
30010 Programmeringsprojekt: Reflex Ball



Mads Friis Bornebusch (s123627)



Tobias Tuxen (s120213)



Kristian Sloth Lauszus (s123808)

Reflex Ball
Group Number: 15
DTU Space
June 26, 2013

Resumé

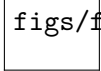
I dette programmeringsprojekt har vi skrevet og designet al koden til spillet Reflex Ball. Koden er skrevet i C i compileren Z8Encore! og implementeret på en Zilog 6403 microcontroller.

Vi har konkluderet at den skrevne kode implementerer det ønskede kredsløb på Microcontrolleren, da alle spillets facetter er blevet gennemtestet og giver det ønskede output. Som slutresultat har vi et fuldt funktionelt ReflexBall-spil med mange udvidelser der bl.a. inkluderer brikker (Arkanoid-stil), power-ups, forskellige levels og styring med ret. Der er indtil videre ikke fundet nogle bugs i slutversionen af spillet.

Abstract

In this programming project we have written and designed all the code to the game Reflex Ball. The code has been written in C in the compiler Z8Encore! and has been implemented on a Zilog 6403 Microcontroller.

We have concluded that the written code implements the desired circuit on the Microcontroller as as all the different facilities of the game has been thoroughly tested and is in compliance with the expected result. In the end we have a fully functional Reflex Ball Game with many expansions, which amongst others, include bricks (Arkanoid style), power-ups, different levels and steering-wheel game controller. So far no bugs has been found in the final version of the game.



figs/forside.png

Figure 1: Screenshot from our implementation of ReflexBall

Fordord

Denne rapport er skrevet som en del af eksaminationen i DTU kursus 30010 Programmeringsprojekt. Alle tre, på forsiden nævnte, gruppemedlemmer har bidraget til rapporten på lige vis. Vi har lavet alle forberedelsesøvelser, skrevet koden og afsnittene i rapporten i fællesskab. Denne rapport beskriver vores arbejde og resultater.

Contents

Resumé	ii
Abstract	ii
Fordord	iii
1 Introduktion	1
2 Struktur	1
2.1 Basics	1
2.2 Advanced	2
2.3 Brikker	4
2.4 Display	5
2.5 Styring med rat	5
2.6 Rettelser og fintuning	5
2.7 Brikker	5
3 Konklusion	7

1 Introduktion

Denne rapport dokumenterer de overvejelser vi har gjort os i forhold til design og struktur af Reflex Ball spillet.

Programmeringsprojektet har overordnet set været inddelt i to faser.

Første del, der inkluderede de fire første arbejdsdage, blev brugt på at lave håndregningsøvelser omhandlende manipulation af hexadecimal- og bitrepræsenterede tal og flere programmeringsøvelser i C, også omhandlende hexadecimal- og bitmanipulation samt ansi-koder, driverhåndtering og vektorregning og styring af LED-display. Formålet med disse øvelser var samlet set at blive bedre til at håndtere hexadecimal- og bitrepræsenterede tal, og desuden at lære om/genopfriske pointers, headers og opsætning af hardware-drivere til microcontrolleren.

Anden del af projektet, der inkluderede de sidste ni arbejdsdage, har vi brugt på først at designe Reflex Ball-spillet ud fra de i opgaven specificerede krav og derefter implementere udvidelser både på hardware- og softwareniveau, sideløbende med vi hele tiden har videreudviklet spillets 'engine', så spillet bruger mindre RAM og kører så fejlfrit som muligt, samtidig med at koden gradvis er blevet gjort mere fleksibel og overskuelig.

Til sidst i anden del af projektet, er denne rapport blevet skrevet.

2 Struktur

2.1 Basics

Da vi skulle planlægge projektet valgte vi efter en brainstorm at dele projektet op i en mængde delmål vi ville have implementeret. Delmålene i kategorien basics var dem vi skulle implementere for at have et fungerende spil. Vi fandt frem til at det ville være følgende funktioner:

- **Keyboard input**
- **Banen.** Banen har vi implementeret i et koordinatsystem på den måde at vi har x1- og x2-koordinater, svarende til banens bredde (x1 = banens venstre væg, x2 = banens højre væg) og y1- og y2-koordinater, svarende til banens højde (y1 = loft, y2 = 'gulv'). Vi arbejder i vores implementation med et venstredrejet koordinatsystem, hvor y-aksen er inverteret i forhold til et almindeligt koordinatsystem, der
- **Bevægelig bold.** Vi har indstillet timeren til hver gang der er et tick bevæger bolden sig. Her i basics implementationen harcodede vi dette til at boldens x-værdi ændrede sig med plus 1 for hvert tick og boldens y-værdi ændrede sig med minus 1 for hvert tick, når bolden blev skudt afsted for strikeren. Dette skyldes at vi jo arbejder med en inverteret y-akse i forhold til et almindeligt koordinatsystem.
- **Man dør når bolden kommer under strikeren.** Strikeren
- **Bevægelig striker.** Til Strikeren har vi lavet et `struct`, sådan så den kender sin egen x-værdi (yderste venstre karakter af strikeren), samt sin bredde. Strikeren er så blevet

implementeret på den måde at når den rykkes til venstre, tjekkes der om strikerens x-værdi -1 er mindre end eller lig med banens venstre side (dvs. banens x1-værdi). Hvis strikeren er ude i sin yderste venstre position er strikerens x-værdi nemlig 1 større end banens x1-værdi, derfor fratrækkes 1 fra strikerens x-værdi inden der tjekkes. Hvis strikerens x-værdi minus 1 er mindre end eller lig med banens x1 når strikeren forsøges rykket, sker der ingenting og strikeren bliver stående. Hvis dette ikke er tilfældet, slettes strikerens yderste højre-karakter det vil sige at feltet `striker.x + striker.width -1` bliver overskrevet med et mellemrum og strikerens x-værdi formindskes så med 1 og strikeren gentegnes.

Når strikeren bevæger sig til højre fungerer samme princip, bare hvor der tjekkes om `striker.x + striker.width` er større end eller lig med banens x2-værdi. Grunden til der ikke minuses med 1 her er at `striker.x + striker.width` svarer til feltet lige til højre for strikerens yderste højre felt. Hvis testen er positiv sker ingenting og strikeren bliver stående, hvis testen er negativ overskrives feltet `striker.x` med et mellemrum og derefter forøges `striker.x` med 1, hvorefter strikeren gentegnes.

I denne implementation af spillet blev strikeren bygget til først at rykke sig et monospace ad gangen, derefter satte vi det op til at strikeren bevægede sig to monospaces ad gangen. Dette var vi nødt til at gøre eftersom vi styrede strikeren med keyboardet og der i keyboardets hardware er en lavere grænse for maksimumsfrekvensen hvormed en knap bliver genaktiveret når knappen på keyboardet holdes nede, i forhold til microcontrolleren der har en højere maksimumsfrekvens. Her i Basics-implementationen byggede vi banen sådan så det passede med at strikeren ved altid at rykke to felter ad gangen kunne komme ud i banens yderpositioner.

- **Reflex-logik på kanter og hjørner**
- **Game start og game over**

Vi vil i det følgende forklare hvordan disse delmål blev implementeret.

Hvordan implementerede vi keyboardinput? `kode tekst tekst`

2.2 Advanced

Efter alle basics var færdigimplementeret, havde vi planlagt at lave nogle mindre udvidelser til spillet. Disse omfatter dels implementation af liv og pointsystem, men også videreudviklinger af basic-funktionaliteterne, sådan så deres bagvedliggende virkemåde gøres mere avanceret. Advanced inkluderer følgende.

- **Liv.** Som i ethvert andet arkade-spil har vi implementeret liv, sådan så man starter med tre og bliver Game Over, når der er nul liv tilbage.
- **Score.** Point-systemet er opbygget sådan så det giver 1 point hver gang man rammer en brik og et ekstra point når brikken dør (går fra 1 til 0 liv). Det giver ingen point når bolden rammer strikeren, hvilket skyldes at man ikke skal have credit for tålmodighed.
- **Hastighed.** Hastigheden hvormed bolden bevæger sig afhænger dels af hvor mange point man har, dels af hvilken sværhedsgrad man spiller på. På easy bliver boldens hastighed sat op for hver 10. point man får, på medium for hver 5. point man får, på hard for hver 2. point man får og på Chuck Norris, tja hvem ved?

- **Vilkårlig startvinkel.** Når bolden skydes af bliver den sendt afsted i en vilkårlig vinkel på mellem 45 og 135 grader. Det vil sige lodret op fra strikeren plus minus op til 45 grader. Dette er implementeret for at man ikke kan time startvinklen så man er sikker på den altid rammer et bestemt sted.
- **Internt 18.14 koordinat-system.** I basic-implementationen af spillet satte vi bolden til at rykke sig et bestemt antal monospaces hver gang den bevægede sig. Men med denne implementation er hele banens indre struktur blevet gentænkt sådan så bolden bevæger sig som en vektor i et koordinatsystem. Vi har lavet en `Ball struct` sådan så bolden hele tiden kender sine egne (x,y)-koordinater, samt dens egen enhedsvektor (altså retning hvori den bevæger sig). Når bolden bevæger sig sker der det at dens vektors (x,y)-koordinat lægges til boldens (x,y)-koordinat. Derefter tjekkes om bolden er død eller har ramt en brik, en væg, et hjørne, strikeren eller loftet. Hvis intet af dette er tilfældet, slettes boldens gamle koordinater ved at der på disse felter tegnes blanke mellemrum. Derefter afrundes boldens (x,y)-koordinater til nærmeste heltal ved at kigge på 1. bit efter kommaet, det vil sige boldens x- hhv. y-koordinats 19. bit (Da det tænkte komma er sat mellem 18. og 19. bit). Hvis denne er 0 rundes ned og ellers rundes op. Nu kendes boldens (x,y)-koordinat i heltal og den kan derfor indtegnes på banen.
- **bolden bevæger sig med samme hastighed i både x- og y-retning.** Vi havde et problem med at det på skærmen lignede at boldens hastighed variede meget afhængigt af om dens vektor var relativt vandret (lille absolut y-komponent) eller relativt lodret (lille x-komponent). Dette skyldes at monospace-karakteren er meget tæt på at være dobbelt så høj som den er bred. Vores løsningen på dette var at gøre boldens vektors x-komponent dobbelt så stor, ved at bitshifte den 1 til venstre. På den måde er bolden ikke længere en enhedsvektor med mindre dens y-komponent er nul. Når bolden så rammer ind i noget bitshiftes `ball.vector.x` 1 til højre (divideres med to), for at gøre boldens vektor til enhedsvektor igen, sådan så der ikke opstår rod når boldens vektor roteres.
- **Striker-zoner.** Strikeren er opbygget sådan så den altid skal bestå af et lige antal felter. Dens midterste venstre felt har en afbøjningsvinkel på indgangsvinkel plus 0 grader, og dens yderste venstre felt har en afbøjningsvinkel på indgangsvinkel plus 45 grader. Hvert felt mellem det midterste venstre til det yderste venstre felt, har så en stigende afbøjningsvinkel, hvor den vinkel der bliver lagt til hvert felt er 45 grader divideret med antallet af felter fra det midterste venstre (eksklusiv) til det yderste venstre (eksklusiv). Højresiden af strikeren fungerer på præcis samme måde, bare med omvendt fortegn, sådan så afbøjningsvinklen på midterste højre felt er lig indgangsvinklen og afbøjningsvinklen på yderste højre felt er lig indgangsvinklen minus 45 grader. Strikeren er lavet på denne måde så dens bredde er meget fleksibel. Fx bruges der forskellige Striker-størrelser på spillets forskellige sværhedsgrader og dette er så bare implementeret ved at ændre på størrelsen af `striker.width`. Så sørger spillet selv for at inddele Striker-zonerne.

Der er desuden implementeret en minimums-afbøjningsvinkel på 15 grader og en maksimums-afbøjningsvinkel på 165 grader. På samme måde som når bolden rammer noget andet, bitshiftes boldens vektors x-komponent 1 til højre når bolden rammer strikeren, sådan

så boldens vektor igen er en enhedsvektor. Derefter beregnes boldens afbøjningsvinkel. Hvis dennes y-værdi er mindre end 0,25, svarende til sinus til ca. 15/165 grader, ved vi at boldens vinkel er enten under 15 grader eller over 165 grader efter afbøjning og afbøjningsvinklen bliver derfor sat til 15 grader hvis `ball.vector.x` er positiv og til 165 grader hvis `ball.vector.x` er negativ.

- **Stor bold.** I `Ball struct` er der ud over x- og y-koordinater og en vektor to andre variable som er bredde (`width`), højde (`height`). Den bold vi bruger i spillet har en højde på 2 og en bredde på 4. Dette komplicerer en del ting eftersom bolden nu både kan ramme alle objekter (Striker, vægge, loft, brikker) på mange flere måder. Desuden er der en grænse for hvor hurtigt uarten kan arbejde, selvom vi har sat baudraten op til maksimum (115200), dette kan godt gå hen og blive lidt problematisk, da uarten ved en vis hastighed af bolden ikke kan nå at slette og gentegne den ordentligt. Som løsning på dette, har vi sørget for at bolden ved en vis hastighed kun tegnes hver anden gang, hvilket letter uartens arbejde en smule. Læs mere om dette under afsnittet om **Rettelser og fintuning**.
- **Putty Terminal.** Som en del af de avancerede mål, skiftede vi fra Hyper Terminal over til Putty Terminal. Den eneste grund til dette er man i Putty Terminal kan lave meget større opløsning, sådan så spillet ser flottere ud og kan spilles som et fullscreen-spil. De eneste komplikationer der var forbundet med dette var at nogle få af ASCII-karaktererne blev tegnet på en anden måde i Putty end tiltænkt, selvom både Hyper og Putty terminalerne bruger charset 850. Løsningen på dette var bare at bruge nogle andre ASCII-karakterer.

2.3 Brikker

- Tegne brikker. DU KAN SE et billede af brikkerne her [3](#)
 - Baner der let kan gemmes i et array i ROM'en
 - Levels i ROM
- Tjekke om man rammer brikker
 - Højre/venstre og oppe/nede
 - Kanter?
- Trække liv fra brikkerne
- Lave deflect på baggrund af om der er brikker omkring brikken
 - Forklare tilfældene med at ramme to brikker af gangen
 - Forklare tilfældene med at ramme flere hjørner
- Briklogik korrigeret fordi der bevæges 2 karakterer i x-retningen

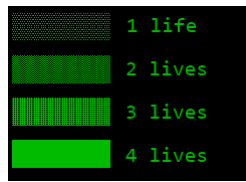


Figure 2: Sådant ser brikkerne ud.

2.4 Display

- Videobuffer
- Opdatere LED i interrupt

2.5 Styling med rat

- Gameport breakout board
- Gameport driver
- ADC-converter
- (Kalibreringsrutine og linearitet)
- Manuel kalibrering

2.6 Rettelser og fintuning

- Random vinkel den bliver roteret hver gang den reflekteres for at bolden ikke bliver fanget i mønstre
- Justeret hastighed og lavet så den kun tegner bolden hver anden gang

✗

2.7 Brikker

Formålet med at implementere brikker i spillet var at give gameplayet en helt ny dimension.

- **Tegne Brikker.** Du kan se et billede af brikkerne her [3](#) For lettest at kunne holde styr på brikkerne har vi lavet en struct, `Brick` der repræsenterer en brik. Den indeholder position i x- og y-koordinater, bredde, højde og brikkens liv. Brikkerne i en bane er gemt i et array: `Brick bricks[BRICK_TABLE_HEIGHT][BRICK_TABLE_WIDTH];`. Når banen initialiseres gennemløbes dette array og hver brik tildeles koordinater, bredde, højde og liv. Efter dette tegnes brikken. Hvis brikken har 0 liv svarer det til at der ikke er en brik. Ved at give de forskellige brikker i array'et forskellige antal liv kan man lave forskellige brikmønstre og på den måde kan man lave forskellige baner.

- [illegible]

- 6

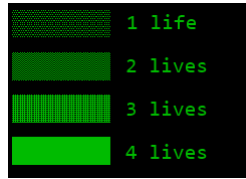


Figure 3: Sådan ser brikerne ud.

3 Konklusion

Efter spillet et blevet designet, skrevet og uploadet til Zilog 6403 microcontrolleren, er alle spillets facetter blevet gennemtestet og fundet i overensstemmelse med det forventede resultat. Vi kan derfor konkludere at den skrevne kode implementerer spillet Reflex Ball på microcontrolleren og i hyperterminalen korrekt i forhold til specifikationskravene. Derudover kan vi også konkludere at alle udvidelse virker som forventet.