

# Komplikationer med skærmopløsning i et 2D/3D miljø

David Recke, Emil Vad, Simon Vinther

27. marts 2015

---

# Indhold

<b>1</b>	<b>Introduktion</b>	<b>1</b>
1.1	Abstract . . . . .	1
1.2	Problemformulering . . . . .	1
1.3	Indledning . . . . .	1
1.4	Forord . . . . .	1
<b>2</b>	<b>Position og skalering</b>	<b>2</b>
2.1	Tekstur skalering . . . . .	3
<b>3</b>	<b>Aspekt ratio</b>	<b>5</b>
3.1	Field of View . . . . .	5
3.2	Letterbox . . . . .	6
3.3	Virtual viewport . . . . .	6
<b>4</b>	<b>Konklusion</b>	<b>8</b>

# 1. Introduktion

## 1.1 Abstract

## 1.2 Problemformulering

Hvilke tekniske komplikationer er forbundet med skærmopløsning i forhold til 2d/3d spil og hvordan kan de forebygges? Hvilke komplikationer kan der opstå når et spil skal kunne bruges på skærme med forskellige aspekt ratioer? Hvilke komplikationer opstår når teksturer skal tilpasses forskellige skærmopløsninger? Hvilke tekniske udfordringer kan der opstå når man bruger skærmen som koordinatsystem?

## 1.3 Indledning

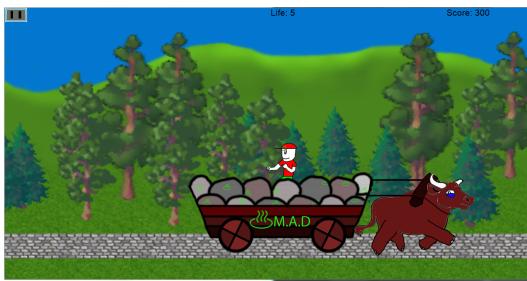
Når man skal konstruerer et spil, skal det typisk kunne spilles på mange forskellige platformer med forskelligt hardware, og for at alle spillerne på de forskellige plafomer, skal kunne få den bedst mulig oplevelse til det hardware de har, bliver man nødt til at have forskellige skærmopløsninger. Når man har forskellige skærmopløsning i sit spil, kan der opstå en række problemer, som man som udvikler bliver nødt til at løse. Disse problemer kan summeres ned til tre: Skalering, forskellige aspekt ratio, og pixelkoordinater.

## 1.4 Forord

Denne artikel er lavet til spil datamatikernes akademisk formidlings projekt / E-bogsantologi for EA Dania. Artiklen er skrevet i tidsrummet fra mandag d. 16 marts til onsdag d. 1 april af tre spil datamatiker studerende ved EA Dania i Grenaa. Emnet omhandler hvilke komplikationer der kan opstå i forbindelse med valg af skærmopløsning i 2d og 3d spil, såvel som hvordan det kan forebygges. Der vil i denne artikel blive diskuteret de centrale problemstillinger der opstår, når at et spil skal kunne tilpasse sig mange forskellige skærmopløsninger, og hvordan det kan påvirke spilleren. Det antages at læseren har en basal forståelse for spilprogrammering og koncepter involveret i spildesign.

## 2. Position og skaling

Hvis man lave et spil uden et moderne framework eller engine som er 2d baseret, ligner det en nem løsning at basere sit spil på skærm koordinator. Vi har et eksempel på dette type system fra et forrige projekt kaldet MAD 2.1, og har ud fra det og andre projekter lært hvad problemet ved skærm koordinater, også kaldt pixel koordinater, er, hvilket for det meste hænger sammen med skærmopløsningen.



Figur 2.1: MAD i produktionens opløsning



Figur 2.2: MAD i 1280x800 opløsning med skærm kordinator

De største problemer ved et skærm koordinatsystem opstår når at skærmstørrelsen bliver ændret. En situation opstår eksempelvis hvis man vil have et objekt nede i det højre hjørne af skærmen, som ikke flytter sig relativt til skærmens størrelse. På figur 2.2 og 2.1 kan man se at forskellen på hvordan spillet ser ud i to forskellige opløsninger. På figur 2.1 er det nederste højre hjørne koordinatet 1290,690, og på det figur 2.2 er det 1280,800. Fordi det ene spil er 110 pixel højere end det originale, betyder det at der er 110 pixels ned af y-aksen, som programmet ikke bruger, da alle spilobjekternes positioner er baseret på koordinater.

Hvis man så også skal skalerer alle spilobjekterne så de passer bedre til den nye opløsning, skal de skaleres med ca. 16% ( $110 \div 690 = 0.1594203 \approx 0.16$ ) på y-aksen. Udeover at det kommer til at se mærkeligt ud når spilobjekterne bliver skaleret, passer deres nye størrelse ikke til deres placering, se figur 2.3. Dette kan bedre ses på et mere ekstremt eksempel, som figur 2.4 hvor opløsningen er 860x345. Dette forårsage at man ikke kan se andet end baggrunden, fordi alle objekterne starter i højere koordinater end hvad skærmen indeholder.



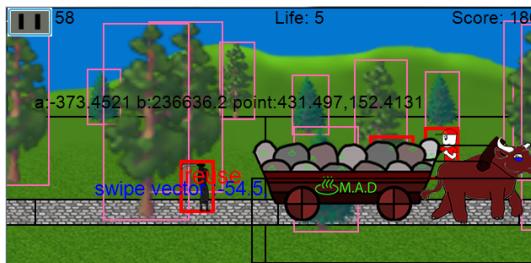
Figur 2.3: MADscaling3



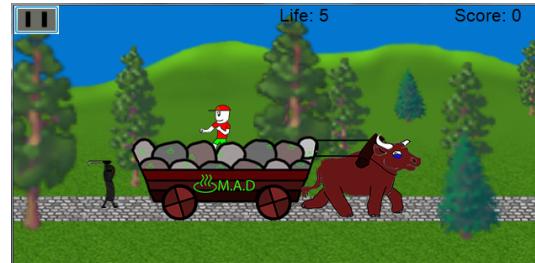
Figur 2.4: MADscaling4

Et andet problem kan være spil objekternes kollisionsboks, da deres størrelse også kan være i pixels. Hvis et spilobjekts tekstur nedskaleres uden at dets kollisions boks skaleres, vil objektets visuelle repræsentation være ude af sync med dets faktiske position og størrelse. Se figur 2.5 fra MAD, hvor positionen følger med opløsningen, men nogle af kollisionsboksene ikke gør. Da spillet blev lavet, var de fleste objekters kollisionsbokse baseret på deres sprites størrelse, hvilket er en god idé hvis man vil skalere senere. Det giver dog sjældent den ønskede størrelse af kollisionbokse, da sprites ofte er større end du vil have din kollisionboks især i 2d spil som platformers [8].

En løsning til dette problem, er ved at lave koordinaterne baseret på procenter i forhold til skærmens størrelse.



Figur 2.5: MAD hvor kollisionsboksene ikke er relative til spilobjekternes størrelse



Figur 2.6: MAD hvor kollisionsboksene er relative til spilobjekternes størrelse

Hvis man vil placere et objekt et sted i spillet, skal dets koordinat være: produktions X × (faktisk skærmbredde ÷ produktions skærmbredde), produktions Y × (faktisk skærmhøjde ÷ produktions skærmhøjde). Det sammen kan gøres med kollisions boksen, bare hvor det er i forhold til spriten:

produktions kollisions boks bredde × (faktisk spritebredde ÷ produktions spritebredde) · produktions kollisions boks højde × (faktisk spritehøjde ÷ produktions spritehøjde).

Disse udregninger er brug I MAD. som det kan ses på figur 2.6, og har løst problemerne involveret med fejlplaceret objekter og kollisionbokse. En anden løsning er at unlade at kode i skærm koordinator, og have game world koordinator, som ved hjælp af en kamera klasse, kan konverteres til skærm koordinator. Se kodeeksempel for et eksempel af en kamera klasse.

Kamera klassen kender opløsningen, og kan tage højde for hvordan objekterne skal tegnes på skærmen. Dette er en løsning der skal implementeres fra starten af et projekt så det egner sig mere som en forebyggelse, end en løsning. Et kamera gør ofte koden meget nemmere at forstå, da spilverdenens koordinater giver mere mening end skærmkoordinater. Mange moderne frameworks og engines bruger også spilverdenens koordinater, eksempelvis Unity [2], da der er så mange problemer med skærmkoordinater. Hvis skærmkoordinater bliver brugt, er det på en måde der er i forhold til skærmen størrelse.

## 2.1 Tekstur skaling

I konstruktionen af assets til spil til digitale platforme, er det et krav at tekstrurer og sprites skal skalere efter størrelsen på skærmen af den givne platform. En umiddelbar løsning ville indebære at tegne alle visuelle assets i høj opløsning, derved er det sikret at en høj kvalitet bliver opnået på høje skærmopløsninger [10].

Der er et problem når man skalerer nedad. Hvis man f.eks. skalerer fire pixels ned til det halve, får man to pixels, men skalere man tre pixels ned til det halve, opstår der 1,5. Da man ikke kan have halve pixels, et det enten én eller to pixels man ender op med, hvilket kan få grafikken til at se aliased ud [11]. Mange moderne algoritmer prøver selv at løse problemet så det er mindre tydeligt at teksturen er blevet skaleret.[9] Det kan dog stadig være synligt, og desuden tager det også lang tid at lave en tekstur med meget høj opløsning end et med lavere opløsning.

En anden løsning er at lave teksturene med lav opløsning, og derefter skalerer dem op. Dette får dog teksturene til at se pixelerede ud [12]. En tredje måde at gøre det på er at lave vektorgrafik, da de skalerer bedst [10]. Det tager dog længere tid at lave vektorgrafik end normal grafik,[10] og derudover kræver vektorgrafik også mere cpu kraft, men mindre ram [10]. Derfor er der nogle platforme hvorpå vektorgrafik kan fungere bedre end andre, da cpu kan varierer en del fra platform til platform [1].

Der er ikke en bedste måde at løse skaleringsproblemer på, da det afhænger meget af det spil man laver, og hvor lang tid man har til at lave det. Generelt er det en god idé at lave tekstrurer i høj opløsning, da meget morderen software sørger for at det stadig ser godt ud når det bliver nedskaleret.

Listing 2.1: Et eksempel på en kamera klasse

```

1 //A struct to represent a point in 2 axes.
2 struct Point
3 {
4     public float x = 0;
5     public float y = 0;
6 }
7
8 class Kamera
9 {
10    //The Position of the camera in game world coordinates.
11    Point position;
12
13    int screenHeight;
14    int screenWidth;
15
16    // the amount of width(x) and height(y) game world coordinates the camera can see.
17    static const float maxXOnScreen = 100;
18    static const float maxYOnScreen = 100;
19
20    Kamera(Point position, int screenHeight, int screenWidth)
21    {
22        this.position = position;
23        this.screenHeight = screenHeight;
24        this.screenWidth = screenWidth;
25    }
26
27    Point positionOnScreen(Point position)
28    {
29        Point newPosition;
30        newPosition.x = screenWidth / 2 +
31            ((position.x - this.position.x) * screenWidth / maxXOnScreen);
32        newPosition.y = screenHeight / 2 +
33            ((position.y - this.position.y) * screenWidth / maxYOnScreen);
34        return newPosition;
35    }
36
37 }
```

## 3. Aspekt ratio

Aspekt ratio refererer til den proportionelle forskel mellem en skærms højde og brede [6]. Der findes mange forskellige aspekt ratios, som bliver brugt til forskellige formål [14]. I dag er de langt mest benyttede opløsning blandt spillere på Steam 1920x1080, efterfulgt af 1366x768. Begge disse opløsninger er 16:9 formater. Andre formater der udgør en betydelig del er 1280x1024 (5:4), 1680x1050 (16:10), og 1440x900 (16:10) [13].

At skifte mellem to forskellige opløsninger, som deler det samme aspekt ratio, er relativt simpelt, da teksturer blot kan skaleres til at have en anden størrelse. Så længe at ratioen er den samme, og teksturen bliver skaleret ned til en dårligere opløsning, er der generelt ikke et problem.

Problematikken opstår når at aspekt ratioen ændres, da en skalering ikke længere blot vil ændre på størrelsen af teksturen, men også på forholdet mellem bredden og højde. Det betyder at akserne ikke bliver skaleret synkront, hvilket kan give billedet et andet visuelt indtryk. Dette har været tilfældet i mange ældre spil, der originalt ikke var lavet til widescreen, blandt andet Warcraft 3, som kan ses på figur 3.1 og 3.2. [15]



Figur 3.1: Warcraft 3 i dets originale 4:3 aspekt ratio



Figur 3.2: Warcraft 3 skaleret asynkront til et 16:9 aspekt ratio

### 3.1 Field of View

Der er to måder at komme uden om dette problem på. Den ene er at ændre på field of view (FOV), som er hvor stor en del af spilleren ser af spilverdenen. Dette bringer imidlertid sine egne problemer på banen. Når FOV ændres, ændres der også på det output spilleren får på sin skærm, hvilket betyder at alle spillere ikke ser spilverdenen ens. Hvis at spillet ikke er kompetitivt spil, behøver dette ikke at være en bekymring, da spilleren kun spiller mod sig selv, og spillet i den forstand er ligestillet for alle. Hvis spillet derimod er kompetitivt, medfører det med stor sandsynlighed at spillere med et vist aspekt ratio har en fordel over dem med et andet aspekt ratio [4].

Hvis vi ser på et 2d spil, og forestiller os et endless runner spil, så som Robot Unicorn Attack, hvor at man kun ser hvad der er lige foran sin karakter, vil et bredere FOV betyde at man har længere tid at forberede sig på de forhindringer der kommer forude.

Hvis vi ser på et 3d spil, så som CS:GO, vil et bredere field of view betyde at spilleren kan se længere til siderne. Det vil sige at spillere med et mindre FOV effektivt set har en blind vinkel, som at deres modspillere med et bredere FOV ikke har, hvilket giver en stor fordel når det kommer til at få visuel kontakt med sine fjender. På figur 3.4 ses f.eks. en dør til højre, som ikke kan ses på figur 3.3 pga. forskellen i FOV.



Figur 3.3: CS:GO med aspekt ratio 4:3



Figur 3.4: CS:GO med aspekt ratio 16:9

## 3.2 Letterbox

Den anden måde at komme uden om skaleringsproblemer på, er at tilføje sorte bars, også kaldet letterbox, til toppen og bunden af skærmen. På den måde tvinges spillet reelt set til at køre i et statisk aspekt ratio, og dermed er der ingen uretfærdigheder når det kommer til FOV [7]. Disse sorte bars kan også blive smidt på højre og venstre side af skærmen, og i disse tilfælde kaldes de pillarboxing [3]. Nogle spil er nødt til at benytte sig af dette koncept, for at deres gameplay kan hænge sammen.

Et af disse spil er Speedrunners. Formålet med Speedrunners er at løbe om kap, og komme fremad hurtigst muligt, og dermed efterlade ens modstander bag sig. Alle fire spillere ser på den samme del af spilverdenen, og når en spiller falder for langt bagefter, og kommer uden for skærmen, bliver spilleren elimineret. I det at spillet afhænger af at lose condition er at komme uden for skærmen, ville forskellige aspekt ratioer betyde at dem med et mindre FOV enten ville dø tidligere, eller vil være ude af stand til at se sin karakter mens han stadig er i live. Ved at tilføje letterboxes til spillere der ikke kører med et 16:9 aspekt ratio, sikrer spillet sig at alle har det samme syn på spil verdenen, og der er enighed om hvornår en spiller er ude af skærmen. På figur 3.6 ses spillet i det tiltænkte 16:9 aspekt ratio, og på figur 3.5 ses spillet i et 4:3 ratio hvor det er letterboxed.



Figur 3.5: Speedrunners med aspekt ratio 4:3



Figur 3.6: Speedrunners med aspekt ratio 16:9

## 3.3 Virtual viewport

På mobilmarkedet findes der ligeledes mange forskellige aspekt ratios, og det bearbejdes i store træk på samme måde som på platforme med større displays. Det er dog i de færreste tilfælde at det er ønskeligt at tilføje letterboxes til mobil platformen, da der i forvejen ikke er meget plads at arbejde med, og det i desuden kan give et indtryk af at spillet ikke er beregnet til den brugte model.

En virtual viewport kan bruges til at vise en del af et environment, der kan justere sig selv efter dit aspekt ratio. Ved at lave en baggrund bred nok til at supportere et bredt aspekt ratio, og højt nok til at supportere et smalt et, kan man ved at tage udgangspunkt i midten af baggrunden, sikre at det vil give et godt indtryk på ethvert aspekt ratio. Dertil kan man sørge for at andre elementer på skærmen også justere sig selv efter kanterne af skærmen, for at sikre at de ikke bliver placeret på en måde der giver et upoleret indtryk [5].

Dette koncept benyttes også på andre platforme. F.eks. har det online kortspil Hearthstone grafik der går ud over selve den kvadratiske brugerflade, således at grafikken på en widescreen skærm også giver et godt visuelt indtryk, uden et behov for at benytte sig af letterboxes i siderne. På figur 3.8 og 3.7 ses det hvordan Heartstone tilpasser sig den givne skærms aspekt ratio.



Figur 3.7: Hearthstone med aspekt ratio 4:3



Figur 3.8: Hearthstone med aspekt ratio 16:9

## 4. Konklusion

Når man konstruerer et spil, skal man tage højde for en række ting. Da alle der kommer til at spille spillet ikke bruger den samme opløsning, er det udviklerens job at sikre at alle spillerne får den bedste mulige oplevelse på deres givne system. I dag bruger nyere PC systemer typisk et 16:9 ratio, og derfor er det fordelagtigt at bruge dette som standard, og så justere andre ratioer derefter. Forskellen på aspekt ratioer kan behjælps på flere måder, og hvilken en der bør vælges, kommer an på hvilken type spil der bliver lavet. Hvis et spil skal være meget kompetitivt, kan letterboxes sikre at spillerne har samme syn på spil verdenen, så spillet er så retfærdigt som muligt. Hvis spillet derimod ikke er kompetitivt, eller hvis oplevelsen prioriteres højere, kan spillets FOV gøres smallere for spilleren, så hele deres skærm kan benyttes, hvilket vil se flottere ud på skærmen. Når teksturer skal skaleres, er der mange måder at man kan lave texturen der bliver skaleret godt, fra høj opløsnings tekstuurer til vektorgrafik. Der er ikke en definitiv løsning, men mange spil lave høj opløsing teksture og nedskalere dem, da moderne billedprogrammer og engines er ret gode til at sørge for at nedskalere grafik ser godt ud. Skræm koordinater har mange negativ følger da de altid afhænger af skærmens og derfor ikke brugt i mange modern engine og frameworks, og fordi