Projektbeskrivning

Fishfishfish

2020-03-22

Projektmedlemmar:
Andreas Franked <andfr210@student.liu.se>
Erik Birgersson <eribi813@student.liu.se>

Handledare:
Handledare <erima882@student.liu.se>

Innehåll

Projektplan

1. Introduktion till projektet

Idén, “Fish Fish Fish”, är baserad på spelet Feeding Frenzy där man spelar som en fisk som ska äta så många andra fiskar som möjligt. Man kan dock bara äta fiskar som är mindre än spelaren, och man måste undvika fiskar som är större.

Feeding Frenzy, spelet som Fishfishfish är baserat på. (Notera att bilden ovan inte är på vårt projekt)

2. Ytterligare bakgrundsinformation

- Spelaren är en fisk som följer efter musen
- Större fiskar äter mindre fiskar
- Om man klickar på vänster musknapp skjuts spelaren fram mot musen

Följande video visar hur Feeding Frenzy ser ut: https://www.youtube.com/watch?v=RroKSbmTcxs

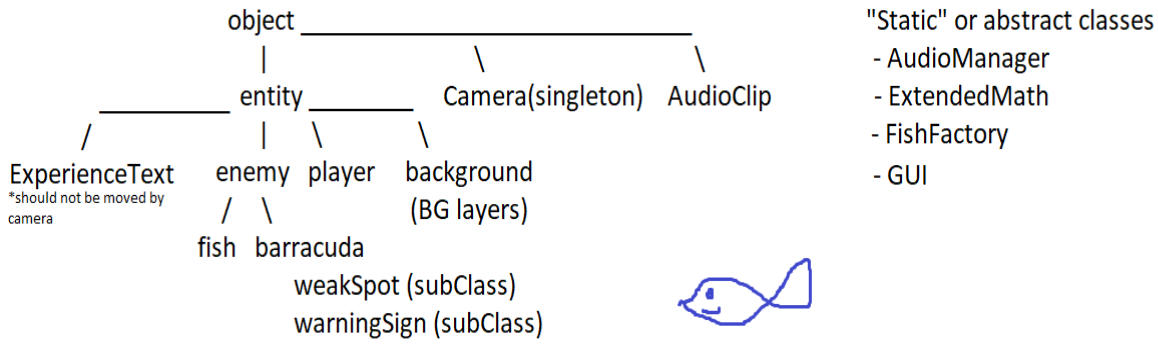
3. Milstolpar

#	Beskrivning
1	Man kan läsa muspekarens position på skärmen
2	Det finns en spelare som för tillfället är en cirkel
3	Spelaren kan följa muspekaren “smooth”, d.v.s med hastigheten som en funktion av avståndet till musen
4	Det finns en klass (Universe) som håller referenser till alla objekt som ska flyttas av kameran (instanser av Fish, bakgrund etc). Med t.ex ArrayList<Entity> … där alla instanser av fiskar, bakgrundsbilder, etc, har Entity som superklass. Testa genom att flytta alla objekt.
5	Det finns en kamera som följer efter spelaren (genom att flytta alla objekt i universe). För att testa så att camera-klassen fungerar kan exempelvis ett rutnät av cirklar skapas, för att se så att dessa rör sig i relation till spelaren/skärmen.
6	Spelaren kan dasha/thrusta. När man klickar på musen (LMB) skjuts spelaren framåt mot musens riktning.
7	Klassen Enemy finns, som håller egenskaper hos fiskar/annat.
8	Klassen Fish finns, som är den mest grundläggande typen av fiende. De kan röra sig från sida till sida och har slumpade storlekar.
9	Det finns en FishFactory klass som skapar fiskar och lägger till dem i universe.
10	Det finns kollision för Enemy och Player. Dessa har två olika kollisionsboxar, en för kroppen och en för munnen.
11	En fisk kan endast dö om den kommer i kontakt med en annan fisks mun (eftersom fiskar märkligt nog inte kan äta med fenorna).

12	(Lägg till fiskar och testa så att allt fungerar.)
13	Kameran ser mindre hackig ut (med acceleration)
14	Det finns ett poängsystem som håller koll på hur mycket xp/poäng spelaren har. Större fiskar ger mer xp.
15	När spelaren har tillräckligt mycket xp så växer spelaren ett steg.
16	När man växer ett steg ökar mängden xp som behövs för nästa steg.
17	Det finns bilder för alla fiskar och bakgrunden.
18	“Barracudan” finns i spelet, en speciell fisk som är större än alla andra fiskar. Den åker snabbt från en sida av skärmen till den andra, och äter alla fiskar i sin väg.
19	Det finns ett blinkande varningstecken där barracudan kommer ifrån.
20	Man kan bita barracudan i svansen.
21	Efter tre bett dör barracudan.
22	Om man biter barracudan i svansen stannar den upp i några sekunder och börjar sedan jaga spelaren.
23	Det finns ljud i spelet.
24	Fiskar är animerade: när de dör minskar de i storlek istället för att försvinna direkt. Likadant ökar spelaren i storlek när man växer till nästa steg.
25	Det finns en kant runt banan. Kameran stannar när man kommer till kanten.
26	Det finns en meny med en ”Play”-knapp, en ”Quit”-knapp och, vid behov, en ”Settings”-knapp

27	När fiskar blir ätna blinkar de vitt i en frame och rendertråden (eller hela spelet) pausas i några få millisekunder. Spelet pausas bara snabbt när spelaren äter fiskarna, och inte när fiskarna äter varann.
28	Parallax-scrolling finns, d.v.s vissa bakgrundsbilder ska röra sig långsammare än andra för att simulera djup
29	Det finns power-ups

4. Övriga implementationsförberedelser



Bilden ovan visar en ungefärlig struktur på koden till projektet.
Ljud: AudioManager klass som kan spela upp ljudklipp, och en AudioClip klass som laddar in ljudklipp att spela.
Bilder: BufferedImage
(notera: det finns inte längre singletons i projektet – så kameran är inte en singleton)

5. Utveckling och samarbete

En av oss siktar på en femma, och den andra är nöjd med att bli godkänd. Hur vi hanterar detta är att personen som siktar på en femma tänker jobba lite mer. Båda vill bli klara till deadline. Vi tänker framför allt jobba på projektet då vi har tid över från andra kurser, till exempel på helger. Vi tänker gå på alla resurstillfällen om inte sjukdom eller liknande ger förhinder.

Ingen av oss kommer vara ofrånkomligt upptagen. Det enda fallet där någon inte kommer kunna jobba med projektet kommer antagligen vara ifall någon råkar bli sjuk,

vilket är en godtarbar anledning att missa ett tillfälle. Vi känner inte att vi behöver sätta upp en tidslinje då vi kommer jobba mycket för att bli klara i god tid i vilket fall, och en tidslinje är därmed inte nödvändig.

Vi kommer oftast arbeta tillsammans (på samma dator), men personen som siktar på en femma kommer jobba lite extra hemma, och sedan förklara koden så att båda förstår. Vi byter mellan vem som skriver kod ungefär varje halvtimme så att båda jobbar.

Vi båda känner att det kommer bli roligt att jobba med projektet.

6. Implementationsbeskrivning

Fishfishfish

Point2D

Point2D är en klass som vi skapade för att lagra par av variabler och innehåller funktioner som exempelvis ListAverage som tar en Collection av Point2D(x,y) och ger average(x), average(y) som en ny point.

Fish

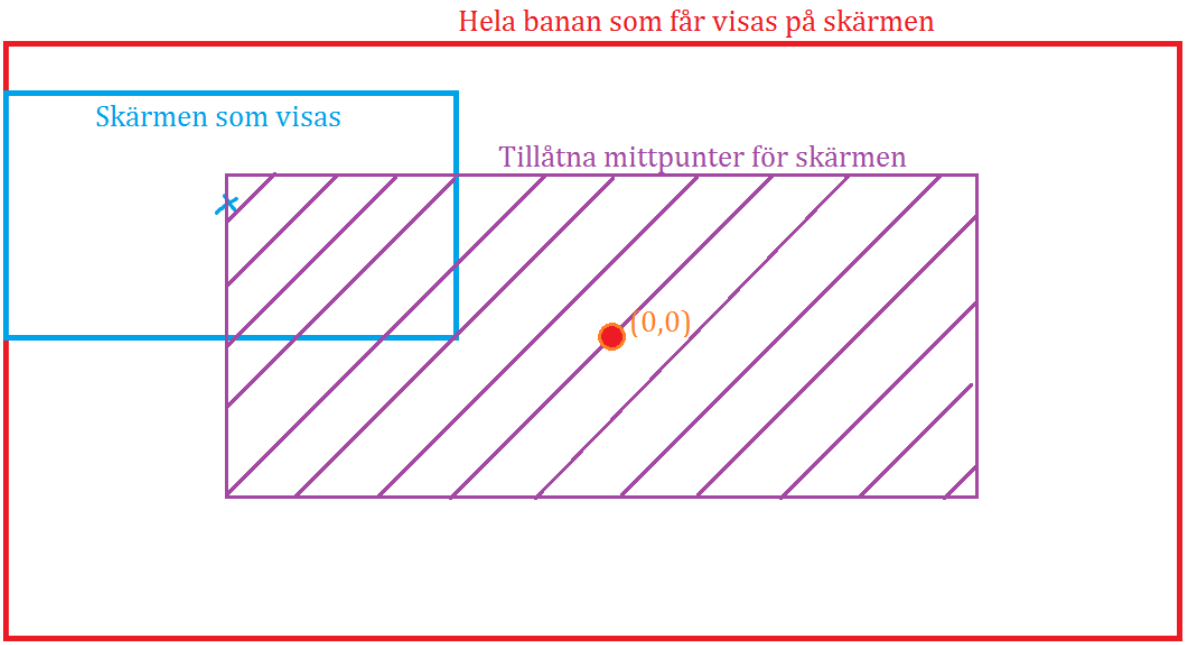
Fish är klassen för alla sorters fiskar som dyker upp på skärmen, inklusive spelaren. Alla fiskar, bland annat spelaren och barracudan, är en subklass av Fish och ärver därmed alla metoder som sköter kollision o.s.v. Det finns även en klass som heter FishFactory som med hjälp av getFish(type) skapar objekt av subklasserna till Fish beroende på vilken typ av fisk det är.

Kamerasystemet

För att flytta kameran så måste man egentligen flytta alla objekt på skärmen i motsatt riktning för att det ska se ut som att det är kameran som flyttas. Detta görs genom att klassen Universe, som lagrar alla objekt av typen Entity (d.v.s dess subklasser) med de entities som inte alltid ska flyttas som undantag, såsom Player.

För att jämna ut kamerans rörelser finns det en CopyOnWriteArrayList som ständigt tar in Point2D-objekt som håller spelarens nuvarande hastighet, och fungerar i princip som en variant av Queue då det första objektet som lades in i listan tas bort när storleken på ArrayListen överstiger ett specifikt värde (first-in first-out / fifo). Egentligen skulle

man kunna använda en ConcurrentLinkedQueue men CopyOnWriteArrayList är lite snabbare med att läsa värden, vilket görs upp till ett hundratal gånger per frame (ifall detta har någon substantiell påverkan på performance är dock oklart). Anledningen till att CopyOnWriteArrayList används istället för en vanlig ArrayList är för att den behöver vara thread-safe då värden läggs till i en scheduler (d.v.s parallellt med att värdena itereras i en for-loop). Syftet med listan är att spara spelarens hastighet de senaste *n* frames. Kamerans hastighet sätts till medelvärdet av denna lista för att uppnå en jämn och "smooth" rörelse. Dessutom rör sig kameran konstant (linjärt) så att spelaren alltid eventuellt kommer tillbaka till mitten av skärmen.



Skärmens kant funkar genom att kameran sparar hur långt den har färdats från punkten (0, 0). För att faktiskt kunna stoppa kameran från att färdas räknar vi först ut gränserna för x och y där mitten av skärmen måste stanna. Om, till exempel, spelarens x-hastighet är högre än 0 (spelaren färdas åt höger) och kamerans position når gränsen för hur långt den får åka så sätts hastigheten som skickas in i den ovannämnda listan till 0. Detta gör så att kameran "tror" att spelaren alltid är inom kamerazonen (den lila zonen - se bilden ovan). Dessutom slutar kamerans konstanta/linjära rörelse då spelaren är i den zonen.

6.1. Milstolpar

Fetstilta milstolpar är avklarade.

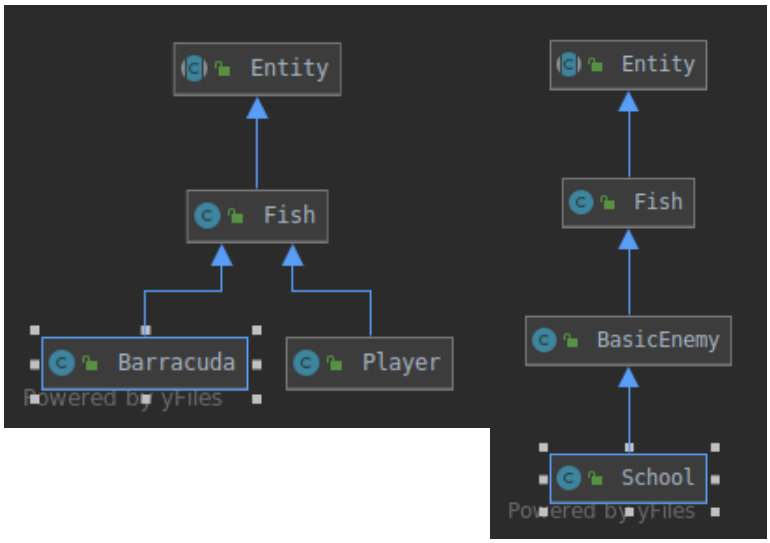
#	Beskrivning
1	**Man kan läsa muspekarens position på skärmen**
2	**Det finns en spelare som för tillfället är en cirkel**
3	**Spelaren kan följa muspekaren “smooth”, d.v.s med hastigheten som en funktion av avståndet till musen**
4	**Det finns en klass (Universe) som håller referenser till alla objekt som ska flyttas av kameran (instanser av Fish, bakgrund etc). Med t.ex ArrayList<Entity> … där alla instanser av fiskar, bakgrundsbilder, etc, har Entity som superklass. Testa genom att flytta alla objekt.**
5	**Det finns en kamera som följer efter spelaren (genom att flytta alla objekt i universe). För att testa så att camera-klassen fungerar kan exempelvis ett rutnät av cirklar skapas, för att se så att dessa rör sig i relation till spelaren/skärmen.**
6	**Spelaren kan dasha/thrusta. När man klickar på musen (LMB) skjuts spelaren framåt mot musens riktning.**
7	**Klassen Enemy finns, som håller egenskaper hos fiskar/annat.**
8	**Klassen Fish finns, som är den mest grundläggande typen av fiende. De kan röra sig från sida till sida och har slumpade storlekar.**
9	**Det finns en FishFactory klass som skapar fiskar och lägger till dem i universe.**
10	**Det finns kollision för Enemy och Player. Dessa har två olika kollisionsboxar, en för kroppen och en för munnen.**
11	**En fisk kan endast dö om den kommer i kontakt med en annan fisks mun (eftersom fiskar märkligt nog inte kan äta med fenorna).**
12	**(Lägg till fiskar och testa så att allt fungerar.)**
13	**Kameran ser mindre hackig ut (med acceleration)**
14	**Det finns ett poängsystem som håller koll på hur mycket xp/poäng**

	spelaren har. Större fiskar ger mer xp.
15	**När spelaren har tillräckligt mycket xp så växer spelaren ett steg.**
16	**När man växer ett steg ökar mängden xp som behövs för nästa steg.**
17	**Det finns bilder för alla fiskar och bakgrunden.**
18	**“Barracudan” finns i spelet, en speciell fisk som är större än alla andra fiskar. Den åker snabbt från en sida av skärmen till den andra, och äter alla fiskar i sin väg.**
19	Det finns ett blinkande varningstecken där barracudan kommer ifrån.
20	**Man kan bita barracudan i svansen.**
21	**Efter tre bett dör barracudan.**
22	**Om man biter barracudan i svansen stannar den upp i några sekunder och börjar sedan jaga spelaren.**
23	**Det finns ljud i spelet.**
24	Fiskar är animerade: när de dör minskar de i storlek istället för att försvinna direkt. Likadant ökar spelaren i storlek när man växer till nästa steg.
25	**Det finns en kant runt banan. Kameran stannar när man kommer till kanten.**
26	Det finns en meny med en ”Play”-knapp, en ”Quit”-knapp och, vid behov, en ”Settings”-knapp
27	När fiskar blir ätna blinkar de vitt i en frame och rendertråden (eller hela spelet) pausas i några få millisekunder. Spelet pausas bara snabbt när spelaren äter fiskarna, och inte när fiskarna äter varann.

28	Parallax-scrolling finns, d.v.s vissa bakgrundsbilder ska röra sig långsammare än andra för att simulera djup
29	Det finns power-ups

6.2. Dokumentation för programstruktur, med UML-diagram

Den abstrakta Entity-klassen hanterar position, storlek och bilden, om det finns en, för en entitet tex en barracuda eller spelaren. Entity är en superklass och den kallas i Fish konstruktorn med super(position, size) för att ärva position och storlek på objeketet.



Fish klassen är en subklass av Entity och ärver därmed metoder från Entity. Den metod som overridas av Fish är Render metoden för att se vilket håll fiskens bild ska vara vänt åt. Render metoden i Fish ritar också ut en vit rektangel om en bild saknas för objektet, om det nu händer. Den sköter också om spelet är i Debug mode och ritar ut kollisions rektanglar för fiskarna i spelet. Kollisions rektanglarna innebär då fiskens mun, som ritas ut rött, och kroppen som ritas ut grönt. Detta är för att se så att allt fungerar som det ska med kollision. Fish klassen hanterar då hastighet, level, kollision samt vilka fiskar som finns i spelet, här räknas spelaren med för Player klassen är en subklass av Fish.

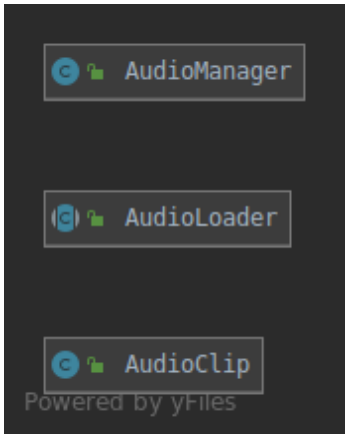
Player klassen är en subklass till Fish och ärver därmed metoder från både Fish och Entity klasserna. Player klassen hanterar allt som rör spelaren tex position, storlek av

spelare fisken, poäng, erfarenhetspoäng och vad som behövs till nästa level. Player klassen har en konstruktor och i konstruktorn kallas konstruktorn i Fish genom super(position, size, velocity, level, true), för att ärva position, storlek, hastighet, level på objeketet. Sist om den ska läggas in till universiumet(true eller false). När erfarenhetspoängen är lika eller mer än gränsen till nästa level kallas metoden levelUp. I levelUp ökas spelarens level med 1, om den inte har nått maxgränsen level 4. Sedan fördubblas spelarens storlek samt kollisions rektanglarna med. Sedan om spelaren dör så kallas metoden resetStats som återställer alla variabler till ursprungsvärdena.

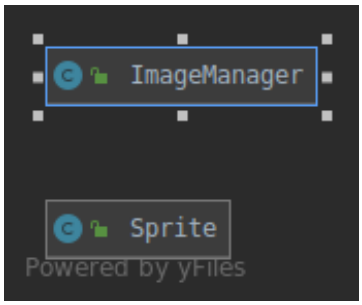
Barracuda klassen är också en subklass till Fish, och ärver därmed metoder från både Fish och Entity klasserna. Samma som Player klassen. Barracuda klassen har en konstruktor och i konstruktorn kallas konstruktorn i Fish genom super(position, size, velocity, level, true), för att ärva position, storlek, hastighet, level på objeketet. Sist om den ska läggas in till universiumet(true eller false). Det enda som behövs i Barracuda är att setSize overridas för barracuda objeket storlek är rätt så mycket större än de andra fiskarna.

BasicEnemy är den vanligaste typen av fiende. De flesta funktionerna som krävs för att BasicEnemy ska fungera anropas från dess superklass, Fish. Det enda som behövs i BasicEnemy är en funktion som sätter egenskaperna (vilken bild som ska visas, storlek på fiskarna, collision boxes, offsets för colliders etc) för fiskar av olika nivåer. Den minsta fisken använder till exempel en bild som hämtas via id:t “SMALL FISH”.

Eftersom School är grupper av typen BasicEnemy som är level 1 så är det lämpligt för School att extenda BasicEnemy. School innehåller en lista med alla fiskar i det stimmet, samt en funktion som anropas i konstruktorn som placerar ut varje individuell fisk i stimmet. Detta görs genom att sätta ut dem i rutnät, med exakt en fisk per rad. Om man vill ha en grupp med exempelvis fem fiskar så itererar den genom fem rader och slumpar vilken kolumn de hamnar på, men istället för att använda en till for-loop för kolumnerna så slumpas det genom att vi tar Math.random() * count, där count är antalet fiskar som ska finnas. Detta gör så att det alltid kommer finnas exakt så många fiskar, till skillnad om man exempelvis skulle ha en nästlad for-loop och en viss chans för en fisk att skapas på varje enskilda plats. Medan fiskarna skapas så slumpas även två variabler, xDisplacement och yDisplacement som slumpmässigt flyttar fiskarna några pixlar för att det inte ska se ut som att fiskarna är placerade i ett perfekt rutnät.



För att spela upp ljud används tre klasser, AudioLoader, AudioManager, och AudioClip. **AudioLoader** innehåller likt ImageManager en hashmap med ett namn/id som nyckel och ett AudioClip som nyckel så att man enkelt kan hämta ut ett AudioClip. **AudioClip** tar in antingen ett filnamn eller en array med filnamn. Om AudioClip skapas med en array av filnamn så slumpas en fil då man spelar upp ljudklippet (detta görs för att ge variation så att inte samma ljud spelas upp varje gång man t.ex äter en fisk). AudioClip har play() och loop(). Play() spelar helt enkelt upp en ljudfil med hjälp av AudioManager, och loop() räknar ut längden av ljudklippet för att kontinuerligt spela upp ljudfilen tills man stoppar den genom att sätta loop-fältet till false. Detta är bra för musik och liknande. **AudioManager** sköter det tunga arbetet då det är denna klass som faktiskt laddar in och spelar klippen. En viktig sak att notera är att en del av denna kod inte är skriven av oss, utan hittades online. Den har modifierats för att nå våra behov, och källan har angetts längst upp i filen. För att kunna spela upp flera ljudfiler samtidigt använder klassen ArrayLists av typerna AudioFormat, Dataline.Info, etc, så att den kan hålla reda på flera ljudfiler/klipp åt gången.



För bilderna används en HashMap med en sträng som nyckel (namnet på bilden, till exempel “SMALL FISH” för den minsta fisken) och ett Sprite-objekt som värde. Sprite-klassen innehåller bilden i sig (BufferedImage) samt filvägen till bilden. I ImageManager finns (bortsett från getters/setters) metoderna initImages, loadImages och getResourceURI. **InitImages** lägger till värden i hashmappen där man anger ett namn/id och skapar en ny sprite där bildstorlek och filväg anges, vilket görs för varje bild. **LoadImages** sätter BufferedImage för varje sprite med hjälp av dess filväg. Den itererar genom varje objekt av typen Sprite i hashmappen och sätter dess bufferedImage-fält till vad som fås ut av getResourceURI-metoden. **GetResourceURI**

tar in en sökväg och använder ImageIO för att hämta filen som en BufferedImage. För att sedan rita en bild kan man helt enkelt anropa g.drawImage(sprite, position, storlek) där *g* är en instans av awt.Graphics. Detta anropet finns endast i Entity-klassen som är en superklass till alla andra klasser som behöver rita bilder. I till exempel Player-klassen så behöver bara sprite-fältet sättas till ImageManager.getSpriteHashMap().get(“PLAYER”). Om en bild saknas (om sprite är null) för en instans av typen Fish eller dess subklasser så ritas istället en vit rektangel. Designmönstret singleton används för att det endast behöver finnas en instans av ImageManager.

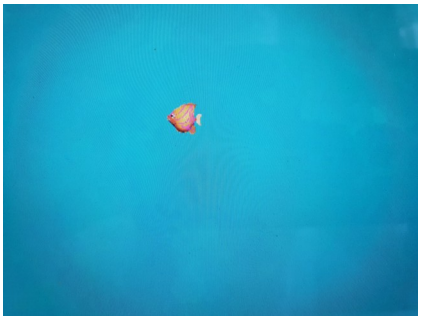
Intressanta lösning(ar)

Det finns separata colliders för fiskarnas munnar och kroppar. Detta är för att spelet ska bli lite mer förutsägbart, då man inte vanligvis förväntas dö om man simmar in i en större fisks bakfena eller ett hörn som inte matchar bilden helt och hållet. Därför fungerar kollisionen så att den först kollar om en annan fisks body-collider kolliderar med denna fisks mun. Om den gör det så jämför den levlarna på båda fiskarna samt kollar om någon av fiskarna är spelaren. För att effektivisera jämföringarna av varje fisk på skärmen så skulle man kunna implementera chunks, där man endast kollar på enstaka sektioner av skärmen som innehåller fiskar, men detta är endast relevant om man ska öka storleken på banan eller göra den oändligt stor, vilket inte är något vi har planer för att implementera.

7. Användarmanual

Allmän spelinformation:

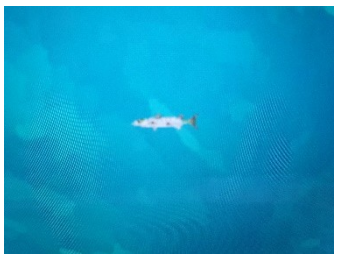
Spelaren spelar som en orange fisk.



Målet med spelet är att nå max level, vilket är level 4. När man har nått level 4 så visas ett vinstmeddelande på skärmen “YOU WIN!”. Sedan stängs spelet av efter 3 sekunder.

För att få erfarenhetspoäng och gå upp i levlar så måste spelaren äta andra fiskar som simmar runt i spelet. När man har ätit tillräckligt med fiskar så går man upp en level, som vi nämnde tidigare, men man växer också en storlek. Man kan bara äta fiskar som är samma level som en själv eller lägre level. Det finns flera olika sorters fiskar i spelet. De fiskar som finns är följande:

Small fish:



Small fish är level 1 fiskar och de är dessa fiskar man äter i början av spelet. De ger 300 erfarenhetspoäng när man äter dem. De kan inte heller äta upp spelare för de är lvl 1 och spelaren börjar på lvl 1.

Medium fish:

Medium fish är level 2 fiskar och de ger 848 erfarenhetspoäng när man äter dem. Dessa fiskar kan äta upp spelare om spelaren är lvl 1. Spelaren kan äta dessa fiskar om spelaren är lvl 2.

Large fish:

Large fish är level 3 fiskar och de ger 1558 erfarenhetspoäng när man äter dem.
Spelaren kan äta dessa fiskar om spelare är lvl 3.

Barracuda:

Barracudafiskar är level 15 fiskar och de ger 17428 erfarenhetspoäng när man äter dem. För att kunna äta dem så måste man bita dem i svansen 3 gånger. Första gången man biter dem så står de helt stilla i 3 sekunder.
Sedan så kommer den jaga en i 5 sekunder.
Om man biter barracudas svans 2 gånger till, totalt 3 gånger, så dör den och man får erfarenhetspoäng.

School:

School är ett stim av flera small fish, mellan 2-6st. De åker i samma riktning och håller ihop.

Kontroller information:
För att styra spelare fisken så använder man musen och pekar med den vart man vill åka till. Om musen är vänster om fisken så pekar fiskens mun dit och vice versa.
Om man vänsterklickar på ett ställe så får man en thrust dit.