AMS projekt

Flappy bird

(Flappy bird but with a yellow block)

Gruppe 3

29 / 5 – 2020

Navn	AU-ID	Studie Nummer	Retning
Mathias Nortvig Thomassen	Au580521	201706287	IKT / SW
Valdemar Tang	Au586626	201707689	IKT / SW

Antal tegn m. mellemrum: 35333

Antal sider: 15

Ansvarsområder

På Tabel 1 nedenfor, ses fordelingen af ansvarsområder.

Ansvarsområder	Valdemar Tang	Mathias Thomassen
Touchdriver	X	X
TFTdriver	X	X
GameController	X	
CollisionDetection		X
CheckScore		X
PhysicsEngine	½X	X
FontGenerator og Text	X	
Test	X	X

Tabel 1 - Illustrerer fordelingen af ansvarsområder

Indhold

Indledning

Gruppen kunne godt tænke sig at prøve at udvikle et spil vha. embedded programmering. Hertil anvendes en Mega2560 microcontroller, samt et TFT LCD Display hvorpå spillet vises. Displayet har desuden touch-funktionalitet, som spilleren kan bruge til at interagere med spillet. Til at udvikle spillet vil pensum for AMS-kurset blive taget i brug, herunder implementeringen af drivers, som skal kunne håndtere tidskritiske parametre. Dette vil dække over en touchdriver, samt udvidelse af TFT-driver fra kursets opgave i lektion 3 omkring grafiske displays. [(5)]

Denne rapport indeholder nogle krav, som vi selv har stillet til produktet, samt analyser af vigtige valg. Derudover præsenteres tankerne bag arkitektur, design og implementering også. Dette indebærer beskrivelser og illustrationer af vigtig funktionalitet som er essentiel for at kunne forstå hvordan spillet virker og er implementeret. Der vil blive lagt mest vægt på TouchDriveren, da det er dette område som ikke er blevet undervist i, i kurset. De ting som der er blevet undervist i igennem kurset, vil kun blive beskrevet kort da dette antages for at være kendt viden.

Læsevejledning

Referencer

Referencer markeres med et lille ophævet tal i parentes: [(1)]. Disse tal refererer til et nummereret element i referencelisten sidst i rapporten. Ved nogle referencer vil der være et angivet et sidetal som følger: [(1, s 125)]. Sidetallet angiver den side i dokumentet som der refereres til.

Forkortelser

I Tabel 2 ses en oversigt over de anvendte forkortelser og navne, samt deres betydning.

Forkortelse	Betydning
TFT	Thin-film-transistor. Mere forklaring se ref[(19)]
Mega2560	Arduino Mega 2560 microcontroller
XPT2046	Touch Screen Controller placed on the ITDB02-3.2 v2[(3)]
ILI9341	TFT LCD Display Controller placed on the ITDB02-3.2 v2[(3)]
V SYNC	Vertikal synkronisering
Flappy	Flappybird (gule fugl i spillet).

Tabel 2: Tabel of forkortelser og deres betydning.

System- og applikationskrav

MoSCoW

Der laves en MoSCoW analyse[10] som bruges som en form for overordnet prioritering af krav. På denne måde identificerer vi de kernefunktionaliteter, som er nødvendige for at spillet kommer til at fungere.

Must have

- Skærmen skal opdateres minimum 10 gange i sekundet.
- Spilleren skal kunne interagere med spillet ved at trykke på skærmen.
- Spillet skal køre på en microcontroller med kompatibelt grafisk TFT-display.
- En Touch Driver, som kan læse hvor på skærmen spilleren trykker.
- En TFT driver, som kan vise grafiske elementer på TFT-display

Should have

- Bør kunne vise spillerens score mens spillet er i gang, samt opdatere scoren løbende.
- Bør have en menu med tilhørende knapper.
- Bør have en Game-Over skærm med tilhørende knapper og visning af score og high-score.

Could have

- Gemme highscore i statisk hukommelse.
- Sværhedsgrad forøges baseret på spillerens nuværende score.

Wont have

- Detaljeret grafik.

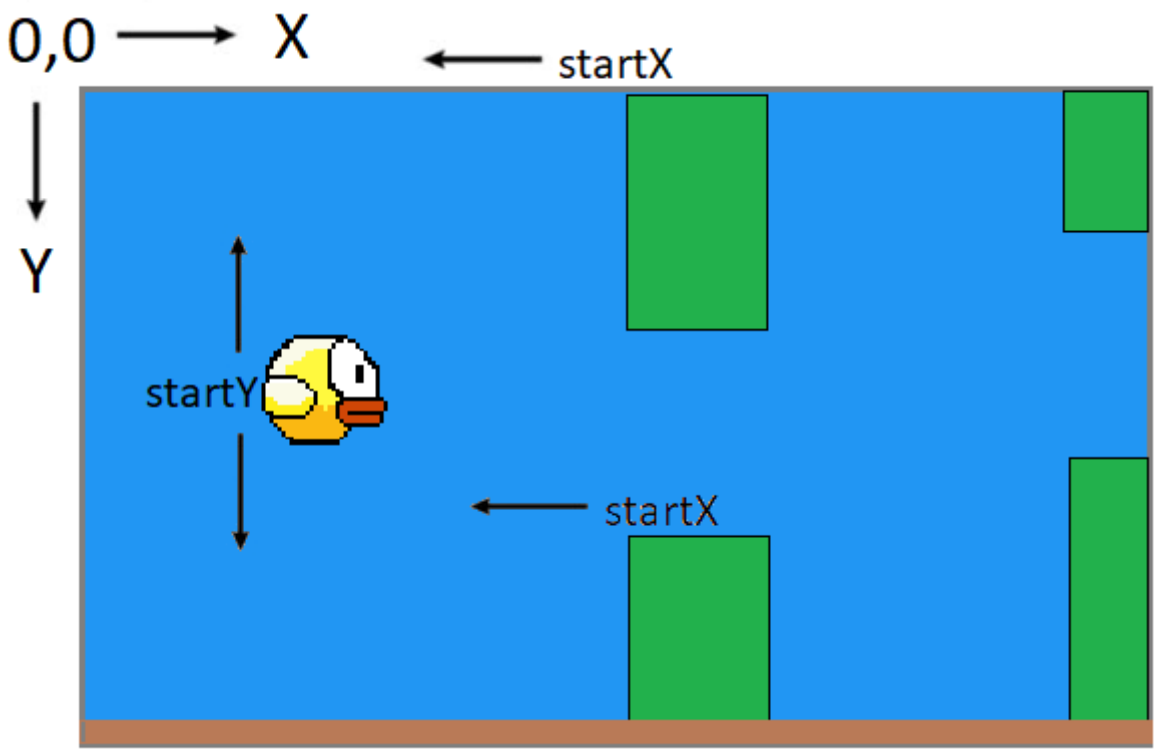
Must-haves vil blive prioriteret højest og bør implementeres før Should-have, som igen bør blive implementeret før Could-have osv.

Projekt oversigt

Spillet som projektet forsøger at efterligne, hedder Flappy Bird[15], og er et spil, hvor en fugl skal flyve igennem nogle søjle-par for at score point. Et søjlepar består af én søjle for oven og én for neden, hvor der er et hul i midten, som fuglen skal flyve igennem. Hver gang fuglen har passeret et søjlepar, får spilleren ét point. Spillet er et såkaldt side-scrolling spil, hvilket vil sige at man ikke kan styre den horisontale position på fuglen, men at det i stedet er søjlerne der rykker sig. Man kan derimod styre den vertikale position af fuglen, hvilket man gør ved at trykke et vilkårligt sted på skærmen, hvorefter fuglen flyver opad. Der skal desuden være en

tyngdekraft, som trækker fuglen mod bunden af skærmen med gradvis mere kraft hvis brugeren ikke trykker på skærmen.

Der laves et mockup af, hvordan spillet gerne skulle se ud når det er færdigt. Billedet kan ses på Figur 1.



Figur 1 Billede der viser spillet, samt hvordan de forskellige akser er orienteret.

Analyse

Dette afsnit indeholder analyser for forskellige aspekter af projektet. Dette indebærer indledende valg inden projekt start, samt essentielle valg undervejs i processen af projekt arbejdet.

Valg af microcontroller

Spillet laves på en ATMega2560 microcontroller[14]. Valget er baseret på, at denne microcontroller er blevet benyttet i undervisningen, samt at vi havde i forvejen godt kendskab til denne fra et tidligere kursus: MSYS[16]. Andre muligheder kunne have været en raspberry pi microcontroller med tilhørende LCD-skærm[11], som vi også har arbejdet med før. I det tilfælde ville det grafiske aspekt måske have været nemmere at håndtere, da et Linux OS kunne være installeret, og programmer som qtCreator[12] kunne have været benyttet. Fordelen med mega2560, er det at det er nemmere at håndtere optimering af ressourcer på hardware niveau med drivers. Desuden var det, grundet situationen, svært at få fat i andet hardware, hvilket også medvirkede til

valget af Mega2560. Der blev desuden undersøgt om det overhovedet kunne lade sige gøre at implementere spillet, og det viste sig at flere andre personer havde implementeret et lignende spil på denne Microcontroller.

Touch

TouchControlleren XPT2046, har mulighed for 8- og 12-bits opløsning af touchinput. Vi vurderer, at denne controller kan bruges på baggrund af, at den understøtter en opløsning, som er større end opløsningen på displayet.

Det er ikke nødvendigt med en meget præcis driver til formålet, da spillet funktionalitet ikke er sensitiv overfor dette. Dog vil der muligvis være enkelte knapper som bliver anvendt til bl.a. at starte spillet med, så det skal alligevel være muligt at detektere nogenlunde hvor der er trykket på skærmen.

Kodesprog

Da vi havde med et spil at gøre, som skal indeholde flere forskellige elementer, så som søjler, en fugl, samt har flere forskellige "komponenter" valgte vi at bruge kodesproget C++, hvorved vi kunne få en mere objekt-orienteret kode. Dette vil være en fordel for at kunne få mere struktur på koden, så den bliver nemmere at læse, implementere og fejlfinde. Atmel Studio[17] vil blive benyttet som IDE, samt til at håndtere overførelsen af programfiler til Mega2560. Der anvendes en AVR8 GNU c++ 5.4 kompiler.

Opdaterings frekvens

For at spillet er spilbart vurderer vi, at vi skal have en minimum opdaterings frekvens på omkring 10 Hz. Displayet skal naturligvis kunne understøtte dette, og da den understøtter opdaterings frekvenser på op til 119 Hz[2, s 155], kan den bruges. En højere opdaterings hastighed end 10Hz vil være fordelagtigt, for at spillet bliver mere flydende og reagerer bedre på input.

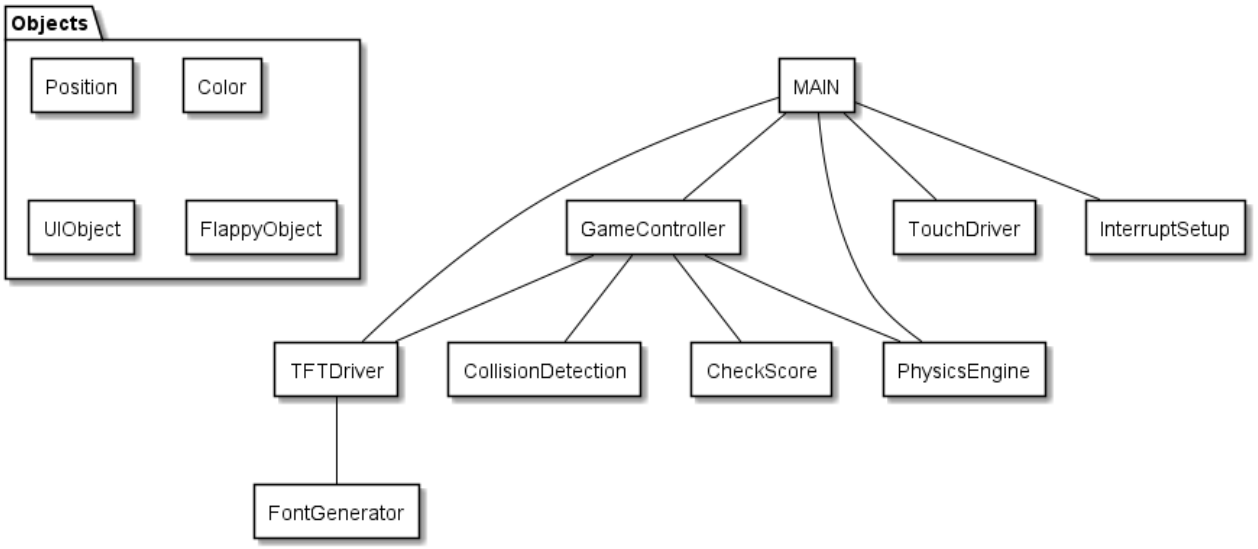
Materialeliste

Nedenfor ses en liste, af de moduler, som var nødvendige at benytte, for at dette projekt kunne laves.
Valget af moduler er også baseret på analysen.

- Arduino Mega2560 board
- ITDB02 Arduino Mega Shield[4]
- ITDB02 TFT display modul v2. [3]

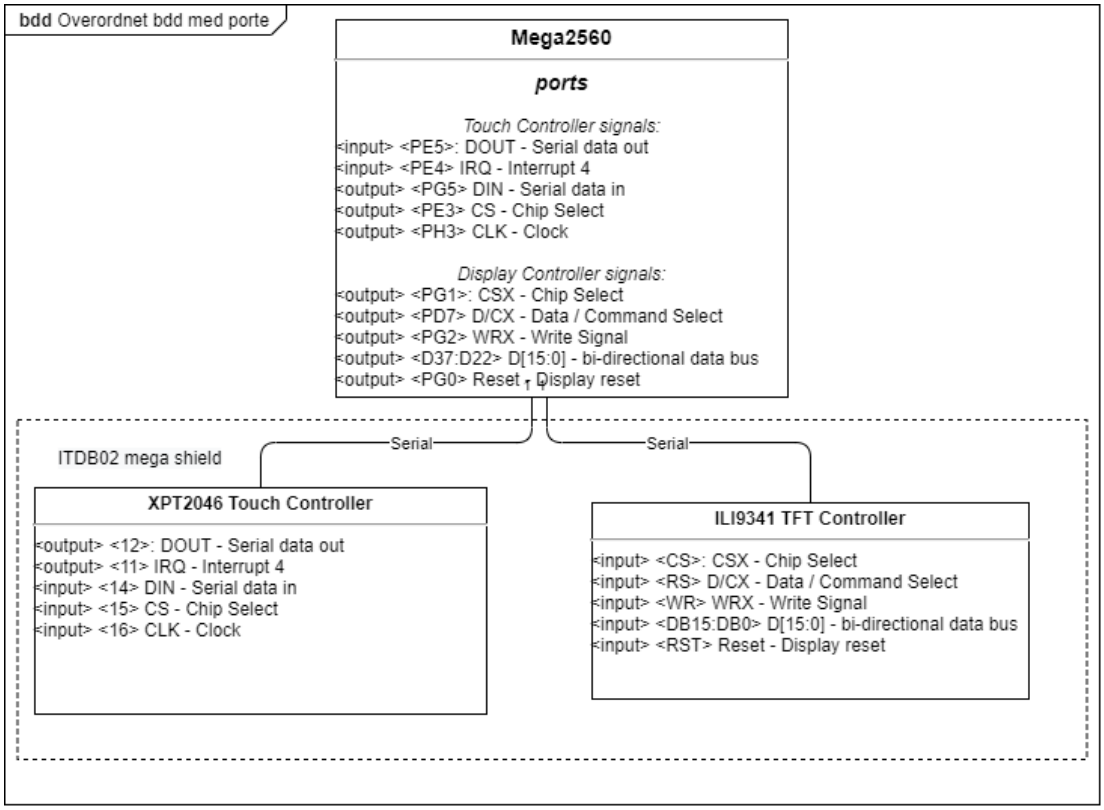
Systemarkitektur

Der laves en domænemodel for hele systemet for at give et overblik, samt at opdele forskellige ansvarsområder ud i forskellige klasser for at forsøge at overholde SRP[13]. Domænemodellen kan ses på Figur 2.



Figur 2: Domæne model for spillet. Grunden til at MAIN skal kende til TFTDriver og PhysicsEngine er at disse gives med i contructoren for GameController. Konfigurationen af de forskellige afhængigheder for GameController kan derved ske ude fra, ved brug af dependency injection.

For at illustrere hardwaresignalerne mellem de forskellige moduler, herunder microcontrolleren og de to andre kontrollere, er et BDD-diagram udarbejdet. Dette kan ses nedenfor på Figur 3.

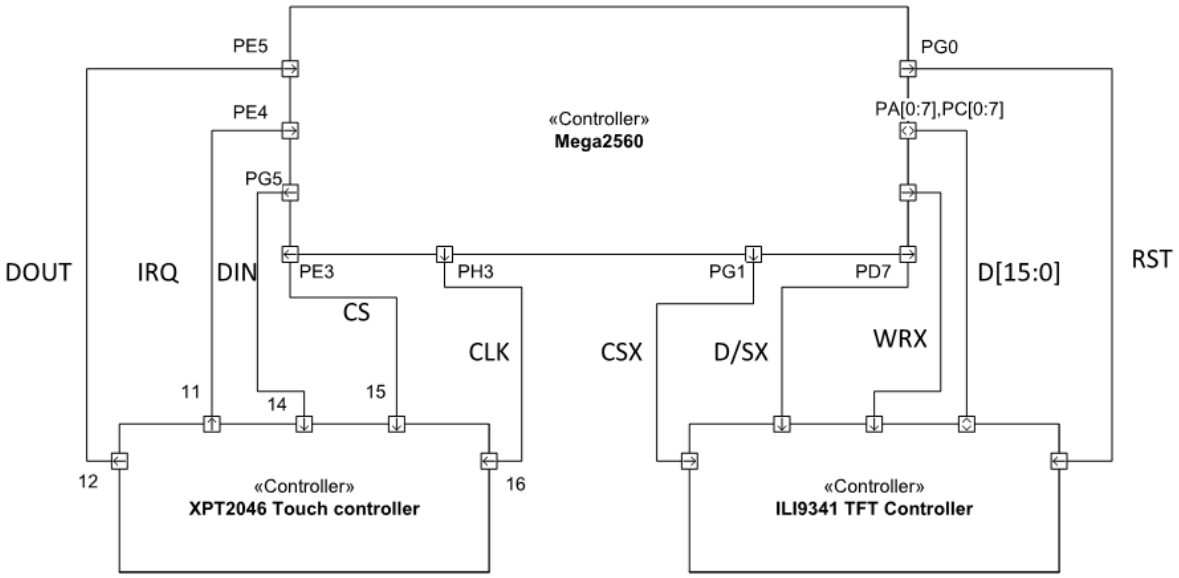


Figur 3: BDD, som illustrerer hardwarekomponenterne nødvendige for at spillet virker. Samt tilhørende porte skrevet i blokkene. Her er det vigtigt at pointere, at VCC og ground ikke er skrevet på, selvom de stadig er der.

Signalerne vil blive uddybet i næste afsnit om Forbindelser, hvorefter brugen af signalbusserne vil blive beskrevet i afsnittet efter Tabel 3.

Forbindelser

Ud fra BDD-diagrammet på Figur 3 har det været muligt at udarbejde et IBD-diagram, som visuelt viser forbindelserne mellem Mega2560 og de to andre controllere. IBD-diagrammet kan ses nedenfor på Figur 4.



Figur 4: IBD over systemet med tilhørende signaler

Ud fra IBD-diagrammet på Figur 4, er der udarbejdet en tabel med signal beskrivelser som kan ses nedenfor i Tabel 3.

Signaler	Beskrivelse
DOUT	DOUT er data output fra XPT2046, som mega2560 microcontrolleren kan læse på. Dataene bliver sendt serielt, og bliver skrevet ud, når CLK-signalet er en nedadgående kant.
IRQ	IRQ er et interrupt signal.
DIN	DIN er data input signal til XPT2046 controlleren, som mega2560 skriver til. DIN læses fra XPT2046 på CLK-signalets opadgående kant.
CS	CS er chip select input, som bruges til at signalere, hvornår en samtale starter og slukker. Den aktiveres ved at blive sat lav.
CLK	CLK er et ekstern clock input til XPT2046, så den kan time læsningen og skrivningen af de andre signaler.
CSX	CSX er chip select input, som aktiveres ved at sætte signalet lavt.
D/CX	D/CX er data eller command signalet. Hvis signalet går højt, kommer der data ind, og hvis den går lavt, så kommer der kommandoer. Data og kommandoer bliver skrevet i D[15:0] signalet.
WRX	WRX er et write signal, som skriver data til ILI9341 controlleren, signalet bliver læst på den opadgående kant, når der pulses. Dette skal times med at ILI9341 skal læse fra D[15:0] på den opadgående kant.
D[15:0]	D[15:0] fungerer som parallelle databusser, som bliver benyttet til at skrive kommandoer eller data. Dette indebærer f.eks. RGB.
RST	RST er et reset signal.

Tabel 3: Tabel med beskrivelse af signalerne benyttet til kommunikation mellem modulerne

Protokoller

Der anvendes 2 forskellige protokoller for at kommunikere med XPT2046 Touch controlleren og ILI9346 TFT Controlleren. Disse to protokoller beskrives kort herunder, samt hvordan vi har implementeret dem og hvor man kan finde en mere udførlig beskrivelse af protokollen.

TouchDriver

For at kommunikere med TouchDriveren anvendes en SPI-lignende protokol, som beskrives i databladet for XPT2046 Touch controlleren[1, side 21]. Forkortelsen TC anvendes for Touch Controller i dette afsnit. Protokollen består overordnet set af 2 trin. Først sendes en control byte, som er en kommando, der fortæller TC, hvad den skal udføre. Control-byten indeholder 4 forskellige elementer, som kan bruges til at fortælle, hvilken information TC skal sende tilbage, samt et start bit. En oversigt over de forskellige dele kan ses i Tabel 4.

Bit	Navn	Beskrivelse
7	S	Start bit. Control byte markeres med en høj bit i starten.
6-4	A2-A0	Kanal selektion. Bestemmer hvilken koordinat der måles, samt hvilke drivers der er slået til. Beskrivelse af kombinationer kan findes på tabel 4 og 5 i XPT2046 datablad.
3	MODE	12/8 bit mode. Low : 12-bit, high: 8-bit.
2	SER / $\overline{DFR}$	Single-ended/Differential select bit.
1-0	PD1-PD0	Power down mode. Se tabel 8 i datablad.

Tabel 4 Beskrivelse af Control byte for XPT2046 Touch Driver protokol. Tabellens indhold er taget fra tabel 7 i databladet for XPt2046 controlleren og oversat. Nogen information er udeladt.

Control byten sendes først serielt på DIN, når Mega2560 genererer en tilhørende clock på CLK linjen. XPT2046 sætter herefter busy linjen høj, hvilket vi ikke har mulighed for at undersøge, da forbindelsen ikke går videre gennem shieldet til Mega2560. Vi skal derfor sørge for at overholde timingen som fremgår på figur 15 i databladet for TC. Efterfølgende genereres der endnu en gang en clock af Mega2560, hvor data clockes ud af TC. Her afhænger det af hvilken konverterings opløsning man vælger, hvor mange clock pulser man skal generere; 8 for 8-bit eller 16 for 12-bit.

TFTDriver

ILI9341 controlleren indeholder mange forskellige interfaces til at overføre data til skærmen. I øvelsen til implementering af en TFT-driver, som blev lavet tidligere i kurset, anvendes 16 bit MCU interface 1.[22, side 24] Da vi gerne vil genbruge så meget af driveren som muligt for at mindske udviklingstiden, undersøges det om dette interface kan anvendes. ILI9341 controlleren har et VSYNC-interface[2, side 52] som kan bruges til at styre opdateringshastigheden [2, s 52]. Dog har vi ikke mulighed for at tilgå dette signal gennem det shield som vi anvender, hvorved vi ikke selv kan styre opdateringshastigheden manuelt. Man kan dog sætte den interne opdaterings hastighed ved at skrive kommandoer til ILI9341. Opdaterings hastigheden er dog 70 Hz som

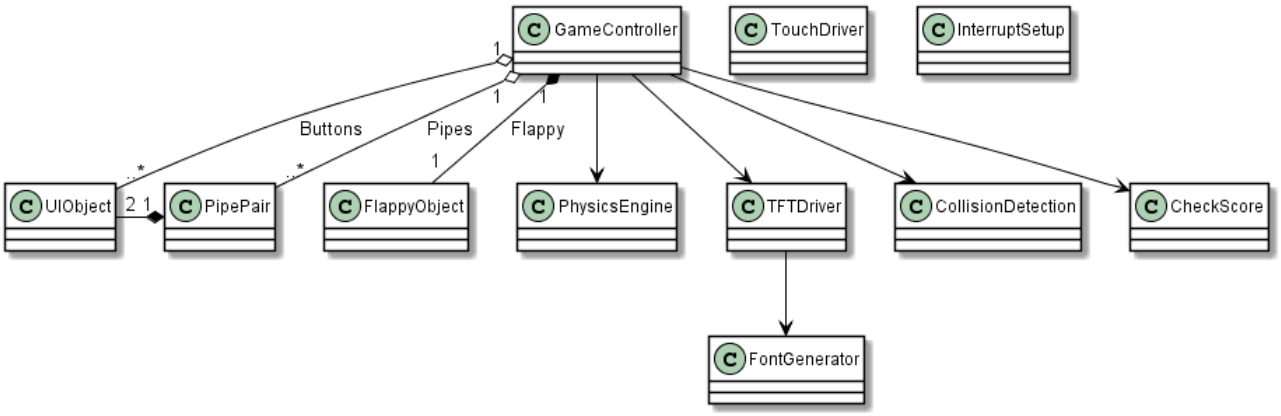
standard[2, s155] ,når MCU-interfacet anvendes. Dog skal vi så sikre os, at vores Mega2560 kan skrive data hurtigt nok for, at det faktisk svarer til, at skærmen opdateres med de 10 Hz. Vi kan finde den maksimale teoretiske opdaterings hastighed ved at udregne, hvor lang tid det tager at skrive én pixel, og dermed hvor lang tid det vil tage at skrive samtlige pixels på skærmen. De følgende skridt er krævet for at skrive data.

- Set up data = 3 cycles
- Set write port low = 1 cycle
- Wait = 1 cycle
- Disable write port = 1 cycle
- Wait = 1 cycle
- Outer foor loop = 1 cycle
- Function call = 1 cycle
- Total cycles = 9

Skærmen indeholder 240*320 = 76800 pixels. Da vores ATMega2560 har en clock frekvens på 16MHz[14], tager det den $\frac{76800*9}{16000000} = 0{,}0432\ sekunder$ at skrive data til hele skærmen. Det vil sige at vi kan opdatere hele skærmen $\frac{1}{0{,}0432} = 23{,}15$ gange i sekundet, hvis vi blot skrev data hele tiden. Til ovenstående kommer også setup af kolonne addresse, page adresse, kommandoer, opdatering af elementer i spillet mm. Dog er det helt klart at opdatere displayet ved ovenstående kommandoer som kommer til at tage længst tid. Vi kommer heller ikke til at opdatere hver pixel, men vil så vidt muligt kun opdatere de pixels der reelt bør opdateres i henhold til hvordan spillet udvikler sig. Det skal desuden pointeres at ovenstående ikke er en præcis udregning af hvad den maksimale opdaterings hastighed er men derimod et overslag så vi kan få en idé om vi kan overholde vores minimum krav på 10 Hz. Vi konkluderer, at clock frekvensen på Mega2560 er tilstrækkelig.

Design og implementering

Der udarbejdes et overordnet klassediagram med udgangspunkt i domænemodellen på Figur 2. Klassediagrammet kan ses på figur Figur 5. Her er det vigtigt at bemærke at GameController ikke har en kompositions relation til UIObject eller PipePair. Dette skyldes at antallet af PipePairs afhænger af størrelsen på skærmen og nogle andre parametre og er derfor dynamisk. Der blev forsøgt indledningsvist med dynamisk allokering men der blev så fundet ud af at det frarådes på microcontrollere efter at have snakket med underviser. Derfor allokeres hukommelsen til disse objekter i main, så de ikke går ud af scope og bliver destructet.

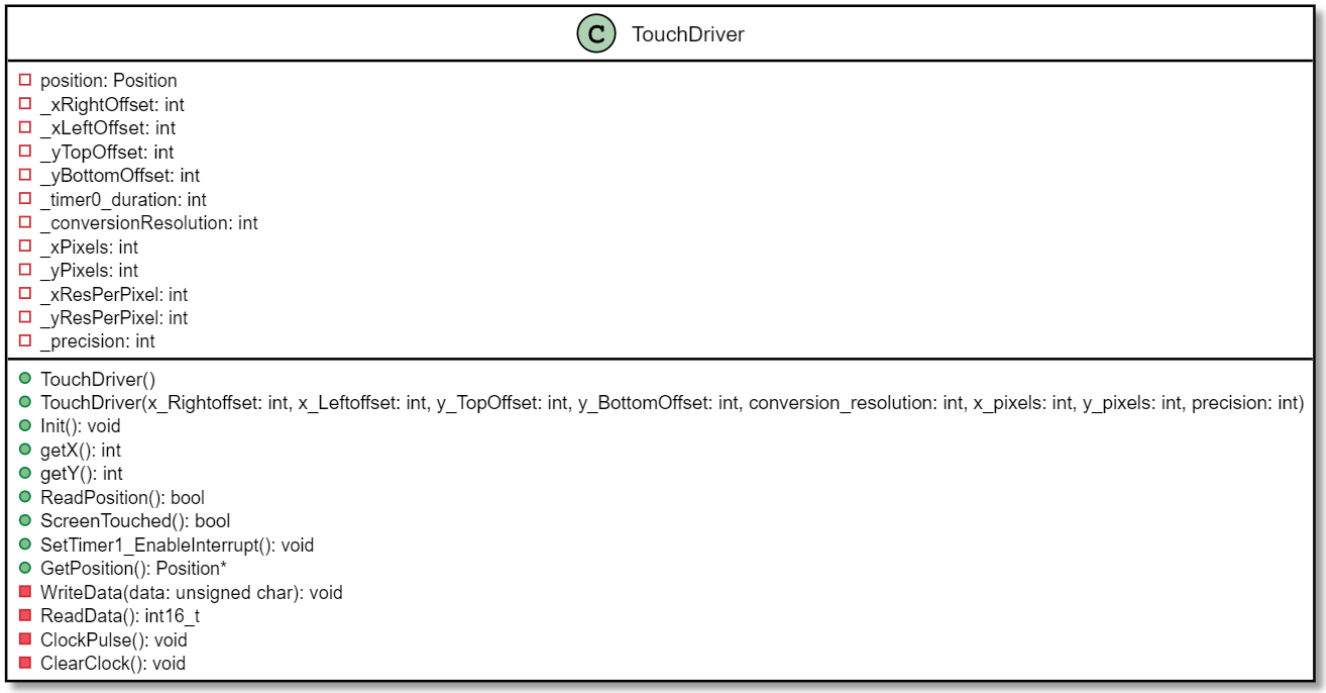


Figur 5: Overordnet klasse diagram for spillet. Main er ikke medtaget, da denne ikke er særligt interessant rent reference mæssigt. Main står blot for allokering af GameController, TouchDriver, PhysicsEngine, TFTDriver, FontGenerator, samt PipePair og UIObject arrays.

Touch Driver

Implementeringen af TouchDriveren vil blive beskrevet mere dybdegående end nogen af de andre ting som er lavet i kurset, hvilket skyldes at det er én af de dele som vi selv har undersøgt og implementeret. Herunder følger en kort beskrivelse af de vigtigste funktioner. Derefter vil der være nogle generelle design overvejelser, hvorefter der bliver beskrevet, hvordan det endte med at blive implementeret.

Nedenfor på Figur 6, vil det endelige klassediagram kunne ses for Touchdriveren. Ud fra dette diagram er det udvalgt nogle funktioner, som vil blive beskrevet i funktionsafsnittet længere nede.



Figur 6: Klassediagram for touchdriveren

Funktioner

bool ReadPosition()

Parameter: Ingen

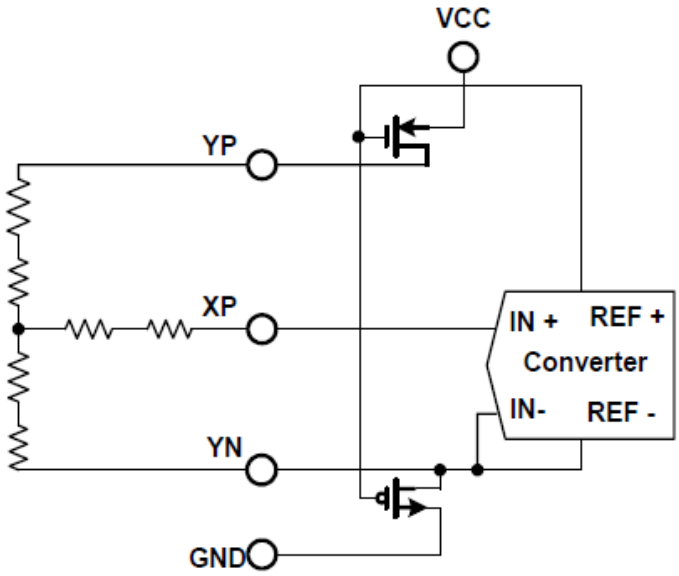
Returværdi: Boolean der indikerer om positionen er læst korrekt

Beskrivelse: Funktionen kaldes efter man har registreret at en bruger har trykket på skærmen. Funktionen skal da kommunikere med XPT-2046 controlleren, for at aflæse hvor den registrerer at brugeren har trykket på skærmen. Dette gøres ifølge protokollen[1, s15] af to omgange, én for x-koordinaten og én for y-koordinaten. Hver gang der læses undersøges der er registreret at brugeren har trykket på skærmen, (IRQ-ben lavt) hvorefter positionen modtages.

Virkemåde

Skærmen er en resistiv touchskærm hvilke vil sige at der er indbygget modstand i skærmen. Touchskærmen er opbygget af to lag af resistivt elektrisk ledende materiale, samt et mellemlag. Når brugeren trykker på skærmen, kommer de to resistive lag i kontakt med hinanden og man kan da finde ud af hvor brugeren har trykket ved at måle spændingen i det punkt. Et diagram fra databladet for XPT2046 kan ses på figur Figur 7, der viser hvordan det er sat op. Hvis man ønsker at måle y-positionen, slutter man YP til REF+ og YN til REF – , samt XP til indgangen på ADC'en. Da indgangs impedansen på ADC'en er meget høj, vil spændingen her være meget tæt på den samme som der hvor brugeren trykker og derved forbinder de to lag (X og Y). Y-laget

fungerer da som en spændingsdeler og derved måles Y-positionen. Det samme kan gøres for X positionen blot ved at bytte om på forbindelserne.



Figur 7: Diagram der viser hvordan positionen måles af ADC'en ratiometrisk måling som er det der anvendes i projektet. Taget fra datasheet for XPT2046 side 17.

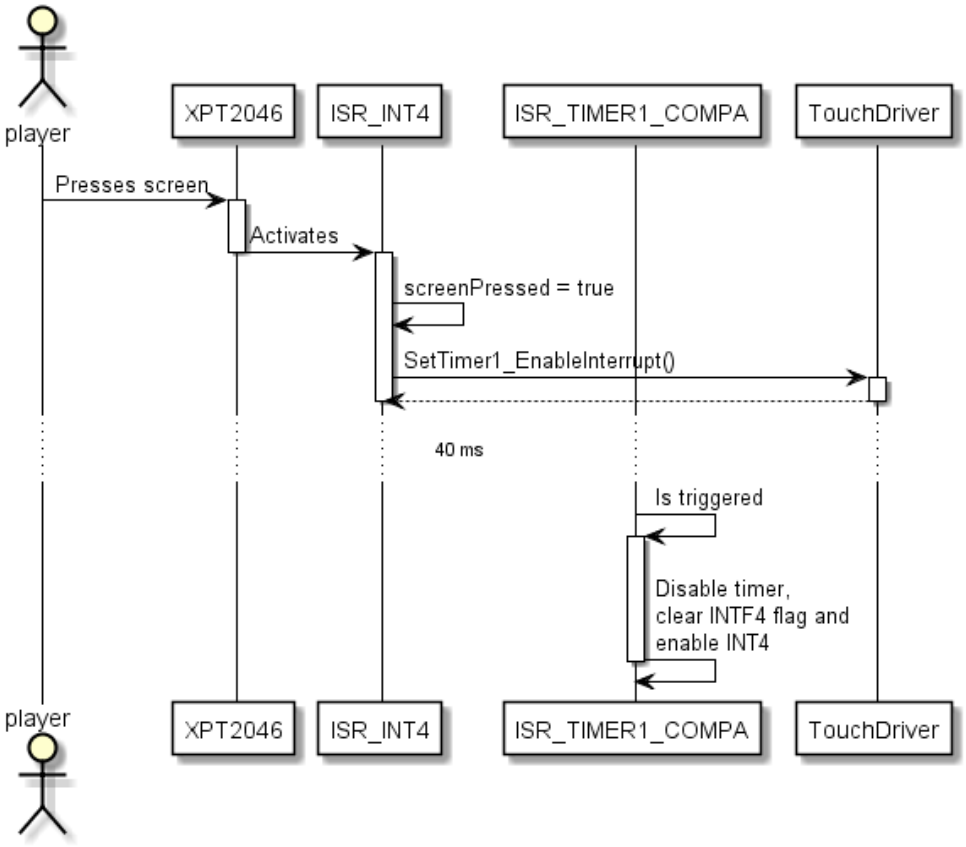
Med eller uden interrupt

Implementeringen med at bruge interrupt og uden at bruge interrupt, minder meget om hinanden. De følger overordnet den samme struktur som kan ses på sekvensdiagrammet på Figur 8, hvor den eneste forskel så vil være at der ved brug af interrupt sættes et flag i interrupt-rutinen, som bliver kaldt når IRQ-benet går lavt. Heri sættes screenPressed == true, hvor man derimod uden interrupt undersøger for hver iteration i while-løkken i main, om brugeren trykker på skærmen. Uden interrupt kan man altså være uheldig ikke at detektere at brugeren trykker på skærmen. Dette vurderer vi dog vil være yderst sjældent. Der er dog den ulempe ved ikke at bruge interrupt, at man faktisk ikke vil kunne detektere hvorvidt brugeren rent faktisk har trykket på skærmen, men kun om brugeren holder en finger på skærmen. Altså vil Flappy accelerere ved at man holder fingeren på skærmen, hvilket ikke er hensigtsmæssigt. Dette vil ikke ske på samme måde med interrupt, da den indikerer at brugeren trykker på skærmen efter at have taget fingeren af, hvis interruptet er sat til at trigger på falling edge, da IRQ-signalet er aktiv lav.

Valg af metode

Grundet vores manglende erfaring med at lave spil generelt, samt at det var længe siden vi havde arbejdet med et større embedded software projekt, besluttede vi os for at prøve os frem, for at se hvad der virkede bedst. Der blev vurderet at med interrupt var den bedste metode, da man da kunne undersøge om brugeren trykker på skærmen og ikke blot om der er en finger på skærmen. Vi sætter desuden en begrænsning på hvor

ofte man kan trykke på skærmen. Dette gøres ved brug af en timer. Et sekvensdiagram der beskriver hvordan dette implementeres, kan ses på Figur 8.



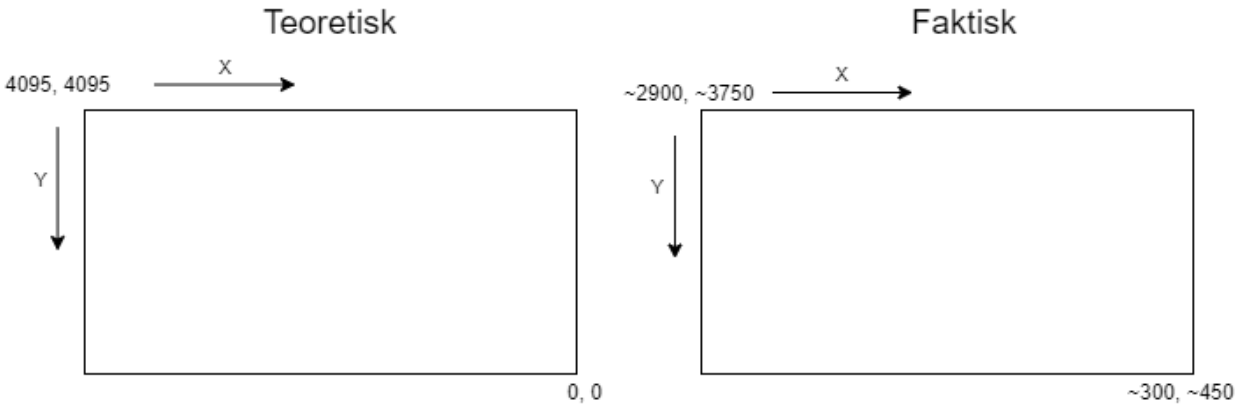
Figur 8 Sekvensdiagram der beskriver hvordan interruptet fra XPT2046 skal håndteres.

Reference spænding

Det er muligt at vælge mellem single-ended reference spænding eller differential reference spænding. I den differentielle mode, laves en ratiometrisk måling, hvilket ifølge databladet giver den mest præcise måling[1, s 22] og ikke behøver nogen ekstern reference. Vi vælger derfor denne type, da det er den mest simple og mest præcise. Det betyder at SER/$\overline{DFR}$ bitten altid er 0.

Kalibrering

Vi kalibrerer vores TouchDriver ved at sende de værdier som læses fra XPT2046 controlleren, via UART som modtages i en terminal på en PC. Det vi får ud, er værdien fra ADC'en som sidder på XPT2046 controlleren. På billedet nedenunder kan vi se den teoretisk mulige værdi for aflæsningen af data til venstre og den faktisk til højre.



Figur 9 Billede der viser hvad offsettet for X og Y er i højre og venstre side og top og bund. Ud fra denne information kan man kalibrere skærmen.

Denne information gives med i constructoren for TouchDriveren, hvorefter den efterfølgende udregner ADC opløsning pr pixel. Dette gøres ved brug af følgende formel:

$$ADCResPerPixel = \frac{ADC_{maxResolution} - offset1 - offset2}{pixels}$$

Her er offset1 og offset2, forskellen fra den teoretiske værdi for x eller y koordinat og pixels er antallet af pixels i x eller y retningen.

Når vi har læst data fra XPT2046 kan vi da udregne positionen ved brug af følgende formel

$$x = xPixels - \frac{(x_{read} - xLeftOffset)}{xResPerPixel}$$

Ved at udskifte x med y, og xLeftOffset med yTopOffset, fås formlen for beregning af y-koordinaten. Vi trækker den beregnede koordinat fra antallet af pixels på den givne akse for at invertere koordinaterne, så vores position kommer til at stemme overens med den orientering som anvendes i TFTDriveren.

Præcision

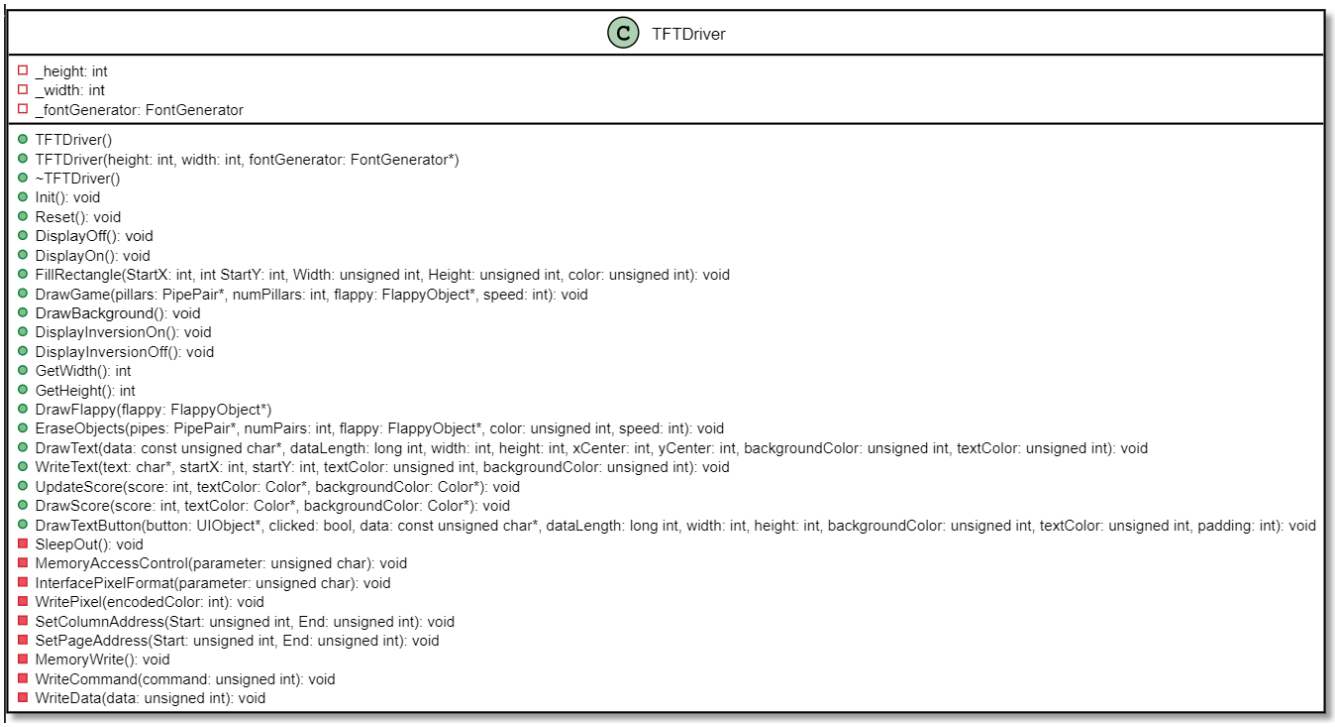
Det er muligt at vælge både 12 og 8 bit konvertering ved at sætte MODE bitten, se Tabel 4. Vælger vi 8 bit konvertering kan det højeste tal som vi får ud være 8^2 – 1 = 255. Da vi har en opløsning på 320 på x-aksen, vil outputtet fra XPT2046 have en mindre opløsning end skærmen. Spillet som vi implementerer, kræver ikke nogen meget nøjagtig præcision på noget tidspunkt, men samtidigt kommer vi ikke til at læse positionen på tidskritiske tidspunkter (mens vi spiller). Positionen aflæses kun når spillet er i menuen eller på game over skærmen. Vi vurderer derfor at vi godt kan anvende med 12-bit konvertering for at få en bedre præcision men bør lave driveren således at der nemt kan skiftes til 8 bit hvis der skulle blive behov i fremtiden.

Efter test på samme måde som da vi skulle finde ud af x- og yOffset, var det tydeligt at der nogle gange kom vidt forskellige værdier ud, også selvom man trykkede på det samme sted. For at løse dette problem og gøre aflæsningen mere præcis, læser vi koordinaterne flere gange, hvor antallet af gange bestemmes af "precision" parameteren i TouchDriver constructoren. Efter der er læst det opgivne antal gange, findes middelværdien af både x- og y-koordinatet, som giver den endelige position.

TFT Driver

TFT-Driveren skal håndtere opdatering TFT-displayet. Herunder ses en kort beskrivelse af de vigtigste funktioner, hvorefter diagrammer vil illustrere funktionsforløbet. Det er vigtigt at understrege, at nogle af de grundlæggende dele af TFTdriveren er lavet i forbindelse med undervisningen, og at der så blot er bygget videre på denne driver.

TFT-driveren kan ses illustreret ved et klassediagram nedenfor på Figur 10.



Figur 10: Klassediagram for TFTDriver. Det kan her ses, at mange af de oprindelige metoder er her, samt nogle nye, som DrawGame og Drawflappy etc.

Nogle af de vigtige metoder fra Figur 10, vil blive beskrevet nedenfor.

Funktioner

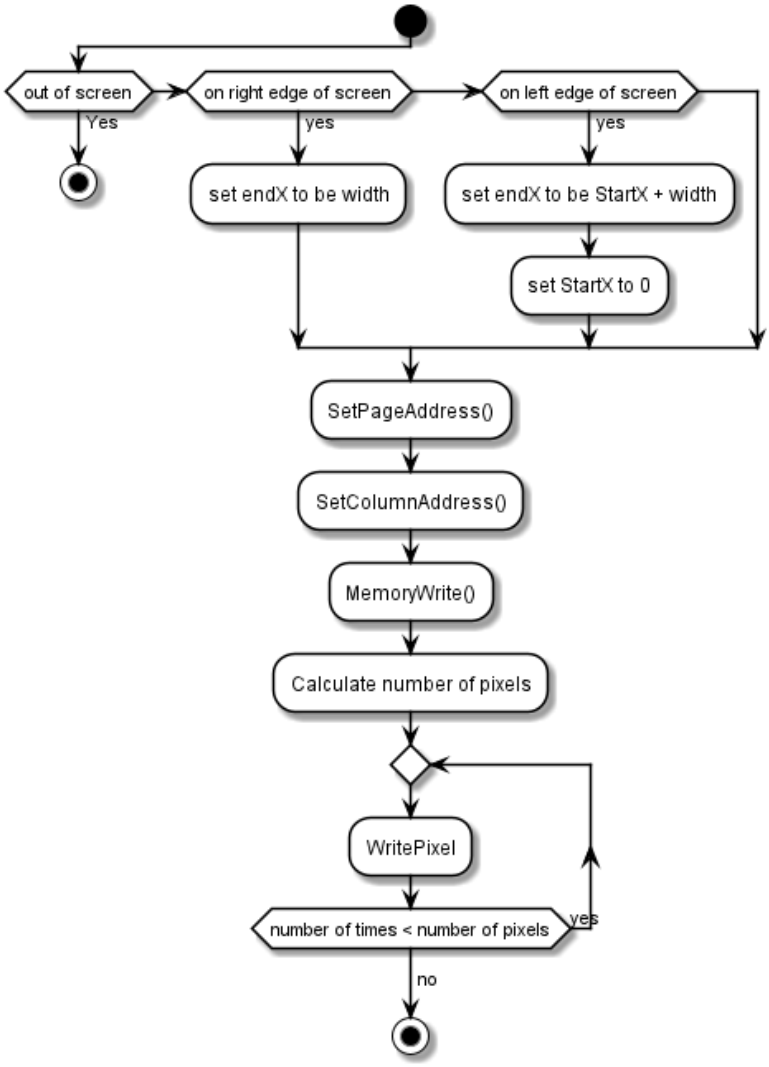
Void FillRectangle()

Parameter: X-koordinat, Y-koordinat, bredde, højde og farve. Husk X- og Y-koordinaterne tager udgangspunkt iFigur 1 Figur 1.

Returværdi: ingenting

Beskrivelse: Funktionen bliver kaldt, når et rektangel skal udskrives på skærmen ved bestemte koordinater og farve. Først så håndteres de forskellige scenarier af koordinater, hvor nogle af dem kan være uden for skærmens størrelse eller nogle af koordinaterne kan være uden for skærmen. Efter dette så skal PageAddress og ColumnAddress sættes, så ILI9341 ved, hvor på skærmen rektanglet skal placeres. Hertil skal der skiftes til MemoryWrite, så dataene kan sendes. Efter dette itereres der igennem hver pixel på rektanglet, hvorved der skal skrives en farve til hver pixel.

Funktionen kan ses illustreret ved et aktivitetsdiagram på Figur 11 .



Figur 11: Aktivitetsdiagram, som viser flowet i FillRectangle funktionen i TFT-driveren

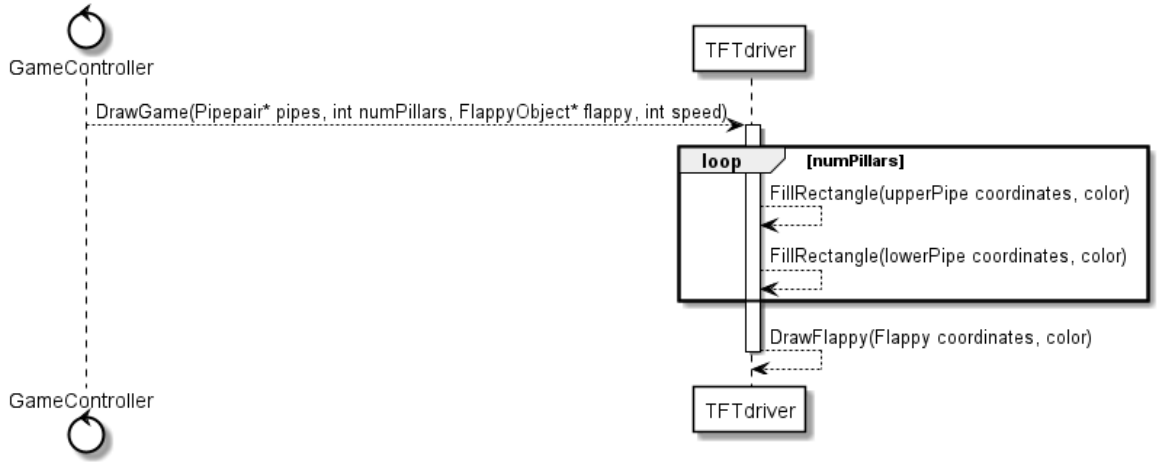
Void DrawGame()

Parameter: PipePair objekter indeholdende 2 pipes, antal pipes i første parameter. FlappyObject objekt indeholdende fuglen. spilhastigheden, som indikerer antal pixels en pipes bevæger sig pr. frame

Returværdi: Ingen

Beskrivelse: Funktionen bliver kaldt efter man har trykket på (start game)-knappen i menuen, og efter hver frame i spillet. Funktionen skal sørge for at oprette de PipePairs, som er givet i parameter på de korrekte koordinater. Derudover skal den sørge for at oprette Flappy.

Funktionen kan ses illustreret ved et sekvensdiagram på Figur 12.



Figur 12: Sekvensdiagram, som viser DrawGame funktionen i TFT-driveren

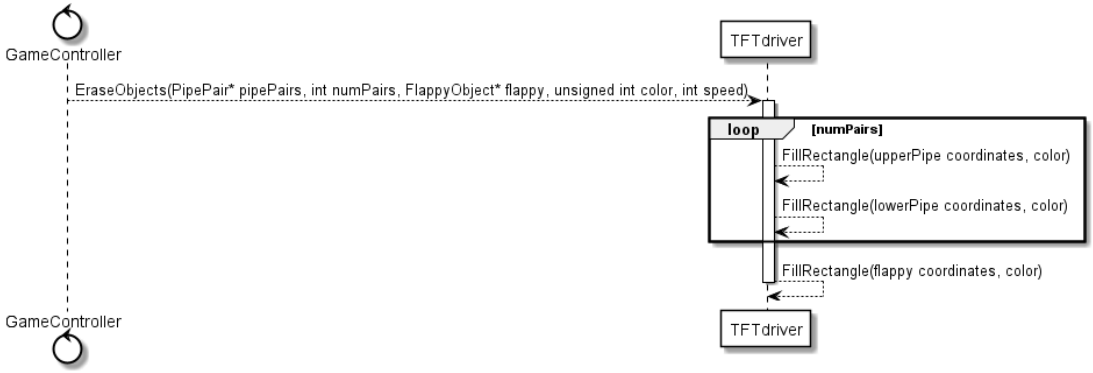
Void EraseObjects()

Parameter: PipePair objekter indeholdende 2 pipes, antal pipes i første parameter, FlappyObject objekt indeholdende fuglen, en farvekode, som bliver placeret hvor objekter fjernes, spilhastigheden, som indikerer antal pixels en pipe bevæger sig pr. frame

Returværdi: Ingen

Beskrivelse: Funktionen kaldes efter hver frame. Funktionen skal sørge for at slette de oprettede objekter på skærmen ved at overskrive dem med en farve.

Funktionen kan ses illustreret ved et sekvensdiagram på Figur 13.



Figur 13: Sekvensdiagram, som viser EraseObjects funktionen i TFT-driveren

Generering af grafik

Til generering af grafik blev der anvendt programmet LCD Font Maker, hvori der kan vælges en font, hvorefter man kan skrive tekst ind. Ud fra dette genereres en bitmap fil som gemmes og efterfølgende importeres i et andet program: LCDAssistant. Dette program konverterer bitmap filen til et unsigned char array. For at spare på RAM anvendes et format hvor hver bit repræsenterer én pixel. Da det er tekst der som udgangspunkt skrives, vil man kunne vælge en baggrunds farve, samt en tekst farve. Hvis vi anvender rød som tekst farve og grøn som baggrundsfarve får vi resultat for "10110010" som kan ses på tabel

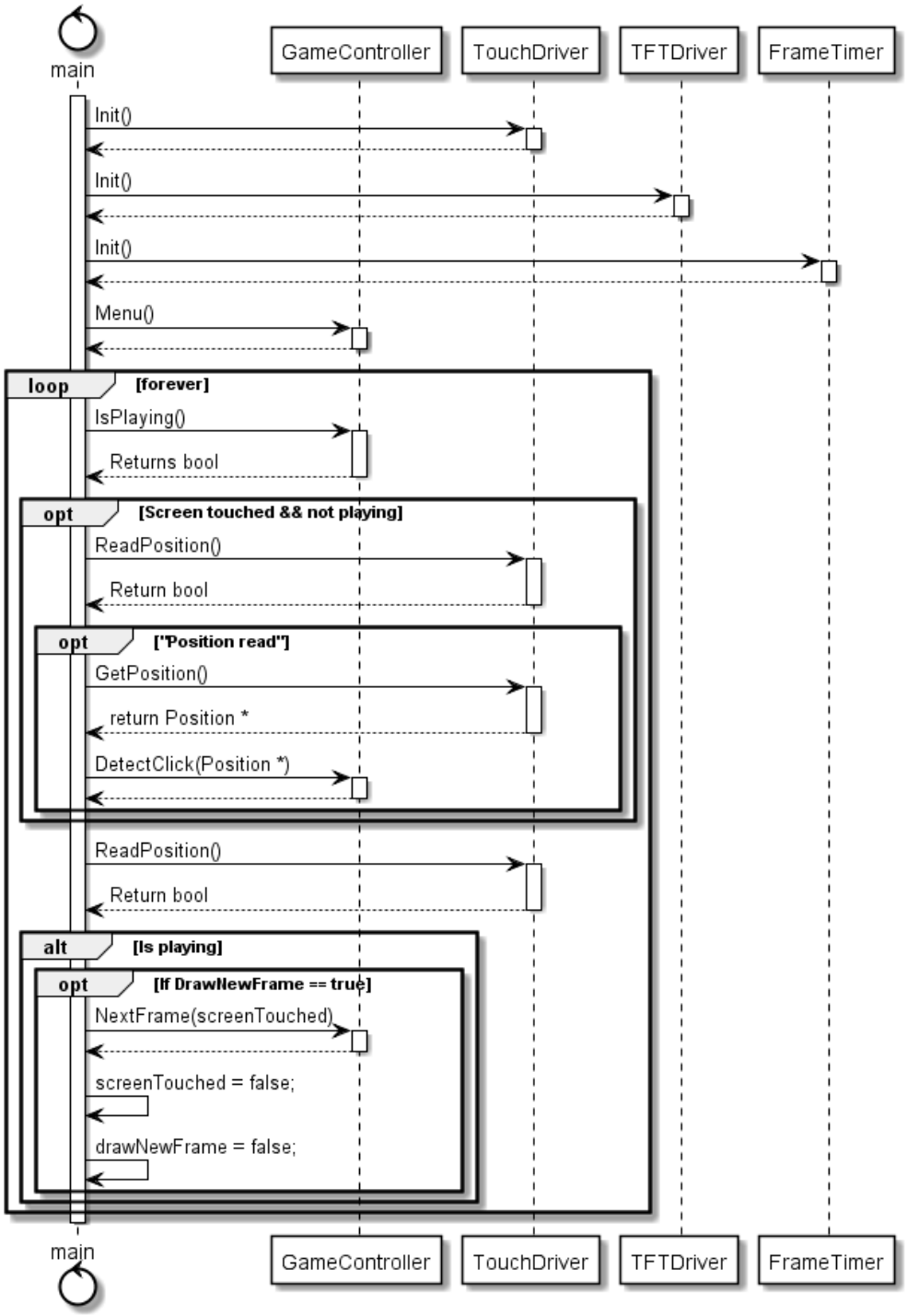
Bit	7	6	5	4	3	2	1	0
Værdi	1	0	1	1	0	0	1	0
Farve								

Der laves hard-codede arrays ved brug af denne metode i stedet for at lave hvert enkelt bogstav som et separat array, hvortil der kunne lave en opslagsfunktion der tog en streng og returnerede et sammenhængende array af data svarende til teksten i inputtet. Dette tager lang tid, da alle karakterer skal konverteres, og kommer til at fylde mere i hukommelsen, end hvis vi blot hard-coder de få tekst-strenge, som vi skal bruge. Arrays indeholdende data for de hardcoded strenge kan findes i filen Text.h . Der laves dog en opslagsfunktion for tallene fra 0-9 som bruges til at hente data til at vise scoren med. Dette implementeres i FontGenerator klassen.

Main

Main funktionen skal stå for den ydre logik i spillet. Den håndterer bl.a. input fra touch-skærmen, samt kalder void NextFrame(bool) funktionen hver gang der skal tegnes en ny Frame. Et sekvensdiagram der beskriver forløbet i koden kan ses på Figur 14. Main filen indeholder desuden en interrupt rutine som kaldes når timeren der styrer opdateringshastigheden, rammer en bestemt væ. Den sætter *drawNewFrame* flaget til true, hvorefter skærmen opdateres med *void NextFrame()* kaldet til *GameController* klassen ved næste loop i while-løkken. Det er her vigtigt at bemærke at positionen på brugerens tryk på skærmen ikke læses hvis

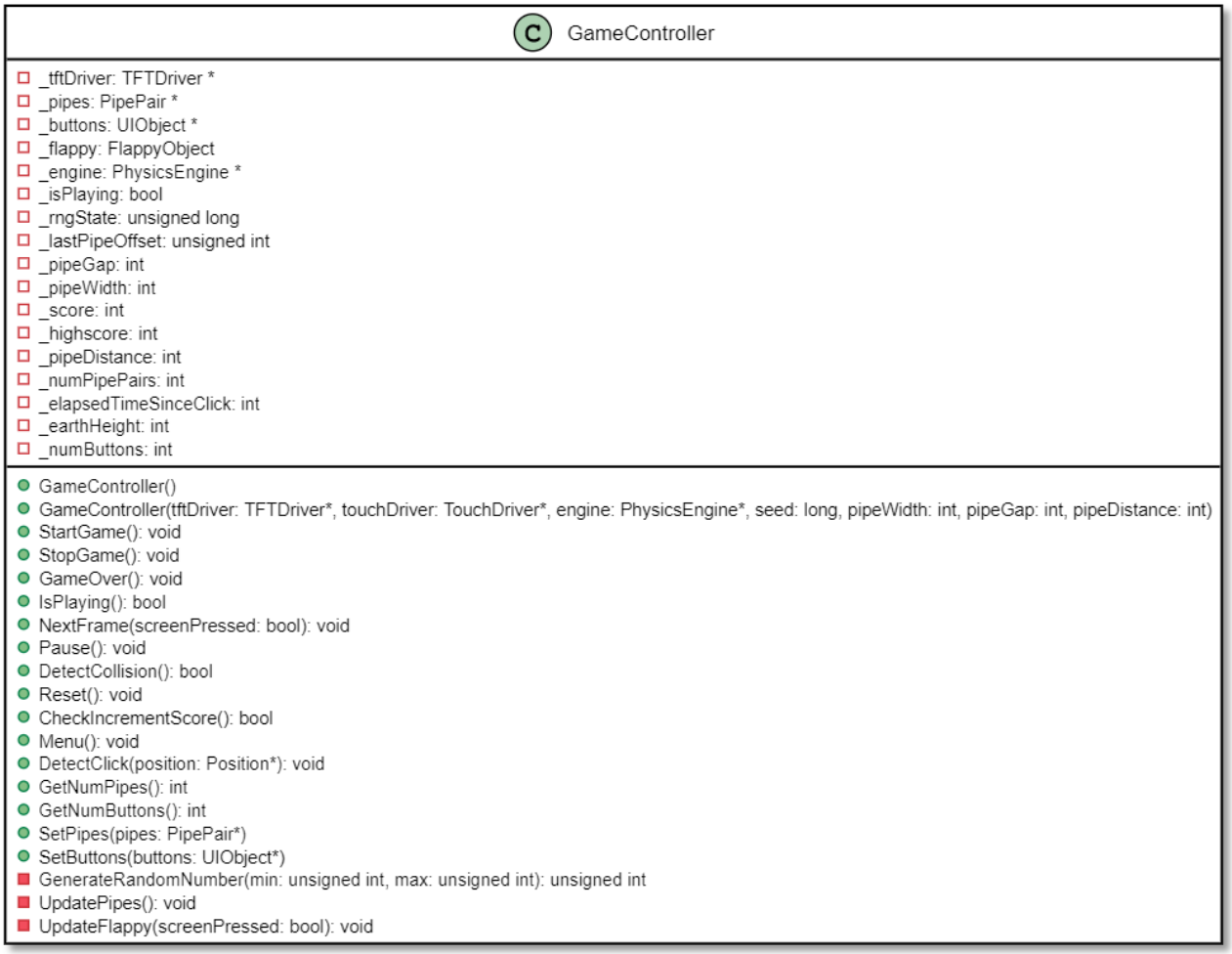
spillet er i gang. Dette er der ikke brug for da spillet blot har brug for at vide om brugeren har trykket på skærmen eller ikke, og det sparer os for at afvikle unødvendig kode. Her vises ikke de interne funktions kald for GameController til bl.a. TFTDriveren, callback fra UIObject mm. for at give et mere overskueligt diagram som har mere fokus på forløbet af koden i main.



Figur 14 Sekvens diagram der beskriver forløbet i main.

Game Controller

GameControlleren er den klart største og vigtigste klasse som står for selve håndteringen og alt logikken i spillet. Den skal således blandt andet undersøge om fuglen kolliderer med nogle af søjlerne, anvende TFTDriveren til at vise en Menu, samt "Game Over" hvis en kollision er detekteret. Et klassediagram for Game Controlleren er udarbejdet og kan ses nedenfor på Figur 15.



Figur 15: Klassediagram for Game Controlleren, som er den klasse, der skal styre spillet

Én af de vigtigste funktioner er *void NextFrame()* som står for at opdatere spillet hver gang der skal tegnes en ny frame. Denne beskrives nedenfor.

Void NextFrame()

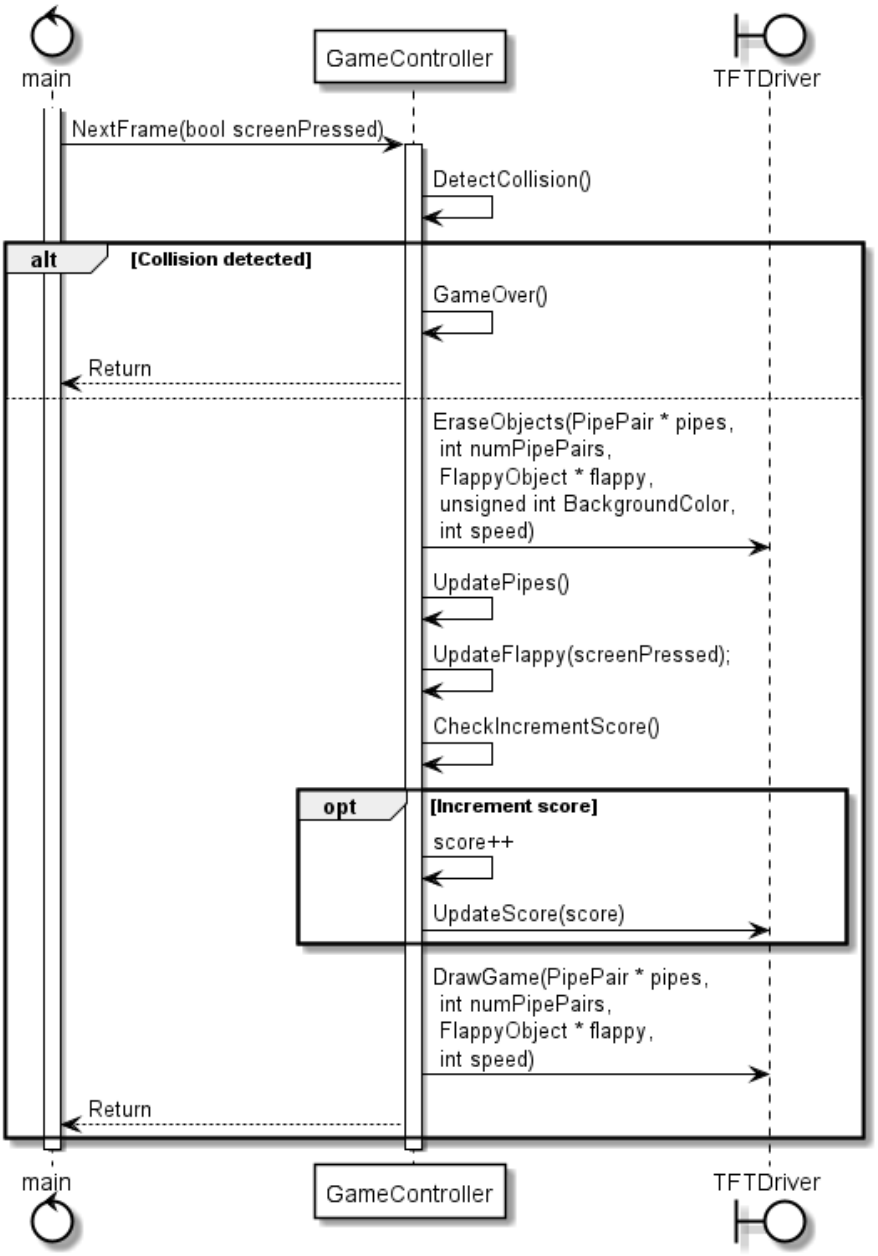
Parameter: bool ScreenPressed

Returværdi: ingenting

Beskrivelse: Funktionen bliver kaldt mellem hver frame i spillet, og bruges til at klargøre den kommende frame. Først skal den tjekke hvorvidt der er sket en kollision mellem Flappy og en pipe. Hvis dette er tilfældet,

så er spillet slut, og GameOver kaldes. Hvis der ikke er detekteret en kollision, så skal EraseObjects() kaldes. Efter at de nødvendige pixels er opdateret, så skal koordinaterne til de nye pipes og flappy opdateres. Efter dette, så tjekkes der så om scoren kan incrementeres, altså om flappybird har passeret en pipe uden at kollidere med den.

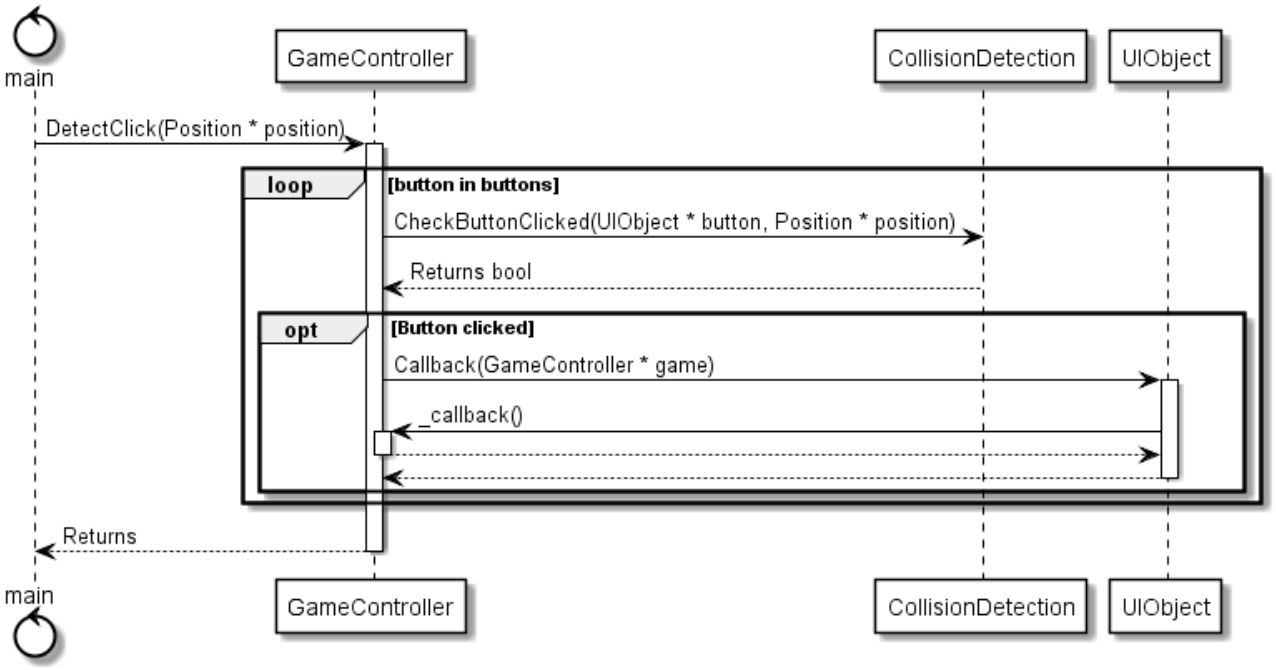
Nedenfor på Figur 16 ses et sekvensdiagram, som illustrerer NextFrame funktionen.



Figur 16: Sekvensdiagram som viser NextFrame Funktionen i GameControlleren

Tryk på knapper

Spillet kommer til at indeholde én eller flere knapper, afhængig af hvor langt vi når med implementeringen. Det vil derfor være en god idé implementere noget kode som gør at knappen kan genbruges. Vi vælger klassen UIObject, hvor der tilføjes en callback til en member-funktion i GameController, som så kan kaldes, hvis der detekteres at der trykkes på en knap. Denne funktion kan nu kaldes på GameController objectet, når der detekteres at der er trykket på knappen. Et sekvensdiagram der beskriver hvordan der undersøges for om der er trykket på knapperne og callback sekvensen, kan ses på Figur 17.



Figur 17 Sekvensdiagram der beskriver hvordan der detekteres om der er trykket på en knap, samt hvordan callback funktionen for UIObject fungerer.

Styring af opdaterings hastighed.

For at styre opdateringshastigheden er der blevet forsøgt med flere forskellige metoder. Vi havde til at starte med udtænkt to forskellige måder som vi kunne implementere opdatering af skærmen på. En kort beskrivelse af deres virkemåde bliver forklaret herunder hvorefter valget af metode begrundes.

Fast eller variabel opdaterings frekvens.

Ved en fast opdaterings frekvens forstås det at opdaterings frekvensen er konstant og ikke vil ændre sig hvis vi fx havde en kraftigere microcontroller. Hvis opdaterings frekvensen er fast, skal PhysicsEngine klassen kende opdateringsfrekvensen så den kan beregne hvor meget et givent element skal flytte sig for hver frame.

Anvender vi derimod en variabel opdaterings frekvens er vi ikke afhængige af et interrupt. Implementeringen af en variabel opdaterings frekvens er derimod en smule anderledes, hvad angår PhysicsEngine, samt hvor langt et PipePair flytter sig per frame. For at kunne beregne hvor langt de forskellige elementer flytter sig, er vi nødt til at indføre et tidskoncept, eller sagt med andre ord skal de funktioner der beregner hvor langt hvert element flytter sig, kende tiden siden sidste frame. Der kan således beregnes hvor langt hvert element skal flytte sig. Hvis ikke vi havde dette tidskoncept, ville spillet bevæge sig hurtigere når der var færre elementer på skærmen (mindre data at udregne og skrive til TFT Displayet), hvilket resulterer i en højere opdaterings frekvens . Med en variable opdateringsfrekvens kan vi udnytte processorens kapacitet fuldt ud. Dette tidselement kan implementeres på flere måder, hvoraf der her nævnes 3 metoder som vi kunne prøve i en fremtidig implementering.

1. Implementér egen counter ved brug af Timer interrupt på Mega2560
2. Ekstern HW timer
3. FreeRTOS xTaskGetTickCount()[7, s41]

Valg af metode

Vi vælger i første omgang at implementere det med en fast opdaterings frekvens, da det er klart det mest simple, og vi vurderer, at vi kan få et nogenlunde resultat ud af dette. Dette gør, at vi hurtigere kan få implementeret andre essentielle dele, som gør spillet spilbart. Dog ville implementeringen med en variabel opdaterings frekvens helt klart være at foretrække på den lange bane, da vi på denne måde får mest muligt ud af vores Microcontroller.

Utility klasser

For at holde en ordentlig struktur i koden, og opretholde en objekt-orienteret tilgang til koden, så har vi benyttet os af en masse hjælpeklasser, som både indeholder data-objekter, men også metoder til udregning etc.

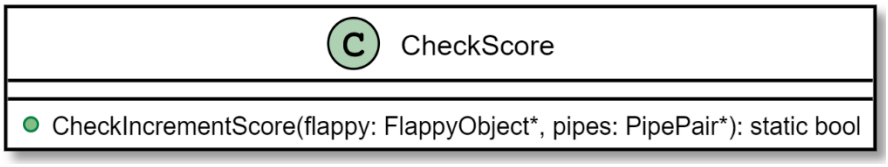
Utility klasserne i dette projekt består af:

- CheckScore
- CollisionDetection
- UIObjects
- PipePair
- FlappyObject
- Color
- FontGenerator

- Position

CheckScore

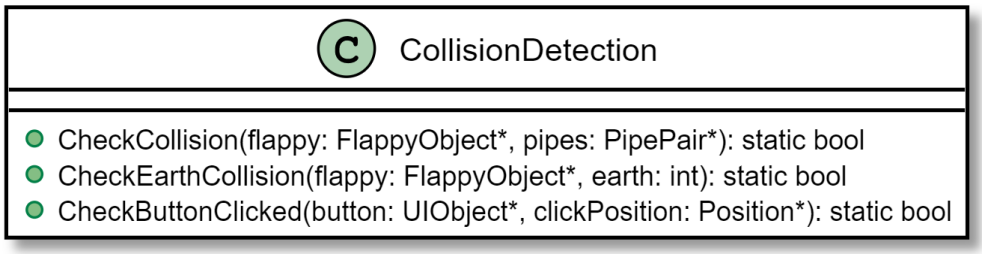
CheckScore klassen udelukkende brugt til at tjekke om flappybird er kommet forbi en pipe, således at scoren kan forøges. Nedenfor på Figur 18, kan et klassediagram af CheckScore ses.



Figur 18: klassediagram for CheckScore, hvor 1 statisk metode kan ses.

CollisionDetection

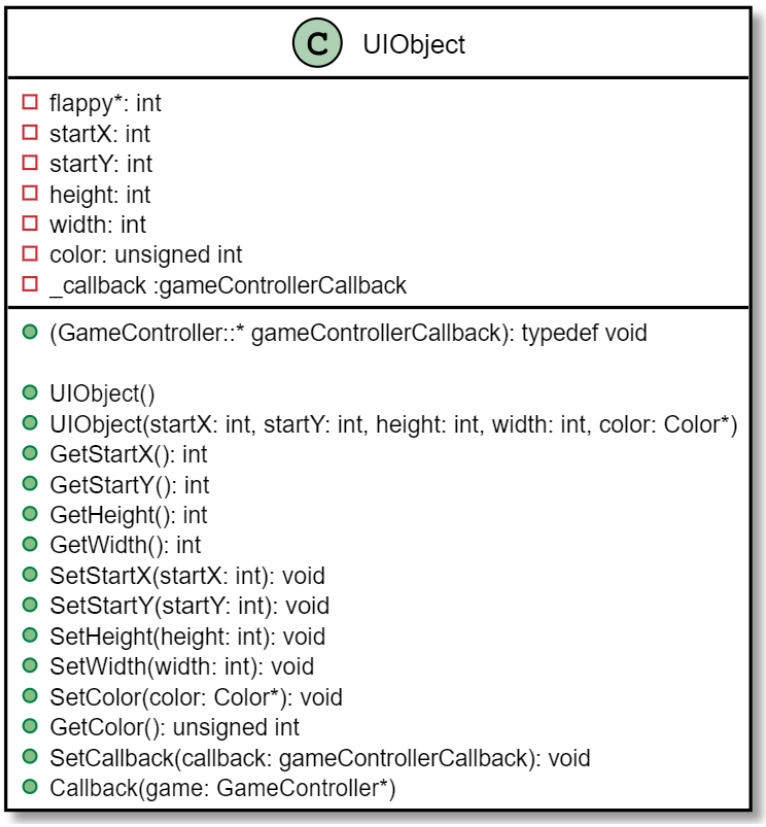
CollisionDetection bruges til at at tjekke om flappybird er kollideret med enten jorden eller med en pipe. Derudover benyttes den også til at tjekke om et Positions objekt fra TouchDriveren er kommet i kontakt med et UIObject objekt. Dette er hvis en person trykker på en knap i menuen. Nedenfor på Figur 19, kan et klassediagram af CollisionDetection ses.



Figur 19: Klassediagram for CollisionDetection

UIObjects

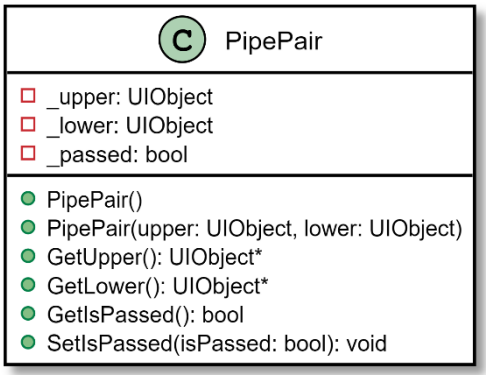
UIObjects fungere som en modelklasse, som kan indeholde data. Ud over data, indeholder et UIObject i nogle tilfælde en pointer til en member metode i GameControlleren. Der kan således kaldes en metode i GameController når der trykkes på knappen. Klassediagrammet for UIObject klassen kan ses nedenfor på Figur 20.



Figur 20: Klassediagram for UIObjects

Pipepair

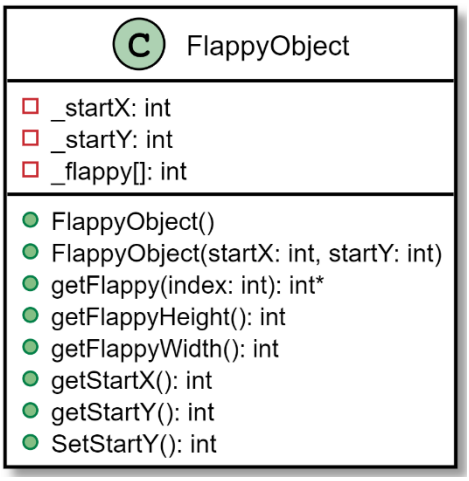
Pipepair klassen benyttes til at holde på 2 UIObject objekter, en upperPipe og en lowerPipe. Ud over de 2 UIObject objekter, så indeholder den også en bool kaldet _passed. Denne bool holdes false indtil flappybird har passeret lowerPipe og upperPipe objekternes x-koordinater. Implementeringen af denne klasse kan ses på Figur 21 nedenfor.



Figur 21: Klassediagram for Pipepair

FlappyObject

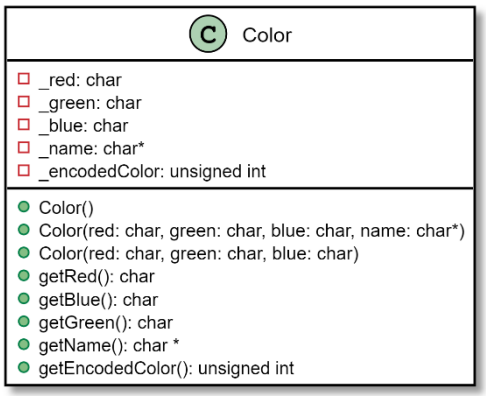
FlappyObject klassen skal indeholde flappybird i et objekt, så den er nemmere at håndtere i TFT-driveren samt GameControlleren. FlappyObject klassen er at samme type som UIObject, dog ikke af samme type, for at nemmere at kunne implementere en mere rigtig flappybird i fremtiden. Klassediagrammet for denne kan ses nedenfor på Figur 22. Klassen indeholder ikke en Setter til StartX, da den ikke skal kunne bevæge sig på sin X-akse. Se også Figur 1.



Figur 22: Klassediagram for FlappyObject

Color

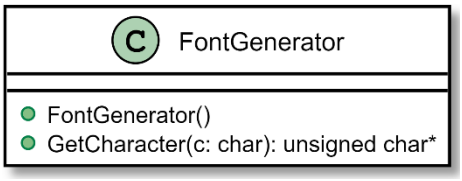
Color klassen benyttes til at håndtere rgb til 16-bit konvertering, som så bliver opbevaret i et objekt at color klassen. TFT-driveren benytter sig nemlig af 16 bit farvekoder, og vha. denne klasse kan mange lave formatet rgb om til 16 farvekode. Konverteringen foregår i constructoren, hvorved farvekoden gemmes i en privat attribut. Nedenfor på Figur 23, kan et klassediagram for Color ses.



Figur 23: Klassediagram for Color klassen.

FontGenerator

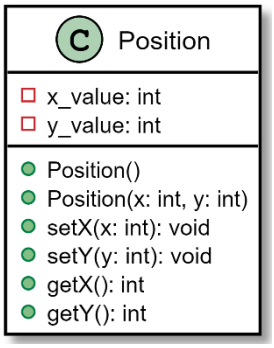
Nedenfor på Figur 24 kan klassediagrammet ses for FontGenerator. De private attributer code0-9 er hardcoded unsigned char arrays indeholdende bytes, som indeholder information om hvordan karakteren skrives på displayet.



Figur 24: Klassediagram for Fontgenerator

Position

Position klassens objekter skal indeholde koordinaterne for, hvor på skærmen en spiller har trykket. Nedenfor på Figur 25, kan implementeringen af Position ses.



Figur 25: Klassediagram for Position.

Debugging

Det er ingen hemmelighed at det kan være udfordrende at fejlfinde på embeddede systemer, hvilket der er blevet brugt meget tid på i løbet af projektet. For at fejlfinde er der blevet taget flere forskellige metoder i brug. I hvert underafsnit beskrives én eller to debugging metoder som er blevet anvendt til det givne problem.

Debugging af TouchDriver signal

Indledningsvist var der for eksempel problemer med at få touch driveren til at fungere ordentligt. Til dette loddede vi nogle pins på ITDB02 Arduino Shieldet, som vi derefter forbandt et Analog Discovery 2 oscilloskop til.[22] Vi kunne da se spændingen på portene, og kunne derefter bruge dette som hjælp til at finde ud af hvad der var galt i koden.

UART

For at finde fejl i koden er der oftest blevet brugt UART driveren[8] som blev udviklet på første semester i MSYS. Driveren blev mere specifikt brugt til at sende resultater over UART til en terminal[24] på en PC. Dette kan bruges til fx at finde ud af om programmet følger det rigtige flow i en given situation. Denne metode er tilstrækkelig i de fleste tilfælde, men nogle gange kan det være svært at finde fejlen, specielt hvis man ikke har en idé om hvor den stammer fra.

Simulering og JTAG

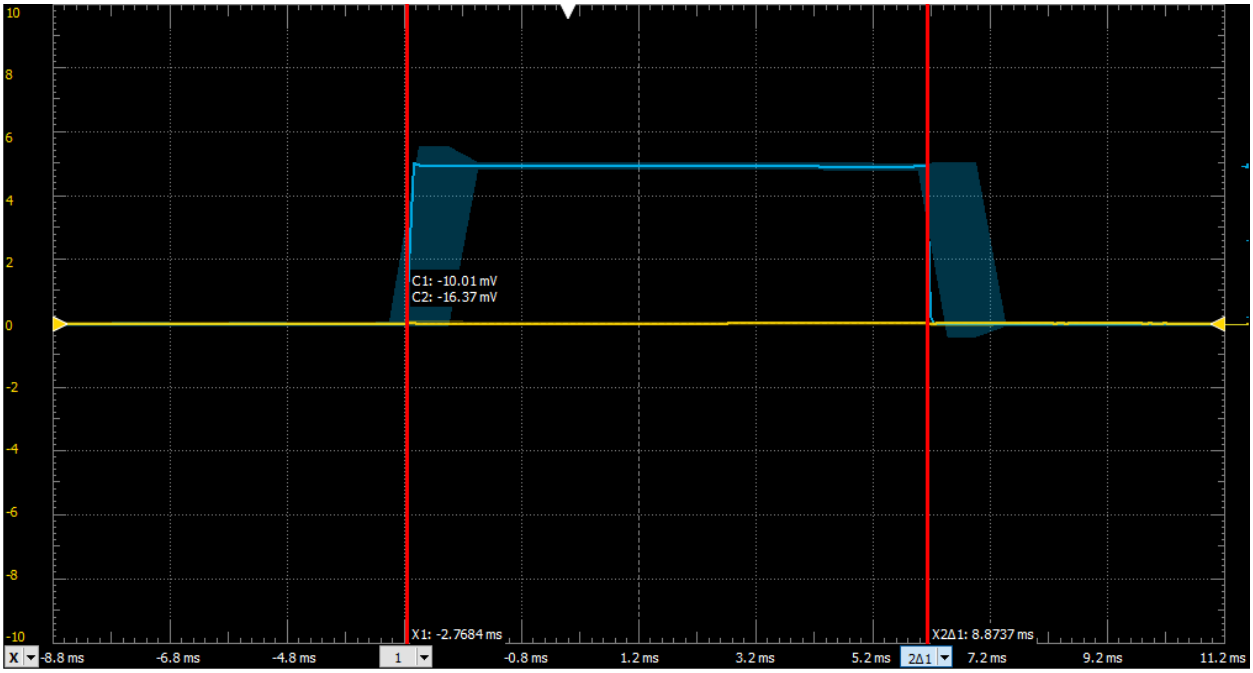
Når problemet ikke var til at debugge ved brug af UART, blev der i stedet anvendt simuleringsværktøjet i Atmel Studio, eller en Atmel ICE JTAG-debugger. Simuleringen var tilstrækkelig i de fleste tilfælde men nogle gange var kommunikationen med touchdriveren og TFT-displayet vigtig at have med. Der blev anvendt en JTAG-debugger til at debugge programmet i realtid. Denne metode er klart den bedste og mest effektive, men kræver dog en del opsætning. Denne metode blev blandt andet brugt at finde fejl ved allokering af objekter. Til opsætning af JTAG debugger anvendes guiden fra lektionen om JTAG.[18]

Resultater og diskussion

Testresultater af krav

Test af FPS

Test af FPS kan man ikke blot gøre ved at ”se om det virker” da det kan være svært at vurdere hvor lang tid den egentligt er om at opdatere skærmen. Derfor anvender vi i stedet en port som vi sætter høj før vi tegner spillet i en ny frame, hvorefter den sættes lav, når alle elementerne på skærmen er opdateret. På denne måde kan vi måle hvor mange gange vi kan opdatere spillet per sekund. Vi anvender Analog Discovery 2[22] som oscilloskop og forbinder den positive indgang til porten, og referencen til ground på Mega2560. Data kan nu opsamles ved brug af Digilent WaveForms.[23] Resultatet kan ses på Figur 26.



Figur 26 Oscilloskop billede af måling af FPS. Den blå kanal er tilsluttet PORTF ben 7 på Mega2560.

Ud fra ovenstående figur kan vi altså se at det tager 8,8737 ms at opdatere skærmen. Denne tid er forsøgt målt mens der var flest mulige elementer på skærmen som skulle opdateres. Dette giver os en teoretisk maksimal opdaterings hastighed på omkring $\frac{1}{0{,}008737} \sim 114\ Hz$. Den maksimale reelle opdaterings hastighed er nok en smule mindre, men stadig væsentligt over de 10 Hz som var vores minimale krav. Rent praktisk anvender vi blot en timer der kører med en fast frekvens på ca. 30Hz, som sætter et flag om, at elementerne skal opdateres. Opdateringen af objekter kan altså godt følge med timeren, og vi imødekommer altså vores krav på minimum 10 Hz.

Test af resterende krav

De resterende tests af krav fra MoSCoW, testes blot visuelt for at se om de virker. En tabel med en oversigt over de forskellige tests kan ses herunder i Tabel 5. Udførelsen af alle tests, på nær måling af FPS, kan ses i den vedlagte video-præsentation. De bliver derfor ikke nærmere beskrevet her.

Krav	Testet	Resultat
Must have		
Skærmen skal opdateres minimum 10 gange i sekundet.	x	Godkendt
Spilleren skal kunne interagere med spillet ved at trykke på skærmen.	x	Godkendt
Spillet skal køre på en microcontroller med kompatibelt grafisk TFT-display.	x	Godkendt
En Touch Driver, som kan læse hvor på skærmen spilleren trykker.	x	Godkendt

En TFT driver, som kan vise grafiske elementer på TFT-display	x	Godkendt
Should have		
Bør kunne vise spillerens score mens spillet er i gang, samt opdatere scoren løbende.	x	Godkendt
Bør have en menu med tilhørende knapper.	x	Godkendt
Bør have en Game-Over skærm med tilhørende knapper.	x	Godkendt
Could have		
Gemme highscore i statisk hukommelse.		Ikke implementeret
Sværhedsgrad forøges baseret på spillerens nuværende score.	x	Godkendt
Wont have this time		
Detaljeret grafik.		Ikke implementeret

Tabel 5: Testresultater af MoSCoW krav

Konklusion

Ud fra projekt arbejdet, er det lykkedes at udarbejde en efterligning af spillet FlappyBird i c++ til en Mega2560 microcontroller, med et TFT LCD-display styret af en ILI9341 display controller, samt XPT2046 Touch controller. Derudover er der blevet implementeret en Touchdriver, som bruges til at kommunikere med touch controlleren. Ud over en Touchdriver, så er det blevet arbejdet videre på TFTdriveren fra undervisningsopgaven fra lektion 3.[(5)] Hertil blev der tilføjet en fontgenerator til tal, samt diverse nødvendige metoder for at få spillet til at blive vist på skærmen. Det er også lykkedes at kunne opfylde alle must haves, alle should haves, samt én could have fra MoSCoW analysen. Dette kan ses i Tabel 5, hvor MoSCoW testes. De resterende could-haves og wont-have (this time), kan passende implementeres i fremtidigt arbejde.

Fremtidigt arbejde og overvejelser

GUI og grafik

GUI og grafikken generelt er ikke specielt pæn, og er relativt begrænset hvad angår funktionalitet. Én af tingene som kunne være oplagt at arbejde lidt videre med, er at få lavet et dataformat som gør at vi kan tegne Flappy i en bedre opløsning. Da Flappy udelukkende består af 3 farver, kan man i princippet lave et 3 bit format, hvor den første bit fortæller om der skal tegnes en farve eller baggrund, og de næste to bits bestemmer farven. Desuden kunne det være sjovt at lave en form for animation når spilleren trykker på en knap, som langt de fleste moderne GUI'er har.

Touch og PhysicsEngine

Touch input fungerer men det er stadig muligt at holde fingeren på skærmen hvorefter Flappy accelererer. Dette er ikke meningen og kræver mere debugging af kode og signaler for at finde ud af hvordan dette

forhindres. PhysicsEngine klassen kunne desuden godt trænge til en lille tweak af parametre for at få den til at virke mere som det oprindelige spil, da det nogle gange kan være lidt svært at kontrollere positionen på Flappy.

Variabel opdaterings frekvens og synkronisering

Dette blev nævnt som en mulighed tidligere i rapporten, men gruppen kom aldrig i mål med at implementere dette, hvilket kunne være godt at få implementeret, så man får en bedre spiloplevelse. Det kunne også være interessant at få den til at synkronisere bedre med den nuværende opdaterings hastighed, så vi undgår flimmer, når vi opdaterer især Flappy.

Gem indstillinger og highscore i statisk hukommelse

Dette blev nævnt som en wont-have i MoSCoW analysen og var derfor prioriteret lavest. ITDB02 shieldet med TFT display og touch controller, indeholder dog også et SD kort som kunne være ideélt til formålet. Alternativt har Mega2560 også noget statisk hukommelse som kan anvendes.

Referencer

Datablade

1. Xpt2046-datasheet, *pdf*
2. ILI9341_v1.11, *pdf*
3. DS_IM120419005_ITDB02_3.2S, *pdf*
4. DS_IM120417024_ITDB02ArduinoMEGAShield, *pdf*

Filer

5. AMS_Lab3a_Graphic, *pdf*, *undervisningsøvelse*
6. AMS_Lab10_Touch_Driver, *pdf*, *undervisningsøvelse*
7. FreeRTOS_manual, *pdf, FreeRTOS manual*
8. MSYS Lektion vedr. UART, *sourcekode benyttet i projektet.*

Websider

9. http://www.rinkydinkelectronics.com/library.php?id=92, *Rinky-Dink Electronics, Henning Karlsen.*
10. https://en.wikipedia.org/wiki/MoSCoW_method, *MoSCoW analyse værktøj til kravspecifikationer*
11. https://raspberrypi.dk/produkt/officielt-raspberry-pi-7-touchscreen-display/?gclid=Cj0KCQjwn7j2BRDrARIsAHJkxmyb8pAruQX-HqTdkYQzl8TAdoGm_Hw2mc-2170E-1gQn8QrBbPq2P8aAiEUEALw_wcB, *raspberry pi med touchskærm*
12. https://www.qt.io/download, *qtCreator*
13. https://en.wikipedia.org/wiki/Single-responsibility_principle, *single responsibility principle fra SOLID principperne*
14. https://www.geeetech.com/wiki/index.php/Arduino_Mega_2560, *Mega2560 board*
15. https://flappybird.io/, *Flappy bird spillet I webversion.*
16. https://www.kursuskatalog.au.dk/da/course/89897/E1MSYS-01-Microcontroller-systemer, *MSYS kursus beskrivelse.*
17. https://www.microchip.com/mplab/avr-support/atmel-studio-7, *Atmel Studio*
18. https://www.microchip.com/DevelopmentTools/ProductDetails/ATATMEL-ICE, *Atmel-Ice debugger*
19. https://en.wikipedia.org/wiki/Thin-film-transistor_liquid-crystal_display, *TFT forklaring*

Programmer

20. http://en.radzio.dxp.pl/bitmap_converter/ (LCD Assistant)
21. https://www.buydisplay.com/lcd-font-maker (LCD font maker)

22. https://store.digilentinc.com/analog-discovery-2-100msps-usb-oscilloscope-logic-analyzer-and-variable-power-supply/ Analog discovery oscilloskop for debugging
23. https://store.digilentinc.com/waveforms-download-only/, *Waveforms – software for analog*
24. https://ttssh2.osdn.jp/index.html.en Tera Term til UART-kommunikation.

Figur og tabel liste

Figurer

Tabeller