Examensarbete

HÖGSKOLAN I SKÖVDE

1977

SHADERPRESTANDA OCH OPTIMERING I UNITY

Skillnad i prestanda mellan HLSL och Unity Shader Graph shaders

SHADER PERFORMANCE AND OPTIMIZATION IN UNITY

Difference in performance between HLSL and Unity Shader Graph shaders

Examensarbete inom huvudområdet Informationsteknologi
Grundnivå 15 högskolepoäng
Vårtermin 2025

Johan Kvist

Examinator: Sanny Syberfeldt

Sammanfattning

Inom spelutveckling krävs skapandet av shaders för att uppnå vissa visuella effekter. Denna studie fokuserar på att jämföra prestandan hos shaders utvecklade med Unity Shader Graph (USG) mot shaders skapade med High-Level Shader Language (HLSL). De HLSL-shaders som utvärderas använder optimeringstekniker enligt Crawford och O'Boyle (2018).

Syftet med studien är att testa prestandaskillnader mellan shaders gjorda och implementerade med USG jämfört med de i HLSL. Resultaten av analysen visar att shaders som utvecklades i HLSL inte uppvisar några betydande prestandaförbättringar jämfört med de som skapades i USG. Detta tyder på att valet av shaderspråk i spelutveckling kan vara av mindre betydelse för att uppnå prestandaförbättringar och att användningen av USG kan vara ett användarvänligare alternativ utan att kompromissa med prestanda i jämförelse med HLSL. Vidare forskning och analys är dock viktigt för att få en djupare förståelse av shaderskapandets påverkan på renderingsprestandan inom spelapplikationer.

Nyckelord: Optimering, Shaders, Unity Shader Graph, High-Level Shading Language, Spel

Innehållsförteckning

1 Introduktion ... 1

2 Bakgrund ... 2

2.1 Spelindustrin och Unity ... 2
2.2 Prestanda ... 2
2.3 Shaders ... 3
2.4 Optimering av shaders ... 4
2.5 Shaders i Unity ... 5

3 Problemformulering ... 8

3.1 Frågeställning ... 8
3.2 Metodbeskrivning ... 8
3.2.1 Artefakter ... 8
3.2.2 Datainsamling och databehandling ... 9

4 Implementation ... 11

4.1 Glas-shader ... 11
4.2 Vattenshader ... 13

5 Utvärdering ... 15

5.1 Presentation av undersökning ... 15
5.1.1 Resultat av glas-shader ... 15
5.1.2 Resultat av vattenshader ... 18
5.2 Analys ... 19
5.2.1 Glas-shader ... 20
5.2.2 Vattenshader ... 20
5.3 Slutsatser ... 21

6 Sammanfattning och diskussion ... 22

6.1 Sammanfattning ... 22
6.2 Diskussion ... 22
6.3 Samhälleliga och etiska aspekter ... 24
6.4 Framtida arbete ... 24

Referenser ... 26

1 Introduktion

I strävan efter en givande spelupplevelse är en jämn och hög bildhastighet (FPS) avgörande. Spelare har en tendens att föredra spel som erbjuder en hög FPS, vilket resulterar i en mer responsiv och engagerande spelupplevelse (Claypool & Claypool, 2007). För att upprätthålla en hög prestanda i dagens mer komplexa spel är optimering viktigt, särskilt när det kommer till grafiska resurser som shaders. Shaders, som används för att reglera hur ytor renderas, spelar en central roll i att skapa visuellt tilltalande spel. Två vanliga metoder för att skapa shaders är genom High-Level Shader Language (HLSL) eller Unity Shader Graph (USG), ett verktyg utvecklat av Unity. Här uppstår en intressant fråga: Vilka prestandaresultat kan förväntas när shaders skapade med USG jämförs med de utvecklade i HLSL?

För att besvara denna fråga har en systematisk metod tillämpats. Shaders har utvecklats med både USG och HLSL med målsättningen att uppnå så likvärdig visuell representation som möjligt. Shaderna som utvecklades har genomgått optimeringsförsök och deras prestanda har därefter jämförts i detalj. För att belysa potentiella variationer i prestanda har resultaten analyserats och presenterats genom figurer och tabeller. Efter analysen konstaterades att det fanns en prestandaskillnad mellan shaders implementerade i HLSL och USG, men att skillnaden var marginell och kunde bero på flera faktorer.

2 Bakgrund

Bakgrunden ger en översikt över varför prestanda är avgörande inom spelutveckling, samt hur olika variabler påverkar och mäts i relation till prestanda. Det belyses också hur spelmotorer som Unity, med verktyg som Unity Shader Graph, används för att skapa och hantera shaders, vilket påverkar både visuell kvalitet och prestanda. Vidare diskuteras hur optimeringar av grafik och kod är nödvändiga för att möta de ständigt växande kraven på högre prestanda och skapa en bättre spelupplevelse.

2.1 Spelindustrin och Unity

Spelindustrin är en kontinuerligt växande marknad världen över, men framför allt i Europa och Sverige. I sin senaste årliga rapport redovisar Interactive Federation of Europe att intäkterna ökade med 22 procent från 2019 till 2020 (ISFE & EGDF 2021, s. 18). Även i Sverige växer spelmarknaden kraftigt; enligt Spelutvecklarindex 2024 har omsättningen nästan dubblerats de senaste fem åren (Spelbranschen 2024).

Tekniskt har spelindustrin utvecklats de senaste åren, med bättre grafik och ljud, mer realistiska fysiksimuleringar och förbättrad artificiell intelligens. En spelmotor är ett program som underlättar skapandet av spel genom att hantera viktiga komponenter som fysiksimulering, grafik och animationer. Unity används av utvecklare med varierande förkunskaper då det är en intuitiv spelmotor med låg inlärningskurva, omfattande dokumentation och guider (Sharp Coder Blog, u.å.). Ett exempel på en populär mobilapplikation som använder Unity är Pokémon GO, som lanserades 2016 (Unity Technologies 2005).

Unity är populärt av flera skäl. För det första är det en intuitiv spelmotor med låg inlärningskurva. Det har också en flexibilitet som gör det möjligt för användare med varierande förkunskaper att skapa en mängd olika typer av spel, från enkla mobilspel till avancerade PC- och konsolspel. Dessutom är Unity en av de mest tillgängliga spelmotorerna eftersom den är kostnadsfri att använda. Med tidigare presenterade fördelar i beaktning är Unity lämpad som spelmotor för denna studie.

2.2 Prestanda

Prestanda är en avgörande faktor för att skapa en optimal spelupplevelse. Med den tekniska utvecklingen och de ständigt växande kraven på högre grafik- och processorkapacitet, ställs nya krav på hårdvara och optimering. För att kunna köra mer komplexa spel, särskilt de med avancerad grafik, måste både spel och plattformar kunna upprätthålla hög och stabil prestanda. En av de mest centrala aspekterna av prestanda är bildhastigheten (FPS), som har en direkt inverkan på spelupplevelsen, särskilt i snabbare, mer intensiva spel.

Vikten av bildhastighet bekräftas i en studie av Kajal T. Claypool och Mark Claypool (2007). De visar att högre FPS inte bara förbättrar spelupplevelsen utan också spelarens prestation. Studien fann att spelare presterar betydligt bättre i förstapersonsskjutspel vid högre bildhastigheter, där snabb reaktionstid och precision är avgörande. Spelare som spelade med 60 FPS eller högre upplevde spelet som mer responsivt och presterade bättre jämfört med de som spelade med 30 FPS.

En högre FPS även bidra till att minska ögonbelastning och trötthet. Claypool & Claypool (2007) visar att en hög bildhastighet i vissa fall kan leda till en mer behaglig visuell upplevelse, vilket gör det möjligt för spelare att spela längre sessioner utan att uppleva samma nivå av fysisk ansträngning som vid lägre FPS. Enligt Cervantes (2025) förbättrar hög bildhastighet inte bara den visuella jämnheten utan minskar även inmatningsfördröjningen, det vill säga den tid som går mellan en användares handling och skärmens respons, vilket leder till en ökad responsiv spelupplevelse.

När det gäller plattformar spelar prestandan en central roll. Datorer, med sina mer kraftfulla komponenter och anpassningsmöjligheter, kan ofta uppnå högre FPS än konsoler eller mobila enheter. Detta innebär att spelupplevelsen kan variera beroende på plattform, vilket gör det ännu viktigare att optimera både grafik och kod för att säkerställa en jämn och stabil upplevelse på alla enheter.

Sammanfattningsvis är prestanda en nyckelfaktor för att säkerställa en positiv spelupplevelse. Hög FPS och optimerad grafik är inte bara preferenser, utan centrala komponenter för att uppnå en smidig och engagerande upplevelse, särskilt i snabba actionspel. Studien från Claypool & Claypool bekräftar vikten av att satsa på hög För att ge spelare en så optimal spelupplevelse som möjligt.

2.3 Shaders

Shaders är en grundläggande del av datorspelsgrafik och spelar en viktig roll i rendering av grafik. En shader är en samling kod som körs på grafikkortet och styr hur ytan på ett objekt ska renderas. Shaders används för att skapa allt från texturer och färger till ljus- och skuggeffekter, vilket bidrar till att skapa visuellt tilltalande och realistiska spelmiljöer. Genom att använda shaders kan utvecklare skapa detaljerade och dynamiska effekter som gör spelvärlden mer levande och engagerande.

En shader är vanligtvis uppdelad i två huvudkomponenter: vertex-shadern och fragment-shadern. Vertex-shadern är ansvarig för att transformera varje punkt i en 3D-modell till en 2D-yta som kan visas på en skärm. Detta inkluderar att flytta punkterna, rotera dem och skala dem för att matcha den virtuella kamerans perspektiv, vilket bidrar till att skapa mer verklighetstrogna karaktärer och miljöer.

Fragment-shadern är ansvarig för att bestämma färgen för varje pixel på skärmen baserat på information från vertex-shadern. Detta inkluderar att beräkna hur mycket ljus som träffar ytan på objektet och hur mycket skuggor som ska kastas. Fragment-shaders möjliggör mer detaljerade och dynamiska ljus- och skuggeffekter, vilket bidrar till att göra spelvärldarna mer realistiska.

Vad som vill uppnås med en shader kan variera beroende på spelets visuella mål. Shaders ger spelutvecklare möjlighet att skapa unika visuella effekter som vatten, vågor eller realistiskt glas, vilket anpassar spelupplevelsen visuellt. Ett exempel på detta är användningen av shaders för att generera vatten med realistiska vågeffekter (figur 1).

Figur 1 Bild på hav med vågor skapad med shaders (Shadertoy: afl_ext 2017)

2.4 Optimering av shaders

En väl optimerad shader kan göra en skillnad för prestandan i ett spel. Enligt Lam (2016) finns det flera sätt att optimera shaders för spelutveckling, såsom att minska antalet instruktioner, använda lägre upplösning på texturer och undvika onödiga beräkningar. En annan viktig faktor att tänka på vid optimering av shaders är att undvika överflödiga beräkningar på grafikkortet. Till exempel kan *frustum culling* användas för att bara rendera objekt som är synliga på skärmen, i stället för att rendera alla objekt i scenen.

När det gäller att skriva effektiva shaders är det också viktigt att använda sig av rätt algoritmer och tekniker för att uppnå det önskade resultatet. Detta innefattar att använda rätt typ av ljusberäkningar, material och texturer. Samtidigt är det lätt hänt att antalet så kallade *draw calls* blir för högt, det är en av flera saker som Unity själva varnar för (Unity u.å. b). Detta problem kan bli extra tydligt när vissa grafiska effekter används, som kan vara resurskrävande om de inte hanteras korrekt, till exempel *ambient occlusion* och *bloom*. En vanlig optimeringsteknik som tidigare nämndes är *view frustum culling* som är en process för att stoppa renderingen av objekt som inte är i skärmens synfält (Su et al. 2017).

Crawford och O'Boyle (2018) lägger fram metoder som eventuellt kan förbättra skrivandet av fragment-shaderkod. Vissa av metoderna som de visar är av simplare natur som exempelvis omstrukturering av aritmetik, se figur 2. Division är mer kostsamt än multiplikation och bör därför om möjligt skrivas om till multiplikation av inversen. Kompilatorn kan optimera vissa av sig självt men koden borde vara optimerad från början för att garantera maximal optimering.

$$a + a + a \rightarrow 3a$$

Figur 2 Omstrukturering av aritmetik för optimering av shader

Då shaders påverkar prestanda kan optimering av shaders vara avgörande för att uppnå hög prestanda i spel och andra applikationer. En källa som tar upp vikten av optimering av shaders för prestanda i spel är artikeln *The Importance of Shader Optimization for Game Developers* skriven av Joel Bradley på Pluralsight (2018). I artikeln betonas vikten av att optimera shaders för att maximera prestandan i spel, särskilt på mobil och konsol där hårdvaran är mer

begränsad. En teknik som Joel Bradley nämner för att optimera shaders är att förenkla eller ta bort onödiga eller för komplexa delar av shaderkoden. Detta kan hjälpa till att förbättra prestandan utan att påverka det visuella resultatet. Liknande tekniker nämns i Unitys dokumentation om shaderprestanda (Unity u.å. a), där förberäkningar av potentiella konstanter borde göras i förhand för att undvika onödiga kalkyleringar under runtime. Andra rekommendationer är att använda variabler med så liten precision som möjligt om det inte drastiskt ändrar resultatet. De tre datatyperna som existerar för nummer inom HLSL är *float* som är 32-bits, *half* som är 16-bits och sedan *fixed* som generellt sett är 11 bits. Dokumentationen föreslår att använda *float* där mer precisa beräkningar som trigonometri och multiplikationer används.

2.5 Shaders i Unity

Ett programmeringsspråk som används för att skapa shaders är *High-Level Shader Language* (HLSL). HLSL används för pipelinen Direct3D 11 som är gjort för 3D rendering där prestanda är ett fokus. Det är en av huvudanledningarna till att det används i Unity, som är en av de vanligaste spelmotorerna (Dillet 2018).

Att skriva shaders i HLSL, som kan se ut som i figur 3, kan vara komplicerat för nya användare så Unity har utvecklat *Unity Shader Graph* (USG). USG är ett grafiskt scripting-verktyg som skapar noder som representerar funktioner eller variabler, se figur 4. Det används för att skapa shaders på ett enklare sätt. Med detta verktyg är det enklare att observera och förstå händelserna vid varje nod, då USG har en förhandsvisning av den skapade shadern. USG kompilerar sedan automatiskt sina grafer till HLSL. Eftersom koden genereras kanske den inte är fullständigt optimerad enligt alla principer som Crawford och O'Boyle (2018) föreslår.

```
1   Shader "Unlit/NewUnlitShader"
2   {
3       Properties
4       {
5           _MainTex ("Texture", 2D) = "white" {}
6       }
7       SubShader
8       {
9           Tags { "RenderType"="Opaque" }
10          LOD 100
11
12          Pass
13          {
14              CGPROGRAM
15              #pragma vertex vert
16              #pragma fragment frag
17              // make fog work
18              #pragma multi_compile_fog
19
20              #include "UnityCG.cginc"
21
22              struct appdata
23              {
24                  float4 vertex : POSITION;
25                  float2 uv : TEXCOORD0;
26              };
27
28              struct v2f
29              {
30                  float2 uv : TEXCOORD0;
31                  UNITY_FOG_COORDS(1)
32                  float4 vertex : SV_POSITION;
33              };
34
35              sampler2D _MainTex;
36              float4 _MainTex_ST;
37
38              v2f vert (appdata v)
39              {
40                  v2f o;
41                  o.vertex = UnityObjectToClipPos(v.vertex);
42                  o.uv = TRANSFORM_TEX(v.uv, _MainTex);
43                  UNITY_TRANSFER_FOG(o,o.vertex);
44                  return o;
45              }
46
47              fixed4 frag (v2f i) : SV_Target
48              {
49                  // sample the texture
50                  fixed4 col = tex2D(_MainTex, i.uv);
51                  // apply fog
52                  UNITY_APPLY_FOG(i.fogCoord, col);
53                  return col;
54              }
55              ENDCG
56          }
57      }
58  }
59
```

Figur 3 En standard HLSL shader för en ”Unlit” shader i Unity

Figur 4 Exempel på shader skapad i Unitys Unity Shader Graph (USG) (https://unity.com/features/shader-graph)

3 Problemformulering

Detta kapitel presenterar den frågeställning som är grunden för rapporten. Kapitlet presenterar även hur data samlas in och hur det sedan hanteras. Viss problematik med metodiken tas även upp.

3.1 Frågeställning

Prestanda inom spel är en avgörande faktor för att säkerställa en positiv spelupplevelse. En hög och jämn bilduppdateringsfrekvens är särskilt viktig eftersom den gör spelet mer responsivt och visuellt tilltalande (Claypool & Claypool 2007). Att bevara prestandan genom att använda sina verktyg och tillgångar på klokt vis kan underlätta detta. Om två likadana shaders skapas i HLSL och USG sedan optimeras med principerna som Crawford och O'Boyle (2018) nämner, blir det någon skillnad på prestanda? För att undersöka skillnaden väljs frågeställningen:

"Finns det en skillnad i prestanda mellan en shader implementerad i HLSL och en shader implementerad i USG i Unity?"

3.2 Metodbeskrivning

För att besvara frågeställningen utvecklades först en shader i USG, följt av en shader utvecklad i HLSL. Båda shaders har optimerats så långt det varit möjligt inom ramen för denna studie, för att säkerställa att de presterar på bästa sätt inom sina respektive implementationsmiljöer. Målet har varit att de ska vara så lika som möjligt i utseende och funktionalitet, med den huvudsakliga skillnaden att den ena är implementerad i HLSL och den andra i USG.

Denna process har upprepats för två olika shaders för att jämföra skillnaden i prestanda mellan flera typer av grafiska effekter. Detta minskar risken att resultaten påverkas av en möjlig mänsklig faktor i form av suboptimal implementation och kan ge en bredare förståelse för skillnaderna i prestanda mellan de två metoderna.

De shaders som valdes för denna studie är en vattenshader och en glas-shader. Dessa valdes eftersom de är vanligt förekommande inom spelutveckling och representerar två typer av grafiska effekter som ställer olika krav på teknisk implementation. Genom att välja shaders med olika funktioner kan studien fånga eventuella prestandaskillnader beroende på shaderspecifika tekniska utmaningar.

För att analysera skillnaden i prestanda mellan de olika shaderserna har ett oberoende t-test (independent samples t-test) genomförts, enligt metoden beskriven av Chatzi (2024). Detta test används för att avgöra om det finns en statistiskt signifikant skillnad i prestanda mellan en shader implementerad i HLSL och en shader implementerad i USG.

3.2.1 Artefakter

De två typer av shaders som valdes ska spegla vanligt förekommande shaders inom spelutveckling. Genom att inkludera både en glas-shader och en vattenshader representeras två olika kategorier av grafiska effekter, vilket gör det möjligt att analysera hur olika shaderfunktioner påverkar prestandan.

Dessa två shaders valdes eftersom de har olika tekniska krav och funktioner, vilket kan vara avgörande när man jämför hur USG och HLSL hanterar dem. Vattenshadern innehåller rörliga element och djupbaserad färghantering, medan glas-shadern fokuserar på transparens och ljusrefraktion. Att undersöka två olika shaders ger en bredare insikt i hur implementeringen av shaders påverkar prestandan beroende på funktionalitet.

De specifika shaders som utvecklades är:

- En vattenshader. En shader som används på ett tesselerat plan. En vanligt förekommande shader som används för att skapa både realistiskt och tecknat vatten. Effekter som kommer inkluderas i denna shadern är bland annat fysiska vågor, *normal maps* som rör sig, vattenskum vid vattnets kanter och färgjustering beroende på djup.
- En glas-shader. En shader som används på plan eller på objekt, ofta rätblock. Även det är vanligt förekommande i spel. Effekter som den planeras inneha är bland annat ljusrefraktion och användning av *normal map* för att skapa effekten av till exempel skadat eller deformerat glas.

3.2.2 Datainsamling och databehandling

Denna undersökning baserades på en kvantitativ metod där mätvärden samlades in genom empiriska tester av shaders. Unitys inbyggda *profiler* användes för att mäta prestanda och resultaten analyserades vidare med Unitys *Profile Analyzer*. Den primära datan som samlades in var renderingstiden för hela scenen som testobjektet befinner sig i, mätt i millisekunder. Dessa shaders renderades över en längre tidsperiod, men eftersom *profilern* har en gräns på 299 frames, registrerades och analyserades endast dessa. Medelvärdet och spridningsmått beräknades och datan exporterades till kalkylprogrammet Excel.

För att statistiskt analysera skillnaden mellan shaders implementerade i HLSL och USG genomfördes ett oberoende t-test (independent samples t-test), enligt metoden beskriven av Chatzi (2024). Ett t-test används för att avgöra om skillnaden mellan två oberoende grupper är statistiskt signifikant (Chatzi 2024). Eftersom de två implementationerna av shaders representerar separata grupper med oberoende mätvärden är detta test en lämplig metod för att undersöka om skillnaden i prestanda är statistiskt pålitlig. Även om studien av Chatzi (2024) är inriktad på sjukvård, är den statistiska metoden generell och kan tillämpas på andra forskningsområden där man jämför två oberoende grupper. I denna studie testades huruvida medelvärdet för renderingstiden skiljde sig signifikant mellan de två shader-implementationerna. Resultatens signifikansvärde (p-värde) analyserades för att avgöra om eventuella skillnader kunde tillskrivas slumpen eller om de indikerade en faktisk prestandaskillnad.

För att formellt testa detta formulerades följande hypoteser:

- Nollhypotes (H_0): Det finns ingen signifikant skillnad i renderingstid mellan shaders implementerade i USG och HLSL.
- Alternativ hypotes (H_1): Det finns en signifikant skillnad i renderingstid mellan shaders implementerade i USG och HLSL.

Signifikansnivån sattes till 0.05, vilket är en vanlig standard inom statistisk analys. Om p-värdet från t-testet, som beräknas med hjälp av Excels t-fördelningsfunktion, är mindre än 0.05, avvisas nollhypotesen. Detta indikerar att skillnaden i renderingstid mellan shader-implementationerna är statistiskt signifikant. Standardavvikelsen för varje mängd beräknas

med hjälp av övre och nedre kvartil, eftersom Unitys *profiler* inte kunde exportera alla 299 datavärden till Excel. Genom att använda kvartiler kunde en representativ uppskattning av spridningen ändå erhållas.

Flera faktorer kan påverka undersökningens resultat, bland annat eventuella brister i shader-implementationerna. För att minska denna risk har implementationerna itererats och testats löpande med de tidigare nämnda metoderna för att säkerställa en rättvis jämförelse. En annan potentiell felkälla är variationer i hårdvarans prestanda, då vissa grafikprocessorer kan ge olika procentuella prestandavinster (Crawford & O'Boyle 2018). För att undvika detta har samtliga tester genomförts på samma utrustning under identiska förhållanden.

4 Implementation

Detta avsnitt beskriver hur varje shader har implementerats med hänsyn till dess designval och eventuella förenklingar som har gjorts för att uppnå en liknande estetisk känsla. Avsnittet diskuterar även de utmaningar som uppkom under implementationsprocessen samt de åtgärder som vidtogs för att hantera dessa problem.

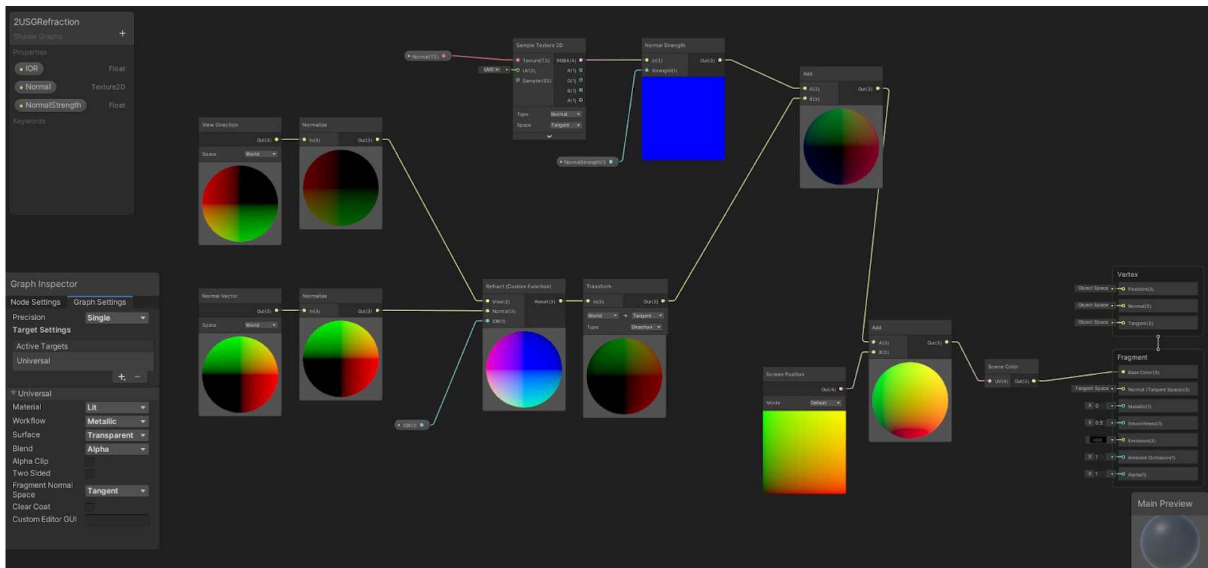
För implementeringen av HLSL-shaders har inspiration och vägledning hämtats från den allmänna shaderkunskap som Freya Holmér (2021) förmedlar i sina videor. Även om hon inte erbjuder någon specifik tutorial för vattenshader eller glas-shader, ger hennes material värdefulla insikter i hur man skriver och strukturerar shaders i HLSL. Denna kunskap har varit en viktig grund vid utvecklingen av båda shaders.

4.1 Glas-shader

En glas-shader används främst för att simulera genomskinliga och ljusbrytande material på ytor, såsom glasfönster, kristaller eller andra transparenta objekt. I spel är glas-shaders vanligt förekommande, särskilt på olika rätblock eller plana ytor, där de används för att skapa realistiska visuella effekter som ljusrefraktion och förvrängning. En viktig egenskap hos en glas-shader är möjligheten att visa det som finns bakom objektet på ett naturligt sätt, vilket kräver hantering av ljus, transparens och ytförvrängningar. Ljusrefraktionen i en glas-shader innebär att objekten bakom ytan förskjuts, vilket ger intrycket av att ljuset bryts genom materialet. Med hjälp av en *normal map* som textur kan även effekten av skadat eller förvrängt glas återskapas då denna lägger till extra ljusrefraktion. Styrkan på dessa effekter är viktiga att kunna justera, till exempel för att skapa en kraftigare effekt på tjockare glasobjekt eller kristaller, där en tydligare brytning kan bidra till ett mer imponerande och dynamiskt utseende.

Skapandet av experimentets första glas-shader i USG inspirerades av PabloMakes (2021) *Refraction ShaderGraph tutorial for Unity*. USG visade sig vara smidigt för justering av shadern tack vare dess visuella gränssnitt. Med hjälp av USGs kontinuerliga förhandsvisning, som ses i figur 5, kunde justeringar göras snabbt utan att behöva spara shadern och byta till scenvyn vid varje ändring. Detta var en klar fördel när olika materialegenskaper och ljusbrytningsvärden testades.

Utvecklingen av den andra glas-shadern i HLSL var mer komplex då liknande guider inte kunde hittas. Strategin blev därför att utgå från den USG-shader som utvecklats, analysera dess struktur och översätta dess funktionalitet till HLSL med hjälp av den grundläggande kunskap inom shaders som Freya Holmér (2021) lär ut. Målet var att efterlikna effekten från USG-versionen med så likvärdig prestanda som möjligt. Unitys dokumentation för HLSL och USG uppfattades stundvis mer förvirrande än hjälpande då vissa funktioner inte är likadana i båda, trots samma namn. Vissa texturtagare var dessutom förskjutna vilket kunde göra att det rent visuellt såg annorlunda ut.



Figur 5 Slutgiltiga glas-shadern i USG

Dokumentationen för de olika USG-noderna hade ibland felaktig kod för hur samma funktioner skulle användas i HLSL. En nod i USG *Scene Color* var en av de med felaktig kod i dokumentationen. Namnet på funktionen (figur 6) stämde inte för användning i HLSL.

```
void Unity_SceneColor_float(float4 UV, out float3 Out)
{
    Out = SHADERGRAPH_SAMPLE_SCENE_COLOR(UV);
}
```

Figur 6 Vad Scene Color nodens funktion heter och hur den ska fungera enligt Unitys dokumentation

För att få åtkomst till kamerans färgbuffert krävdes alternativa tillvägagångssätt, vilket i flera fall ledde till instabilitet i shadern. De tidigare försöken att hantera detta resulterade i oväntade visuella artefakter och funktionsbortfall i andra delar av shadern. Slutligen implementerades en ny lösning där funktionen *SAMPLE_TEXTURE2D* användes tillsammans med *CameraOpaqueTexture* för att korrekt extrahera färgbufferten.

Under optimeringsarbetet identifierades ytterligare möjligheter att minska beräkningskostnaden. I enlighet med de optimeringsprinciper som Crawford och O'Boyle (2018) beskriver, ersattes *float* med *half* i delar av shadern där hög precision inte var nödvändig. Denna förändring syftade till att minska minnesanvändningen se figur 7 och figur 8.

Den slutgiltiga HLSL-glasshadern utför nu alla beräkningar som avsetts och koden finns bifogad som **Appendix A**.

```
//Före
float _RefractionStrength;
float _NormalMapRefractionStrength;
float4 _Tint;
```

```
//Efter
half _RefractionStrength;
half _NormalMapRefractionStrength;
half4 _Tint;
```

Figur 7 Byte från float till half vid materialegenskaper

```
//Före
float2 normalOffset = modifiedNormal.xy * _RefractionStrength;
float2 refractedUV = i.screenUV + normalOffset;
```

```
//Efter
half2 normalOffset = modifiedNormal.xy * _RefractionStrength;
half2 refractedUV = i.screenUV + normalOffset;
```

Figur 8 Byte från float till half i beräkningen av UV-förskjutningar

4.2 Vattenshader

En vattenshader används inom spel främst för att simulera realistiska eller stiliserade vattenytor. Dessa shaders är vanligt förekommande i spel som inkluderar sjöar, floder, hav eller andra vattenkällor. De används ofta på tesselerade plan för att skapa en levande och dynamisk vattenyta. En vattenshader innefattar vanligtvis tre centrala aspekter: färghantering baserat på djupet, fysisk rörelse i form av vågor och rörliga *normal maps*.

Den djupbaserade färghanteringen gör att vattnets färg förändras beroende på hur djupt det är från kamerans vy, vilket skapar en naturlig övergång från ljusare mer transparenta områden nära ytan till mörkare och mer ogenomskinliga nyanser på djupare delar. Den fysiska rörelsen i en vattenshader består oftast av vågeffekter som med hjälp av brus förskjuter punkter av det tesselerade planet. Rörelsens intensitet och hastighet kan justeras för att passa olika miljöer, från lugna sjöar till stormiga hav. För att ytterligare förstärka vattenytans visuella rörelse används *normal maps* som rör sig över ytan. Dessa skapar en illusion av små vågor och krusningar på vattenytan, vilket gör att vattnet ser mer levande och detaljerat ut.

Båda dessa skapade shaders saknar funktionen med *normal maps* som rör sig. Funktionen lyckades lösas i USG-shadern men eftersom den inte fungerade i shadern gjord i HLSL valdes den funktionen att tas bort så de båda skulle vara lika varandra. USG-shadern implementerar *Gradient Noise*-noden för fysiska vågor, medan HLSL-shadern använder sig av det genererade kodexemplet från Unitys dokumentation. Det bör noteras att, som tidigare nämnts, detta inte alltid resulterar i identiska resultat, vilket också var fallet för fysiska vågor i detta sammanhang. När vattenshaderns vågor är mycket stora kan det uppstå artefakter i HLSL-shadern, där ytor mellan vertiserna ibland blir felaktigt förskjutna. Dessa artefakter blir emellertid inte märkbara om mer vanlig storlek på vågorna används.

I glas-shadern som utvecklades, fungerade USG-shadern som baslinje för att guida och sätta standarden för HLSL-shadern. Vid utvecklingen av vattenshadern i USG användes HLSL-shadern som referens för att implementera effekten av vattenskum vid vattenkanten. I HLSL var det en simpel *if-sats* följt av vilka färger som vattnet skulle interpolera mellan. USG har inte en simpel *if-sats* som kan användas lika enkelt. I USG krävdes det i stället att en *Comparison*-nod användes tillsammans med flera *Branch*-noder för att uppnå samma resultat. En nackdel med att använda flera *Branch*-noder är att båda sidorna av grenen kommer att beräknas i shadern, även om en av dem aldrig används i utdatan. I syfte att säkerställa en effektiv beräkning och minimera överflödig arbetsbelastning, är det önskvärt att ha dessa beräkningar så tidigt som möjligt i grafen.

Under utvecklingen av vattenshadern identifierades vissa optimeringsmöjligheter för att förbättra prestandan utan att påverka den visuella kvaliteten. I enlighet med de optimeringsprinciper som Crawford och O'Boyle (2018) beskriver testades två specifika förändringar för att minska beräkningskostnaden:

1. Byte från float till half för vissa beräkningar

Enligt Crawford och O'Boyle (2018) kan minnesanvändningen och beräkningshastigheten förbättras genom att ersätta *float* med *half* i situationer där hög precision inte är nödvändig. Därför genomfördes denna förändring i shadern på relevanta platser. Figur 9 visar före- och efterbild av denna optimering.

```
//Före
float _Depth;
float _FoamDepth;
float _Strength;
float _Displacement;
float4 _DeepWaterColor;
float4 _ShallowWaterColor;
float4 _FoamWaterColor;
```

```
//Efter
half _Depth;
half _FoamDepth;
half _Strength;
half _Displacement;
half4 _DeepWaterColor;
half4 _ShallowWaterColor;
half4 _FoamWaterColor;
```

Figur 9 Före och efter byte från float till half i HLSL-shadern

2. Byte från division till multiplikation med inversen

Division är en mer beräkningskrävande operation än multiplikation och Crawford och O'Boyle (2018) föreslår att divisioner med konstanter bör ersättas med multiplikation av inversen för att förbättra prestandan. Figur 10 illustrerar denna förändring, där en division ersatts med en multiplikation.

```
//Före
_Displacement /= 10000;
v.vertex.z = vWave * _Displacement;
```

```
//Efter
_Displacement *= 0.0001;
v.vertex.z = vWave * _Displacement;
```

Figur 10 Före och efter byte från division till multiplikation med inversen

För fullständiga HLSL-shadern se **Appendix B**.

5 Utvärdering

Detta kapitel presenterar resultaten av prestandamätningarna för de implementerade shadersen. Inledningsvis beskrivs de observerade resultaten både visuellt och prestandamässigt, följt av en statistisk analys. Slutligen diskuteras slutsatserna som kan dras baserat på analysen.

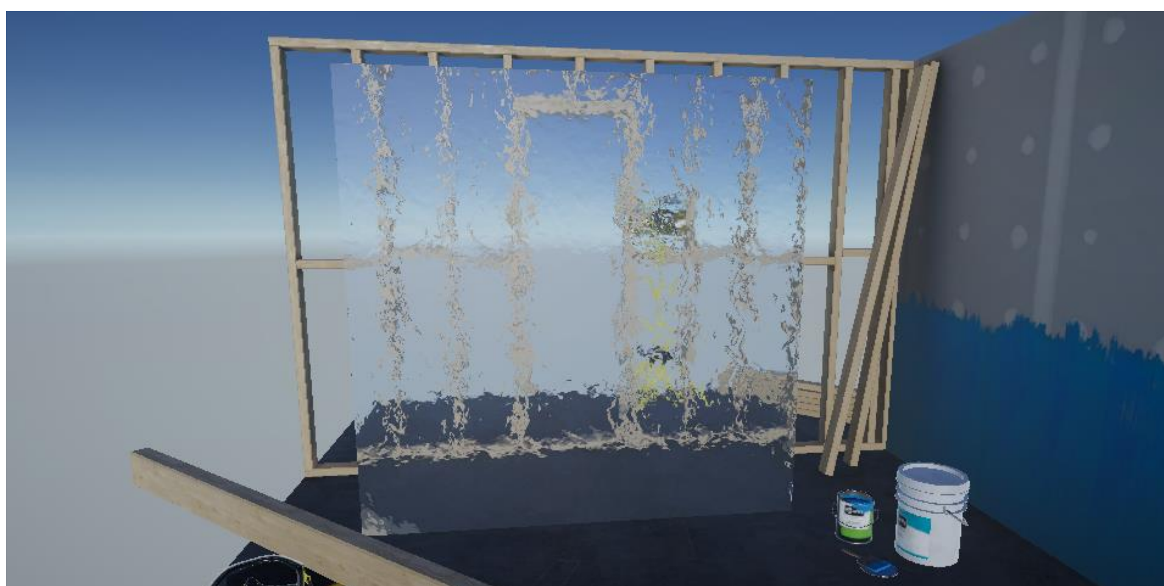
5.1 Presentation av undersökning

I denna undersökning testades prestandan för shaders implementerade **i** Unity Shader Graph (USG) och HLSL för att utvärdera eventuella prestandaskillnader. För att undersöka detta skapades två olika shaders: en glas-shader och en vattenshader.

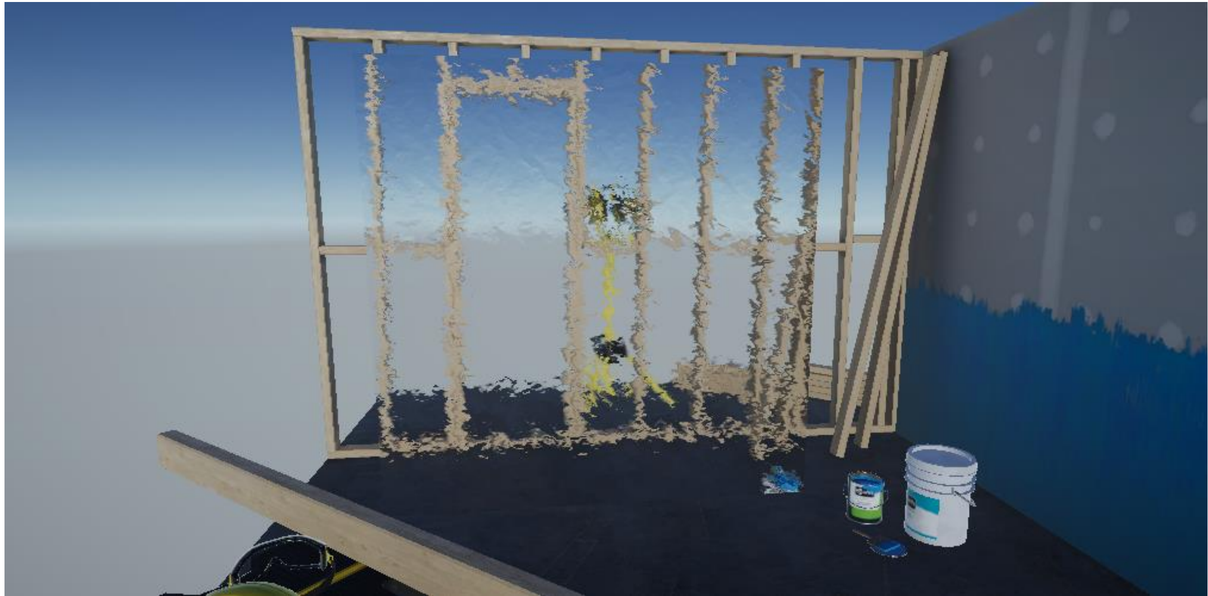
För att säkerställa en rättvis utvärdering av prestandan användes en exempelscen från Unity skapad för den aktuella renderingspipelinen. Testet upprepades fyra gånger för att säkerställa att resultaten var tillförlitliga och inte påverkades av enskilda avvikelser.

5.1.1 Resultat av glas-shader

Båda glas-shaders har testats och validerats för att säkerställa deras korrekta funktionalitet enligt tidigare definierade specifikationer. Den implementerade ljusrefraktionen har visat sig fungera som avsett, där objekt i bakgrunden förskjuts beroende på glasets brytningsegenskaper. Dessutom har en *normal map* använts för att simulera skador och ojämnheter i glaset, vilket ytterligare påverkar refraktionen och skapar en mer realistisk visuell effekt. Figur 11 och figur 12 visar hur refraktionen och *normalmappens* påverkan uttrycks i den slutgiltiga renderingseffekten.

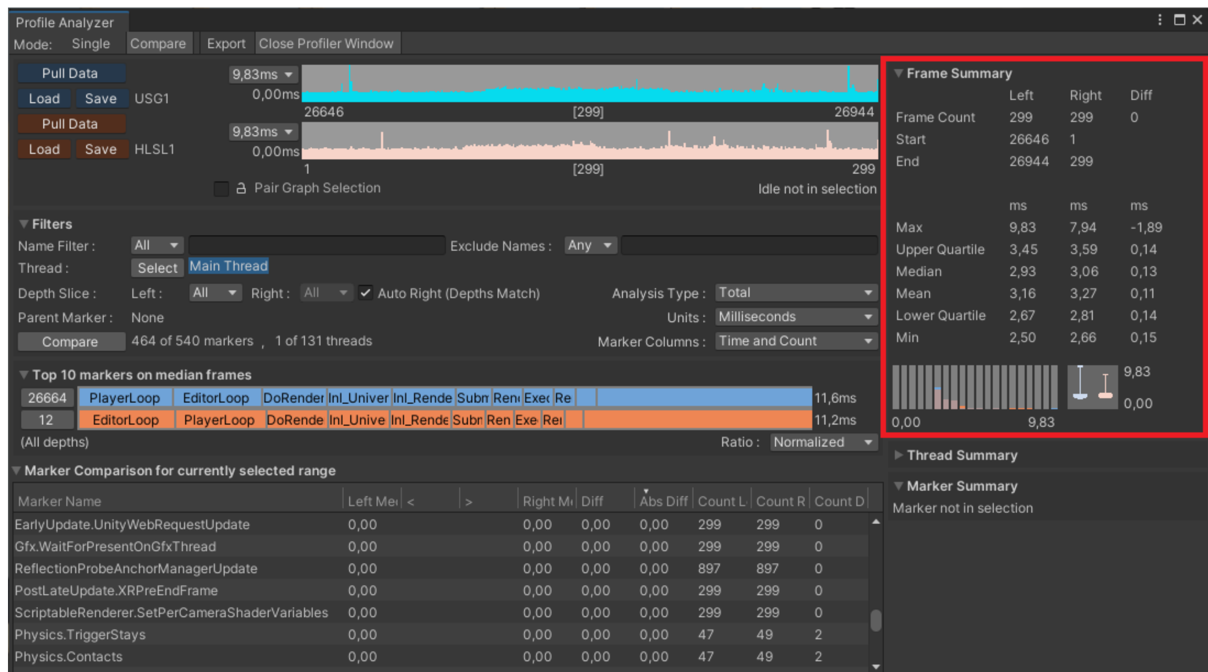


Figur 11 Glas-shadern gjord i USG



Figur 12 Glas-shadern gjord i HLSL

Testningen av glas-shaderns prestanda genomfördes enligt den metodik som tidigare beskrivits i arbetet. För att säkerställa en rättvis utvärdering av prestandan användes en exempelscen från Unity skapad för den aktuella pipelinen, vilket möjliggör en realistisk testmiljö som speglar en faktisk användning av shadern. Testet upprepades fyra gånger för att säkerställa att resultaten, som presenteras i figur 13 och tabell 1, inte är missvisande. I figur 13 ser vi spridningsmåtten till höger under rubriken *Frame Summary*, där *Left* är USG-versionen och *Right* är HLSL-versionen. Detta möjliggör en pålitlig bedömning av shaderns prestanda och ger en relevant grund för de slutsatser som dras i studien.



Figur 13 Bild från Profile Analyzer där båda glas-shadersens beräkningstid i scenen visas.

Tabell 1 Spridningsmåtten för beräkningstiden (i millisekunder) för HLSL- och USG-versionerna av glas-shadern.

	USG	HLSL
Max-värde	9,83	7,94
Övre kvartil	3,45	3,59
Median	2,93	3,06
Medelvärde	3,16	3,27
Nedre kvartil	2,67	2,81
Min-värde	2,50	2,66

För att avgöra om det fanns en statistisk signifikant skillnad i beräkningstid mellan HLSL- och USG-versionerna av glas-shadern genomfördes ett oberoende t-test. T-testet jämförde de två gruppernas medelvärden och tog hänsyn till deras standardavvikelse och antal observationer. Standardavvikelsen uppskattades utifrån kvartilavståndet. Figur 14 visar den formel som användes för att beräkna t-värdet.

$$t = \frac{M_1 - M_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}}$$

Figur 14 Formeln för t-test

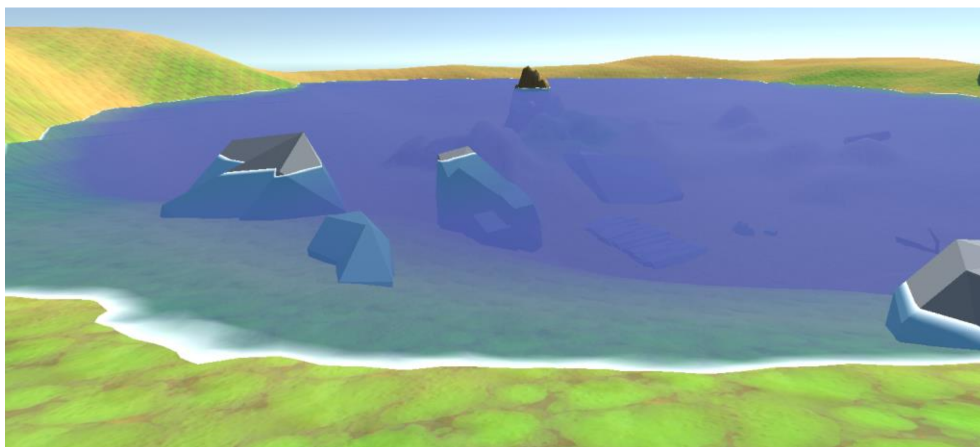
Formeln som användes för t-testet definieras enligt följande:

- M_1, M_2 = Medelvärden för de två grupperna (USG och HLSL)
- s_1, s_2 = Standardavvikelser för varje grupp
- n_1, n_2 = Antal observationer i varje grupp

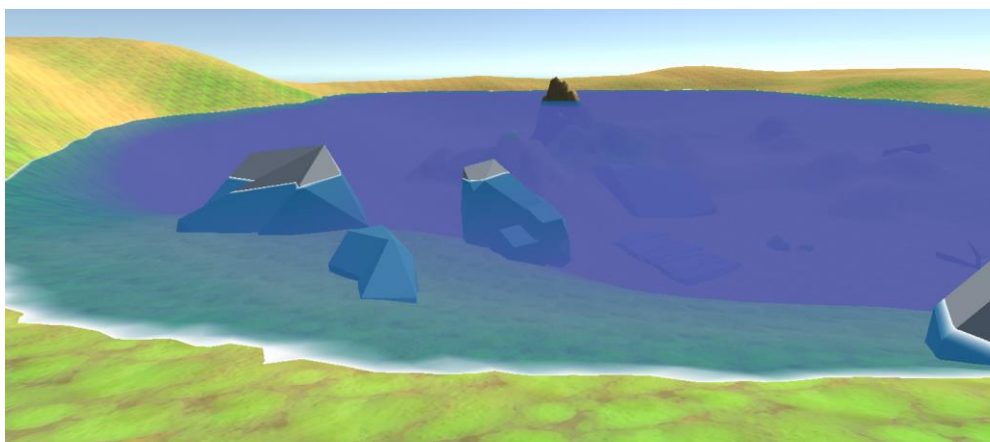
Efter att ha applicerat de uppmätta värdena resulterade beräkningen i ett t-värde på -2,33. Med hjälp av en t-fördelningstabell beräknades p-värdet till 0,020. Då p-värdet är mindre än den valda signifikansnivån 0,05, kan nollhypotesen förkastas. Detta innebär att det finns en statistiskt signifikant skillnad i prestanda mellan USG- och HLSL-versionen av glas-shadern.

5.1.2 Resultat av vattenshader

Båda vattenshaders har testats och validerats för att säkerställa att de fungerar som förväntat enligt de specifikationer som satts upp, utöver rörliga *normal maps*. Efter en jämförelse ansågs de uppvisa liknande visuella effekter, där skillnaderna var svåra att urskilja med blotta ögat. Båda shaders hanterar djupet och färghantering korrekt i kombination med de fysiska vågorna som kan ses i figur 15 och figur 16.



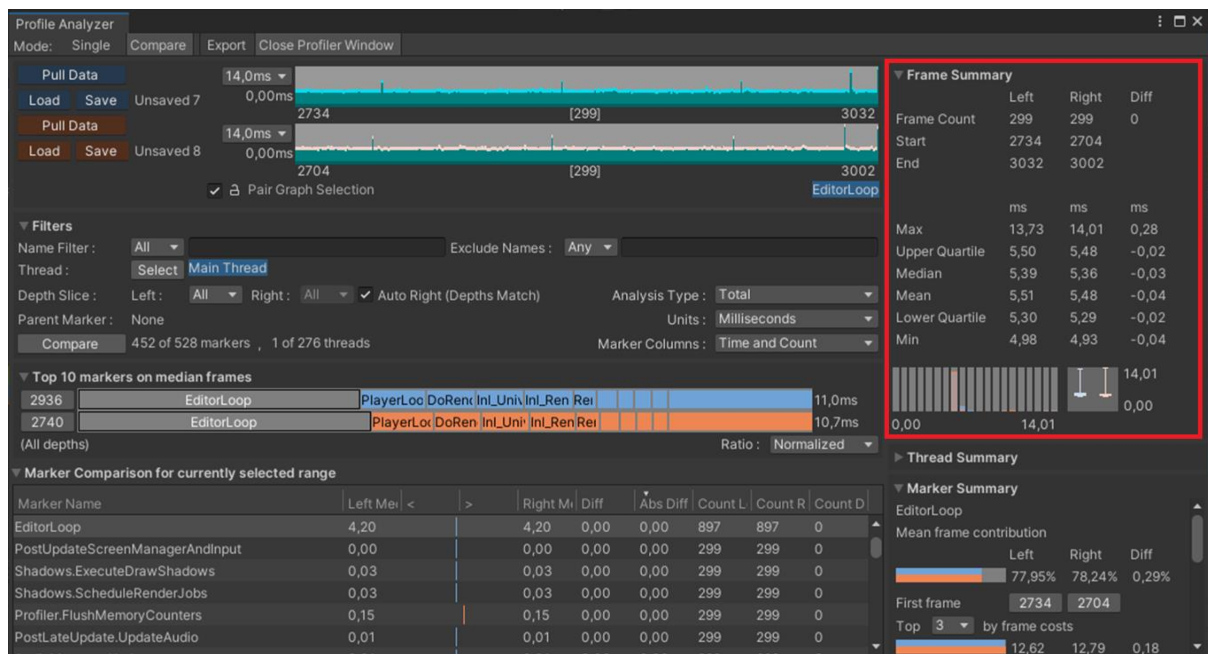
Figur 15 Vattenshadern gjord i USG



Figur 16 Vattenshadern gjord i HLSL

För att utvärdera prestandan hos de två implementationerna av vattenshadern genomfördes prestandamätningar enligt den metodik som beskrivits tidigare i rapporten. För att säkerställa en rättvis jämförelse användes testscen, skapad för att efterlikna en praktisk användning i en vanlig renderingsmiljö. Testet utfördes flera gånger för att säkerställa att resultatet som visas till höger i figur 17 och i tabell 2 inte är missvisande. I figur 17 under rubriken *Frame Summary* visas spridningsmåtten, där *Left* är USG-versionen och *Right* är HLSL-versionen.

För att undersöka skillnaden i prestanda mellan de två implementationerna av vattenshadern beräknades t-värdet enligt formeln i figur 14. De observerade värdena för medelvärde och standardavvikelse tillsammans med antalet observationer användes i beräkningen, vilket resulterade i ett t-värde på 2,54. Utifrån detta t-värde och en t-fördelningstabell beräknades p-värdet till 0,011. Eftersom p-värdet är mindre än den valda signifikansnivån 0,05, kan nollhypotesen förkastas. Detta innebär att det finns en statistiskt signifikant skillnad i prestanda mellan USG- och HLSL-versionen av vattenshadern.



Figur 17 Bild från Profile Analyzer där båda vatten-shadersens beräkningstid i scenen visas.

Tabell 2 Spridningsmåtten för beräkningstiden (i millisekunder) för HLSL-och USG-versionerna av vattenshadern

	USG	HLSL
Max-värde	13,73	14,01
Övre kvartil	5,50	5,48
Median	5,39	5,36
Medelvärde	5,51	5,48
Nedre kvartil	5,30	5,29
Min-värde	4,98	4,93

5.2 Analys

Genom att analysera resultaten från föregående kapitel kan slutsatser dras om prestandaskillnader mellan Unity Shader Graph (USG) och High-Level Shader Language

(HLSL). Analysen fokuserar på prestandamätningarna, statistisk signifikans och hur resultaten förhåller sig till tidigare forskning, särskilt de optimeringsprinciper som Crawford och O'Boyle (2018) presenterar.

5.2.1 Glas-shader

Resultaten visar att HLSL-versionen av glasshadern inte nödvändigtvis presterade bättre än USG-versionen, trots att manuella optimeringar lättare kunde göras. Medelvärdet för renderingstiden i HLSL var 3,27 millisekunder jämfört med 3,16 millisekunder för USG, en skillnad på 0,11 millisekunder (≈3,4%). Detta visar att USG presterade något bättre.

En möjlig förklaring kan vara att HLSL-versionen inte kunde optimeras i samma utsträckning som förväntat, eller att USG automatiskt applicerar optimeringar vid kompilering, vilket minskar behovet av manuell optimering. Crawford och O'Boyle (2018) visade att manuella optimeringar i shaders i genomsnitt kan förbättra prestandan med 4%, men i detta fall tyder resultaten på att de genomförda optimeringarna, såsom att byta ut divisioner mot multiplikationer och att använda half i stället för float, hade en begränsad påverkan.

Utöver detta kan även mänskliga faktorer ha spelat en roll i resultaten. Eftersom implementationen av HLSL-versionen krävde en manuell översättning från USG, finns en möjlighet att vissa delar av koden kunde ha optimerats ytterligare. Dessutom skiljer sig metoden för att hämta kamerans färgbuffert mellan USG och HLSL, vilket kan ha påverkat prestandan på sätt som inte fullt ut har analyserats i denna studie.

5.2.2 Vattenshader

Till skillnad från glasshadern uppvisade vattenshadern en minimal skillnad i prestanda mellan USG och HLSL. Medelvärdet för renderingstiden i USG var 5,51 millisekunder, medan HLSL låg på 5,48 millisekunder, en skillnad på endast 0,03 millisekunder (≈ 0,5%). Trots denna lilla skillnad var t-testet signifikant, med ett t-värde på 2,54 och ett p-värde på 0,011. Eftersom $p < 0,05$, kan nollhypotesen förkastas, vilket innebär att det finns en statistiskt signifikant skillnad mellan de två shaderimplementationerna.

Det är dock viktigt att särskilja statistisk signifikans från praktisk signifikans. Trots att skillnaden i prestanda var signifikant enligt t-testet, är den faktiska prestandaskillnaden försumbar i praktiken. En skillnad på 0,5% är så pass liten att den inte skulle påverka den övergripande prestandan i en verklig spelmiljö.

En möjlig förklaring till denna minimala skillnad är att vattenshadern redan var relativt optimerad i sin grundstruktur, vilket gjorde att de tillämpade optimeringarna hade begränsad påverkan. Crawford och O'Boyles (2018) studie visar att optimeringar i genomsnitt kan förbättra prestandan med 4%, men i detta fall var effekten betydligt mindre, vilket tyder på att shaderns struktur redan var effektiv.

Sammanfattningsvis visar analysen att valet mellan USG och HLSL inte hade någon avgörande inverkan på vattenshaderns prestanda, då skillnaden var statistiskt signifikant men praktiskt obetydlig. Detta tyder på att för denna typ av shader kan båda implementationssätten ge likvärdiga resultat.

5.3 Slutsatser

Resultaten visade små prestandaskillnader, men dessa varierade beroende på shader. För glas-shadern presterade USG-versionen avsevärt bättre än HLSL-versionen, medan vattenshadern endast uppvisade en marginell skillnad (≈0,5%), där HLSL presterade något bättre. Detta indikerar att prestandaskillnaderna mellan de två metoderna inte är generella utan beror på shaderspecifika faktorer.

Tidigare forskning har visat att manuella optimeringar kan förbättra prestandan med i genomsnitt 4% (Crawford & O'Boyle, 2018), men i denna studie var effekten av sådana optimeringar begränsad. Möjliga förklaringar inkluderar USGs automatiska optimeringar vid kompilering och att vissa optimeringar, såsom att ersätta divisioner med multiplikationer och använda half i stället för float, inte nödvändigtvis gav märkbara prestandaförbättringar i detta sammanhang.

Utöver optimeringsfaktorer kan även implementationen i sig ha påverkat resultaten. Eftersom HLSL-shadern delvis krävde en manuell översättning från USG finns en möjlighet att vissa delar av koden kunde ha optimerats ytterligare. Metoden för att hämta kamerans färgbuffert skiljer sig dessutom mellan USG och HLSL, vilket kan ha påverkat prestandan.

Sammanfattningsvis konstaterades att ingen av de två implementationsmetoderna konsekvent presterade bättre än den andra, utan att resultaten beror på den specifika shadern och dess beräkningskrav. En vidare diskussion kring möjliga faktorer som kan ha påverkat utfallet samt konsekvenserna av dessa resultat presenteras i kapitel 6.2.

6 Sammanfattning och diskussion

I detta kapitel presenteras en sammanfattning av studiens resultat samt en diskussion kring de huvudsakliga slutsatserna och deras betydelse. Vidare analyserades begränsningar i metoden och potentiella felkällor i studien. Kapitlet avslutas med en reflektion över samhälleliga och etiska aspekter samt förslag på framtida forskning inom området.

6.1 Sammanfattning

Denna studie undersöker skillnaden i prestanda mellan shaders gjorda i *Unity Shader Graph* (USG) och *High-Level Shader Language* (HLSL). De shaders som utvecklats med HLSL har aktivt försökts optimeras enligt de tekniker Crawford och O'Boyle (2018) framhäver. USG är ett visuellt verktyg som automatiskt kompilerar shaders till HLSL. Eftersom valet av shaderspråk kan ha inverkan på prestanda, formulerades följande frågeställning för att undersöka eventuella prestandaproblem:

"Finns det en skillnad i prestanda mellan en shader implementerad i HLSL och en shader implementerad i USG i Unity?"

För att besvara frågeställningen genomfördes prestandatester på shaders som utvecklades med USG och motsvarande shaders som skapades med HLSL. Under utvecklingsprocessen stöttes problem på med HLSL-shadersen, där dokumentationen visade sig vara bristfällig. Dessutom fungerade en av HLSL-shadersen inte som förväntat visuellt.

Efter genomförda tester och analys av resultaten kunde ingen konkret prestandaförbättring i någon riktning observeras vid jämförelsen mellan shaders utvecklade med HLSL och de som skapades med USG. Resultaten indikerade att valet av shaderspråk, det vill säga HLSL eller USG, inte nödvändigtvis påverkar prestandan signifikant.

Denna undersökning ger en inblick i de två shaderverktygens prestandaegenskaper och deras användbarhet inom Unity. Slutsatsen bekräftar att prestandaaspekterna av shaderskapande inte är entydiga och att användningen av ett visuellt verktyg som USG inte nödvändigtvis kompromissar med prestanda jämfört med shaders som skapats med det mer traditionella HLSL.

6.2 Diskussion

Resultaten av denna studie visar att prestandaskillnaderna mellan USG och HLSL är små och varierar beroende på shader. Detta motsäger en vanlig uppfattning om att HLSL alltid skulle vara mer optimerat än USG. I stället tyder resultaten på att prestandan påverkas av en kombination av automatiserade optimeringar i USG, shaderspecifika implementeringar samt skillnader i hur olika beräkningar utförs.

En viktig faktor som kan ha påverkat resultaten är hur vissa beräkningar genomförs internt i USG jämfört med HLSL. Till exempel används i USG inbyggda funktioner för att hämta färgbufferten från kameran, medan HLSL-versionen krävde manuella anrop till funktioner som *SAMPLE_TEXTURE2D* tillsammans med *CameraOpaqueTexture*. Även om båda metoderna uppnår samma visuella resultat, kan de skilja sig i prestanda. Eftersom USGs inbyggda funktioner hanterar vissa beräkningar automatiskt, är det inte alltid tydligt exakt

hur de påverkar prestandan jämfört med en manuell HLSL-implementation. En mer ingående analys av dessa funktioner skulle kunna ge en bättre förståelse för deras optimeringspotential

En annan aspekt att beakta är hur USG hanterar automatiserade optimeringar vid kompilering. USG kompilerar shaders till HLSL-kod, men den exakta optimeringsprocessen som Unitys Shader Graph tillämpar är inte alltid synlig. Det är därför möjligt att vissa optimeringar automatiskt applicerades i USG-versionen men saknades i den manuella HLSL-implementationen, vilket kan ha bidragit till att USG-versionen av glasshadern presterade signifikant bättre än HLSL-versionen.

Vidare kan den mänskliga faktorn ha påverkat resultaten, särskilt eftersom HLSL-versionerna delvis krävde en manuell översättning av USG-shaders. Små implementationstekniska skillnader kan ha påverkat prestandan, även om kodstrukturen och optimeringarna i möjligaste mån följde samma principer. Det här är en faktor som kan vara svår att helt undvika i den här typen av studier, men det är ändå viktigt att ha den i åtanke när man tolkar resultaten.

En metodologisk begränsning i denna studie är att HLSL-varianten av glas-shadern i huvudsak utformades genom att utgå från den gjord i USG. Detta kan innebära att HLSL-implementationen i praktiken blir en översättning snarare än en oberoende framtagning. Denna process riskerar att leda till att båda shaders får en liknande struktur och därför likartad prestanda, vilket kan försvåra en rättvis jämförelse. Dessutom innebär detta att mänskliga faktorer i översättningen kan påverka HLSL-versionens prestanda på ett vis som inte förekommit om de utvecklats fullständigt oberoende av varandra.

Ytterligare en aspekt som bör beaktas är skillnader i arbetsflödet mellan USG och HLSL. En av de största utmaningarna med att arbeta direkt i HLSL är bristen på välutvecklade kodredigerare som är anpassade för shaderskapande. Till skillnad från många andra programmeringsspråk saknas ofta avancerade utvecklingsmiljöer (IDEs) som erbjuder stöd för exempelvis autokomplettering och felsökning för shaders. Detta kan göra arbetet mer tidskrävande och kräva en djupare förståelse för grafikprogrammering. I kontrast till detta erbjuder USG ett visuellt och mer tillgängligt arbetsflöde, där utvecklare kan se realtidsförändringar i shaderns funktionalitet och utseende utan att behöva skriva kod.

Trots att USG kan vara enklare att använda, finns det vissa begränsningar i vad som kan åstadkommas med Shader Graph. En betydande nackdel är avsaknaden av avancerade ljushanteringsfunktioner, vilket innebär att vissa beräkningar för ljusinteraktioner måste skrivas i HLSL och inkluderas som anpassade funktioner. Detta kan göra att vissa typer av mer komplexa shaders fortfarande kräver en HLSL-baserad implementation.

Enligt Crawford och O'Boyle (2018) kan manuella optimeringar förbättra prestandan med i genomsnitt 4%, men i denna studie var effekten begränsad. Det är möjligt att vissa av de genomförda optimeringarna, såsom att byta divisioner mot multiplikationer eller använda half i stället för float, endast hade en marginell effekt på den specifika hårdvaran och de renderingstekniker som användes.

En annan begränsning med den här studiens metod är att den beräknade prestandaskillnaden i procent baseras på renderingstiden för hela scenen, inte enbart på varje individuell shader. Detta innebär att skillnaden i renderingstid som anges kan vara mindre än den faktiska

skillnaden, om resultatet för varje shaders renderingstid isolerades. Detta är en aspekt som bör beaktas när resultaten tolkas.

Sammanfattningsvis pekar resultaten på att USG kan vara ett fullgott alternativ för många shaders, särskilt där prestandakritiska optimeringar inte är avgörande. HLSL ger större kontroll och flexibilitet, men kräver mer teknisk kunskap, och i vissa fall kan prestandaskillnaderna vara försumbara. Vidare diskussion om potentiella forskningsområden och möjliga utökningar av denna studie behandlas i kapitel 6.4.

6.3 Samhälleliga och etiska aspekter

Under hela denna studie har det varit viktigt att upprätthålla en hög forskningsetisk standard. Inga data har förvanskats för att passa en viss agenda, och resultaten har analyserats objektivt för att säkerställa studiens trovärdighet. Eftersom ingen datainsamling involverade deltagare, har heller inga etiska risker kopplade till personlig integritet eller informerat samtycke förekommit.

Studien kan ha en positiv samhällspåverkan genom att belysa vikten av mjukvaruoptimering. Effektivare shaders kan minska energiförbrukningen, vilket är särskilt relevant för mobila enheter där batteritid är en kritisk faktor. Mobilspel och andra grafikintensiva applikationer ställer allt högre krav på CPU- och GPU-resurser, vilket kan öka enhetens energiförbrukning (Choi et al., 2022). Optimering kan därför bidra till längre batteritid och minskad miljöpåverkan.

Utöver energieffektivitet kan optimerad mjukvara också förbättra tillgängligheten. Genom att minska prestandakrav kan även äldre och billigare enheter hantera mer avancerad grafik, vilket gör tekniken mer tillgänglig för fler användare. Detta kan både minska ekonomiska barriärer och förlänga livslängden på befintlig hårdvara.

Sammanfattningsvis kan optimerad mjukvara gynna både användare och miljön genom effektivare resursutnyttjande. Att minska energiförbrukning och hårdvarukrav är viktiga steg mot en mer hållbar och tillgänglig digital framtid.

6.4 Framtida arbete

Denna studie har identifierat flera områden där framtida forskning kan fördjupa analysen och förbättra förståelsen för prestandaskillnader mellan shaders implementerade i USG och HLSL.

En viktig aspekt är förbättrad datainsamling. Den nuvarande metoden hade vissa begränsningar, bland annat att alla 299 uppmätta renderingstider *Profile Analyzern* inte kunde exporteras till kalkylprogram för vidare bearbetning. Detta påverkade möjligheten att utföra mer detaljerade analyser och beräkna vissa statistiska mått med högre noggrannhet. Genom att använda mer avancerade profileringsverktyg eller bättre mätmetoder kan framtida studier säkerställa en mer exakt och pålitlig analys av renderingstid och prestandapåverkan.

Ett bredare urval av shaders hade också kunnat ge en mer generaliserbar bild av prestandaskillnaderna. I denna studie testades en glas-shader och en vatten-shader, men andra typer av shaders, såsom vegetation, hud eller komplexa ljuseffekter, hade kunnat

inkluderas. Att jämföra fler shaders med varierande komplexitet hade kunnat ge en mer representativ bild av hur valet mellan USG och HLSL påverkar prestanda i olika scenarion.

Vidare skulle en djupare analys av optimeringstekniker kunna vara en viktig aspekt för framtida forskning. Denna studie fokuserade på grundläggande optimeringar såsom att ersätta *float* med *half* och byta divisioner mot multiplikationer. Genom att utforska fler optimeringsstrategier, exempelvis minimering av *draw calls*, bättre hantering av texturhämtningar eller förbättrad minnesanvändning, kan en mer omfattande bild av optimeringarnas inverkan skapas. Det hade även varit intressant att undersöka hur olika hårdvaruplattformar påverkar prestandaskillnaderna mellan HLSL och USG.

Den mest intressanta aspekten för framtida forskning vore att undersöka USGs automatiserade optimeringar mer ingående. Eftersom USG kompilerar till HLSL i bakgrunden utan full insyn i de optimeringar som tillämpas, är det svårt att avgöra i vilken utsträckning prestandan förbättras eller försämras jämfört med en manuellt skriven HLSL-shader. En djupare analys av den genererade koden från USG och dess skillnader gentemot en manuellt optimerad HLSL-implementation skulle kunna ge en tydligare bild av hur effektivt USG hanterar prestandaförbättringar och resursanvändning. Särskilt intressant vore att undersöka hur USG hanterar *custom functions* och om dessa genomgår någon form av optimering eller om de påverkar prestandan negativt jämfört med motsvarande kod skriven direkt i HLSL.

Referenser

Bradley, J. (2018). *The Importance of Shader Optimization for Game Developers*. Pluralsight. https://www.pluralsight.com/blog/film-games/importance-shader-optimization-game-developers [hämtad:2023-02-03].

Cervantes, J. (2025). *Refresh Rates of Display Screens,* SciTechMag. https://scitechmag.com/2025/01/refresh-rates-of-display-screens/ [hämtad: 2025-03-31]

Chatzi, A.V. (2025). Understanding the independent samples t test in nursing research, *British Journal of Nursing*, 34(1), ss. 56–62. doi:10.12968/bjon.2024.0133.

Choi, Y. et al. (2022). Optimizing Energy Consumption of Mobile Games, *IEEE Transactions on Mobile Computing, Mobile Computing, IEEE Transactions on, IEEE Trans. on Mobile Comput*, 21(10), ss. 3744–3756. doi:10.1109/TMC.2021.3058381.

Claypool, K.T. and Claypool, M. (2007). On frame rate and player performance in first person shooter games, *Multimedia Systems*, 13(1), ss. 3–17. doi:10.1007/s00530-007-0081-1.

Crawford, L. & O'Boyle, M. (2018). A Cross-platform Evaluation of Graphics Shader Compiler Optimization. IEEE International Symposium on Performance Analysis of Systems and Software, ss. 219-228.

Dataspelsbranschen (2024). *Spelutvecklarindex 2024*, Swedish Games Industry. https://dataspelsbranschen.se/spelutvecklarindex [hämtad: 2025-03-30]

Dillet, R. (2018). Unity CEO says half of all games are built on Unity. Disrupt SF 2018.

Holmér, F. (2021). *Shader Basics, Blending & Textures • Shaders for Game Devs [Part 1]* [Video]. https://www.youtube.com/watch?v=kfM-yu0iQBk&ab_channel=FreyaHolm%C3%A9r [hämtad: 2023-03-02]

Holmér, F. (2021). *Healthbars, SDFs & Lighting • Shaders for Game Devs [Part 2]* [Video]. https://www.youtube.com/watch?app=desktop&v=mL8U8tIiRRg&ab_channel=FreyaHolm%C3%A9r [hämtad 2023-03-02]

Interactive Software Federation of Europe & European Games Developer Federation (ISFE & EGDF) (2021). KEY FACTS 2020. Interactive Software Federation of Europe. www.isfe.eu/wp-content/uploads/2021/10/2021-ISFE-EGDF-Key-Facts-European-video-games-sector-FINAL.pdf [hämtad: 2023-02-12]

Lam, A. (2016). *Optimize Shaders for Game Development*. Game Development Envato Tuts+. tutsplus.com/tutorials/optimizing-shaders-for-game-development--cms-27741 [hämtad:2023-03-06]

PabloMakes (2021). *Refraction ShaderGraph tutorial for Unity* [Video]. https://www.youtube.com/watch?v=tIW2zM6ed80&ab_channel=PabloMakes [hämtad: 2023-02-15]

Shadertoy: afl_ext (2017). *Very fast procedural ocean.* https://www.shadertoy.com/view/MdXyzX [hämtad 2023-03-15]

Sharp Coder Blog (u.å.) *Brief History of Unity*. https://sv.sharpcoderblog.com/blog/brief-history-of-unity [hämtad: 2025-04-03]

Su, M., Guo, R., Wang, H., Wang, S. & Niu, P. (2017). View Frustum Culling Algorithm Based on Optimized Scene Management Structure. *2017 IEEE International Conference on Information and Automation (ICIA)*. Macau SAR, China, July 2017, ss. 838-842.

Unity Documentation (u.å. a). *Optimizing shader runtime performance*. Unity Technologies. https://docs.unity3d.com/Manual/SL-ShaderPerformance.html [hämtad: 2023-02-13]

Unity Documentation (u.å. b). *Optimizing draw calls*. Unity Technologies. https://docs.unity3d.com/6000.0/Documentation/Manual/optimizing-draw-calls.html [hämtad 2025-04-01]

Unity Technologies. (2005). Unity Engine [Mjukvara: PC]. San Francisco, Kalifornien. unity.com

Appendix A - Glas-shader i HLSL

```
Shader "Custom/GlassRefractionWithNormal"
{
  Properties
  {
    _NormalMap ("Normal Map", 2D) = "bump" {}
    _RefractionStrength ("Refraction Strength", Range(0, 0.5)) = 0.02
    _NormalMapRefractionStrength ("Normal Map Refraction Strength", Range(-0.5, 0.5))
= 0.02
    _Tint ("Tint Color", Color) = (1,1,1,1)
  }

  SubShader
  {
    Tags { "RenderType"="Transparent" "Queue"="Transparent" }
    LOD 100

    Pass
    {
      Name "GlassRefractionWithNormal"
      Tags { "LightMode"="UniversalForward" }
      Blend SrcAlpha OneMinusSrcAlpha
      ZWrite Off

      HLSLPROGRAM
      #pragma vertex vert
      #pragma fragment frag
      #include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Core.hlsl"
```

```
TEXTURE2D(_CameraOpaqueTexture);
SAMPLER(sampler_CameraOpaqueTexture);

TEXTURE2D(_NormalMap);
SAMPLER(sampler_NormalMap);

CBUFFER_START(UnityPerMaterial)

half _RefractionStrength;
half _NormalMapRefractionStrength;
half4 _Tint;

CBUFFER_END

struct appdata
{
   float4 position : POSITION;
   float2 uv : TEXCOORD0;
   float3 normal : NORMAL;
};

struct v2f
{
   float4 position : SV_POSITION;
   float2 screenUV : TEXCOORD1;
   float3 worldNormal : TEXCOORD2;
   float2 uv : TEXCOORD0;
```

```
        };

        v2f vert(appdata v)
        {
          v2f o;
          o.position = TransformObjectToHClip(v.position.xyz);
          o.screenUV = float2(o.position.x / o.position.w * 0.5 + 0.5,
                      1.0 - (o.position.y / o.position.w * 0.5 + 0.5));
          o.worldNormal = TransformObjectToWorldNormal(v.normal);
          o.uv = v.uv;
          return o;
        }

        half4 frag(v2f i) : SV_Target
        {
          half3 normalMap = SAMPLE_TEXTURE2D(_NormalMap, sampler_NormalMap, i.uv).rgb;

          normalMap = normalize(normalMap * 2.0 - 1.0);

          half3 modifiedNormal = normalize(i.worldNormal + normalMap * _NormalMapRefractionStrength);

          half2 normalOffset = modifiedNormal.xy * _RefractionStrength;
          half2 refractedUV = i.screenUV + normalOffset;

          half4 sceneColor = SAMPLE_TEXTURE2D(_CameraOpaqueTexture, sampler_CameraOpaqueTexture, refractedUV);
```

```
            return sceneColor * _Tint;
        }

        ENDHLSL
    }
  }
}
```

Appendix B - Vattenshader i HLSL

```
Shader "Unlit/NewUnlitShader"
{
  Properties
  {
    _Depth ("Depth", Float) = 1.0
    _FoamDepth ("FoamDepth", Float ) = 1.0
    _Strength ("Strength", Float) = 1.0

    _DeepWaterColor ("DeepWaterColor", Color) = (1,1,1,1)
    _ShallowWaterColor ("ShallowWaterColor", Color) = (1,1,1,1)

    _FoamWaterColor("Foam Color", Color) = (1,1,1,1)

    _Displacement ("Displacement", Float) = 0.001

  }
  SubShader
  {
    Tags {"Queue"="Transparent" "RenderType"="Transparent" }

    ZWrite Off
    Blend SrcAlpha OneMinusSrcAlpha

    Pass
    {
      CGPROGRAM
      #pragma vertex vert
```

```
#pragma fragment frag

#include "UnityCG.cginc"

half _Depth;
half _FoamDepth;
half _Strength;
half _Displacement;
half4 _DeepWaterColor;
half4 _ShallowWaterColor;
half4 _FoamWaterColor;

sampler2D _CameraDepthTexture;

struct MeshData
{
    float4 vertex : POSITION;
    float3 normals : NORMAL;

    float4 uv0 : TEXCOORD0;
};

struct Interpolators
{
    float4 vertex : SV_POSITION; //clip space position
    float3 normal : TEXCOORD0;
    float2 uv : TEXCOORD1;
    float2 normalUV : TEXCOORD4;
```

```
    float2 secondNormalUV : TEXCOORD3;

    float4 screenPosition : TEXCOORD2;
};




float2 unity_gradientNoise_dir(float2 p)
{
    p = p % 289;
    float x = (34 * p.x + 1) * p.x % 289 + p.y;
    x = (34 * x + 1) * x % 289;
    x = frac(x / 41) * 2 - 1;
    return normalize(float2(x - floor(x + 0.5), abs(x) - 0.5));
}

float unity_gradientNoise(float2 p)
{
    float2 ip = floor(p);
    float2 fp = frac(p);
    float d00 = dot(unity_gradientNoise_dir(ip), fp);
    float d01 = dot(unity_gradientNoise_dir(ip + float2(0, 1)), fp - float2(0, 1));
    float d10 = dot(unity_gradientNoise_dir(ip + float2(1, 0)), fp - float2(1, 0));
    float d11 = dot(unity_gradientNoise_dir(ip + float2(1, 1)), fp - float2(1, 1));
    fp = fp * fp * fp * (fp * (fp * 6 - 15) + 10);
    return lerp(lerp(d00, d01, fp.y), lerp(d10, d11, fp.y), fp.x);
}
```

```
float Unity_GradientNoise_float(float2 UV, float Scale)
{
    return unity_gradientNoise(UV * Scale) + 0.5;
}

float2 Unity_TilingAndOffset_float(float2 UV, float2 Tiling, float2 Offset)
{
    return UV * Tiling + Offset;
}



Interpolators vert (MeshData v)
{
    Interpolators o;

    float2 os = Unity_TilingAndOffset_float(v.uv0, float2(1,1), _Time.y * 0.01);
    float vWave = (Unity_GradientNoise_float(os, 20));

    _Displacement *= 0.0001;
    v.vertex.z = vWave * _Displacement;

    o.vertex = UnityObjectToClipPos(v.vertex);

    o.screenPosition = ComputeScreenPos(o.vertex);

    o.normal = v.normals;
```

```
      return o;
    }

    fixed4 frag (Interpolators i) : SV_Target
    {
      float4 waterColor;

      //Depth color

      float existingDepth01 = tex2Dproj(_CameraDepthTexture, UNITY_PROJ_COORD(i.screenPosition)).r;
      float existingDepthLinear = LinearEyeDepth(existingDepth01);

      float depthDifference = existingDepthLinear - i.screenPosition.w;

      if (depthDifference <= _FoamDepth )
      {
        float waterDepthDifference02 = saturate(depthDifference / _FoamDepth);
        waterColor = lerp(_FoamWaterColor, _ShallowWaterColor, waterDepthDifference02);

      }
      else
      {
        float waterDepthDifference01 = saturate(depthDifference / _Depth);
        waterColor = lerp(_ShallowWaterColor, _DeepWaterColor, waterDepthDifference01);
```

```
            }

            return waterColor;
        }
        ENDCG
    }
  }
}
```

X