Examensarbete

HÖGSKOLAN I SKÖVDE

DYNAMISKT MEDIANFILTER I REALTIDSSPEL

DYNAMIC MEDIAN FILTER IN REAL-TIME GAMES

Examensarbete inom huvudområdet Datavetenskap
Grundnivå 30 högskolepoäng
Vårtermin 2015

Johannes Sinander

Handledare: Sanny Syberfeldt
Examinator: Henrik Gustavsson

Sammanfattning

Detta arbete undersöker hur medianfilter kan anpassas för att användas i realtidsspel. Ett medianfilter är en bildbehandlingsalgoritm som går igenom varje pixel i en bild och ersätter den med medianvärdet av alla pixlar inom en viss radie. Två olika metoder för att anpassa medianfilter för realtidsspel jämförs. Den ena metoden använder en fast radie genom hela körningen, medan den andra metoden ändrar radie dynamiskt baserat på programmets FPS. Det som undersöks är vilken metod som är lämpligast att använda i realtidsspel.

En implementation av ett medianfilter skapas med de två metoderna som ska jämföras. Mätningar görs på implementationen för att utvärdera problemet. Resultaten tyder på metoden som använder dynamisk radie är bättre lämpad för realtidsspel, men att det kan finnas orsaker till att använda båda metoderna. Något som skulle vara intressant att forska vidare på är olika metoder för att göra medianfiltret snabbare, till exempel genom att variera antal samplingspunkter.

Nyckelord: Medianfilter, realtidsspel, bildbehandling, prestanda

Innehållsförteckning

1 Introduktion

Digitala grafiska filter av olika typer har under många år använts för att behandla bilder och video för att öka bildkvalitén eller för att få en viss effekt på bilden. Men många av dessa filter används ganska sällan i realtidsapplikationer som till exempel datorspel. En orsak till det är att många filter är väldigt beräkningstunga eftersom de ofta gör beräkningar på varje pixel i en bild.

Medianfilter är ett filter som används flitigt inom både bild- och videobehandling främst för sin förmåga att kunna reducera brus utan att bilden tappar särskilt mycket skärpa. Ett medianfilter går igenom varje pixel i en bild och ersätter den med medianvärdet av närstående pixlar inom en viss radie. Ett vanligt medianfilter kan endast hantera en färgkanal, vilket betyder att det inte automatiskt kan hantera färgade bilder. Det går däremot att bygga ut medianfiltret för att hantera flera färgkanaler, men det är inte helt trivialt. En metod som kan användas för det är adaptiv skalärmedian. Medianfilter är normalt sett väldigt beräkningstunga, men det finns en del olika sätt som de kan snabbas upp på, till exempel genom att använda histogram eller samtidiga processer.

I det här arbetet utvecklas och utvärderas ett medianfilter som ska kunna användas i realtidsspel. Medianfiltret hanterar färgbilder genom att använda adaptiv skalärmedian och använder sig av samtidiga processer för att snabba upp exekveringen. Två olika metoder används för att bestämma radien på medianfiltret. Den ena metoden ändrar filtrets radie dynamiskt beroende på programmets FPS och den andra metoden använder en fast radie. De två metoderna utvärderas och jämförs för att se vilken metod som bäst håller stabil FPS under varierande processorbelastning. Utvärderingen tar även hänsyn till hur utseendet på filtret skiljer sig mellan de två varianterna.

Ett experiment görs för att besvara problemet. Det vill säga att det skapas en implementation av det som beskrivits ovan som det sedan görs mätningar på. Mätningar görs i form av prestandatester som mäter programmets FPS under olika förhållanden. De uppmätta värdena analyseras och utvärderas sedan och resultatet diskuteras.

Implementationen av medianfiltret kör på GPU:n för att vara så effektiv som möjligt. För att göra allmänna beräkningar på GPU:n så används beräkningsplattformen och programmeringsmodellen CUDA. Medianfiltret implementeras med adaptiv skalärmedian och använder quicksort och insertion sort för sortering. Implementationen använder parallellism genom att det skapas en tråd för varje pixel i bilden. Varje tråd gör endast beräkningar på sin tilldelade pixel och när alla trådar är klara så är medianfiltret klart. Ett enkelt shoot'em up spel utvecklas dessutom för att ha något att testa och utvärdera på.

2 Bakgrund

Detta kapitel inleds med en kort generell genomgång av digitala grafiska filter och ger sedan en djupare beskrivning av medianfilter som detta arbete kommer undersöka. Olika metoder för att implementera medianfilter samt metoder för att göra medianfilter effektivare beskrivs också.

Digitala grafiska filter är kraftfulla metoder för att göra ändringar på delar av bilder eller hela bilder. De kan användas för att till exempel reducera brus eller för att förvränga en bild. Grafiska filter används mycket inom bland annat bild- och videobehandling för att öka kvalitén eller för att få en viss effekt på bilden. Filter används även en del inom spel i form av till exempel rörelseoskärpa när en karaktär vänder sig snabbt i ett förstapersonsspel. Men det är många filter från exempelvis bildbehandling som inte används i spel, förmodligen beror det delvis på att många filter är väldigt beräkningstunga eftersom de gör beräkningar på varje pixel.

2.1 Medianfilter

Medianfilter används mycket inom video- och bildbehandling för sin förmåga att reducera brus utan att bilden förlorar särskilt mycket skärpa. Medianfilter är en av de populäraste teknikerna för att avlägsna impulsbrus. Anledningen till dess framgång är den goda prestandan och sin beräkningsenkelhet (Singh och Bora, 2004).

Ett medianfilter går igenom varje pixel i en bild och ersätter den med medianen av alla närstående pixlar inom en viss radie. För att få ut medianen så sorteras de undersökta pixlarna med avseende på sitt pixelvärde, och sedan väljs den mittersta pixeln ut till att vara medianen. Det här leder till att brus kan reduceras eftersom pixlar som skiljer sig från sina grannar blir ersatta med ett värde som bättre passar in. Eftersom medianvärdet måste komma från någon av de närstående pixlarna så kommer det aldrig att skapas helt nya orealistiska värden. Det gör att bilden fortfarande håller sig väldigt skarp jämfört med till exempel ett medelfilter som fungerar likadant som ett medianfilter men som använder medelvärdet av de närstående pixlarna istället för medianvärdet. Av denna anledning så är medianfiltret mycket bättre på att bevara skarpa kanter än något annat filter (Bagni, 2014). Figur 1 nedan visar en jämförelse mellan medianfilter och gaussisk oskärpa som är en annan teknik som kan användas för att minska brus. Båda filtren i figuren använder ungefär motsvarande styrka.

Figur 1 En jämförelse mellan medianfilter (mitten) och gaussisk oskärpa (höger). Originalbilden visas till vänster.

2.1.1 Adaptiv skalärmedian

Ett problem med ett vanligt medianfilter är att det endast kan hantera en färgkanal. Det finns inte heller någon enkel lösning för att bygga ut medianfilter till att hantera flera färgkanaler. Pixlar kan på ett smidigt sätt representeras som vektorer, där varje element i vektorn är en egen färgkanal. Men anledningen till att det är svårt att använda medianfilter direkt på vektorer beror på att det inte finns någon standard för hur vektorer, som till exempel representerar färgerna röd, grön och blå, ska sorteras i storleksordning.

En metod för att använda medianfilter på bilder med flera färgkanaler är adaptiv skalärmedian. Adaptiv skalärmedian fungerar så att ett medianfilter appliceras på varje färgkanal var för sig. Resultatet blir då att man får ut en vektor för varje färgkanal, där var och en av vektorerna har medianvärdet för en av färgkanalerna. Dessa vektorer kan sedan kombineras till en medianmatris där varje kolumn representerar en riktig pixel. För att få ut en medianvektor är det sedan bara att kombinera ett element från varje rad i matrisen, till exempel genom att skapa en vektor från matrisens diagonal (Koschan och Abidi, 2001). Detta kan däremot leda till missfärgningar eftersom det inte är säkert att den nya vektorn finns med i den ursprungliga bilden. Koschan och Abidi (2001) förklarar hur missfärgningar kan uppstå med följande exempel: Tre pixlar med färgkanalerna röd, grön och blå definieras av de tre vektorerna P1=(10, 40, 50), P2=(80, 50, 10) och P3=(50, 100, 150). Om medianfiltret skulle appliceras på varje komponent var för sig så skulle den resulterande vektorn till exempel kunna bli P'=(50, 50, 50). Denna vektor representerar en grå pixel som inte finns med bland någon av de tre undersökta pixlarna.

Som nämndes tidigare så är en bra egenskap med medianfiltret att det inte introducerar några nya värden som inte redan fanns i bilden från början. Detta, säger Astola, Haavisto och Neuvo (1990), bör därför även gälla för medianfilter som använder vektorer. Även om de har en god poäng med det och man förmodligen undviker missfärgningar om man följer den regeln, så är adaptiv skalärmedian en enkel metod som i många fall fungerar tillräckligt bra så länge kvalitet inte är det viktigaste.

2.1.2 Fler medianfilter för färgbilder

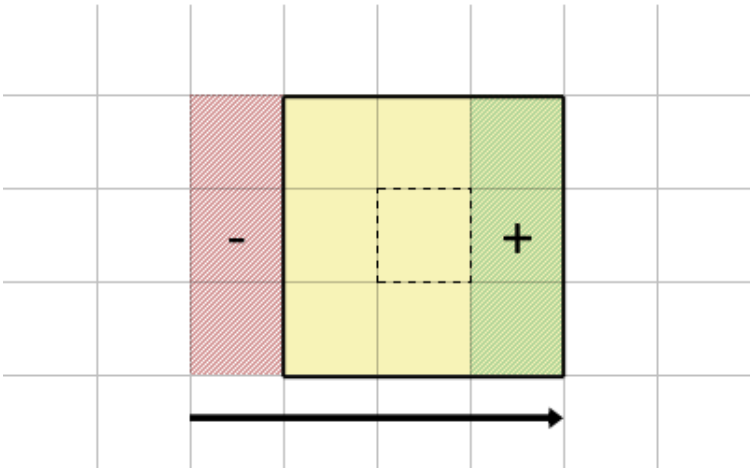
Det finns ett stort antal andra föreslagna metoder för att implementera medianfilter som kan hantera flera färgkanaler. Vissa som kan hitta ett medianvärde utan att skapa nya pixelvärden. Alla dessa metoder kommer däremot inte att beskrivas i det här arbetet, men några av dem kommer tas upp kort här. Koschan och Abidi (2001) beskriver i sin workshop följande metoder utöver den ovan beskrivna adaptiva skalärmedianen:

- *Vektor medianfilter:* I denna metod ersätts pixlarna med den pixel inom radien som minimerar summan av distanserna till de andra pixlarna i området med avseende på en godtycklig vektornorm. Fördelen med den här metoden är att det inte skapas några nya pixelvärden. Metoden beskrivs även av Astola m.fl. (1990).
- *Reducerat vektormedianfilter:* En approximation av vektor medianfiltret för att minska beräkningsåtgången för algoritmen. Metoden använder så kallade rum-fyllnings kurvor för att beskriva tredimensionella vektorer i en endimensionell rymd. I den endimensionella rymden går det sedan enkelt att hitta medianen på samma sätt som för ett vanligt medianfilter. Denna metod är däremot sämre på att reducera brus än ett vanligt medianfilter.

- *Medianfilter som appliceras på kromaticiteten i HSI-rymden:* En metod som använder HSI-färgrymden för att hitta en median. Även om HSI-rymden används så kvarstår däremot problemet att det inte enkelt går att rangordna vektorer. Därför används istället kromaticitetsplanet i HSI-rymden för att hitta en median. På så sätt garanteras det även att det valda värdet är ett av inputvärdena, alltså bildas det inga nya pixelvärden.
- *Medianfilter baserat på villkorlig sortering i HSV-rymden:* Villkorlig sortering är ett sätt att sortera vektorer i en form av storleksordning. Vektorerna sorteras först baserat på en av komponenterna, till exempel den första komponenten. Sedan sorteras vektorer med samma värden på en annan komponent, till exempel den andra komponenten, och så vidare. I den här metoden så sker sorteringen i HSV-rymden, vilket betyder att färgerna består av de tre komponenterna nyans, mättnad och intensitet. Den förslagna sorteringsordningen för pixlar i HSV-rymden är att först sortera vektorer baserat på intensitet från lägst till högst, sedan mättnad från högst till lägst och till sist nyans från lägst till högst. Den mittersta av de sorterade vektorerna väljs sedan ut som median. Även denna metod undviker att skapa nya pixlar.
- *Vektorriktningsfilter:* En typ av filter som är baserat på polära koordinater som sorterar vektorerna med hänsyn till vinkeln mellan dem. Denna metod är bra för att reducera kromaticitetsfel, men den är inte bra på att reducera stora intensitetsskillnader.

2.1.3 Histogram

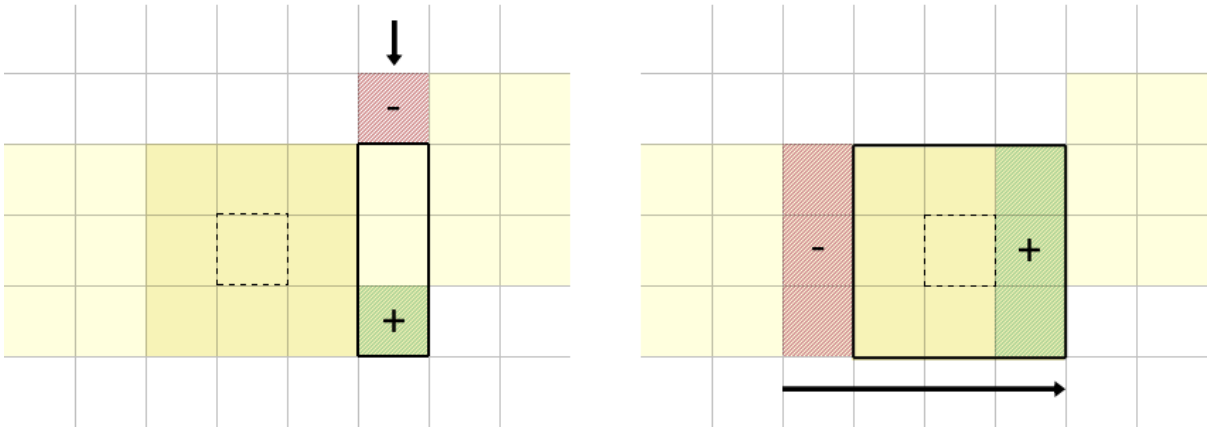
Medianfilter kan använda sig av histogram för att minska tidskomplexiteten av filtret. Perreault och Hébert (2007) skriver att ett histogram används för att ansamla pixlar till den så kallade kärnan. Med kärnan menas det område (vanligtvis kvadratformat) runt en pixel som medianen ska beräknas från. Ett histogram lagrar tillfälligt pixelvärdena så att när filtret går från en pixel till en annan så behöver det bara läggas till och tas bort ett fåtal pixlar, istället för att behöva göra om nästan exakt samma sak en gång till. Som går att lista ut av illustrationen i Figur 2 så behövs det 2r+1 tillägg och 2r+1 subtraktioner för att uppdatera histogrammet, om r är radien på filtret. Den här användningen av histogram gör att tidskomplexiteten per pixel blir O(r) istället för $O(r^2 \log r)$ som algoritmen normalt har.



Figur 2 Ett histogram uppdateras genom att lägga till nya pixlar på höger sida och ta bort lika många från vänster sida.

Perreault och Hébert (2007) noterar även en stor nackdel med den här typen av histogram och ger ett förslag på hur det kan förbättras till att ha en tidskomplexitet som är O(1) per pixel. Problemet med det beskrivna histogrammet är att ingen information behålls när en ny rad påbörjas. Så varje pixel kommer att läggas till och tas bort från histogram ett flertal gånger under tiden som hela bilden behandlas. Antalet gånger varje pixel läggs till beror på filtrets radie. Weiss (2006) beskriver detta som att klippa en gräsmatta fram och tillbaka och endast förflytta sig en centimeter i sidled varje gång man vänder. Det är alltså väldigt mycket redundans som skulle kunna undvikas.

Lösningen som Perreault och Hébert (2007) föreslår använder ett histogram för varje kolumn i bilden. Dessa histogram bevaras över alla rader under tiden hela bilden behandlas. För att beräkna kärnans histogram så är det då bara att summera de 2r+1 närstående kolumnhistogrammen. Skillnaden jämfört med det tidigare beskrivna histogrammet är att när filtret nu går vidare till nästa rad så finns informationen om beräkningarna från föregående rad kvar. För att uppdatera kärnans histogram så uppdateras först det kolumnhistogram som ligger till höger om kärnans histogram genom att lägga till en pixel längst ner samt ta bort pixeln högst upp. På så sätt sänks kolumnhistogrammet en rad. Efter det så flyttas kärnans histogram genom att kolumnhistogrammet till höger läggs till och kolumnhistogrammet till vänster i kärnan tas bort. Figur 3 nedan visar hur kärnans histogram uppdateras i de två steg som beskrivs.



Figur 3 Kärnans histogram uppdateras genom två steg.

2.1.4 Samtidiga processer

Ett av de största problemen med samtidiga processer är när två eller flera processer behöver dela information. För att processer ska kommunicera med varandra så krävs det oftast någon synkroniseringsmekanism som ser till att exekveringen av en process fördröjs för att vissa händelser ska ske i en viss ordning (Andrews och Schneider, 1983). Dessa fördröjningar gör exekveringen mindre effektiv, så därför är det viktigt att det sker så lite synkroniserad kommunikation som möjligt mellan processer.

Bosakova-Ardenska m.fl. (2009) beskriver hur medianfilter kan implementeras med parallella processer. Bilden som ska behandlas kan delas upp i segment av rader som kan fördelas mellan ett antal trådar som kan exekveras samtidigt av olika processorkärnor. Varje processorkärna blir tilldelad de rader som den ska bearbeta samt några rader över och under som den kommer att behöva för att beräkna medianen av de översta och nedersta raderna. Det leder till att det kommer att finnas dubblering av viss information, men den dubblerade informationen kommer däremot endast att behöva läsas, vilket gör att det inte kommer

behöva ske någon kommunikation mellan de olika processerna. Detta gör medianfiltret väldigt lämpligt att parallellisera eftersom det kan implementeras på ett mycket effektivt sätt.

Om GPU:n används för parallellism så ser implementationen lite annorlunda ut. Eftersom en GPU har många gånger fler processorkärnor än en CPU så är det mycket effektivare att använda många trådar. Perrot, Domas & Couturier (2014) beskriver i sin journal att de låter varje tråd behandla endast en till två pixlar. Detta skiljer sig en del från CPU-implementationen som beskrivs av Bosakova-Ardenska m.fl. (2009), där varje tråd behöver ta hand om ett flertal rader.

3 Problemformulering

Syftet med det här arbetet är att undersöka hur väl medianfilter kan anpassas för att användas i interaktiva realtidsapplikationer som till exempel spel. Medianfilter har länge använts inom till exempel bildbehandling, men det används sällan i spelsammanhang. Förmodligen på grund av att det är en väldigt beräkningstung algoritm som gör att den inte lämpar sig så bra för realtidsapplikationer, och dels för att den inte har så stort användningsområde inom spel. Idén är att applicera filtret på delar, eller hela renderingen av ett spel för att få en intressant grafisk stil på spelet.

För att bättre anpassa medianfilter till spel så kan radien på filtret ändras för att datorer som är olika kraftfulla ska kunna hantera det. En större radie på filtret betyder att det kommer ske många fler beräkningar per pixel, vilket gör att filtret tar längre tid på sig. Om radien istället minskas så kommer filtret att köras snabbare.

Två olika metoder för att ändra radien på medianfiltret undersöks och jämförs. Den första metoden är att helt enkelt bestämma en fast radie som anses vara lämplig för programmet. Den andra metoden går ut på att dynamiskt ändra radien på filtret baserat på programmets uppmätta FPS. När den andra metoden används så är det tänkt att programmet bättre bibehåller jämn uppdateringsfrekvens med så hög kvalité på filtret som möjligt, även när belastningen på processorerna förändras.

Det är viktigt att ett spel behåller en hög och stadig uppdateringsfrekvens. Enligt en undersökning av Claypool och Claypool (2007) så ökar spelarens prestanda logaritmiskt med spelets uppdateringsfrekvens. När ett spel hade 60 FPS istället för 3 FPS så presterade spelarna cirka sju gånger bättre på att skjuta i ett förstapersonsskjutspel. Däremot så nämns det även att spelarna inte uppfattar någon större skillnad vid högre uppdateringsfrekvenser, som till exempel mellan 30 FPS och 60 FPS. Undersökningen hanterar endast förstapersonsskjutspel, men det går att förvänta sig liknande resultat även för andra spelgenrer.

Eftersom medianfiltret används i spelsammanhang i undersökningen så är det ett krav att det ska kunna hantera färgbilder. Detta på grund av att i princip alla spel idag använder färg, så därför anses det lämpligast att även göra undersökningen på det.

Medianfiltrets effektivitet spelar stor roll eftersom det hela ska ske i realtid. Därför använder implementationen samtidiga processer som drar nytta av alla processorkärnor i en modern flerkärnig processor. Algoritmen för medianfiltret kan delas upp i mindre delar som är helt oberoende av varandra, så som beskrivs av Bosakova-Ardenska m.fl. (2009). De mindre delarna exekveras sedan samtidigt på de olika processorkärnorna, vilket leder till att algoritmen genomförs snabbare när den använder flera processorkärnor istället för bara en.

Problemet kan delas upp i två delmål som ska uppnås för att lösa problemet. Det första målet är att utveckla en implementation och det andra målet är att utvärdera implementationen.

3.1 Delmål 1: Utveckling

Det första målet är att utveckla ett medianfilter som använder samtidiga processer i sin exekvering. Filtret ska kunna hantera färgade bilder och det ska vara möjligt att applicera filtret på godtycklig bild i programmet, som exempelvis bakgrunden eller en karaktär. Medianfiltret ska kunna använda ett godtyckligt antal processorkärnor för att på så sätt vara skalbart och oberoende av antalet kärnor som processorn består av.

Två olika metoder används för att bestämma radien på medianfiltret. Den ena metoden använder en fast radie som anses vara lämplig med avseende på att den ska kunna köras i realtid och den andra metoden är en dynamiskt förändrande radie som är baserad på programmets FPS.

3.2 Delmål 2: Utvärdering

Det andra målet är att göra en utvärdering som undersöker resultatet av delmål 1. Det som undersöks är vilken metod som är lämpligast att använda i realtidsspel. Vilken metod som är lämpligast avgörs genom prestandatester som mäter programmets FPS under olika omständigheter. Den metod som bäst håller stabil FPS under varierande processorbelastning kommer anses vara lämpligast. Det bör dock noteras att antalet FPS är väldigt beroende på hur kraftfullt systemet som programmet kör på är, och under vilka omständigheter som testet sker på. Det gör att utvärderingen har en del osäkerhetsfaktorer som diskuteras mer under utvärderings- och avslutningskapitlen. En till sak som utvärderas är utseendet på filtret. Utseendet är däremot inte något som ligger i fokus under arbetet, men om medianfiltret ska användas i spel så är det ändå viktigt att ta hänsyn till utseendet.

3.3 Metodbeskrivning

Denna sektion beskriver de metoder som kommer användas för att lösa problemet och uppfylla de två delmålen som beskrivs ovan.

3.3.1 Metod för Delmål 1: Utveckling

Den lämpligaste metoden för att besvara delmålen är genom att göra ett experiment. Ett experiment går till så att man gör mätningar av viss data som man sedan tolkar och utvärderar (Bailey, 2008). För att göra mätningar så måste en implementation av medianfiltret skapas som består av de egenskaper som beskrevs i delmål 1. Det ska dessutom vara möjligt att testa implementationen i en spelmiljö. Därför skapas det även ett exempelspel som kan användas i delmål 2 för att göra prestandatester på. En viktig del med exempelspelet är att det måste finnas något som gör att belastningen på processorn ökar eller minskar, förutom medianfiltret.

En metod för att implementera medianfiltret bestäms också. Eftersom det är ett krav att medianfiltret ska kunna hantera färgbilder så väljs en metod som klarar det. Den metod som väljs är ett medianfilter med adaptiv skalärmedian som beskrivs av Koschan och Abidi (2001). Anledningen till att denna metod väljs, trots att den inte nödvändigtvis ger perfekt kvalitet, är för att den är enkel att implementera, och för att kvalitén på medianfiltret inte har någon större betydelse för den här undersökningen. Som redan beskrivits tidigare så använder implementationen även samtidiga processer. Detta implementeras enligt den beskrivning som ges i kapitel 2.1.4.

3.3.2 Metod för Delmål 2: Utvärdering

Som nämndes ovan så används experiment som metod. Utvärderingen sker därför i form av mätningar på den implementation och exempelspel som produceras. Ett exempelspel är nödvändigt för att utföra mätningarna eftersom ett medianfilter måste appliceras på någonting för att kunna testas. Det är också viktigt för att kunna testa programmet under olika hög belastning.

Mätningar sker i form av prestandatest som mäter programmets FPS när det utsätts för olika hög belastning. Testerna görs både på filtret som använder dynamiskt varierande radie och

för filtret med fast radie. En utvärdering av de uppmätta värdena görs för att avgöra vilken metod som bäst håller stabil FPS när processorbelastningen varierar.

Att mäta medianfiltrets utseende är däremot inte lika lätt eftersom vad som ser bra ut kan skilja sig mellan olika betraktare. Det finns inte heller något direkt värde som är enkelt att mäta. Utvärderingen av utseendet sker därför på en väldigt ytlig nivå som visuellt jämför hur de två varianterna av filtret skiljer sig under körningen. Det som granskas är hur utseendet beter sig när radien förändras dynamiskt jämfört med när den är fast.

För att testerna ska ha någon betydelse och kunna jämföras med varandra så måste alla testerna utföras på ett och samma system. Systemet som testerna utföras på består av följande:

- Windows 7 Professional 64-Bit Service Pack 1
- Processor: Intel Core™ i7-2600K 3.40Ghz
- Grafikkort: NVIDIA GeForce GTX 660
- 8192MB RAM

4 Implementation

Det här kapitlet beskriver ingående hur problemformuleringens första delmål besvaras med en implementation som är baserad på metodbeskrivningen. Implementationen skapas i Microsoft Visual Studio Ultimate 2012 med programmeringsspråket C++ och använder mediabiblioteket SDL (Lantinga, 2014) för att behandla och rita ut grafik.

När programmeringsspråket valdes så var det främst en sak som var viktig; vilket mediabiblioteket som skulle användas för att hantera grafiken i programmet. Det valdes mellan de två mediabiblioteken SDL och SFML (Gomila, 2014). En snabb implementation av medianfiltret med de två olika biblioteken visade att SDL ger tillgång till mer lågnivå-funktionalitet för hantering av grafik än vad SFML gör. Det har till exempel gjort det enklare att göra så att medianfiltret kan hantera godtycklig grafik i programmet, såsom en kombination av bakgrund och entiteter, men exkludering av text som ritas ovanpå. Eftersom SDL valts så används dessutom programmeringsspråket C++ på grund av att SDL har inbyggt stöd för det. Det finns även tillgång till wrappers för bland annat programmeringsspråket C#, men det ger inte tillgång till all funktionalitet som SDL erbjuder, därför anses C++ vara lämpligare.

4.1 GPU vs CPU

Det finns två val för hur medianfiltret ska implementeras: på CPU:n eller på GPU:n. Om medianfiltret implementeras med CPU:n så har det fördelen att det går att använda histogram på ett effektivt sätt för att snabba upp exekveringen. Histogram kan även användas på GPU:n, men då ger det inte nödvändigtvis bättre resultat än att inte använda det, som förklaras i kapitel 4.2.3. En till fördel med CPU:n är att implementationen blir något enklare eftersom det tillkommer några steg vid GPU-programmering. Till exempel måste argument som skickas till GPU-funktioner kopieras till GPU:ns minne innan funktionen anropas.

En test av medianfiltret som använde CPU:n visade på att exekveringen gick väldigt långsamt. Denna implementation skulle förmodligen kunna optimeras mycket för att göras snabbare, men istället gjordes valet att använda GPU:n, som enligt Malcolm (2010) har potential att beräkna medianfiltret mycket snabbare. GPU:n har en mycket lägre klockfrekvens än CPU:n men fördelen är att GPU:n kan bestå av hundratals eller tusentals processorkärnor vilket gör GPU:n väldigt bra på att göra parallella beräkningar. Eftersom ett medianfilter kan delas upp i många små enskilda delar, som kan beräknas samtidigt, passar det bra att använda GPU:n för det.

4.1.1 CUDA

För att kunna göra beräkningar på GPU:n används beräkningsplattformen och programmeringsmodellen CUDA. CUDA är varken ett programmeringsspråk eller ett API, så all programmering kommer fortfarande att ske i C++. CUDA ger ett enkelt sätt att göra allmänna beräkningar på GPU:n, genom att tillföra ett antal grundläggande nyckelord till de programmeringsspråk som stöds.

CUDA är utvecklat av NVIDIA och fungerar därför endast på NVIDIA processorer som är CUDA aktiverade, vilket är det som kommer att användas i det här arbetet. Detta är en nackdel eftersom det begränsar vilka system som programmet kan köra på, men det har inte något direkt betydelse för det här arbetet. Fördelen med att CUDA är utvecklat specifikt för NVIDIA-

processorer är att det förmodligen är väldigt bra optimerat och kommer resultera i snabbare program än en lösning som är byggd för många olika processorer.

4.2 Medianfilter

Medianfiltret implementeras som en GPU-funktion som anropas från CPU:n. Funktionen har parametrar för all information som behöver skickas till och från GPU:n. Figur 4 nedan visar funktionens alla parametrar. Parametrarna används till följande:

- *Uint32 *inputPixels:* En referens till en lista som innehåller alla pixlar från bilden som ska behandlas. Pixlarna representeras med positiva 32-bitars heltal där en byte står för röd, en för grön, en för blå och en för alfa. Listan används endast för att läsa av pixelvärdena i bilden och kommer därför inte att modifieras något under funktionens körning.
- *Uint32 *outputPixels:* En referens till en tom lista som är lika stor som bilden som blir behandlad. I listan placeras resultaten från medianfiltrets beräkningar. När hela funktionen är klar så hämtas det färdiga resultatet från den här listan.
- *Uint32 *sortPixels:* Ytterligare en pekare till en tom lista men som används för att sortera pixlarna i kärnan. Anledningen till att listan skickas som en parameter istället för att allokeras dynamiskt när funktionen exekveras är för att GPU-funktioner i CUDA behöver veta exakt hur mycket minne som kommer att användas av funktionen. Eftersom storleken på denna lista beror på medianfiltrets radie måste den därför definieras innan funktionsanropet.
- *int width:* Bredden av bilden i antal pixlar. Används för att veta var nästa rad i bilden börjar och även för att se till att funktionen inte försöker behandla pixlar som inte finns.
- *int height:* Höjden av bilden i antal pixlar. Används för att se till att funktionen inte försöker behandla några pixlar som inte finns.
- *int radius:* Den radie som medianfiltret ska använda. Radien används för att bestämma storleken på kärnan. Om radien till exempel är satt till 1 så blir kärnan 3x3, och om radien är satt till 2 så blir kärnan 5x5.

```
__global__ static void GPU_MedianFilter(Uint32 *inputPixels, Uint32 *outputPixels,
                                        int *width, int *height, Uint32 *sortPixels, int *radius)
{
   //Median filter implementation...
}
```

Figur 4 Definition av medianfiltrets funktion.

Funktionen anropas från CPU:n med alla nödvändiga argument samt en startkonfiguration som anger hur många block och trådar som ska användas för att köra funktionen på GPU:n. Startkonfigurationen sätts till 32 trådar per block och sedan beräknas antalen block utifrån hur många som behövs för att varje pixel ska bli tilldelad en tråd vardera. Antalet trådar som valts utgår från ett enkelt test som gjorts. I testet användes till en början max antal trådar, vilket är 1024, sedan halverades antalet iterativt för att se när uppdateringsfrekvensen var som högst. Resultatet blev då 32 trådar.

Medianfilter-funktionen börjar med att räkna ut ett unikt index genom att kombinera sitt tilldelade block-index och tråd-index. Detta värde används sedan för att låta varje tråd

behandla en unik pixel av bilden. Varje tråd räknar ut koordinaterna för den pixel som den motsvarar genom att använda sitt index och bredden av bilden som skickats med som argument till funktionen. Utifrån koordinaterna för pixeln samlas sedan alla närstående pixlar inom filtrets radie och läggs in i sortPixels-listan som skickats med till funktionen. Sedan sorteras pixlarna i listan så att medianen kan läsas av.

Medianfiltret implementeras med adaptiv skalärmedian enligt beskrivningen i kapitel 2.1.1. Alltså sorteras pixlarna en gång för varje färgkanal och den pixel som har medianvärdet för färgkanalen sparas tillfälligt undan. De sparade pixlarna bildar sedan implicit en matris som det går att läsa av en medianpixel ifrån. Medianpixeln bildas i det här fallet av medianvärdet för varje färgkanal. För att minska beräkningarna något så kommer alfavärdet för alla pixlar antas vara högsta möjliga (255) hela tiden. Därför behöver endast färgkanalerna röd, grön och blå sorteras.

4.2.1 Sorteringsalgoritm

Att välja sorteringsalgoritm är inte helt enkelt då dess effektivitet beror mycket på hur många element som ska sorteras och hur sorterade elementen redan är. I ett medianfilter kan antalet objekt som ska sorteras kraftigt variera, från att vara väldigt få till att vara väldigt många, beroende på hur stor radie som används. I ett inlägg av Wild (2012), på hemsidan stackexchange.com beskrivs det att sorteringsalgoritmen quicksort har ett väldigt bra genomsnittligt resultat, speciellt när det är många element som sorteras. Med många menas över cirka 100 element. Quicksort väljs därför som sorteringsalgoritm eftersom antalet element som sorteras överstiger 100 redan när filtrets radie är fem. Quicksortalgoritmen implementeras enligt den pseudokod som visas på Algorithmist (2015).

Eftersom quicksort inte nödvändigtvis är så snabb när endast ett fåtal element ska sorteras implementeras ytterligare en sorteringsalgoritm som tar hand om medianfiltrets låga radier. Den algoritm som valts för det kallas insertion sort. Anledningen till att insertion sort valts är att den är väldigt effektiv på att sortera ett fåtal element. Algoritmen implementeras enligt den pseudokod som visas på Wikipedia (2015).

För att avgöra när medianfiltret ska växla mellan de två sorteringsalgoritmerna gjordes ett test där programmets FPS jämfördes när de två olika algoritmerna användes. Testet visade att insertion sort presterade mycket bättre till en början, men när medianfiltret hade radie sex eller större så gav quicksort högre FPS. Därför använder medianfiltret insertion sort när radien är mellan ett och fem, och quicksort när radien är sex eller större.

4.2.2 Parallellisering

Eftersom GPU:n används för att köra medianfiltret fungerar inte implementationen som Bosakova-Ardenska m.fl. (2009) föreslår särskilt bra eftersom den endast använder ett fåtal trådar, och därmed inte använder GPU:ns kapacitet fullt ut. Istället används den GPU-parallellisering som beskrivs i slutet av kapitel 2.1.4, där det används ett stort antal trådar som endast tar hand om ett fåtal pixlar var. I den här implementationen så tar varje tråd endast hand om en pixel, alltså behövs det minst lika många trådar som antalet pixlar i bilden. Genom att göra detta så kan i princip alla pixlar i bilden beräknas samtidigt.

Varje tråd beräknar medianvärdet för pixeln den blivit tilldelad precis på samma sätt som tidigare. Det vill säga genom att samla ihop alla närstående pixlar inom radien och sortera dem. Skillnaden är att när medianvärdet beräknats för en pixel fortsätter inte tråden till nästa pixel, eftersom den behandlats samtidigt av en annan tråd.

4.2.3 Histogram

Nackdelen med att använda parallelliseringsmetoden som beskrivs i kapitel 4.2.2 är att det förhindrar användningen av histogram, eftersom varje tråd endast tar hand om en pixel. För att ha nytta av histogram är en CPU-implementation lämpligast, eftersom varje tråd då tar hand om ett stort antal pixlar var. En möjlighet är att låta trådarna på GPU:n också ta hand om ett flertal pixlar var, men det kommer förmodligen att minska effektiviteten även om histogram används. Anledningen till att effektiviteten förmodligen skulle minska är att antalet trådar som används minskar och varje tråd behöver utföra fler beräkningar. Om antalet trådar minskas kommer inte GPU:n användas fullt ut och om varje tråd dessutom behöver göra mer så kommer den totala exekveringstiden att öka. Detta gäller däremot främst om GPU:n inte har några andra uppgifter utöver att beräkna medianfiltret. Om GPU:n samtidigt gör många andra beräkningar är det möjligt att det är effektivare att ha färre trådar.

4.2.4 Dynamisk radie

För att implementera dynamisk radie används ett tröskelvärde som bestämmer när filtrets radie ska öka eller minska. Om programmets FPS är högre än tröskelvärdet så ökas filtrets radie ett steg, och likaså minskas filtrets radie om programmets FPS är lägre än tröskelvärdet. Ett problem med det här är att filtrets radie börjar fluktuera kring tröskelvärdet, eftersom när radien sänks ökar programmets FPS, vilket i sin tur leder till att radien ökas igen.

För att undvika detta används olika metoder för att öka eller minska radien. Att minska radien fungerar precis likadant som tidigare, det vill säga att om programmets FPS kommer under ett bestämt värde, till exempel 30 FPS, kommer medianfiltrets radie att minskas ett steg.

Att bestämma när filtrets radie kan öka ser däremot lite annorlunda ut. En ökning sker endast när man tror att programmets FPS fortfarande kommer överstiga tröskelvärdet efter att radien har ökats. Detta implementeras genom att det sker en estimering av hur hög programmets FPS kommer vara om filtrets radie ökas ett steg. Om det estimerade värdet överstiger tröskelvärdet så antas programmets FPS även vara högre än tröskelvärdet om radien ökas. För att göra estimeringen används tre värden: tiden det tog den senaste uppdateringen, tiden det tar att exekvera medianfiltret med den nuvarande radien och tiden det tar att exekvera medianfiltret med ett steg större radie. Den sistnämnda räknas ut genom att helt enkelt köra ett medianfilter med ett steg större radie än det nuvarande, ifall det inte redan finns ett sparat värde. En nackdel med detta är att programmet kan frysa till kort första gången en ny radie används, men om detta är ett problem så skulle det kunna undvikas genom att till exempel göra beräkningarna direkt programmet startas istället för under körningen. Figur 5 nedan visar implementationen för att ändra medianfiltrets radie dynamiskt.

```
void GameState::HandleDynamicRadius(float deltaTime)
{
   float estimatedFPS = 1.0 / (deltaTime - radiusExecutionTime[mFilterRadius] +
                                           radiusExecutionTime[mFilterRadius + 1]);
   if (mCurrentFPS < TARGET_FPS)
   {
      DecreaseRadius();
   }
   else if (estimatedFPS > TARGET_FPS)
   {
      IncreaseRadius();
   }
}
```

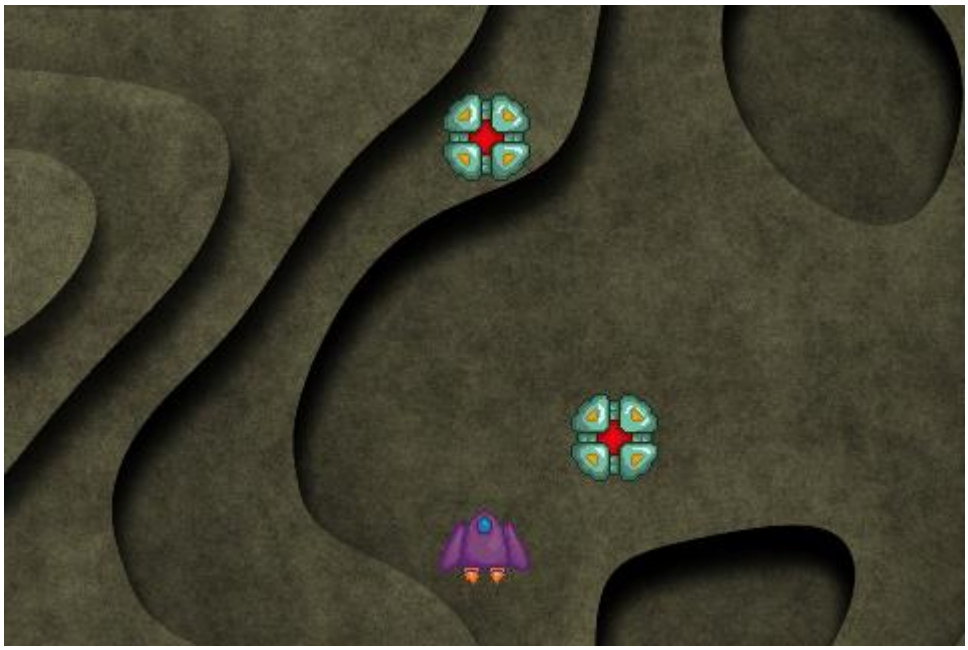
Figur 5 Implementation för dynamisk radie.

4.3 Testmiljö

För att testa och utvärdera lösningen har ett enkelt spel, som använder medianfiltret i realtid, utvecklats. Till spelet finns några snabbknappar som kan påverka programmet under körning för att underlätta testandet.

4.3.1 Spel

Det spel som byggts är en enkel variant av ett shoot'em up spel där det finns fiender som flyger över skärmen, samt en spelare som kan röra sig kontrollerat. Anledningen till att just ett shoot'em up spel valts är för att det går snabbt att implementera grunderna och det är även ett spel där belastningen potentiellt kan variera mycket eftersom det kan gå från att vara bara en spelare på skärmen till att vara ett stort antal fiender och skott. Detta är däremot inte fallet i det här exemplet. Figur 6 nedan visar en skärmdump av spelet.

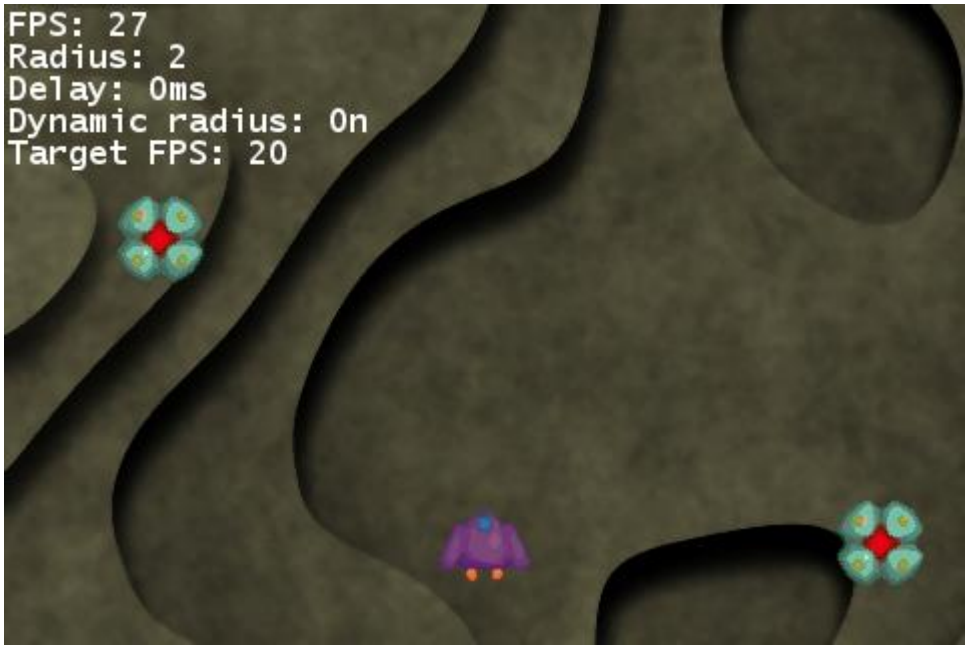


Figur 6 Skärmdump av testmiljön.

4.3.2 Utvärderingsverktyg

För att enkelt kunna utvärdera programmet så finns det snabbknappar för att göra ändringar i programmet under körning. Det finns en knapp som växlar mellan de två olika lägena på filtret, dynamisk radie eller fast radie. När fast radie används så finns det även knappar för att bestämma radien på filtret. Det finns två knappar för att öka eller minska en uppdateringsfördröjningsvariabel som används för att simulera förändring av processorbelastningen. Variabeln ökar helt enkelt tiden det tar för en uppdatering att ske med några millisekunder, vilket leder till att programmets FPS minskar. Det finns dessutom två knappar för att öka eller minska hur hög FPS den dynamiska radien ska försöka hålla.

För att kunna utvärdera programmet skrivs all intressant information ut i realtid under programmets körning. Det som är intressant är programmets FPS, uppdateringsfördröjning, den nuvarande radien på filtret, om den dynamiska radien är aktiverad eller inte och hur hög FPS den dynamiska radien ska försöka hålla. Figur 7 visar ytterligare en skärmdump av spelet men där informationen är utskriven på skärmen. All denna information sparas dessutom undan till en loggfil så att det går att undersöka resultaten steg för steg i efterhand. Resultaten sparas på ett sätt som gör det enkelt att importera till Microsoft Excel så att det enkelt går att visualisera resultaten.



Figur 7 Skärmdump med utskriven information.

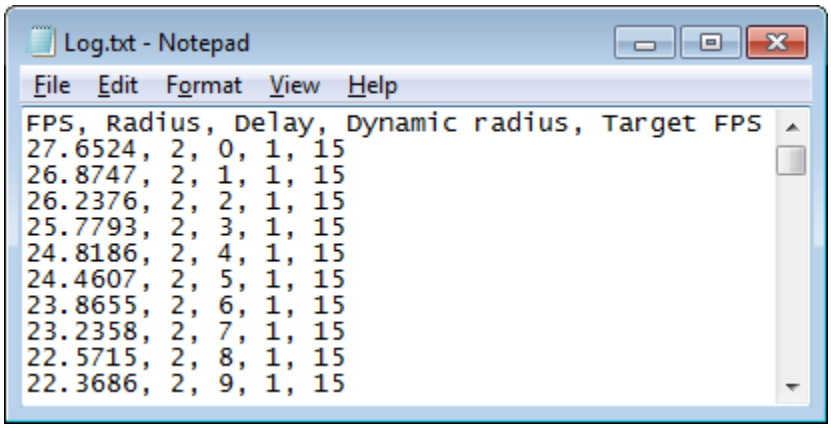
5 Utvärdering

Målet med det här arbetet var att jämföra två olika metoder för att anpassa medianfilter för användning i realtidsspel. Det här kapitlet utvärderar de resultat som uppmätts, samt beskriver de tester som utförts för att få fram resultaten. Först så beskrivs genomförandet av undersökningen, sedan presenteras mätresultaten och till sist görs en analys och en slutsats.

5.1 Presentation av undersökning

Undersökningen gick till så att det sattes upp ett antal olika situationer som det gjordes mätningar på. Resultaten från mätningarna skrevs ut till en loggfil. Det som mäts och skrivs ut till loggfilen är: programmets nuvarande FPS, medianfiltrets radie, en fördröjningsvariabel som används för att simulera varierande processorbelastning, om dynamisk radie används eller inte och hur hög FPS programmet ska hålla när dynamisk radie används. Den simulerade belastningen fungerar så att programmets huvudloop fördröjs med ett antal millisekunder varje uppdatering, vilket resulterar i att programmet får lägre FPS. Om till exempel fördröjningsvariabeln är satt till 10, så kommer varje uppdatering att ta 10 millisekunder längre tid på sig än vanligt. Figur 8 nedan visar ett exempel på en loggfil. Värdena i loggfilen är placerade i samma ordning som de beskrevs ovan. Om dynamisk radie används eller inte avgörs via en etta eller nolla. Är värdet noll används inte dynamisk radie, men om värdet är ett så används det. I exemplet används alltså dynamisk radie.

För att underlätta undersökningen automatiserades mätningarna delvis. En mätning påbörjas med en snabbknapp på tangentbordet. När en mätning är igång skrivs informationen, som nämndes ovan, ut till loggfilen varje uppdatering. Eftersom uppdateringar sker olika snabbt beroende på vilken radie och simulerad belastning som används så skulle antalet mätvärden skiljas mycket om mätningarna alltid skedde vid en viss tid. Därför skedde mätningarna istället i ett bestämt antal uppdateringar innan de automatiskt stängdes av. På så sätt gick det att spara lika många mätvärden varje gång. Medan en mätning pågår går det även att ändra variabler vid varje uppdatering. Om till exempel en mätning gjordes över 100 uppdateringar, så kunde man samtidigt välja att öka den simulerade belastningen med 1ms per uppdatering. På så sätt gjordes 100 mätningar medan den simulerade belastningen ökar från 0 till 100ms.

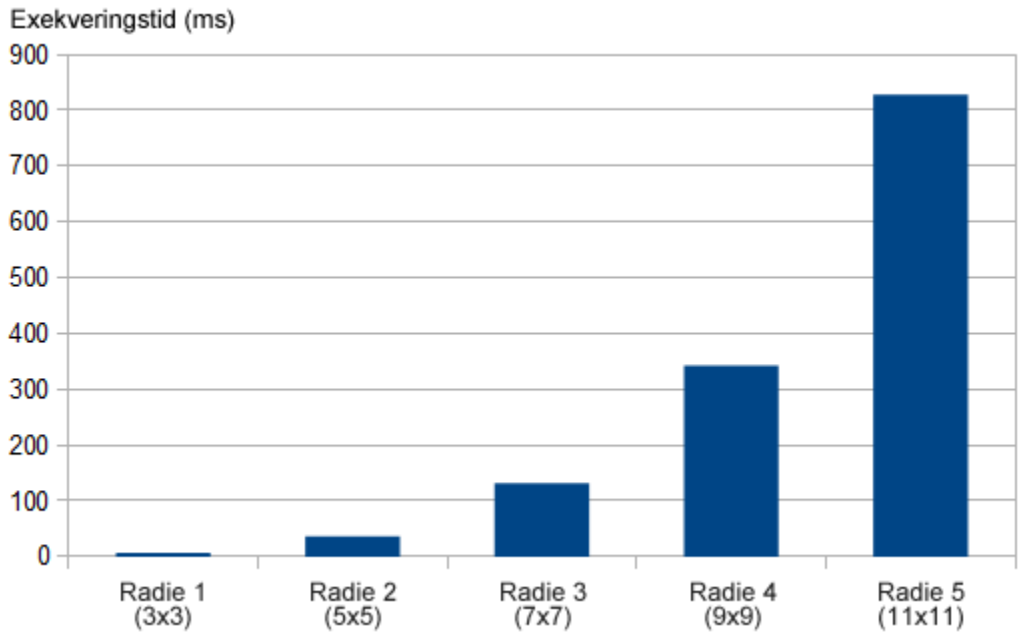


Figur 8 Exempel på mätresultat.

5.2 Resultat av mätningar

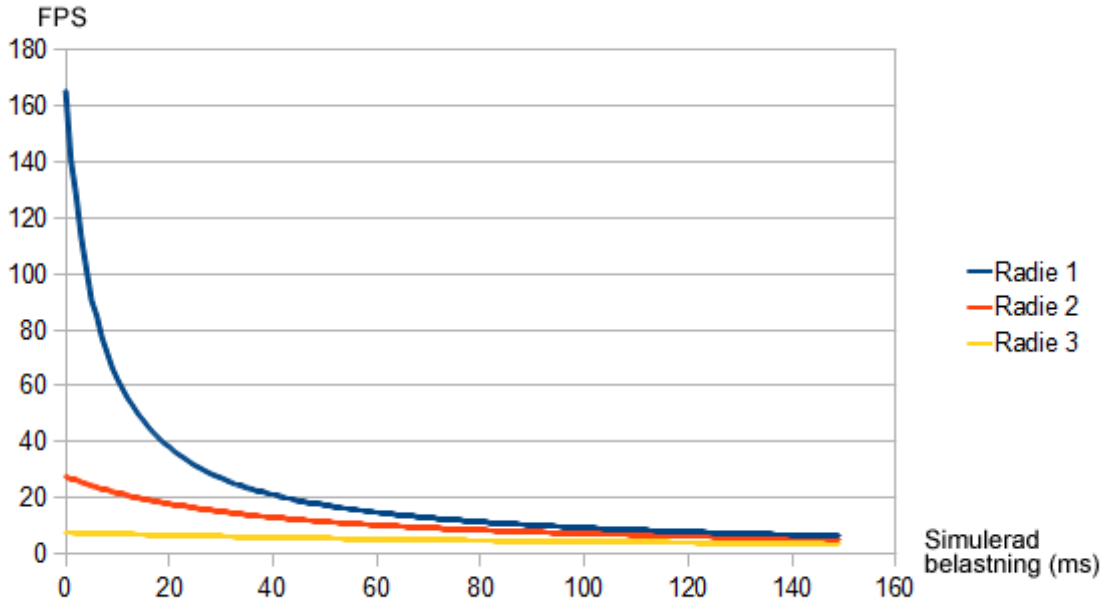
Denna del presenterar resultaten från de mätningar som gjorts. Först presenteras mätningar som gjorts på medianfilter med fast radie. Därefter visas resultaten för ett medianfilter med dynamisk radie. Sedan redovisas en jämförelse av de två metoderna och till sist görs en jämförelse av hur utseendet skiljer sig mellan olika radier.

Stapeldiagrammet i Figur 9 nedan visar en jämförelse av medianfilteralgoritmens exekveringstid när radierna 1 till 5 används. Värdena är baserade på medelvärdet av 100 mätningar. Diagrammet visar att exekveringstiden ser ut att öka kvadratiskt med medianfiltrets radie. Detta stämmer överens med det förväntade resultatet av algoritmen, eftersom antalet pixlar som behandlas, och därav antalet beräkningar som sker, ökar kvadratiskt med radien.



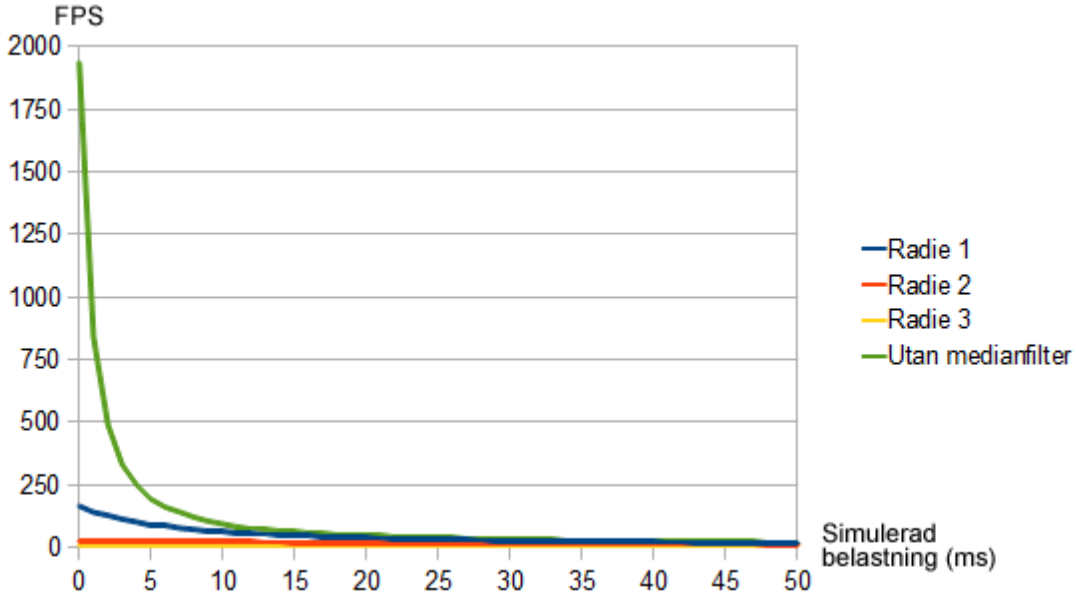
Figur 9 Stapeldiagram som jämför exekveringstiden av medianfilter med olika radie.

Figur 10 visar hur programmets FPS förändras när den simulerade belastningen ökar för ett medianfilter med fast radie. Figuren visar resultaten för radierna 1, 2 och 3 när den simulerade belastningen går från 0 till 150ms. Varje kurva är baserad på medelvärdet av 10 mätningar. Anledningen till detta är att få bort små ojämnheter från kurvorna så att de ser mjukare ut. Små ojämnheter kan uppstå till exempel på grund andra processer som behandlas av processorn samtidigt.



Figur 10 Sambandet mellan belastning och programmets FPS när fast radie används.

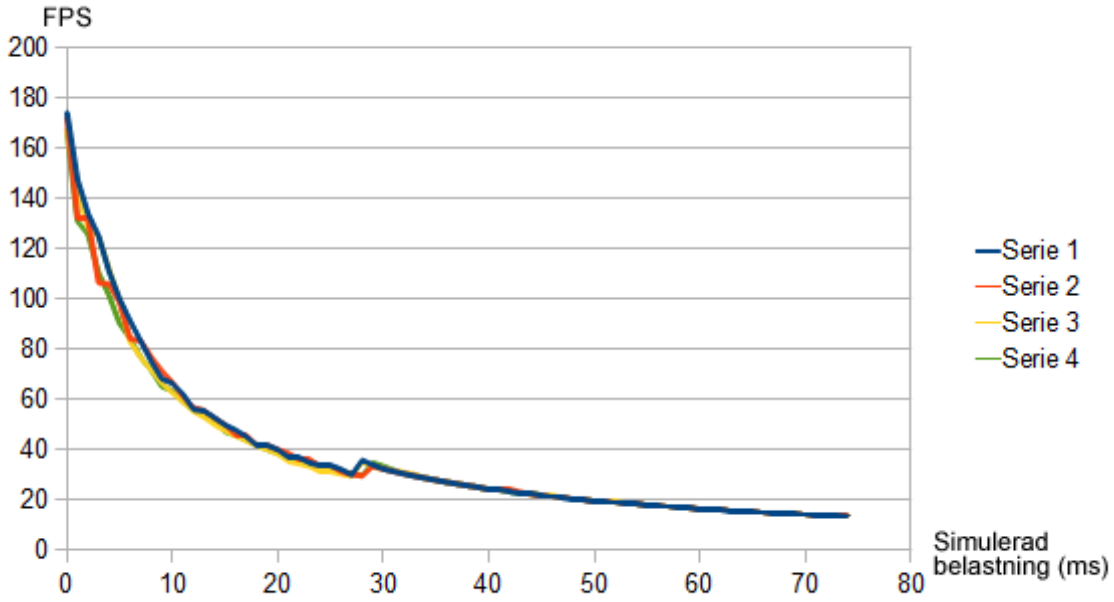
Något som också är intressant är hur programmets FPS skiljer sig när medianfilter används jämfört med när det inte används. Detta togs inte med i den föregående figuren på grund av att skillnaden i FPS är så stor att de andra kurvorna inte skulle bli lika tydliga. Figur 11 nedan visar samma diagram som i Figur 10, men inkluderar även programmets FPS när medianfiltret inte används



Figur 11 Skillnaden när medianfilter används eller inte.

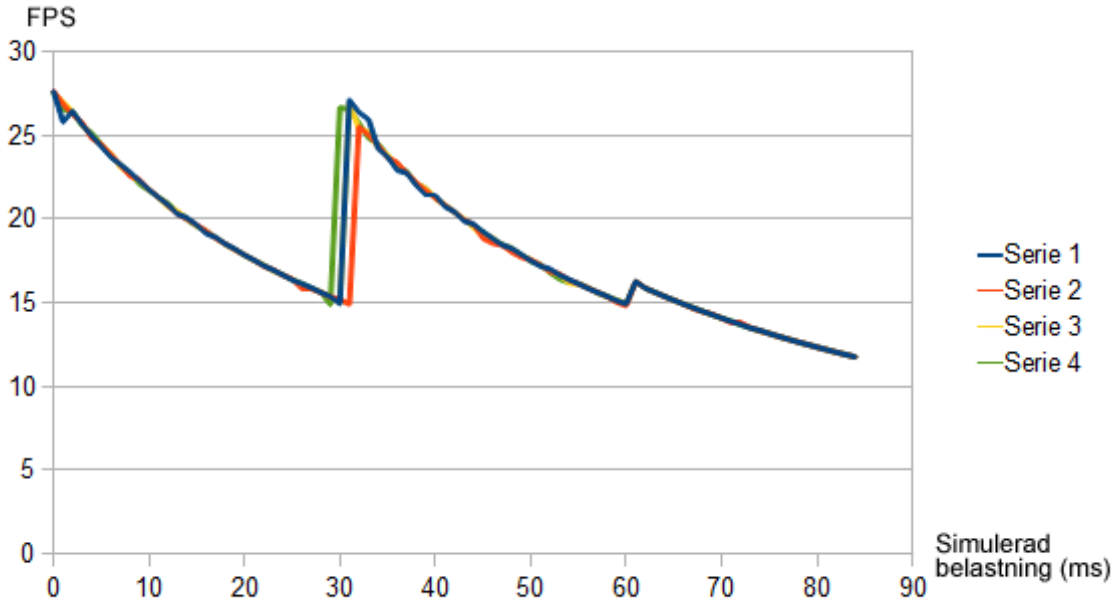
Figur 12 visar ett medianfilter som använder dynamisk radie och som siktar på att hålla över 30 FPS. Anledningen till att 30 FPS valts är för att det gör att medianfiltret till en början kan

hålla radie 1. När programmet sedan hamnar under 30 FPS stängs medianfiltret av, vilket leder till att programmets FPS ökar något. Efter det finns det inget mer att göra för att öka programmets FPS, så den fortsätter sjunka allteftersom belastningen ökar. Diagrammet visar värden från 4 olika mätningar. Fler mätningar utfördes, men på grund av att resultaten blev så lika visas endast 4 värden för att diagrammet ska vara tydligare.



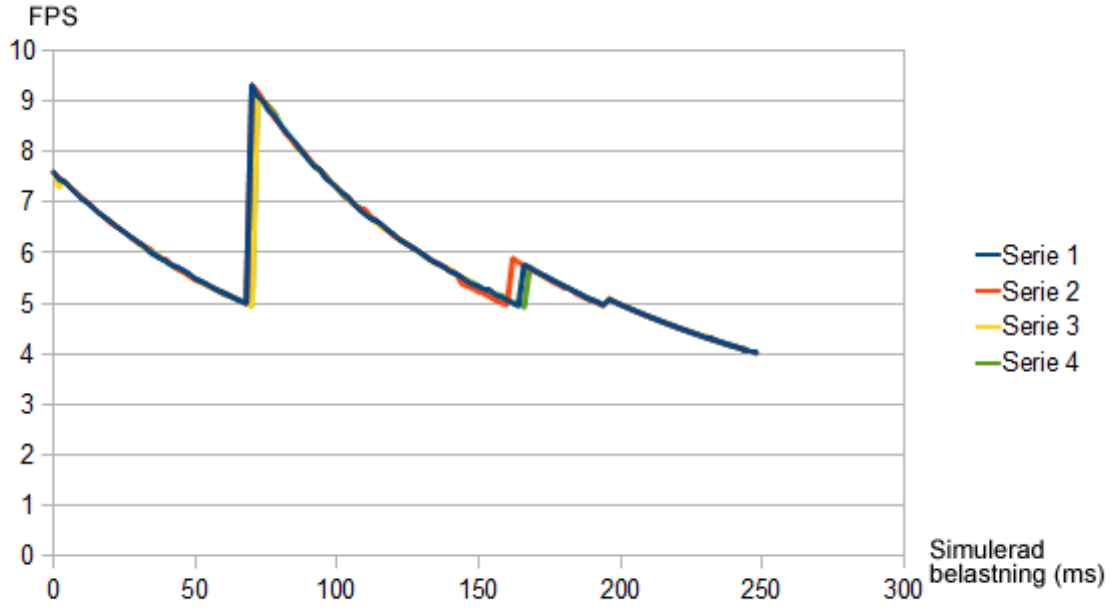
Figur 12 Sambandet mellan belastning och programmets FPS när dynamisk radie används som försöker hålla över 30 FPS.

Figur 13 nedan visar ett till medianfilter som använder dynamisk radie, men som istället försöker hålla en FPS högre än 15. Gränsen 15 används för att medianfiltret ska kunna hålla upp till och med radie 2. Att så är fallet går att härleda från Figur 10. När programmet hamnar under 15 FPS sänks först radien till 1 vilket ökar programmets FPS. Andra gången det händer stängs medianfiltret av vilket igen leder till att programmets FPS ökar lite. Detta diagram visar också resultaten av 4 mätningar, av samma anledning som tidigare.



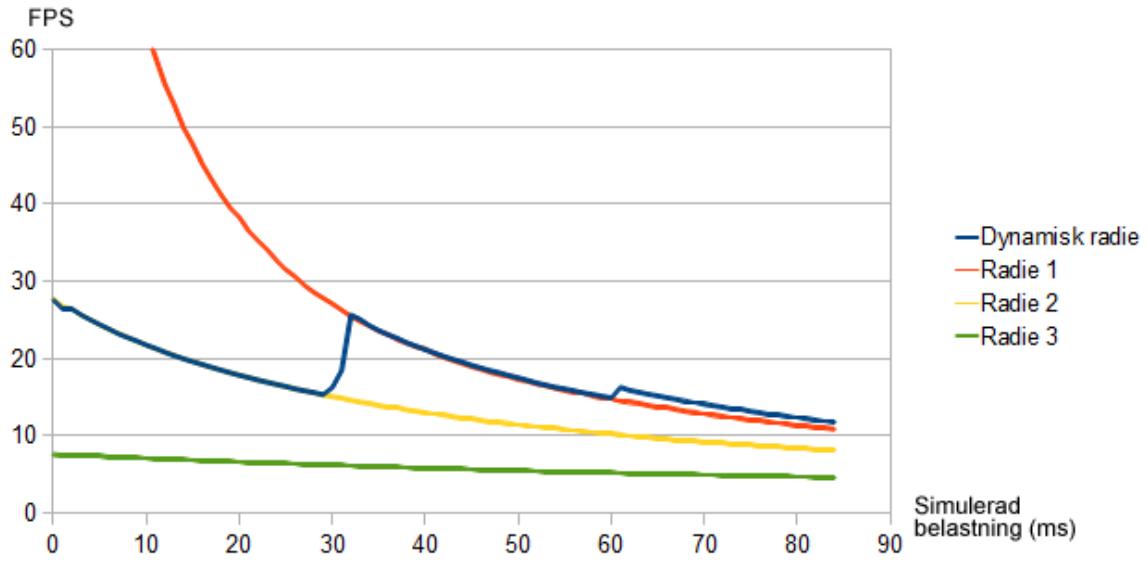
Figur 13 Sambandet mellan belastning och programmets FPS när dynamisk radie används som försöker hålla över 15 FPS.

Nästa figur (se Figur 14) visar ännu ett medianfilter med dynamisk radie. Den här gången försöker det hålla programmet över 5 FPS. Anledningen till att 5 FPS valts är för att medianfiltret då kan hålla en radie på upp till och med 3, vilket även här kan härledas från Figur 10. När ingen simulerad belastning är på får filtret radie 3. Sedan minskar radien till först 2, sedan 1 och till sist stängs medianfiltret av. Detta gör att kurvorna i grafen ökar vid tre separata tillfällen. Även detta diagram visar resultaten av 4 mätningar.



Figur 14 Sambandet mellan belastning och programmets FPS när dynamisk radie används som försöker hålla över 5 FPS.

Figur 15 visar en sammanställning av diagrammet i Figur 10, och diagrammet i Figur 13, det vill säga ett medianfilter med fast radie som visar radierna 1, 2 och 3, och ett medianfilter med dynamisk radie som ska hålla högre än 15 FPS. För tydlighetens skull används medelvärdet av mätningarna från Figur 13.



Figur 15 Jämförelse mellan Figur 10 och Figur 13. Det vill säga en jämförelse mellan medianfilter med fast radie respektive dynamisk radie.

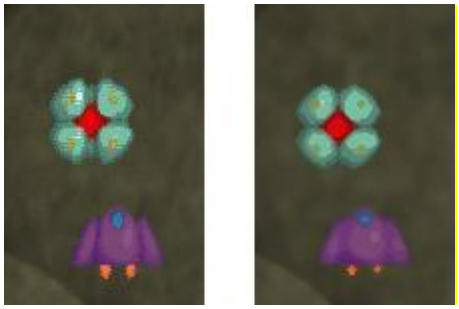
Till sist görs en jämförelse av hur utseendet på filtret skiljer sig mellan olika radier. Figur 16 nedan visar hur en bild går från att inte använda medianfilter till att använda medianfilter med radierna 1, 2, 3 och 4.

Figur 16 Jämförelse av utseendet för olika radier på medianfiltret.

5.3 Analys

Resultaten bekräftar att medianfiltret är väldigt beräkningstungt. Redan när radie 2 används får programmet lägre än 30 FPS i den här undersökningen. Detta beror delvis på att det inte gjorts så mycket optimeringar av implementationen. Men om medianfiltret ska kunna användas på ett bra sätt i realtidsspel så behöver det ske ganska stora förbättringar av effektiviteten, speciellt om man vill kunna använda högre radier.

Ett sätt man skulle kunna minska antalet beräkningar på när höga radier används är att använda varierande antal samplingspunkter. Ett medianfilter använder normalt sett en samplingspunkt per pixel. Så för ett medianfilter med radien 1 sker det 3x3, det vill säga 9 stycken, samplingar. Ett sätt man skulle kunna minska antalet beräkningar på är genom att använda färre samplingspunkter, men sprida ut dem över en större yta beroende på vilken radie som används. Till exempel skulle ett medianfilter med radie 3, som normalt sett använder 7x7 samplingspunkter, kunna använda lika många samplingspunkter som ett filter med radie 1, men utspritt över en större area. Detta skulle leda till mycket färre beräkningar. Problemet är att detta får konsekvenser för utseendet, eftersom alla värden inte räknas med. Ett snabbt test av detta gjordes för att se om idén verkar rimlig. Testet utgick från samma implementation som skapats i arbetet, men varannan samplingspunkt hoppades över i beräkningarna. Detta resultera i att programmets FPS var mycket högre än tidigare för alla radier, men det framgick att medianfiltret inte riktigt såg lika bra ut. Figur 17 nedan visar en jämförelse av ett medianfilter med radie 3, där vänster sida i figuren hoppar över varannan pixel medan höger sida fungerar som vanligt. Testet tyder på att det här skulle kunna vara ett möjligt sätt att öka prestandan på, men en mer genomgående utvärdering skulle behövas för att avgöra hur många färre samplingspunkter som kan användas utan att medianfiltret ser alltför dåligt ut. Det kan också finnas bättre metoder för att välja vilka pixlar som ska räknas bort än att bara hoppa över varannan pixel.



Figur 17 Jämförelse av medianfilter med olika många samplingspunkter. Till vänster visas ett medianfilter där varannan samplingspunkt hoppas över och till höger visas ett vanligt medianfilter.

Ytterligare ett sätt att öka prestandan på är genom att använda en snabbare metod för att hitta medianvärden. McGuire (2008) beskriver en metod som snabbt kan hitta ett medianvärde utan att sortera pixlarna. Metoden består av en fast uppsättning min- och maxinstruktioner som ser till att medianvärdet flyttas till mitten av en lista. En testimplementation som är baserad den implementation som McGuire (2008) har skapat visar att metoden resulterar i nästan dubbelt så hög FPS när radie 1 används än vad insertion sort, som använts i det här arbetet, ger. Detta är en väldigt stor förbättring. Problemet med metoden är att eftersom den består av en fast uppsättning instruktioner så fungerar implementationen endast för en bestämd radie. Alltså måste en unik implementation skapas för varje radie som ska användas, vilket gör att metoden inte är särskilt lämpad att använda med dynamisk radie. Om metoden skulle användas så skulle medianfiltret behöva begränsas så att endast vissa radier kan användas.

Diagrammen i Figur 12, Figur 13 och Figur 14 som visar medianfilter med dynamisk radie visar att det inte är någon större skillnad mellan resultaten av samma sorts mätning. Det man kan se är en liten variation när ett filter ändrar radie. Detta beror förmodligen på att estimeringen som görs för att bestämma när filtret ska öka eller minska i radie inte är helt exakt. Eftersom

estimeringen är baserad på tidigare mätningar av hur snabbt medianfiltret exekveras, så är det inte säkert att det stämmer överens med den nuvarande exekveringstiden.

Ett potentiellt problem till följd av detta, men som inte syntes när mätningarna gjordes, är att om medianfiltret av någon anledning skulle ta extra lång tid att exekveras när exekveringstiden mäts, så skulle det leda till att estimeringen blir felaktig. Om estimeringen är felaktig får det som konsekvens att medianfiltret byter radie vid fel tillfälle, eller inte alls. Att exekveringstiden skulle variera kan bero på många saker. Det kan till exempel bero på andra processer som använder CPU:n eller GPU:n samtidigt som medianfiltret, vilket gör att beräkningarna tar längre tid.

Som Figur 16 visar är det stor skillnad på utseendet mellan två olika radier. Detta gör att man ganska tydligt märker när radien förändras om dynamisk radie är aktiv. Anledningen till att man tydligt ser skillnad är att det är stora beräkningskillnader mellan varje steg av radie. En möjlig lösning till detta kunde vara att istället för att öka radien ett helt steg, vilket kvadratiskt ökar antalet undersökta pixlar, så skulle man kunna öka antalet undersökta pixlar med ett, på ett sådant sätt att man först lägger till alla pixlar inom radie 1, sedan radie 2 och så vidare. Detta borde logiskt sett resultera i en mjukare övergång när radien förändras, eftersom det inte sker lika stora steg. Ett följdproblem av detta är hur man väljer i vilken ordning pixlar som ska behandlas läggs till. Eftersom de undersökta pixlarna i ett vanligt medianfilter ökar uniformt på alla sidor så kan det vara lämpligt att försöka likna det så mycket som möjligt. En lösning kan därför vara att lägga till en pixel på varje sida i taget, så att man till exempel börjar lägga till en på vänster sida, sedan en på höger sida, sedan en på nedre sidan och till sist en på övre sidan och sedan repetera samma mönster tills en hel radie har fyllts ut.

I kapitel 2.1.1 beskrevs det att metoden som används för att hantera färgade bilder, adaptiv skalärmedian, kan resultera i att vissa pixlar blir missfärgade på grund av att det kan skapas nya pixlar som inte fanns med i originalbilden. Astola, Haavisto och Neuvo (1990) nämnde dessutom att de tycker det är viktigt att det inte skapas några nya pixlar, utan endast används pixlar från originalbilden av just den anledningen. Men som Figur 16 visar så går det inte tydligt att se några missfärgningar. Så att det skulle vara viktigt att alla pixlar kommer från originalbilden kanske man får ta med en nypa salt. Att inga missfärgningar syns kan också delvis bero på att spelet som använts i utvärderingen inte är särskilt färgglatt.

5.4 Slutsatser

Resultatet tyder på att metoden som använder dynamiskt varierande radie är bättre på att hålla en stabil FPS när processorbelastningen varierar. Detta kan man bland annat se av jämförelsen i Figur 15 där man ser att filtret som använder dynamisk radie håller sig över det bestämda FPS-värdet mycket längre än filtren som använder fast radie. Dock bör det noteras att den endast slår filtret med radie 1 genom att helt stänga av filtret när programmets FPS blir för låg. Om detta bör göras eller inte beror till stor del på vad det är för typ av applikation som medianfiltret används på. Om applikationen fungerar i stort sett lika bra utan medianfiltret så är det förmodligen bättre att det stängs av för att behålla en bra FPS längre. Men om applikationen är beroende av medianfiltret så är det bättre att låsa filtret så att radie 1 är det minsta det kan ha.

Den enda skillnaden utseendemässigt mellan de två metoderna är att metoden som använder fast radie alltid kommer se likadan ut under körningen, medan den metod som använder dynamisk radie förändras om belastning i programmet ändras. Som beskrevs i analysdelen

ovan kan man ganska tydligt se när radien förändras om dynamisk radie används. Om detta är något som sker väldigt sällan under spelets gång så är det inte direkt något man lägger märke till. Men om radien förändras väldigt frekvent kan det lätt uppfattas som störande.

Slutsatsen är att om man med stor sannolikhet vet att ett spel kommer klara av att behålla tillräckligt hög FPS under hela körningen när en viss radie används, så är det förmodligen bättre att använda fast radie. Ett undantag är om det är viktigt att hela tiden hålla så hög radie på filtret som möjligt. Om man däremot har ett spel där belastningen kan variera kraftigt och det är sannolikt att programmet ibland får väldigt låg FPS kan det vara bättre att använda dynamisk radie på medianfiltret, förutsatt att det inte spelar så stor roll att det går att se när radien ändras.

6 Avslutande diskussion

Detta kapitel börjar med att sammanfatta arbetet utifrån den problemformulering och det resultat som erhållits. Sedan sker en diskussion kring arbetet och till sist avslutas det hela med tankar kring framtida arbeten inom området.

6.1 Sammanfattning

Syftet med det här arbetet var att anpassa medianfilter för användning i realtidsspel. För att göra det jämfördes två olika metoder för att bestämma filtrets radie. Ju större radie som används, desto längre blir exekveringstiden för algoritmen, på grund av att antalet beräkningar ökar. Den första och enklaste metoden går ut på att bestämma en fast radie som kommer att användas under hela programmets körning. Radien sätts då lämpligast till ett värde som gör att programmet behåller en acceptabel FPS genom hela körningen.

Den andra metoden går ut på att bestämma medianfiltrets radie baserat på programmets nuvarande FPS. Om programmet har väldigt låg FPS minskas radien vilket får programmets FPS att öka, och likaså ökas radien om programmet har väldigt hög FPS. På så sätt kan medianfiltret i många fall använda en högre radie än om fast radie skulle användas, och på samma gång automatiskt anpassa sig ifall programmet oväntat minskar i FPS.

Denna problemformulering besvaras genom att det utförs ett experiment. Experimentet gick till så att det skapades en implementation av ett medianfilter som kan använda de två olika metoderna för att ändra radie. Den färdiga implementationen används sedan för att utvärdera problemet. Det som utvärderas är vilket av metoderna som är bäst anpassad för att användas i realtidsspel. Detta bestäms genom att jämföra hur programmets FPS förändras när de olika metoderna används. Den metod som bäst håller en stabil FPS när processorbelastningen varierar anses vara lämpligast. Något som också kommer utvärderas är hur programmets utseende skiljer sig när de två olika metoderna används. Utseendet är däremot inget som ligger i fokus för undersökningen, men det är ändå viktigt att tänka på om medianfiltret ska användas i spelsammanhang.

För att utvärdera problemet görs det mätningar på implementationen. Mätningarna går till så att all intressant information skrivs ut till en loggfil när programmet körs. Det som skrivs är programmets FPS, medianfiltrets radie, den fördröjningsvariabel som används för att simulera varierande processorbelastning, om dynamisk radie används eller inte och vilken FPS programmet ska försöka hålla när dynamisk radie används. Utseendet av medianfiltret är inget som direkt går att mäta. Detta jämförs därför endast visuellt med fokus på hur de två metoderna skiljer sig under körningen.

Resultaten av mätningarna tyder på att metoden som väljer radie dynamiskt är bättre på att hålla en stabil FPS när processorbelastningen varierar. Metoden behåller tillräckligt hög FPS längre i alla situationer än metoden som använder fast radie. Däremot så slår den endast ett filter som använder radie 1 genom att stänga av medianfiltret när programmets FPS blir för lågt. Det enda som skiljer sig utseendemässigt mellan de två metoderna är att metoden med fast radie ser likadan ut under hela körningen, medan metoden som ändrar radie dynamiskt varierar i utseende under körningen. Eftersom det är stora beräkningsskillnader mellan två olika radier, går det tydligt att se när ett medianfilter med dynamisk radie förändras. Om detta är något som sker väldigt frekvent så kan det därför upplevas störande.

6.2 Diskussion

Som nämndes i kapitel 2.1.2 finns det många olika sätt för att låta medianfilter hantera färgade bilder. Koschan och Abidi (2001) skriver om ett antal av dem i sin workshop. Den metoden som använts i det här arbetet, adaptiv skalärmedian, valdes främst för att den är enkel att förstå och att implementera. Adaptiv skalärmedian är däremot väldigt beräkningstung eftersom den använder ett medianfilter per färgkanal. I det här arbetet användes färgkanalerna röd, grön blå och alfa för utritningen. Men för att spara beräkningar ignorerades alfakanalen. Detta betyder att det sker tre separata sorteringar per pixel i bilden, eftersom varje färgkanal sorteras var för sig. Eftersom sorteringen är en av de beräkningstyngsta delarna av algoritmen kan det vara värt att pröva någon av de andra metoderna som Koschan och Abidi (2001) beskriver, som endast utför en sortering per pixel.

I kapitel 5.2 noteras det att medianfiltrets exekveringstid ser ut att öka exponentiellt. Det här stämmer även överens resultaten som fåtts av Perrot, Domas & Couturier (2014). I sin journal nämner de att för en bild som är 512x512 pixlar stor, så är exekveringstiden följande: 0.05ms för radie 1, 0.19ms för radie 2 och 0.6ms för radie 3. Medianfiltret i det här arbetet har använts på en bild som är 480x320 pixlar stor, men trots att bilden är mindre så är exekveringstiden betydligt högre än den som Perrot, Domas & Couturier (2014) har fått. Det här beror nog till stor del på att implementationen som Perrot, Domas & Couturier (2014) har gjort används på en gråskalig bild, vilket gör att några av de tyngsta operationerna som medianfiltret i det här arbetet gör, inte behövs. Det kan även betyda att deras implementation är väldigt väl optimerad, jämfört med implementationen i det här arbetet.

Histogram nämndes som en metod som kan användas för att snabba upp exekveringen av medianfilter i kapitel 2.1.3. Histogram användes inte vid implementationen av medianfiltret eftersom det inte fungerade tillsammans med den parallelliseringsmetod som använts. Ett stort problem med att använda medianfiltret i realtidsspel är att tidskomplexiteten ökar kvadratiskt med radien, vilket gör att det endast går att använda några av de lägsta radierna innan exekveringstiden blir för hög. Perreault och Hébert (2007) nämnde en speciellt intressant variant av histogram som sägs kunna hålla konstant tidskomplexitet, O(1), per pixel. Om denna metod används så borde det inte skilja lika mycket i exekveringstid mellan olika radier, vilket skulle göra det möjligt att använda många fler radier. Detta förutsatt att det går att implementera tillräckligt effektivt för att kunna köras i realtid.

I problemformuleringen (kapitel 3) nämndes en undersökning av Claypool och Claypool (2007) som diskuterar vikten av att ha hög FPS i spel. I undersökningen nämndes inget exakt tal för hur mycket FPS som behövs för att spelet ska flyta på bra. Men någonstans under 30 FPS brukar det tydligt gå att se när en ny bildruta ritas ut, genom att objekt ser ut att förflytta sig hackigt. Som visades av mätresultaten (kapitel 5.2) så är det endast radie 1 som håller sig en bra bit över 30 FPS. Radie 2 ligger strax under 30 FPS och radie 3 ligger mellan 5 och 10 FPS. Det är alltså endast radie 1 och kanske radie 2, som rimligen skulle kunna användas i realtidsspel i den här undersökningen. Resultaten från undersökningen borde däremot skala väldigt bra om prestandan skulle förbättras, så även om bättre hårdvara skulle användas eller att implementationen skulle optimeras bättre borde resultaten här kunna användas.

Implementationen av medianfiltret har i det här arbetet använt CUDA för att göra beräkningar på GPU:n. Men det finns andra sätt som det skulle kunna implementeras på. Ett sätt är att skapa filtret som en shader. En shader är ett litet program som normalt sett också kör på GPU:n och som används för att rita något på ett specifikt sätt. Shaders används ofta inom spel för att till exempel framställa ljus, skuggor, oskärpa, kontrast och mycket mer. En fördel med

shaders är att de kan använda textursampling till en väldigt låg kostnad, vilket är bra om man behöver göra många samplingar. Detta är något som skulle kunna påverka medianfiltrets prestanda positivt, speciellt vid högre radier. Ytterligare en fördel med shaders är att de inte är specifikt bundna till en viss processortyp, till skillnad från CUDA som endast fungerar på NVIDIA-processorer. Anledningen till att implementationen ändå gjorts med CUDA istället för en shader beror på att CUDA är mer flexibelt. CUDA kan utföra allt som en shader kan och har även en del funktionalitet utöver det. Detta gör att CUDA har potential att vara lika snabbt som en shader, och i vissa fall även snabbare. Några av fördelarna med CUDA är att det går att anpassa lastbalanseringen mellan trådar och det går att välja hur många trådar och block som ska användas vid exekveringen. Det skulle vara intressant att göra en likadan implementation som gjorts i den här undersökningen, men som en shader, och jämföra hur de olika metoderna står sig mot varandra.

Medianfiltret har implementerats med samtidiga processer, genom att det skapas lika många trådar som antalet pixlar i bilden. Detta betyder att varje tråd endast tar hand om en pixel. Då dagens GPU:er består av några hundra eller upp till några få tusen processorkärnor, så kommer det inte finnas tillräckligt med kärnor för att varje kärna endast ska behöva behandla en tråd, om det inte är en väldigt liten bild som behandlas. Detta betyder att exekveringen skulle kunna bli mycket snabbare om fler processorkärnor används, helst lika många som antalet pixlar i bilden. Det här gör att implementationen är väl anpassad för framtida GPU:er som är kraftfullare än de som finns tillgängliga idag. Implementationen är däremot inte nödvändigtvis optimal för dagens GPU:er, eftersom varje kärna behöver ta hand om ett flertal trådar. En bättre lösning skulle kunna vara att istället låta varje tråd ta hand om ett antal pixlar var, till exempel baserat på antalet pixlar i bilden dividerat med antalet processorkärnor.

Trovärdigheten av resultaten är något som skulle kunna diskuteras. Som skrevs i kapitel 5.2 så gjordes ett ganska begränsat antal mätningar. Det går säkerligen att få exaktare resultat om alla mätningar skulle göras om flera gånger. Men resultaten har ansetts vara tillräckliga för att kunna besvara problemformuleringen och dra en slutsats ifrån. Jämförelsen som gjorts av hur medianfiltrets utseende skiljer sig mellan de två olika metoderna kan också ifrågasättas. Eftersom utseendet inte är så lätt att mäta gjordes endast en visuell jämförelse, vilket gör att resultatet är subjektivt. Resultaten pekar på att det kan upplevas störande med dynamisk radie om radien förändras väldigt ofta, eftersom det tydligt går att se när radien ändras. Det är däremot inte säkert att alla upplever detta som störande. Det är ganska vanligt förekommande att någon som arbetar med något grafiskt, lägger märke till, eller stör sig på, något som en genomsnittlig användare inte märker av. För att få ett noggrannare resultat skulle det behöva göras en större undersökning med test på riktiga användare som ger sin uppfattning av utseendet. Detta har däremot inte ansetts nödvändigt för det här arbetet eftersom utseendet av medianfiltret inte är det som undersökningen fokuserar på.

6.2.1 Etiska och samhälleliga aspekter

De etiska och samhälleliga aspekterna av det här arbetet är ganska få. Arbetet kan användas som referens vid spelutvecklingssammanhang för att introducera en metod som sällan används inom spel. Medianfilter skulle i vissa fall kunna användas inom spel som alternativ till filter för oskärpa, till exempel för att få bort fokus från spelet när en meny kommer upp i förgrunden. Ett annat användningsområde är att censurera känsligt innehåll. Metoden som används för att anpassa grafiken i spelet efter programmets FPS skulle också kunna användas till andra saker än just medianfilter. Dessutom kan resultaten av arbetet användas för vidare

forskning inom ämnesområdet. Några intressanta saker man skulle kunna undersöka tas upp i kapitel 6.3 nedan.

I undersökningen så har implementationen gjorts på en stationär dator med operativsystemet Windows 7. Ett problem med detta är att långt ifrån alla enheter i världen har samma specifikationer, vilket betyder att lösningen som tagits fram i den här undersökningen kanske inte fungerar lika bra på andra enheter. Det finns bland annat en stor grupp mobila enheter som kör operativsystem som Android och iOS, som ofta har svagare hårdvara än en genomsnittlig stationär dator. Mobila enheter har i nuläget normalt sett en kraftfull CPU, men ofta en relativt svag GPU. Med tanke på detta är det inte säkert att lösningen i det här arbetet fungerar så bra för mobila enheter, då lösningen förlitar sig mycket på GPU:n. Det här arbetet kan nog ändå användas som grund för att implementera medianfilter i realtid på andra enheter, men det kan behöva göras vissa förändringar för att det ska fungera bra. Exempelvis kan det vara bättre att göra beräkningarna på CPU:n istället för GPU:n, eller kanske till och med en hybrid av båda.

Alla program och bibliotek som använts i arbetet är öppna för allmänheten och är, förutom i ett fall, helt gratis. Det ända undantaget är programutvecklingsmiljön Microsoft Visual Studio Ultimate 2012, men det finns en motsvarande gratis version som heter Microsoft Visual Studio Express 2012 som fungerar lika bra för allt som skapats i det här arbetet. Detta är bra forskningsetiskt eftersom det gör att hela arbetet är återupprepningsbart. Dessutom uppdateras frekvent både Visual Studio, SDL, SFML och CUDA, vilket gör att arbetet kommer att kunna återupprepas även en bra bit in i framtiden. En begränsning med arbetet är att det använder CUDA för att göra beräkningar på GPU:n. Av denna anledning krävs det att den som återupprepar arbetet har en NVIDIA processor som är CUDA-aktiverad.

6.3 Framtida arbete

En intressant metod som diskuterades i analysen (kapitel 5.3) är att använda färre samplingspunkter men som är utspridda över en större yta, för att minska antalet beräkningar av algoritmen. Vanligtvis ökar antalet samplingspunkter kvadratiskt med radien som används, vilket gör att medianfiltret blir i princip oanvändbart i realtidsspel redan efter några av de första radierna. Om man skulle minska antalet samplingspunkter skulle man potentiellt kunna använda mycket högre radier. Detta får däremot konsekvenser för utseendet av medianfiltret eftersom alla mätpunkter som ett vanligt medianfilter skulle räkna med inte kommer hanteras. Frågan är om detta går att göra utan att medianfiltret tappar alltför mycket kvalitet.

Ett av de största problemen med medianfilter är att tidskomplexiteten ökar exponentiellt med radien. Något som möjligtvis skulle kunna minska tidskomplexiteten är att upprepa ett medianfilter med låg radie flera gånger istället för att öka radien. Detta skulle kunna ge ett liknande resultat som ett medianfilter med stor radie, men med färre beräkningar, eftersom det totalt är färre pixlar som behandlas.

En annan metod som också togs upp i analysen är att öka antalet pixlar som undersöks en i taget, istället för att de ökas kvadratiskt med radien. På så sätt skulle ett stort problem med att använda medianfilter i realtid försvinna, eller i varje fall minskas, nämligen problemet att det är väldigt stor beräkningsskillnad mellan varje radie, vilket gör det svårt att veta när radien kan ändras så att programmet fortfarande behåller en bra FPS. Detta löstes i det här arbetet med en estimering av programmets FPS, men som diskuterades i analysen så finns det vissa problem med att göra på det viset. Ytterligare ett problem med att det är sådan stor skillnad

mellan radier är att det tydligt går att se när radien ändras. Om man istället skulle öka antalet undersökta pixlar med en i taget så borde det resultera i en mjukare övergång.

Något som också skulle vara intressant är att jämföra en implementation som liknar den som skapats i det här arbetet med en implementation som använder histogram för att se vilken metod som resulterar i bäst prestanda. Speciellt histogrammet som beskrivs av Perreault och Hébert (2007) som sägs ha tidskomplexitet O(1) per pixel. Om denna metod används så borde det inte bli så stor beräkningsskillnad mellan olika radier, så frågan är vad som skulle hända om det användes tillsammans med dynamisk radie. Om skillnaden mellan radier är tillräckligt liten så finns det en risk att radien ökas till ett väldigt högt värde, vilket inte nödvändigtvis är önskvärt eftersom det kan leda till att bilden blir alldeles för otydlig. Men om det är ett problem går det enkelt att lösa genom att sätta en maxgräns på radien.

Trots att det här arbetet har behandlat medianfilter i just realtidsspel så skulle resultaten av undersökningen kunna användas för mycket mer än bara spel. Ett stort användningsområde är datorseende, som handlar om att analysera och förstå bilder. Resultaten skulle kunna användas vid utveckling av robotar eller självkörande bilar, till exempel för att i realtid reducera brus som kan framkomma av mörker eller signalstörningar.

Ett annat exempel kan vara att underlätta konvertering av pixel-baserade bilder till vektorgrafik. Genom att använda medianfilter skulle man kunna förenkla pixel-baserade bilder för att göra det enklare att beskriva bilderna som vektorgrafik. Det här skulle kunna göras i realtid för att till exempel konvertera en pixel-baserad videofilm till en vektor-baserad, vilket skulle kunna reducera mängden data som behövs för att beskriva videon. Detta kan till exempel vara användbart för att strömma video över internet när internetuppkopplingen är ostabil eller långsam, som bland annat är vanligt förekommande för mobila enheter.

Som nämndes i diskussionen så är det inte ett helt självklart val att använda CUDA för att implementera medianfilter, utan man skulle kunna använda en shader istället. Det är inte helt uppenbart vilken metod som skulle kunna resultera i den effektivaste implementationen, så det är något som skulle vara intressant att undersöka undersöka. Det kan också finnas mer än bara effektiviteten som spelar roll, som till exempel hur lätt det är att implementera eller om utseendet av medianfiltret skiljer sig något mellan metoderna.

Ytterligare något som nämndes i diskussionen var att resultatet av medianfiltrets utseende när dynamisk radie används skulle kunna ifrågasättas, eftersom resultatet är subjektivt. Slutsatsen var att det kan upplevas störande när dynamisk radie används eftersom det går att se när filtret ändrar radie. Det är däremot inte säkert att en genomsnittlig användare som inte specifikt granskar spelets utseende skulle lägga märke till att radien ändras. Det skulle därför vara bra att göra en undersökning som gör tester på ett flertal slutanvändare för att se om det här faktiskt är ett allvarligt problem eller inte.

Det vore även intressant att använda resultaten för att implementera medianfilter i realtid i till exempel spelmotorer eller andra applikationer. Detta skulle exempelvis kunna användas som ett filter i en kameraapplikation till mobiltelefoner, eller för att ge grafiken i ett spel en unik stil.

Referenser

Algorithmist. (2015) *Quicksort*. Tillgänglig på Internet: http://www.algorithmist.com/index.php/Quicksort [Hämtad 9 april, 2015]

Andrews, G.R. & Schneider F.B. (1983) Concepts and Notions for Concurrent Programming. *ACM Computing Surveys*. 15 (1), s. 3-43.

Astola, J., Haavisto, P. & Neuvo, Y. (1990) Vector median filters. *Proceedings of the IEEE*. 78 (4), s.678-689.

Bagni, D. (2014) Median Filter and Sorting Network for Video Processing with Vivado HLS. *Xcell Journal*. 86, s. 20-28.

Bailey, R.A. (2008) *Design of Comparative Experiments*. Cambridge, Cambridge University Press.

Bosakova-Ardenska, A., Stoichev, S., Vasilev, N. & Stoicheva, E. (2009) Fast Median Filtering Based on Bucket Sort. *Journal of Information Technologies and Control*. 2.

Claypool, K.T. & Claypool, M. (2007) On frame rate and player performance in first person shooter games. *Multimedia Systems*. 13 (1), s. 3-17.

Gomila, L. (2014) *SFML – Simple and Fast Multimedia Library* (Version 2.1) [Multimedia-API] Tillgänglig på Internet: http://www.sfml-dev.org/ [Hämtad 5 december, 2014]

Hoesly, P. (2011) iPhone Background – Chipboard [Digital bild]. Licensierad enligt CC BY 2.0. Tillgänglig på Internet: http://www.flickr.com/photos/zooboing/5417691934/in/photostream/ [Hämtad 31 mars, 2015] Författare: http://www.patrickhoesly.com/ [Hämtad 31 mars, 2015] Licens: http://creativecommons.org/licenses/by/2.0/ [Hämtad 31 mars, 2015]

Koschan, A. & Abidi, M. (2001) A Comparison of Median Filter Techniques For Noise Removal in Color Images. *Proc. 7th German Workshop on Color Image Processing*. 34 (15), s. 69-79.

Lantinga, S. (2014) *SDL - Simple DirectMedia Layer* (Version 2.0.3) [Multimedia-API] Tillgänglig på Internet: https://www.libsdl.org/ [Hämtad 12 mars, 2015]

Malcolm, J. (2010) Median Filtering: CUDA tips and tricks. *GPU and Accelerator Software Blog,* 4 mars. Tillgänglig på Internet: http://blog.accelereyes.com/blog/2010/03/04/median-filtering-cuda-tips-and-tricks/ [Hämtad 31 mars, 2015]

McGuire, M. (2008) A Fast, Small-Radius GPU Median Filter. I Wolfgang Engel. *ShaderX6: Advanced Rendering*. *s*. 165-174. New York: Delmar Publishing.

Perreault, S. & Hébert, P. (2007) Median Filtering in Constant Time. *IEEE Transactions on Image Processing*. *16 (9)*, s. 2389-2394.

Perrot, G., Domas, S. & Couturier, R. (2014) Fine-tuned High-speed Implementation of a GPU-based Median Filter. *Journal of Signal Processing Systems*. 75 (3), s. 185-190.

Singh, K.M. & Bora, P.K. (2004) Adaptive vector median filter for removal of impulse noise from color images. *Journal of Electrical & Electronics Engineering*. 4 (1).

Weiss, B. (2006) Fast Median and Bilateral Filtering. *Proceedings of ACM SIGGRAPH 2006 Transactions on Graphics*. 25 (3), s. 519-526.

Wikipedia. (2015) *Insertion sort*. Tillgänglig på Internet: http://en.wikipedia.org/wiki/Insertion_sort [Hämtad 9 april, 2015]

Wild, S. (2012) Why is quicksort better than other sorting algorithms in practice? *StackExchange – Computer Science,* 7 mars. Tillgänglig på Internet: http://cs.stackexchange.com/questions/3/why-is-quicksort-better-than-other-sorting-algorithms-in-practice [Hämtad 9 april, 2015]