

XYZ-Pilot

1 Inledning

Den här tekniska rapporten behandlar ett projektarbete i kursen TSEA83: Datorkonstruktion på Linköpings Universitet.

1.1 Bakgrund

Kursen TSEA83 läses av studenter på civilingenjörsprogrammet datateknik och avser att ge studenter kunskap om hur datorer fungerar i dess minsta beståndsdelar. Förutom projektarbetet består kursen av en förberedande laborationsdel som lägger behandlar de grundläggande kunskaperna i processorkonstruktion och VHDL-kod.

1.2 Syfte

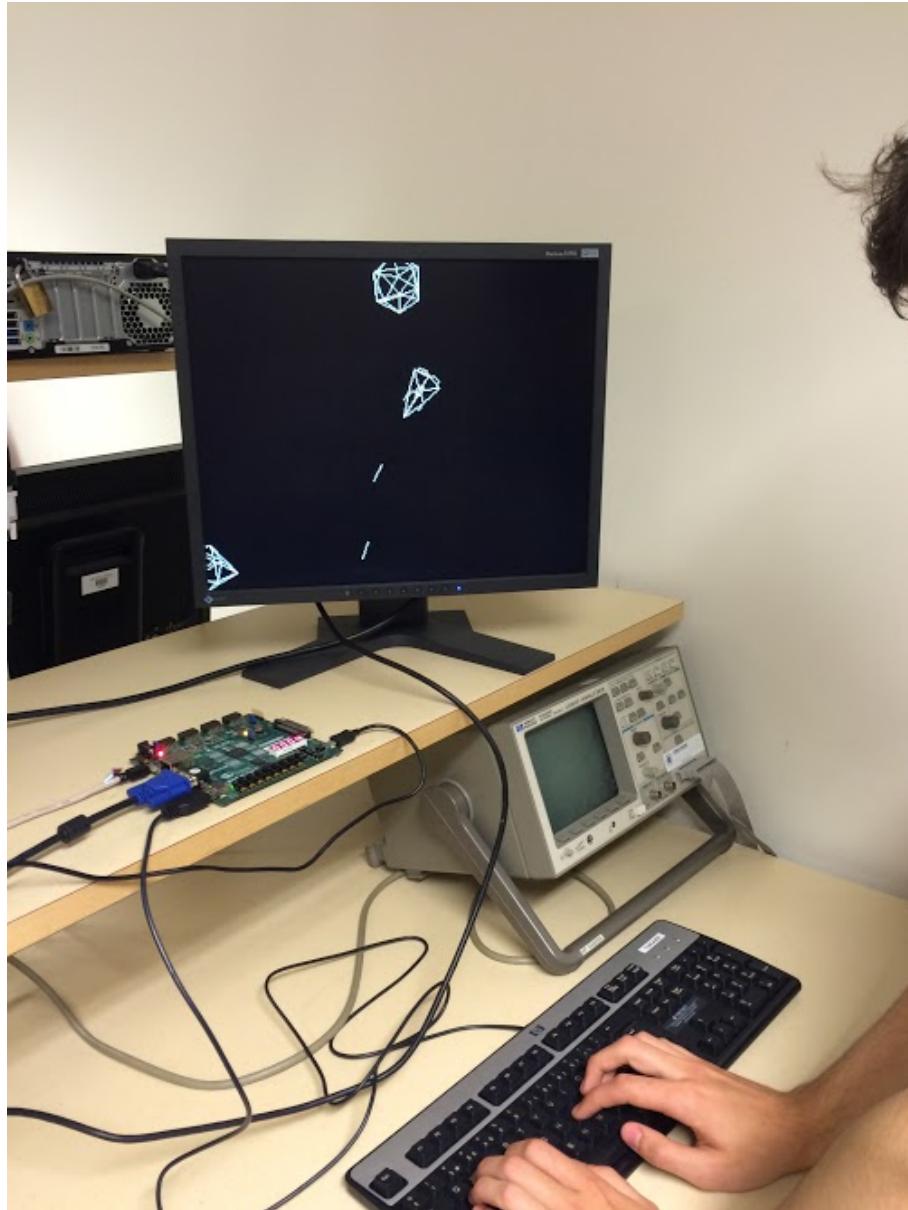
Denna rapport syftar att ge en förståelse för vår konstruktions arkitektur och tekniska detaljer samt vilka lösningar vi tillämpat för att nå vår färdiga produkt. Läsaren ska efter att ha tagit del av rapporten ha förståelse för apparatens olika komponenter och de grundläggande teorier som krävs för detta. Rapporten förklrar också hur apparaten används för att fungera.

1.3 Källor

TODO

2 Apparaten

Vår apparat använder ett tangentbord för input och visar sedan spelet på en skärm med hjälp av VGA. Programkoden, som innehåller instruktioner för processorn, laddas in via UART. Det program vi valt att realisera för apparaten är ett rymdspel som går ut på att skjuta sönder asteroider för poäng. Poängen man får syns hexadecimalt på 7-segments-displayen. Figur 1 visar en bild som beskriver hur konstruktionen ser ut.

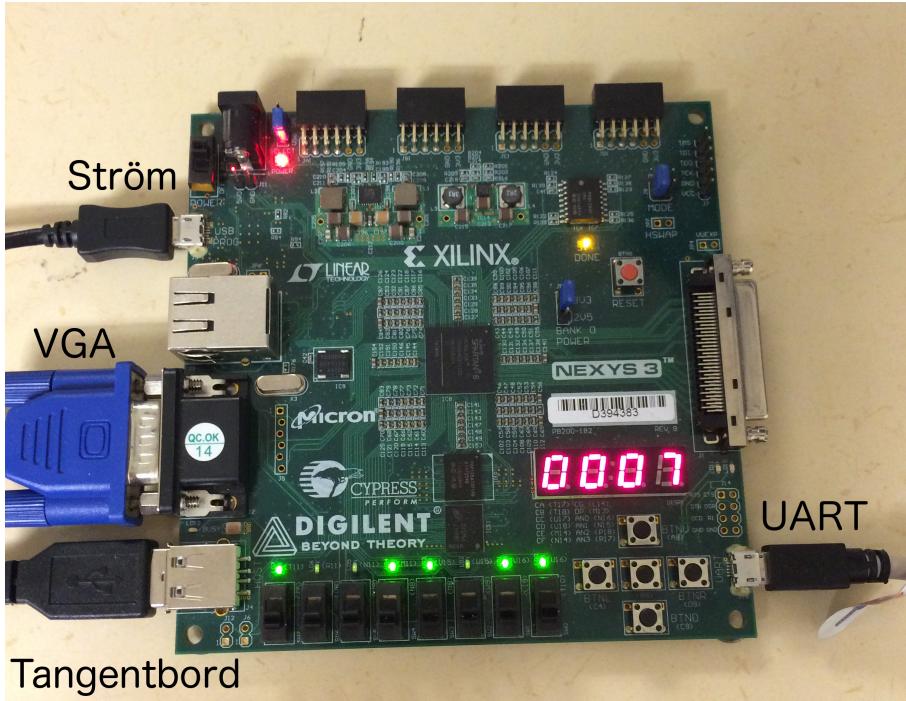


Figur 1: Med hjälp av tangentbordet är det möjligt att styra skeppet på skärmen och skjuta asteroider

Konstruktionen byggs genom att koppla in VGA från skärmen, UART och strömsladd från valfri USB-port på en dator samt en tangentbordssladd i FPGAn enligt figur 2. För att starta programmet behöver du vara inne i mappen XYZ-Pilot/assemblers och skriva in kommandot:

```
./assemble.sh game.asm
```

Detta skript programmerar hårdvaran och laddar in mjukvaran via UART. Inom kort borde skeppet synas på skärmen och tangentbordet kan användas för att spela.



Figur 2: Bilden beskriver de sladdar som ska kopplas in i kortet

3 Teori

3.1 3D-grafik

3d objekt representeras oftast som en mängd vektorer som beskriver hörnpunkter i objektet, linjer som binder ihop dessa vektorer och ytor som binder ihop dessa linjer. I vårat projekt finns inga ytor och för enkelhetens skull beskriver vi inte linjer som två index till punkter utan direkt som två vektorer.

I moderna GPUer används 4 dimensionella vektorer istället för 3d vilket möjliggör rotation, skalning och translation av en vektor i en enda matrismultiplikation. Från början tänkte vi också göra det vilket är anledningen till att våra vektorer är 4 dimensionella.

3.2 Bresenhams linjealgoritm

Bresenhams linjealgoritm utvecklades på IBM under tidigt 1960-tal och hade den stora fördelen att dess implementation endast kräver heltalsaddition, subtraktion och shiftande av bitar. Eftersom algoritmen är så pass snabb och simpel finns den realiseras i många existerande grafikprocessorer och vilket är varför vi valt att använda den i vårt projekt.

Algoritmen bygger på att räkna ut felet som uppstår jämfört motstående axel när algoritmen rör sig från början till slut. Blir felet tillräckligt stort korrigeras det och felet minskar då med en enhet. Felet ökar med absolutvärdet av kvoten mellan komponenternas längd för varje passerad pixel i algoritmens ursprungliga form, som då kräver decimaltalsaritmetik. En implementation av detta följer nedan.

```

func drawLine(x0, y0, x1, y1)
    dx = x1 - x0
    dy = y1 - y0
    e = -1
    de = abs(dy / dx)

    x = x0
    y = y0

    while x < x1 do
        fill(x, y)
        e += de
        if e >= 0 then
            y += 1
            e -= 1
        endif
        x += 1
    endwhile
endfunc

```

I och med att det är mer kostsamt och ineffektivt att använda decimaltalsaritmetik än heltalsaritmetik var en viktig egenskap med just Bresenhams algoritm att den går att skriva om för detta. En annan begränsning med algoritmen i dess grundutförande är att den endast fungerar för linjer som rör sig med riktningen av den sista oktanten och därfor ser den algoritm som är implementerad i vår hårdvara aningen mer komplicerad ut än grundfallet. Vad som sker i hårdvaran går att göra enklare genom att använda transformering av in- och utdata beroende på vilken oktant linjen rör sig i istället för att använda if-satser.

```

func drawLine(x0, y0, x1, y1)
    dx = abs(x1 - x0)
    dy = abs(y1 - y0)
    x = 0
    y = 0
    i = 0

    x_incr = 1 if (x1 - x0) > 0 else -1
    y_incr = 1 if (y1 - y0) > 0 else -1

    if dx > dy then
        len = dx
        D = 2 * dy - dx
    else
        len = dy
        D = 2 * dx - dy
    endif

    while i < len do
        fill(x, y)
        if D > 0 then
            if dx > dy then
                y += y_incr
                D -= 2 * dx
            else
                x += x_incr
                D -= 2 * dy
            endif
        endif

        if dx > dy then
            x += x_incr
            D += 2 * dy
        else
            y += y_incr
            D += 2 * dx
        endif

        i += 1
    endwhile

    fill(x, y)
endfunc

```

4 Hårdvaran

Vår konstruktion består i grunden av en hårdvaruaccelererad grafikprocessor för 3-dimensionella objekt och modeller som utökas med en centralprocessor som kan manipulera objekten som renderas och även utföra aritmetik på tal och vektorer. Programmet som körs på apparaten programmeras vid start via UART. En översikt av konstruktionen ses i blockschemat i figur ??.

Figur 3: Översiktligt blockschema över konstruktionen