Chaos With a Chance of Asteroids

Slutrapport för grupp 8

Operating Systems & Process-Oriented Programming 2016
(1DT096)

Anton Larsson 931119-6456,
Axel Poromaa 931218-1754,
Hampus Falk 930217-1898,
Karl Johansson 860728-0099,
Marcus Andersson 910202-4735,
Peter Berglund 900927-0191

June 2, 2016

Innehållsförteckning

1 Inledning **3**

2 Översikt av systemet **4**
2.1 Concurrency-modell 4
2.2 Intern kommunikation i back-end 4
2.3 Kommunikation mellan front- och back-end 4
2.4 Intern kommunikation front-end 5
2.5 Gränssnitt 5
2.6 Systemarkitektur 6
2.6.1 Server 6
2.6.2 Session 7
2.6.3 AsteroidManager 7
2.6.4 Asteroids 7
2.6.5 ListenerManager 8
2.6.6 Listener 8
2.6.7 CollisionManager 8
2.6.8 Client 9

3 Implementation **10**
3.1 Val av programmeringsspråk 10
3.2 Centrala algoritmer 10
3.2.1 Initialisering eller ny spelare 10
3.2.2 Ett steg i spelet 10

4 Slutsats **12**

A Appendix **14**
A.1 Utveckling 14
A.2 Installation 14
A.2.1 Kompilera och köra 14
A.2.2 Tester 14
A.2.3 Dokumentation 14
A.3 Reflektioner 15
A.3.1 Sökta lärdomar 15
A.3.2 Oväntade svårigheter 15

1 Inledning

De flesta har förmodligen någon gång hört talas om det gamla arkadspelet Asteroids. Asteroids är ett mycket enkelt spel i två dimensioner där en spelare manövrerar en rymdfarkost och skjuter ner och undviker asteroider och diverse fiender. Produkten som valts att utveckla är en concurrent version av detta spel. Vår spin är helt enkelt att flera spelare kan spela mot varandra i realtid, över nätverk. Grundtanken med spelet kommer vara att spelarna ska kunna skjuta sönder både asteroiderna och de andra spelarna för att öka sin chans för överlevnad. Dessutom tänker vi, med tanke på kursen, baka in lite extra överflödig concurrency. Exempelvis så kommer varje asteroid att vara sin egen isolerade process som regelbundet måste kommunicera med spelvärlden.

Majoriteten av koden kommer att skrivas i Google's hårdvarunära språk Golang som valts mycket tack vare dess utmärkta stöd för concurrency men också av ren nyfikenhet. För enkelhetens skull visualiseras spelet i 2D med hjälp utav spelmotorn Unity.

2 Översikt av systemet

2.1 Concurrency-modell

Programmeringsspråket som används i projektets back-end är Golang och Golang använder sig av concurrency-modellen CSP[2], som står för Communicating Sequential Processes. CSP använder lättviktiga trådar som kommunicerar genom att skicka meddelanden via kanaler. CSP liknar till stor del aktörsmodellen som används i programmeringsspråket Erlang[3].

I Golang kallas de lättviktiga processerna som skapas för Goroutines och de skiljer sig från de trådar som används i exempelvis C[4]. En Goroutine är inte alls lika krävande i avseende på både minne och systemtid[4].

Skillnaden med minnet och systemtiden beror på att en Goroutine inte använder sig av ett delat minne utan istället kommunicerar genom att skicka meddelanden via synkroniserade channels. Hur en channel fungerar beskrivs närmare i avsnittet om intern kommunikation i back-end.

Dessutom är tiden för att skapa en ny Goroutine kort i jämförelse med andra modeller. Detta möjliggör en snabb concurrency och att det blir lätt att skapa skalbara projekt.

I front-end används spelmotorn Unity, med kod skriven i C#. Concurrency i front-end utgörs utav två standardtrådar. Den ena läser av datan som skickas från back-end och den andra ritar upp den mottagna datan.

2.2 Intern kommunikation i back-end

I back-end sker kommunikationen mellan de olika modulerna och Goroutines på två olika sätt. Det första är via synkroniserade channels, vilka är blockerande i de fall där det inte finns en mottagare eller sändare redo. Ifall en Goroutine "A" ska skicka ett meddelande till en annan Goroutine "B", är "A" blockerad tills "B" har mottagit meddelandet. På samma sätt är "B" blockerad om den vill läsa ett meddelande tills "A" har skickat det. Detta kommunikationsätt används främst för att skicka en spelsessions tidsenhet, Tick, till övriga Goroutines.

Det andra är inhämtning av information via objektorienterade metodanrop[5], exempelvis vid uppsamling av alla spelare och asteroiders positioner.

2.3 Kommunikation mellan front- och back-end

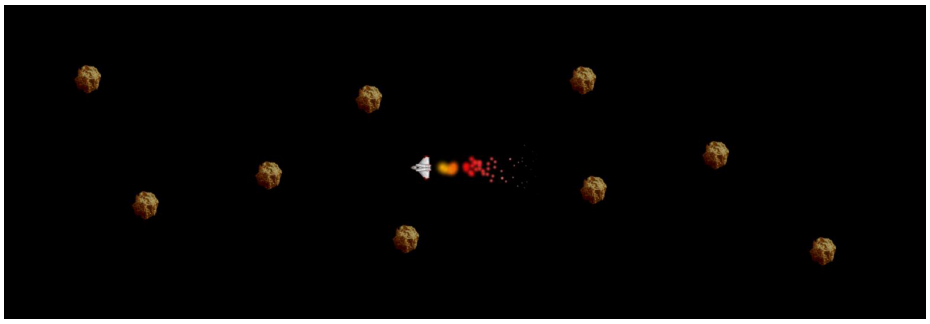
För att kunna kommunicera mellan front-end implementerad i C#/Unity och back-end implementerad i Golang utnyttjas TCP-protokollet för alla nätverkssockets och JSON (Javascript Object Notation) för att formatera all data som skickas över dessa. Detta sätt valdes för att minimera problemen som uppstår av att spelet skall kunna spelas på flera olika operativsystem, vilket hade kunnat innebära problem med allt till ordningen som enskilda bitar skickas till formattering av strängar i de olika programmeringsspråken.

Dessa problem löses genom att använda JSON-strängar formaterade enligt UTF-8, detta innebär att specifikationen är låst i ett stadie som är fullständigt plattformsoberoende, vilket gör att alla möjliga kombinationer av klienter från Windows, Linux och Mac datorer kan spela i samma session.

2.4 Intern kommunikation front-end

Klienten i front-end läser hela tiden av en TCP-socket för att ta emot den senaste spelvärlden från servern och för att inte detta ska blockera uppritningen av objekten i spelvärlden har denna avläsning lagts i en egen tråd. Den andra tråden som finns i klienten kommunicerar med avlyssningstråden genom att med ett objektorienterad metodanrop hämta hem den senaste spelvärlden och sedan uppdaterar de uppritade objekten enligt denna information.

2.5 Gränssnitt



Figur 1: Spelets utseende (beskuret)

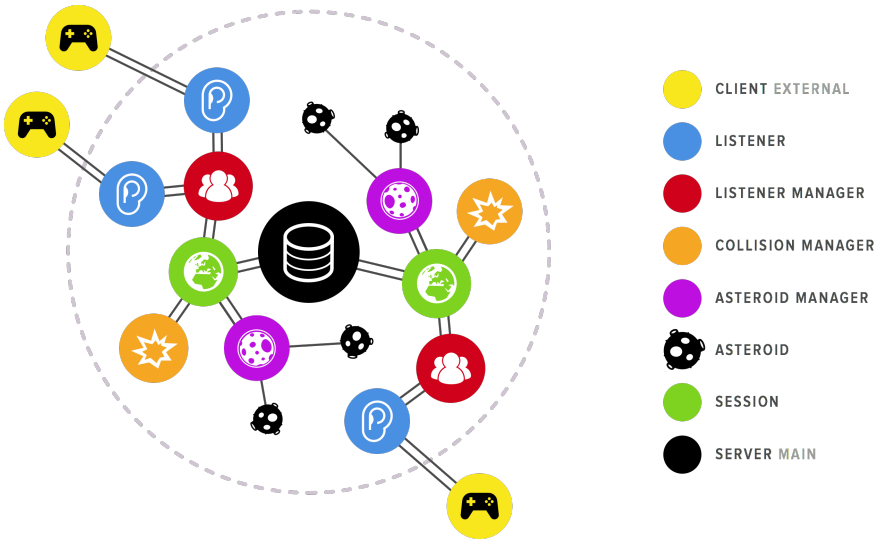
När spelaren först ansluter presenteras denna med en ruta där den kan skriva in sitt namn, vilket representerar "taggen" över spelarens huvud under resten av spelet. När servern sedan har upprättat en anslutning dyker spelaren rakt in i spelet, där det flyger asteroider som måste undvikas.

Spelaren rör sig genom att använda tangenterna W-A-S-D på tangentbordet, där W är uppåt, A är vänster, S är neråt och D är höger.

Om spelaren blir träffad har spelaren ytterligare två liv på sig att försöka samla poäng på och börjar spela igen genom att clicka Q.

När spelaren har förbrukat sina 3 totala liv är spelet för den spelaren slut, och denna spelare får då vänta tills alla andra spelare har slut på liv, då vinnaren presenteras.

2.6 Systemarkitektur



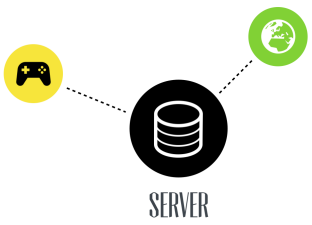
Figur 2: Övergripande systemarkitektur

Systemet består i sin mest abstrakta form utav en server och flera externa klienter eller spelare. Spelarna kommunicerar över nätverk med servern, bland annat genom att ständigt skicka sin nya position. Servern tar in datan från alla anslutna klienter, kombinerar och bearbetar den och skickar sedan tillbaka den nya spelvärlden till varje ansluten klient. Klienterna ritar sedan upp den nya datan grafiskt. Proceduren upprepas sedan för varje nytt steg i spelet, även kallat "Tick".

Klienten må vara mycket avskalad men servern är desto mer komplex och under ytan gömmer sig flertalet delar som samtidigt samverkar. De olika delarna syns inuti den streckade cirkeln i figur 2 och beskrivs nedan i ytterligare detalj.

2.6.1 Server

Server är den centrala delen i systemet och hanterar alla spelsessioner. När programmet initieras skapas en ny server som i sin tur startar två nya Goroutines. Ena för en ny session och den andra används för att kontrollera om en nya spelare vill ansluta till spelet. Ifall en ny spelare vill ansluta gör servern att den läggs till i en icke full session, om ingen finns skapas en ny session i en ny Goroutine.



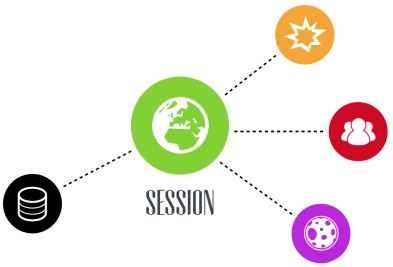
Figur 3: Server

Kommunikationen i server sker via synkroniserade kanaler som används till att skicka den port

en spelare ska använda för att sätta upp kommunikationen mellan back-end och front-end.

2.6.2 Session

Session används till att styra spelet i en specifik session. En ny session startar två nya Goroutines för sin Listenermanager och Asteroidmanager. I Session skapas en ny spelare om en sådan förfrågan uppkommit, som är max 8 st per session. Därefter inhämtas spelar-, och asteroidpositioner från respektive manager. Deras koordinater skickas sedan vidare till ytterligare en manager, Collisionmanager, som kalkylerar om kollisioner inträffat. Slutligen skickas den nya världen till Listenermanager.



Figur 4: Session

De olika managers synkroniseras genom en spelloop som skickar ut ett tick till dem. Dessa tick skickas via synkroniserade kanaler. Skapandet av nya spelare, inhämtningen av datan från managers och samt skickandet tillbaka sker på ett objektorienterat sätt via metodanrop.

2.6.3 AsteroidManager

Funktionen med AsteroidManager är först att samla in data om alla asteroider. Därefter kontrollera vilka av dem som har kolliderat eller åkt utanför spelplanen och därmed kan tas bort. Om det finns plats för mer asteroider i spelet skapas mer asteroider slumpartat.

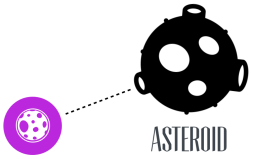
Kommunikationen till Asteroidmanager sker via en synkroniserad kanal som sänder spelets tick, som Asteroidmanagern sedan skickar vidare till alla asteroider. Inhämtningen av alla positioner sker via ett metodanrop.

Figur 5: AsteroidManager

2.6.4 Asteroids

Alla asteroider är egna Goroutines och som initieras med slumpartade x,y positioner på spelplanen, x och y differens som representerar riktning och hastighet den ska förflyttas vid varje tick.

Det enda en asteroid gör är att ändra sin position varje gång spelets tick mottages via



Figur 6: Asteroid

den synkroniserade kanalen från asteroidmanager.

2.6.5 ListenerManager

ListenerManager används för att samla in spelar positionerna ifrån alla listeners och sedan skicka vidare den uppdaterade spelvärden som kalkylerats fram av CollisionManager till alla listeners.

Figur 7: ListenerManager

Ifall en ny spelare vill ansluta, skapar ListenerManager en ny listener i en ny Goroutine och tar fram vilken port som den ska använda.

Kommunikationen till ListenerManager sker via en synkroniserad kanal som sänder spelets tick, som ListenerManager sedan skickar vidare till alla listeners. Inhämtningen av alla positioner sker via ett metodanrop.

2.6.6 Listener

Detta är den del som kopplar samman front-end med back-end. Då en ny spelare ansluter till sessionen skapas en listener, som innehåller anslutningsinformation samt spelarens karaktär i spelet. Anslutningsinformationen används för att via TCP skapa en förbindelse mellan front-end klienten och back-end servern. Denna förbindelse är viktig då den möjliggör dataöverföring mellan de två olika programmeringsspråken samt tillåter flera spelare att vara ansluta till samma session samtidigt.

Figur 8: Listener

2.6.7 CollisionManager

CollisionManager ansvarar för att kalkylera ifall det har skett kollisioner under spelet senaste tick. Kollisions beräkningen görs genom att jämföra varje spelares position, X- och Y-koordinat med övriga spelare och asteroider. Till sist jämförs alla asteroider med varandra.

Figur 9: CollisionManager

Kollisioners påverkan varierar beroende på objektet. Om en spelare kolliderar tappar den ett liv. Får spelaren slut på liv, är spelet över. Om en asteroid kolliderar med ett annat objekt stängs asteroidens process ner och asteroiden försvinner från spelvärlden.

CollisionManager är den den enda managern som inte körs via en goroutine utan enda kommunikation den har är med Session via objektorienterat metodanrop av Session.

2.6.8 Client

För att representera allt grafiskt för spelaren och för att ge spelaren möjligheten att flytta i spelvärlden implementerades en front-end klient genom spelmotorn Unity. Denna läser konstant av spelvärldens tillstånd från servern för att kunna visualisera spelvärlden. Klienten skickar all input från användaren till servern för att skapa en komplett spelupplevelse. Det sker minimalt med beräkningar i denna klient, utan alla beräkningar som inte är rent grafiska görs på servern, detta för att säkerställa att alla spelare ser samma sak på sin skärm.

Figur 10: Player

3 Implementation

3.1 Val av programmeringsspråk

Eftersom projektet är inriktat på concurrency föll valet av programmeringsspråk för implementationen av back-end på Golang för dess utmärkta stöd för concurrency[8]. Fördelen som fås genom att använda Golang och CSP-modellen är att det finns möjlighet till att köra många Goroutines samtidigt, vilket görs i detta projekt. Golang är även ett enkelt språk att förstå då det använder sig av ett syntax som är likt programmeringsspråket C[7].

Tyvärr är dock Golang bristande när det gäller grafik, därmed valdes att implementera front-end i spelmotorn Unity. Där används C# och Unitys funktionsbibliotek för att rita upp grafiken.

3.2 Centrala algoritmer

3.2.1 Initialisering eller ny spelare

Första steget i initialiseringen är att starta upp servern. Servern initialiserar och blockerar sedan sig själv i väntan på nya, eller i det här fallet, den första användaren.

När den första klienten väl ansluter sig direkt mot servern är det dags att vakna upp. Servern frågar då varje levande session om det finns en plats för en till spelare, annars skapar den en ny. I vårt fall finns ingen session så servern initierar helt enkelt en ny.

Väl inne i sessionen är det dags att starta upp diverse managers för att hantera olika delmoment i spelet. Asteroid-, Collision- samt Listener-manager startas och initieras. Session skickar sedan en bekräftelse till servern att den är redo varpå servern svarar att den vill ansluta en ny spelare. Detta delegeras sedan via session till Listenermanager som reserverar en unik port till vår nya spelare och en lokal listener som sköter kontakten externt. Porten bubblar sedan hela vägen tillbaka upp till servern som i sin tur skickar den tillbaka till klienten. Klienten ansluter sedan mot sin dedikerade listener via porten och är därefter fri att börja spela.

3.2.2 Ett steg i spelet

En annan central algoritm i Asteroids är hur varje enskilt skede i spelet hanteras, internt kallat ett “tick”. Med tick menas tiden mellan ett tillstånd och nästa direkt efterföljande tillstånd.

För varje steg eller tick beräknar varje asteroid själv sin nya position. Asteroiderna blockerar sedan sig själva i väntan på nästa steg. Asteroidmanager samlar ihop alla spelets asteroider och returnerar dessa till sessionen. Listener och listenermanager opererar på liknande manér men här hämtas spelarens nya position externt ifrån klienten. Session tar den färska datan och skickar den

vidare till Collisionmanager. Collisionmanager hanterar alla olika möjliga kollisioner i spelet. För varje kollision skickas ett ID tillbaka till respektive manager som tar hand om konsekvenserna. En spelare kanske förlorare ett liv medans att asteroiderna exploderar. Dessutom kan det vara aktuellt att skapa nya asteroider om världen börjar bli tom. Efta det att Collisionmanager returnerar så sammanställer sessionen all data och skapar det nya tillståndet i spelet. Tillståndet skickas sedan vidare till klienten via Listenermanager och listener för uppritning. Därefter återupprepas samma procedur tills spelets slut.

För att spelet ska flyta på bra rent grafiskt är det ypperst viktigt att dessa beräkningar går så snabbt som möjligt. Lämpligtvis under 16 ms (för att uppnå 60 frames per second) för en stabil och mjuk uppdateringsfrekvens hos klienten.

4 Slutsats

Vårt projekt är återskapandet av det klassiska spelet “Asteroids” med diverse utökningar. Att utveckla spelet till ett multiplayer-spel där spelarna kan spela med eller mot varandra är vårt slutgiltiga mål. Resultatet blev en version där spelet går ut på att få mest poäng av alla anslutna spelare genom att undvika asteroider och andra spelare så länge spelaren kan.

Spelets server är skriven mestadels i programmeringsspråket Golang, och är spelets back-end. Spelets front-end består av spelmotorn Unity och därmed indirekt programmeringsspråket C# som är ett av de standardiserade programmeringsspråken i Unity. Valet av språk grundades till största del i att processer är minimala storleksmässigt i Golang och att concurrencymodellen som Golang använder sig av passar till projektet. Detta gör att vi kan få en skalbar concurrency utan att hela tiden tänka på en synkronisering i form av mutual exclusion. Unity valdes då det är ett kraftfullt verktyg för att uppnå grafik i spel.

Överlag har projektet gått bra. En vidareutveckling som behövs innan spelet är helt efterliknat orginalspelet är att implementera skott för att skjuta på asteroider. Efter det kan en ytterligare utökning vara att till exempel skapa en mobilapplikation av spelet etc.

Slutsatsen om programmeringspråken som används är att det finns bättre språk för att skriva spel, men att det att det då inte går att få med så pass mycket concurrency som man får med golang. Eftersom concurrency har varit något som velats eftersträva i utvecklingen av spelet så är valet av golang för back-end ändå inget som ångras.

För att skriva front-end så skulle Unity kunna bytas ut mot något annat utvecklingsverktyg, då det visade sig att det var mer utmanande och onödigt komplicerat än vad som hade räknas med. Nu har projektgruppen till viss del åtminstone fått lärdom om Unitys fördelar och begränsningar vilket ses som något positivt i slutändan.

Referenser

[1] Wikipedia , Asteroids , https://en.wikipedia.org/wiki/Asteroids_(video_game)
Accessed: 25 Maj 2016

[2] Google , Golang concurrency , https://golang.org/doc/effective_go.html#concurrency
Accessed: 25 Maj 2016

[3] Wikipedia , Erlang concurrency , https://en.wikipedia.org/wiki/Erlang_(programming_language)
Accessed: 25 Maj 2016

[4] Google , Goroutine , https://golang.org/doc/faq#goroutines
Accessed: 25 Maj 2016

[5] Google , Golang Objektorienterat , https://golang.org/doc/faq#Is_Go_an_object-oriented_language
Accessed: 25 Maj 2016

[6] Wikipedia , TCP , https://en.wikipedia.org/wiki/Transmission_Control_Protocol
Accessed: 25 Maj 2016

[7] Google , Syntax , https://golang.org/doc/faq#different_syntax
Accessed: 25 Maj 2016

[8] Rob Pike , Concurrency , https://talks.golang.org/2012/concurrency.slide#1
Accessed: 25 Maj 2016

A Appendix

A.1 Utveckling

Mjukvarukrav: Golang 1.6 och Unity 5.3.1

Under projektet har all kodhantering skett via github och det är därifrån den slutgiltiga produkten går att ladda ner `https://github.com/uu-it-teaching/ospp-2016-group-08`. All kod finns under /src mappen och är uppdelad i en serverdel och en klientdel.

I mappen /meta finns hela gruppens dagböcker från projektets gång. Det finns också en beskrivning av alla medlemmar, gruppkontrakt och reflektion på gruppens arbete

Testning har genomförts på serversidan med golangs inbyggda verktyg go test och dokumentationen är genererad med go docs.

A.2 Installation

A.2.1 Kompilera och köra

För att starta igång systemet behövs servern först sättas igång. Detta görs genom att gå in i mappen /src/Server i en terminal och köra kommandot "make". Servern kommer då att kompileras och köras igång.

Serven kan även startas med en låtsasklient för att testa serven utan att använda Unity genom att skriva "make fake".

Därefter måste klienten köras igång och för att göra det måste mappen /UnityClient/Asteroids öppnas upp i Unity. När detta är gjort så välj "build and run" i Unity vilket kommer kompilera och köra igång klienten.

A.2.2 Tester

För att genomföra tester och endast få resultatet i terminalen skrivs "make test". Där fås det ut ett felmeddelande om något test inte är korrekt samt hur mycket av koden som testas skrivs ut. Skrivs "make $test_doc$" fås en detaljerad bild av vilka delar av koden som är testad.

För att kontrollera om det finns konkurrenstillstånd i systemet skrivs "make race"

A.2.3 Dokumentation

Dokumentation generas genom att skriva "make doc". Då kommer dokumentationen upp i html format för all kod skriven i Go.

A.3 Reflektioner

A.3.1 Sökta lärdomar

Projektet intresserade oss eftersom lärdom om nätverksprogrammering, processhierarkier samt kommunikation mellan olika programmeringsspråk skulle krävas för att genomföra projektet. Detta är alla viktiga delar som används mycket i verkligheten och även någonting alla i gruppen ville lära sig. Efter projektets gång kan man med säkerhet säga att vårt arbete har gett oss en betydligt bättre överblick över hur detta går till.

A.3.2 Oväntade svårigheter

Under tidigare projekt som gruppmedlemmarna tagit del av har det funnits en specifikation till handa. Denna specifikation gör uppdelning av projektet enklare då det i relativ tydlighet finns strukturerat vad som skall göras. I detta fall saknades detta vilket skapade tydliga svårigheter i uppdelning och även planering. Projektet innehöll mycket nytt gruppmedlemmarna aldrig tampats med tidigare, vilket ledde till planeringssvårigheter då ingen visste hur, vad som skulle göras.

Skulle projektet ha inletts idag så skulle vi förmodligen börjat med att försöka specificera gränssnitt och dela upp arbetet ytterligare. Detta har ofta varit flytande under projektets gång, mycket tack vare att ingen av oss byggt ett spel tidigare eller programmerat i Go. Dessutom hade det varit förmånligt att bryta ner produkten i mindre milstolpar så att det redan i ett tidigt skede finns en fungerande, men avskalad, produkt. I kontrast mot att vänta med att knyta ihop säcken till absolut sista veckan!