
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

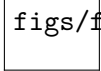
I dette programmeringsprojekt har vi skrevet og designet al koden til spillet Reflex Ball. Koden er skrevet i C i compileren Z8Encore! og implementeret på en Zilog 6403 microcontroller.

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 Specifikationer	1
3 Struktur	2
3.1 Basics	2
3.2 Advanced	5
3.3 Brikker	7
3.4 Helhedsindtryk	7
3.5 Styring med rat	7
3.6 Rettelser og fintuning	7
3.7 Brikker	8
4 Konklusion	11

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 Specifikationer

Nedenfor er vores forskellige delmål i projektet. Spillet skal have de nedenstående funktioner. Listen kan derfor ses som kravene til hvad vores spil skal kunne og er samtidig opdelt på en logisk måde så man kan starte fra toppen af og arbejde sig igennem delmålene indtil man har et færdigt spil der fungerer og ser ud efter vores hensigt. Delmålene er samtidig bygget op på en sådan måde, at skulle der ikke være tid til at nå det hele, har man stadigvæk et fuldt funktionelt spil. Det vil altså sige at vi arbejder fra basale funktionaliteter ud mod mere specifikke og detaljerede elementer.

Delmål Basics

- Detektere tastatur-input så keyboardet kan bruges til at styre striker
- Tegne banen
- Bevæge strikeren
- Bevægelig bold
- Bolden reflekteres af striker, loft og vægge
- Det skal detekteres når bolden er under strikeren (man er død)

Delmål advanced

- Liv-system, så man har 3 liv inden man bliver Game Over
- Pointsystem

- Ændring i boldhastighed som ens score stiger
- Internt 18.14 koordinatsystem, så boldens bevægelse kan laves som en enhedsvektor der kan roteres rundt
- Bolden skal starte i en tilfældig vinkel
- Striker-zoner, så boldens afbøjningsvinkel afhænger af hvor den rammes
- Stor bold, der fylder 4x2 monospace-karakterer.
- Forskellige sværhedsgrader
- Brikker og forskellige baner

Brikker

- Banen skal indeholde brikker
- Der skal være forskellige typer af brikker (forskellig antal liv)
- Der skal være forskellige levels med brikker i forskellige mønstre
- Når bolden rammer brikker og kanter skal den reflekteres på en logisk måde så fysikken i spillet ser realistisk ud

Helhedsindtryk

- LED display med score, liv og levels
- Menu hvor man kan vælge sværhedsgrad
- Beskeder ved Game Over
- Besked hvis spillet vindes
- Implementering af ASCII-Art tekst og billeder til Menu, Game Over og Win

Hardware

- Styring med intern hardware (knapper på microcontrolleren)
- Tilslutning af ekstern hardware (DEXXA Steering Wheel)

Vi har lagt en del vægt på at spillet skal se realistisk ud og at bolden, strikeren og brikkerne skal flytte sig på en realistisk måde. Vi vil forsøge at lave spillets fysik så dette er opfyldt, så bolden bevæger sig så naturligt og logisk som muligt.

3 Struktur

3.1 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.** Vi har valgt at man i basic implementeringen af spillet laver al styring vha. keyboardet, da knapperne på microcontrollerne var al for slidte og derfor virkede dårligt. Vi implementerede det sådan så man bevæger strikeren ved hjælp af piletasterne på tastaturet og sender bolden af sted ved space-tasten. Hvis man dør og bliver Game Over startes spillet forfra ved tryk på space-tasten. Der er dog et 1000ms delay fra man dør til spillet kan startes forfra. På den måde kommer man ikke til at starte spillet forfra uden at opdage at man døde.
- **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 implementering med et venstredrejet koordinatsystem, hvor y-aksen er inverteret i forhold til et almindeligt koordinatsystem, så y2 har altid en større værdi end y1.
- **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.
- **Bevægelig bold.** Vi har indstillet timeren til hver gang der er et tick bevæger bolden sig. Her i basics implementationen hardcodede 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 fra strikeren. Dette skyldes at vi jo arbejder med en inverteret y-akse i forhold til et almindeligt koordinatsystem. Det fungerer på den måde at vi har et `Ball struct` sådan så bolden hele tiden kender sin egen position i form af

x- og y-koordinat, samt en vektor (den retning og hastighed den bevæger sig i). Så når bolden rykker sig, overskrives feltet som er boldens (x,y)-koordinat med et mellemrum (da bolden i basic implementationen kun fylder et enkelt felt) og derefter tillægges boldens vektor til boldens (x,y)-koordinat og den gentegnes.

- **Reflex-logik på kanter og hjørner.** Hver gang bolden har rykket sig, inden den tegnes, laves en række tjek for at se om bolden har ramt en væg, et hjørne eller loftet. Bl.a. tjekkes om boldens x-koordinat bliver større end eller lig med banens x2-koordinat. Hvis dette er tilfældet ved vi at bolden har ramt højre væg af banen og bolden tegnes ikke umiddelbart. I stedet trækkes boldens vektor fra boldens nye (x,y)-koordinat, sådan så bolden er tilbage på den plads den stod på inden den ramte ind i væggen, derefter inverteres boldens vektors x-koordinat og vektoren lægges til boldens position og bolden tegnes igen på næste tick. Der udføres altså væsentligt flere operationer hver gang bolden rammer en væg i forhold til når den ikke gør. Når bolden rammer venstre væg fungerer der på præcis samme måde, bortset fra at tjekket er om boldens x-koordinat bliver mindre eller lig med banens x1-koordinat.

Loftet fungerer også mere eller mindre på samme måde. Her tjekkes bare om boldens y-koordinat kommer under banens y1-koordinat (husk y-aksen er inverteret). Hvis dette er tilfældet trækkes boldens vektor fra boldens position, boldens vektors y-komponent inverteres og vektor lægges til bolden inden den tegnes igen.

- **Game Start og Game Over.** Når spillet starter kan man rykke strikeren rundt som normalt. Her følger bolden med sådan så den bliver ved med at være på midten af strikeren. Det fungerer på den måde at når variabelen `gameStarted` er 0 så gentegnes både strikeren og bolden hver gang man rykker strikeren. Både boldens og strikerens x-koordinat sættes herefter. På samme måde som når strikeren rykkes normalt laves de samme check for om strikeren har bevæget sig ud over banens vægge. Når der så trykkes på space-tasten på tastaturet skydes bolden afsted og `gameStarted` sættes til 1, og når strikeren rykkes fra nu er det kun strikeren der gentegnes.

På samme måde som bolden hver gang den har bevæget sig, inden den tegnes, testes for om den har ramt væg eller loft, tjekkes også om boldens y-koordinat er større end eller lig med banens y2-koordinat minus 2. Hvis denne test er sand betyder det at bolden ligger på feltet lige over strikeren eller et felt under det. Her testes så om boldens x-koordinat ligger indenfor strikeren, dvs. imellem `striker.x` og `striker.x + striker.width`. Hvis dette er tilfældet bliver bolden skudt op igen, hvilket fungerer ved at boldens vektors y-komponent bliver inverteret, på samme måde som når bolden rammer loftet. Hvis bolden er uden for strikeren bliver variabelen `alive` sat til 0 og der bliver displayet **Game Over!** midt på skærmen. I dette state kan man ikke bevæge strikeren. Dette har vi implementeret fordi man ellers kan 'køre bolden over', hvilket ikke ser så grafisk pænt ud. Når der trykkes space igen gentegnes hele spillet banen og om variablerne `gameStarted` sættes til 0 og `alive` sættes til 1.

3.2 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. Som en ekstra feature har vi implementeret at man får 2 ekstra liv, når en bane er gennemført.
- **Pointsystem.** Scoren er opbygget sådan så det giver 1 point hver gang man rammer en brik. Vi har valgt at det ikke skal give nogle point når bolden rammer strikeren, hvilket skyldes at man ikke skal have credit for tålmodighed.
- **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.
- **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.
- **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 baud-raten 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**.
- **Sværhedsgrader og ændring i boldens hastighed.** I spillet er der implementeret fire forskellige sværhedsgrader. Disse er Easy, Medium, Hard og Chuck Norris. Forskellen på sværhedsgraderne er hvor bred strikeren er, hvor hurtig starthastighed bolden har og hvor mange point man skal have før at boldens hastighed bliver sat op. På Easy er `striker.width = 30` og boldens starthastighed er 40ms. Det vil sige der går 40ms fra bolden bliver tegnet, til den tegnes igen. På Medium er `striker.width = 20` og boldens starthastighed 40ms, på Hard er `striker.width = 10` og boldens starthastighed 40ms og på Chuck Norris er `striker.width = 4` og boldens starthastighed er 10ms.

Som man får flere point sættes boldens hastighed op. På Easy bliver delayet mellem to på hinanden følgende gange hvor bolden tegnes, sat 1ms ned for hvert 10 point man får. På Medium ved hvert 5. point man får og på Hard ved hvert 2. point man får. Når delayet mellem bold-aftegningerne kommer under 20 ms begynder bolden kun at blive tegnet hver anden gang for at uarten kan følge med. Minimumshastigheden som bolden kan bevæge sig i er 10ms delay mellem aftegningerne. Grænsen er sat her fordi spillet ellers bliver for umuligt at gennemføre. Da Chuck Norris-sværhedsgraden starter på 10ms delay øges boldhastigheden altså ikke uanset hvor mange point man får. Når der skiftes bane bliver boldhastigheden reset'et til starthastigheden ved den givne sværhedsgrad. Dvs. 40ms delay på Easy, Medium og Hard og 10ms på Chuck Norris.

- **Brikker og baner.** Som en del af de avancerede mål havde vi at lave brikker, så gameplayet bliver sjovere. Se mere om implementering af brikker, forskellige brik-typer og forskellige baner i afsnit **3.3**.

3.3 Brikker

3.4 Helhedsindtryk

- **LED display med score, liv og levels** Skriv om videobuffer og om opdatering af LED i interrupt HER
- Menu hvor man kan vælge sværhedsgrad
- Beskeder ved Game Over
- Besked hvis spillet vindes
- Implementering af ASCII-Art tekst og billeder til Menu, Game Over og Win
 - HUSK at nævne Java-program til ASCII-Art strenge
- Opdatere LED i interrupt

3.5 Styring med rat

Som et delmål fandt vi hurtigt ud af vi ville have et DEXXA Steering Wheel sluttet til micro-controlleren, sådan så styringn

- Styring med intern hardware (knapper på microcontrolleren)
- Tilslutning af ekstern hardware (DEXXA Steering Wheel)
 - Gameport breakout board
 - Gameport driver
 - ADC-converter
 - (Kalibreringsrutine og linearitet)
 - Manuel kalibrering

3.6 Rettelser og fintuning

- **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-komposant) eller relativt lodret (lille x-komposant). 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-komposant dobbelt så stor, ved at bitshiftte den 1 til venstre. På den måde er bolden ikke længere en enhedsvektor med mindre dens y-komposant 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.

- **skift til 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.
- **Implementering af en lille smule vilkårlighed ved hver afbøjning.** Som i professionelle computerspil, har vi sørget for at implementere en mindre vilkårlighed i hvor meget bolden afbøjes hver gang den roteres, dvs. både når den rammer striker, brikker, loft eller vægge. Bolden får da sin almindelige beregnede afbøjningsvinkel med en adderet pseudovilkårlig rotation på mellem minus 3 og plus 4 ud af 512. Det vil sige en vinkel på mellem ca. minus 2,1 og plus 2,8 grader. Dette er en vinkel lille nok til man med øjet aldrig ved ligge mærke til afbøjningsvinklen ikke er helt lig med indgangsvinklen på brikker, loft og vægge, men samtidig stor nok til at bolden ikke bliver reflekteret rundt sådan at den er fanget i et fast mønster.
- **Justering af hvor ofte bolden tegnes.** Som nævnt i afsnit 3.2 havde vi lidt problemer med at uarten ikke altid kunne tegne bolden hurtigt nok, efter vi lavede bolden til at fylde 4x2 monospace-karakterer. Vi oplevede at hvis delayet mellem aftegningerne af bolden blev for lille, så kunne man risikere ofte ikke at kunne se bolden, da vores kode jo er lavet på den måde at bolden først slettes, derefter bevæger sig og så tegnes igen. Altså uarten kunne ikke nå at gennemføre hele processen med det resultat at boldene kun slettedes og ikke blev tegnet ordentligt op igen. Desuden opleve vi flere gange at terminalen frøs, hvis vi prøvede at tegne bolden for hurtigt. Som en løsning på dette justerede vi hastigheden sådan så når der er under 20ms delayet mellem aftegningerne af bolden, så tegnes den kun hver anden gang. Dette letter arbejdet for uarten nok til at det bolden ser ordentlig ud hele tiden. Vi eksperimenterede også med kun at tegne bolden hver 3 gang, men det ser for forstyrrende ud for øjet, så vi fjernede det igen.

3.7 Brikker

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

- **Tegne Brikker.** 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 brikkerne i array'et forskellige antal liv kan man lave mønstre med brikkerne så man på den måde har flere forskellige baner i spillet. På Figur 2 er der vist hvordan brikker med forskelligt antal liv er tegnet. Jo flere liv de har, des mere solide tegnes de. Som et lille twist i gameplayet har vi valgt at gøre brikker med mere end fem liv usynlige så de pludseligt kan dukke op når man rammer dem.

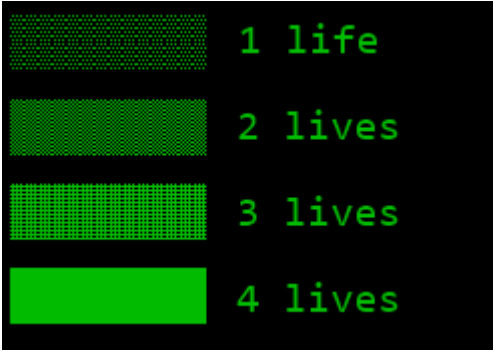


Figure 2: Brikker med forskelligt antal liv tegnes forskelligt

- **Baner gemt i et array i ROM'en.** Vi har valgt at hardcode brikkernes bredde og højde for at spare plads når vi gemmer banerne i ROM'en. På denne måde kan en bane i ROM'en gemmes i arrayet `unsigned char rom_levels[4][BRICK_TABLE_HEIGHT][BRICK_TABLE_WIDTH]`. Dette array er et tredimensionalt array af chars der er gemt i ROM'en. Det indeholder fire "lag" der hver indeholder de todimensionale data til en bane. Værdien af hver char er antallet af liv den tilsvarende brik får. Når banen initialiseres, løbes dette array og arrayet `bricks[][]` igennem og brikkerne i `bricks[][]` får det antal liv der står i `levels[][][]` arrayet. Den dynamiske hukommelse i mikroprocessoren indeholder således kun et array af brikker der svarer til den aktuelle bane. Et eksempel på hvordan en bane er gemt i arrayet `levels[n][][[]]` er vist nedenfor.

[illegible]

Banen der svarer til det ovenstående array, er vist på Figur 3.


figs/level1.png

Figure 3: Level 1 i spillet

- **Tjekke om man rammer brikker.** Den helt store udfordring med brikkerne var at tjekke om bolden ramte dem. Ved hver eneste iteration af spillet gennemløber vi arrayet `bricks[] []` og for alle brikker med mere end 0 liv tjekker vi om bolden har ramt og i givet fald hvordan.

Når spillet itereres og bolden flyttes tjekker vi først om bolden rammer brikkerne før vi tegner den. Hvis bolden har ramt en brik reflekteres den og får sin nye position før den tegnes. Hvis bolden har ramt en brik, er boldens koordinater altså inde i brikken når vi tjekker.

For at bolden skulle bevæge sig lige hurtigt i begge retninger på skærmen har vi lavet x-komposanten af boldens vektor dobbelt så stor. Det giver nogle ekstra udfordringer når bolden rammer brikkerne. Når bolden kan bevæge sig 2 karakterer i x-retningen kan den ramme brikkerne fra siden på en række forskellige måder. Disse er vist for højre side af brikken på Figur 4. Når vi har itereret spillet og tjekker boldens position kan den på x-aksen befinde sig både en eller to karakterer inde i brikken. Den kan således ramme brikken på seks forskellige måder som vist på ovennævnte figur. Det er vigtigt at tage højde for at bolden kan ramme to karakterer ind i brikken når man tjekker om bolden har ramt brikken for siden i midten. Hvis man ikke tager højde for dette, vil bolden bevæge sig igennem brikken mens den reflekteres af toppe og bunden.

For at give et godt gameplay har vi besluttet at bolden fortrinsvis skal reflekteres ned og op så brugeren kan få den. Dette er vist på Figur 5. Når bolden rammer brikken fra toppen og er 2 karakterer inde i brikken vil den blive reflekteret som vist på figuren. Hvis bolden havde ramt på samme måde, men var kommet nedefra var den blevet reflekteret som om den havde ramt siden af brikken.

Det der er vist på denne figur er, bortset fra de omgivende brikker, samme situation som på Figur 4 Top, 2 brikker inde.

- 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

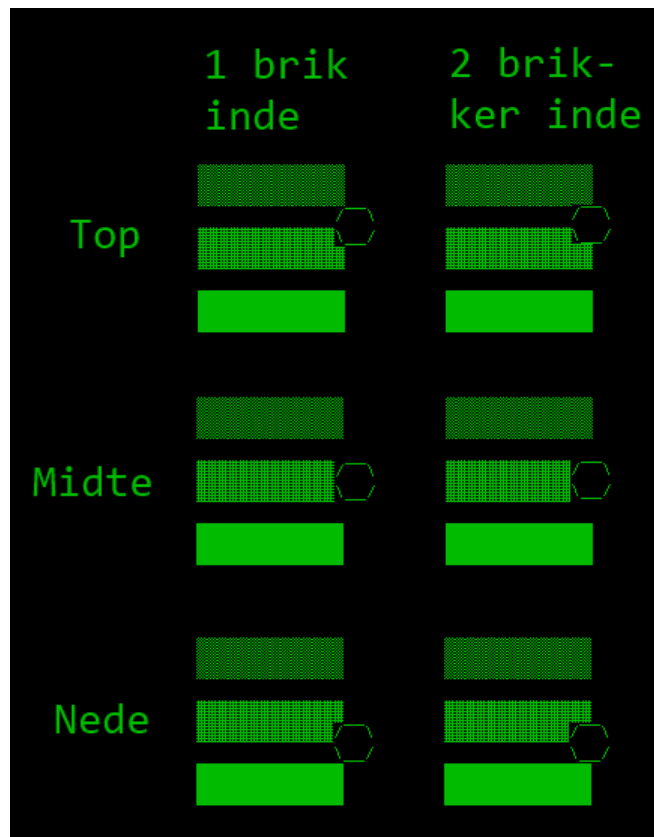


Figure 4: Forskellige måder bolden kan ramme fra siden

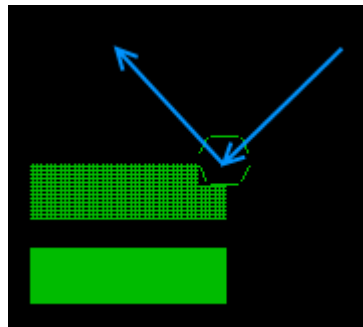


Figure 5: Forskellige måder bolden kan ramme fra siden

4 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.