der fortæller, om dens tilhørende knap, eller key, er presset ned. *update(keys)* tager så det dictionary og bruger det til at bestemme, hvordan tanken skal bevæge og rotere sig (ved at kalde på sin *rotate()* metode). *update(keys)* bliver kaldt af Game-klassens *update()*-metode.

Player-klassen har også en metode, *collide()*, som håndterer, hvordan Player-objektet skal håndtere sammenstød med både vægge og det andet Player-objekt. Den metode bliver kaldt af Game-klassens *collideGroups()*-metode, når den registrerer at Player-objektet er stødt ind i noget.

Til sidst er der Player-klassens *reset()*-metode, som sætter variablen angle til nul og giver objektet nye koordinater. Det skal gøres, når en af tanksene er blevet ramt af en laser. Player-klassens *reset()*-metode bliver kaldt af Game-klassens *reset()*-metode.

Projectile, LeadProj og SubProj

Vi forklarer disse klasser i det samme afsnit, da de er tæt knyttet til hinanden. De bruges allesammen til at oprette laserne, som bevæger sig på skærmen. For at forklare det, vil vi generelt forklare, hvordan laseren fungerer:

Laseren bliver skudt af en af tanksene (spillerne). Den består af en masse cirkler i en række, så det ligner en linje. Det er fordi, at så ser det bedre ud, når laseren rammer en væg og rikochetterer. Det er den forreste cirkel der holder styr på, hvordan de efterfølgende cirkler skal bevæge sig, da resten af cirklerne bare følger efter den. Den forreste cirkel er repræsenteret af klassen LeadProj, og resten af cirklerne er repræsenteret af klassen SubProj

Der bliver ikke oprettet nogen Projectile-objekter, når programmet kører. Det bruges kun til at blive arvet af to andre klasser; LeadProj og SubProj. I starten havde vi kun lavet LeadProj og SubProj, men de havde så meget tilfælles, at vi valgte at lave en Projectile-klasse, som indeholder alle de fælles dele af koden.

Projectile-klassen har de følgende nævneværdige egenskaber. Den har alle variablerne, som den skal bruge for at kunne tegnes på skærmen; radius, farve, image og rect. Den har også en variabel, kaldet *length (int)*, som bestemmer hvor mange flere Projectile-arvende objekter, der skal dannes, før laseren er lang nok. Den har kun én metode udover konstruktøren; *update()*. Det vigtige ved *update()* er, at den danner det næste Projectile-arvende objekt i kæden, som så bliver et SubProj-objekt. Det dannes så

selvfølgelig med en *length*-variabel, som er 1 mindre end den foregående cirkels *length*. *update()* skal også kalde på det nye objekts *update()*-metode. Derudover opdaterer *update()* også variablen *prevCoords*, som er projektilets forrige position.

Så kan SubProj bare have alle de egenskaber, som arves fra Projectile, med 2 tilføjelser. Den første er, at dens konstruktør har en parameter *proj*, som er det objekt før det nye SubProj-objekt i rækkefølgen. Det parameter sættes så i variablen *proj*. Så kan SubProj-objektet vha. dets *update()* metode sætte sine egne koordinater til at være *proj*-objektets forrige koordinater (*prevCoords*). Den anden tilføjelse er, at dens konstruktør har en *screen* parameter, som bliver puttet i variablen *screen*. Det er display-surfacet. Formålet med det er, at så kan SubProj-objektet "blitte" (eller tegne) sig på skærmen. Det er den nødt til at gøre selv, da den ikke tilhører et af pygames Group-objekter.

Til sidst forklarer vi LeadProj-klassen. Da den repræsenterer den forreste cirkel i kæden, er den nødt til at have mulighed for at bevæge sig og rikochettere fra vægge. Til det formål har den metoderne *update()* og *wallCollide(walls)*, og variablerne *speed, angle,* og *mask*. *speed* og *angle* bruges til at opdatere objektets position i *update()*, mens *mask* bruges af Game-objektet til at opdage sammenstød. *update()* bruger programmets funktion *updateCoords()* til at opdatere objektets position. Den funktion forklares i Funktionsbeskrivelse-afsnittet. *wallCollide(walls)* bruger den matematik, som er forklaret i Metoder-afsnittet. Derudover, har LeadProj variablerne *frames*, som sættes til nul i konstruktøren, og *timer*, som sættes til 1100 i parameteren. *update()* trækker så én fra frames, og når frames = nul kaldes dens *kill()*-metode, som den arvede fra pygames Sprite-klasse. *kill()* sletter spriten fra alle Group-klasser, der indeholder den. Så bliver LeadProj, og dermed resten af projektilerne der følger den, ikke tegnet eller opdateret længere. Til sidst, har LeadProj noget funktionalitet med *Upgrade*-klassen, som vi forklarer senere i Class beskrivelsen.

Wall

Wall er en simpel klasse, da den ikke har nogen metoder ud over konstruktøren. Den arver egenskaber fra pygames *Sprite*-klasse, ligesom alle de andre klasser i programmet gør (udover Game-klassen).
Wall-klassen eksisterer kun for at give andre sprites muligheden for at støde ind i den. Den har derfor alle de nødvendige egenskaber til at kunne tegnes på skærmen og stødes sammen med; *coords, length, width, image, rect, bitmask, orient* og *invert*. De forklarer sig selv, undtagen *orient* og *invert*. Orient er en boolean, som bestemmer om væggen er vandret (*orient = True)* eller lodret (*orient = False)*. De har ikke mulighed for at have

diagonale vinkler. *invert* bruges af Game-objektets konstruktør, til at skabe objekter på den højre halvdel af skærmen, som er omvendt med den venstre side.

Upgrade

Upgrade er også en simpel klasse, og som de andre arver den pygames *Sprite*-klasse. Den kan samles op af en af spillerne for at give dem en opgradering. Da den ikke bevæger sig, har den ikke bruge for metoder ud over konstruktøren. Den har alle de nødvendige variabler for at kunne tegnes på skærmen og kollideres med; *color*, *image, mask,* og *rect*. Dens størrelse er altid det samme, og der vil ikke være grund til at ændre det, så den sættes bare i konstruktøren, når *image* oprettes.

I teorien ville denne klasse have mulighed for at give mange forskellige bonusser til den spiller der samler den op. Til det formål har det variablen *upType*, som er en string, der fortæller, hvilken bonus det giver. Game-klassen ville så havde haft en liste med forskellige af disse strings, som der så vælges tilfældigt fra, når Upgrade-objektet oprettes. Som det er, har den kun én opgradering, som gør at spillerens næste 3 skud skydes gennem væggene. *upType* bliver passeret ind i et Player-objekts *upgrade*-variabel, når der er sammenstød mellem det Player-objekt og Upgrade-objektet.

For at muliggøre disse opgraderinger ville der være if-statements forskellige steder i programmet, der spørger om spillernes opgraderinger, alt efter hvilke funktionaliteter opgraderingerne ændrer.

Game

Game er den største klasse i programmet, da den opretter objekter fra alle de andre klasser (undtagen SubProj), tegner dem (undtagen SubProj), opdaterer dem (undtagen SubProj), opdager sammenstød mellem dem, tegner tekst, kontrollerer lyd og styrer display-surfacen.

Ift. styrelse af display-surfacen har Game-klassen variablerne *size*, som er størrelsen i skærmen, og *screen*, som er selve display-surfacen.

Konstruktøren opretter også alle Group-objekterne, der indeholder Sprite-objekterne. De grupper puttes ind i variabler, der hedder *players, laser, upgrades,* og *walls*. Væggene og spillerne oprettes med det samme, mens laserne ikke oprettes indtil de bliver skudt, og opgraderingerne bliver skabt på tilfældige tidspunkter.

Konstruktøren opretter også alle lydene der skal bruges, og putter dem i variablerne *laserSound*, *laserBounceSound* og *explosionSound*. Så opretter den også Font-objekter (fra pygame), der bruges til at tegne tekst på skærme. De hedder

scoreFont og *upgradeFont*. *scoreFont* bruges sammen med variablerne *p1score* og *p2score* til at tegne hvor mange point, hver spiller har fået på skærmen. *upgradeFont* bruges til at fortælle en spiller, at de har fået en opgradering.

Så opretter konstruktøren også variablerne *laserLength* og *cooldown*. *laserLength* bestemmer hvor mange cirkler (LeadProj- og SubProj-objekter) laserne består af, og *cooldown* bestemmer hvor lang tid man skal vente, før man kan skyde igen.

Funktionsbeskrivelse

updateCoords og Main

Vi starter med at beskrive funktionerne, der ikke hører til en klasse.

updateCoords(angle, coords, backwards=False)

updateCoords bruges af Player-klassen og LeadProj-klassen til at bevæge sig, og bliver givet en bestemt retning og hastighed.

Den starter med at sætte vinklen lig med sig selv modulo 360, for at sikre at vinklen ikke er større end 360°. Så oprettes en lokal variabel deltaCoords, som er en tuple, der indeholder, hvor meget der skal tilføjes til x- og y-koordinaterne for at bevæge objektet en bestemt længde. Så bestemmes koordinaterne ved at bruge cosinus og sinus, som forklaret i matematik-delen af Metoder-afsnittet.

Hvis *backwards* er True, bliver begge elementer i *deltaCoords* ganget med -1, da objektet så vil bevæge sig baglæns. Til sidst tilføjes *deltaCoords* til *coords* for at opdatere koordinaterne, og så returneres *coords*.

Main()

Main bruges til at køre programmet. Den opretter Game-objektet, og går så ind i et while-loop, som kalder på Game-objektets *draw()* og *update()* metoder igen og igen. Den tjekker også pygames events for at se, om krydset på vinduet er blevet trykket på, og hvis den er, lukkes programmet ved at bruge *sys.exit()*

Player-klassens metoder

__init__(playerNumb, coords, speed, turnSpeed, image, mapSize)

__init__ er Player-klassens konstruktør. Den starter med at kalde dens parent-klasses (*Sprite*) konstruktør. Så gør den følgende:

- *mapSize*, *speed, turnSpeed og playerNumb* variabler oprettes og sættes lig med parametrene af samme navn.
- *originalImage* oprettes ved at bruge *image*-parameteren, som er en filsti til et tank-image på computeren, for at oprette et pygame Surface-objekt med et billede af tanken med størrelsen 30x30.
- der kaldes *orginalImage*-objektets *set_colorkey(color)* metode med (0, 0, 0) som parameteren. Det gør, at når surfacen bliver "blittet" til skærmen, bliver de sorte pixels gennemsigtige. Derudover, gør det også, at når en bitmask tages af surfacen, bliver de sorte pixels ikke set. Det gør det så muligt at ignorere de sorte dele af billedet, når sammenstød registreres.
- *image*-variablen oprettes og sættes lige med *originalImage*.
- pygame.masks funktion *from_surface(surface)* bruges med *image* som parameter, for at lave en bitmask (pygame mask-objekt) af billedet, og putte det ind i variablen *mask*.
- *rect*-variablen oprettes ved at bruge *image*-variablens *get_rect()* metode. Den opretter et Rect-objekt med samme størrelse.
- *coords, prevCoords* og *rect*-objektets variabel *center* oprettes, og bliver sat lig med coords-parameteren.
- *angle* og *prevAngle* oprettes og sættes til nul.
- *upgrade* oprettes og sættes til None, og *upgradeCounter* oprettes og sættes til nul.

update(keys)

Starter med at sætte *prevCoords* lige med *coords*, før *coords* bliver opdateret. På samme måde bliver *prevAngle* sat lig med *angle*.

Så sker det vigtigste. Først tjekkes der, hvilket spiller-objekt er repræsenteret, ved at tjekke om dets *playerNumb*-variabel er lig med 1 eller 2. Så tjekkes der, om forskellige knapper på tastaturet er blevet presset ned, alt efter hvilken spiller det er. Hvis knappen for at køre fremad bliver presset, sættes *coords* lig med det, som *updateCoords*-funktionen afleverer, for at bevæge tanken fremad. Hvis knappen for at

køre baglæns bliver presset, kaldes *updateCoords*-funktionen igen, men denne gang med dens *backwards*-variabel sat til True. Dette gør, at tanken vil bevæge sig baglæns. Hvis knapperne for at dreje til henholdsvis højre eller venstre bliver presset, tjekkes der også om baglæns-knappen er presset. Hvis den ikke er, bliver *turnSpeed*-variablen henholdsvis trukket fra eller lagt til *angle*. Hvis den er presset ned, er det omvendt. Dette gør, at knapperne, til at rotere mens man bakker, bliver mere intuitive. *angle* bliver også sat lig med sig selv modulo 360, for at vinklen ikke bliver større end 360°. Derefter sættes *rect*-objektets *center*-variabel lig med *coords*, for at tanken rent faktisk bevæger sig hen på det nye koordinat. Til sidst tjekkes der om *upgradeCounter*-variablen er lig med nul. Hvis den er, sættes *upgrade* til None, eller med andre ord, opgraderingen tages fra spilleren.

rotate()

Denne metode roterer tanken, så den matcher vinklen givet af *angle*-variablen. Det gør den, ved at først oprette en lokale variabel, som hedder *center*, og som sættes lige med *rect*-objektets *center*-variabel. Så bruges pygame.transform's funktion *rotate(surface, angle)* med *originalImage* som den første parameter og *angle* som den anden. Det afleverer et nyt Surface-objekt, hvor tanken er roteret til den givne vinkel. Dette nye Surface-objekt sættes så i *image*-variablen. Så sættes *rect*-objektets *center*-variabel lige med *center*, som sikrer at dens centrum er på samme position som før rotationen. Ellers kommer billedet til at blævre og blive uklart. Til sidst sættes *mask*-variablen til en opdateret bitmask af billedet med den nye rotation, ved at bruge pygame.masks funktion *from_surface(surface)* med *image* som parameteren.

collide()

collide() er en meget simpel metode, som kaldes af Game-objektet, når der er sammenstød mellem Player-objektet og en væg eller det andet Player-objekt. Metoden sætter *coords* lig med *prevCoords* og sætter *angle* lig med *prevAngle*. Det sikrer, at hvis man prøver at køre gennem en væg, bevæger man sig ikke.

reset()

Denne funktion kaldes af Game-objektet, når en af spillerne dør, og spillet skal genstartes. Den sætter objektets vinkel tilbage på nul, ved at sætte *angle*-variablen til nul og så kalde *rotate()*. Den sætter også *upgradeCounter* tilbage til nul og *upgrade* til None.

Projectile-klassens metoder

__init__(coords, radius, color, screen, length)

Denne funktion er selvfølgelig Projectile-klassens konstruktør. Den opretter følgende variabler:

- *radius, color, screen* og *length* sættes lig med parametrene med samme navne.
- *image* bliver et nyt Surface-objekt fra pygame. Størrelsen af surfacen er *radius* gange 2 på begge akser.
- *rect* sættes lig med afleveringen fra *image*-objektets *get_rect()* metode, som afleverer et Rect-objekt med samme størrelse.
- *rect*-objektets variabel *center, prevCoords* og *coords* sættes lig med *coords*-parameteren.
- *spawned* sættes lig med False
- *subProj* sættes lig med None. Den bliver til et SubProj-objekt senere.

Derudover tegner den en cirkel på *image*, ved at bruge pygame.draws funktion *circle(surface, color, center, radius)* med følgende passeret i den (*image, color, (radius, radius), radius)*. Så fylder cirklen hele surfacen.

update()

update() gør 2 ting. Først, sætter den *prevCoords* lig med *coords*. Det andet den gør er lettest at vise med pseudokode med indrykning:

- hvis *length* er større end nul:
    - hvis *spawned* er False:
        - oprette et SubProj-objekt, med følgende passeret i dens konstruktør i følgende rækkefølge: (*prevCoords, radius, color, screen, length - 1,* dette objekt)
        - sætter *spawned* lig med True
    - kalder på SubProj-objektets *update()* metode

LeadProj-klassens metoder

__init__(coords, radius, color, screen, length, speed, angle, player, timer=1100)

Denne funktion er konstruktøren.

Den starter med at kalde på forældre-klassens konstruktør (forældre-klassen er Projectile-klassen), med følgende passeret i parametrene: (*coords, radius, color, screen, length - 1)*. Så opretter den følgende variabler, udover dem der blev oprettet af forældre-klassens konstruktør:

- *speed, angle, player og timer* sættes lig med parametrene med samme navne. Mærk, at *player* er et af Player-objekterne.
- bruger pygame.masks funktion *from_surface(surface)* med *image* som parameter, for at lave en bitmask (pygame mask-objekt) af billedet, og putte det ind i variablen *mask*. Før dét, sættes *image*-variablens (den blev oprettet af forældre-klassen) colorkey til (0, 0, 0), ved at kalde på dens *set_colorkey(color)* metode.
- *frames* sættes til 0.
- *projUpgrade* sættes lig med *player*-objektets *upgrade*-variabel.

Til sidst tjekker konstruktøren om *projUpgrade* er lig med 'passWall'. Hvis den er, trækkes der 1 fra *player*-objektets *upradeCounter* variabel.

update()

Denne metode starter med at tilføje 1 til *frames*. Så kalder den forældre-klassens *update*-metode. Derefter bruges funktionen *updateCoords(speed, angle, coords)*, med variablerne med samme navne passeret i den. Det er for at opdatere objektets koordinater. Til sidst, tjekker metoden om *frames* er lig med *timer*, og hvis den er, kaldes LeadProj-objektets *kill()*-metode, som det har arvet fra Sprite-klassen. Den metode fjerner objektet fra alle Group-objekter, der indeholder det. Det vil gøre, at objektet ikke længere bliver opdateret eller tegnet.

wallCollide(wall)

Først, tjekker vi om *projUpgrade* er lig med 'passwall'. Hvis det er, gør denne metode ingenting. Ellers tjekker den om *wall*-parameterens (det er Wall-objektet, som der er sammenstød med) *orient*-variabel er True eller False. Hvis det er True, sættes *angle* lige med 360 - *angle*. Hvis det er False, tjekkes er også om *angle* er større eller mindre end 180. Hvis det er mindre, sættes det til 180 - *angle*. Hvis det er større, sættes det til 360 - (180 - *angle*). Formålet med disse regninger er at rikochettere laseren, så dens indfaldsvinkel er lige med den udfaldsvinkel.

SubProj-klassens metoder

__init__(coords, radius, color, screen, length)

Dette er klassens simple konstruktør. Den starter med at kalde dens forældre-klasses (som er Projectile-klassen) konstruktør, med parametrene *coords, radius, color* og *screen* (fra SubProj-klassens konstruktørs parametre). Så sættes *proj* lig med *proj*-parameteren. Det er det Projectile-arvede objekt, der oprettede SubProj-objektet.

update()

Denne metode er også ret simpel. Den starter med at kalde på forældre-klassens *update()*-metode. Så sættes *coords* og *rect*-objektets variabel *center* lig med *proj*-objektets variabel *prevCoords*. Til sidst "blittes" objektet til (eller tegnes på) *screen*-variablen, som indeholder display-surfacen, ved at kalde på surfacens *blit(surface, rect)* metode med *image* og *rect* passeret ind i parametrene.

Wall-klassens metoder

__init__(coords, orient, length=100, width=3, invert=False)

Denne konstruktør er Wall-klassens eneste metode. Den starter med at sætte alle parametrene undtagen *invert* ind i variabler med samme navne.

Så tjekkes der, om *orient* er True eller False. Hvis den er True, oprettes der en variabel *image*, som sættes lig med en oprettet Surface(*size*) med *(length + width, width)* som parameteren. Det betyder, at murens længde bliver langs med x-aksen og dens bredde bliver langs med y-aksen. Med andre ord, væggen ligger vandret. Hvis *orient* er False, sker det samme, men der byttes rundt på længden og bredden, så væggen kommer til at stå lodret.

Som det næste, fyldes *image*-surfacen med farven hvid, ved at kalde på dens metode *fill(color)*. Dens colorkey sættes til (0, 0, 0). Selvom der ikke er nogen sorte pixels på væggen, er man nødt til at sætte colorkey'en, fordi det ellers ikke kommer til at fungere, når en bitmask tages af surfacen. Og at tage en bitmask er lige præcis det vi gør, og den sættes i variablen *mask*.

Til sidst, tjekkes der om *invert* er True eller False. Hvis det er True sættes *rect*-objektets *bottomleft*-variabel lige med *coords*. Ellers sættes dens variabel *topright* lige med *coords*. Det er for hurtigt at kunne oprette den højre halvdel af spillebanen, da den er en omvendt version af den venstre halvdel.

Upgrade-klassens metoder

__init__(color, coords, upType)

Upgrade-klassen har også kun 1 metode, som er konstruktøren. Den starter med at oprette variablerne *color* og *upType*, som den sætter lig med parametrene af samme navne. Så oprettes der et Surface-objekt, med (10, 10) passeret ind i dets *size*-parameter. Det puttes derefter i variablen *image*. Så fyldes surfacen, med farven som blev givet af *color*-parameteren, ved at bruge dens *fill(color)*-metode. Så sættes dens colorkey til (0, 0, 0), så der derefter kan laves en bitmask af den, som sættes i variablen *mask*. Så laves der også et Rect-objekt ud fra *image*, ved at bruge *image*-objektets metode *get_rect()*, og den rect sættes i variablen *rect*. Til sidst sættes *rect*-objektets variabel *center* lig med *coords*-parameteren.

Game-klassens metoder

__init__(size, laserLength=10, cooldown=500)

Denne konstruktør opretter følgende variabler:

- *size, laserLength* og *cooldown* sættes lig med parametrene af samme navne.
- *lastShot* sættes til den negative værdi af *cooldown*.
- *screen* sættes til en ny display surface, som oprettes ved at bruge pygame.displays funktion *set_mode(size)*, hvor *size* passeres i parameteren. Den funktion afleverer en display surface.
- *player1* sættes til et nyt Player-objekt, med følgende passeret i dets konstruktørs parametre: (*1, (0, 0),* [hardkodet float], [hardkodet float], [filsti til red tank.png], *size*).
- *player2* sættes til en ny Player-objekt, med følgende passeret i dets konstruktørs parametre: (*2, (0, 0),* [hardkodet float], [hardkodet float], [filsti til blue tank.png], *size*)
- *players* bliver et nyt Group-objekt, der indeholder *player1* og *player2*. Så kaldes Game-klassens metode *fixPlayerSpawn()*, som giver spillerne tilfældige startpositioner.
- *lasers* bliver et nyt tomt Group-objekt.
- *upgrades* bliver et nyt tomt Group-objekt.
- *walls* bliver et nyt Group-objekt, der indeholder alle Wall-objekterne på den venstre side af spillebanen. De er allesammen hardkodede. Så oprettes den højre

side af spillebanen, som er en omvendt version af den venstre side, ved at bruge et for-loop, der kører igennem *walls* og tilføjer Wall-objekter til den, med de samme koordinaterne som de forrige, men omvendte, og med omvendte retninger.
- *p1score* og *p2score* sættes til nul.
- *scoreFont* sættes til et pygame font-objekt med størrelse på 50.
- *upgradeFont* sættes til et pygame font-objekt med størrelse på 10.
- Font-objekterne oprettes ved at bruge pygame.font.Font(*filename, size*), hvor *filename* hentes ved at bruge pygame.fonts funktion *get_default_font()*.
- *laserSound, laserBounceSound,* og *explosionSound* sættes til nye pygame Sound-objekter, som indeholder en lydfil.
- *clock* sættes til et nyt pygame Clock-objekt, som holder styr på tid

draw()

Denne metode tegner alle objekterne (undtagen SubProj-objekterne, de tegner sig selv), og den kaldes af funktionen *Main()* hver frame.
Den starter med at kalde *screen*-objektets *fill(color)* med (0, 0, 0) passeret i den, for at viske alt ud, som der står på skærmen. Så kaldes alle Group-objekternes (*players, walls, lasers,* og *upgrades) draw(surface)*-metoder, med *screen* passeret i dem. Det tegner alle objekterne på skærmen, som de indeholder, ved at bruge deres *image* og *rect* variabler.
Så tegner den også tekst over spillebanen, som fortæller hvor mange point hver spiller har, ved at bruge *scoreFont*-objektets metode *render(text, antialias, color*) for at få en surface med teksten tegnet på, som så "blittes" til skærmen.
Så tjekker metoden om mindst et af Player-objekternes *upgrade*-variabler er True. Hvis et Player-objekts *upgrade*-variabel er True, tegnes der teksten "UPGRADED!" over den, på samme måde som den anden tekst tegnes, men hvor *upgradeFont* bruges.

update()

Denne metode kaldes hver frame af *Main()*-funktionen.
Først kalder metoden *clock*-variablens *tick(framerate=0*)-metode med 120 passeret i den. Det sikrer, at programmets framerate ikke kan være større end 120.
Så opretter metoden en lokal variabel *keys*, som sættes lig med afleveringen fra pygame.keys funktion *get_pressed()*. Som forklaret i Player-klassens klassebeskrivelse afleverer det en dictionary, hvor alle key'erne er pygame konstanter, der repræsenterer alle de forskellige knapper på tastaturet. Elementerne i dictionary'et er så en boolean, der fortæller om den knap er presset ned.

Så kaldes *players*-objektets *update(keys)*-metode, som kalder på *update(keys)*-metoderne af alle Player-objekterne, som det indeholder. *keys* bliver selvfølgelig passeret i parameteren.
Derefter kaldes også *lasers*-objektets *update()*-metode, som kalder på *update()*-metoderne af alle LeadProj-objekterne, som det indeholder.
Så tjekkes der, om antallet af millisekunder siden programmets start (hentet af pygame.times' funktion *get_ticks*) er større end *lastShot + cooldown*. Det er for at sikre, at man ikke kan skyde for hurtigt. Hvis det er større, tjekkes der så, om enten Q-tasten eller M-tasten er presset ned. Hvis Q er presset ned, skydes der en laser fra fronten af *player1*-objektet, med samme vinkel som *player1*-objektet. Hvis M er presset ned, sker der det samme, men hvor laseren er skudt fra fronten af *player2*-objektet. Laserene bliver dannet ved at oprette et LeadProj-objekt. Da en laser kan ramme spilleren, der affyrede den, er vi nødt til at garantere, at objektet ikke oprettes ovenpå Player-objektet. Det gør vi, ved at passere *updateCoords(20, self.player.angle, self.player.coords*) i dets *coords*-parameter når det oprettes. Det gør, at det er som om, laseren har bevæget sig 20 pixels væk i den rigtige retning, før det er oprettet. Så afspilles der en lydeffekt ved at kalde på *laserSound*-variablens *play()*-metode, og derefter sættes variablen *lastShot* lig med antallet af millisekunder siden programmets start, hentet på samme måde som før.
Så kaldes Game-objektets metode *collideGroups(group1, group2)* 5 gange, med forskellige værdier passeret i parametrene hver gang. Værdierne er følgende:

- (*players, walls*)
- (*lasers, walls*)
- (*lasers, players*)
- (*players, players*)
- (*players, upgrades*)

Det er for at tjekke, om der er sammenstød mellem bestemte objekter, og for at udføre de nødvendige handlinger, hvis der er.
Derefter kaldes Game-objektets metode *spawnUpgrade(10000)*. Det har den effekt, at der hver frame er en 1/10000 chance for, at en opgradering (Upgrade-objekt) oprettes et tilfældigt sted på spillebanen.
Til sidst kaldes pygame.displays funktion *flip()*, som opdaterer display-surfacen.

collideGroups(group1, group2)

Denne metode tjekker, om der er sammenstød mellem objekterne i 2 grupper, og hvis der er, gør metoden noget bestemt, afhængigt af hvilke grupper de er.
Til at starte med, oprettes der en lokal variabel *collisions*, som sættes lig med afleveringen fra pygame.sprites funktion *groupcollide(group1, group2, dokill1, dokill2, collided=None)*. Afleveringen en dictionary, hvor key'erne er alle objekterne i group1,

der støder sammen med et eller flere objekter fra group2. Elementerne til hver key er en liste med alle de objekter fra group2, som key'en støder sammen med. For eksempel, hvis der var et objekt i group1, der stødte sammen med 2 objekter fra group2, og et andet objekt fra group1 der stødte sammen med 1 objekt fra group2, ville dictionary'et se sådan ud:

key	element
group1.spriteA	[*group2.spriteA, group2.spriteB*]
group1.spriteB	[*group2.spriteC*]

Funktionen virker ved, at den tjekker om objekternes *rect*-variabler overlapper med hinanden.
Vi passerer (*group1*, *group2*, *False, False*) ind i funktionens parametre.
Så kører vi et for-loop gennem *collisions*.
For-loopet sætter *collisions* lig med en ny aflevering, fra et andet kald af *groupcollide*-funktionen. Vi passerer de samme værdier som før, undtagen at vi denne gang også passerer pygame.sprites funktion *collide_mask* ind i *collided=None* parameteren. Det gør, at funktionen tjekker objekternes *mask*-variabler (altså, deres bitmasks) i stedet for deres *rect*-variabler. Det er meget mere præcist, da kollisionen bliver pixel-perfekt, men det sænker frameraten. Grunden til, at vi tjekker med *rects* først, er for at skåne frameraten, da registreringen med bitmasks kun sker, hvis der er sammenstød med rects.
Derefter kører vi et for-loop gennem *collisions* igen. Dette for-loop indeholder 5 if-statements, der tjekker, hvilke Group-objekter blev passeret i *group1* og *group2*, ved at sammenligne dem med Game-objektets variabler for dem.

- Hvis *group1* er *players* og *group2* er *walls*, kaldes *collide()*-metoden af Player-objektet i den nuværende key.
- Hvis *group1* er *lasers* og *group2* er *walls,* kaldes *wallCollide(wall)*-metoden af LeadProj-objektet i den nuværende key. Det første *wall-objekt* i key'ens element-liste i dictionary'en bliver passeret i metodens parameter. Så er der et while-loop, hvor kriteriet, for at den fortsætter, er, at pygame.sprites metode *collide_rect(sprite1, sprite2)* afleverer True. Vi passerer de tidligere nævnte LeadProj- og Wall-objekter ind i parametrene. While-loopet kalder så LeadProj-objektets update()-metode. Alt dette er for at sikre, at laseren bevæger sig ud af væggen, før der igen registreres sammenstød mellem dem.

- Hvis *group1* er *lasers* og *group2* er *players*, tjekkes der først om det nuværende element i dictionary'en er en liste med *player1* eller *player2*. Hvis det er *player1*, tilføjes der 1 til *p2score*-variablen, og modsat. Derefter afspilles der en eksplosionslyd ved at kalde på *explosionSound*-variablens *play()* metode. Til sidst kaldes så Game-objektets *reset()*-metode, som sætter spillerens positioner til tilfældige koordinater, og sletter alle laserne og alle opgraderingerne.
- Hvis *group1* er *players* og *group2* er *players*, tjekkes der først om den nuværende key er lig med det nuværende element. Dette er for, at spillere ikke støder sammen med sig selv, som ville gøre, at de ikke kunne bevæge sig. Hvis key'en ikke er lig med elementet, kaldes der på begge Player-objekternes *collide()*-metoder.
- Hvis *group1* er *players* og *group2* er *upgrades*, sættes den nuværende key's (som er et Player-objekt) *upgrade*-variabel til det nuværende elements (som er et Upgrade-objekt) *upType*-variabel, og der tilføjes 3 til Player-objektets *upgradeCounter*-variabel. Så kaldes Upgrade-objektets *kill()*-metode, som sletter det fra alle Group-objekter, der indeholder det. Til sidst afspilles der en lyd, ved at kalde på *upgradeSound*-variablens *play()*-metode.

reset()

Denne metode genstarter spillet. Den kalder først på begge Player-objekternes *reset()*-metoder, kalder på Game-objektets *fixPlayerSpawn()*-metode, og sletter alle objekterne fra *upgrades*-objektet og *lasers*-objektet ved at kalde på deres *empty()*-metoder.

fixPlayerSpawn()

Denne metode giver Player-objekterne nye delvise tilfældige positioner.
Metoden starter ved at oprette en lokal variabel *spawnCoords*, som er en liste af tuples, hvor hver tuple indeholder koordinater x og y.
Så sættes variablerne *coords, prevCoords* for begge spillere til tilfældige elementer i listen *spawnCoords*. Player-objekteternes *rect*-objekters *center*-variabler bliver også sat til det element.
Så er der en while-loop, med kriterien at *player1*-variablens *coords*-variabel er lig med *player2*-variablens *coords*-variabel. I while-loopet vælges der et nyt element fra *spawnCoords*, som *player1*-objektets samme variabler som før bliver sat til. Det sikrer, at spillerne ikke starter med samme koordinater.

fixUpgradeSpawn(upgrade, group)

Denne metode sikrer, at et nyskabt Upgrade-objekt ikke kan oprettes ovenpå et Wall-objekt eller et Player-objekt.
Den indeholder et while-loop, som altid er True. While-loopet starter med at oprette en anden lokal variabel *collisions*, som sættes lig med afleveringen fra et kald af pygame.sprites funktion *spritecollide(sprite, group, dokill)*. Den funktion afleverer en liste af alle objekterne i *group*-parameteren, som har en overlappende *rect* med *sprite*-parameteren. Metoden passerer (*upgrade, group, False)* ind i parametrene. Så er der en if-statement, der spørger, om længden på listen er større end nul. Hvis den er, betyder det, at Upgrade-objektet blev oprettet ovenpå et andet objekt, og den skal derfor have nye koordinater, som sættes til et nyt tilfældigt sted på spil-vinduet. Ellers stopper while-loopet.

spawnUpgrade(prob)

Denne metode har parameteren *prob*, hvilket er sandsynligheds-variablen. Hvis *prob* er lig med 1, tilføjes et Upgrade-objekt til Group-objektet *upgrades*. Derefter kaldes *fixUpgradeSpawn*-metoden, med Upgrade-objektet og først *walls*, og derefter *players* i parameteren for at løse eventuelle kollisioner.

Test

Da vi testede programmet gjorde vi det i 2 faser. I fase 1 testede vi brugergrænsefladens interaktionsmuligheder. Resultaterne af denne test kan findes i nedenstående skema:

Input	Forventet output	Faktisk output
Trykker “pil ned”	Tanken bakker	Tanken bakker
Trykker “pil op”	Tanken kører frem	Tanken kører frem
Trykker “pil højre”	Tanken drejer mod højre	Tanken drejer mod højre
Trykker “pil venstre”	Tanken drejer mod venstre	Tanken drejer mod venstre
Trykker “M”	Tanken skyder	Tanken skyder

I fase 2 testede vi programmets beregninger. Dette inkluderede kollisioner, timere og tilfældige spawn-lokationer. Vi testede kollisionerne ved at køre ind i mure og opgraderinger skiftevis med den blå tank, den røde tank og dem begge på samme tid. Vi gennemførte samme procedure med hensyn til projektiler.

Med hensyn til kollisioner viste testen, at tanken ville stoppe hvis den kolliderede med en mur eller den anden tank. Hvis tanken kolliderede med en opgradering, ville denne opgradering forsvinde, og tanken ville modtage opgraderingen i den tilhørende variabel. Hvis et projektil kolliderede med en mur, rikochetterede den med den rigtige udgangsvinkel. Hvis projektilet kolliderede med en tank, ville alle projektiler og opgraderinger forsvinde, og tanksene ville få nye koordinater. Hvis projektilet kolliderede med en opgradering ville intet ske, og efter et passende stykke tid fik timeren projektilet til at forsvinde. Alt i alt virkede alt dette, som det skulle.
De tilfældige spawn-lokationer viste sig også at virke fint. Vi startede og lukkede programmet mange gange, og vi testede også, om det virkede, hvis tanksene blev skudt, hvilket det gjorde.

Alle vores tests gav altså det forventede resultat, og derfor kan vi erklære vores program funktionsdygtigt.

Konklusion

Igennem arbejdsprocessen har vi skullet overveje de spørgsmål vi stillede os selv under Problemformuleringen. Vi vil her hurtigt gennemgå, hvordan vi løste problemerne.

Det færdige program er et 2D skydespil for to spillere. Vi lavede programmet sådan, at man spiller på samme computer. Spiller 1 bruger tasterne T, F, G og H til at bevæge sig og Q til at skyde, mens spiller 2 bruger piletasterne og M. Som skrevet i matematik-delen af Metoder, bestemte vi bevægelsesretningen, ved at danne en retvinklet trekant af enhedscirklen og benytte trigonometri til at finde kateternes længder. Kollisionen blev tjekket ved hjælp af pygames gruppe- og mask-funktioner, idet vi, ved at bruge disse, kunne bestemme kollision hurtigt og pixel-præcist. Og sidst men ikke mindst, har vi de tilfældige spawn-positioner. Disse bestemte vi ved at have en masse hardkodede koordinater, som hver spiller trak tilfældigt imellem.

Bilag

Kode med ”style” og kommentar

Koden ligger i en pastebin inde på det nedenstående link, da det vil gøre det meget lettere at læse koden, end hvis den havde været inde i dokumentet:
https://pastebin.com/KKxWjS3s

Litteraturliste

- Buch, Jesper. (3 februar 2017). *Programmering*. Systime. https://programmering.systime.dk/

- Buch, Jesper; Damhus, Martin; Husum, Elisabeth; m.fl.. (2017). *Gestaltlovene | Informatik*. Systime. https://informatik.systime.dk/?id=1132&L=0

- Gardener, Indoor. (s.d.). *Preview - Tank 2d Top View, HD Png Download*. PNGITEM. https://www.pngitem.com/middle/JoJxxw_preview-tank-2d-top-view-hd-png-download/

- Madsen, Preben. (marts 2010). *Teknisk Matematik 4. udgave*. Erhvervsskolernes Forlag.

- Shinners, Pete. (s.d.). *Pygame Documentation*. Pygame. https://www.pygame.org/docs/tut/PygameIntro.html