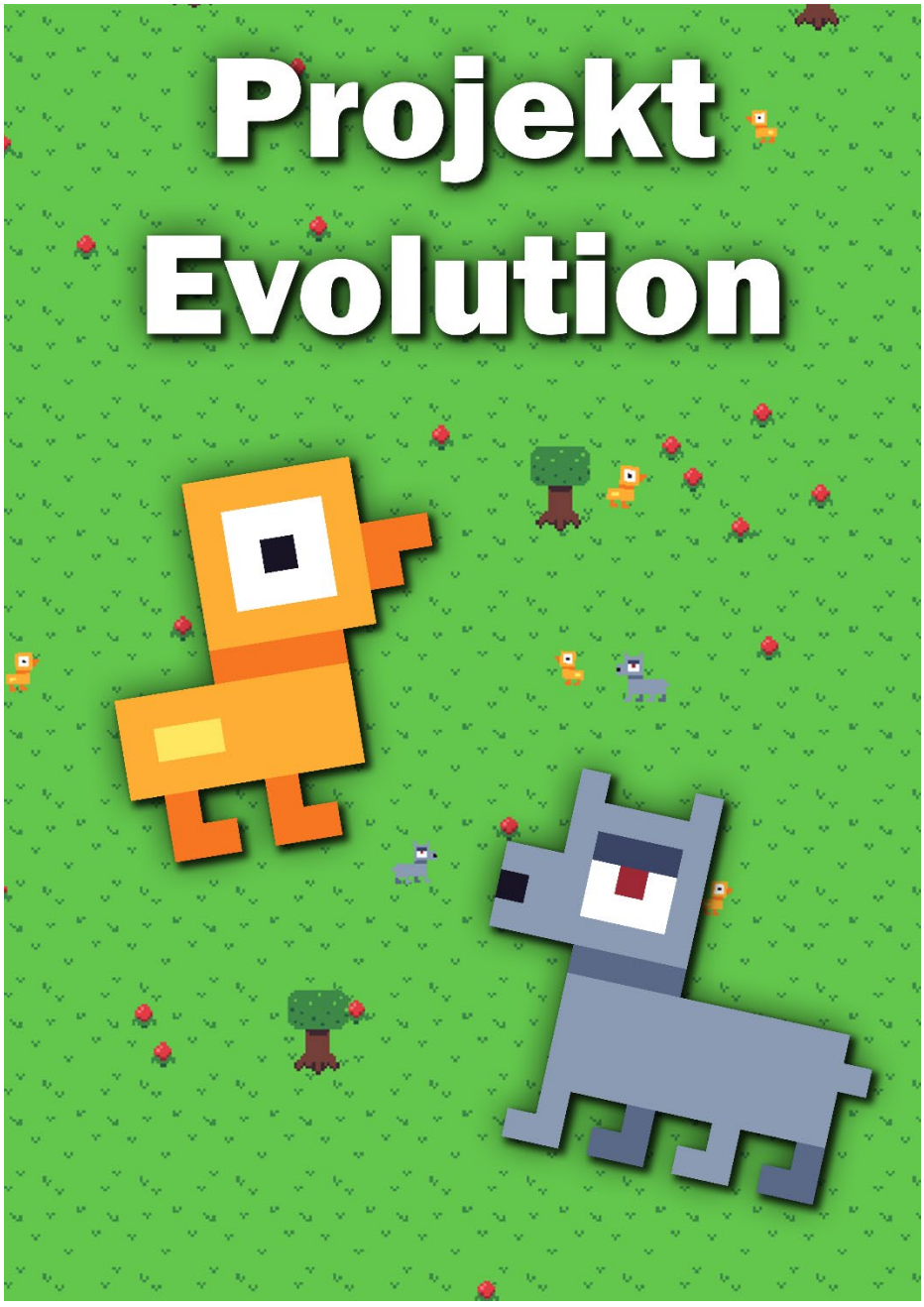
Programmering B - Eksamenssynopsis

Kunstprodukt om simulation af evolution

Af Rasmus Aagot Riis



Gruppemedlemmer: Rasmus Aagot Riis, Mikkel Agerholm Erikstrup

Vejleder: Magnus Håkon Petersen

Afleveringsdato: 10-05-2023

Tegn: 16.691 (6,95 sider)

Abstract

This synopsis starts off by choosing one of three themes, and creating a defined problem, based on the theme, art. The defined problem is making an art-product that visualizes evolution in an entertaining and understandable way. Next, the synopsis dives into describing the functionality of the product. The program contains two types of animals; the prey, visualized by ducks, and the predators, visualized by wolves. The wolves hunt the ducks, and the ducks eat strawberries. When two animals breed, their genes mix, which creates a survival-of-the-fittest dynamic. The user of the product has the ability to inspect each animals' genes/stats/data. Next up in the synopsis is the code explanation. The various choices, specifically choices related to good code architecture, are explained. After this, part of the animals' behavioral code is explained with the help of flowcharts, diagrams and pseudocode. The program was also optimized, which is explained. The next part of the synopsis explores how the optimization affected the performance of the program. The results show an improvement of a 35,6% CPU usage reduction. In the reflection chapter, the author looks back and explores what they learned during this process. This being the importance of the use of clean, readable and optimized code. Lastly, it is concluded that the created program succeeded in visualizing evolution in an entertaining way but could use with more understandability.

Indhold

Abstract 2
Problemformulering 1
Funktionsbeskrivelse 1
Genetikinspektør 2
Knapper 2
Statuslinje 3
Dokumentation 3
Kodestruktur 3
Nedarvning 3
States og mindskning af if-else statements 4
Dyrenes adfærdskode 5
Optimering 7
Test af programmet 10
Data 10
Resultater 10
Refleksion 10
Konklusion 11
Bilag 12
Bilag 1 - Screenshot af den originale tilgangs effektivitet 12
Bilag 2 - Screenshot af den endelige tilgangs effektivitet 12
Bilag 3 - Kildekode 12
Bilag 4 - Build 12

Problemformulering

Det første skridt i produktionen af dette projekt, var at vælge et af tre temaer. Temaerne er *retro games*, *automatisering* og *kunst*. Gruppen valgte at tage udgangspunkt i temaet kunst, både fordi dette er et meget bredt tema, men også fordi gruppen allerede i forvejen havde en idé til et projekt. Dette projekt ville nemmest kunne udføres via kunst teamet. Idéen til dette projekt var at lave et program, der kan visualisere evolution, specifikt konceptet om "*Survival of the fittest*". Ud fra idéen til dette projekt og temaet kunst, blev den følgende problemformulering udviklet og valgt:

Hvordan kan vi lave et kunstprodukt, der visualiserer evolution på en underholdende, overskuelig måde?

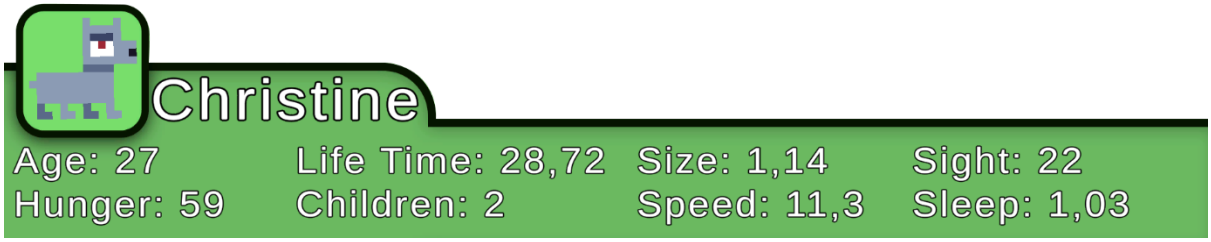
Funktionsbeskrivelse

I dette program er der to typer dyr. Disse er rovdyr, ofte refereret til som *predators* i koden, og byttedyr, ofte refereret til som *prey* i koden. Rovdyr er repræsenteret af ulve, og byttedyr er repræsenteret af ænder. Begge dyrearter har to centrale mål. Disse mål er at spise mad og at pare sig. Forskellen på dyrene, er dog, at ænderne spiser jordbær og ulvene spiser ænder. Alle dyr ejer en række af variabler, der beskriver deres genetik. Dette er f.eks. hvor hurtige de er, eller hvor langt de kan se. Når dyrene parer sig, får deres unger en blanding af begge forældres gener, med en smule tilfældig varians. Dette skaber en spændene dynamik, da det promoverer at de bedste gener går videre til den næste generation.

Figur 1 - Screenshot af det endelige program

Det overstående billede (Figur 1) viser et screenshot fra det endelige program. I midten af skærmen vises *spillebrættet*. Her refererer spillebrættet til den centrale del af skærmen, hvor handlingen tager sted. I dette skærmbillede ses der en ulv og en and på spillebrættet, samt en masse jordbær og et enkelt træ. Brugeren kan navigere spillebrættet ved brug af WASD-tasterne eller piletasterne. Ved at rulle med musehjulet, kan brugeren også zoome både ind og ud. Udover spillebrættet kan spillets UI (*User Interface*) ses. Denne UI inkluderer en genetikinspektør, fire knapper og en statuslinje.

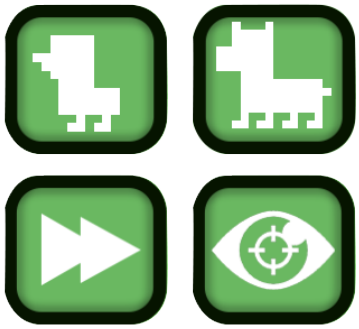
Genetikinspektør



Figur 2 - Billede af genetikinspektøren

Når der klikkes på et dyr, blive genetikinspektøren (Figur 2) vist. Genetikinspektøren indeholder diverse information om det valgte dyr. Dette er information såsom dyrets navn, dets alder, sultniveau, livstid, antal børn, størrelse, hastighed osv. Samt disse informationer, bliver der også vist et billede af dyret, som afspiller samme animationer, som dyret på spillebrættet. Genetikinspektøren bruges til at undersøge forskellig data/genetik om et specifikt dyr, så brugeren f.eks. kan holde øje med om et dyr er hurtigere eller langsommere end dets forældre. For at skjule genetikinspektøren igen, skal der blot klikkes et vilkårligt sted på spillebrættet.

Knapper



Figur 3 - Billede af de fire knapper i spillet

Der er i alt fire knapper i programmet (Figur 3). De to første knapper har næsten samme funktion. Dette er *gennemsnitsknapperne*. Der er både en gennemsnitsknap til ænderne og hundene. Ved at klikke på en af disse gennemsnitsknapper, bliver genetikinspektøren vist. Denne gang viser genetikinspektøren de gennemsnitlige variabler af en hel dyregruppe. Dette er så brugerne kan få et generelt overblik over, hvordan dyrenes gener udvikler sig over tid.

Den næste knap er *tidsspole knappen*. Denne knap har en meget simpel funktion, og det er at gøre så tiden forløber hurtigere. Ved at trykke på denne knap, gør brugeren, så tiden går tre gange hurtigere. Brugeren kan trykke på knappen igen, for at få tiden til at forløbe normalt igen.

Den sidste knap er *forfølg knappen*. Denne knap er kun synlig imens genetikinspektøren er aktiv. Når brugeren trykker på forfølg knappen, begynder kameraet at følge efter det valgte dyr. Samme funktionalitet kan opnås ved at trykke på *F*-tasten, samtidig med at genetikinspektøren er aktiv. Formålet med dette, er at gøre det nemmere for brugeren, at holde øje med et specifikt dyr. Alternativet til forfølg knappen, er nemlig manuelt at styre kameraet efter det ønsket dyr. For at stoppe med at forfølge et dyr, behøver brugeren blot at manuelt bevæge kameraet, hvilket med det samme deaktivere forfølg-funktionen.

Statuslinje

Statuslinjen sidder altid helt øverst på skærmen (Figur 1). Statuslinjen er repræsenteret af en gul og grå bar; samme farver som ænderne og ulvene. Formålet med statuslinjen, er at repræsentere ratioen af ænder og ulve. Det vil sige, at hvis f.eks. 30% af de resterende dyr er ænder, vil statuslinjen være 30% gul og 70% grå. Statuslinjen giver brugeren en hurtig indikation over, hvordan for forholdet mellem antal af ænder og antal af ulve ser ud.

Dokumentation

Til at skabe dette program, bliver programmet Unity brugt. Unity er et gratis stykke spiludviklingssoftware, som kommer indbygget med mange funktioner, der gør det muligt og hurtigt at lave f.eks. en prototype til et spil. Der er blandt andet fra starten indbygget støtte for grafik og UI. Unity bruger objekter ved navn *GameObjects*, til at opbygge dets *scener*. Hver af disse GameObjects, kan blive tildelt scripts, der indeholder en class, som nedarver fra *MonoBehavior*. MonoBehavior er en class, der kører kode, når programmet initialiseres. MonoBehavior indeholder blandt andet funktioner såsom *Start*, *Update*, og *FixedUpdate*. Start kaldes når GameObjectet initialiseres. Update kaldes hver frame. FixedUpdate kaldes 60 gange i sekundet.

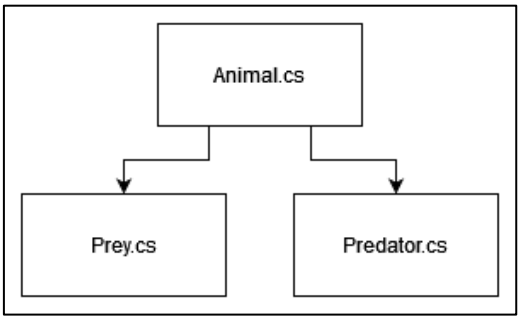
Under dette forløb har jeg produceret alt koden i de følgende scripts:

- Animal.cs
- Prey.cs
- Predator.cs
- Food.cs
- Variables.cs
- InspectAnimals.cs

Kodestruktur

Allerede fra starten var det vigtigt at holde koden organiseret. Dette afsnit handler om, hvordan god kodestruktur blev opnået i projektet.

Nedarvning

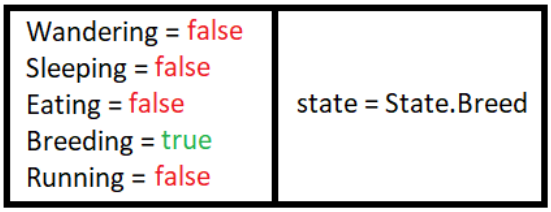


Figur 4 - Diagram over nedarvning

Der er i alt tre scripts, der står for at håndtere dyrenes adfærd. Disse scripts er *Animal.cs*, *Prey.cs* og *Predator.cs*. Animal.cs nedarver fra MonoBehavior, og både Prey.cs og Predator.cs nedarver fra Animal.cs. Nedarvningen er illustreret på figur 4. Grunden til denne opsætning, er at både byttedyr og rovdyr er dyr. Dette vil sige at byttedyr og rovdyr deler en masse kode. For at undgå at skrive denne kode to gange, bliver så meget kode som muligt opsat i Animal.cs, så koden kan genbruges af både Prey.cs og Predator.cs.

States og mindskning af if-else statements

Allerede fra starten var der opmærksomhed på, at adfærdskode hurtigt kan blive meget indviklet og uoverskueligt. Dette bliver hurtigt accelereret, hvis adfærden bliver håndteret med en masse booleans og if-statements. For at undgå dette, bliver der i stedet udnyttet enumerators og switch-statements.



Figur 5 - Diagram over brugen af enumerators i forhold til booleans

Som illustreret ovenover (Figur 5), kan arbejdet af en masse booleans, under rette omstændigheder, udføres af en enkelt enumerator. Endnu en fordel ved at bruge enumerators fremfor booleans, er evnen til at integrerer dem effektivt med switch-statements. Dog switch-statements og en lange række af if-else statements har samme funktion, er switch-statements generelt mere overskuelige, modulere og nemmere at debugge.

```
void Eksempel()
{
    if (condition)
    {
        if (condition)
        {
            if (condition)
            {
                //Kode
            }
        }
    }
}
```

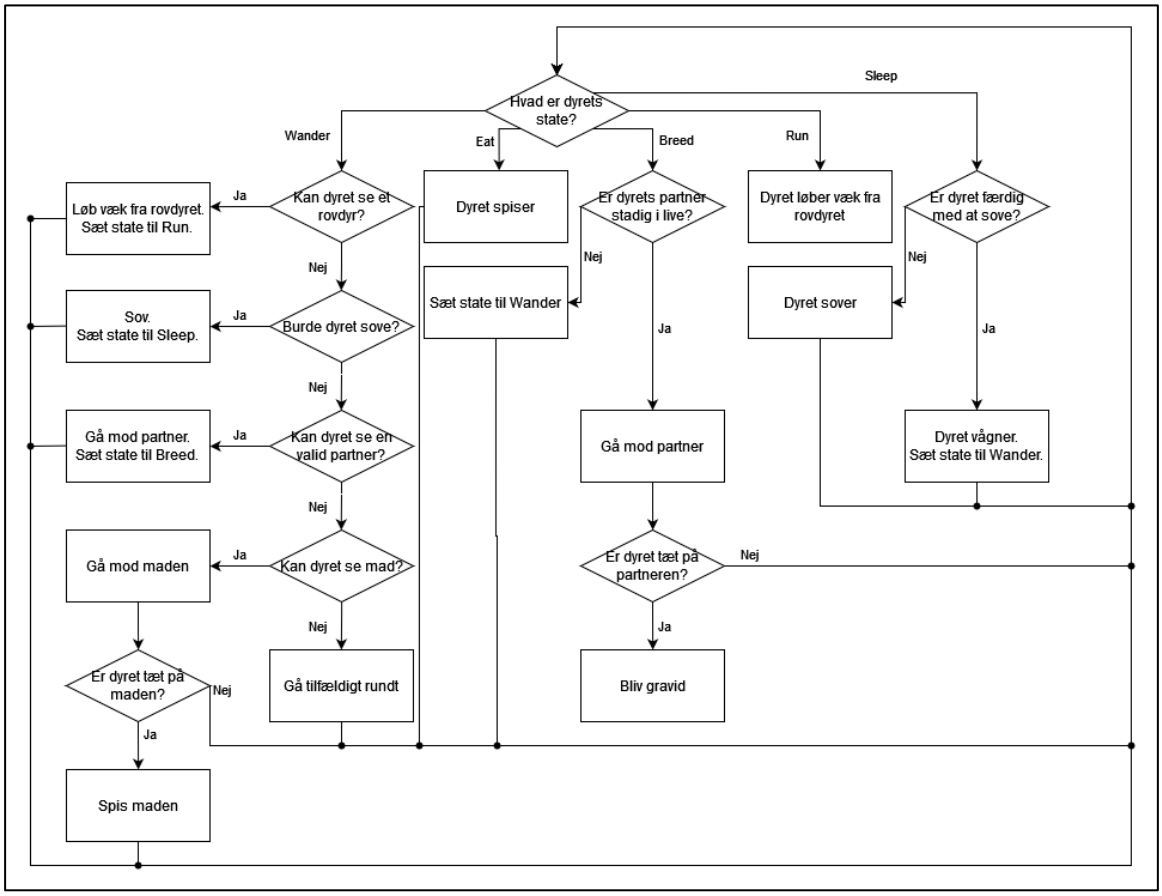
```
void Eksempel()
{
    if (!condition)
    {
        return;
    }

    if (!condition)
    {
        return;
    }

    if (!condition)
    {
        return;
    }

    //Kode
}
```

Figur 6 - Pseudokode, der viser forskellen på brug af indlejret if-statments og brugen af return-statements

Endnu en måde at mindske brugen af if-else statements, er den hyppige brug af return statements. Return-statements kan under de rette omstændigheder erstatte lange indlejret if-else statements. Ovenover (Figur 6) ses et eksempel af indlejret if-statements omskrevet til at bruge return-statements. Hvis der i fremtiden opstår fejl/bugs i koden, bliver kodestumpen, der bruger return-statements, meget mere overskuelig at fikse.

Dyrenes adfærdskode

Det følgende afsnit beskriver et par af de væsentlige kodestumper i dette projekt. Kodestumperne er understøttet med diagrammer, flowdiagrammer og pseudokode.



Figur 7 - Flowdiagram, der visualiserer FixedUpdate loopet i Prey.cs

Det overstående billede (Figur 7), viser et flowdiagram, der visualiserer FixedUpdate loopet i Prey.cs. Her tages der altså udgangspunkt i Prey.cs. FixedUpdate loopet i Prey.cs og Predator.cs er næsten de samme. Forskellen er at Prey.cs tjekker hvis den burde løbe væk fra rovdyr, og Predator.cs spiser byttedyr i stedet for jordbær.

FixedUpdate loopet indeholder kun et enkelt switch-statement, der tager imod en enumerator, som håndterer dyrets nuværende stadie, refereret til i koden som *state*. Den første af disse states, der bliver tjekket for, er *Wander*. Wander er standard stadiet, som dyrene for det meste befinder sig i. I dette stadie, bliver der tjekket for en række af betingelser.

```
Predator closestPredator = null;

//Holder styr på behaviour
void FixedUpdate()
{
    switch (state)
    {
        case State.Wander:
            //Hvis der er en predator i nærheden
            //bør dyret løbe væk
            closestPredator = GetPredatorInSight();
            if (closestPredator)
            {
                StartCoroutine(Run());
                state = State.Run;
                break;
            }
```

Figur 8 - Uddrag fra FixedUpdate loopet i Prey.cs

Den første betingelse, der tjekkes, er hvorvidt der er et rovdyr i nærheden. På billedet ovenover (Figur 8), kan der ses et udsnit af denne betingelse. Dette gøres ved at sætte variablen *closestPredator* til at være lig med resultatet af funktionen *GetPredatorInSight*. GetPredatorInSight funktionen returnerer det tætteste rovdyr, hvis der findes et. Ellers returnerer funktionen null. If-statementet tjekker om closestPredator indeholder data, altså om den er null. Hvis closestPredator indeholder data, betyder det, at byttedyret bør løbe fra rovdyret. Derfor startes coroutinen *Run*, og dyrets state sættes til *Run*. Dernæst er der et break-statement. Break-statementet afbryder switch-statementet, så intet af den nedenstående kode køres.

```
//Hvis det er nat, bør dyret sove
if (timeManager.Time >= timeManager.DuskTime && !slept)
{
    animator.SetTrigger("Sleep");
    state = State.Sleep;
    break;
}
```

Figur 9 - Betingelse i FixedUpdate, i Prey.cs, der tjekker om dyret bør sove

Resten af betingelserne, under Wander casen, tjekkes på samme måde, med to undtagelser. På det overstående billede (Figur 9) ses en af dem. Dette if-statement håndterer om dyret burde sove. Denne betingelse er sand, hvis den globale tid er over tiden for solnedgang, og hvis dyret ikke har sovet endnu.

```
//Ellers skal dyret bare går tilfældigt rundt
if (CurrentWanderPoint == Vector3.zero)
{
    float clampMin = -MaxDistFromCenter + MaxDistFromCenter / 10f;
    float clampMax = MaxDistFromCenter - MaxDistFromCenter / 10f;
    CurrentWanderPoint = new Vector2(
        Mathf.Clamp(transform.position.x + Random.Range(-20, 20), clampMin, clampMax),
        Mathf.Clamp(transform.position.y + Random.Range(-20, 20), clampMin, clampMax));
}
else
{
    if (idle) { break; }

    if (WalkTowards(CurrentWanderPoint))
    {
        StartCoroutine(WanderIdle());
    }
}
```

Figur 10 - Kode i FixedUpdate, i Prey.cs, der håndterer tilfældig omvandring

Den sidste del af koden, under Wander casen i switch-statementet, håndterer hvordan dyret rent faktisk vandre tilfældigt rundt. Denne kodestump kan ses på billedet ovenover (Figur 10). Det første der tjekkes, er om variablen *CurrentWanderPoint* er lig med Vector3.zero. CurrentWanderPoint indeholder koordinaterne på et tilfældig valgt punk, som dyret bør bevæge sig imod. Når dyret er nået dette punkt, sættes CurrentWanderPoint til Vector3.zero, hvilket er hvad if-statementet tjekker. Det vil sige at betingelsen, der tjekkes, er hvorvidt dyret er færdig med at gå imod sit nuværende tilfældige vandrepunkt. Hvis dette er tilfældet, skal CurrentWanderPoint sættes til et nyt tilfældigt punkt. Dette nye punkt findes ved at vælge et tilfældigt punkt, inden for en 20 unit radius af dyret. Denne værdi bliver dog *clampet*, eller klemt, så værdien altid ligger mellem to specifikke værdier. Disse værdier er maksimumslængden, som dyrene burde vandre fra midten, men dog 10% tættere på midten, end denne maksimumslængde. Disse 10% er tilføjet, så dyrene ikke bare opholder sig på kanten af spillebrættet. Hvis dyret ikke er nået til CurrentWanderPoint, burde dyret fortsætte mod punktet, medmindre dyret er i gang med at holde en pause. Når dyret endelig når CurrentWanderPoint, skal dyret holde en pause.

Optimering

Da dette program indeholder mange forskellige *GameObjects*, altså objekter, der alle kører kode samtidig, var det meget vigtigt at tænke over hvordan programmet kunne optimeres.

Hvert dyr, som hver er tildelt et GameObject, skal konstant have styr på om der er mad i nærheden og om der er en valid partner i nærheden. For at finde ud af dette, skal dyret først finde et array af alle dyr og alt mad, og derefter gå igennem disse arrays, for at tjekke om objekterne i arrayene opfylder betingelserne. Disse arrays findes ved at bruge en indbygget funktion i Unity, kaldt *FindObjectsOfType*. FindObjectsOfType funktionen, returnerer alle GameObjects, der indeholder et specifik komponent. Problemet med denne funktion, er dog at den er langsom, så det burde undgås at kører funktionen ofte.

```
//Køres 60 gange i sekundet
void FixedUpdate()
{
    if (CheckFood(FindObjectsOfType<Food>()))
    {
        //Kode
    }

    if (CheckAnimalPartner(FindObjectsOfType<Prey>()))
    {
        //Kode
    }
}
```

Figur 11 - Pseudokode, der viser den originale tilgang

I tidlige versioner af programmet, kørte hvert dyr FindObjectsOfType funktionen op til 120 gange i sekundet. Inde i FixedUpdate funktionen, som kørers 60 gange i sekundet, blev der nemlig tjekket for både mad og potentielle partnere. Denne tilgang er illustreret med pseudokode på det overstående billede (Figur 11). Dette er selvfølgelig ikke optimalt, så det blev besluttet at kigge på måder at optimere denne tilgang.

Den første løsning, der blev fundet, var at gøre så hvert dyr kun kører FindObjectsOfType funktionen to gange i sekundet på dyr. Dette er opnået ved at kører en coroutine, der opdaterer et par arrays. Når et dyr efterfølgende skal bruge information om f.eks. mad, skal dyret blot bruge en af arrayene. Coroutine loopet bliver startet, når classen initialiseres. Tilgangen kan ses illustreret med pseudokode, på figur 12. Dette er dog stadig ikke den mest optimale løsning, da disse FindObjectsOfType funktioners antal af kald, stiger med antallet af dyr. På grund af dette blev den sidste tilgang udviklet.

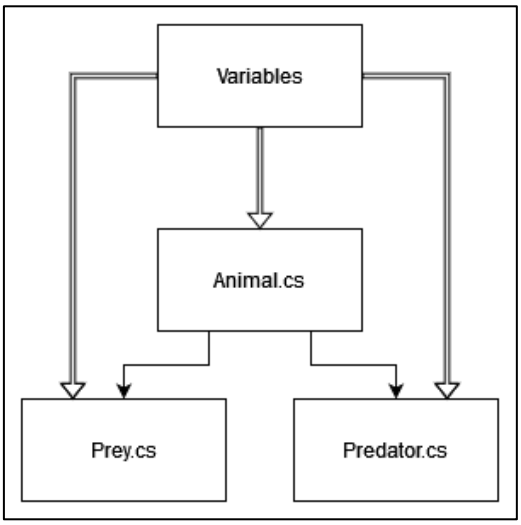
```
void Start()
{
    StartCoroutine(UpdateArrays());
}

Food[] foodArray;
Prey[] animalArray;
IEnumerator UpdateArrays()
{
    foodArray = FindObjectsOfType<Food>();
    animalArray = FindObjectsOfType<Prey>();
    yield return new WaitForSeconds(1);
    StartCoroutine(UpdateArrays());
}

//Køres 60 gange i sekundet
void FixedUpdate()
{
    if (CheckFood(foodArray))
    {
        //Kode
    }

    if (CheckAnimalPartner(animalArray))
    {
        //Kode
    }
}
```

Figur 12 - Pseudokode, der viser den opdateret, men ikke endelige, tilgang

```
public class Variables : MonoBehaviour
{
    private void Awake()
    {
        //Starter async loop, der opdaterer globale lister
        StartCoroutine(UpdateArrays());
    }

    //Opdaterer de arrays, der holder styr på alle predators, prey og mad
    public Predator[] AllPredators;
    public Prey[] AllPrey;
    public Food[] AllFood;
    IEnumerator UpdateArrays()
    {
        AllPredators = FindObjectsOfType<Predator>();
        AllPrey = FindObjectsOfType<Prey>();
        AllFood = FindObjectsOfType<Food>();
        yield return new WaitForSeconds(1);
        StartCoroutine(UpdateArrays());
    }
}
```

Figur 13 - Et billede af den reelle kode og endelige tilgang

Dette billede (Figur 13) viser den endelige tilgang. Tilgangen her minder meget om sidste tilgang, men med én drastisk forskel. Her bliver koden nemlig kørt i et helt nyt script, ved navn *Variables.cs*. Dette script kører uafhængigt af dyre scriptsene. Scripts, der skal bruge en liste af f.eks. rovdyr, kan dog stadig få den liste, ved at refererer til Variables classen (Figur 14). På den måde stiger antallet af FindObjectsOfType kald ikke, med antallet af dyr.



Figur 14 - Diagram over den endelige tilgang

Test af programmet

I dette afsnit, vil der testes og undersøges hvor stor en effekt optimeringen havde på programmet. Dette gøres ved at omskrive programmet, så programmet bruger den originale tilgang, og notere, hvordan programmet kører. Derefter gøres det samme, men med den nuværende tilgang. For at undersøge hvordan spillet kører, vil Unity's indbygget *profiler* bruges. Unity's profiler kan bruges til at undersøge hvilket effekt forskellige dele af et program, har på hvordan programmet kører.

Data

Denne data (Tabel 1) er taget fra den 20'ene frame, efter programmets opstart.

	Original tilgang	**Optimeret tilgang**	**Forbedring**
%CPU brugt på hele programmet	94,5%	58,9%	35,6%
%CPU brugt på GetFoodInSight	26,8%	0,1%	26,7%
%CPU brugt på GetPredatorsInSight	25,2%	0,1%	25,1%
Millisekunder per frame	732,33 ms	9,99 ms	722,34 ms (98,63%)

Tabel 1 - Tabel med data over resultaterne af optimeringen

Se bilag 1 og bilag 2, for screenshots af dataet i Unity profileren.

Resultater

Det er tydeligt, at optimeringen har gjort en stor forskel. De funktioner, som blev påvirket af ændringerne, gik fra hver at tage en fjerdedel af CPU'ens kræfter, til kun at tage 0,1%. Tiden mellem hver frame, blev også reduceret med over 98%. Dette beviser altså alt sammen, at optimeringen har virket.

Refleksion

Dette forløb har været en læringsrig proces. Jeg har blandt andet lært om god kodestruktur. Specifikt har jeg lært at skære ned for if-statements og skrue op for brugen af return-statements. I undervisningen har vi lært, at hvis samme kode står to steder, er der nok en måde at omskrive det, så koden kun står en enkelt gang. Dette tog jeg til hjerte under projektet, hvilket blandt andet er en af grundene til valget af en nedarvningsfokuseret kodestruktur. Så mange funktioner som muligt, er blevet flyttet fra Prey og Predator, over i Animal. Dette har, for mig, forstærket idéen om at lave genbrugelig og ren kode.

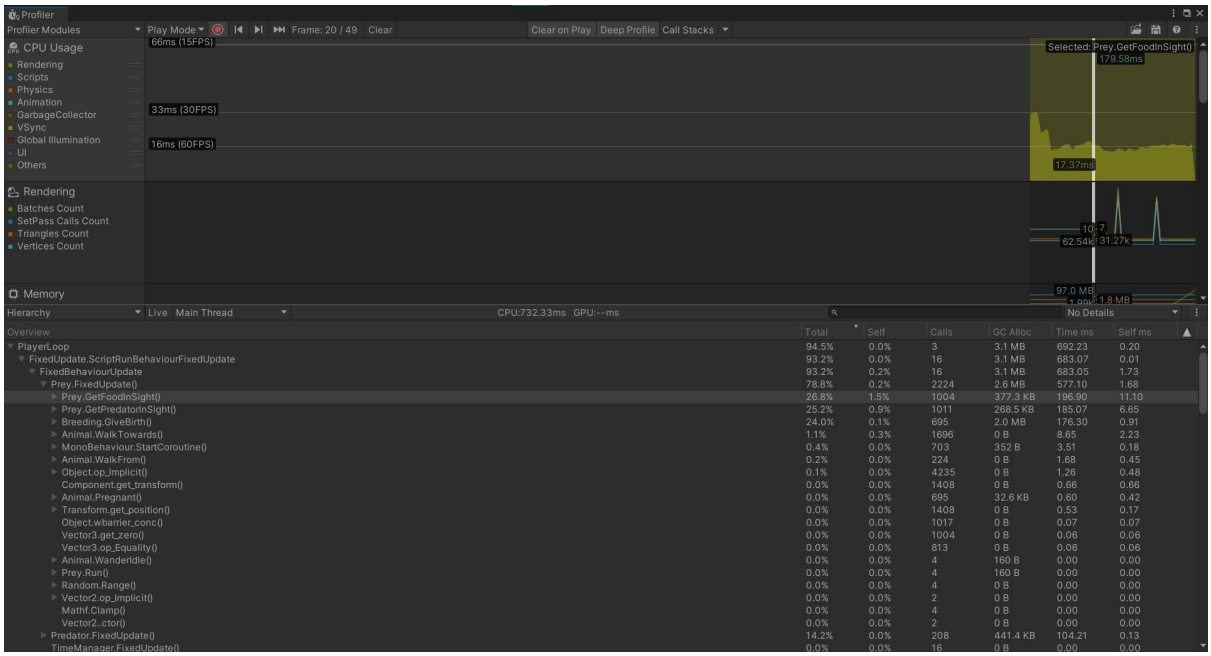
Jeg har også lært om vigtigheden af optimering. Da dette program stadig er en prototype, var jeg originalt skeptisk over, om det var værd at dedikere tid til optimering. Ud fra testresultaterne, er det tydeligt, at selv i prototyper, giver det mening at tage højde for effektiviteten af projektets kode.

Konklusion

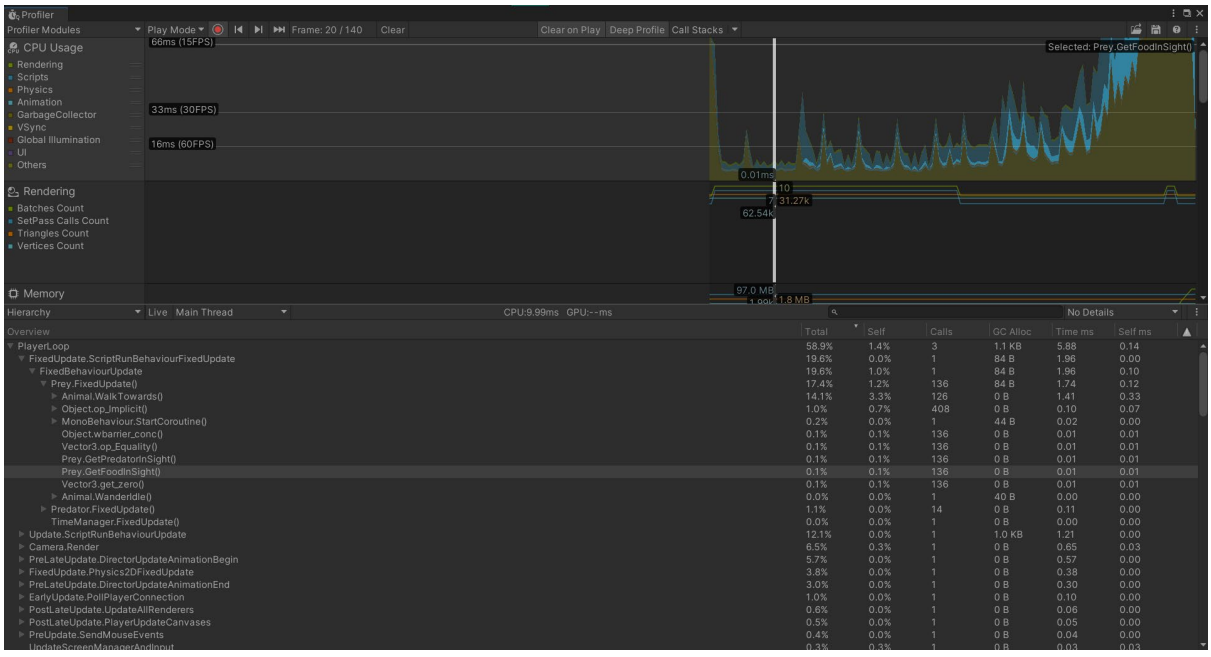
Under dette projekt, er der blevet udviklet et kunstprodukt, med formål om at visualiserer evolution på en underholdende, overskuelig måde. Dette blev gjort ved hjælp af programmet Unity, hvori koden for byttedyr, rovdyr og meget andet, blev produceret. Programmet, der blev udviklet, visualiserer i hvert fald evolution på en underholdende måde. Spørgsmålet er dog til hvilket grad visualiseringen er overskuelig. Der var originalt snak om at lave en kurve, der viste dyrenes udvikling over tid, men denne feature blev ikke nået på grund af tidsmangel. Brugerne kan dog stadig få et meget generelt overblik, ved at bruge gennemsnitsknapperne, men dette er muligvis mindre detalje end ønsket. Udover det, opfølger programmet dog problemformuleringen. Vi har altså haft succes med at skabe et program, der kan visualisere evolution på en underholdende måde.

Bilag

Bilag 1 - Screenshot af den originale tilgangs effektivitet



Bilag 2 - Screenshot af den endelige tilgangs effektivitet



Bilag 3 - Kildekode

Kildekoden er vedhæftet i zip-filen, i mappen ”Kildekode”

Bilag 4 - Build

Et kørbart build er vedhæftet i zip-filen, i mappen ”Build”