WEM1 Mini projekt:

Wii NunChuck / MicroSpace

Af:

_____

Martin F. Fisker 06839

_____

Jonas Lange 06715

_____

Torben Porsgaard 05241

Indhold

Figur og tabel-fortegnelse

Introduktion

Dette projekts mål er defineret ved et hovedmål og 2 delmål.

Projektets hovedmål var at interface Tahoe-II boardet med en Wii NunChuck via I2C bussen, og lave en driver der kan aflæse dens forskellige inputs:

- En analog stick (X/Y Position)
- En 3 axis accelerometer (X/Y/Z axis)
- 2 digitale tryk knapper

Projektets 1. delmål var at udvide device driveren til et egentligt API, der egner sig til anvendelse i spil og applikationer. Desuden ønskede vi at lave en applikation der anvendte dette API til at styre et objekt på skærmen.

Projektets 2. delmål er at udvide førnævnte applikation til et egentligt spil, med et rumskib der skyder forfølgende asteroider ned.

Arkitektur

Løsningen består af et Tahoe-II kort, der er udstyret med en meridian CPU, touch screen, ethernet, accelerometer , serial port, trykknapper og expansion headers der muliggør forbindelse til bl.a meridian cpu'ens I2C og SPI bus, UART enheder og GPIO pins. Vi forbinder I2C bussen på kortet til NunChuck kontrolleren, da NunChucken indeholder en I2C slave enhed til at kommunikere sine sensor data. Vores program sampler NunChuckens tilstand, dette bruges så til at styre rumskibet i programmet. Opstillingen kan ses i Figur 1.

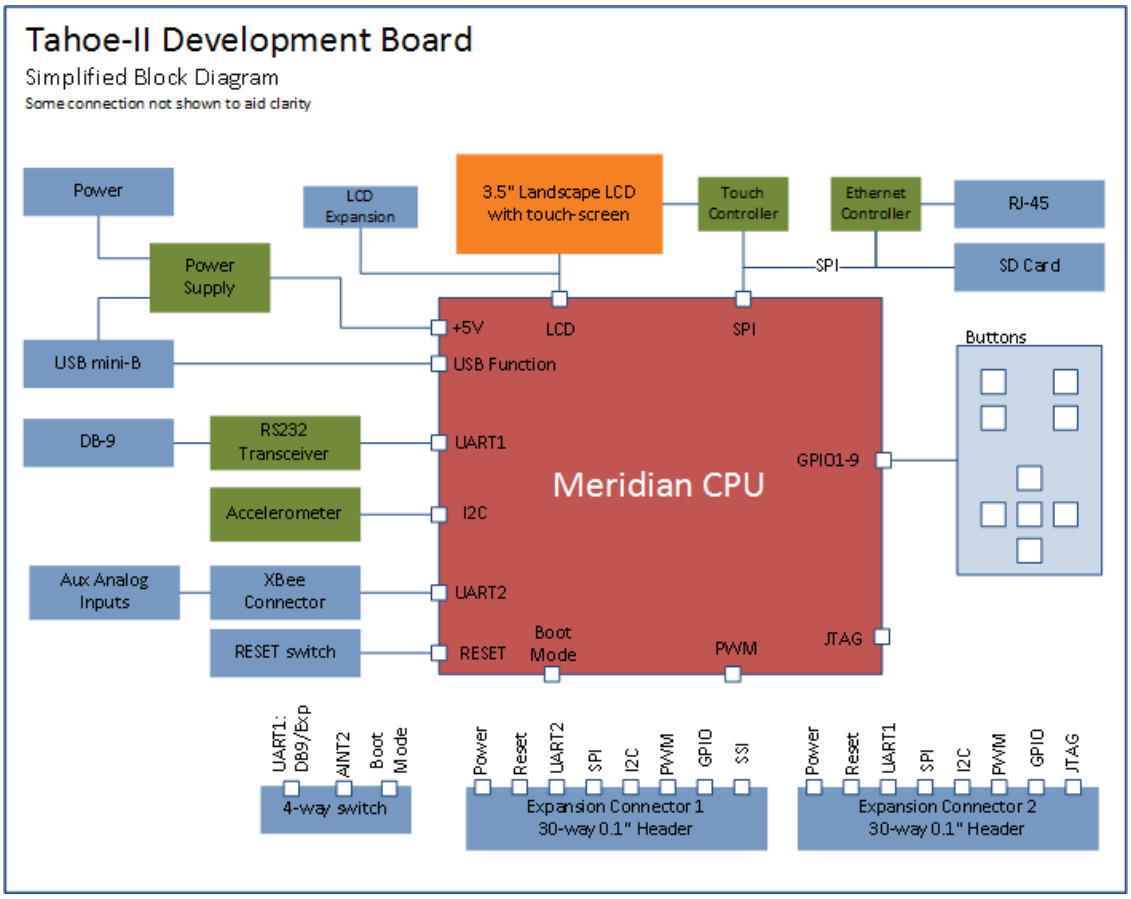
Figur 1 Tahoe-II kortet med en Wii NunChuck tilsluttet

Hardware design

Som nævnt i forrige afsnit består vores løsning af 2 dele. Nemlig et Tahoe-II kort og en Wii NunChuck.

Tahoe-II kortet indeholder en 3.5" LCD touch-screen. Der er mulighed for trådet netværksforbindelse med den integrede ethernet controller og mulighed for trådløs netværk med en optionel XBee enhed. Boardet indeholder også et accelerometer samt mulighed for interfacing til externe enheder ved brug af serielle porte, I2C, SPI og GPIO pins som stilles til rådighed med kortets expansion headers. Til at forbinde kortet med en PC bruges den integrerede USB controller. Dette bruges når man vil deploy et program på Tahoe-II kortet.

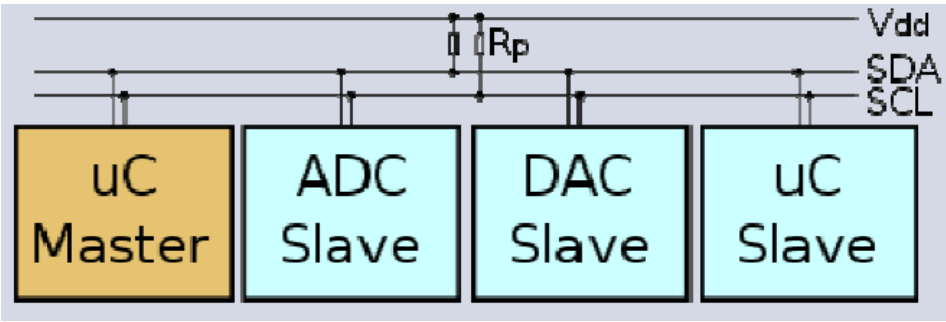
Tahoe-II kortet er bygget op omkring Meridian CPU'en der består af en Freescale i.MXS ARM9 processor, 4Mbytes Flash og 8Mbytes RAM. Kortet kommer fra producenten præpareret med en version af .NET micro der indeholder BSP'en der gør at .NET micro framework'et kan afvikles på dette kort. Nedenfor ses et blok diagram over Tahoe-II kortet.



Figur 2 Block diagram for Tahoo-II Development Boardet

Vi har som sagt forbundet Tahoe-II kortet og NunChucken med en I2C forbindelse.

I2C er en synkron multi-master seriel bus. I2C bruger to tovejs linjer, nemlig Serial Data (SDA) og Serial Clock (SCL), begge trukket op med pull-up modstande. Typisk anvendes spændingen 5 V eller 3,3 V, i vores tilfælde bruger vi 3,3 V. Et I2C bus eksempel er afbildet I figuren nedenfor.



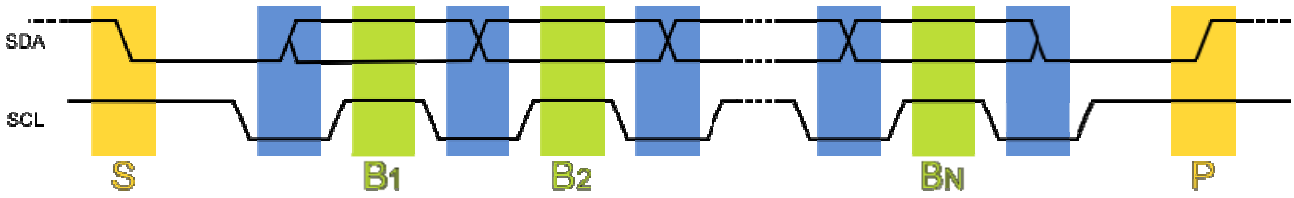
Figur 3 I2C bussens opbygning

I2C reference designet har et 7-bit adresse rum med 16 reserverede adresser. Det betyder at højst 112 noder kan kommunikere på samme bus. De mest almindelige modes er standard mode, der giver en hastighed på 100kbit/s og low speed mode der giver en hastighed på 10kbit/s. I de seneste revisioner af I2C

er der kommet mulig for 10-bit adressering og højere klok hastigheder i form af Fast mode(400 kbit/s), Fast mode plus(1 Mbit/s) og High Speed mode(3.4Mbit/s).

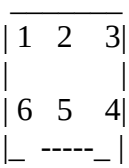
Kommunikation over I2C bussen foregår ved at master sender et start bit efterfulgt af adressen på den slave masteren ønsker at kommunikere med. Dette efterfølges af et enkelt bit som repræsenterer, om masteren ønsker at skrive(angivet med 0) til eller læse(angivet med 1) fra slaven.
Hvis slaven findes på bussen, vil den reagere med et ACK bit (angivet med 0). Master fortsætter derefter med enten at sende eller modtage og slaven fortsætter med henholdsvis at modtage eller sende. Modtageren af data sender efter byte modtaget et ACK.

Masteren slutter transmissionen med et stop bit. Masteren kan også afslutte med et start bit hvis denne ønsker at beholde kontrollen over bussen.
Data sendes med mest signifikante bit først. Start bit angives med en høj til lav overgang for SDA mens SCL er høj. Stop bit er angivet med en lav til høj overgang på SDA mens holdes SCL høj kontinuerlig. Illustration på dette er vist nedenfor.



Figur 4 Timing diagram for I2C bussen

NunChucken bruger et proprietært stik til I2C forbindelsen. Stikket har 6 ben men kun 4 er i brug. Følgende er en illustration og forklaring af stikkets layout.



1. Grøn, data
2. Ikke i brug
3. Rød, 3.3+v
4. Gul , bus clock
5. Ikke i brug
6. Hvid, stel

Data kan læses fra NunChucken i I2C fast mode når der bruges 5v og i standard mode når der bruges 3.3v. NunChuckens hardware adresse er "0x52". Først udføres en handshake transaktion hvor 2 bytes "0x40,0x00" sendes til NunChucken. Herefter er NunChucken initialiseret og klar til at modtage forespørgelser på status. En sådan forespørgsel forestages ved at sende byten “0x00” til NunChucken. NunChucken svarer så ved at sende 6 bytes som indeholder NunChuckens status på dens sensorer.

Da Nintendo har valgt at kryptere data er det nødvendigt først at dekryptere de modtagne bytes. Da det er en simpel kryptering kan dette gøres ved at anvende nedenstående formel på hvert modtagne byte.

```
<dekrypteret byte> = (<krypteret byte> XOR 0x17) + 0x17
```

De 6 ukrypterede bytes man nu har, kan så bruges til at aflæse den analoge stick med 2 akser, accelerometeret med 3 akser og de 2 digitale knapper navngivet Z og C. Tabel 1 viser hvorledes data er formateret.

Tabel 1 WII NunChuck data format

	Bit							
Byte	7	6	5	4	3	2	1	0
0	< - - - - - - - - - - - - - -	Analog Stick X akse værdi bit [7:0]				- - - - - - - - - - - - - - - - - - >		
1	< - - - - - - - - - - - - - -	Analog Stick Y akse værdi bit [7:0]				- - - - - - - - - - - - - - - - - - >		
2	< - - - - - - - - - - - - - -	Accelerometer X akse værdi bit [9:2]				- - - - - - - - - - - - - - - - - - >		
3	< - - - - - - - - - - - - - -	Accelerometer Y akse værdi bit [9:2]				- - - - - - - - - - - - - - - - - - >		
4	< - - - - - - - - - - - - - -	Accelerometer Z akse værdi bit [9:2]				- - - - - - - - - - - - - - - - - - >		
5	Accelerometer Z akse værdi bit [1:0]		Accelerometer Y akse værdi bit [1:0]		Accelerometer Y akse værdi bit [1:0]		C knap Status	Z knap Status

Som ovenstående diagram viser angives den analoge sticks tilstand ved 2 bytes, 1 for hver akse. For accelerometerets vedkommende er værdien for hver af akserne angivet med 10 bit, de 8 mest betydende bit for akserne er angivet i bytene 2 til 4. De 2 mindst betydende bit for hver akse er samlet i bit 7 til 2 i byte 5. Bitene 1 og 0 i byte 5 angiver om knapperne C og Z er nede eller oppe.

En kalibrering er nødvendig for de analoge input for at kunne bruge de værdier vi modtager. For den analoge sticks vedkommende består det blot i at notere værdien vi får, når vi for hver akse bringer den i hver af yderpositionerne samt center positionen. Accelerometeret kalibrerer vi ved for hver akse at notere de værdier vi får, når vi holder aksen parallel med tyngdekrafts-vektoren, først i aksens positive retning og derefter i aksens negative retning. Vi har nu ekstrema værdierne for accelerometeret ved +/- 1g påvirkning. Værdierne kan blive optil den dobbelte magnitude da accelerometeret kan måle +/-2g [1].

Software design

NunChuck driveren

Da hovedmålet i dette projekt var at udvikle en driver til NunChuck'en var dette den første ting vi lavede. Som driver til NunChuck'en har vi således udviklet klassen NunChuck. Det var meningen at driveren skulle være velegnet til brug i spil, og vi er derfor gået langt i vores bestræbelser på at abstrahere alle detaljer der vedrører kommunikationen med NunChucken og fortolkning af resultater væk.

NunChuck klassen er lavet som en singleton der er trådsikret og implementerer IDisposable mønstreret. Den implementerer singleton mønstret da Tahoo II boarded kun har én I2C bus og NunChuckens adresse ikke kan ændres. NunChuck driveren er derfor både nem og sikker at tilgå alle steder i koden man måtte ønske det. For at gøre driveren mere velegnet til spil, bliver den analoge sticks rå værdier, der ligger ca. i

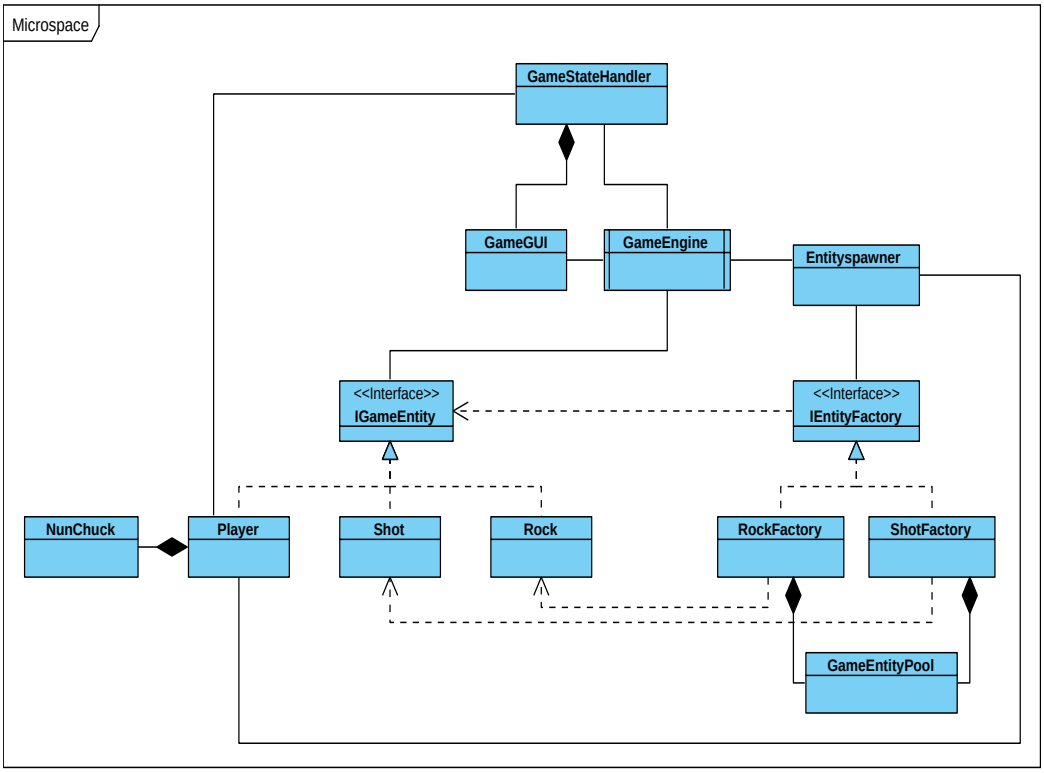
området 30 – 130 for hver akse, omregnet og lavet til et Vector3D objekt, hvor værdierne af de enkelte komponenter i vektoren ligger fra -1.0 til 1.0, da den analoge stick er 2-akset er z komponenten i vektoren altid 0. Det samme går sig gældende når beder NunChuck'en om værdierne for accelerometeret, dog med de forskelle at alle 3 dimensioner i vektoren benyttes da accelerometeret er 3 dimensional, og de rå værdiers "range" er anderledes, og at koordinatrummets akser ligger i værdi området -2.0 til 2.0. Vektorens længde angiver kraftens styrke, ved længden 1 svarer styrken til 1g. Dette vil bl.a. opleves når NunChuck'en holdes stille, vektorens retning vil så angive NunChuck'ens rotation. Da accelerometeret kan måle op til 2g, kan vektoren der returneres have en længde op til 2. Hvis vektorens længde er over 1 er dette udtryk for at brugeren påfører NunChuck'en en acceleration. Desuden er der lavet properties i driver klassen så vektorerne kan fås i en udgave hvor y komponentens fortegn er inverteret. Dette gør det lettere at anvende vektorerne direkte i spil da mange frameworks, inklusiv den i .NET micro, repræsenterer skærmen med et koordinatsystem hvor alle punkter der ligger på skærmen har en y komponent der er lig 0 eller negativ.

Spillet "MicroSpace"

Spillets design bærer i stor udstrækning præg af at være styret af performance hensyn, desuden var spillets begrænsede omfang også medvirkende til at der aldrig blev tale om et decideret lagdelt design.

Skulle man have lagdelt designet ville det være naturligt at se NunChuck driveren som værende i service laget, det nederste. Der hvor en yderligere opdeling ville være passende var at fratage domæne klasser, så som Player, Rock og Shot, opgaver relateret til at sende bitmaps til skærmen og placerer dem i domæne laget sammen med de forskellige hjælpe klasser EntitySpawner , factories o.s.v.. Tegning af bitmaps skulle håndteres af GameEngine ved anvendelse af en EntityDrawing class, der sendte bitmaps til skærmen ud fra domæne viden fra de respektive klasser. Disse skulle så placeres i et præsentationslag. IGameEntity skulle udvides til at kunne levere nødvendig information til tegning af de korrekte bitmaps med den korrekte placering, f.eks. billede type, billedeindex og position.

I Figur 5 ses et forsimplet klassediagram for systemet.



Figur 5 Forsimplet klasse diagram

GameStateHandler og GameGui

GameStateHandleren er den klasse der holder spillets state. Den sætter spillet op, aflæser knap tryk, spawner nye Rocks når disse dør, justerer spillets sværhedsgrad/level og holder styr på spillerens score. Alt information til spilleren, i form af level, score og "Game over" meddelelser, foretages af klassen GameGUI der således varetager visnings delen af state håndteringen.

State justeres ved at lytte på GameEnginens PoststepEvent, PreStepEvent og KilledEntityEvent.

GameEngine

GameEngine klassen er en central del i spillet, i det at den står for at indkapsle og håndtere den del af spillets logik som i stor udstrækning kan siges at være generel for alle spil. Samtidig er den implementeret på sådan en måde at logik der er specifik for vores spil, ikke indgår i vores GameEngine klasse, men håndteres andre steder. Disse forhold skulle gerne betyde at denne klasse i stor strækning vil kunne genbruges til andre spil.

De generelle og centrale opgaver som skal løses i et spil og som er håndteret af GameEngine klassen er sekvensering og timing af events i spillet. GameEngine klassen er aktiv og funktionen run kører i sin egen tråd. Denne tråd kan betragtes som spillogikkens hovedtråd. Overordnet set fungerer GameEngine ved at køre en uendelig løkke(eller til man kalder stop på GameEngine). Hver iteration svarer til en "step" i spillets logik og en frame. GameEngine indeholder en collection med IGameEntity objekter. Enheder der er aktive i spillet tilføjes til denne collection. Da GameEngine arbejder med objekter udelukkende gennem IGameEntity interfacet, er det kun de properties og methods der udstilles i dette interface og som er general for alle objekter der indgår i spillet som benyttes af GamEngine. Dette gør at GameEngine ikke er afhængig af eller skal bekymrer sig om de enkelte GameEntity typers faktiske opbygning og adfærd. For

hver iteration i den nævnte løkke i GameEngine tråden, bliver der på alle GameEntity objekter i den nævnte collection kaldt metoderne NotifyCollision(ArrayList IGameEntity), UpdateWorld(TimeSpan gameTime) og Render(Bitmap screen), i nævnte rækkefølge. Det er så GameEntity objekternes eget ansvar at foretage sig hvad de skal i forbindelse med at nævnte metoder er blevet kaldt på dem.

Når NotifyCollision bliver kaldt på et GameEntity objekt, får den af GameEngine medsendt en ArrayList med alle IGameEntities der er aktive i spillet. Der så det pågældende GameEntity objekts ansvar at checke om den er kollideret med et eller flere andre objekter i spillet. Det er muliggjort ved at IGameEntity interfacet definerer en property der returnerede objektets bounding shape. Det er derfor mulig for pågældende IGameEntity objekt at skaffe bounding shape for alle Game Entities og afgøre om den er involveret i en eller flere kollisioner samt herefter reagerer i overensstemmelse hermed.

Når UpdateWorld(TimeSpan gameTime) bliver kaldt på et GameEntity objekt bliver der medsendt et TimeSpan objekt som fortæller hvor længe det er siden sidst at UpdateWorld blev kaldt på pågældende objekt. Udregningen af denne tid varetages af GameEngine klassen.

Når UpdateWorld bliver kaldt på et GameEntity objekt er det dette objekts ansvar at opdaterer sin interne tilstand, dette kunne f.eks. bestå i at opdatere hvor i spil verdenen at objektet befinder sig. Tiden siden sidste kald bliver medsendt så at GameEntity objektet kan tage disse beslutninger på baggrund af den ”wall time”[1] der er gået. Hvis dette ikke var tilfældet ville hændelser i spillet blive afviklet med en hastighed der var afhængig af hvor mange GameEngine kan nå rundt i dens tråds hovedløkke, som igen ville være afhængig af hardwarens hastighed.

Som den sidste af de nævnte funktioner bliver Render kaldt på hver GameEntity objekt i GameEngine’s collection. Når Render bliver kaldt på et GameEntity objekt er det pågældende objekts ansvar at tegne sig selv på rette sted og på rette måde i det bitmap som den får medsendt af GameEngine. Dette bitmap bruges til repræsentation for skærmen og bliver ved slutningen af hver iteration af forrige nævnte løkke flush’et til skærmen.

EntitySpawner

De af spillets game entities der tilføjes under spillets afvikling, det være sig Rocks eller Shots, tilføjes ved hjælp af en entity factory der implementerer IEntityFactory, som navnet antyder, er disse baseret på Factory mønstret. For hver type instantieres en EntitySpawner der får den korrekte IEntityFactory med ved construction.

Oprindeligt instantierede de konkrete entity factories nye game entities hver gang der var behov for en ny f.eks. hver gang der blev skudt en Rock, blev der lavet en ny og GameEngine kaldte Dispose på den netop nedskudte Rock. Denne opførsel gav markante performance problemer, disse viste sig ved at spillet hakkede en smule umiddelbart efter en Rock har helt væk, ikke altid efter helt det samme tidsrum. Vi identificerede problemet til at være garbage collectoren der arbejdede umiddelbart efter, da vi presser CPU’en så meget vi kan (hvis den har tid bliver den brugt til flere fps) og da størrelsen af en Rock ikke er ubetydeligt var dette stykke arbejde meget tydeligt for performance. For at løse problemet introducerede vi en GameEntityPool der kan holde alle oprettede game entities og på opfordring kan levere game entities,

[1] Wall time, også kaldet real-world time, refererer til den forløbne tid målt af et ur. Altså ikke et antal ticks eller lign.

skulle der være nogle der er blevet fjernet fra spillet i mellemtiden. Dette løste problemet fuldstændigt – man kunne fristes til at sige ”hvis man ikke smider noget væk bliver garbage collection ikke noget problem”.

Utility klasser

Da .NET micro framework’et mangler, hvad der må anses for, basal matematik og geometri funktionalitet har det været nødvendig, til brug for vores spil, at dels udvikle og dels at skaffe følgende klasser fra 3. part.

MathHelper[2], der indeholder en række funktioner som mangler i .NET micro, bl.a. en kvadrat rods funktion. Derudover indeholder klassen en implementation af cosinus og sinus til erstatning for de eksisterende funktioner i .NET Micro framework’et. Dette er brugt til dels på grund af at de eksisterende funktioner kun kan arbejde med hele grader. Dette er nærmere omtalt i næste afsnit. Den vigtigste årsag til at bruge en anden implementation er dog, til vores store overraskelse, at de i forvejen eksisterende cosinus og sinus funktioner i framework’et regner forkert med en ikke ubetydelig margen.

Til dækning for vores behov for basal geometri har vi udviklet klasserne Circle, Rectangle, Point og Vector3D der indeholder en implementation af de geometriske konstruktioner som navnene antyder.

Vi har også udviklet klassen BitmapHelper der indeholder funktionen CounterClockwiseRotate, der kan dreje et bitmap om sit eget centrum med en vinkel der medgives som parameter til funktionen. Det var i starten meningen at denne skulle bruges i forbindelse med animeringen af objekterne i spillet. Det viste sig dog hurtig at denne var alt for langsom til brug i vores spil.

.Net Micro udfordringer

.Net Micro frameworkets begrænsninger i forhold til det almindelige .Net framework, oplevede vi i løbet af projektet – og nok i højere grad end forventet.

Et af de første problemer vi stødte på, var da vi havde brug for matematiske funktioner til udregning af vinkler og afstande i forbindelse med vores applikation. Det viste sig de indbyggede matematik-biblioteker (Microsoft.Spot.Math og System.Math) kun indeholdt ganske få funktioner, hvoraf sinus og cosinus var de eneste vi potentielt kunne bruge. Forsøg med brug af disse gav dog nogle mærkelige resultater (decideret forkerte resultater), og så kunne de kun operere på grader i heltal, dvs. sinus(13.4°) ikke understøttedes.

Vi valgte derfor at gøre brug af et frit matematik-bibliotek skrevet til .Net Micro, der indeholdt de funktioner vi havde brug for, såsom sinus, cosinus, arc-tangens og kvadratrod.

Kvadratrodsudregningen, som ellers virkede upåklageligt, viste sig desværre at være for tung, da den blev kaldt mange gange i vores applikation, og det resulterede i en meget lidt responsiv applikation. Vi undersøgte derfor mulighederne for optimering af denne funktion, og endte med at bruge en metode der approksimerer afstanden mellem to punkter, bl.a. ved brug af forskellige konstanter, der derfor er hurtigere men også en smule upræcis (kilden[3] angav max 8 % unøjagtighed). På grund af dette kan der i sjældne tilfælde opleves situationer hvor kollisioner ikke håndteres fuldstændig korrekt i applikationen.

[2] http://www.microframework.nl/wp-content/uploads/file/exMath.cs
[3] http://www.azillionmonkeys.com/qed/sqroot.html

Ifølge .Net Micro specifikationen findes der en metode til at rotere bitmaps, men vi kunne underligt nok ikke finde den. Et kig på relaterede fora på nettet afslørede at denne funktionalitet rent faktisk var fjernet fra frameworket, men at dokumentationen rigtigt nok ikke afspejlede det.

Vi implementerede selv en funktion til rotering af bitmaps, men den viste sig at være for langsom til brug i vores applikation. Vi har derfor valgt at lagre flere bitmaps for hvert objekt, og så blot vælge bitmap ud fra objektets tilstand, da vi har nok plads til billederne på boardet.

Under udvikling af GameEntityPoolen fandt vi at Generics ikke er en del af .NET Micro Frameworket. I dette konkrete tilfælde løste vi det med at klassen udelukkende holder allerede oprettede entities og dermed kun arbejder med interfacet IGameEntity. Manglen på generics bunder vel dels i hvad det ville gøre for kernens footprint og del i at .NET Micro Frameworket, skulle det have haft generics med, ville være nød til at håndtere generics runtime eller compiletime frem for JIT time, som .NET Frameworket gør det.

Resultater

Projektet har frembragt, primært et fuldt udbygget API til aflæsning af en Wii NunChuck, sekundært et spil der demonstrerer NunChuckens muligheder, udviklet i .NET Micro Frameworket på en Tahoo-II udviklings kort.

Udover disse mere håndgribelige produkter har vi erhvervet os en forståelse og overblik over de muligheder der ligger i .NET Micro Frameworket og i særdeleshed de begrænsninger frameworket har. Ved begrænsninger tænker vi dels på de sprog specifikke begrænsninger og mangler der ligger i C# fortolkeren og del de begrænsninger som target, i dette tilfælde et Tahoo-II kort, giver i form af begrænset processor og memory kapacitet.

Konklusion

Vi havde i dette projekt primært sat os for at lave en driver til en Wii NunChuck, tilsluttet en I2C-Bus på et device der kører. Net Micro framework. Sekundært ønskede vi at udvikle et generelt anvendeligt API til denne driver, samt en applikation (et spil) der demonstrerer anvendelsen af dette API.

Disse mål blev alle nået ved projektets afslutning, og vi er generelt godt tilfredse med vores løsning.

Driveren til NunChucken blev lavet på baggrund af værdifulde informationer[4] fra personer der tidligere har forsøgt at afkode signalerne der sendes. Uden disse havde det formegentlig ikke været muligt at komme særligt langt med udviklingen af driveren, da der foretages en del opsætning og de-kryptering.

Det er muligt at aflæse værdier for den analoge stick (x- og y-retninger), værdier for accelerometeret (x-, y- og z-retninger) samt værdier for de to knapper på NunChucken. Alle stick- og accelerometer-værdier repræsenteres som vektorer der er lette at benytte i applikationer og i særdeleshed spil, og knapperne repræsenteres som booleans.

[4] http://wiibrew.org/wiki/Wiimote/Extension_Controllers

MicroSpace er i høj grad præget af at det skal afvikles på .Net Micro platformen, primært på grund af de begrænsede CPU resurser[5]. Det har været nødvendigt at forbruge mere memory for at spare beregninger, ligesom vi har måttet gå på kompromis med nøjagtigheden af de beregninger vi foretager for at opnå højere hastighed.

Ved projektets start havde vi nok ikke forestillet os at så centrale dele som f.eks. cosinus, sinus, collections og generics ikke er at finde i .NET Micro Framework. Særligt manglen af et lidt mere udbygget Math bibliotek er noget forbavsende, det ville ikke have haft den store betydning for størrelsen af core.

[5] Tahoo-II kortets Meridian processor er baseret på: ARM920T, 100Mzh, 4Mb. Flash, 8Mb. SDRAM.