

Shapes

— ett hyfsat funktionellt ritspråk

Henrik Tidefelt

LiTH

25 september 2007

Mål

Med den här UppLYSningen hoppas jag (kanske lite naivt)

- Att ni ska få ett hum om vad Shapes är.
- Få höra era invändningar mot designen som den ser ut idag.
- Lyckas hitta någon testpilot.
- Väcka intresse för utvecklingssamarbete.

Plan

De stora inslagen idag är:

- Beskriva hur språkets struktur ser ut idag.
- Visa lite av de funktioner som kärnan erbjuder.
- Diskutera intressanta utmaningar för framtiden.
- Små avstickare med konkreta exempelkörningar.

Introduktion

Historia

- Hösten 2003: Första kontakt med MetaPost.
- Sommaren 2004: Toolbox för plottning i Matlab tar form.
- Hösten 2004: Börjar undersöka möjligheten att ersätta MetaPost.
- September 2005: Shapes, då kallat *MetaPDF*, versionshanteras.
- Januari 2007: Shapes, då kallat *Drool*, har använts till stort antal figurer i en bok.
- April 2007: Kontrollerade tillstånd.
- September 2007: Språket heter *Shapes*, och presenteras för första gången.

Rötter

Shapes har sina rötter i många av de språk jag varit i kontakt med:

- MetaPost
- Scheme
- Haskell
- C++

Alternativ

Några andra ritspråk som finns och/eller används idag:

- MetaPost
- Asymptote
- PGF och TikZ
- Haskell PDF
- Functional MetaPost

Varför Shapes?

Givet utbudet av alternativa rit-språk, varför utveckla ett till? Här är några skäl:

- Inte funktionellt orienterade (alla utom Haskell PDF och Functional MetaPost).
- Dålig beräkningskapacitet (MetaPost och PDF/TikZ).
- Saknar domän-specifik syntax (Haskell PDF).
- Inte publicerade när Shapes påbörjades (Asymptote och Haskell PDF).
- Osmidig arbetsgång (Functional MetaPost).

Hello, shaper!

- `page << [stroke (0cm,0cm)--(1cm,1cm)]`

- `page << stroke [] ((0cm,0cm)--(1cm,1cm))`

```
[(\ •dst pth .> { •dst << stroke [] pth } )  
  •page (0cm,0cm)--(1cm,1cm)]
```

Språkets struktur

Exempel på enkla typer

- Flyttal: 14, 14.5, 1
- Heltal: '5, '~12, '0xFF
- Längd: 7cm, ~3mm, 72bp
- Sträng (mer detaljer senare): `Hej!`
- Symbol: 'left

Lexikala bindningar

Lexikala bindningar fungerar som i Scheme, men kan inte bindas om.

```
a: 42
```

Räckvidden (eng: *scope*) är begränsad till en kod-klammer (eng: *code bracket*):

```
{  
  a: 42  
  ● stdout << a  
}
```

Lexikala bindningar — detaljer

- Högerledet evalueras i samma scope som bindningen tillhör.
(Jämför letrec i Scheme.)

```
odd: \ n .> [if n = '0 false [even n - '1]]
```

```
even: \ n .> [if n = '0 true [odd n - '1]]
```

- Skuggade bindningar kan nås:

```
a: .. / a + 7
```

Dynamiska bindningar

Dynamisk bindning infördes som ett sätt att undvika den imperativa spaghetti-struktur som ett skrivbart *graphics state* lätt kan leda till.

```
@width:4bp | [stroke mypath]
```

- Dynamiska variabler inleds med @.
- Den dynamiska variabeln tillsammans med ett värde blir ett nytt värde som representerar en potentiell dynamisk bindning.
- Bindningsvärden kan kombineras:

```
@width:4bp & @dash:[dashpattern 1cm 4mm]
```
- Dynamiska bindningar sätts i scope med en “pipe”.
- En dynamisk variabel har ett filter och ett skönsvärde (eng: *default value*).

Dynamiska värden

En dynamisk variabel kan bindas till ett *dynamiskt värde*.

- Ser ut så här:
`@bigmargin: dynamic 1.3 * @smallmargin`
- Undviker behovet av att binda alla dynamiska variabler till argumentlösa funktioner.

Funktionsdefinitioner

Exempel:

```
\ x y .> x * x + y * y
```

- Argumentens namn är en del av funktionens signatur.
- En slask (eng: *sink*) kan ta hand om ytterligare argument.

```
\ x y <> rest .> x + y + (foo [] <>rest)
```

- Vilka argument som helst kan få skönsvärden:

```
\ x:3 y z:2 .> x + y + z
```


Enkla funktionsanrop

Ett enkelt funktionsanrop kan ange argument både genom ordning och genom namn.

```
hypot: \ x y .> [sqrt x*x + y*y]
```

- Ordnade argument: se anrop till sqrt.
- Namngivna argument: [hypot y:3 x:4]
- Blandat: Ordnade argument måste komma först.
- Endast ett argument: square [] 3 eller square [] x:3

Märk att namngivna argument kan inte ändra betydelsen av ordnade argument!

Snitt

Scheme: *evaluated cuts*

```
[hypot 3 ...]
```

```
[hypot y:4 ...]
```

- Ordnade argument blir helt osynliga i den nya funktionen.
- Namngivna argument (er)sätter skönsvärden.
- Endast ett argument: `hypot [...] 3` eller `hypot [...] y:4`

Scenario

Utgångsbudet för att skapa en komplex bild i ett funktionellt språk är att skriva ett stort uttryck som sätter ihop de ingående delarna till en helhet.

- Det skapar lätt en krystad struktur i koden.
- Det stämmer illa med *painter's model* och hur de flesta av oss tänker på att skapa en bild.
- Intuitionen är snarare *imperativ*!
- ...men variabler kunde ju inte bindas om...

Kontrollerade tillstånd

Shapes erbjuder *kontrollerade tillstånd* (eng: *limited states*) för att tillåta en viss grad av imperativ stil.

- Kontrollerade tillstånd binds till variabler som inleds med `•` eller `#`: `•page`
- De kan skickas *by reference* till “funktioner”, men kan inte returneras.
- Några finns globalt, andra kan skapas genom avknoppning från speciella *värden*.

Grundläggande operationer

Det finns tre huvudsakliga operationer på ett tillstånd:

- Lägga till (eng: *tack on*): `●dst << pic`
- Frysa (eng: *freeze*), erhålla slutgiltigt värde, och förstöra (endast egen kod-klammer): `●dst ;`
- Tjuvtitta (eng: *peek*), bör ge samma resultat som att frysa, men förstör inte tillståndet: `(●dst)`

Exempel 1

```
mark: \ •dst .>
{
  •dst << [stroke (0cm,0cm)--(1cm,1cm)]
}

[mark •page]
```

Exempel 2

```
•page <<
{
  •dst: newGroup2D
  •dst << house << car
    << dog
  •dst << cow
  •dst;
}

•page << ( newGroup2D << sun << clouds )
```

Mer om tillstånd

- Inbyggda tillstånds-avknoppare:
`newIgnore`, `newGroup2D`, `newGroup3D`, `newString`,
`newTimer`, `newText`, `newFont`, `newZBuf`, `newZSorter`
- Högre-nivå-konstruktorer:
`newRandom`, `devRandom`
- Globalt definierade (interaktion med omvärlden) tillstånd:
 - `page`, • `catalog`, • `stdout`, • `stderr`, • `randomdevice`,
 - `time`, • `ignore`

“Funktioner” och tillstånd...

- Tillstånd kan skickas både ordnat och per namn, precis som argument.
- En funktion kommer inte åt tillstånd utanför sin kropp.
- Anrop bör ses som makro-expansion snarare än funktions-anrop.
- En kod-klammer med tillstånd blir ett *uttryck*; utifrån syns det inte att tillstånd används för att konstruera klammerns värde.
- Notera att en funktion kan inte frysa tillstånd som den tar emot.
- Tillstånds-parametrar kan inte ges skönsvärden.
- Ett rent funktionsanrop kan inte påverka några tillstånd!

Procedurer

En *procedur* kan påverka tillstånd utanför sin egen kropp.

- Kan vara praktiskt ibland.
- Svårt att analysera; att användas under kontrollerade former.
- Skapas med egen syntax:
`proc: \ arg1 arg2 .> ! body`
- Anropas med egen syntax (annars skulle det se ut som ett rent uttryck!):
`[!proc arg1 arg2]`

Utmaning

Dynamiska tillstånd...

Strukturer

En *struktur* (eng: *structure*) generaliserar en namn-värde-avbildning för att mer likna hur en funktions formella argument binds till värden vid ett funktionsanrop.

- Både ordnade och namngivna fält.
- Kan bara innehålla värden; inga tillstånd i dagsläget.
- Kan vecklas ut vid funktionsanrop och bindning av variabler.
- Används som slask vid funktionsanrop.
- Användas för att “returnera många värden”.

På så sätt uppnås en viss grad av symmetri mellan att funktionsanrop med många värden, och retur av många värden.

Konstruktion och adressering

- Ordnade fält:
s: (> 12 13 14 <)
- Namngivna fält:
s: (> c:14 a:12 <)
- Liksom vid funktionsanrop måste ordnade fält anges före namngivna.
- Adressering av namngivet fält:
s.c
- Hur komma åt ett givet ordnat fält?

Funktionsanrop och slaskar

- För att anropa en funktion (eller procedur) med en struktur:

```
fun [] <> s
```

```
proc [!] <> s
```

- Skapa snitt:

```
fun [...] <> s
```

```
proc [!...] <> s
```

- Funktioner med slask har vi sett tidigare:

```
\ x y <> rest .> x + y + (foo [] <>rest)
```

Binda till delarna

För att binda nya variabler till delarna av en struktur används en särskild och förhållandevis rik syntax. Några exempel:

- `(< first second third >) : (> 1 2 3 <)`
- `(< first second third:40 >) : (> 1 2 <)`
- `(< a:.y b:.x >) : (> x:1 y:2 <)`
- `(< a:.y:8 b:.x >) : (> x:1 <)`
- `(< a:.y:8 b:." >) : (> b:1 <)`

Lat evaluering

Shapes använder lat evaluering vid funktionsanrop och bindning av variabler.

- Imperativ kod fördröjs aldrig (viktigt att lätt känna igen).
- Fördröjd evaluering kan avstyras manuellt:
a : **!!** expr
[fun expr1 **!!**expr2 expr3]
- En funktion kan peka ut vilka argument som ska skickas evaluerade (vanligt i kärnan).

Continuation passing style

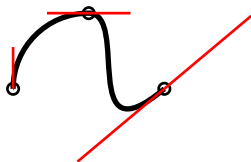
Shapes evalueras *continuation passing style*.

- Svansrekursion blir naturligt.
- Öppnar upp för lekstuga.
- Jobbigt att implementera.
- Inget *dynamic-wind* i dagsläget.
- Begränsad tillgång; endast *escape continuations* i dagsläget.
- ...duger för enklare felhantering.

Funktioner i kärnan

Introduktion

Shapes arbetar med kubiska splines som enda kurv-typ.
Bezier-parameterisering:



Hur vill man ange koordinaterna?

Syntax

- En kurva byggs upp av sammankopplade *kurvpunkter* (eng: *path points*) (och delkurvor).
- En kurvpunkt har en mittpunkt som den interpolerande kurvan passerar genom, och två kontroll-punkter (en framåt och en bakåt).

pp1--pp2--pp3

rear<mid>front

- Koordinaterna *kan* anges absolut:

(0mm,0mm)>(2mm,5mm)

--(4cm,~5mm)<(6mm,0mm)>(9mm,3mm)

--(10mm,0mm)



Relativa och polära koordinater

- Kontroll-punkterna kan anges relativt den mittpunkt de tillhör, och en mittpunkt kan anges relativt föregående mittpunkt på kurvan.
- Relativa koordinater kan anges med ett relativ-uttryck:
Båda koordinaterna tillsammans: $(+(x, y))$
Var och en för sig: $(x, (+y))$
- Kontrollpunkter kan även anges relativt på polär form med speciell syntax:
 (r^a)
- I de polära koordinaterna kan endera eller båda komponenterna utelämnas.

Semantik

Värden för utelämnade komponenter i polära koordinater bestäms i grova drag enligt:

- Vinklar propageras genom mittpunkter (eventuellt hörn).
- Resterande vinklar bestäms baserat på mittpunktens läge i förhållande till angränsande mittpunkter.
- Radier propageras genom mittpunkter.
- Resterande radier beräknas baserat på vinklar.

Alla effekter är lokala, vilket gör processen lättare att hantera.

Smarta enheter

När en radie beräknas baserat på vinklar används en *special-enhet* för längd.

- Avbildar mitt-mitt-avstånd och vinklar på radie.
- 9 special-enheter är definierade, och gör det enkelt att approximera cirkelbågar, undvika inflexioner, göra vågor, med mera.
- Vilken enhet som används bestäms av @specialunit, men det går också bra att använda special-enheter direkt som radie-angivelse.

Travare

En *travare* (eng: *slider*) är ett kurva–kurvtid-par.

- Skapa utifrån kurvlängd, kurvtid, eller andra beräkningar:
[pth 1.3], [pth 7mm], pth.begin + 12mm,
[continuous_approximator pth (4cm,7cm)]
- Punktvisa egenskaper för kurvan (i 3D även binormaler):
sl.p, sl.v, sl.rv, sl.t, sl.rt, sl.n, sl.rn, sl.ik,
sl.rik, sl.time, sl.length, sl.past, sl.looped, sl.mod
- Del-kurvor:
[pth 2cm] -- [pth 2.5cm]

Grundläggande kurv-målning

För att komma igång med ritandet:

- För att måla kurvor:
`[stroke pth], [fill pth], [fillodd pth]`
- Välja färg och dylikt:
`@stroking, @nonstroking, @width, @dash, @cap, @join,`
`@miterlimit, @blend, @nonstrokingalpha,`
`@strokingalpha`
- Utmatning:
 - `page << pic1 << pic2 << pic3`

Transformationer

Transformationer är affina avbildningar som applicerar på grafik och andra geometriska objekt.

- Används som vilken funktion som helst:
`[tf obj]`
- Kan sättas ihop med multiplikationsoperatoren:
`[tf2*tf1 obj] = [tf2 [tf1 obj]]`
- Generella konstruktörer:
`[affinetransform (1,2) (3,4) (5mm,6mm)],`
`[affinetransform3D (1,2,3) (3,4,5) (5,6,7)`
`(8mm,9mm,0mm)]`
- Specialiserade konstruktörer:
`[rotate 25°], [shift (5mm,0mm)],`
`[rotate3D dir:(~1,0,0) angle:45°], [inverse tf]`
`[scale 2.5], [scale y:~1], [scale3D x:2 z:3]`

Grundläggande text-målning

Shapes har stöd för de typsnitt som ingår i PDF-standarderna.

- Manuell och automatisk kernering(?) (eng: *kerning*).
- Tyvärr begränsat urval av tecken som kan kodas. Dock inte sämre än att svenska tecken klarat sig.
- Typsnittsegenskaper sätts som vanligt med dynamiska variabler, fångas i regel upp vid kernering.
- Grafiken skapas genom att text-operationer samlas i ett tillstånd:
(`newText` << op1 << op2)
- En text-operation är oftast en sträng eller en sträng med kernering:
(`newText` << [kern `LINK` 0.15 `ÖPING`])

Pilhuvuden

- Pilhuvuden kan sättas i båda ändar av en kurva som ska målas:
`[stroke pth head:hfn tail:tfn]`
- Utseendet definieras av en funktion som avbildar kurvan på ett pilhuvud och ett avstånd som ska klippas bort från kurvan.
- I *arrowheads.drect* hittar man bland annat `metaPostArrow`.
- Ett pilhuvud med många parametrar specialiseras typiskt innan det används:
`myHead: [metaPostArrow ahAngle:60° ...]`

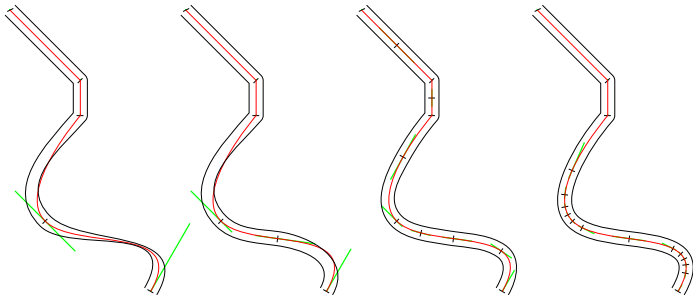
Geometriska beräkningar

- Maximering längs kurva.
- Kortaste avstånd kurva–punkt.
- Skärning med kurva.

Uppsampling och avbildning

Om en kurva ska avildas icke-affint, måste resultatet i regel approximeras.

- Kärnan i Shapes tillhanda håller inga approximationsmetoder.
- Istället uppsampling.
- Se *pathmapping.drext!*



Mer om uppsampling

- Uppsampling kan vara ett lätt sätt att krympa onödigt stora bounding boxes.
- Nersampling är svårt och finns inte i dagsläget.

Genomskinlighet

Att hantera genomskinlighet är inte helt okomplicerat.

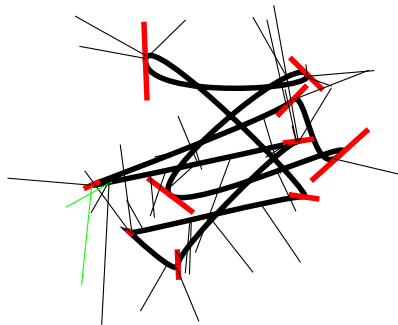
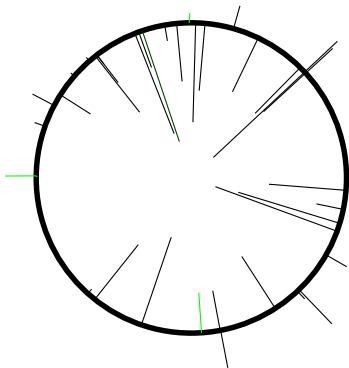


Grundläggande användning

- Grafik i 3D måste *betraktas* (eng: *view*) från ett öga i $(0, 0, @eyez)$ med blicken mot origo, innan den kan kombineras med 2D-världen.
- Grafik i 2D kan bäddas in (eng: *immerse*) i 3D genom att lägga till en z-koordinat med värdet noll.
- Kurvor i 3D (formen, men inte linjebredd) approximeras till önskad precision när de betraktas.
- Streckning (eng: *dash*) hanteras streck för streck.
- (Uppsampling saknas i dagsläget.)

Geometriska beräkningar

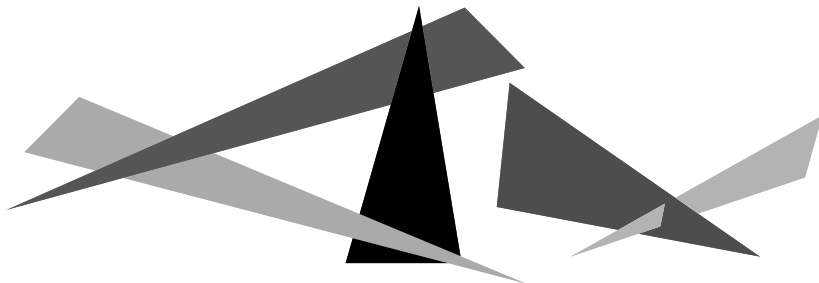
- Maximering längs kurva.
- Kortaste avstånd kurva–punkt



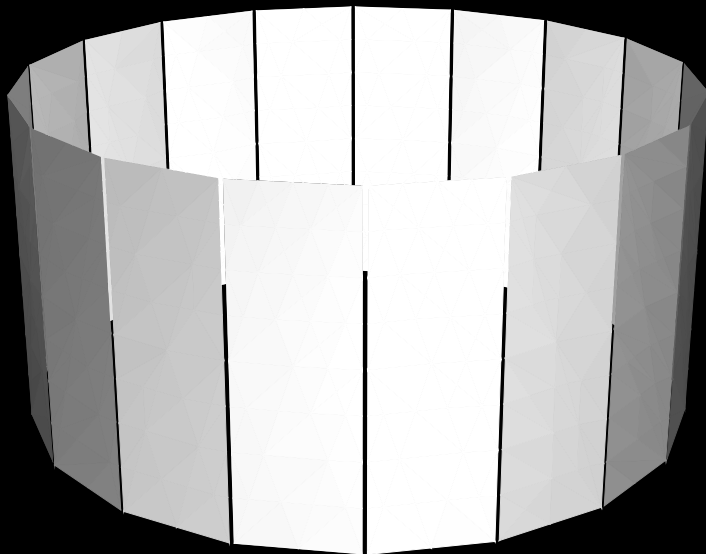
Avancerad användning

- En yta som färgas av ljusmodellen skapas med `facet`, och kan förfinas med `facetnormal`.
- Ytor kan vara enkel- (kan spara beräkningar) eller dubbelsidiga (om man är lat).
- En *tiebreaker* kan användas för att ordna ytor som ligger i varandra.
- Ytor kan ges reflexionsegenskaper:
`@reflexions`, `@autointensity`, `@autoscattering`
- Ljuskällor av olika typer:
`ambient_light`, `specular_light`, `distant_light`
- Dynamiska parametrar styr upplösning och teknik:
`@facetresolution`, `@shadeorder`
- Överlap hanteras med två metoder, `z-buffer` och `z-sortering`.

Exempel: z-buffer



Exempel: ljussättning



PDF

Shapes har stöd för

- Dokument med många sidor. Varje sida läggs till `•catalog`.
- Korsreferenser och dokumentöversikt.
- Sidnummer.
- Text-tillägg (eng: *text annotations*).
- Se *multipage.pdf*!

\LaTeX och strängar

- För att använda pdf \LaTeX :
- Syntaxen för strängar är noga utformad för att inte hamna i konflikt med \LaTeX .
- Ovanliga escape-tecken: \textbackslash och \textasciitilde
- Undviker onödigt många anrop till pdf \LaTeX genom att spara resultat mellan körningar.
- Se *boxedeq.drool!*

Slumptal

Slumptal i ett funktionellt språk?

- Två globala tillstånd utgör inkörsport: `•time` och `•randomdevice`
- Funktionerna `newRandom` och `devRandom` skapar *slump-frön*.
- Ett slump-frö knoppar alltid av sig ett lika dant *slump-tillstånd*, och används normalt sett bara en gång:
 - `rand1: [newRandom (•time)]`
 - `rand2: [devRandom •randomdevice]`
- Slump-tillståndet måste skickas med när man vill ha slumptal: `[random2D •rand1]`
- Funktionellt, om än inte jätte-smidigt!

Utmaningar för framtiden

Trixelering

Att rendera 3D genom en z-buffer kompliceras enormt av att det inte finns ett givet raster.

- Analysera vilka och hur många toleranser som måste sättas.
- Designa algoritm med testbara invarianter.
- Implementera...
- Kan även användas för skuggor!

Kompilera funktioner till PDF

Just nu används färgövergångar endast vid en viss typ av ljussättning.

- Användaren vill kunna göra egna övergångar.
- Funktioner från åskådliga planet till färgrummet ska kunna definieras på hög nivå.
- Funktionerna ska fungera som vanligt i Shapes, men också gå att kompilera till PDF!
- Mycket begränsad PostScript-kalkylator.

Bättre stöd för typsnitt

Snart lessnar man på de inbyggda typsnitten och begränsningar i teckenkodning.

- Reda ut hur man hanterar teckenkodningen systematiskt.
- Förmodligen ta hjälp av FreeType.

Mer grund-struktur

- Namespaces/packages.
- Användar-typer (och -tillstånd!).

Sammanfattning

Sammanfattning

Seminariet har förhoppningsvis förmedlat något om...

- Shapes ursprung och sammanhang idag.
- Språkets grundelement.
- Funktionalitet från kärnan.
- Hur kompilatorn används.
- Hur Shapes-kod kan se ut.

