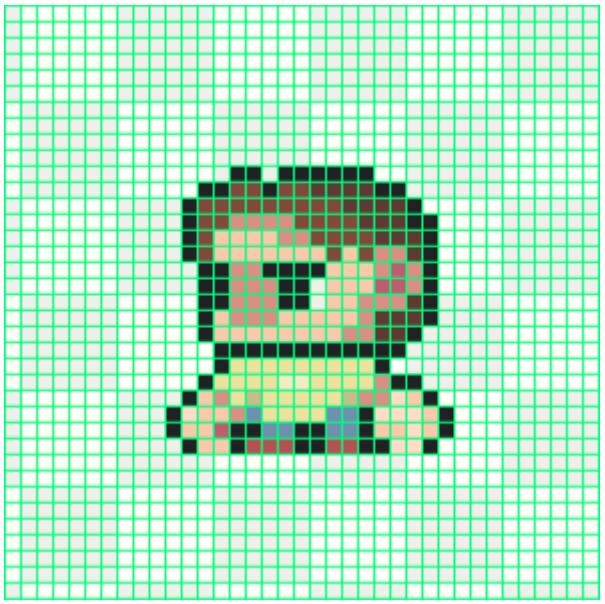
Katrineholms Tekniska College
Teknikprogrammet
Gymnasiearbete 100p
Läsår 2023-2024

Retrodatorgrafik möter moderna system

Utvecklingen av ett system som återupplivar datorns tidiga grafik



Författare: Viggo Vusir TE21
Handledare: Mats Carlsson Frölund

Abstract

The development of computer graphics is constantly evolving. Homages to the first computer-generated visuals are today seen through the art style “pixel art”. Pixel art is primarily used in game development and consists of graphics made out of distinct squares with contrasting colors. When rendering pixel art, which is inherently low resolution, on high resolution screens, all kinds of problems come up. Examples of such problems are distorted pixels, choppy movement and blurry graphics. Some major game engines solve this problem by taking shortcuts such as scaling up pixel art graphics to fit the desired resolution. This only solves some problems and in some cases makes other issues worse. To achieve crisp low-resolution pixel art on the whole spectrum of resolutions, new systems must be made. Through research and experimentation, such a system has been made. The system allows for low-resolution pixel art to remain both crisp and highly qualitative regardless of resolution. The issue with choppy movement has also been resolved by creating an algorithm that displaces the rendered graphics by a position dependent on the camera’s position. The system also enables features such as procedural pixel art, which could lead to impressive graphics. This work was done fully using C# and MonoGame but it is entirely possible to translate it over to use in any language or framework one may desire.

Innehållsförteckning

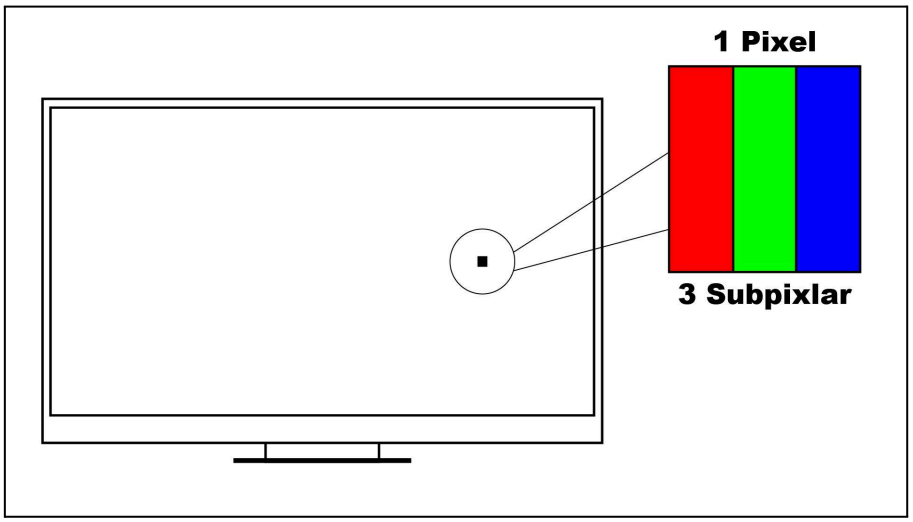
1. Inledning

Datorgrafiken utvecklas konstant och tillåter mer realistiska grafiker för varje generation av grafikkort. Trots detta används fortfarande samma estetik som de första videospelen i ett stort antal spel. Återskapandet av tidig datorgrafik har fått beteckningen “pixel art” och trots moderna system är grundprinciperna detsamma – skarpa kvadrater på små ritytor. Moderna system tillåter upplösningar flerfaldigt större än upplösningar som pixel art vanligtvis ritas på. Detta medför olika problem som detta arbete söker att lösa.

2. Bakgrund

2.1 Vad är en pixel och pixel art?

En pixel är den minsta enheten på en datorskärm och i LCD-skärmar är den uppbyggd av tre subpixlar (Se figur 1). Subpixlarna består av färgerna röd, grön och blå och tillsammans kan de producera 16777216 färger. Pixel art är en konststil och estetik för främst spel som grundar sig i att uppnå samma visuella stil som tidig datorgrafik. Att rendera i låg upplösning var det enda alternativet för datorgrafik från början till slutet av 80-talet och antalet färger var ytterst begränsat. Stilen går alltså ut på att med hjälp av dessa pixlar skapa motiv på en ofta begränsad rityta. Pixel art har blivit väldigt populärt inom indie-spelbranschen på grund av de låga krav grafiken har när det kommer till att både skapa och rendera den.

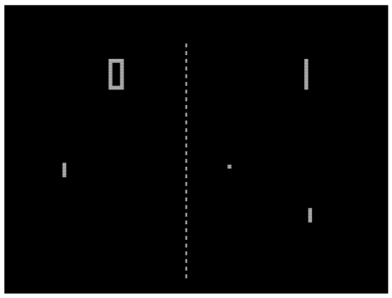


Figur 1: *Pixlar och subpixlar* (Författarens bild. 2023)

2.2 Utvecklingen av pixel art

Videospelet *Pong* var det första spelet med grafik gjord av pixlar. När spelet släpptes 1972 bestod dess grafiker av endast fyrkantiga former och var monokromt i färgerna vit och svart. Sedan dess har pixel art genomgått stora utvecklingar (Se figur 2), främst tack vare spelutvecklingen. I samband med att Nintendo släppte sin konsol *Nintendo Entertainment System* (NES) och Segas lansering av deras motsvarighet till NES, *Sega Master System*, uppstod “8-bit”-grafiken under tidigt 80-tal. Grafiken kunde nu bestå av 256 olika färger samtidigt och mer detaljerade figurer kunde användas. Upplösningen var dock fortfarande

väldigt liten och 256x224 eller 256x240 pixlar var standard. Senare på 80-talet lanserades Nintendo och Segas nya konsoler *Super Nintendo Entertainment System* (SNES), respektive *Sega Genesis*. Dessa konsoler tillät "16-bit"-grafik vilket gav möjlighet att visa 65536 unika färger samtidigt. Detta gav uppsåt till ännu mer detaljerade grafiker när det kom till visualisering med färg, upplösningen hade dock inte hunnit ikapp utan var fortfarande lika liten. I takt med att högre upplösningar och högre bitars-system under 90-talet så utvecklades inte pixel art lika mycket. Under 2000-talet fick däremot pixel art ett nyvaknande i samband med lanseringen av kraftigare system. Skillnaden där var att man inte längre skapade pixel art på grund av tekniska limitationer, utan pixel art blev istället ett estetiskt val i videospelsbranschen. Idag så ser vi alltså spel som använder den distinkta stilen enbart för att det helt enkelt är en uppskattad estetik av många.



Pong, 1972[A]

NES Super Mario Bros[B]
©Nintendo

SNES Super Mario World[C]
©Nintendo

Figur 2: *Pixelgrafikens tidiga utveckling* (Författarens bildkomposition. 2024)
A: *Pong* (Wikimedia Commons. 2006)
B: *NES Super Mario Bros* (Nintendo. 1985)
C: *SNES Super Mario World* (Eurogamer. 2012)

2.3 Vad innebär "pixel perfect"?

Tidigare pixel art-spel var renderade i en upplösning på 256x224 pixlar. Idag är standarden för skrivbordsskärmar 1920x1080 pixlar. Denna skillnad i upplösning gör att renderingen av 256x224 pixlar upplevs som väldigt liten på de stora skärmarna. Detta kan undgås genom att skala upp pixel art till att passa skärmens upplösning. Att bara skala upp pixel art medför olika problem. Bland annat blir pixlarna suddiga och pixlarna slutar att följa det distinkta rutnätet vilket leder till deformerade pixlar. Att pixlarna renderas korrekt oavsett upplösning kallas för "pixel perfect". Begreppet "pixel perfect" inom spelutveckling innebär alltså en säkerställning av att pixel art alltid förblir skarp och är korrekt justerat i programfönstret oavsett upplösning.

2.4 Existerande lösningar för högupplöst pixel art

Flera spelmotorer tillåter högupplöst pixel art genom användningen av uppskalad grafik. Vid en närmare titt kan man då se att grafikerna inte längre är justerade efter ett och samma rutnät och när grafiken roteras ser man att det egentligen inte är lågupplöst i grunden.

2.5 Terminologi

I detta arbete förekommer en del termer som ej har någon lämplig motsvarighet på svenska. Därmed skrivs vissa ord mer “försvenskat”, till exempel “The SpriteBatch’s” blir “SpriteBatchens”.

3. Syfte och frågeställning

3.1 Syfte

I detta gymnasiearbete utforskas tillvägagångssätt för att rendera “pixel perfect” grafik utan användning av färdiga sådana system. Gymnasiearbetet tar även reda på möjligheten att kunna använda sig av kamerarörelser som upplevs som mjuka trots att upplösningen egentligen är väldigt låg. Syftet med detta är att utveckla ett system som slår samman den älskvärda retro datorgrafiken tillsammans med modern teknologi. Det vill säga ett system som tillåter pixel art att behålla sin kvalitet över vilken upplösning som helst och att alla pixlar är justerade efter samma rutnät. Systemet ska även tillåta mjuka kamerarörelser oavsett hastighet. Trots att teknologin och kapaciteten för grafikrendering konstant utvecklas så har pixel art mer eller mindre stått still när det kommer till realtidsgrafik. Att rendera på låga upplösningar skapar möjligheter till att bland annat procedurellt generera pixel art och omvandla 3D-modeller till grafik som påminner om tidiga spelgrafiker. Möjligheterna är oändliga och detta arbete lägger grunden till att skapa dem alla.

3.2 Frågeställning

Hur utvecklar man ett “pixel perfect” renderingssystem i C#?

4. Metod och material

4.1 Metod

4.1.1 Informationssökning

Trots att detta är ett system jag själv skrivit och kommit på från grunden har det ändå krävts informationssökning. Detta har mest bestått av att undersöka hur liknande system fungerar. På grund av att den övervägande majoriteten av sådana system som finns tillgängliga på internet antingen är gömda bakom en betalvägg eller inte har öppen källkod har jag funnit nästintill noll information. Den lilla information jag hittat har varit på onlineforum där utvecklare diskuterat liknande ting. Dessa diskussioner har jag använt mig av som bollplank för att se om mina idéer är på rätt väg eller om jag borde utveckla systemet på ett annat, mer koncist sätt. Diskussionerna har dock inte kunnat användas som en fullständig vägledning då de ofta har haft ett helt annat programmeringsspråk i tanken.

4.1.2 Utförande

Utvecklingen av systemet skedde fortlöpande under en kortare period. Först planerades de olika stegen för att uppnå önskad effekt. Sedan skrevs pseudokod och aktivitetsdiagram skapades för att ge en bild på hur utvecklingen ungefär ska se ut. Därefter skrevs koden och olika parametrar justerades tills rätt effekt var nådd. Under utvecklingens gång användes dokumentationen för C# och MonoGame för att bidra med förståelse för programmeringsspråket och ramverket i den mån det behövdes.

4.2 Material

Det material jag använt mig av i utvecklingen av detta system är främst en kompilation av olika mjukvaror. Dessa mjukvaror har uppfyllt olika funktioner i arbetet och är alla tillgängliga att ladda ner från respektive programs webbplats. Utöver mjukvaror har jag använt mig av programmeringsspråk och ramverk, vilka båda också är helt öppna för allmänheten att installera och använda.

4.2.1 C#

C# är ett objektorienterat programmeringsspråk skapat av Microsoft. All programmering i detta arbete har skett i C#. Jag valde just C# på grund av min tidigare erfarenhet av språket och för att inte manuellt behöva sköta hanteringen av datorns minne då C# redan har ett adekvat system för att sköta detta.

4.2.2 Visual Studio 2019

Visual Studio 2019 är Microsofts IDE (Integrated Development Environment). Den tillåter utveckling av program och hemsidor i diverse programmeringsspråk. IDE:n ger även tillgång till smarta system som visar exakt var koden kör fast ifall den gör det. Visual Studio 2019 är det program jag använt till att utveckla mitt “pixel perfect”-system i.

4.2.3 MonoGame

MonoGame är en “open source”-fortsättning på Microsofts XNA-ramverk. XNA står för Xbox New Architecture och var en samling av verktyg och bibliotek för att skapa videospel för Xbox 360 och PC. XNA slutade utvecklas 2013 och ur detta avslut skapades MonoGame. MonoGame är en fortsättning på den senaste och sista versionen av XNA (4.0). MonoGame tillåter, precis som XNA, utvecklingen av videospel till olika plattformar med stöd för funktioner som 2D- och 3D-grafik, ljud, nätverk och inmatning. MonoGame är det ramverk jag använt för att snabbt och relativt enkelt kunna rendera 2D-grafik.

4.2.4 Aseprite

Aseprite är en mjukvara specifikt designad för att skapa pixel art och animeringar. Aseprite har jag använt mig till att rita de grafiker jag använt mig av.

4.2.5 Photoshop

Photoshop är en mjukvara utvecklad av Adobe i syftet med bildredigering. Jag har använt mig av Photoshop för att skapa och redigera bilder som kompletterar texten.

4.2.6 Illustrator

Illustrator är en mjukvara utvecklad av Adobe som tillåter redigering och skapande av vektorgrafiker. Detta har jag använt för att göra skarpa grafiker som komplement till texten och bilder redigerade i Photoshop.

5. Källkritik

Trots att jag inte behövt använda mig av mer akademiska källor som vetenskapliga avhandlingar och studier är källkritiken ändå viktig i allt sökande av information. Mina primära källor har varit onlineforum på grund av bristen av mer akademiska källor som rör

detta område. Onlineforum kan innehålla mycket dålig information och i vissa fall till och med medvetet dålig information. Det gör att källkritiken fortfarande är viktig och måste nyttjas väl. Eftersom mitt arbete i grunden handlar om att programmera behöver jag inte förlita mig så mycket på existerande fakta utan bara på vad jag själv kan om programmering och datorvetenskap. Därmed har källkritiken bestått mycket av att jag själv värderar den information jag hittar utifrån mina egna kunskaper. I de fall jag varit osäker har jag kunnat söka mig till Microsofts egna dokumentation om programmeringsspråket C# (Microsoft, 2023). Deras dokumentation är givetvis en källa att lita på då de själva utvecklat programmeringsspråket. Det största problemet med källor jag stött på är utdaterad information. Eftersom MonoGame som sagt bygger på XNA refererar många källor till dokumentation som rör fel ramverk. Detta har dock inte skapat större motgångar eftersom det inte är alltför svårt att översätta koden mellan de olika ramverken. MonoGames egna dokumentation (MonoGame Foundation, Inc. 2024) har även använts flitigt.

6. Avgränsning

För att detta arbete mest ska handla om utvecklingen av “pixel perfect” och inte om skapandet av spelmotorer och dylikt har vissa avgränsningar satts. Vissa avgränsningar ser även till så att arbetet är hanterbart och möjligt att slutföra inom tidsramen.

6.1 Programmera renderingen

På grund av tidsramen är det inte relevant att utveckla en renderare helt från grunden. Detta är väldigt krävande och arbetet skulle röra sig utanför frågeställningen. Istället för att utveckla renderaren själv har jag använt mig av färdiga ramverk som gör det möjligt att rendera grafik utan större möda. Principen är exakt densamma och enda skillnaden blir att en beskrivning om hur grafikrendering fungerar förekommer.

6.2 Användning av färdiga system

Användingen av färdiga system, ramverk och spelmotorer som skapar en “pixel perfect” effekt kommer inte att användas då detta arbete handlar om skapandet av ett sådant.

7. Resultat

7.1 “Pixel perfect”-grafik

För att rendera något “pixel perfect” oavsett upplösning krävs det nästan endast att man renderar något på en låg upplösning och att man sedan skalar upp de ritade grafikerna till önskad storlek. Skalar man bara upp grafiken utan någon vidare eftertanke uppstår problemen som tidigare nämnt. För att behålla samma krispiga kvalitet behöver man skala upp grafiken med ett jämnt tal och ändra texturinterpoleringstekniken. Följande kod deklarerar ritytorna som behövs och de variabler som kommer användas.

```
private RenderTarget2D fluidRender;
private RenderTarget2D staticRender;

public static int viewportWidth = 320, viewportHeight = 180;
public static int sceneWidth = viewportWidth + 2, sceneHeight = viewportHeight +
2;

public static int scaleFactor = 5;
public static int windowWidth = viewportWidth * scaleFactor, windowHeight =
viewportHeight * scaleFactor;
```

RenderTarget2D fluidRender: En rityta för allt i programmet som kommer påverkas av kameran.

RenderTarget2D staticRender: En rityta för allt i programmet som kommer vara fristående från kamerans rörelser.

int viewportWidth, viewportHeight: Dessa heltal bestämmer storleken för programmets *viewport*. En viewport innebär ett område, i detta fall rektangulärt, där grafik visas i. Alltså kommer grafiken visas genom ett fönster på 320 * 180 pixlar stort.

int sceneWidth, sceneHeight: Dessa heltal bestämmer storleken för något jag kallar för en scen. Denna scen är en viewport som är fristående från den vanliga viewport:en. Anledningen till detta är för att ge utrymme till potentiella kamerarörelser. Som det framgår i koden så bestäms en scens dimensioner av att addera ett par extra pixlar till viewport:en. Utan dessa extra pixlar kommer det uppstå svarta kanter på ritytan då kameran fungerar genom att förflytta ritytan.

int scaleFactor: Heltalet scaleFactor bestämmer hur mycket den tidigare bestämda viewport:en ska skalas upp. Detta behöver vara ett heltal för att undvika en uppskalning som slutar upp med decimaltal som resultat. En sådan felaktig uppskalning skulle leda till missformade pixlar.

int windowWidth, windowHeight: Heltalen window- width och -height bestämmer programfönstrets dimensioner utifrån produkten av viewport:ens dimensioner multiplicerat med uppskalningsfaktorn.

Alla dessa variabler lägger grunden för att kunna skala upp pixel art till högre upplösningar utan att förlora varken pixlar eller kvalitet. Senare under koden för spel-loopen används variablerna för att rendera spelet. Under programmets “Draw”-loop ritas grafik på följande sätt:

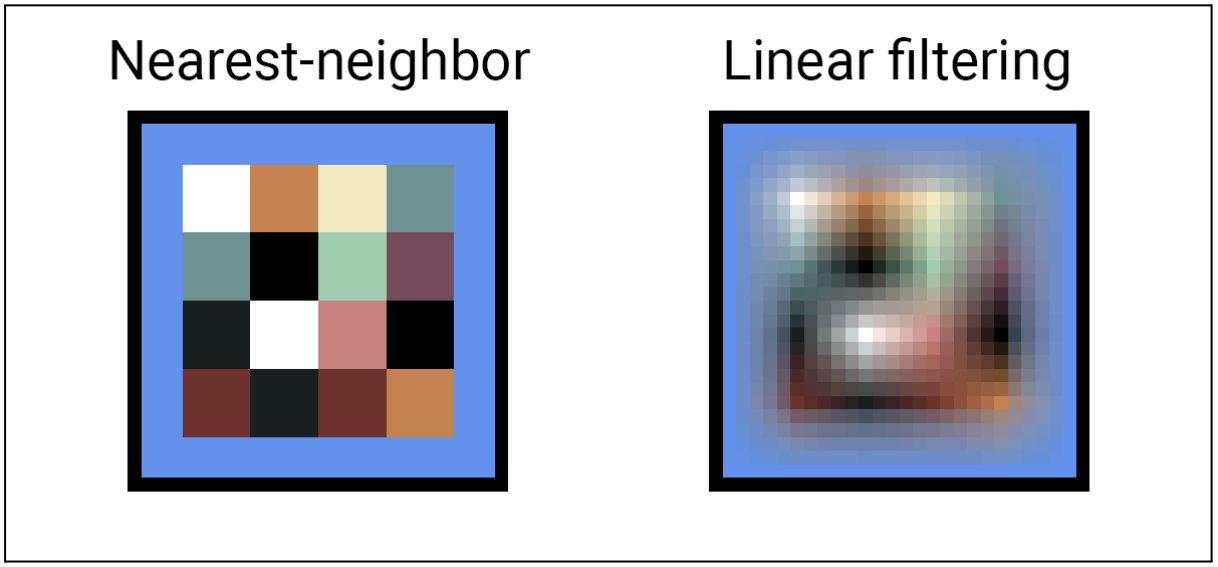
```
GraphicsDevice.SetRenderTarget(fluidRender); // Bestämmer rityta
GraphicsDevice.Clear(Color.CornflowerBlue);  // Rensar buffern
_spriteBatch.Begin(                          // Startar ritandet
samplerState: SamplerState.PointClamp,
```

```
transformMatrix: camera.transformMatrix);

            // Ritandet sker här

_spriteBatch.End(); // Avslutar ritandet
```

Denna del av koden är den sista för att rita allt i programmet som kommer påverkas av kameran. I början bestäms målet för all rendering via GraphicsDevice.SetRenderTarget. I detta fall sätts renderingsmålet till *fluidRender*. Därefter används GraphicsDevice.Clear för att innan varje bildruta rensa backbuffern till en specifik färg. Om rensningen inte sker uppstår artefakter som repeterade pixlar till exempel. Sedan används kamerans förflyttningsmatris som SpriteBatchens "transform matrix" för att kamerarörelserna ska synliggöras. Parametern *samplerState* sätts till *PointClamp* för att tillåta krisp grafik vid storleksändringar som förstoring. PointClamp använder en en interpoleringsteknik som kallas "nearest-neighbor" vilket endast tittar på den närmaste pixelns färger – inte de kringliggande. Detta gör att färgerna är konstanta och någon övergång mellan färger uppstår inte. Denna interpoleringsteknik är något som oftast undviks då texturerna blir väldigt blockiga, dock är det den önskade effekten systemet ska åstadkomma. Skillnaden mellan "nearest–neighbor"– och "linear"–interpolering är tydlig (Se figur 3) och att använda något annat än "nearest-neighbor" ger inte en önskvärd grafik.



Figur 3: *Texturinterpoleringstekniker* (Författarens bild. 2024)

Exakt samma princip gäller för rendering av grafik som inte påverkas av kamerarörelser. Enda skillnaden i koden är att den valda ritytan sätts till *staticRender* där ingen *transformMatrix* används.

7.2 Rendera ritytan

Ritytan renderas efter programmets grafiker har renderats. Detta är det mest väsentliga steget för att få sökt effekt. Ritytan ritas på samma sätt som tidigare grafik med skillnaden att ritytemålet nollställs via *GraphicsDevice.setRenderTarget(null)*. Koden för ritandet av en rityta som inte påverkas av kameran är följande:

```
GraphicsDevice.SetRenderTarget(null); // Nollställer ritytan
_spriteBatch.Begin(samplerState: SamplerState.PointClamp); // Startar ritandet

_spriteBatch.Draw(staticRender, new Rectangle(0, 0, sceneWidth * scaleFactor,
sceneHeight * scaleFactor), Color.White); // Ritar den statiska ritytan

_spriteBatch.End(); // Avslutar ritandet
```

Koden skiljer sig åt för en rityta som är statisk mot en rörlig eftersom detta arbete även implementerar ett system som gör kamerarörelserna mjuka oavsett upplösning. Förklaringen för hur systemet fungerar och ser ut kommer i följande underkapitel. Det som händer i koden ovan är att ritytan ritas ut på fönstrets origo (0,0) och skalas sedan upp med uppskalningsfaktorn för att fylla ut fönstret. Här blir vikten av att skala allt med samma faktor tydligt. Om förstoringen av ritytan skulle skilja sig mot förstoringen av fönstret skulle det uppstå antingen tomma ytor eller beskurna delar. Ritytan ritas med färgen vit (Color.White) för att inte missfärgas. Däremot öppnar denna parameter upp för möjligheten att ändra hela ritytans färg. Ett användningsområde för det vore till exempel att göra hela ritytans färg lite varmare eller kallare beroende på miljön. Precis som ritandet av grafiken används även "nearest-neighbor"-interpolering här för att behålla de konstanta och skarpa kanterna.

För att rendera en rörlig rityta, alltså en som påverkas av kamerans rörelser så används följande kod:

```
GraphicsDevice.SetRenderTarget(null); // Nollställer ritytan
_spriteBatch.Begin(samplerState: SamplerState.PointClamp); // Startar ritandet

_spriteBatch.Draw(fluidRender,
new Rectangle((int)Math.Round((camera.currentPosition.X % 1) * scaleFactor),
(int)Math.Round((camera.currentPosition.Y % 1) * scaleFactor),
sceneWidth * scaleFactor, sceneHeight * scaleFactor),
 Color.White); // Ritar den rörliga ritytan

_spriteBatch.End(); // Avslutar ritandet
```

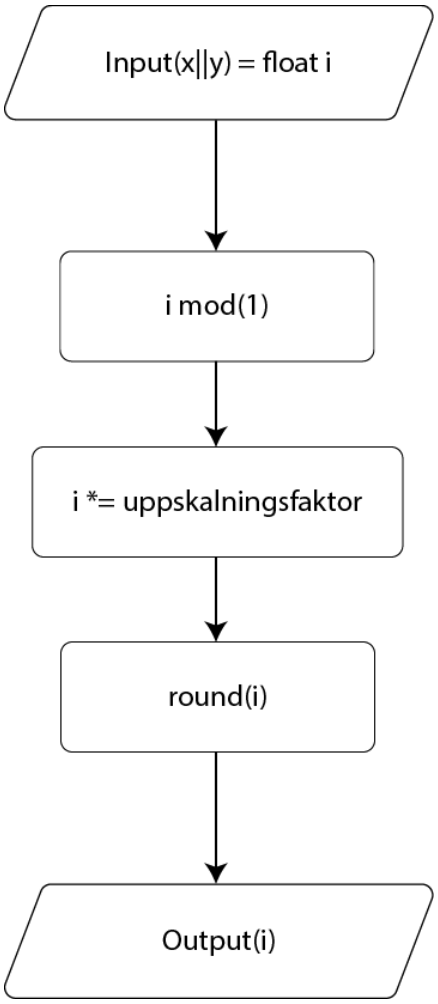
Denna kod är väldigt lik koden för den statiska ritytan. Det som skiljer sig åt är positionen av ritytan. Positionen för den rörliga ritytan bestäms av en algoritm som utifrån kamerans position förskjuter den för att upplevas som att ritytan förflyttas på enheter mindre än en pixel. I följande underkapitel förklaras detta närmre. Dessa två delar gör så att den eftersökta effekten skapas och en rityta som från början var 320x180 pixlar stor kan visas på upplösningar hur stora som helst utan att tappa kvalitet eller skärpa.

7.2.1 Mjuka kamerarörelser

Något som vanligen förekommer i moderna 2D-spel är kamerasystem med mjuka rörelser. Om ett sådant system implementeras i ett spel renderat i låg upplösning utan några ytterligare anpassningar uppstår ett tydligt problem. Då kameran rör sig långsamt upplevs rörelserna som väldigt ryckiga. Detta beror på att grafikerna fortfarande renderas på en väldigt liten upplösning i grunden och därmed förhindrar rörelser som på skärmen är mindre än en synlig

pixel. Detta går att korrigera med hjälp av en algoritm som använder sig av kamerans position för att förskjuta ritytans position. Algoritmen skapades under detta arbete och går att användas inom vilket ramverk som helst. Denna algoritm bör användas för beräkningen av både ritytans x- och y-position för att skapa mjuka kamerarörelser oavsett axel. Beräkningen av vardera koordinat görs genom att mata in respektive värde i algoritmen. Se diagram 1 för flödesschemat av algoritmen.

Diagram 1: *Algoritm – Flödesschema* (Författarens diagram. 2024)



Användningen av algoritmen i koden är följande:

```
_spriteBatch.Draw(fluidRender,
new Rectangle((int)Math.Round((camera.currentPosition.X % 1) * scaleFactor),
(int)Math.Round((camera.currentPosition.Y % 1) * scaleFactor),
sceneWidth * scaleFactor, sceneHeight * scaleFactor),
Color.White);
```

Det som bestämmer ritytan *fluidRender*s position är algoritmen i rektangelns variabler. En "Rectangle" inom MonoGame är en struktur som i detta fall håller fyra heltal. De två första heltalen är *x* och *y*, vilket bestämmer rektangelns position. De två sista heltalen har namnen *width* och *height* och precis som namnet låter bestämmer heltalen rektangelns bredd respektive höjd. Rektangelns x- och y-värde bestäms av algoritmen som helt enkelt tar

kamerapositionens decimaler och multiplicerar med uppskalningsfaktorn för att sedan omvandlas till ett heltal. Alltså ser algoritmen för att räkna ut förskjutningsvärdet för x-positionen ut så här:

- Använd modulo-operator för att få ut decimaltalen från kamerans x-position
- Multiplicera resultatet med uppskalningsfaktorn
- Omvandla slutgiltiga resultatet till ett heltal som används till ritytans position

Om kameran har en x-position på -50.439 och en uppskalningsfaktor på 7 skulle ritytans x-position räknas ut på följande vis med denna algoritm:

$$-(50,439\ mod(1)) = -0,439$$
$$-0.439 \cdot 7 = -3,073$$
$$round(-3,073) = -3$$

Exemplets slutresultat på -3 innebär alltså att ritytans x-position kommer ritas på position -3, alltså -3 pixlar från fönstrets origo. Exakt samma process sker för y-positionen och tillsammans skapas en effekt av att kameran förflyttas mjukt fast egentligen är det ritytan som justerar sig efter kamerapositionen. En synnerligen positiv egenskap med att detta gick att bryta ner till en simpel algoritm är att det tillåter universell användning av förskjutningen. Eftersom kameran, oavsett spelmotor, alltid har en position går det att använda sig av detta mjukningssystem inom alla spelmotorer och ramverk.

8. Sammanfattning, Diskussion och slutsats

8.1 Sammanfattning

Detta arbete gick ut på att skapa ett system som tillåter rendering av högkvalitativa pixel art-grafiker oavsett upplösning och en lösning för mjuka kamerarörelser i sammanhang där det vanligtvis inte är möjligt. Anledningen till skapandet av detta system var att framställa en lösning för de problem som uppstår vid rendering av lågupplöst grafik på högupplösta skärmar. Lösningen på detta problem skulle tillåta moderna tekniker att samverka med den retrografik som fortfarande används inom främst indie-spelutveckling. Resultatet gav ett system med förväntade effekter och även en direkt inblick på de möjligheter systemet ger. Den felfria uppskalningen av pixelgrafiker beskrivs på ett sätt som tillåter enkel översättning till valfritt programmeringsspråk. Ytterligare gick den väsentliga delen med mjuka kamerarörelser att bryta ner till en enkel algoritm och går därmed att använda inom alla programmeringsspråk och ramverk.

8.2 Diskussion

Utifrån min frågeställning och dess syfte är den resulterande koden tillfredsställande. Som önskat tillåter den process jag kom fram till att skala upp pixel art till vilken upplösning som helst utan att förlora kvalitet. Speciellt nöjd är jag däremot med den algoritm jag skapade som tillåter kameran att röra sig i enheter som till synes är en subpixel. Något liknande har jag inte kunnat hitta under min informationssökning och att det gick att få ner systemet till en simpel algoritm är väldigt positivt. Algoritmen går som sagt att använda i alla programmeringsspråk utan någon större förändring vilket tillåter den att användas i diverse sammanhang. Vidare är även uppskalningen av grafiken något som går att använda inom i princip vilket

grafikramverk som helst så länge möjligheten att rita till en rityta finns – vilket det i majoriteten av fallen gör. I de fall det inte finns en rityta/”render target” går det att skapa en egen rityta genom att rita till en mindre “frame buffer”. Ritytan öppnar även upp för vidare optimering av systemet. Eftersom det nu kräver att man ritar grafiken två gånger (först till den mindre ritytan, sedan till den större) kan det dra ner prestandan lite. Detta skulle säkerligen gå att lösa genom att direkt rita i mindre upplösning på standard-ritytan och sedan zooma in på grafiken med en omvandlingsmatris eller dylikt. Dock gillar jag hur systemet använder sig av ritytor då det öppnar upp för att kunna ha lager av olika skalor. Till exempel kan allt i spelvärlden ritas på en låg upplösning medan användargränssnittet är högupplöst. Detta är såklart endast estetiska val och här får man helt enkelt välja om man värdesätter en lite mer optimerad grafik över en mer estetiskt tilltalande grafik. Något jag märkte när jag skalade upp grafiken var avrundningsproblem. Exempelvis brukar man hantera objekts positioner med decimaltal och sedan omvandla dem till heltal. Detta är inte ett problem vid högre upplösningar där man inte kan urskilja enskilda pixlar. När grafiken är extremt lågupplöst kan man däremot se hur rörelserna kan bli ryckiga. Detta kunde jag lösa genom att konsekvent använda mig av samma avrundning då det behövdes istället för att blanda olika omvandlingar.

8.3 Slutsats

Syftet med arbetet var att skapa ett system som tillåter lågupplösta grafiker (pixel art) att skalas upp till högre upplösningar utan att förlora kvalitet. Möjligheten att kunna förflytta grafik i enheter liknande till synliga subpixlar skulle även finnas. I detta arbete skapades ett system som i enlighet med dess frågeställning och syfte ger önskad funktion.

9. Demo av användningsområden

Detta system kan givetvis användas på många olika sätt. I och med detta kommer detta kapitel visa olika demos av användningsområden. Dessa demos visar hur detta system kan utnyttjas för att skapa unika grafiker som har en bra kvalitet oavsett upplösning.

9.1 Procedurellt genererad pixelgrafik

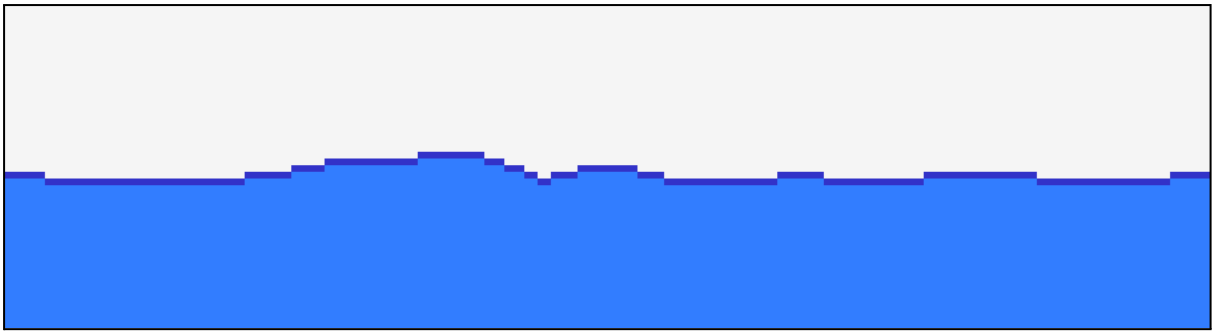
Något som detta system tillåter är procedurell pixelgrafik. Detta kan innebära allt från 3D-modeller som nu visas som pixel art till enklare grafiker såsom former som följer normer inom pixel art.

9.1.1 Ritande av former

Ett praktexemplar på den procedurella pixelgrafiken är ritandet av former. Genom att rita enkla former såsom linjer, fyrkanter och cirklar kan man åstadkomma ett stort antal effekter. Det charmiga med detta är att allt ser ut att vara manuellt skapat pixel art som följer normer inom pixel art. Nedan finns exempel på hur sådana former kan användas i ett sammanhang.

9.1.2 Vattenytor

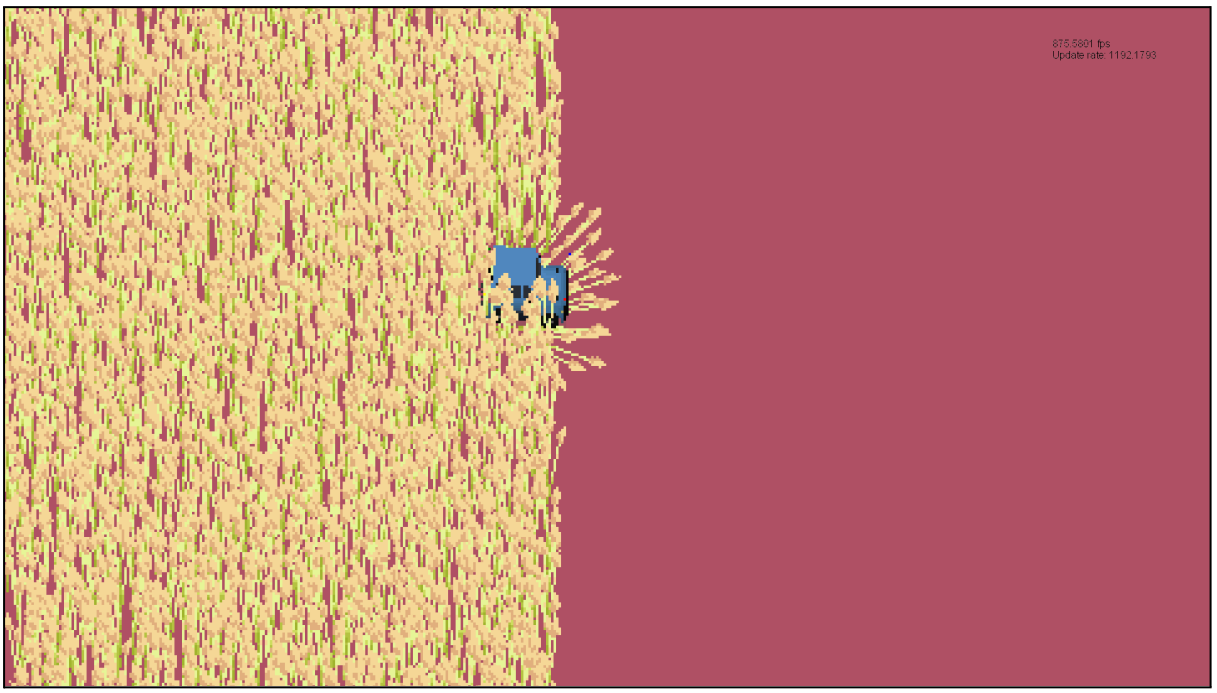
Genom att utnyttja algoritmer för att rita bezierkurvor kan man skapa en effekt som ser ut att vara en vattenyta (Se figur 4). Detta är en effekt som blir väldigt kraftfull om den tillåts interagera med omgivningen.



Figur 4: *Vattenyta* (Författarens bild. 2024)

9.1.3 Grönska

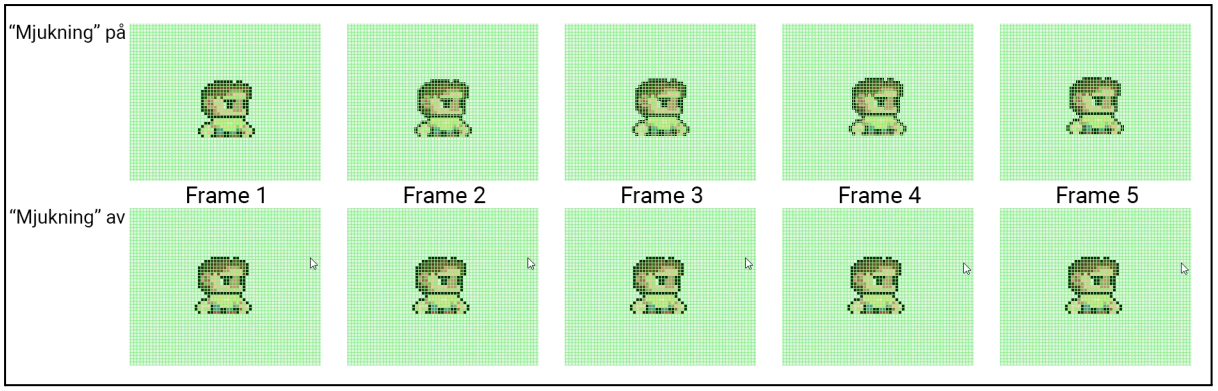
Genom att rita linjer som justerar sig efter pixel-rutnätet kan man skapa grafik som ser ut som organiska element. Nedan visas ett vetefält som skapas genom att procedurellt rita mängder av vetestrån nära varandra.



Figur 5: *Vetefält* (Författarens bild. 2024)

9.2 Kamerakontroll

Algoritmen för de mjuka kamerarörelserna blir som tydligast med en klassisk “twin-stick shooter” kamerakontroll. I vanliga fall upplevs kamerarörelserna på en låg upplösning som ryckiga när kameran rör sig långsamt. Med mitt system är kamerans rörelser så mjuka att alla negativa sidoeffekter av den låga upplösningen mer eller mindre upphör. Se figur 5 för en sida-vid-sida jämförelse av när mjukningen är på och av. För video på funktionen, se *“Sub-pixel”-kamerarörelser* (Vusir, Viggo. 2024)



Figur 6: *Jämförelse av kameramjukning* (Författarens bild. 2024)

10. Källförteckning

Microsoft. (2023). *C# reference*
https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/

MonoGame Foundation, Inc. (2024). *Reference | MonoGame*
https://monogame.net/api/index.html

Vusir, Viggo. (15 februari 2024). *"Sub-pixel"-kamerarörelser* [Video]. Vimeo.
https://vimeo.com/913377583?share=copy