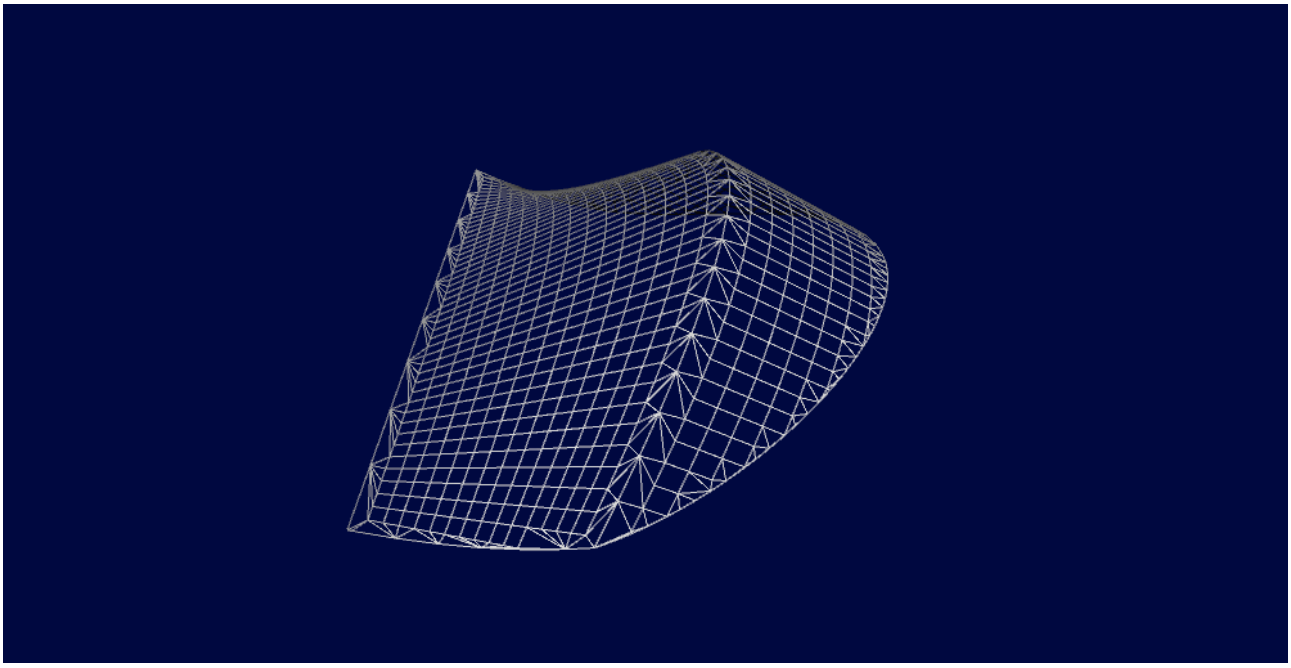
TOGL: Text OpenGL

J. Andreas Bærentzen



TOGL er et lille program, der skal gøre det nemmere at lære computergrafik. Du kan beskrive en simpel scene i en tekstfil og vise den med TOGL. De kommandoer TOGL forstår minder meget om OpenGL, og derfor lærer man en del om OpenGL (og grafikprogrammering generelt) ved at arbejde med TOGL. Dette dokument beskriver TOGL og de grafikkomandoer programmet forstår.

Kom igang

TOGL (eller Text OpenGL) er et lille program, der læser en tekstfil (som vi kalder for en togl fil) og visualiserer den scene, som filen beskriver. En linie i tekstfilen svarer som regel meget præcist til en enkelt OpenGL kommando. Et simpelt eksempel er:

```
ViewTransform
Scale 10 20 10
Box
```

Lad os se hvad de tre linier gør: Du starter det lille program ved at skrive togl i det katalog, hvor programmet ligger sammen med en række tekstfiler. Hvis du loader den fil, der hedder simple.tgl (evt. ved at kalde programmet med filens navn), så kan du se hvad de tre linier herover resulterer i. Det kommer næppe som en overraskelse, at man kan se en kasse på skærmen, der er dobbelt så høj som bred.

Prøv at åbne tekstfilen i en editor samtidigt med at du ser modellen i togl.

Sæt et "#" foran "ViewTransform"

Der forsvandt kassen fordi vi udkommenterede linien. Det er fordi ViewTransform instuerer togl om at transformere kassen ved hjælp af den virtuelle trackball, som er indbygget i programmet. Hvis vi fjerner denne kommando, så ser vi scenen ud fra 0,0,0 og er faktisk inde i kassen. Prøv at tilføje en linie øvers, så filen ser således ud:

```
Translate 0 0 -30
#ViewTransform
Scale 10 20 10
Box
```

Husk at gemme filen. Husk også, at det gør en forskel om du skriver med stort eller småt! Nu kan vi se kassen igen, fordi vi flyttede den 30 enheder ned ad den negative z akse.

Hvorfor TOGL?

OpenGL er et API (applikationsprogrammerings interface) til hardwareaccelereret computergrafik. Sædvanligvis skriver man OpenGL programmer i C eller C++ og i forhold til de muligheder som man har i et programmeringssprog er de tekstfiler togl forstår naturligvis meget primitive. Man kan f.eks. ikke lave et computerspil i togl fordi den kun fortolker OpenGL kommandoer og ikke har faciliteter til generel programmering. Hvorfor så ikke bare skrive programmer i C++, java, Python eller et andet højniveausprog med OpenGL bindings?

Det er absolut også en mulighed, men TOGL har nogle fordele:

- ingen compile-run cycle: Du ændrer togl filen, gemmer den og ser umiddelbart resultatet. Det er fordi togl læser filen hver eneste gang den tegner scenen. Selvom det ikke er effektivt giver det umiddelbar feedback. Det er langt nemmere at eksperimentere på denne måde end det er hvis man skal compilere programmet først.

- Enklere syntax: OpenGL funktioner findes i mange varianter, og der er mange af dem. togl forstår kun et meget begrænset subset af OpenGL og syntaxen er noget enklere. Det betyder, at det er hurtigere og nemmere at lære at skrive togl tekstfiler end at bruge OpenGL.
- Interaktiv viewer: togl programmet er jo en interaktiv viewer, hvor du kan vælge mellem filer, rotere modellen, skifte projektion etc. Der er med andre ord en masse funktionalitet som man selv skulle kode (eller få udleveret) hvis vi brugte et programmeringssprog.

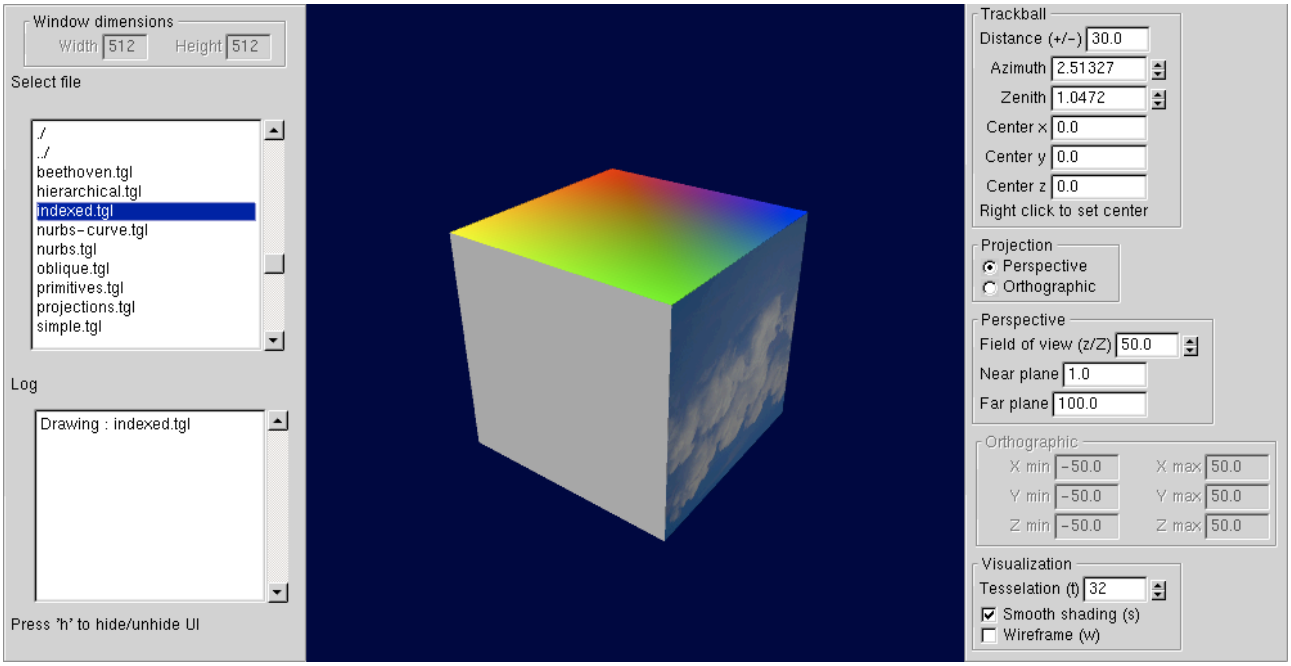
Med andre ord gør togl det meget nemmere at lære de basale principper i computergrafik. Når disse er forstået skal man naturligvis programmere OpenGL programmer i et rigtigt programmeringssprog, men forhåbentligt er principperne nemmere at forstå med togl.

TOGL Programmet

Herunder er et screenshot af selve togl programmet. I midten vises den scene du har indlæst. Du kan vende og dreje objektet ved hjælp af musen blot ved at venstreklikke og trække. +/- styrer afstanden, og ved at højreklikke kan du centrere den virtuelle trackball på et punkt på objektet.

Ude til venstre er en liste over filer, der kan loades ganske enkelt ved at man klikker på dem. Nedenunder er et log vindue, der udskriver fejlmeddelelser, hvis filen indeholder fejl.

Vinduet til højre styrer trackballen, projektion og diverse parametre. Bemærk at de fleste ting både kan styres via den grafiske brugerflade og samtidig har hotkeys som står i parantes efter navnet på parameteren i den grafiske brugerflade. Du kan skjule brugerfladen ved at trykke på h.



TOGL sproget

TOGL minder meget om VRML, X3D eller andre sprog til at beskrive 3D scener. Forskellen er måske at det er mere beslægtet med OpenGL og ikke specielt skalerbart. Til gengæld er det forhåbentligt lidt mere overskueligt. Den bedste måde at forstå det på er nok at kigge på eksemplerne. Herunder er en liste over de centrale features med henvisning til

eksempel filerne. Det bedste er nok ret hurtigt at læse det følgende igennem og så eksperimentere.

Viewing
simple.tgl er et godt udgangspunkt. Det er en fil med de tre linier som blev gengivet først i dette dokument. Den første kommando

```
ViewTransform
```

kalder simpelthen trackball koden - d.v.s. den sørger for at scenen vises sådan som brugeren har angivet ved interaktivt at trække med musen. Man behøver ikke at bruge `ViewTransform` kommandoen. Man kan også benytte transformationer

Transformationer
Der er tre transformationskommandoer som svarer præcist til dem man finder i OpenGL:

```
Rotate 30 1 0 0
Translate 2 2 2
Scale 10 20 10
```

I eksemplet med `Rotate` er det første argument vinklen i grader. De andre tre angiver aksen, der roteres om.

Der er også varianter som giver mulighed for simpel animation

```
RotateIncremental 10 1 0 0 36
TranslateIncremental 4 0 0 20
ScaleIncremental 1.02 1.02 1.02 36
```

De virker som de almindelige transformationer, men det sidste argument er tiden (i sekunder) som transformationen skal forløbe over. Man ganger så henholdsvis vinklen eller translationsvektoren med tiden ved rotation eller translation. Ved skalering opløftes skaleringsfaktoren i tiden.

Tranformationer ses beskrevet i hierarchical.tgl

Mere om transformationer
I stedet for at specificere om man vil rotere, skalere o.s.v. er det også muligt at angive en generel 4x4 transformationsmatrix. Det sker med kommandoen

```
Transform
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

Kommandoen `Transform` tager 16 tal som argument, former en 4x4 matrix som ganges på det der i OpenGL kaldes modelview matricen. For at øge læsebarheden har jeg skrevet de 16 tal i fire rækker herover, men det er sådan set lige meget.

En anden kommando er

```
LookAt 10 10 10  0 0 0  0 1 0
```

der bruges til at placere kameraet i forhold til scenen. De første tre tal er punktet man kigger fra, de næste tre tal er punktet man kigger på, og de sidste tre tal er en vektor i retningen opad. Med andre ord er kameraet i eksemplet placeret i (10, 10, 10) og det kigger mod (0,0,0) og op retningen er (0,1,0).

`LookAt` slår den virtuelle trackball fra. Derfor er det som regel smartere at skrive `ViewTransform`. Den virker ligesom LookAt, men det er musen der angiver hvor øjepunktet er. `LookAt` sammen med `ViewTransform` giver ingen mening! Brug kun LookAt hvis I vil specificere et præcist øjepunkt og synsretning - f.eks. for at tegne et X perspektiv eller trepunktsperspektiv etc.

Kig i transforms.tgl eller oblique.tgl for at se eksempler på brug af `LookAt` og `Transform`

Transformationer til projektion
Ligesom `LookAt` sjældent skal bruges, så skal I også sjældent selv specificere en projektionsmatrix. Via brugerfladen kan man jo skifte mellem ortografisk projektion og perspektiv. En sjælden gang kan det dog være nødvendigt for at løse en opgave, og der er faktisk hele tre metoder til at specificere en projektionsmatrix direkte i en TOGL fil. Den første funktion angiver en helt generel projektionsmatrix (her identitetsmatricen) som projektionsmatrix.

```
Projection 1 0 0 0  0 1 0 0  0 0 1 0  0 0 0 1
```

Denne funktion er kun nødvendig til meget specielle ting som f.eks. skråprojektioner. Et eksempel på netop dette finder du i oblique.tgl

Ortografisk projektion specificeres med

```
Ortho -10 10 -10 10 -10 10
```

hvor man angiver det volumen som skal parallelprojiceres. I dette tilfælde er volumenet en box med sidelængde 20 der er centreret omkring origo. Perspective

```
Perspective 50 1 1 1000
```

tager field of view (en vinkel i grad) i y retningen (her 50 grader), aspekt ratio, og afstanden til nærmeste og fjerneste clipping plane som argumenter. Aspekt ratio er vinduets bredde divideret med højden. I eksemplet ovenover antager vi, at vinduets bredde og højde er den samme. Du kan se vinduets bredde og højde øverst til venstre i brugerfladen. transforms.tgl viser eksempler på de to funktioner.

Der er en meget væsentlig forskel mellem de projektionstransformationer som er beskrevet i dette afsnut og så `Transform` eller `LookAt` eller de andre transformationer. Når man angiver en projektion med `Projection`, `Ortho` eller `Perspective` så **overskriver** man den gældende projektionsmatrix. Med `Rotate`, `Translate`, `Scale`, `Transform`, `LookAt` eller `ViewTransform` angiver man også en matrix, men den ganges på den nuværende transformationsmatrix (modelview matrix i OpenGL terminologi).

Farve, belysning og materialer
Det er muligt at sætte nogle parametre for materialerne med

```
AmbientColor 0 0 0.3
DiffuseColor 0.9 0.5 0
SpecularColor 0 1 0
Shininess 10
```

Her er argumenterne til de første tre kommandoer naturligvis rød, grøn og blå. Til den sidste er det Phong eksponenten. Det er også muligt at sætte lysets position med

```
LightPosition 0 0 1 0
```

Det er en position i homogene koordinater og det sidste argument er derfor w. Bemærk, at hvis w er 0 så er lyskilden "uendeligt" langt væk. I dette tilfælde er det altså en retningslyskilde og ikke en punktlyskilde.

Se igen hierarchical.tgl.

Enable og Disable
Det er muligt at Enable og Disable forskellige effekter. Belysning er slået til fra start. Det kan slås fra med

```
Disable Lighting
```

og til igen med

```
Enable Lighting
```

Det er også muligt at slå teksturering til og fra med

```
Enable Texture
Disable Texture
```

og dybde-test

```
Enable DepthTest
Disable DepthTest
```

Hierarkisk modellering
Der er nogle krøllede paranteser i mange af eksemplerne - specielt i hierarchical.tgl.

```
{
```

betyder at transformationsmatricen og belysningsparametre skal gemmes på en stack. Tilsvarende betyder

```
}
```

at man tager transformationsmatricen og belysningsparametre fra stacken. Det bevirker at alt hvad der står mellem { og } bliver "glemt" når man passerer }. Så du kan lave en

kompleks transformation mellem to paranteser uden at den påvirker senere kommandoer. Det samme gælder ændringer af belysningsparametrene (herunder materiale og lys parametre).

Se igen hierarchical.tgl

Platoniske (og ikke så Platoniske) solider
Det er muligt at tegne en lang række simple former med tilsvarende simple kommandoer:

```
Sphere
Box
Cone
Torus
Tetrahedron
Dodecahedron
Icosahedron
Octahedron
Teapot
```

Spørg ikke om hvorfor tekanden er der. Det ville føre for vidt at forklare. De tager ingen parametre, så man kan kun tegne dem på een måde, men tesseleringsniveauet i brugerfladen påvirker hvor fint opdelte nogle af disse objekter bliver tegnet, og desuden kan de jo transformeres som beskrevet med Rotate, Scale og Translate.

De fleste solider er vist i hierarchical.tgl

Geometriske primitver
Hermed forstår vi helt simple ting som punkter, trekanter o.s.v. De tegnes ved det man kalder for Begin End paradigmet. Et eksempel er nok på sin plads:

```
Begin Triangles
Normal 1 0 0
Vertex 0 0 0
Vertex 0 1 0
Vertex 0 0 1
End
```

Det ovenstående tegner en trekant med hjørner i 0,0,0; 0,1,0 og 0,0,1 og normalen 1,0,0. `Normal` angiver altså normalen og `Vertex` angiver et hjørnepunkt. Hvis vi havde skrevet `Points` i stedet for `Triangles` var resultatet blevet tre punkter.

Princippet er altså at man mellem `Begin` og `End` angiver en række punkter (vertices) og de forbindes så til primitiver af den type der er angivet som argument til `Begin`. I det ovenstående tilfælde er det trekanter, men der er også:

```
Begin Points
Begin Polygon
Begin Quads
Begin TriangleStrip
Begin Lines
```

`Polygon` er lidt anderledes, for der skabes kun en polygon af et `Begin Polygon ... End` par. `TriangleStrip` er også lidt anderledes fordi der her produceres en trekant for hvert eneste vertex man angiver. Den deler så en kant med den foregående trekant.

Bemærk også at der udover `Vertex` og `Normal` er mulighed for at skrive `Color`. `Color` har dog kun indflydelse hvis man Disabler belysning. For både `Color` og `Normal` gælder det, at farven eller normalen skal komme før det tilsvarende kald til `Vertex`! OpenGL er en tilstandsmaskine, og `Color` eller `Normal` gælder fra kaldet og til næste gang du kalder. Men `Vertex` er speciel: Når du kalder `Vertex` produceres f.eks. et hjørne i en trekant, og til dette hjørne knyttes al den tilstand der gælder præcis når du kalder `Vertex`. F.eks. normal og farve, og derfor skal du kalde `Normal` og `Color` inden du kalder `Vertex`.

Der er eksempler i primitives.tgl

Indexerede Primitiver
Det er muligt at lave arrays af vertices, normaler, farver, og teksturkoordinater. Et vertex array ser ud som følger:

```
VertexArray 8 :
0 0 0
1 0 0
1 1 0
0 1 0
0 0 1
1 0 1
1 1 1
0 1 1
```

Her angiver vi at der følger et `VertexArray` med otte punkter. Det er nødvendigt med et kolon efter antallet af vertices. Du kan måske regne ud at det ovenstående array er hjørnerne i en kube. Man kan nu tegne en side af kuben med:

```
Begin Quads
Element 0
Element 1
Element 2
Element 3
End
```

Det ovenstående tegner en enkelt firkant. I stedet for `Vertex` står der `Element`. Tallet efter `Element` henviser til en entry i det array vi specificerede før. Det virker umiddelbart mere omstændeligt end direkte at skrive `Vertex`, men husk at hvert hjørne i en kube deles af tre sider. Vi kan altså bruge det samme hjørne mange gange med indexerede primitiver. End anden fordel er at grafikkortet kan cache vertices, og det kræver naturligvis at de er indekserede. Udover `VertexArray` har vi

```
NormalArray
ColorArray
TexCoordArray
```

De fungerer på helt samme måde. De behøver ikke at have samme størrelse, men man må naturligvis ikke indeksere ud over kanten.

Der er et eksempel i indexed.tgl

Teksturer
Den nedenstående stump tegner en firkant med tekstur.

```
Enable Texture
TextureMode Modulate
TextureImage skyemboss.png
Normal 1 0 0
Begin Quads
TextureCoord 0 0
Vertex 0 0 0
TextureCoord 1 0
Vertex 0 1 0
TextureCoord 1 1
Vertex 0 1 1
TextureCoord 0 1
Vertex 0 0 1
End
```

Bemærk at man skal `Enable` teksturen.

```
TextureMode Modulate
```

Betyder at teksturen ganges på den farve som shadingen producerer. `TextureImage` bestemmer hvilket billede der indlæses. Endeligt skal der vælges teksturkoordinater. Det er muligt at knytte teksturkoordinater til hvert `Vertex` med

```
TextureCoord 0 1
```

TextureCoord's er altså 2D - det er jo positioner i et billede.

Eksemplet er i texture.tgl

DisplayLists
Hvis nu du har store togl filer, så er det lidt ærgerligt, at de indlæses hver gang scenen tegnes. En løsning på problemet er display lists. En display list er en meget simpel mekanisme, hvor du kan indspille nogle kommandoer i en slags makro som bagefter kan afspilles. Faktisk gemmes denne makro i mange tilfælde helt ude på grafikkortet, så det er en meget effektiv mekanisme. Et eksempel ses herunder:

```
BeginList ball
Scale 2 2 2
Sphere
EndList

Translate 0 9 -5
CallList ball
```

Intet af det der sker mellem `BeginList` og `EndList` udføres umiddelbart, men når du senere kalder `CallList`, så tegnes kuglen. Ordet "ball" er valgt af brugeren, og det eneste krav er at argumentet til `CallList` svarer til argumentet til `BeginList`

Se ellers beethoven.tgl

NURBS Flader
Det er rart at have mere generelle og avancerede primitiver end bare polygoner. NURBS er et meget kraftigt redskab til at modellere overflader. Herunder ses et eksempel.

```
Begin NURBS
UOrder 3
VOrder 4
UKnots 8 : 0.1 0.2 0.3 0.4 0.5 2.0 2.0 2
VKnots 8 : 0 0 0 0 1 1 1 1
ControlPoints 20 :
 0 0 0 1  0 1 0 1  0 2 0 1  0 3 0 1
 1 0 0 1  1 1 2 1  1 2 2 1  1 3 0 1
 2 0 0 1  2 1 2 1  2 2 2 1  2 3 0 1
 3 0 0 1  3 1 0 1  3 2 0 1  3 3 0 1
 4 0 0 1  4 1 -2 1  4 2 3 1  4 3 0 1
Trim 1 0.5 0.7
End
```

En NURBS flade har to parametre som vi her kalder u og v. Det er nødvendigt at specificere ordenen og knuder for hver parameter samt at specificere et gitter af kontrolpunkter. UKnots, VKnots og ControlPoints er angivet på samme måde som VertexArrays - der er antallet af elementer et : og så elementerne.

Bemærk at der er fire kontrolpunkter vandret og fem lodret. Søjlerne svarer altså til u retningen og rækkerne til v retningen. Husk at antallet af knuder skal være lig orden plus antallet af kontrolpunkter.

```
Trim 1 0.5 0.7
```

betyder at vi trimmer med en cirkel i parameterdomænet der er placeret i 1,0.5 med radius 0.7. Hvis man udelader `Trim` bliver fladen ikke trimmet, men det er ikke muligt at vælge en anden form til trimming end cirklen. Dog kan man angive en negativ radius og det betyder at et hul bliver skåret (det der er inden for kurven fjernes).

Se nurbs.tgl

NURBS kurver
NURBS kurver defineres næsten ligesom flader. Der er kun en u parameter og derfor også kun en familie af knuder. Herunder ses et eksempel som også er i nurbs-curve.tgl

```
Begin NURBSCurve
UOrder 3
UKnots 8 : 0 0 0 .25 .75 1.0 1.0 1.0
ControlPoints 5 :
```

```
 0 0 0 1   1 1 0 1   2 2 0 1   3 3 0 1   4 1 0 1
End
```