Linköpings universitet | Institutionen för datavetenskap
Kandidatuppsats, 16hp | Innovativ Programmering
Vårterminen 2016 | LIU-IDA/LITH-EX-G--16/052—SE

Utveckling och optimering av 2D spel i Javascript med ramverket Phaser.io

Jesper Bäck

Handledare, Erik Berglund
Examinator, Anders Fröberg


LINKÖPINGS
UNIVERSITET

Utveckling och optimering av 2D spel I JavaScript med ramverket Phaser.io

Jesper Bäck
Tekniska högskolan
Linköpings universitet
jesba289@student.liu.se

INLEDNING

Det är väldigt viktigt i dagens samhälle att vara effektiv med resurser, speciellt inom teknologi. Ooptimerad hårdvara kan leda till försämrad batteritid på t ex smartphones och liknande enheter. För andra enheter som inte använder batteri innebär det istället att mer el används, vilket beroende på utsträckning kan ha en väldigt negativ inverkan på miljön.

Detta examensarbete går ut på att utföra en analys av prestandan av ett 2D matematiskt pusselspel för mobila enheter. Eftersom det finns en så stor variation på mobila enheter är det viktigt att se till att det kan spelas med så bra prestanda som möjligt på så många olika enheter som möjligt utan att använda för mycket batteri.

Inom spel handlar resurseffektivitet eller optimering om att få ut mer prestanda av samma hårdvara. Optimering är ett väldigt viktigt steg inom spelutveckling då det annars hade krävts väldigt mycket dyrare hårdvara för att spela spelen på en spelbar framerate. Detta märks extra tydligt på konsoler där samma hårdvara har använts i nästan 10 år och ändå kan de driva moderna spel på en acceptabel nivå.

Optimering görs ofta i slutet av utvecklingen, när det inte kommer skrivas någon mer ny kod. Mindre optimering görs ibland under utveckling men detta är oftast när det är uppenbara orsaker till dålig prestanda som upptäcks. Utöver dessa undantag är den populära åsikten att det är enklast att göra det i slutet. Detta är bland annat på grund av att det är mycket mer tidseffektivt att fixa ett stort problem en gång än att kontinuerligt leta små källor till prestandaförluster. För tidig optimering kan också leda till mer komplex och svårhanterad kod. Det egentliga stora motargumentet till optimering vid projektslut är att man då kan ha ett väldigt komplext system med invecklade beroenden som kan göra refaktorering väldigt svår att genomföra.

En vanlig typ av optimering och även den som kommer användas i detta arbete är profilering. Profileringen är ett väldigt vanligt sätt att optimera speciellt inom webbutveckling då de flesta av dagens webbläsare kommer med inbyggda verktyg just för profilering. Profilering ger även väldigt lättolkade resultat och är över lag enkelt att använda. Profilering innebär att mäta hur länge eller hur många gånger specifika funktioner exekveras. Detta ger ett väldigt överskådligt sätt att se hur mycket av den ”tillgängliga” prestandan som används och hur mycket som är ledigt. Med hjälp av detta kan en utvecklare implementera en ny funktionalitet och sedan mäta hur mycket processortid det kostar att exekvera koden. Om det kostar för mycket så har utvecklaren möjlighet att hitta alternativa lösningar på problemet och jämföra resultat.

Ett annat vanligt verktyg som ofta används tillsammans med CPU-profilern är tidslinjeinspelningar, även kallat timeline recordings. Dessa visar exakt vad som händer vid exakta tidpunkter under ”inspelningen”. Som med en CPU-profiler spelar den alltså in en sekvens, men istället för att visa hur mycket tid som går åt till olika funktioner visar den en tidslinje där man kan se vilka funktioner som exekveras vid vilken tidpunkt, en varning kommer visas om funktionerna använder för mycket tid.

Spelet är utvecklat i Javascript och använder ramverket Phaser.io. Phaser.io använder i sin tur pixi.js för att rendera webbgrafik i webbläsaren.

För att flytta spelet från webbläsaren till mobila enheter används verktyget Intel XDK. Verktyget anpassar koden automatiskt och skapar en naitive applikation för valda plattformar.

Som tidigare nämnt kommer jag göra en prestandaanalys av spelet. I analysen hoppas jag kunna ta fram vilka operationer och designval som har störst effekt på prestandan och dra en slutsats om ifall det är värt tiden att försöka optimera denna typ av spel och till vilken grad man ska försöka optimera i så fall. Analysen kommer använda just CPU-profilering för att mäta exekveringstiden i olika funktioner. För profileringen har vi valt att använda webbläsaren Google Chromes inbyggda CPU-profiler då projektet ändå kommer utvecklas med hjälp av samma webbläsare.

SYFTE OCH MÅL

Phaser.io är ett väldigt populärt JavaScriptramverk för spelutveckling till webbläsare, alltså är det väldigt många

spel som utvecklas med Phaser.io. Dock är prestanda inom ramverket ett väldigt outforskat område där man har tur om man hittar ett fåtal foruminlägg som diskuterar ämnet. Därför är målet med arbetet att hitta vilka vanliga prestandaproblem man kan stöta på under utveckling med Phaser.io. Förhoppningen är då att informationen når ut till andra utvecklare som använder ramverket så att de kan undvika dessa problem och inte själva behöva upptäcka dem i optimeringsfasen.

FRÅGESTÄLLNING

Spelet har inte utvecklats med någon hänsyn till prestanda. Därför kan man anta att det troligtvis kommer innehålla många vanliga misstag som upptäcks och förbättras i ett senare skede i utvecklingen. Men att i förväg veta vad som kommer orsaka dålig prestanda är omöjligt.

- Hur ska ett mobilspel utvecklas med HTML5 och JavaScript för att säkerställa god prestanda?

BEGRÄNSNINGAR

En potentiellt stor begränsning i undersökningen är att analysen bygger på ett vanligt projekt. Vi kommer alltså med stor sannolikhet inte kunna täcka alla tänkbara scenarion som orsakar dålig prestanda. Det finns inte tid och skulle vara väldigt svårt att ta fram en metod för att täcka så många olika lösningssätt.

Undersökningen kan heller inte täcka prestandaproblem som orsakas av andra faktorer än lång exekveringstid. Prestandaproblem kan uppstå på grund av saker som att arbetsminnet är fullt eller att garbage collection börjar köras i bakgrunden.

Mätningarna körs även i webbläsaren på en vanlig laptop, detta för att vi inte har tillgång till många mobila enheter att utföra testning på. Vi vet alltså inte hur representativ undersökningen kommer vara men eftersom mätningarna är baserade på tiden processorn spenderar i en funktion antar vi att resultaten och förbättringarna kommer kunna översättas någorlunda till mobila enheter.

Eftersom mätningarna görs på en vanlig laptop finns det också risk för variationer i värden mellan olika mätningar av samma sekvens. En dators beräkningskraft kan variera beroende på vilka andra program som är aktiva på datorn eller på faktorer som processorns turbo-läge. Vi tänker dock att detta inte påverkar resultaten för mycket eftersom vi inte söker exakta mätvärden utan generella orsaker till försämrad prestanda.

BAKGRUND

Projektet går ut på att bygga vidare på ett existerande spel. Spelet utvecklas med open source JavaScript-ramverket Phaser.io. Det utvecklas och exekveras i webbläsaren och görs sedan tillgängligt på flera andra plattformar via cross-platform verktyget Intel XDK. Min uppgift är att fortsätta utvecklingen av spelet och avslutningsvis göra en analys av prestandan i spelet för att hitta och sammanställa vanliga orsaker till dålig prestanda.

TEORI

HTML5

HTML5 är den senaste versionen av HTML och kommer med canvas elementet inbyggt. Alla spel i HTML renderas i en canvas med hjälp av WebGL och spelet manipuleras med JavaScript.

JAVASCRIPT

JavaScript är ett mycket populärt programmeringsspråk. Det stöds av en en stor mängd av alla moderna webbläsare och en signifikant mängd av moderna webbsidor. Webbapplikationer använder det för att manipulera sitt innehåll, detta inkluderar även mobiler och mobila webbläsare vilket gör JavaScript till ett mycket attraktivt språk för att nå ut till många plattformar samtidigt.

JavaScript brukade vara ett interpreterat språk men har under senare år gått över till vad man kallar ett Just-in-Time eller JIT-kompilerat språk. JIT innebär att koden kompileras precis innan exekvering. Detta medför att källkoden kan skickas runt och exekveras på olika maskiner utan några hinder men också att prestandan stiger avsevärt jämfört mot interpretering i många fall.

Figur 1: JavaScripts prestandaökning

JavaScript erkänns idag som ett högpresterande språk. Detta kan antas vara på grund av ökningarna som skett i benchmarks sedan V8 motorn introducerades. Men som författarna av artikeln ”JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications” [4] skriver så representerar JavaScriptbenchmarks inte reella

fall som hemsidor. En hemsida är väldigt dynamisk och oförutsägbar. Därför är benchmarks mer applicerbara på t ex en webbserver.

Även artikeln ”Improving JavaScript performance by deconstructing the type system” [5] påpekar att nya optimeringar till kompilatorer som görs för V8 lägger fokus på att prestera bra i benchmarks och att resultaten inte alls översätts bra till dynamiskt innehåll så som webbsidor vilket är de mer praktiska användningsområdena.

JAVASCRIPT V8 MOTOR

Anledningen till att JavaScript har fått ett sådant användsningsuppsving inom utveckling är V8-motorn som utvecklades av The Chromium Project. Motorn har öppnat upp för större användningsområden inom webbutveckling men det har också gjort JavaScript till ett alternativ för andra projekt. Det används till exempel för ramverket Node.js och det ganska populära databashanteringssystemet MongoDB.

PHASER.IO

Phaser.io är ett open source spelramverk för JavaScript som renderar med hjälp av pixi.js. Phaser.io stödjer både WebGL och canvas vilket gör att spelen kan köras effektivt på alla webbläsare med stöd för tekniken. Tanken med Phaser.io är alltså att det ska vara direkt spelbart i webbläsaren men också kunna nå ut till mobila enheter som native-applikationer. Detta görs med hjälp av cross-platform-verktyg som t ex Cordova eller genom wrappers som cocoon.js som Phaser.io har direkt stöd för.

CROSS-PLATFORM

Cross-platform innebär att man har ett verktyg eller en tjänst som tar en kodbas och sedan konverterar den till en naitive applikation för en eller flera andra plattformar. Detta är praktiskt för utvecklare då man bara behöver implementera och underhålla en kodbas istället för en för varje plattform.

Det finns några nackdelar med cross-platform. De största är att man är begränsad till vad verktyget eller tjänsten låter dig göra. Tjänsten kommer ha ett eget Application Programming Interface, eller ’API’ och riktlinjer för hur saker ska implementeras och kommer ofta inte kunna utnyttja alla små fördelar som de olika plattformarna kanske stödjer.

Det är också oftast en prestandaförlust att använda sig av cross-platform då applikationen inte är skräddarsydd för plattformen, vilket leder till att den blir sämre anpassad.
Utöver verktyg för att konvertera en kodbas till andra enheter finns det en populär alternativ lösning där applikationen kodas som en komplett webbapplikation och sedan används plattformarnas olika webbkomponenter för att visa hemsidan som en native-applikation.

OPTIMERING

Det finns en otrolig mängd olika tekniker som används för att optimera hemsidor och webbapplikationer idag. Ett stort problem är tiden det tar att hämta data från en server, det kan ta väldigt lång tid och brukar undvikas med hjälp av tekniker som att minimera statiska filer och att cacha data i största möjliga utsträckning. [7]

Cachad data används i alla moderna webbläsare idag och sparar in enormt med tid, speciellt på hemsidor som använder mycket stora bilder och liknande. Men det är inte bara hemsidor som cachas, spel för webbläsaren så som de som utvecklas i Phaser.io nyttjar caching. Detta leder till att spel och assets kommer ladda mycket snabbare efter första gången de används.

För lite mer avancerad och djupgående optimering finns det lite andra typer av verktyg som blir intressanta. Som tidigare nämnt är profiling och timelineverktygen populära sätt att optimera mer på djupet. Dessa verktyg kan hjälpa till med att analysera den specifika källkoden och mäta i nästan exakt tid hur lång tid olika operationer tar. Om en operation då tar för mycket tid vet man vilket kodstycke som behöver optimeras.

CPU PROFILING

Som nämnt tidigare så går profiling ut på att mäta hur mycket tid det tar att köra koden och vad i koden som tar tid. Det finns några olika sätt att samla denna data.

En vanlig mätmetod är sampling, den fungerar genom att periodiskt avbryta exekveringen för att läsa anropsstacken. Sampling är väldigt bra eftersom det praktiskt taget inte tillför någon overhead alls. Den kan alltså göras i realtid parallellt med exekveringen av koden som ska analyseras utan att det märks någon större skillnad i exekveringstiden. Det enda stora problemet med sampling är att den kan missa vissa funktioner fullständigt. Om en funktion hinner läggas till och försvinna från anropsstacken innan profilern gör en läsning kommer profilern inte ha en aning om att den funktionen ens har existerat. Detta är dock sällan ett problem då de funktioner som missas av profilern ofta har så kort exekveringstid att de inte har någon påverkan på prestandan i slutändan i alla fall.

Ett annat inte lika vanligt alternativ är Instrumentation. Instrumentation går ut på att anropa funktioner som räknar anrop och tid innan och efter att funktioner i källkoden exekveras. Metoden är bra för att den räknar antalet anrop och tid väldigt exakt men tyvärr kan även data från denna metod bli felaktig. Eftersom funktionerna som räknar tiden körs tillsammans med den vanliga källkoden räknas de med till exekveringstiden. Med stora funktioner som tar lång tid är det inte så farligt men för korta funktioner som anropas

ofta kommer utdata antyda att dessa tar mycket mer tid än vad de egentligen gör.

Figur 2:En call graph som visar hur en funktion grenar ut i anrop till andra funtkioner.

Insamlad data behöver också presenteras på ett sätt som är förståeligt för användaren. All data samlas in som call stacks, vilket är en serie med funktionsanrop från den aktiva funktionen upp till ursprungsfunktionen som startade hela programmet. Från denna data kan en call graph skapas likt den i figur 2, ovan skapas.

En call graph är en trädstruktur där varje funktion är en nod som grenar ut nedåt i funktioner som den anropar. Varje nod har en tid angiven som representerar hur mycket tid som spenderades i denna nod och i dess undernoder, för att veta hur mycket av den tiden som spenderades exklusivt i den aktuella noden har denna ett ”self” värde som anger hur mycket tid som gick åt till att exekvera enbart den egna noden.

I de flesta fall sammanställs all data till en trädstruktur som börjar med ursprungsfunktionen som rot till trädet. Från roten kan man sedan ”veckla ut” trädet och se alla funktioner som anropas av den noden. På detta sätt kan man följa exekveringen av koden och hitta vilka funktioner som tar tid att köra. Att funktionen använder mycket tid kan bero på olika saker, som till exempel att funktionen innehåller tunga beräkningar eller att den anropas väldigt ofta.

PRESTANDA

Med prestanda menas i första hand framerate, vilket är ett mått på hur många gånger per sekund som skärmen uppdateras med en ny bild. Med en hög framerate kommer alltså alla rörelser i spelet att se mycket mjukare och mer väldefinierade ut, men framför allt så kommer spelet att kännas mer responsivt då spelet reagerar på spelarens inmatningar tidigare. Med låg framerate kommer spelet upplevas som långsamt, ryckigt och oresponsivt.

TIMELINE

Chromes tidslinjeverktyg är ganska lik CPU-profilern, tidslinjeverktyget mäter också hur lång tid som spenderas i specifika funktioner, men istället för att göra en summering över en hel inspelning visar en timeline exakt vad som händer vid varje tidpunkt.

Syftet med en timeline är alltså inte att ta reda på hur mycket tid olika operationer tar utan snarare att analysera ett exekveringsscenario för att se om resurserna räcker till. Tidslinjeverktyget räknar hur lång tid som spenderas på att beräkna varje frame.

Målet är att alltid köra i 60 frames per sekund, detta för att de flesta moderna skärmar är begränsade till 60hz. Därav finns det ingen anledning att försöka uppnå något högre än så; det skulle endast bidra till batteriförlust i detta fall. Om webbsidan ska hålla konstant 60fps innebär det att varje frame måste beräknas på ungefär 16.6ms.

Om tidslinjeverktyget upptäcker att en frame tar längre tid än så att rendera kommer den visa en varning och markera vart i tidslinjen som just den framen beräknas. Det visas även hur lång tid den tog och hur hög framerate webbsidan hade under tiden. Verktyget kommer då visa exakt vilka funktioner som kördes under denna tid vilket gör det väldigt enkelt att avgöra vad som kan göras för att höja prestandan. Även här behöver det inte alltid vara en ineffektiv funktion som orsakar problem det kan också vara så att för många saker försöker exekveras samtidigt och i ett sådant fall är det kanske mer aktuellt att tänka om strukturen på koden och försöka sprida ut dess operationer.

FRAMERATE

Olika spel brukar kräva olika hög framerate för dess ändamål. De tidigare nämnda artiklarna skriver om first person shooters, vilket är en av de typer av spel som kräver absolut högst framerate. Pusselspel har generellt mycket mindre rörelse på skärmen och kräver inte samma responstid så de har inte nödvändigtvis samma krav på en hög framerate. Under projektet kommer jag ändå eftersträva att spelet ska hålla konstant 60 fps, frames per second.

Figur 3: Spelplanen

PUSSELSPEL

Spelet som utvecklas är ett matematiskt pusselspel som går ut på att matcha siffror och lista ut okända variabler för att få ihop en summa.

Man spelar i lugnt tempo, inga snabba reaktioner krävs och det händer inte så mycket på skärmen som spelaren måste reagera på. Bilden på skärmen kommer alltså vara ganska statisk och därav kommer inte låg framerate märkas alls lika mycket som det gör i t ex ett First-Person Shooter-spel. Därav kan gränsen för framerate som ”måste” uppehållas sättas mycket lägre utan att spelaren märker någon skillnad.

Som man kan se i figur 1 så kommer spelet i de flesta fall vara just statiskt och vänta på input från användaren. I de lägena kommer prestanda med stor sannolikhet inte vara ett problem men istället kommer det förväntas stor respons när spelaren interagerar med spelet och till exempel vinner på en bana, då kommer vi vilja utnyttja så mycket animationer som möjligt och då måste spelets framerate ändå vara tillräcklig.

TIDIGARE FORSKNING

Tidigare forskning inom spel och JavaScriptprestanda visar tydliga trender där framerate är det främsta måttet på hur bra ett spel uppfattas att prestera av en spelare, förutsatt att spelet håller en viss grundläggande visuell kvalité.

En Artikel[2] av Mark Claypool och Kajal Claypool undersöker hur framerate och bildupplösning påverkar spel med olika typer av perspektiv. Artikeln undersöker isometriskt, tredjeperson och förstaperson och kommer fram till att låg framerate har en väldigt stor negativ inverkan på hur bra spelare presterar.

En annan undersökning [3] om hur framerate och upplösning påverkar first person shooters kom fram till att den typen av spel blir ospelbara vid framerates under 15 fps och att spelare drar nytta av att spela vid 60 fps eller ännu högre framerate.

Det har gjorts en tidigare undersökning [1] av Johan Alm baserat på samma kodbas som jag kommer jobba med men som utforskar Phaser.io som ett spelramverk, vad det har för funktionalitet och vad som urskiljer det från andra liknande alternativ. Johan skriver lite om optimering av JavaScript och artikeln kommer även vara till stor hjälp för att lära sig Phaser.io och dess funktionalitet.

METOD

I denna avdelning tänker jag beskriva hur jag kommer gå till väga för att fortsätta utvecklingen av spelet och hur jag tänker mäta och utvärdera prestandan.

Jag kommer i arbetet främst använda mig av utvecklingsmiljön WebStorm då den har många användbara verktyg inbyggda så som versionshantering och är specialanpassad för webbutveckling med språk som JavaScript. Jag kommer också använda webbläsaren Google Chrome då den har bra och enkla utvecklarverktyg för JavaScript och HTML.

UTVECKLINGSMETODIK

Under projektets gång kommer jag och min handledare tillämpa flera agila metoder. Jag kommer arbeta iterativt med dagliga avstämningsmöten med handledaren för att diskutera eventuella problem som har dykt upp, göra en statuskoll på den aktuella arbetsuppgiften och prata om vad som ska göras härnäst.

TESTMILJÖ

Utveckling och testning kommer göras på en laptop med processorn i7-4510U och ett diskret grafikkort, modell Nvidia GeForce 840.

ANALYS

Analysen går ut på att hitta vilka operationer och tekniker som kostar mycket prestanda att använda och om möjligt försöka hitta en lösning eller ett effektivare sätt att implementera samma sak.

För att ta fram konsekvent data kommer jag ha en serie med olika sekvenser av spelet som jag kommer spela in med CPU-profileringsverktygt och tidslinjeverktyget på exakt samma sätt varje gång. Detta är för att jag ska kunna jämföra mot tidigare mätvärden och se om jag lyckas förbättra prestandan eller inte.

Figur 4: Exempel av hur en timeline inspelning ser ut

TIMELINE VERKTYG

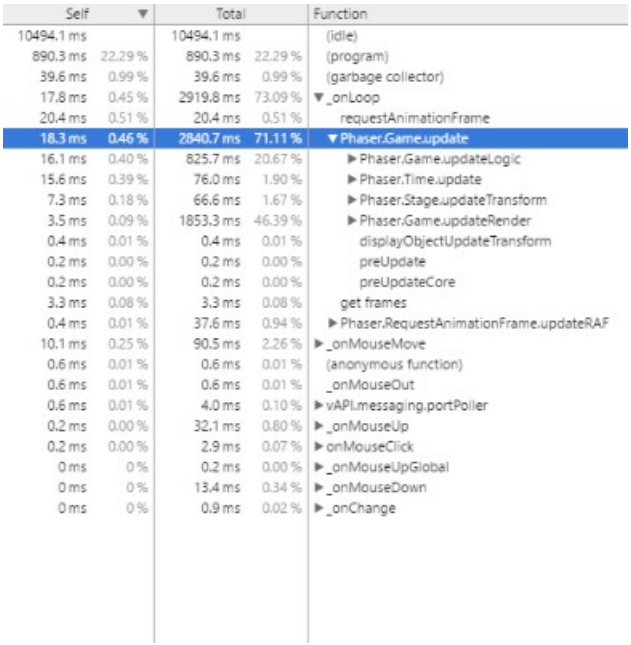
Med timelineverktyget kan man se om det finns några kritiska punkter där exekvering tar för lång tid. Eftersom vi vill hålla spelet vid 60 fps konstant så har processorn i regel ungefär 16ms på sig att hantera varje frame innan nästa behöver börja beräknas.

Med hjälp av profilerns timeline kan man då leta efter tillfällen där operationer kostar för mycket tid. Eftersom tidslinjen visar vilka funktioner som exekveras vid vilken tidpunkt kommer jag kunna se vilka funktioner som körs och hur mycket tid dom använder vid ett drop i framerate. Detta behöver inte vara ett tecken på dålig prestanda utan kan bara vara att det är för mycket som försöker göras samtidigt, vilket man kanske kan lösas genom att flytta vissa beräkningar så att dom görs vid en annan tidpunkt. Men om så inte är fallet så kommer jag se över koden som körs och se vad som kan göras för att snabba upp exekveringen.

För att spela in sekvenser kommer jag följa några riktlinjer som Google själva rekomenderar[6]. Jag vill hålla inspelningarna så korta som möjligt för att det ska vara enkelt dra en koppling mellan tidslinjen och vad som händer i spelet. Jag vill hålla en ”ren” testmiljö, alltså ingen cachad data och inga extensions eller liknande som kan påverka testresultaten. Även interaktioner som att scrolla på webbsidan eller musklick ska undvikas i största möjliga mån.

När jag har en färdig inspelning kommer jag gå igenom alla varningar för långsamma frames för att se om samma funktion eller funktioner syns flera gånger eller om dom helt enkelt inte passar in. Eftersom det är ett stort API jag jobbar med blir arbetet ganska mycket svårare, majoriteten av alla funktioner jag kommer se i tidslinjen kommer vara funktioner som tillhör Phasers API. Det kommer vara svårt för mig att veta vad alla funktioner gör och hur lång tid de egentligen ska använda. Men eftersom många av dem anropas regelbundet kan jag jämföra mot andra gånger funktionen har körts och hur mycket tid den använde då.

När jag fastställt en funktion som jag tror orsakar försämrad prestanda kommer jag börja med att ta bort den och köra om testet. Har exekveringstiden då blivit bättre jämfört med innan kan jag vara säker på att den borttagna koden var orsaken till den försämrade frameraten.



Self		Total		Function
10494.1 ms		10494.1 ms		(idle)
890.3 ms	22.29 %	890.3 ms	22.29 %	(program)
39.6 ms	0.99 %	39.6 ms	0.99 %	(garbage collector)
17.8 ms	0.45 %	2919.8 ms	73.09 %	▼ _onLoop
20.4 ms	0.51 %	20.4 ms	0.51 %	requestAnimationFrame
18.3 ms	0.46 %	2840.7 ms	71.11 %	▼ Phaser.Game.update
16.1 ms	0.40 %	825.7 ms	20.67 %	▶ Phaser.Game.updateLogic
15.6 ms	0.39 %	76.0 ms	1.90 %	▶ Phaser.Time.update
7.3 ms	0.18 %	66.6 ms	1.67 %	▶ Phaser.Stage.updateTransform
3.5 ms	0.09 %	1853.3 ms	46.39 %	▶ Phaser.Game.updateRender
0.4 ms	0.01 %	0.4 ms	0.01 %	displayObjectUpdateTransform
0.2 ms	0.00 %	0.2 ms	0.00 %	preUpdate
0.2 ms	0.00 %	0.2 ms	0.00 %	preUpdateCore
3.3 ms	0.08 %	3.3 ms	0.08 %	get frames
0.4 ms	0.01 %	37.6 ms	0.94 %	▶ Phaser.RequestAnimationFrame.updateRAF
10.1 ms	0.25 %	90.5 ms	2.26 %	▶ _onMouseMove
0.6 ms	0.01 %	0.6 ms	0.01 %	(anonymous function)
0.6 ms	0.01 %	0.6 ms	0.01 %	_onMouseOut
0.6 ms	0.01 %	4.0 ms	0.10 %	▶ vAPI.messaging.portPoller
0.2 ms	0.00 %	32.1 ms	0.80 %	▶ _onMouseUp
0.2 ms	0.00 %	2.9 ms	0.07 %	▶ onMouseClick
0 ms	0 %	0.2 ms	0.00 %	▶ _onMouseUpGlobal
0 ms	0 %	13.4 ms	0.34 %	▶ _onMouseDown
0 ms	0 %	0.9 ms	0.02 %	▶ _onChange

Figur 5: Exempel på hur en profiler inspelning ser ut, som man ser så görs i princip alla beräkningar i update-loopen.

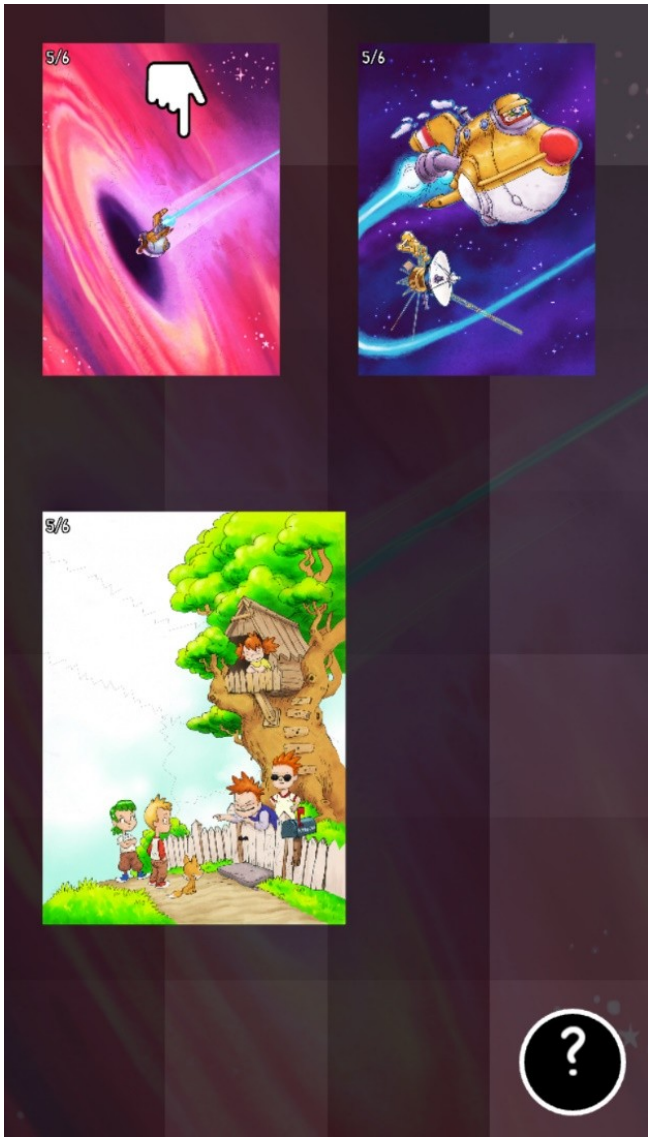
CPU PROFILER

Tidslinjen kommer vara väldigt bra för att hitta situationer där beräkningskraften inte räcker till så att jag kan se exakt när det finns optimeringsproblem att lösa men det kommer inte ge en så stor inblick i hur mycket tid det tar att köra olika delar av koden. Till detta kommer jag använda Google

Chromes profileringsverktyg. Här får man inte samma överblick över när saker händer men man ser mycket tydligare hur mycket tid som går åt till olika funktioner. Verktyget visar hela exekveringsträdet, hur mycket tid som spenderas i ”self” alltså i den nämnda funktionen och hur mycket tid som spenderas i funktioner som anropats av den nämnda funktionen. Det kan vara svårt att avgöra vad som ska ta mycket tid och inte eftersom mycket av utdatan från profilern är full med funktioner som hör till Phasers egna bibliotek. Det är dock ofta ganska tydligt vad Phasers egna funktioner gör baserat på deras funktionsnamn men annars kan man använda Phasers dokumentation eller dess källkod för att avgöra vad som händer i funktionen. Om man vet ungefär vad funktionen ska göra och märker en förändring från tidigare mätningar är det ganska säkert att anta att någon ny implementation har påverkat prestandan.

En egenskap som gör profilern väldigt användbar är de olika sätten att sortera data. De två primära sätten att sortera är ”Top down” och ”Bottom Up”. Top down sorteringen visar toppen av trädet och hur stor andel av tiden som spenderas i funktionerna högst upp i ”anropsträdet”. Bottom Up gör som namnet antyder och vänder upp och ned på trädet och visar de funktioner som själva använder mest tid och sedan vilka funktioner som anropat dem.

Jag kommer troligtvis ha mest nytta av bottom up läget, då det är betydligt mycket enklare att gå igenom och det kommer visa både dyra operationer i phasers bibliotek och funktioner vi själva har skrivit i fall de skulle visa sig bli väldigt ineffektiva.

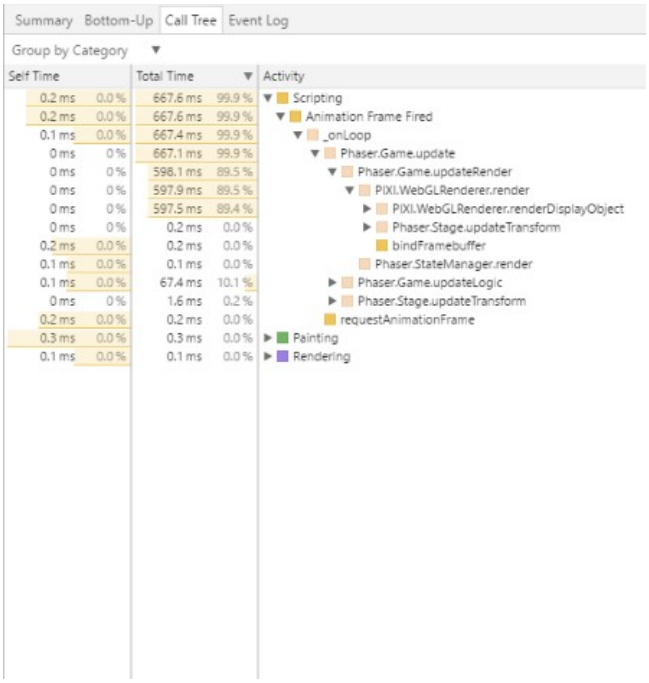


Figur 6: Spelmenyn, varje bild låses upp bit för bit

RESULTAT

Figur 7 visar en mätning från när spelet precis startats, efter att laddningsskärmen är färdig. Spelaren tas då till huvudmenyn där spelaren kan välja vilken del av spelet som den vill spela härnäst.

Figur 6 visar menyn. Den är uppbyggt av knappar i form av tavlor som spelaren ska låsa upp, dessa tavlor är då högupplösta assets vi har skapat.

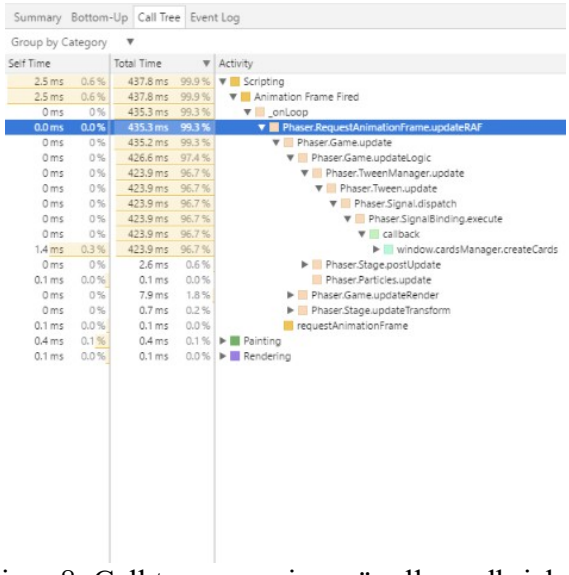


Figur 7: Call tree som visar när alla tavlor skapas

Under utvecklingen satte vi in dessa bilder i spelet istället för lågupplösta placeholders som användes tidigare och det blev en enorm ökning i tiden det tog att starta spelet.

När jag gjorde en mätning upptäckte jag att det tar ca 2000ms att rendera alla bilder. Eftersom det sker i en laddningsskärm behöver det inte beräknas snabbt, men 2 sekunder skapar en märkbar stamning i spelet mellan spelarens input och när menyn kommer fram på skärmen.

Figur 3 visar hur spelplanen ser ut, varje bricka är ett eget objekt med egna sprites. Figur 8 visar en inspelning av precis när spelplanen skapas. Då skapas först hela spelobjektet som innehåller alla andra objekt och all data som har med spelsessionen att göra, sedan skapas alla brickor.

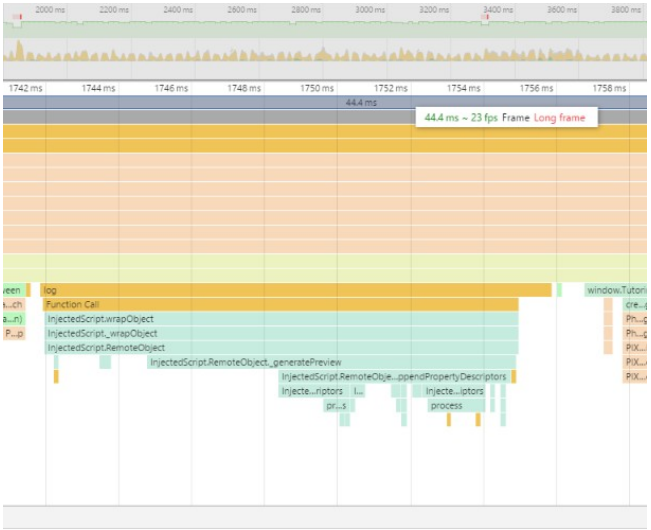


Figur 8: Call tree som visar när alla spelbrickor skapas

Först sker en stor drop i framerate när spelet byter från menyläge till spelläge och all ny data måste laddas in. Detta visas inte i figuren då det inte är så relevant, användaren kommer inte märka denna drop då det döljs av banans loadingscreen.

Det som visas i figuren är när alla brickor till spelet skapas, det är en 423.9ms lång frame som uppgörs nästan fullständigt av funktionen window.cardsManager.createCards. En bana är 4x6 brickor stor vilket innebär att 24 brickor måste skapas där varje bricka har en grupp med asset som ska laddas och sedan animeras in på spelplanen.

I figur 9 visas en mätning där jag upptäckte att utskrifter via utskrifsfunktionen console.log() ibland använder väldigt mycket tid. I denna mätning gick det åt 15ms i en frame för att exekvera log funktionen men då var ett stort komplext objekt som skrevs ut så samma resultat bör ej förväntas ifrån andra utskrifter av mindre komplex data.



Figur 9: Timeline som visar ett log anrop som tar ~14ms

DISKUSSION

RESULTAT

Något som har varit väldigt tydligt genom hela projektet är att det är dyrt att skapa objekt, speciellt när dessa objekt är sprites eller innehåller assets som ska laddas. Man ser på flera av mätningarna att det är detta som tar mycket tid. Så fort man klickar på en knapp och ett fönster visas så går spelets framerate ner. Ofta är det inte så tydligt då det handlar om en eller två frames som blir långsamma och sedan är spelet tillbaka på samma framerate som tidigare

igen, men i vissa fall tar det väldigt lång tid att skapa objektet. Om det tar upp mot 100 – 200ms att rendera en frame uppstår en ganska tydligt stamning i spelet där allt fryser.

En lösning på detta kan vara att skapa alla objekt som kommer användas i början av varje state och sedan dölja de som inte ska visas eller inte användas än. Detta gör att man kan maskera allt lagg som uppstår vid skapandet av objekt bakom någon form av statisk loadingscreen eller enkel animation där det lagg som uppstår inte kommer upptäckas av spelaren.

Detta leder då till att när spelaren till exempel startar spelet så kan man i förväg ladda alla menyer och fönster som annars laddas vid behov och istället för att skapa dom så gör man dom bara ”synliga” vilket är en mycket mindre krävande operation prestandamässigt. Samma sak gäller även i spelläget, då kan alla fönster och bilder som ska visas när spelaren vinner eller förlorar skapas i förväg vilket undviker lagg som annars uppstår vid skapandet av dessa.

Vi upptäckte också väldigt stora problem vid hantering av högupplösta assets, det tar enorma mängder tid att rendera högupplösta bilder. Från början hade vi bilder som varierade i storlekar på ungefär 1500 x 2600px per bild och då var det 18 sådana bilder som hämtades samtidigt, detta tog runt 2000ms att skapa och rendera. När vi skalade ner bilderna till strax under hälften av den tidigare upplösningen så sjönk tiden det tog att ladda bilderna från 2000 till runt 500ms som man kan se i figur 7. Detta är fortfarande ganska mycket tid och bilderna kan absolut göras ännu mindre men förbättringen är ändå ett bra bevis.

För prestandan gjorde det alltså väldigt stor skillnad vad det var för upplösning på bilderna men filstorleken verkade vara ganska irrelevant, den påverkar endast hur mycket RAM som används. Detta är dock ett problem som kan bli mycket mer aktuellt på mobila enheter där mängden ram är mer begränsad. Många enheter har 1GB eller mindre att nyttja till hela systemet. Då ska detta räcka till operativsystemet och andra applikationer som ligger i bakgrunden samtidigt.

Ett annat problem som behöver hanteras är mängden assets i projektet. I början av ett projekt är detta inte speciellt märkbart men allt eftersom antalet sprites stiger krävs fler diskaccesser och fler objekt i koden för att hålla ordning på alla assets som ska laddas in. Spritesheets är en populär lösning på detta problem. Istället för att ladda in bilder var för sig slår man ihop alla bilder till en eller några få större bilder som innehåller alla assets. När man i spelet sedan behöver hämta en speciell asset bestämmer man istället vilken ruta av spritesheeten som man vill hämta. Resultatet blir färre hämtningar från servern vilket kan göra väldigt stor skillnad vid laddning av spelet.

Figur 10: Exempel på hur ett spritesheet kan vara uppdelat

METOD OCH PROBLEM

Det största problemet under arbetet har varit att förstå hur man tar fram intressant data från profileringsverktygen. Dem är ganska överväldigande att använda i början och det är väldigt mycket data som presenteras på samma gång.

En stor majoritet av de funktioner som presenteras i verktygen kommer ifrån Phasers egna bibliotek. Det var väldigt svårt att analysera vad som händer när nästan hela anropsstacken är funktioner som man inte vet vad de gör.

Ibland är namnen på funktionerna ganska självförklarande och i andra fall kunde jag kolla uppåt i anropsstacken tills jag hittade en funktion som tillhörde min egna kodbas, därefter brukade det bli ganska självklart vad funktionen gjorde. Om jag till exempel ser att en frame är 300ms lång och att drawImage använder ungefär 70% av den tiden kan det vara svårt att avgöra vad som händer, men när jag ser att alla dessa grenar ut ifrån funktionen createCards, då är det ganska säkert att anta att dessa drawImage anrop kommer ifrån alla brickor som skapas och ritas på skärmen.

Utöver svårigheterna med utförandet av analysen har programmeringen varit svår vid tillfällen. När jag tog emot projektet innehöll det redan några tusen rader kod och det fanns ingen dokumentation, det tog ungefär två veckor att lära sig kodbasen bra nog för att kunna implementera någonting nytt.

Flödet i ett Phaser projekt är ganska invecklat, det var väldigt svårt att förstå övergångarna mellan olika states, när de händer, var de händer och vilken data som tillhörde vilket state.

Ett stort problem vi fick var när vi skulle flytta ett spelfönster som först visades i spelplanen från spelplanen till menyn istället, men fönstret behövde fortfarande komma åt data som genererades när spelplanen skapades. Situationen blir alltså att vi behövde visa data som inte hade skapats än.

Detta var ett väldigt specifikt exempel men det beskriver den största svårigheten under projektet. Utan dokumentation är det väldigt svårt att avgöra hur den tidigare personen tänkte sig att koden ska skrivas och i början slutade det ofta upp med att jag jobbade mot koden och gjorde saker på helt andra sätt än vad det var tänkt.

FRAMTIDA ARBETE

Eftersom tiden har varit en stor begränsning i undersökningen finns det ganska mycket kvar som är outforskat. På grund av detta gjordes analysen på ett existerande projekt, detta leder då till att vi kan ha missat saker eller att vi helt enkelt inte har behövt implementera tekniker som annars vanligtvis orsakar dålig prestanda.

En mer utförlig studie om Phaser.io där man testar tekniker i en mycket mer kontrollerad miljö behövs för att ordentligt avgöra vilken effekt de har på prestandan. I en sådan miljö kan man då kontrollera vilken och hur mycket indata som används och det går att mäta utdatan på ett mycket enklare och exakt sätt.

SLUTSATS

Syftet med arbetet var att ta fram hur man kan utveckla ett spel med HTML5 och JavaScript för att säkerställa god prestanda. Utifrån den frågan och arbetets metod har det tagits fram några vanliga problem som kan tänkas framkomma under utvecklingen av ett sådant spel.

Det är ganska vanliga problem som har upptäckts. Att skapa objekt är inte ett problem som är unikt för spelutveckling men eftersom det är bilder och andra assets som hanteras så blir alla problem avsevärt större. Därför är arbetets resultat något som kan vara värt att ha i åtanke redan från början av ett spels utveckling.

KÄLLOR

1. Alm, Johan. "Utveckling av 2D matematikpusselspel med Phaser. io." (2015).
2. Claypool, Mark, and Kajal Claypool. "Perspectives, frame rates and resolutions: it's all in the game." Proceedings of the 4th International Conference on Foundations of Digital Games. ACM, 2009.
3. Claypool, Mark, Kajal Claypool, and Feissal Damaa. "The effects of frame rate and resolution on users playing first person shooter games." Electronic Imaging 2006. International Society for Optics and Photonics, 2006.
4. Ratanaworabhan, Paruj, Benjamin Livshits, and Benjamin G. Zorn. "JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications." WebApps 10 (2010): 3-3.
5. Ahn, Wonsun, et al. "Improving JavaScript performance by deconstructing the type system." ACM SIGPLAN Notices. Vol. 49. No. 6. ACM, 2014.
6. https://developer.chrome.com/devtools/docs/timeline
7. Souders, Steve. "High-performance web sites." Communications of the ACM 51.12 (2008): 36-41.