# Grafisk udvikling

Når der udvikles til mobile platforme er der adskillige elementer, som skal tages højde for. Samtidig er det heller ikke altid muligt at udnytte den viden og de ressourcer, som er gældende til udvikling af eksempelvis computerspil. Udvikler man derfor til smartphones og tablets er det nødvendigt at undersøge og redegøre for de faldgruber, som et projekt kan bremses af.

## Begrænsninger og optimering

Den største forskel man ser, når man sammenligner fx computerspil eller konsolspil med mobile spil, er den computerkraft, som de forskellige har til rådighed. Hvorimod computere og konsoller har rig mulighed for at blive udrustet med kraftfulde komponenter, så er fokuspunktet i smartphones og tablets mobilitet herunder kompakthed. Dette stiller høje krav til hardwaren, da det fysisk begrænser pladsen til ydeevnen. Samtidig er temperaturen et element, som skal indberegnes, hvilket kræver nedkøling af hardwaren enten i form af blæsere eller vandkøling. Dette betyder, det er balancen imellem computerkraft, nedkøling og fysisk plads, som udgør en enheds arbejdskraft.

Herefter er det relevant at undersøge, hvilke teknologier og programmeringspraktikker, som kræver stor computerkraft. Specielt inden for videospil er der utrolig mange systemer og elementer, som skal spille sammen, ikke mindst det grafiske komponent.

### GPU - Graphics Processing Unit

GPU’en, også kaldet grafikkort, er det stykke hardware, som står for de grafiske udregninger. Det er sammen med processoren oftest flaskehalsen i enhedens ydeevne. I mobile enheder er grafikkortet typisk begrænset af ’fillrate’, som angiver, hvor meget grafikkortet kan rendere og skrive til grafikhukommelsen i sekundet. Fillraten afhænger oftest af shaderen, som er det stykke software, der udregner det visuelle resultat af interaktionen med modeller, materialer, lys, skygger mv. En kompleks shader resulterer derfor i en høj fillrate i applikationen, og det anbefales at bruge en så simpelt som muligt shader.

Dog findes der andre områder, som kan optimeres på. Et er tekstur-detaljegraden, som kræver hukommelse af den mobile enhed. Kræver dette for meget hukommelse, er der et par metoder, som kan tages i brug. Det første der bør overvejes er komprimering af teksturene, hvilket gør dem lettere at indlæse i hukommelsen.

Yderligere bør der gøres brug af e teknik kaldet mipmaps. Disse er en række versioner af den samme teksture med gradvist lavere opløsning. De erstattes med den oprindelige tekstur, typisk når kameraet flyttes længere væk fra objektet med den tilhørende tekstur. Formålet med teknikken er at øge renderingshastigheden, da de mindre teksture anvender mindre hukommelse, og samtidig reducerer den også GPU-forbruget. Ydermere er det muligt at øge den visuelle kvalitet, da rendering af teksture med høj opløsning med eksempelvis høj afstand til kameraet kan resultere i moiré mønstre. Her ses et eksempel på ingen brug af mipmaps imod brugen af mipmaps.



Som det ses på billedet er der stor visuel forskel på de to billeder. Billedet, som gør brug af mipmaps, er langt mere behageligt at se på, selvom billedet uden brug af mipmaps teknisk set har højere opløsning. Desuden er det muligt at se moiré-effekten i det venstre billede, da den høje opløsning ikke kan gengives i horisonten. Brug af mipmaps øger renderingshastigheden, men det skal dog bruges med omtanke, da processen i sig selv kræver omkring 33% mere hukommelse.

En anden måde, at optimere brugen af teksture på, er ved at benytte sig af det, som kaldet et tekstur atlas. Et tekstur atlas er en enorm billedfil, som indeholder en lang række, hvis ikke alle, teksture i en scene. Dette gør renderingen mere effektiv, da filen bliver håndteret som et enkelt enhed i udregningerne, hvilket især er effektivt, hvis spillet indeholder mange små objekter, som bruges ofte. Et typisk atlas ser sådan ud:



Dette kræver dog lidt tilpasning, da UV-mappet skal hænge sammen med tilhørende model. Et UV-map er det, som definere, hvordan en tekstur skal pålægges modellen. Lægges alle teksturene derfor i en ny fil kan det være nødvendigt at sikre sig korrekt pålægning af disse. Det er samtidig muligt for teksturene at overlappe ved brug af mipmaps, og det er derfor igen nødvendigt at sikre sig at underbillederne ikke bliver forurenet af de omkringliggende billeder.

Sidst er det værd at nævne LOD, som står for Level Of Detail. Med LOD forsøger man at mindske antallet af draw calls ved at simplificere objekterne i scenen ud fra afstanden til kameraet. Teknikken minder meget mipmaps, men dækker over modellerne i scenen. Et eksempel kan ses her:



Derudover er det naturligvis også en mulighed at slette objektet helt, når det når en hvis afstand fra kameraet. Samtidig med det er muligt at udnytte denne funktionalitet til terræn, så er det muligt at kontrollere om shaderen skal bearbejde de objekter med høj afstand til kameraet.

Alt i alt er formålet at skabe et produkt, som kan håndteres af flest mulige enheder for at øge det potentielle marked.

### Optimering af lys

Et andet element, som i høj grad påvirker ydeevnen er de forskellige former for lys, som kan indbygges i et produkt. Lys i Unity3D bliver renderet på én af to måder, vertexlys eller pixellys. Vertexlys er lys, som bliver udregnet per vertex i scenen og langsomt aftager fra vertexen. Pixellys bliver herimod udregnet for hver enkelt pixel på skærmen. Forskellen ses herunder, vertex-lys til venstre og pixellys til højre:



Som det ses giver pixellys langt højere kvalitets lys, men stiller også langt større krav til den mobile enhedsydeevne. Dog giver pixellys nogle muligheder, som ikke er tilgængelige med vertexlys. Med pixellys er det muligt at benytte normal maps, light cookies og realtime skygger.

Normal maps er en teknik, hvor det er muligt at gøre overfladen på et objekt mere detaljeret uden at redigere selve 3D-modellen. Light cookies er en funktion, som giver lyset et slags filter, hvor formålet er at give udvikleren mulighed for manuelt at bestemme lysets mønster.   
Realtime skygger er, som navnet indikerer, realtidsskygger, som opdatere løbende som objekter og lys mv. ændres. Realtime skygger kræver mange ressourcer og er derfor et fokuspunkt i optimeringen af lyselementerne. Processen forløber således, at først renderes alle potentielle objekter, som kan kaste en skygge, på et skyggekort, hvorefter alle objekterne, som kan modtage skygger, renderes ind i skyggekortet. Denne proces er derfor mere krævende af grafikkortet end pixellys i sig selv.

#### Lys i dette projekt

Lyset, som er brugt i dette projekt er relativt simpelt. Lyset i selve banen består af en enkelt ’directional’ lys, som fungerer som en sol. Dette betyder, det kun er rotationen på lys-objektet, som gælder, da det belyser samtlige objekter i scenen, og er samtidig den mindst krævende i form af GPU-kraft.

I forsøget på at holde den påkrævede GPU-kraft nede bruges der ikke realtime skygger, da disse er tunge for grafikkortet. Dette betyder ingen pixellys, som er beskrevet i afsnittet ’Optimering af lys’. Som det ses i figur 1, er lyset bygget så simpelt som muligt. Objektet gør ikke brug af cookies, halo, flare mv. Render Mode, som beskriver kvaliteten af renderingen af lys, står til ’Auto’, hvilket betyder den gør brug af Unitys indbygget kvalitetsindstillinger. Det samme gælder for ’Lightmapping’, som definerer hvorvidt der bruges realtime lys, baked lys eller der hentes fra kvalitetsindstillingerne. Da der hverken bruges realtime lys eller baked lys, er det naturlige valg at definere det i kvalitetsindstillingerne.

’Culling mask’ definerer, hvilke objekter, som belyses i scenen. Indstillingen er sat til ’everything’. Dette har dog ikke den store indvirkning på ydeevnen, da det fulde antal af objekter i scenen er relativt småt.

Belysningen i spillet er altså simplificeret for et opnå mindst mulig brug af GPU-kraft, hvilket hovedsagligt er løst ved at eliminere brugen af skygger.

### 3D-modeller

Tredimensionelle modeller, som oftest er brugt i videospil, er krævende komponent i et produkt. Det er dermed også et element, hvor ydeevne burde være et fokuspunkt under udviklingen af disse. Når en 3D-model udvikles til den mobile platform, er der en række punkter, som er værd at undersøge.

Det første, som bør redegøres for, er antallet af polygoner, modellen skal bestå af. Antallet af polygoner bestemmer detaljegraden af figuren og dermed kvaliteten af det visuelle element, men jo højere antal polygoner, jo højere ydeevne er nødvendig. På en mobil platform er 300-1.500 et acceptabelt antal polygoner per model sammenlignet med en stationær computer, som kan håndtere et antal på 1.500-4.000 polygoner. På spilkonsollerne Xbox 360 og Playstation 3 har modeller gennemsnitligt et antal polygoner på 5.000-7.000. Dog er det værd at indberegne antallet af modeller på skærmen samtidig, da et højt antal modeller naturligvis stiller højere krav til ydeevnen.

Det næste mål for optimeringen er at bruge så få materialer som muligt. Et materiale i 3D-modelleringssammenhæng er en samling af teksture samt shaders mv. på modellen. Dog er det ikke typisk nødvendigt med mere end ét materiale, men der kan opstå situationer, hvor kvaliteten kan forbedres.

Endvidere bør det være et mål at bruge så få knogler i modelleringen som muligt. Knogler hentyder til et system, som gør det muligt at animere og styre 3D-modeller uden at skulle ændre på selve polygonerne. Det anbefales at bruge højst omkring 30 knogler i en mobilapplikation. Dette kan sammenlignes med et computerspil typisk med imellem 15 og 50.

Det anbefales kun at bruge en enkelt ”skinned mesh rendere” for hver karakter i spillet. Mesh renderen er den, som indlæser modellen samt materialer mv. Tilføjer man derfor flere mesh rendere tager renderingen tilsvarende længere tid, og der er oftest ikke nødvendigt med mere end én renderer.

## Udvikling til den mobile platform

### Skærmstørrelser

Når man skal udvikle en brugergrænseflade er der flere elementer, som skal tages højde for bl.a. skærmstørrelse og dimensionerne, altså forholdet mellem længden og bredden. Skærmstørrelsen og dimensionerne på kan variere betydeligt alt efter, hvilken platform, der udvikles til, herunder mobile enheder som smartphones og tablets, men også computerskærme, tv mv. I dette projekt, hvor produktet er henvendt til smartphones og tablets, er der også et væld af forskellige skærmstørrelser, som skal indregnes i udviklingen. Konsekvensen af ikke at indarbejde de nødvendige fleksible størrelser kan i værste tilfælde gøre produktet uanvendeligt, da brugeren ikke kan benytte sig af de funktioner, som forventet. I bedste tilfælde er brugergrænsefladen ikke æstetisk nydelig, men dog anvendelig.

#### Analyse af skærmstørrelser

Før udviklingen påbegyndes er det nødvendigt at vide præcist, hvilke skærmstørrelser er mest populære og derved gør sig gældende. Samtidig gør dette det også muligt at spare ressourcer, hvis der ikke behøves at udvikles og testes til småt udbredte skærmstørrelser.

Som kan ses ud fra denne graf, ligger størstedelen af de mobile enheder inden for relativt få forskellige skærmstørrelser, da ’Andre’ kun dækker over 16,8% af alle enheder.

Det, som dog er mere relevant at redegøre for, er dimensionerne på skærmen. Da det samme størrelsesforhold betyder, at det er muligt kun at nøjes med at skalere brugergrænsefladen, volder dette kun få problemer. Derimod er det ikke muligt kun at skulle skalere grænsefladen, hvis dimensionerne er anderledes.

Kigger man derfor på de typiske dimensioner på markedet, ser man at ’16:9’ og ’5:3’ udgør knap 50% af markedet. Da disse udgør de ydre ekstremer af de mest brugte dimensioner, er det derfor muligt at lægge fokus her.

#### Brugergrænseflade i Unity3D

I dette projekt har det været nødvendigt at gøre brugergrænsefladen dynamisk til forskellige skærmstørrelser, hvilket har været i fokus under udviklingen af menusystemet. Systemet, som består af en række GUI-elementer, er bygget i Unity3D’s eget miljø vha. scripting. Miljøet minder meget HTML og ASP, men mangler noget funktionalitet i form af relative koordinater på skærmbilleder, som er under opsejling en ny opdatering.

Menuen, som i første omgang var udviklet ved brug af faste koordinator på skærmen samt faste pixelstørrelser, var ikke optimal til forskellige skærmstørrelser. For at overkomme dette problem er der gjort stort brug af script-funktionerne Screen.Height og Screen.Width, som henviser til skærmens længde og bredde. Er skærmen derfor 800x480, kan man finde midten af skærmen ved at skrive ’Screen.Width / 2’ i scriptet.

Et eksempel på en knap kan derfor se sådan ud:

GUILayout.Button("New Game / Continue", GUILayout.Height(Screen.height / 12));

GUILayout opretter et element, som automatisk bliver lagt i rækker på skærmen, som gør koden mere overskueligt. Derefter bliver der oprettet en knap via ’Button’, som indeholder følgende parametre, tekst, højde og vidde. Teksten henviser til den tekst, som vises på selve knappen på i brugergrænsefladen, hvor i dette tilfælde er ”New Game / Continue”, som er det første punkt hovedmenuen. De næste variabler, højde og vidde, henviser til størrelsen på selve knappen, hvor højden i dette stykke kode er skærmens højde divideret med 12. 12 er her valgt udelukkende ud fra, hvad der så passende ud fra udviklerens side. Vidden er ikke angivet her, da ’GUILayout’ automatisk henter højden og vidden fra parent-objektet, medmindre andet er angivet.

Det parent-objekt, som der hentydes til, kan bestå af flere elementer. I dette projekt er det funktionen, ’GUILayout.BeginArea’, som opretter et område, der kan redigeres. Det bliver eksempelvis brugt i scriptet, ’LevelSelect’, hvor det enkelte områder, som bliver oprettet indeholder en række knapper. Som ses på billedet, findes der 3 grupper i midten af grænsefladen, som hvert består af et ’BeginArea’. Hver enkelt har skærmkoordinater, som fortæller, hvor på skærmen, knappen skal sidde. Disse koordinator kan dermed gøres dynamiske, og derved opnå en større ensformighed i brugergrænsefladen og mere generisk kode.

Kigger man på koden bag en af disse områder, er det muligt at se mekanikken bag sammenhængen af område og knapper. Koden for et enkelt af disse områder ses her:

GUILayout.BeginArea(new Rect(Screen.width/4-Screen.width/8, Screen.height/2-80, Screen.width, 240));

GUILayout.BeginVertical("");

GUILayout.BeginHorizontal("");

GUILayout.Button("1-1", GUILayout.Height(Screen.height/10), GUILayout.Width(Screen.width/15));

GUILayout.Button("1-2", GUILayout.Height(Screen.height/10), GUILayout.Width(Screen.width/15));

GUILayout.Button("1-3", GUILayout.Height(Screen.height/10), GUILayout.Width(Screen.width/15));

GUILayout.EndHorizontal();

GUILayout.Button("Text!", GUILayout.Height(Screen.height/10), GUILayout.Width(Screen.width/5+5));

GUILayout.EndVertical();

GUILayout.EndArea();

Der startes med at oprette et område, som bliver tildelt dynamiske variabler, for at sikre det dynamiske layout. Derefter bliver der påbegyndt et funktion, GUILayout.BeginVertical, som starter en lodret række af elementer. Inde i den påbegyndes den tilsvarende vandrette funktion, som indkapsler 3 knapper, som nu arver ’BeginArea’’s koordinater og automatisk er sat i række via ’BeginVertical’ og ’BeginHorizontal’. Den vandrette række bliver afsluttet med ’EndHorizontal’, hvor efter endnu en knap oprettes, som nu ligger under den første vandrette række. Sidst afsluttes den lodrette række, hvorefter området afsluttes.

Der er altså blevet oprettet en række scener med tilhørende GUI, udviklet i Unity3D’s eget system. Systemet tager højde for skærmens størrelse og dimensioner, og det må dermed siges at opgaven er afsluttet.

## Design af brugergrænseflader

Når der designes grænseflader, er der en god mængde teori og designregler, som støtter udviklingen af disse. I dette projekt forsøges der at tage nogle af disse designregler i brug. Menuen er en grænseflade og dermed gælder der en række designregler, som bør overvejes under opbygningen. De første designprincipper, som bør indlejres i menuen er gestaltlovene. De fem gestaltlove, nærhed, lighed, lukkethed, forbundethed og figur og baggrund:

* Loven om nærhed beskriver, hvordan elementer, som står tæt, opfattes sammenhængende.
* Loven om lighed beskriver, hvordan elementer, som visuelt ligner hinanden, opfattes som sammenhængende.
* Loven om lukkethed beskriver, hvordan brug af rammer og bokse indikerer sammenhæng.
* Loven om forbundethed beskriver, hvordan elementer har sammenhæng ved brug af eksempelvis streger eller baggrundsfarve.
* Loven om figur og baggrund beskriver forskellen imellem, hvornår en figur opfattes som en figur og hvornår den opfattes som baggrund.