1 Övning 5

Det är meningen att man skall titta i detta häfte på datorn eftersom det innehåller ledtrådar som är gömda i filen (peka på dem så dyker det upp en liten ruta), det står

3 **1** Detta är en uppgift med svårhetsgrad 3.

L1

Kodlösningar kommer i vanlig ordning att läggas upp en tid efter föreläsningen. Det finns många olika lösningar till övningarna, men det är rekommenderat att ni tittar på facit i efterhand. Ofta hittar man alternativa lösningar eller ett snyggare sätt att skriva.

1.1 DXF format

I denna övning skall vi lära oss att skriva enligt DXF syntaxen. Öppna **Ex_5_1.cpp**.

För att lättare kunna ta till sig informationen i dessa övningar så föreslår jag att ni skummar igenom koden först.

1.2 DXF

1 **0** I denna uppgift skall vi komplettera koden i **Ex_5_1.cpp**. Öppna filen och kolla igenom lite först.

Först skall vi definiera en global variabel **pi**, låt den vara 3.1415, eller **4*atan(1)**, glöm inte att lägga till **math.h**.

2 **1** I denna uppgiften skall vi bilda en cirkel. Skriv i funktionen **WriteDXF**.

Om ni är osäkra på hur vi skriver till filer, titta i föreläsning 3 "skrivning till fil". Titta också på föreläsning 5 "DXF" för mer information om syntaxen för **line**.

Låt antalet segment som cirkeln skall vara uppbyggd av vara n. Tänk på att vi skriver ut linjesegment, så vi måste "skicka" punkter två gånger.

När ni skrivit till fil, öppna filen i Rhino med File ⇒ **Import**...

Om ingenting dyker upp i Rhino, försökt att se om ni kan markera något runt origo, om inget gult (selection) syns, så har ni gjort ett misstag i er DXF export.

L1 **L2** **L3**

---

1 **2** I denna uppgiften skall vi utöka **line** med en dimension till. Vi skall skapa två cirklar där linjern skall dras mellan, som en cylinder. Låt cirklarna vara på ett visst avstånd från varandra i z-led.

Skriv en funktion **line(x1, y1, z1, x2, y2, z2)** som skriver ett linjesegment som har startkoordinat (x_1, y_1, z_1) och slutkoordinat (x_2, y_2, z_2), direkt till fil. Det blir då mycket enklare att koda resten.

L1 **L2**

---

3 För att få fram nåt lite mer intressant skall vi här modifiera ovanstående uppgift genom att låta den ena cirkeln starta senare än den andra. Vi vill få fram en Hyperboloid **klicka** . 2

L1

---

4 Gör så de olika linjerna ligger i olika lager. Säg att varannan linje ligger i ett lager som vi kallar "Even" och det andra i "odd". I Rhino kan vi då välja de enskilda lagrena för att förvissa oss om att vi gjort rätt. 1

L1

---

5 Skapa cylindrar (använd DXFs Circle) som ligger på slumpmässiga platser inom en cirkelskiva (2d). Vi skall inte packa cylindrarna inom cirkeln, det är ett lite mer komplicerat problem. 4

L1

---

6 Skapa ett torn av cylindrar (3d). Bygg tornet nedifrån och upp. Varje cylinder skall placeras på den undre. För varje cylinder skall radien minskas med höjden på positionen. Ni får gärna själva ändra proportionerna, säg invers kvadratisk. 2

---

7 Skapa en sadelyta med 3DFace, se föreläsning 5 för exempel. 4

L1

1.3 Processing

I denna övningen skall vi testa *Processing*. Tanka ner Processing och spara på **d:** (exempelvis) från hemsidan **här** , den skall också finnas uppackad på **z:** under ARK385/Processing. Processing har ett fönster där man skriver sin kod och kör, ungefär som Code::Blocks. För att köra ert program: tryck Ctrl–R, för att indentera hela filen tryck på Ctrl–T.

Tänk på att Processing väntar sig att koden är i en mapp som heter Ex_5_2/Ex_5_2.pde dvs. mappen måste heta likadant som filen.

Det som är bra med Processing är att man kan publicera sin kod, alltså låta använda köra koden direkt i en webbrowser. För att göra detta tryck på Ctrl–E. Det dyker då upp en mapp med de exporterade filerna, den som är intressant är jar–filen (om ni inte kan se filändelser klicka **här**). Jar–filen kan köras direkt i en webbrowser, dvs. om ni har en hemsida så kan man lägga upp programmet där. Man måste dock ha installerat Java.

1 **1 Denna uppgift handlar om att rita en cirkel i Processing. Använd funktionen line och komplettera koden i funktionen simulation. Använd en for–slinga med loopvariabel theta.**

Först måste ni öppna **Ex_5_2.pde** och försök att se vad som händer i koden. Tänk på att **setup** körs först[i] och alla funktioner som körs anropas därifrån förutom **draw** som körs för varje frame.

I setup definierar vi uppdateringshastigheten genom att anropa **framerate()**.

Referenssidan **här** är nyttig, använd denna flitigt. Den går också att tanka ner och läsa i Processingfönstret, se **Help** → **Reference**.

Fönsterkoordinater fungerar som för OpenGL (Game of Life, övning 5). Origo, dvs. (0,0) är översta vänstra hörnet, tänk på att translatera cirkeln till mitten av fönstret. Jag har definierat fönstrets bredd och höjd med variabeln **windowSize**.

Om ni inte ser cirkeln beror det förmodligen på att ni ritar ut enhetscirkeln, alltså $\in [-1, 1]$. Skala om koordinaterna så cirkeln blir cirka 400 pixlar i diameter, alltså med ungefär 200[ii].

Det finns redan definierad en variabel **segs** som bestämmer hur många linjesegment cirkeln skall bestå av. Vi kan då definiera en variabel **delta** som bestämmer hur många grader **theta** skall stegas ($\in [0, 2\pi]$). I Processing finns redan π definierad som **PI** (stora bokstäver).

Använd polära koordinater för att rita cirkeln, alltså

$$f(\theta) = \begin{cases} r\cos(\theta) \\ r\sin(\theta) \end{cases} \tag{1}$$

L1 **L2** **L3** **L4**

---

2 **2 Kommentera ut tidigare kod och skapa en boll (använd ellipse) som roterar kring cirkeln i föregående uppgift.**

Denna gången skall vi inte ha en for–slinga med theta utan låta en ny variabel **t** stegas ett litet steg för varje gång **simulation** anropas. Tänk på att funktionen **simulation** körs för varje "frame", stega **t** i **draw** med 0.02. Ni får ändra stegningen av **t** beroende på **a** och **b**.

---

1 **3 Gör så variabeln accumulation (se koden) blir 1 om man trycker på knappen "a", när man trycker på den igen så skall den bli 0.**

Bilderna kommer *ackumuleras* om man inte rensar bakgrunden med **background(0)**, det blir fina figurer som vi kommer se i nästa uppgift. Testa att trycka på 'a' i föregående uppgift så vi ser att det fungerar. Matlab gör ju detta också men använder vektorer och ritar med **plot**.

L1

---

2 **4** Gör om föregående uppgift så bollen följer en *Lissajouskurva*, se **länk** . En Lissajouskurva påminner om hur en fritt hängande pendel rör sig, se en "sandpendel" **Youtube** . Sätt theta till 0 för variablerna nedan, ni kan sedan se vad som händer om ni väljer theta=1 för nån kurva.

En Lissajouskurva är definierad som

$$\begin{aligned} x &= \sin(at + \theta) \\ y &= \sin(bt) \end{aligned}$$

Testa följande parametrar:

$$a = 1, b = 2.$$

$$a = 3, b = 2.$$

$$a = 9, b = 8.$$

1.4 Extra uppgifter om ni får tid över

5 Testa att göra likadant som ovan men med en *Roskurva*, se **länk** . 4

En Roskurva är definierad i polära koordinater som

$$r = cos(k\theta)$$

Alltså **r** är avståendet från origo, vi ritar cirkulärt, se formel 1 för definitionen av polära koordinater.

! Om ni bara får en cirkel, tänk på att Processing också har "heltalsdivision".

Rita $n = 4$, vilket ger en rosliknande kurva.

Rita med $n = \frac{1}{3}$ vilket ger en figur som påminner om en cardiod **cardiod** som man kan se naturligt om man lägger en metallring i solljus, se **länk** .

Pröva med $n = \frac{3}{5}$ som ger en annan intressant kurva.

L1

---

6 Ändra färgerna för cirkelskivan beroende på avståndet till mitten (windowSize/2, windowSize/2) så det ser ut lite som rörelsemängden för cirkeln visualiseras. 3

Flytta **fill** från draw till precis innan ellipse i **simulation**.

I **simulation**, skapa flyttalen **x** och **y**. Tilldela värdena som står i **ellipse**. På så sätt kan vi återanvända endast x och y i ellipse och i avståndsberäkningar utan att kopiera massa kod[iii].

Skapa ett nytt flyttal, kalla den **distance** (namnet **dist** används redan av processing). distance skall spara avståndet från (x,y) till (windowSize/2,windowSize/2). Lägg in det värdet i **fill**, som ändrar gråskalan om man ger den ett argument, vill man få färger så måste man ge den tre argument: rött, grönt och blått som är flyttal eller heltal $\in [0, 255]$.

! Får ni inte till färgerna så skriv ut vad distance blir med **println(distance+"\n");**.

[i] Detta är Processings motsvarighet till **main** i C++
[ii] Gör detta innan translation.
[iii] Man riskerar att missa detaljer om man copy/pastar kod när man testar ett program, därför är det viktigt att alltid dela upp repetativ kod i nya variabler/funktioner.

1.5 Extra information, Filformat

Filformat finns i flera utföranden, det kan handla om binära, dvs. allting sparas i ettor och nollor direkt i fil eller textbaserade format.

Ett format är endast syntax, så man vet hur man skall läsa in filen i ett annat program. Det finns många textbaserade format, exempelvis DXF, FBX, TXT, DOC,... **Code::Blocks** använder filändelsen CPP för C++-kod, och CBP (Code Blocks Project). CBP-formatet är i textformat, öppnar man filen så ser man i klartext vad som står, även om man kanske inte förstår vad det betyder. Bland annat ser man vart **freeglut.dll** befinner sig

Efter <**Linker**> i CBP-filen **Ex_4_2.cbp** ser vi följande

```
<Add library="freeglut" />
<Add library="opengl32" />
<Add library="glu32" />
<Add library="winmm" />
<Add library="gdi32" />
<Add directory="C:\\Program Files (x86)...
          \\CodeBlocks\\MinGW\\lib" />}
```

Kompilatorn länkar alltså till dessa biblioteken och utnyttjar freegluts funktioner vilket gör att vi kan kompilera OpenGL-kod.

I en datorkrash så finns det filer som mer eller mindre är toleranta mot skador. När jag stod inför ett val att spara ljudfiler, bildfiler etc. fann jag några intressanta filformat att välja bland. Det stod mellan MP3, WAV eller FLAC (lossless format), bland bildfilerna ville jag bara testa hur väl de klarar felskrivning av en hårddisk.

Efter att ha öppnat filerna i en texteditor och bytt tecken i filen (som ju är binär) så kunde man höra felet i en musikspelare respektive se felet med en bildeditor.

Resultatet av testet var att FLAC klarade av mycket ändringar (fel i hårddisk). Felen begränsades endast till området där ändringen skedde, WAV var inte så förlåtande och är okomprimerad (ungefär 40% större än FLAC). Filformat som PSD, klarar inte av ändringar väl. Små modifikationer i en PSD–fil gjorde filen helt oöppningsbar, väsentligen värdelös. MP3 ger artifakter som “fågelkvitter” i de drabbade områdena.

Kontentan är att man bör använda filformat som är mer feltoleranta och begränsar att felet propagerar över hela filen som gör att det kanske inte går att öppna filen alls.