Examensarbete

HÖGSKOLAN I SKÖVDE

FÄRGBLINDKORRIGERING I REALTID

COLORBLIND CORRECTION IN REALTIME

Examensarbete inom huvudområdet Informationsteknologi
Grundnivå 30 högskolepoäng
Vårtermin 2017

Tobias Löf Melker

Handledare: Peter Sjöberg
Examinator: Henrik Gustavsson

Sammanfattning

Målet med denna studie är att utvärdera möjligheten att göra färgkorrigering i realtid. Denna färgkorrigering ska hjälpa färgblinda att lättare urskilja mellan olika färger. Ett problem med färgkorrigering är att det kan vara för långsamt för realtid. En prototyp där existerande algoritmer har implementerats på GPU istället för CPU tas fram, detta för att testa hypotesen att färgkorrigering tar mindre prestanda ifall den implementeras på GPU. Prestandaresultaten visar att det är lämpligt att göra färgkorrigering med hjälp av shaderprogrammering på GPU istället för CPU. Ett fortsatt arbete utifrån denna studie vore att analysera hur bra olika färgkorrigeringsalgoritmer är för färgblinda personer.

Nyckelord: Färgblindhet, Färgkorrigering, Realtid, Optimering, Pixel Shader

Innehållsförteckning

1 Introduktion 1

2 Bakgrund 2

2.1 Röd-grön färgblindhet 2
2.2 Tidigare forskning 4
2.3 Olika färgrymder 5
2.4 Adaptiv färgkorrigering 5
2.5 Statisk färgkorrigering 5
2.6 Färgkorrigering i realtid 6
2.7 Färgblindhet och spel 6
2.8 Färgkorrigering för spel i realtid 9

3 Problemformulering 10

3.1 Metodbeskrivning 10

4 Genomförande 13

4.1 Implementation 13
4.2 Algoritmer 13
4.2.1 HSV rotering 14
4.2.2 Viktad HSV rotering 15
4.2.3 Daltonisering 16
4.3 Uppdateringsmätning 17

5 Utvärdering 18

5.1 Presentation av undersökning 18
5.2 Analys 18
5.3 Slutsatser 21

6 Avslutande diskussion 23

6.1 Sammanfattning 23
6.2 Diskussion 23
6.2.1 Forskningsetik 23
6.2.2 Samhällelig Nytta 25
6.2.3 Brister med färgblindkorrigering 25
6.3 Framtida arbete 26

Referenser 27

1 Introduktion

Färgblindhet är samlingsnamnet för personer som har svårare för att urskilja färger, beroende på anledningen till en färgblindhet är olika färger svårare att urskilja från andra. De vanligaste formerna av färgblindhet grupperas ofta in i röd-grön färgblindhet. För att hjälpa personer att kunna se skillnad på färger kan färger, som är svåra att särskilja, bytas ut så att enbart färger som är lättare att särskilja används.

Det finns studier vars fokus är att förbättra färgblindas möjlighet att se skillnad på färger, detta genom att procedurellt beräkna nya färger som är lättare att särskilja. Beroende på hur krävande en sådan korrigeringsalgoritm är kan de användas i realtid eller inte.

I spel är det vanligt förekommande att använda färg för att identifiera information. Detta kan vara ett problem för personer mer färgblindhet. Färgblindhet är något som blivit mer uppmärksammat idag i spel och vissa spel har inbyggda hjälpmedel som ska hjälpa färgblinda att se skillnad på färgerna. De vanligaste sätten är att antingen byta ut ett element i spelet mot ett element med en annan färg eller att applicera en färgkorrigeringsalgoritm.

Denna studie beskriver möjligheten för att använda vissa av de existerande algoritmerna i realtid och spel. Studien undersöker om en annorlunda implementation, på GPU istället för CPU, kan snabba upp existerande algoritmer så pass mycket att de blir användbara i spel.

Hypotesen säger att existerande algoritmer, som tas upp i studien, kan implementeras på GPU för att kunna användas i realtid. Algoritmernas grunder ska inte ändras utan algoritmerna ska implementeras så att de kan utnyttja en GPU's snabba förmåga att bearbeta bilder.

GPU implementationerna undersöks genom att mäta den tidsförändring som sker när ett program kontinuerligt ritar ut bilder från en kamera med och utan färgkorrigeringsalgoritmer. Det som är intressant att undersöka är huruvida den större delen av beräkningskraften kommer användas av kameran eller färgkorrgeringsalgoritmerna. Enligt Hypotesen i studien så kommer skillnaden i tidsåtgång vara försumbar för programmets användning.

Ett program har gjorts i Unity3D som kontinuerligt spelar in bilder från en kamera. På dessa bilder appliceras sedan olika färgkorrigeringsalgoritmer som är implementerade på GPU. Hypotesen, att GPU är mer lämpat för färgkorrigering, utvärderas baserat på hur lång tid färgkorrigeringen tar att utföra jämfört med att inte utföra några färgkorrigeringar.

2 Bakgrund

I såväl professionella och vardagliga situationer används färgkodningar för att återge information. Färgerna är utvalda för att vara lätta att särskilja men det finns personer som har nedsatt färgsyn, även kallad färgblindhet. För färgblinda personer kan informationen som återges vara svår att tolka då skillnader i färg inte är lika tydlig som för icke färgblinda. I vissa yrken, exempelvis som polis, pilot eller inom vissa delar av militären, är den färgkodade informationen så pass viktig att personer med färgblindhet utesluts ifrån dessa yrken (Polisen u.å.; Försvarsmakten u.å.; Svensk Pilotutbildning 2015). Även i vardagliga situationer är färgkodning så pass viktig att misstolkning eller ignorering av en färgkodning kan utgöra en fara för sig själv och andra, exempelvis vanliga trafikljus där färgen talar om ifall det är säkert att köra ut/gå över vägen.

En del av befolkningen uppskattas ha någon form av färgblindhet. Beroende på genetiska variationer i världen har olika geografiska områden olika statistik på hur många som är färgblinda. Enligt Machado (2010) är, beroende på etnicitet, 3-8% av alla män och 0.4-0.6% av alla kvinnor har någon form av avvikande färgsyn.

2.1 Röd-grön färgblindhet

Färger fångas upp i ögat med hjälp av tre koner som reagerar på olika våglängd som då motsvarar färgen med samma våglängd. Trikromasi är namnet för att ha normal färgsyn med tre koner som fångar upp färg (Figur 1). De olika typerna av färgblindhet kan delas in i tre grupper beroende på anledning till färgblindheten. Anomal trikromasi är benämningen där en av de tre konerna är defekt. Dikromasi är namnet för att ha två koner och monokromasi är namnet för att enbart ha en kon.

Figur 1 Originalbild med normalsyn

De vanligaste färgblindheterna hos kaukasiska män är enligt Jefferson och Harvey (2006) följande:

1. Deuteranomali: (Figur 2) 5% avvikelse hos grön färgkon

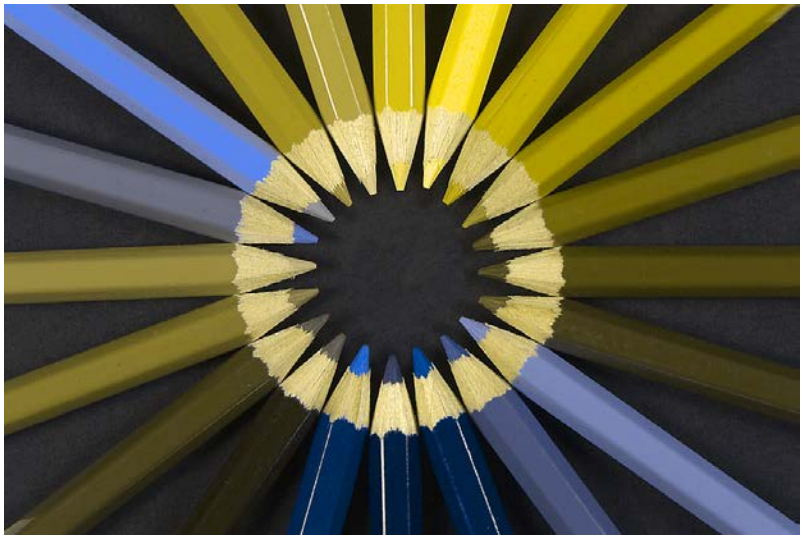
2. Protanomali (Figur 3) 1% avvikelse hos röd färgkon
3. Deuteranopi (Figur 4) 1% avsaknad av grön färgkon
4. Protanopi (Figur 5) 1% avsaknad av röd färgkon

Machado (2010) kan inte styrka att distributionen är korrekt dock är båda studierna överens om ordningen på de vanligaste. Hos kvinnor är risken för färgblindhet förhållande till män låg (totalt <1%), distributionen för vanligaste färgblindhet är samma.

Figur 2 Simulering av deuteranomali

Figur 3 Simulering av protanomali

Figur 4 Simulering av deuteranopi



Figur 5 Simulering av protanopi

Alla dessa fyra olika typer av färgblindheter har enligt Flück (2006) en generisk benämning som är röd-grön färgblindhet. Personer med röd-grön färgblindhet har svårt att se kontraster mellan röda och gröna färger. Det är den röd-gröna färgkodningen som är en av de mest vanligt använda kombinationen i olika situationer, det är allmänt känt att rött används för stop, avbryt eller något negativt gentemot det gröna används som kör, redo eller något positivt. Vilket för de med röd-grön färgblindhet kan ha svårt att särskilja. Trafikljus har alltid oavsett land röd färg som stop och grön färg som kör och i industrin använder maskiner med nödstopknapp generellt röd färg.

2.2 Tidigare forskning

Tidigare forskning har fokuserat på att göra färgkorrigeringar som ökar kontraster mellan röda och gröna färgtoner sådant att färgblinda lättare ska se skillnad på röda och gröna färger. Olika metoder studeras och används för att realisera färgkorrigeringen. I huvudsak är det tre fokusområden som uppkommer. Tanaka, Suetake, och Uchino (2010) har fokuserat på att analysera kontraster i en bild genom att beräkna kluster av färger för att ge en maximal

kontrastökning mellan de existerande färgerna i en bild. Poret, Dony och Gregori (2009) har istället fokuserat på att göra ett filter med målet att personer med normalt färgseende och färgblinda personer ska se samma bild. Det sista fokusområdet är att omvandla enskilda färger till andra färger som är lättare att uppfatta.

2.3 Olika färgrymder

Hur färgerna byts ut för korrigering beräknas och realiseras genom att använda andra färgrymder än standard RGB, där enligt Ribeiro och Gomes (2013) HLS och HSV är mer intuitiva och därmed mer praktiskt användbar för att ändra färger. En konkret anledning till varför det är mer praktiskt att använda HLS eller HSV, än den standard RGB som bilddata lagras i, är att enskilda axlar i dessa färgrymder kan roteras utan att bilden i sig ändras utan enbart färgtonerna. Machado (2010) använde den perceptuellt uniforma färgrymden CIELAB då det observerats att när dikromater upplever förluster i färgkontraster kan den kontrasten återställas med hjälp av projektion i en perceptuellt uniform färgrymd.

2.4 Adaptiv färgkorrigering

Det finns en rad olika metoder för att realisera en kontrastökning för färgblinda. Metoder analyserar ofta vilka pixlar som finns med i bilden och jämför dem med varandra, de anpassar sina resultat beroende på vilka färger som finns i bilden. Dessa metoder ger en kontrastökning även om alla färger i bilden är väldigt lika på bekostnad att de är väldigt beräkningskrävande. En metod för att sänka beräkningskostnaden skapas kluster av färger där klustren jämförs mellan varandra istället för att jämföra alla pixlar mellan varandra (Jefferson & Harvey 2006).

En annan användning av kluster görs i studien av Kim, H.H., Jeong, Yoon, Kim och Ko (2012). I studien konverterar de först alla pixlar till en simulerad version av protanopi för att analysera bilden i relation till den icke simulerade, algoritmen itereras tills det inte längre är lätt att blanda ihop färgerna, kontrollen för detta görs klustervis istället för varje pixel. Andra metoder som inte analyserar pixlar mellan varandra gör ofta en helbildsanalys, såsom Liu, Wang, Yang, Wu, och Hua (2009) som först gör en lokal färgrotation baserat på en pixels färg, sedan räknar ut en global rotation av färgerna beroende på medianen av alla pixlars data i bilden.

Ett problem som uppstår när algoritmerna är adaptiva är om bilden ändrar på sig, såsom i film eller spel. En ändring i ett område i en bild kan göra så att ett annat oförändrat område i bilden kan ge ett resultat där det oförändrade området ger subtil men märkbar ändring efter kontrastökning. Machado (2010) sparar information att jämföra med till nästa bild för att motverka detta problem.

2.5 Statisk färgkorrigering

Det finns metoder som enbart utgår utifrån vad för färg en enskild pixel har och korrigerar denna färg på ett förutbestämt sätt. Det vill säga samma färg, oavsett hur resten av en bild ser ut, resulterar alltid till samma korrigerade färg. Dessa metoder har inte problemet att oförändrade färger får en ny färg vid färgkorrigering.

Den simplaste metoden är att göra en färgrotation baserat på en pixels färgton, en pixel med en viss färg roteras alltid samma antal grader i sin färgton. Fokus blir på beräkningen av rotationen. I studien av Ohkubo och Kobayashi (2008) roteras alla färger bort från röd, färgtonerna mappas om linjärt så att de istället för 0° till 360° går från 45° till 315°. I studien

av Ribeiro och Gomes (2013) görs en liknande metod men som bara roterar färgerna om färgtonen anses vara lätta att förväxla och roterar då bort från den färgtonen.

Tanuwidjaja et. al. (2014) beskriver en metod som istället för att basera alla beräkningar på vinklar och rotationer av färgtoner använder sig av flera omvandlingsmatriser för att få en simulerad färgblindhet. Varje enskild pixel räknas om till en simulering av färgblindhet, sedan jämförs den med originalfärgen för att ge ett uppskattat fel, det uppskattade felet konverteras till en förflyttning som läggs på till originalfärgen.

2.6 Färgkorrigering i realtid

Metoder som är lämpade för realtidsanvändning är enligt Machado (2010) metoder som har en linjär tidsökning baserat på totala antalet pixlar och där större delen av beräkningarna kan hanteras pixel per pixel, beräkningar som är pixel per pixel kan ge en effektiv implementation på GPU.

Jack (2011) skriver om olika standarder för videosignaler, den långsammaste nämnda standarden är minst 24 bildrutor per sekund för att strömma videos i realtid. Machado (2010) framför en metod som kan implementeras på GPU som då går att användas i realtid, med bilder på 800x800 pixlar kan det beräknas som max 36 bildrutor per sekund.

Ribeiro och Gomes (2013) nämner att deras algoritm inte når upp till realtid men video var inte deras focus vid testning. Men deras metod är väldigt lik metoden presenterad i studien av Ohkubo och Kobayashi (2008). De båda byter till en färgrymd där en axel, som innehåller data för färgton, roteras. Den senare visar upp sina färgkorrigeringar i realtid men färgkorrigeringen beräknas också på en speciell bildprocessande enhet. Det är otydligt vad den faktiska prestandan är då varken tid eller bildupplösning står med. Det kan antas att de använder en definition av realtid som är liknande den långsammaste standarden för videosignaler som tas upp i Jack (2011). Detsamma gäller för studien av Tanuwidjaja et. al. (2014) som också uppnår realtid men är otydliga på den faktiska prestandan.

2.7 Färgblindhet och spel

Spel har vanligt förekommande moment där färg används för att identifiera information. Färgerna är utvalda för att vara estetiskt snygga och åtskilda från varandra, vilket för färgblinda ofta är helt andra färger än för icke färgblinda. I de flesta spel finns inga hjälpmedel för de som är färgblinda att se skillnad på färgerna. Färgblindhet är mer uppmärksammat idag än vad det tidigare varit och de spel som implementerar system för att hjälpa färgblinda gör på lite olika sätt.

Ett vanligt sätt att göra ett färgblindhetsläge är att byta ut hela element till ett annat element med en annan färg. Exempel på spel som gör på detta sätt är Dota 2 (Valve 2013) (Figur 6 och Figur 7) och Heroes of the Storm (Blizzard Entertainment 2016).

Figur 6 Dota 2 (valve, 2013). Utan färgblint läge

Figur 7 Dota 2 (valve, 2013). Färgblint läge

En annan metod är att använda ett statiskt postprocessfilter som korrigerar färgerna i realtid på hela den utritade bilden. Exempel på spel med färgblint läge, som ändrar hela bilden med ett filter, är Overwatch (Blizzard Entertainment 2016) (Figur 8, Figur 9, Figur 10 och Figur 11) och Doom (id Software 2016).

Figur 8 Overwatch (Blizzard Entertainment, 2016). Utan färgblint läge

Figur 9 Overwatch (Blizzard Entertainment, 2016). Färgblint läge för deuteranopi

Figur 10 Overwatch (Blizzard Entertainment, 2016). Färgblint läge för protanopi

Figur 11 Overwatch (Blizzard Entertainment, 2016). Färgblint läge för tritanopi (som deuteranopi och protanopi men avsaknad av blå färgkon istället)

2.8 Färgkorrigering för spel i realtid

De olika metoderna för att öka kontrast har olika förutsättningar för att kunna användas i spel. Spel brukar generellt sikta på att kunna nå 60 bildrutor per sekund på dator och 30 bildrutor per sekund på konsol. Att då använda exempelvis metoden av Machado (2010) även om den uppnådde realtid i en situation där den kunde använda all prestanda skulle den förmodligen inte gå att använda i ett spel där resurserna måste användas till att uppdatera spelet. Speciellt inte om man vill försöka uppnå ett av de vanligare höga kraven på bildrutor per sekund. För att förtydliga, även om en algoritm är tillräckligt snabb att den individuellt kan bearbeta 60 bildrutor per sekund kommer resten av spelet högst troligt inte hinna bearbetas utan att den totala bearbetningstiden uppnår mycket mindre än 60 bildrutor per sekund.

Nästan alla spel renderar ut nya bilder där det inte är förutsägbart hur de nya bilderna ser ut. Bilderna kan inte analyseras innan de existerar och spelen i sig kräver större delen av den tillgängliga beräkningskraften för att uppdateras korrekt så färgkorrigeringen måste ske inom en väldigt strikt tidsram. Förutom prestandakrav är ett konsekvent resultat eftertraktat. Att använda adaptiva algoritmer, som ämnar att maximera kontrast varje rendering, kan ge ett resultat som inte är konsekvent. Rörelser i bilden göra att kontrastberäkningarna blir annorlunda på färger som förväntas se likadana ut. De statiska metoderna har inte detta problem och de kräver vanligtvis mindre prestanda än de adaptiva.

3 Problemformulering

Att bearbeta bilder, med till exempel filter, för att förbättra förmågan för färgblinda att se skillnad på färger är ett beräkningsintensivt arbete för en dator. Om det är en stor ström av bilder som ska bearbetas tillräckligt snabbt, för att uppfattas som realtid, krävs algoritmer som inte tar för lång tid per bild.

För att använda en färgkorrigeringsalgoritm i spel bör algoritmen som används göra en knapp märkbar skillnad i prestanda då spelet i sig kräver den största delen av tillgänglig beräkningskraft.

De algoritmer som är anpassade till att vara så snabba som möjligt arbetar pixel per pixel för att få en linjär tidsökning för större bilder. Men i vissa fall räcker inte detta för att åstadkomma realtid såsom i studien av Ribeiro och Gomes (2013), algoritmen i studien testades på CPU. Algoritmen i Riberio och Gomes (2013) är i grunden väldigt lik den som är nämnd i Ohkubo och Kobayashi (2008) som uppnådde realtid på en specialenhet för att processa bilder, men dock saknas detaljer kring hur många pixlar varje bild hade och hur lång tid det tog.

Liknande algoritmer har olika möjligheter för att användas i realtid beroende på vad för processenhet som används. Enligt Machado (2010) bör algoritmerna implementeras på GPU för att uppnå tillräcklig kort beräkningstid för att användas i realtid. Således skulle algoritmerna kunna snabbas upp via en implementation på GPU, utan att ändra algoritmerna. Algoritmen som presenterades i Machado (2010) skulle ta upp en stor del av beräkningskraften i ett spel och är således inte lämplig, men detta har till stor del att göra med att algoritmen är adaptiv och anpassar sig till hela bilden som helhet snarare än pixel per pixel. De mer lämpliga algoritmerna för att användas i spel både för prestanda och för att vara konsekvent i utseende är de som gör statiska färgkorrigeringar.

Frågeställningen är om algoritmer avsedda som hjälpmedel för färgblinda kan implementeras så att de fungerar praktiskt i realtid för spel. Hypotesen är att existerande algoritmer, som har en statisk färgkorrigering och linjär tidsökning för ökande bildstorlek, kan implementeras på GPU för att kunna användas i realtid. Ett utnyttjande av GPU'ns förmåga att snabbt utföra beräkningar på bilder, pixel per pixel, antas sänka beräkningstiden utan att behöva ändra algoritmernas grunder.

3.1 Metodbeskrivning

Det system som algoritmerna körs på är en mobiltelefon av typen Samsung Galaxy S4. För dagens standard av CPU och GPU har denna modell både en CPU och GPU som är gamla och långsamma. Med anledning att om en S4 klarar av att köra algoritmerna så kan det antas att alla modernare system oavsett plattform också kommer klara av att köra algoritmerna, men också för att i viss mån få en liknande prestanda som i äldre studier. Algoritmerna testas också på en bärbar dator, Lenovo Thinkpad e520, med processorn av typen Intel Core i5 2410M och grafikkort av typen AMD Radeon HD 6630M.

Implementationen sker på GPU med hjälp av shaders. Algoritmerna som testas att implementera på GPU är hämtade från studierna av Riberio och Gomes (2013), Ohkubo och Kobayashi (2008) och Tanuwidjaja et. al. (2014). Algoritmerna har samma färgblindhet i fokus; Deuteranopi.

Intresseområdet för studien är ifall algoritmerna i praktiken kan köras i realtid med hjälp av GPU utan att få någon märkbar skillnad. I praktiken förtydligas det som att en applikation ska kunna använda sig av algoritmerna utan att förlora större delen av sin beräkningskraft som applikationen eventuellt behöver använda till andra saker. Validering kommer därför ske i en icke isolerad beräkningsmiljö, en fältstudie genomförs med en egen utvecklad applikation där en inbyggd kamera konstant ritas ut på skärmen, de olika algoritmerna testas genom att appliceras på utritningen.

Validering om algoritmerna uppnår kraven för realtid görs genom att jämföra antalet bilder per sekund före och efter applicering av algoritmerna. Om hypotesen fungerar ska det inte finnas någon märkbar skillnad. De hårda kraven är för att ett spel vill kunna använda all beräkningskraft för att uppdateras.

Att mäta bildrutor per sekund, speciellt då med en mobiltelefon som strömmar bilder från sin kamera kontinuerligt, kan ge olika resultat beroende på vad som sker. Det kan exempelvis tänkas att beroende på vad som filmas kan processandet av den bildrutan ta längre eller kortare tid. För att få ett bra resultat är det fördelaktigt att använda ett medelvärde för tiden det tar per bildruta. Mobiltelefonen kan dessutom ha bakgrundsprocesser igång som kan ta mer än normalt av beräkningskraften både kontinuerligt och periodiskt.

Argument kan ges för att istället testa algoritmerna i en isolerad beräkningsmiljö såsom det görs i studien av Machado (2010) vilket minimerar eventuella beräkningar utanför algoritmen. Att inte testa i en isolerad miljö gör att resultaten kan vara missvisande, om de jämförs med andra algoritmer skulle de också behöva göras i samma miljö för att inte vara relativt sämre i jämförelsen än vad de eventuellt är. Att använda en isolerad beräkningsmiljö är en tydligare baslinje som gör att resultat kan jämföras med varandra så länge hänsyn tas till eventuell hårdvaruskillnad. En nackdel blir dock att det är svårt att översätta resultaten till ett praktiskt sammanhang. Exempelvis frågan ”Kan jag använda algoritmen i realtid i min produkt?” vilket kan vara nej, fastän i studien valideringen kom fram till att algoritmen funkar i realtid, på grund av att det i praktiken är det mer än algoritmen som behöver använda sig av beräkningskraften.

Om målet med studien vore att ta fram en algoritm som funkar i realtid hade en användarstudie kunnat utföras för att undersöka hur bra färgblinda personer tycker att algoritmen fungerar. Detta hade även kunnat göras genom att jämföra redan existerande algoritmer. Då målet snarare är att undersöka om existerande algoritmer, som inte riktigt uppnår realtid, kan snabbas upp genom att implementera dem på GPU är en användarstudie inte nödvändig. Det är enkelt att sätta upp krav och mäta tiden för realtid genom experiment eller fallstudie. Däremot finns det intressanta aspekter som hade kunnat undersökas i en användarstudie, exempelvis i studien av Poret, Dony och Gregori (2009) görs en användarstudie som undersöker vilka färger olika personer, både färgblinda och icke färgblinda, uppfattar att en viss bild innehar. I studien undersöks både originalbilden och efter ett filter har använts på bilden. Deras resultat var att efter filtret är applicerat på bilden, uppfattade både färgblinda och icke färgblinda samma färg. Notera att de ser fortfarande inte samma färg, utan det är det associerade namnet till färgen de ser som uppfattas samma.

Det är intressant att testa olika algoritmer för att undersöka hur algoritmerna faktiskt ändrar uppfattandet av bilderna, exempelvis hur bildens estetik påverkas eller vilka namn som färgblinda associerar med de nya färgerna de ser i en bild. Andra studier har nästan uteslutande ignorerat sådana aspekter och enbart fokuserat på kontrastökningen. Studierna

fokuserar oftast på hur kontrasten mellan färgerna, i en bild, kan ökas så mycket som möjligt utan att det tar för lång tid att göra. En aspekt som kan vara relevant att undersöka är hur onaturlig en bild uppfattas efter att olika färgblindhetsfilter appliceras. Två problem med detta är att det är svårt att definiera något onaturligt och att olika personer kan uppfatta olika filter annorlunda.

4 Genomförande

Detta kapitel beskriver detaljerna kring hur problemställningen undersöktes, algoritmerna testades och även mer i detalj hur varje enskild algoritm implementerades.

4.1 Implementation

Problemställningen undersöktes genom att ett program skapades som kontinuerligt spelade in bilder från en kamera. Programmet implementerades i spelmotorn Unity3D med programmeringsspråket C#. Inför varje uppdatering av programmet hämtades den nuvarande bilden från kameran, varefter den hämtade bilden applicerades som huvudtexturen i ett material, ett material definierar hur ett objekt visas (Unity Technologies 2017a). Om bilden var samma som förra uppdateringen ersattes inte huvudtexturen i materialet. Men en ny utritning sker ändå till skärmen, detta för att programmets uppdateringsfrekvens inte ska vara styrt av kamerans uppdateringsfrekvens.

Algoritmerna skrevs i shaderspråket HLSL för att implementeras på GPU. Varje material använder sedan en shader som grund för att rita ut texturen. En shader gör beräkningar på ett optimerat sätt på varje vertex och på varje pixel. Varje algoritm som implementerats har sin egna shaderkod och tillhörande material.

Att direkt skriva ut en bild med materialet som innehåller färgkorrigeringsalgoritmen ger ett mer tidseffektivt resultat, men i ett spel kan det tänkas att materialet måste använda en annan shader för att uppnå en viss effekt. Då måste istället färgkorrigeringen göras efter att objektet redan renderats på önskvärt sätt. Speciellt om det finns många olika objekt som ska ritas ut på olika sätt, den slutgiltiga färgkorrigeringen bör då inte göras förrän alla objekt har ritats ut.

Även om vissa spel i praktiken inte kan använda färgkorrigeringen i första utritningen kan det jämföras med sättet programmet använder kamerans bild. Ett spel beräknar en textur med virtuell kamera och sedan sker korrigering, programmet beräknar en textur med en faktisk kamera och sedan sker korrigering. I slutändan är det i denna undersökning mest intressant att analysera färgkorrigeringen av en textur, oavsett hur den texturen togs fram.

4.2 Algoritmer

Planering kring detaljer för implementationen ändrades och funderades kring mycket parallellt med problemformuleringen. Innan indelningen i adaptiva och statiska algoritmer var klar planerades det att implementera några algoritmer som var adaptiva, det finns två stora problem med de algoritmerna som i slutändan gjorde att enbart statiska algoritmer togs med i problemformuleringen för denna studie. Ett av de två problemen är att färger i bilden kan ge olika färgkorrigeringsresultat även om det är samma färg på grund av att andra färger i bilden har ändrats. Algoritmerna som inte togs med söker efter en maximal kontrastökning för en bild snarare än en konsekvent färgkorrigering. Det andra problemet är relaterat till det sättet att arbeta då den måste analysera hela bilden samtidigt och inte bara per pixel. Att analysera hela bilden samtidigt är något som inte går att göra i en pixel shader och den delen måste då implementeras på CPU och sedan skickas över till shadern.

De algoritmerna som användes för att svara på frågeställningen är inte beroende av beräkningar från andra delar än GPU och kunde då implementeras i fullo i en pixel shader. Nedan är de använda algoritmerna beskrivna mer i detalj.

4.2.1 HSV rotering

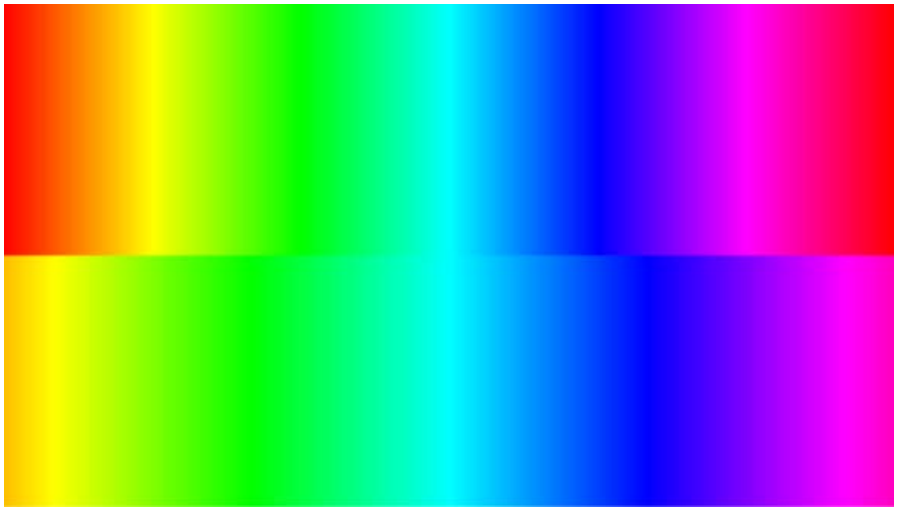
Algoritmen i studien av Ohkubo och Kobayashi (2008) är i grunden väldigt simpel, det görs ett antagande att om en pixels färgton linjärt roteras bort från rött kommer det bli lättare att urskilja färger för färgblinda. Se Figur 12 och Figur 13.

I studien av Ohkubo och Kobayashi (2008) beskrivs att de använde färgrymden HLS och roterade färgtonen. Istället för HLS användes i denna implementationen färgrymden HSV för att rotera färgtonen. Det är ingen skillnad i hur HLS och HSV hanterar data för färgton så att använda HSV ger ett ekvivalent resultat som att använda HLS.

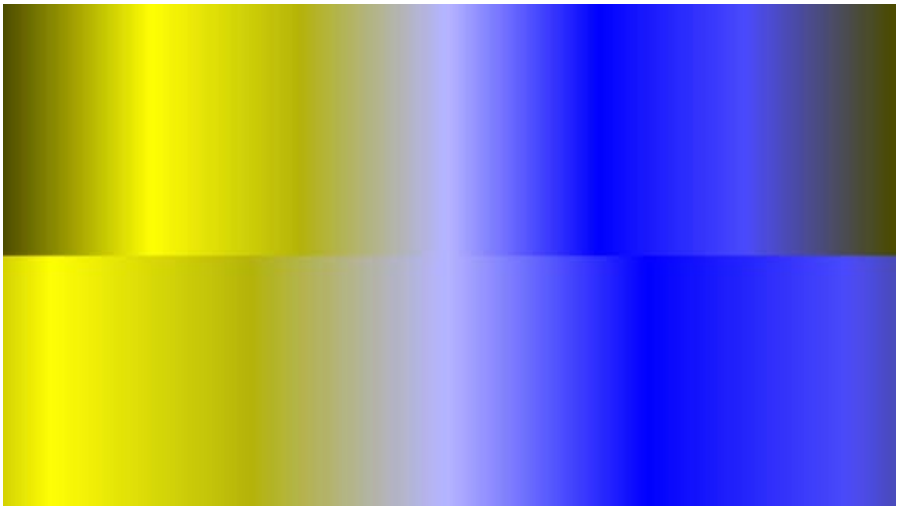
En pixels RGB data görs om till färgrymden HSV, färgtonen H roteras till H' enligt funktionen

$$H' = H/360°*270° + 45°$$

varefter HSV datan görs om till RGB igen då innehållande den korrigerade färgen.



Figur 12 HSV rotering, vanlig färg ovan korrigerad färg under.



Figur 13 HSV rotering med simulering av Deuteranopi, vanlig färg ovan korrigerad färg under.

4.2.2 Viktad HSV rotering

Algoritmen i studien av Ribeiro och Gomes (2013) är likt en vanlig rotering av färgtonen i HSV, men istället för att göra en linjär rotering rakt av är roteringen viktad beroende på vilken färgton H en färg har. I ett visst intervall av färgtonen ändras inte färgen alls. Se Figur 14 och Figur 15.

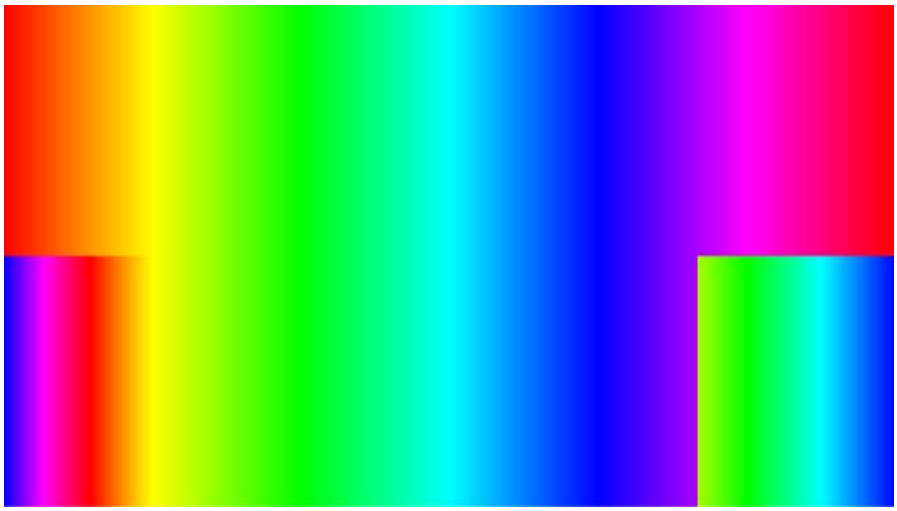
För att optimera koden till grafikkort har alla förgreningar eliminerats. Istället för att förgrena ifall färgtonen är utanför beräkningsintervallet beräknas alltid deltavärdet ΔH, men en kontrolvariabel multipliceras med deltavärdet för att tala om ifall det inte ska användas så att detta blir noll. Om vinkeln för färgtonen är mellan 60°-280° görs ingen rotering. I de andra vinklarna ändras färgtonen H till H' enligt:

$$H' = H + \Delta H$$

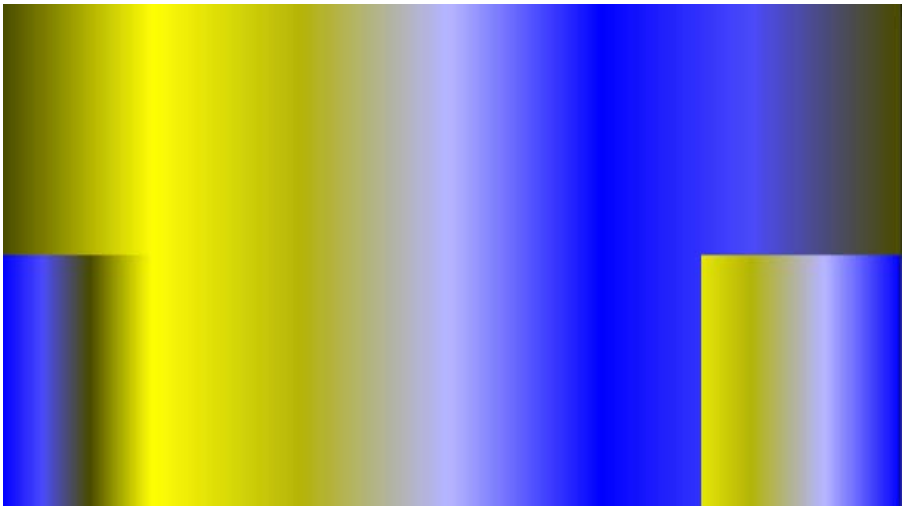
Vikten för hur förvirrande en färg är för färgblinda bestäms genom att jämföra hur mycket magenta som finns i färgen. Ett deltavärde beräknas genom följande funktion:

$$\Delta H = (H-130°) * f(G2)$$

Där G2 är värdet på grönt enligt RGB efter att färgen konverterats från HSV till RGB med maximerade värden för mättnad och värde. Funktionen f är 1-g2/g2max för att få en vikt på hur mycket magenta i färgen och H-130° är själva rotationen.



Figur 14 Viktad HSV rotering, vanlig färg ovan korrigerad färg under.



Figur 15 HSV rotering med simulering av Deuteranopi, vanlig färg ovan korrigerad färg under.

4.2.3 Daltonisering

Algoritmen i studien av Tanuwidjaja et. al. (2014) gör en rad olika matrismultiplikationer för att åstadkomma färgkorrigeringen. Algoritmen beräknar en simulerad version av färgblindhet med hjälp av färgrymden LMS som representerar våglängden som ögats koner kan uppfatta. Det simulerade resultatet används för att beräkna hur mycket färgen ska skiftas. Se Figur 16 och Figur 17.

Matriserna som användes är samma som matriserna som användes i Tanuwidjaja et. al. (2014). Det är en matris för varje följande beräkning:

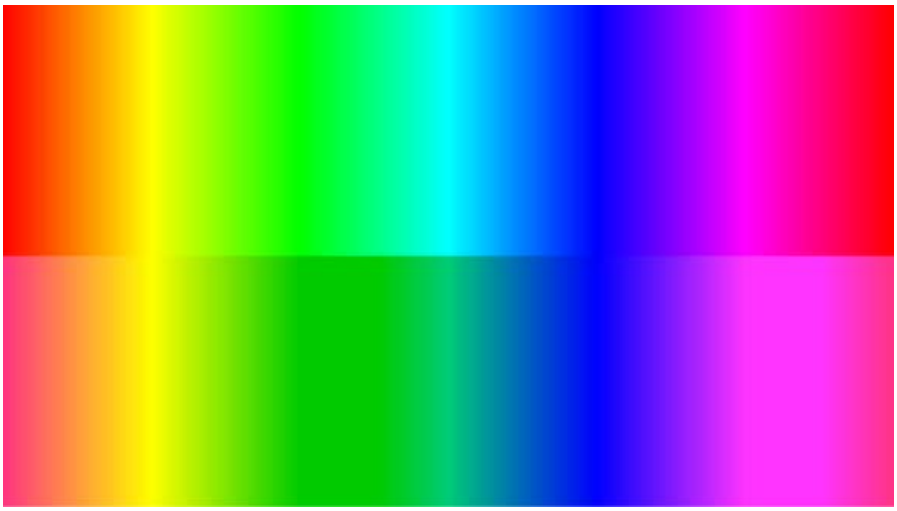
- RGB till LMS
- LMS till RGB
- LMS färgblindhetssimulering (deuteranopi)
- RGB korrigering

Alla matriser är 3x3 och multipliceras med en 3x1 där den nya 3x1 matrisen innehåller resultatet. Till exempel så multipliceras RGB till LMS med RGB för att få ut LMS som resultat. För att beräkna den simulerade färgblindheten används RGB till LMS, LMS till färgblindhetssimulering och sedan LMS till RGB för att få ut färgblindhetssimuleringen i RGB. Sedan subtraheras RGB med simulerade RGB för att få ut skillnaden. Den skillnaden används sedan med RGB korrigeringsmatrisen för att få ut det slutliga resultatet.

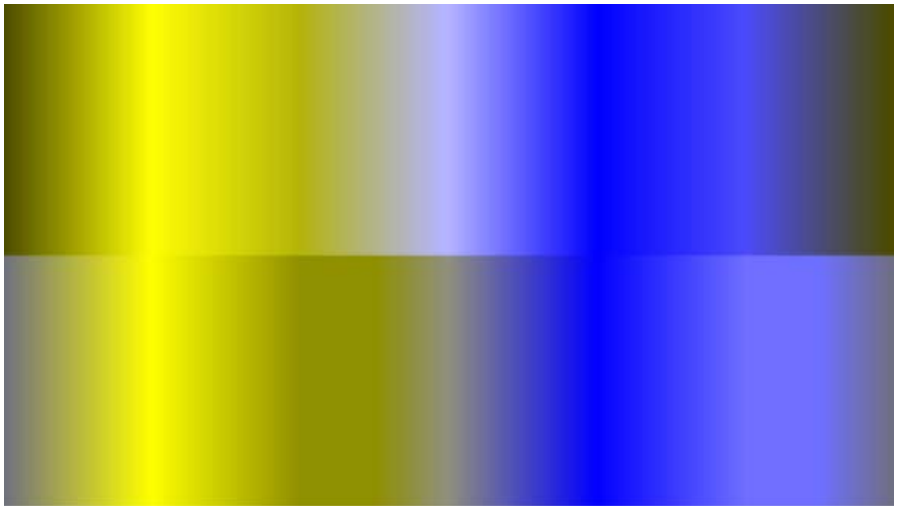
I mer lättläst format:

```
LMS = [RGBtoLMS] * RGB
LMScb = [LMStoLMScb] * LMS
RGBcb = [LMStoRGB] * LMScb

RGBe = RGB - RGBcb
RGBs = [RGBshift] * RGBe
RGBd = RGB + RGBs

cb(color blind) är simulerad färgblindhet, deuteranopia
e(error) är felskillnaden jämfört med inte färgblind
s(shift) är för att försöka kompensera felet
```

```
d(daltonize) är den färdiga daltoniseringen
[] talar om att det är en 3x3 matris, allt annat är 3x1 matriser
```



Figur 16 Daltonisering, vanlig färg ovan korrigerad färg under.



Figur 17 Daltonisering med simulering av Deuteranopi, vanlig färg ovan korrigerad färg under.

4.3 Uppdateringsmätning

För att mäta prestandan på implementationerna mäts tiden för en hel uppdateringsloop istället för att mäta prestandan för att rendera den korrigerade bilden. Detta i enlighet med argumenten i metodbeskrivningen, att få resultatet i ett mer praktiskt kopplat sammanhang.

Ett problem som visade sig vid tidig informell testning var att uppskattning av antalet bilder per sekund var oprecist. Istället för att försöka mäta under en viss tid så mättes istället ett visst exakt antal uppdateringar, den lägsta tiden sparades och medelvärde för tiden beräknas. Detta för att lättare kunna utvärdera hur mycket beräkningskraft som används till korrigeringsalgoritmen gentemot hur mycket beräkningskraft som programmet i sig använder.

5 Utvärdering

I detta kapitel presenteras en utvärdering av algoritmerna med fokus på deras tidseffektivitet. Först presenteras en presentation av undersökningen, vilken hårdvara har använts och hur data har mätts. Vidare analyseras vad mätdata visar och slutligen vad för slutsatser som kan dras med hjälp av dessa.

5.1 Presentation av undersökning

Undersökningen genomfördes på en mobil, Samsung Galaxy S4, med skärmupplösningen 1920x1080 och en bärbar dator, Lenovo Thinkpad e520, med skärmupplösningen 1366x768, processor av typen Intel Core i5 2410M och grafikkort av typen AMD Radeon HD 6630M. Alla system testades med en aktiv kamera.

Värdena från testningen togs fram efter en mätning av 1000 uppdateringar i följd. Den lägsta tiden sparades, samt den totala tiden för de uppdateringarna sparades för att beräkna ett medelvärde för tiden. Varje algoritm testades individuellt och under varje uppdatering appliceras algoritmen igen på en icke färgkorrigerad bild, där den bilden ändrades i takt med att kameran uppdaterats med en ny bild.

5.2 Analys

Alla algoritmer fick en snabb lägsta tid för och hade mellan 5 och 10 gånger så högt medeltidsvärde. När programmet kördes utan algoritm blev det samma resultat. Se Tabell 1, Tabell 2, Tabell 3 och Tabell 4 för detaljerade resultat. Det som är intressant här är hur stor skillnad det är på medelvärdet och den lägsta tiden. Enligt Unity Technologies (2017b) sker alltid rendering varje uppdatering vilket resulterar till att den lägsta tiden motsvarar en uppdatering där inte mycket i programmet hände förutom själva färgkorrigeringsalgoritmen, Tiden för medelvärdet motsvarar den tiden som programmet som helhet kräver förutom färgkorrigeringsalgoritmen, såsom uppdatering av kameran. Den här skillnaden märks inte av i programmet, allting ser ut att flyta på i samma hastighet, se Figur 18, Figur 19, Figur 20 och Figur 21 för ett utdrag av 100 uppdateringar på följd under mätning av den bärbara datorns prestanda.

Programmet fick ett snarlikt resultat för alla algoritmer och resultatet var även liknande utan att använda någon färgkorrigeringsalgoritm. Det enda resultatet där det är en större skillnad är på mobiltelefon där både HSV rotering och viktad HSV rotering var långsammare än daltonisering och ingen algoritm, både HSV rotering och viktad HSV rotering var ungefär lika snabba som varandra och daltonisering och utan algoritm var lika snabbt.

Den viktade HSV roteringen är den algoritmen som har konkret tidsdata att jämföra med. Ribeiro och Gomes (2013) presenterade ett resultat där algoritmen korrigerade en 580x434 stor bild på 0.508 sekunder med hjälp av en Intel Quad Core Q9550 CPU. Denna studies resultat på samma algoritm, fast med en GPU implementation istället för CPU, är ca 250 gånger snabbare på den testade mobilen med en bildstorlek på 1920x1080 och 1000 gånger snabbare på den testade bärbara datorn med bildstorlek på 1366x768.

Tabell 1 Mätvärden för HSV roteringsalgoritmen

HSV Rotering	**Lägsta tid**	**Medelvärde för tid**
Mobil 1920x1080	3,61 ms	21,27 ms
Dator 1366x768	0,59 ms	4,62 ms

Tabell 2 Mätvärden för viktade HSV roteringsalgoritmen

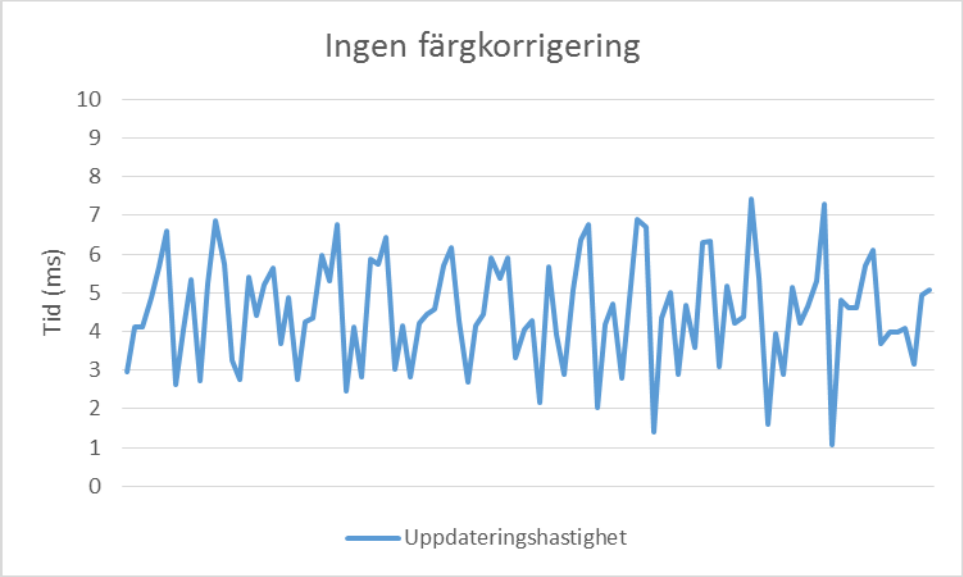
Viktad HSV Rotering	**Lägsta tid**	**Medelvärde för tid**
Mobil 1920x1080	3,75 ms	22,51 ms
Dator 1366x768	0,57 ms	4,64 ms

Tabell 3 Mätvärden för Daltoniseringsalgoritmen

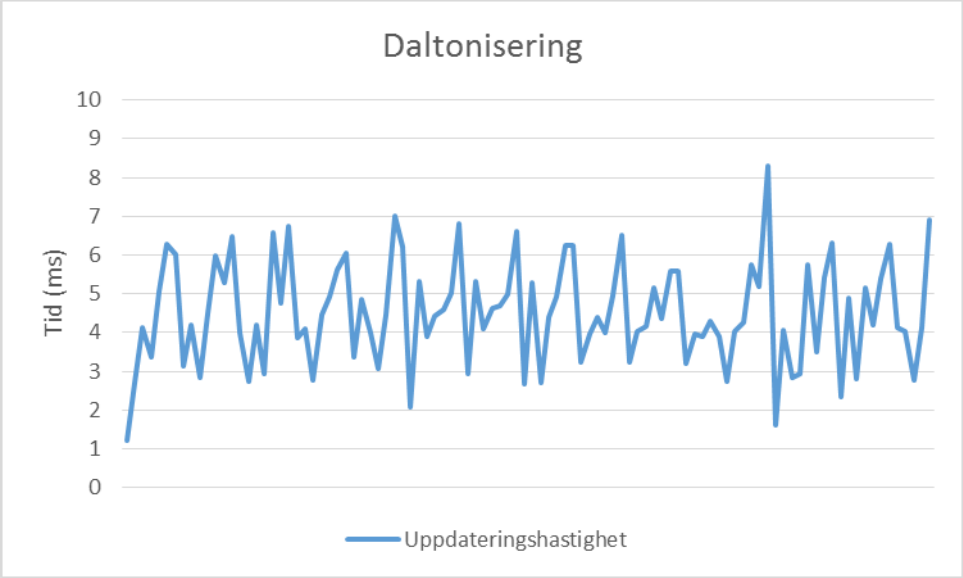
Daltonisering	**Lägsta tid**	**Medelvärde för tid**
Mobil 1920x1080	3,45 ms	16,96 ms
Dator 1366x768	0,52 ms	4,60 ms

Tabell 4 Mätvärden utan färgkorrigeringsalgoritm

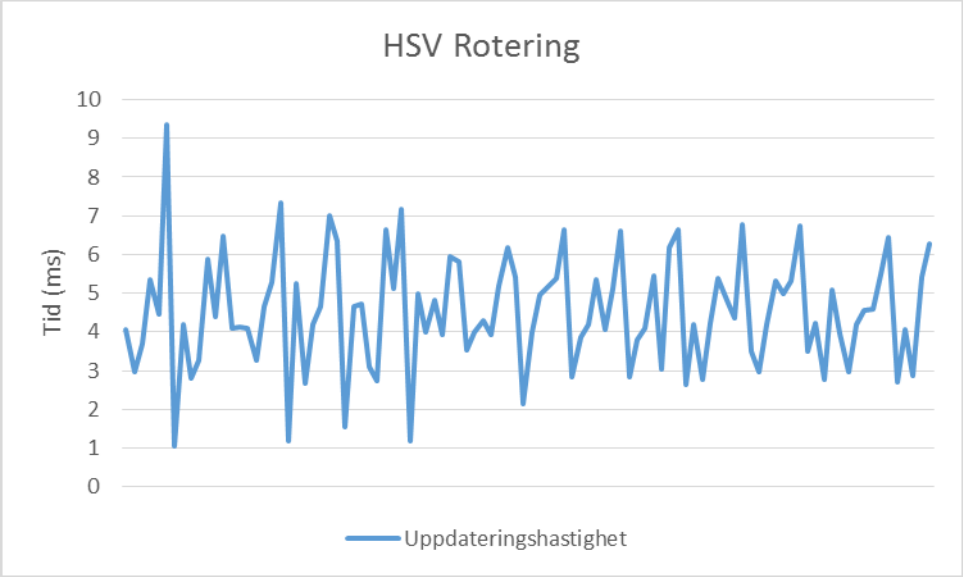
Ingen färgkorrigering	**Lägsta tid**	**Medelvärde för tid**
Mobil 1920x1080	3,42 ms	16,88 ms
Dator 1366x768	0,52 ms	4,65 ms



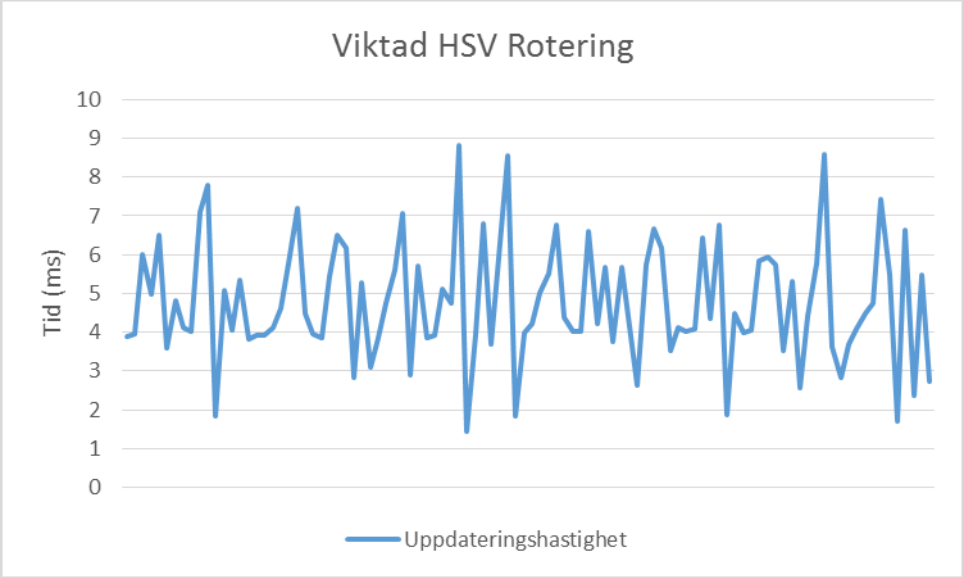
Figur 18 Uppdateringshastighet för 100 uppdateringar utan färgkorrigering.



Figur 19 Uppdateringshastighet för 100 uppdateringar med daltonisering.



Figur 20 Uppdateringshastighet för 100 uppdateringar med HSV rotering.



Figur 21 Uppdateringshastighet för 100 uppdateringar med viktad HSV rotering.

5.3 Slutsatser

Att använda en statisk färgkorrigeringsalgoritm applicerad på ett material i Unity gav inte någon märkbar skillnad gentemot att inte ha en färgkorrigeringsalgoritm applicerad, se Figur 18, Figur 19, Figur 20 och Figur 21 för visuell representation och Tabell 1, Tabell 2, Tabell 3 och Tabell 4 för insamlade mätvärden. Den tiden som redan innan spenderades per pixel vid en utritning fick ingen större påverkan av en addering av en av de färgkorrigeringar som testades. Detta tyder på att den beräkningstid det tar att skriva ut en bild är likt den som krävs för att rita ut en bild med färgkorrigering.

Till skillnad från CPU implementation som skulle appliceras utöver det arbete som krävs för rendering av en bild så utnyttjar en GPU implementation med hjälp av shaders redan det arbete som datorn utför vid en vanlig rendering. I jämförelse med studien av Ribeiro och

Gomes (2013) och denna studie så är hårdvaran som använts i denna studie nyare. Hårdvaran som använts av Ribeiro och Gomes (2013) är däremot betydligt dyrare och var på sin tid bland det bästa som fanns tillgängligt. I jämförelsen av tidsresultat är GPU implementationen så många gånger snabbare, även vid hantering av större bilder och hårdvaruskillnader, att det inte finns några tvivel om att för den algoritmen är just GPU implementationen betydligt bättre. Det är inte uteslutet att även om hårdvaran Riberio och Gnomes (2013) använder är äldre än den denna studie kan hårdvaran Riberio och Gnomes (2013) använder ha en prestandafördel.

Alla de algoritmer som testades skulle prestandamässigt fungera att användas i realtid för ett krävande spel, den lägsta tid som en uppdatering tog är låg nog för att inte ha en påverkan på spelets beräkningskraft. Även om det testade programmet hade ett högre medelvärde för uppdateringshastigheten är det den lägsta tiden som talar om hur lång tid algoritmerna tar då det är enbart rendering som sker. Framförallt var det i princip ingen skillnad med eller utan algoritmer. Den skillnaden som var, där HSV roteringarna var långsammare på mobil, visar att för system med sämre prestanda kan vilken algoritm för färgkorrigering ha en viss betydelse. Det är inte troligt att en statisk färgkorrigering skulle vara en anledning till att ett spel blir prestandakrävande.

Summan av denna studies resultat är att det är ytterst lämpligt att använda shaders för att implementera hjälpmedel för färgblinda som ska fungera i realtid.

6 Avslutande diskussion

Denna avslutande diskussion innehåller en översiktlig sammanfattning av hela examensarbetet, en bredare diskussion kring problemet och resultat utanför studiens specifikation samt en diskussion kring en hypotetisk fortsättning av arbetet.

6.1 Sammanfattning

Studiens syfte var att testa hypotesen att statiska färgkorrigeringsalgoritmer kan anpassas till GPU och då användas till spel i realtid. Många existerande färgkorrigeringsalgoritmer är för krävande för att användas i realtid, vissa av de algoritmernas grunder är beräkningar pixel per pixel. En GPU är byggd för att arbeta pixel per pixel, algoritmer som var för krävande på CPU undersöks ifall de med hjälp av GPU snabbas upp så mycket att de blir användbara i spel. Tidigare studier låg i grund för vilka algoritmer som implementerades och testades samt hur hypotesens detaljer formades.

För att testa hypotesen har en undersökning med syfte att mäta och jämföra beräkningstiden med och utan korrigeringsalgoritm utförts. Till undersökningen skapades ett program som kontinuerligt ritar ut bilder från en kamera med och utan färgkorrigeringar, där GPU'n användes för att beräkna korrigeringarna. Tidsåtgången för varje uppdatering i programmet mättes och sammanställdes. Beräkningstiden jämfördes före och efter applicering av korrigeringsalgoritmerna. Ifall tillgänglig data från de studier algoritmerna hämtades ifrån existerade jämfördes tiden också med de tidigare studierna. Programmet som skapades för att utföra testerna gjordes i Unity och GPU implementationen för algoritmerna gjordes med hjälp av shaderspråket HLSL.

Undersökningen visade att den statiska typen av färgkorrigeringsalgoritmer presterar väldigt bra på GPU då denna är konstruerad för att snabbt göra beräkningar pixel per pixel. Den beräkningstid som uppmättes för programmet var snarlik för alla algoritmer och även utan algoritm. Den största skillnaden uppmättes på mobil. Utan korrigering och daltonisering hade ungefär samma tid och roteringsalgoritmerna var aningen långsammare.

Resultatet visar att användning av shaders är lämpligt för att förbättra prestandan hos korrigeringsalgoritmer. För de korrigeringsalgoritmer som testades blev beräkningstiden så låg att i praktiken skulle de kunna användas utan att påverka beräkningskraften hos ett krävande spel.

6.2 Diskussion

6.2.1 Forskningsetik

Datan varierade mycket beroende på om man tittar på den lägsta tiden eller tidens medelvärde. Om en jämförelse av data mellan andra studier skulle haft mer fokus hade en mer isolerad miljö behövt användas. Det är den lägsta tiden som kan jämföras med andra studier då det är den som mest talar om hur stor del av beräkningskraften hos programmet som färgkorrigeringsalgoritmen använder. Detta gäller enbart om antagandet att den lägsta tiden alltid innehåller en körsning av färgkorrigeringsalgoritmen. Antagandet styrks av Unity Technologies (2017b), då enligt manual, rendering görs varje uppdatering. Dock går det inte dra några slutsatser om vad som gör att medeltiden blir så mycket högre, enbart antaganden, såsom att det verkar troligt att det är kamerans uppdatering som drar mycket kraft. Slutsatsen

att GPU implementation av statiska algoritmer är en klar prestandafördel bör dock kunna anses trovärdigt. Programmet körs i realtid och hastigheten för att enbart köra algoritmerna är väldigt mycket snabbare än motsvarande CPU implementationer. Jämfört med denna studies data sker ingen märkbar förändring av uppdateringshastighet när korrigering används jämfört när ingen korrigering används.

Även andra studier, dock adaptiva istället för statiska, håller med om att GPU implementation är lämplig och har även data som stödjer att en GPU implementation är snabbare (Machado 2010; Kuhn, Oilviera & Fernandes 2008). I studien av Kuhn, Oilviera och Fernandes (2008) användes en algoritm som i grunden gör en stor mängd iterationer för att göra korrigeringen. Prestandan har jämförts mellan CPU, GPU samt en alternativ GPU implementation som gör sin slutgiltiga färgberäkning lite annorlunda i pixel shader, se Tabell 5 för data.

Tabell 5 Mätvärden från studien av Kuhn, Oilviera och Fernandes (2008)

Algoritmtyp	**Iterationer**	**Tid 1024*768 pixlar**
MS CPU	500	0,54 s
MS GPU	500	0,26 s
MS GPU (full)	100 per pixel	0,27 s

Det som går att urskilja från datan är att denna adaptiva algoritm är för långsam för att användas i realtid men också att iterationen i algoritmen är väldigt mycket snabbare på GPU när den är implementerad i pixel shadern. Totalt är det nästan samma tidsåtgång för båda GPU versionerna men det är väldigt många flera iterationer som utförs, för CPU versionen tar det ungefär dubbelt så lång tid. Vad som är värt att notera är att skillnaden mellan den adaptiva algoritmen som Kuhn, Oilviera och Fernandes (2008) använder och de statiska algoritmerna, som använts i denna studie, är att de statiska kan implementeras fullständigt för GPU. De GPU implementationer för den adaptiva algoritmen är implementerade både på CPU och GPU, det blir då svårare att jämföra implementationerna mellan varandra då en del av den adaptiva algoritmen alltid görs på CPU.

Denna studie har inte fokuserat på hur bra algoritmerna är på att göra färgerna lättare att urskilja, men det framkom tydligt att det finns uppenbara fel i den viktade HSV roteringsalgoritmen. Om man tittar på Figur 14 och Figur 15 så syns det att färgtoner återkommer flera gånger, både för färgblinda och icke färgblinda, grön till blå färgton återupprepas i två olika områden vilket resulterar i att algoritmen gör det svårare att urskilja färger snarare än lättare. Det kan tänka sig att algoritmen i denna studie är felimplementerad, men den har jämförts med samma bilder som Ribeiro och Gomes (2013) använde och fick fram samma korrigerade bild. Det är troligt en översikt och inte tänkt att algoritmen ska fungera så. Datan i studien bör fortfarande kunna användas som referens då det är roteringen och korrigeringen i sig som tar prestanda. En ändring av hur vikten beräknas behöver inte betyda att det krävs mer prestanda.

För att studien ska kunna återupprepas enklare har den grundläggande shaderkoden som använts i studien publicerats, se Appendix A. Den kod som valts att publiceras är GPU implementation av de algoritmerna som hämtats från andra studier. Då GPU implementationen är en tolkning av andra algoritmer kan den tänkas innehålla eventuella feltolkningar som gör att algoritmerna fungerar annorlunda.

Det bör observeras att trots att inga personer har medverkat i studien kan en användning av material i studien, exempelvis algoritmerna för färgkorrigering, leda till farliga situationer. Att försätta sig i en situation där färger talar om ifall något är farligt eller inte och sedan utgå från att en färgkorrigering kommer göra det lättare att tolka situationen kan vara farligt. Resultatet i denna studie visar hur implementationen påverkar prestandan, inte hur pålitlig en viss färgkorrigering är.

6.2.2 Samhällelig Nytta

Färgblindhet kan vara ett varierande större och mindre handikapp i olika situationer. För situationer där färger används för kommunikation är det ett större handikapp då det är svårare att särskilja vad som kommuniceras. För att kunna prata om nyttan om färgkorrigering i realtid måste först nyttan specificeras. I fall där det viktiga är att kunna se skillnad på färger är färgkorrigering ett effektivt sätt att förbättra möjligheter för färgblinda att göra detta.

Att kunna färgkorrigera i realtid är ett viktigt verktyg för att hjälpa färgblinda se skillnad på färger som konstant ändras i t.ex. spel. I spel är det som regel inte det viktigaste vilken färg som någon ser utan att man ser skillnad mellan två färger, där den ena färgen kan vara allierad och den andra färgen en fiende. Detta behöver inte bara kunna appliceras på spel, exempelvis kan glasögon utformas som i studien av Tanuwidjaja et. al. (2014) eller en mobilapplikation som kan hjälpa till att se skillnad på färger vid exempelvis trafikljus och matprodukter med samma förpackning. För att kunna användas på ett bra sätt måste färgkorrigeringen ske i realtid.

Förutom att hjälpa de som är färgblinda kan liknande applikationer användas för att istället öka förståelsen, hos personer med normal färgsyn, för vad färgblindhet är. En simulering av färgblindhet kan visa ungefär vilka problem som en färgblind person kan ställas inför som en icke-färgblind person inte skulle ställas inför i vanliga fall.

En förbättring för färgkorrigeringsalgoritmer vore om enskilda personer själva kunde ställa in vilka sorters färger som är svåra att se skillnad. Exempelvis skulle man i realtid kunna interaktivt ändra färgerna för att få ett individuellt anpassat resultat. Färgkorrigeringen behöver inte vara bunden till just färgblindhet. Det skulle kunna tänkas att ett realtidsprogram analyserar en given omgivning och sedan gör någon form av färgkorrigering för att förmedla vad programmet ser i omgivningen.

6.2.3 Brister med färgblindkorrigering

Det finns många olika aspekter som tyder på att en algoritm är bra såsom om en algoritm är snabb, om det blir lättare att se skillnad på färger men också hur naturlig en algoritm är. Det kan vara väldigt svårt att definiera hur man ska mäta hur naturlig en algoritm är, det kan handla om ifall korrigerade färger inte ser ut att passa in i kontexten eller att det färgkorrigerade resultatet är mindre estetiskt tilltalande. Något som gör det svårare att definiera är att både att kunna se skillnad på färger och naturligheten kan vara högst

individuell från person till person, speciellt för personer som har avvikande färgkoner där avvikelserna kan vara olika. Det kan tänka sig att vissa färgkorrigeringar förminskar spannet av synliga färger mera än nödvändigt så att färgtoner som en person kan se korrigeras bort.

Det fokus som oftast legat som grund i många av de studier som lästs är att optimera kontrastskillnader i en bild så mycket som möjligt genom adaptiv färgkorrigering. Detta ger bra resultat för syftet som algoritmen ska göra, öka kontrasten i en bild, men är en brist om algoritmen ska användas i ett större perspektiv. Till exempel blir färgerna inte enhetliga, en röd färg kan ibland bli korrigerad till flera olika färger beroende på hur resten av bilden ser ut. Beroende på kontexten kan detta leda till en ökad förväxling av färger. Till skillnad från en statisk färgkorrigering går det inte lära sig vilken färgton en korrigerad färg kan associeras till då den korrigerade färgen kan ändra utseende.

Det är möjligt att om fokus låg mer på att känna igen färger snarare än kontrastökning hade det funnits mera praktiska hjälpmedel som fungerar i många situationer snarare än några väldigt specifika.

6.3 Framtida arbete

Det finns ett flertal intressanta saker som skulle kunna göras, dels som förbättring av denna studie och dels som en fortsättning kring denna studie. För att inte jämföra data som tagits fram på olika hårdvara med olika förutsättningar hade det kunnat göras både en CPU och en GPU implementation för att då jämföra data mellan dessa varianter.

Fortsatt arbete vore att analysera flera statiska algoritmer och determinera hur faktiskt användbara de är för färgblinda personer. Analysen skulle kunna utgå från den nytta färgkorrigering i realtid kan göra och det skulle kunna vara redan existerande, förbättringar av existerande samt egna algoritmer som analyseras. En metod för att bestämma hur hjälpsam en algoritm är skulle kunna vara att göra ett datorspel som innehåller en form av hinderbana. Denna hinderbana ska sedan olika testpersoner ta sig igenom genom att tolka olika färger. Tiden det tar att ta sig igenom med olika färgkorrigeringsalgoritmer kan då jämföras för se hur användbara de är.

Fokus hade kunnat läggas ännu mer på en korrigeringsimplementation för spel. Det är troligt att implementera algoritmerna via materialens shader inte hade passat för alla spel, många spel använder egna shaders för att göra specialeffekter och ljussättning. En lösning hade varit att göra korrigeringen efter den originella utritningen är klar, så kallad post-process, då kan den färdigrenderade bilden ritas ut en gång till med ett material som innehåller korrigeringsalgoritmen. Troligt hade det inte påverkat studiens resultat något, men det hade kunnat vara fördelaktigt för annan part om denna lösning används.

Det är möjligt att det går att göra speciella spelmekaniker med hjälp färgkorrigeringar, tänk istället för att korrigeringen används för att motverka färgblindhet kan det tänkas att det finns element eller pussel i spelet som simulerar färgblindhet eller gör det värre istället för bättre. Det skulle kunna gå att hitta speciella korrigeringsalgoritmer i spelet som är centrala delar för att komma vidare i spelet. Ett dilemma med ett sådant spel skulle vara att färgblinda ser just andra färger så en balans för att ge lika möjlighet skulle vara väldigt svår, det skulle kunna vara lättare att utgå ifrån ett färgblint perspektiv där spelets normalsyn skulle vara en viss färgblindhet och spelet byggs runt detta, dock kvarstår problemet att olika färgblindheter inte beter sig på samma sätt.

Referenser

Blizzard Entertainment (2016). Heroes of the Storm (Version 2.25.4.2) [programvara]. Tillgänglig https://eu.battle.net/shop/en-us/product/overwatch

Blizzard Entertainment (2016). *Overwatch* (Version 1.11.1.2) [programvara]. Tillgänglig https://eu.battle.net/shop/en-us/product/overwatch

Flück, D., (2006*). Color blind essentials.* Zürich, Switzerland: Colblindor

Försvarsmakten (u.å.). *Specialförbandsoperatör*. http://jobb.forsvarsmakten.se/sv/vagen-in/befattningar/specialforbandsoperator/ [2017-01-27]

id Software (2016). *Doom* (Version 1.0.9) [programvara]. Tillgänglig http://store.steampowered.com/app/379720/

Jack, K., (2011). *Video demystified: a handbook for the digital engineer*. ss. 37-67. Elsevier.

Jefferson, L. och Harvey, R., (2006). Accommodating color blind computer users. I *Proceedings of the 8th international ACM SIGACCESS conference on Computers and accessibility*. October 2006, ss. 40-47. DOI:10.1145/1168987.1168996

Kim, H.J., Jeong, J.Y., Yoon, Y.J., Kim, Y.H. and Ko, S.J., (2012). Color modification for color-blind viewers using the dynamic color transformation. I *2012 IEEE International Conference Consumer Electronics (ICCE)*. January 2012, ss. 602-603. DOI:10.1109/ICCE.2012.6162036

Kuhn, G.R., Oliveira, M.M. and Fernandes, L.A., (2008). An efficient naturalness-preserving image-recoloring method for dichromats. *I IEEE transactions on visualization and computer graphics, 14(6),* ss.1747-1754. DOI:10.1109/TVCG.2008.112

Liu, B., Wang, M., Yang, L., Wu, X. and Hua, X.S., (2009). Efficient image and video re-coloring for colorblindness. I *Multimedia and Expo, 2009. ICME 2009. IEEE International Conference*. Juni 2009, ss. 906-909. IEEE. DOI:10.1109/ICME.2009.5202642

Machado, G.M., (2010). *A model for simulation of color vision deficiency and a color contrast enhancement technique for dichromats*. http://hdl.handle.net/10183/26950

Ohkubo, T. and Kobayashi, K., (2008). A color compensation vision system for color-blind people. I *SICE Annual Conference*. Augusti 2008, ss. 1286-1289. DOI:10.1109/SICE.2008.4654855

Polisen (u.å.). *Antagningskrav*. https://polisen.se/Bli-polis/Om-ansokan/Formella-krav/ [2017-01-27]

Poret, S., Dony, R.D. och Gregori, S., (2009). Image processing for colour blindness correction. I *2009 IEEE Toronto International Conference Science and Technology for Humanity (TIC-STH)*. September 2009, ss. 539-544. IEEE. DOI**:**10.1109/TIC-STH.2009.5444442

Ribeiro, M.G. och Gomes, A.J., (2013). A skillet-based recoloring algorithm for dichromats. I *2013 IEEE 15th International Conference on e-Health Networking, Applications & Services (Healthcom)*. Oktober 2013, ss. 702-706. IEEE. DOI:10.1109/HealthCom.2013.6720766

Svensk Pilotutbildning (2015). *Vanliga frågor*. http://www.svenskpilotutbildning.se/fragor-svar/ [2017-01-27]

Tanaka, G., Suetake, N. och Uchino, E., (2010). Lightness modification of color image for protanopia and deuteranopia. *Optical review*, *17*(1), ss. 14-23. DOI:10.1007/s10043-010-0004-9

Tanuwidjaja, E., Huynh, D., Koa, K., Nguyen, C., Shao, C., Torbett, P., Emmenegger, C. and Weibel, N., (2014). Chroma: a wearable augmented-reality solution for color blindness. I *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. September 2014, ss. 799-810. DOI:10.1145/2632048.2632091

Unity Technologies (2017a). *Creating and Using Materials.* https://docs.unity3d.com/Manual/Materials.html [2017-06-21]

Unity Technologies (2017b). *Execution Order of Event Functions.* https://docs.unity3d.com/Manual/ExecutionOrder.html [2017-07-25]

Valve (2013). *Dota 2* (Version 7.06c) [programvara]. Tillgänglig http://store.steampowered.com/app/570/Dota_2/

Appendix A - Grundläggande Shaderkod

Detta appendix innehåller den grundläggande shaderkod för de, till GPU, översatta algoritmerna.

Converteringsmatriser och funktioner

```
float3 rgb2hsv(float3 c)
{
	float4 K = float4(0.0, -1.0 / 3.0, 2.0 / 3.0, -1.0);
	float4 p = lerp(float4(c.bg, K.wz), float4(c.gb, K.xy), step(c.b, c.g));
	float4 q = lerp(float4(p.xyw, c.r), float4(c.r, p.yzx), step(p.x, c.r));

	float d = q.x - min(q.w, q.y);
	float e = 1.0e-10;
	return float3(abs(q.z + (q.w - q.y) / (6.0 * d + e)), d / (q.x + e), q.x);
}

float3 hsv2rgb(float3 c)
{
	float4 K = float4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
	float3 p = abs(frac(c.xxx + K.xyz) * 6.0 - K.www);
	return c.z * lerp(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}

float changeRange(float val, float oldMin, float oldMax, float newMin, float newMax)
{
	return (((val - oldMin) * (newMax - newMin)) / (oldMax - oldMin)) + newMin;
}

static const float3x3 _rgb2lms =
{
	17.8824, 43.5161, 4.11935,
	3.45565, 27.1554, 3.86714,
	0.02996, 0.184309, 1.46709
};

static const float3x3 _lms2rgb =
{
	0.0809445, -0.130505, 0.1167211,
	-0.0102485, 0.0540193, -0.113615,
	-0.0000365, -0.0041216, 0.6935114,
};

static const float3x3 _lmsCBsim =
{
	1.0, 0.0, 0.0,
	0.494207, 0.0, 1.24827,
	0.0, 0.0, 1.0
};

static const float3x3 _rgbShift =
{
	0.0, 0.0, 0.0,
	0.7, 1.0, 0.0,
	0.7, 0.0, 1.0
};
```

HSV rotering

```
fixed4 frag (v2f i) : SV_Target
{
	fixed4 col = tex2D(_MainTex, i.uv);

	// shift huerange
	float3 colHSV = rgb2hsv(col.rgb);
	colHSV.r = changeRange(colHSV.r, 0, 1, 0.125, 0.875);
	fixed3 newCol = hsv2rgb(colHSV);

	return col;
}
```

Viktad HSV rotering

```
fixed4 frag (v2f i) : SV_Target
{
	fixed4 col = tex2D(_MainTex, i.uv);

	// skillet shift huerange
	float3 colHSV = rgb2hsv(col.rgb);

	float useDelta = step(colHSV.r, 60.0 / 360.0) + step(280.0 / 360.0, colHSV.r);

	float3 maxedSV = hsv2rgb(fixed3(colHSV.r, 1, 1));
	float magenta = 1 - maxedSV.g;

	float delta = (colHSV.r - 1.0 / 3.0) * magenta; // 130/360

	colHSV.r += delta * useDelta;

	fixed3 newCol = hsv2rgb(colHSV);

	return col;
}
```

Daltonisering

```
fixed4 frag (v2f i) : SV_Target
{
	fixed4 col = tex2D(_MainTex, i.uv);

	//daltonize
	fixed3 RGBcb = mul(_lms2rgb, mul(_lmsCBsim, mul(_rgb2lms, col.rgb)));
	fixed3 RGBs = mul(_rgbShift, col.rgb - RGBcb);
	fixed3 RGBd = col.rgb + RGBs;

	return col;
}
```