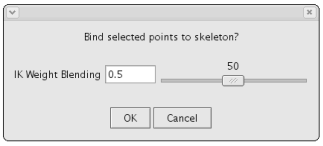
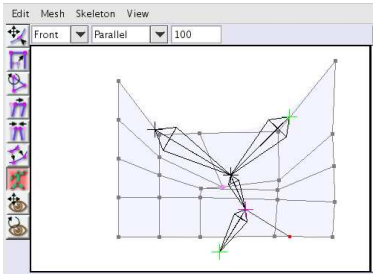
Kinematik i Art of Illusion 3(3)

Vi kan rotera det vänstra benet med handtaget (MK1) i yttersta leden. Ett ben rör sig alltid runt sin moderled. Nu skall vi koppla ihop skelettet med nätet, vilket görs med **Skeleton/ Bind Points to Skeleton**.



Vi funderar inte närmre på det föreslagna värdet utan trycker bara på **OK**.
Nu kan vi röra på ben på det vis som beskrivits och nätet påverkas (i just detta exempel dock inte så tydligt som vore önskvärt). Vi kan markera förgreningsleden (med MK1) och sedan röra hela övre delen av skelettet med den ledens handtag. Detta är **framåtkinematik**.
Vi kan också begära att visa leder skall hållas fixerade. Detta görs med SHIFT-MK1, som gör krysset grönt. I figuren nedan är roten och övre högra leden grönmarkerade (fixerade). Därefter kan vi röra skelettet genom att med MK1 trycka på en led (inte på handtaget) och dra den.
I figuren nedan har jag rört på förgreningsleden. Detta är **invers kinematik**.



Film från en bildföljd

Vi har sett att en del program producerar animeringen i form av en bildföljd. Denna måste i så fall göras om till en film av ett fristående program. Det finns många sådana. Utformatet kan vara bl a MPEG, AVI (ev med DivX) eller Microsofts WMV.
Själv använder jag ett uråldrigt UNIX-program *mpeg_encode* (åtkomligt direkt under Linux om man gjort `setup_course TDA360`), som tillverkar en MPEG-film utifrån bl a TIFF-, PNG- eller XWD-bilder. Bränns den sedan på en CD eller DVD kan filmen betittas i en TV kopplad till en DVD-läsare eller spelas upp med *gmplayer*/*mplayer* (se separat). Det programmet används så här:

```
mpeg_encode P.par
```

där `P.par` är en parameterfil som innehåller information om indata och utdata m m. Det är inte alldeles självklart vilka värden man skall ange för de olika parametrarna. Jag kan lägga upp en mall om någon är intresserad. Ett annat program *ppmtompeg* fungerar på ett likartat sätt.
Rasmus Anthin tipsade om ett fritt PC-program *Virtual Dub*, som kan användas för detta ändamål. Det finns många kommersiella.

Uppspelning av animeringar

Programmet *mplayer*/*gmplayer* finns installerat under Linux (dock inte via menyerna). Man kan spela t ex en enstaka .mpeg eller .vob-fil (ej kopieringsskyddade) enligt modellen

```
mplayer dvd://N VTS_01_1.VOB
```

(`dvd://N` enbart om filen på DVD-skiva) eller starta

```
gmplayer
```

som ger GUI. Tryck på MK3 och välj i menyn, varefter man får ett för oss alla numera välkänd typ av panel.

Programmet klarar en massa andra format, t ex AVI och MOV (Quicktime). Programmet finns även i vår Windows-miljö om man letar bland kursmapparna.
I mappen `$DG/BILDER` finns en del animationer.

Animering från OpenGL

Om vi vill göra en film från OpenGL kan vi låta uppdateringsmetoden producera en bild på t ex PNG-format vid varje anrop och sedan bilda en MPEG-fil som ovan antytts. Ett krux är möjligen att hitta kod som tillverkar bilden. Ett värre krux är förmodligen att bilderna inte tillverkas i helt jämn takt och därför borde tidsmärkas.

Ett enkelt sätt att få fram bilderna under Linux är att använda xwd-kommandot inifrån programmet. Detta X-kommando tar en s k fönsterdump och lagrar den på XWD-format. Man kan i parameterfilen till *mpeg_encode*/*ppmtompeg* tala om att .xwd-filerna skall omvandlas till .ppm-filer som dessa program utgår ifrån. Som vanligt kommando skriver man:

```
xwd -name fönsternamn -out bildfilnamn
```

I ett OpenGL-program gör man så här:

1. Inför två globala variabler

```
static int antal = 0;
static char command_string[] = "/usr/X11R6/bin/xwd -name fffffffff
-out nnnn";
```

2. I uppdateringsmetoden (klarar bilder 01 till 99)

```
antal = antal + 1;
if (antal < 10 ) {
      sprintf(command_string,"/usr/X11R6/bin/xwd -name MyWind
                              -out   bb0%d",antal);
} else {
     sprintf(command_string,"/usr/X11R6/bin/xwd -name MyWind
                             -out bb%2d",antal);
}
system(command_string);
```

Med denna kod får vi bilder *bb01* till *bb99*.

Animering i MATLAB

Borde man säga något om men utrymmet är knapert, så det får räcka att man kan dels skapa interna animationer, dels göra om en sådan till AVI-format (Audio/Video Interleaved).

Animering i Blender och PovRay

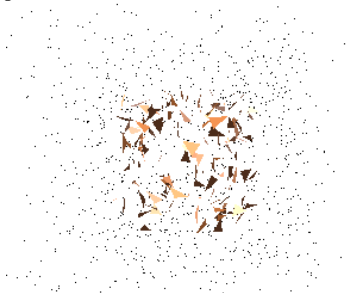
Går utmärkt. Blender ungefär som AoI.

Partikelsystem

Ett partikelsystem utgörs av ett antal partiklar och regler för dessas rörelser och andra beteenden. Partiklarna kan i enklaste fallet vara punkter, men också andra mindre objekt är tänkbara. Det kan t ex finnas föreskrifter om en viss hastighet eller acceleration. Objekt kan eventuellt skapas och dö bort successivt. Eventuellt pågår fenomenet bara så länge "bränslet" (i enklaste fall en varvräknare) räcker. Vill man att det skall gå långsammare kan man låta förloppet styras av fysikaliska lagar.

Naturliga företeelser som kan modelleras med partikelsystem är fyrverkerier och andra explosioner. Men även t ex eld, rök och damm.

Exempel: Gustav Taxéns program `DEMOS/EXPLOSION.c`. När man trycker på mellanslagstangenten sker en explosion som yttrar sig i att dels ett tusental punkter far iväg radiellt från origo i en slumpmässig riktning och med en slumpmässig hastighet, dels att ett antal slumpmässigt skalade trianglar gör likaledes samtidigt som de snurrar. Bilden gör inte programmet rättvisa.



I programmet är trianglarna belysta medan punkterna bara ritas med sin färg. Programmet är uppbyggt som ett typiskt sådant program.

1. Skapa objekten.
2. Låt en idle-procedur räkna ut nya positioner etc och anropa sedan via `glutPostRedisplay` uppdateringsproceduren, som ritar den nya situationen. I just detta fall beräknas nya position genom att man addererar partikelns hastighet.

Exempel: Skärmlåsningsprogrammet *xlock* med `xlockMB -mode pyro`

Modifierat utdrag ur kod för EXPLOSION.c

```
/* Uppdateringsproceduren som ritar om scenen */
void display (void) {
  int    i;
  // ... Sudda och placera kamera
  if (fuel > 0)    {
    glPushMatrix ();
    glDisable (GL_LIGHTING); glDisable (GL_DEPTH_TEST);
    glBegin (GL_POINTS);      // PUNKTERNA
         for (i = 0; i < NUM_PARTICLES; i++) {
            glColor3fv (particles[i].color);
            glVertex3fv (particles[i].position);
         }
    glEnd ();
    glPopMatrix ();
    glEnable (GL_LIGHTING); glEnable (GL_LIGHT0);
    glEnable (GL_DEPTH_TEST);
    glNormal3f (0.0, 0.0, 1.0);
    for (i = 0; i < NUM_DEBRIS; i++) {  // TRIANGLARNA
      glPushMatrix ();
      glTranslatef (debris[i].position[0],
                    debris[i].position[1],
                    debris[i].position[2]);
      glRotatef(debris[i].orientation[0],1.0,0,0);
      glRotatef(debris[i].orientation[1],0.0,1,0);
      glRotatef(debris[i].orientation[2],0.0,0,1);
      glScalef (debris[i].scale[0], debris[i].scale[1],
                debris[i].scale[2]);
      glBegin (GL_TRIANGLES);
          glVertex3f (0.0, 0.5, 0.0);
          glVertex3f (-0.25, 0.0, 0.0);
          glVertex3f (0.25, 0.0, 0.0);
      glEnd ();
      glPopMatrix ();
    }
  }
  glutSwapBuffers ();
}
```


Flockar, boids

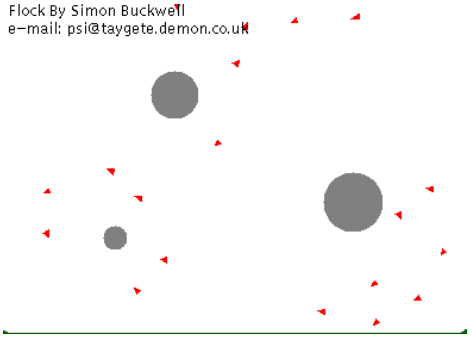
I en del situationer har man ett flockbeteende. T ex fåglars flykt och fiskstim. För detta finns det engelska begreppet boids (av bird-oid), som infördes av Craig Reynolds 1987. Ingår i något man kallar **Artificial Life**.

Han upptäckte att man kan få naturlig flockning med hjälp av tre enkla regler:

1. Separation: Se till att inte komma för alltför nära andra boids eller hinder i scenen.
2. Gemensamt mål (eng alignment): Eftersträva samma hastighet och färdriktning som de boids som är nära.
3. Grupp (eng cohesion): Styr mot masscentrum för de boids som finns i närheten.

Varje boid beter sig individuellt, dvs det finns ingen flockregel. Med N boids blir problemet av komplexiteten $O(N^2)$, men med lagom approximationer kan man få flockar att röra sig i realtid. Man kan fortfarande tala om ett partikelsystem men med tilläggsregler.

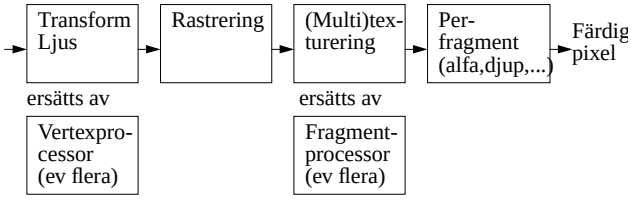
Här visas en (något tryckmodifierad) av många appletar. http://www.taygete.demon.co.uk/java/flock/ (ur funktion 2003/4)



Referenser med många länkar: `http://www.red3d.com/cwr/boids/` och `http://www.red3d.com/cwr/boids/applet/`

Vertex- och fragmentprogrammering, allmänt 1(2)

är begrepp som stöds av NVIDIA, ATI och Microsoft (DirectX) och nu även OpenGL i något olika varianter. Vi belyser det en aning, dels för nyhetsvärdets skull, dels för att vi kan få ökad förståelse. För detaljer hänvisas till teknisk dokumentation. Engelska benämningar vertex programming eller vertex shading, respektive pixel programming, pixel shading eller fragment programming/fragment shading. Beskrivningen görs enbart från ett OpenGL-perspektiv. Följande bild visar OpenGL:s rörledning kraftigt förenklad. Den matas från vänster med bl a hörnkoordinater (i modellkoordinatsystemet). Dessa transformeras i första steget till hörn i homogena projektionskoordinater med bl a tillhörande uträknade ljusvärden. Senare kommer textureringssteget, som vi vet kan modifieras en hel del.



Dessa två steg är i nyare grafikprocessorer programmerbara. Användaren har hittills förväntats skriva program för dem i ett maskinkodsspråk. Nivån höjs en hel del med C-liknande språk som Microsofts HLSL (High Level Shading Language) och NVIDIAS Cg, och OpenGL Shading Language (GLSL). För att öka prestanda har en del grafikprocessorer flera vertex- och fragmentprocessorer som arbetar parallellt. ARB införde först begreppen vertexprogram och frag-

Vertex- och fragmentprogrammering, allmänt 2(2)

mentprogram (utvidgning 2002) som skrivs i ett språk ligger mellan maskinspråken och de nämnda högnivåspråken. Senare - 2003 - kom ARB med utvidgningarna “vertex shader” och “fragment shader”, vilka skrivs i väsentligen GLSL. Hösten 2004 fördes dessa in i OpenGL 2.0 med några mindre namnförändringar. Fr o m hösten 2005 kan vi här använda de riktiga 2.0-namnen. Jag har svårt att hitta någon bra översättning av “shader” och kommer därför att använda benämningarna vertexprogram och fragmentprogram i stället för de korrekta.

Som namnen antyder arbetar ett vertexprogram på ett hörn (vertex), medan ett fragmentprogram arbetar på de fragment (pixlar) som kommer ut från rastreringsetappen. För **varje** hörn respektive fragment kommer motsvarande program att genomgås.

Fragmentprogrammering gör att t ex Phong-toning (se belysningsavsnittet) kan åstadkommas. Varje pixel kan påverkas genom kombination av en mängd texturer. Man har använt fragmentprogram för avancerade ljusberäkningar, procedurtexturer, fraktalbilder, etc.

Innan de två stegen blev programmerbara utvidgades de med ett stort antal speciallösningar för diverse problem, vilket ledde till oöverblickbarhet. Förmodligen var stegen internt “programmerbara” redan då. Nu blir det tydligare för grafikprogrammeraren vad som är möjligt.

Det som sägs här bygger på bl a specifikationstexterna för OpenGL 2.0 (c:a 400 sid) och The OpenGL Shading Language (c:a 100 sid). I den senare beskrivs språket för vertex- och fragmentprogram, medan kopplingen mellan OpenGL och dessa program behandlas i specifikationen för OpenGL 2.0. Länkar till mer dokumentation på kurssidan.

OpenGL Shading Language (GLSL) 1(3)

är det språk i vilket vi skriver våra vertex- och fragmentprogram.

Språket är avsiktligt C-liknande med ett antal nya datatyper, bl a för beskrivning av vektorer. Vi kan bygga upp programmen modulärt genom att skriva egna funktioner. Programmen startar alltid sin exekvering med funktionen *main*. I funktionerna kan vi införa egna lokala variabler. Det finns ett stort antal fördefinierade namn, dels funktioner, dels variabler som är globala i programmen (vissa av dessa är bara läsbara eller bara skrivbara). Dessutom kan vi införa egna globala variabler.

GLSL innehåller de normala styrkonstruktionerna som if, for och while, return samt discard (avbryter exekveringen av ett program och allt blir ogjort). Hårdvaran/drivrutinerna för våra 6600-processorer har stöd för hela GLSL (utom noise-funktionerna). I praktiken finns en del andra restriktioner (begränsad programlängd, begränsat variabelutrymme), som gör att en del programidéer stöter på patrull. Men utvecklingen går vidare.

Ett program utgörs initialt av en sträng som sedan kompileras till en intern maskinkod. Vi kan skriva strängen direkt i vårt vanliga OpenGL-program, men vanligen läser man in vertexprogrammet från en fil och fragmentprogrammet från en annan (härigenom kan man få sitt program att bete sig annorlunda genom att bara ändra i filen). I frånvaro av endera typen av program utförs de normala stegen. T ex ett vertexprogram kan delas upp på flera filer, men det är ju något som rimligen bara är intressant för riktigt stora program, så det går vi inte in på.

OpenGL Shading Language (GLSL) 2(3)

Exempel på datatyper: `float`, `int`, `vec3`, `vec4`, `mat3`, `mat4`, ...

Vid **initieringar** används en **konstruktor**, t ex
```
vec3 color = vec3(1.0, 1.0, 0.0);
```

För vektorer kan utöver den vanliga notationen med [i] **.xyzw-notation** (rumskoordinater) användas liksom .rgba- (färger) och .stpq-notation (texturer). T ex ändrar
`color.rg = vec2(1.0, 0.0)` eller `color.g = 0.0` eller `color[1]=0.0` den tidigare gula färgen till röd.

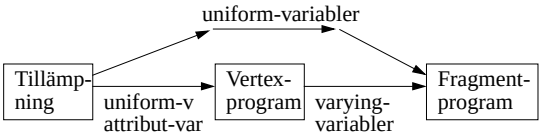
I C används **kvalifikatorn** `const` vid sidan av datatypen för att definiera en konstant. Samma gäller GLSL, t ex
```
const vec3 redcolor = vec3(1.0, 0.0, 0.0);
```
Dessutom finns ytterligare **kvalifikatorer** `uniform`, `attribute` och `varying`, vars uppgift framgår av nästa OH.

De olika matematiska operationerna skrivs som i C, med den skillnaden att de kan verka elementvis på vektorer. T ex
`color1+color2`, `sin(color)`.
För skalärprodukt och vektoriell produkt finns funktionerna `dot` respektive `cross`. Ytterligare en mängd funktioner finns. Ett urval: `cos`, `tan`, `pow`, `exp2`, `log2`, `abs`, `inversesqrt` (ger 1/kvadratroten(...)), `length`, `min`, `max`, `normalize`, `reflect` (ger reflektionsvektorn givet ljusvektorn och normalen i vykoordinater), `noise1/2/3/4`. Vissa är i praktiken approximativa. Nvidia har ännu inte lyckats implementera `noise`-funktionerna.

För matriser `m` avser `m[i]` den i:te kolumnen!!!

OpenGL Shading Language (GLSL) 3(3)

Vår tillämpning, vertexprogrammet och fragmentprogrammet kan kommunicera. Hur visas schematiskt i följande bild.

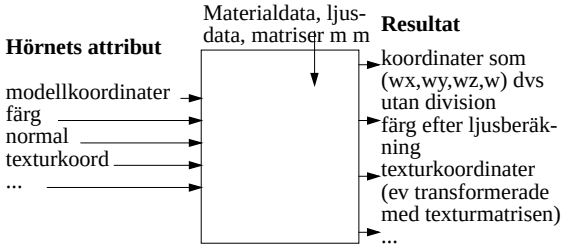


Vertexprogrammet		Fragmentprogrammet	
In	**Ut**	**In**	**Ut**
gl_Vertex	gl_Position	gl_Color	gl_FragColor
gl_ModelViewMatrix	gl_TexCoord[i]	gl_SecondaryColor	
gl_ModelViewProjectionMatrix	gl_FrontColor	gl_TexCoord[i]	
gl_LightSource[i]	gl_BackColor	gl_FrontMaterial	
gl_MultiTexCoord0-7		gl_BackMaterial	
gl_Normal		gl_LightSource[i]	
gl_NormalMatrix		...	
gl_Color			

En uniform-variabel deklareras i vertex/fragmentprogrammet och ges värde i tillämpningsprogrammet. Den är bara läsbar för övrigt. Attribut-variabler används på ett likartat sätt men är tänkta för situationer där värdet ändras från vertex till vertex. En varying-variabel deklareras i vertex- och fragmentprogrammet. Den är skriv- och läsbar i vertexprogrammet men enbart läsbar i fragmentprogrammet. Värdet i fragmentprogrammet interpoleras fram. Det finns fördefinerade variabler som följer de tillstånd (attribut) vi sätter med t ex anropet *glVertex3d*. Några av dessa nämns i kolumnerna **In** i bilden. I kolumnerna **Ut** anges några variabler som måste (bör) ges värden. De används i stegen efter vertex- respektive fragmentprogrammet.

Vertexprogrammering 1(9)

Normalt transformeras all geometrisk information för ett hörn automatiskt från modellkoordinatsystemet till vykoordinatsystemet och projektionskoordinater och ljusvärde beräknas (om så begärts) för hörnet. Detta sker i princip när `glVertex` anropas, dvs övriga intressanta tillstånd (som färg och normaler) skall vara satta dessförinnan.

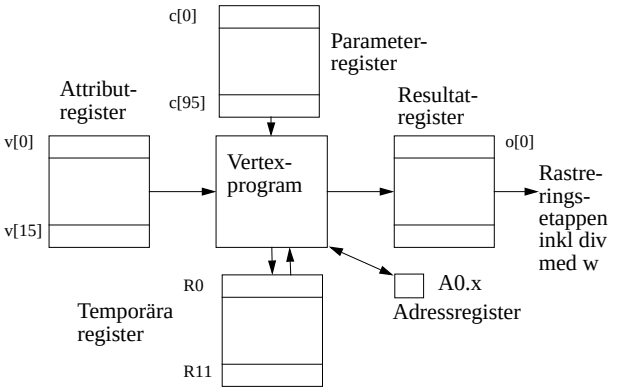


I de enklaste fallen handlar det om att beräkna positionen i projektionskoordinater (exkl divisionen med fjärde komponenten som görs i ett senare steg).

Denna etapp i den grafiska rörledningen är från och med NVIDIAs GeForce3 programmerbar. Men då måste man göra allt själv! Vitsen är att nya effekter kan uppnås och att arbete kan flyttas till grafikprocessorn.

Vertexprogrammering 2(9)

Kanske kan det ha något intresse att se på arkitekturen.



Vertexprogrammet arbetar mot ett antal maskinregister (dvs snabba minnesplatser). Vart och ett av dessa register innehåller fyra flyttal betecknade xyzw (punkter, vektorer) eller strq (texturkoordinater). Adressregistret dock bara ett tal. Antalet register liksom tillåten längd på vertexprogrammet varierar. I **attributregistren** hamnar hörnets attribut - storheter som ofta ändras per hörn. T ex position, färg och texturkoordinater. I **parameterregistren** storheter som ändras mindre ofta. T ex material- och ljusdata liksom diverse transformationsmatriser. Temporärregistren används naturligtvis för mellanräkningar och är unika på så sätt att de är både skriv- och läsbara. Slutligen placerar vertexprogrammet resultaten i **resultatregistren**. Tidigare refererade man till de olika registren med nummer, men GLSL låter oss referera till dem med namn som `vertex.position` och `resultat.color`.

Vertexprogrammering 3(9)

Det enklast tänkbara vertexprogrammet ser ut så här:

```
void main(void) {
    // Transformation till projektionskoordinater
    // Vi multiplicerar MVP-matrisen med positionen
    gl_Position = gl_ModelViewProjectionMatrix*gl_Vertex;
    gl_FrontColor = gl_Color;
}
```

Det finns en inbyggd funktion *ftransform*, som garanterar att transformationen utförs exakt som om vi inte skrev ett eget vertexprogram. Med den blir det i stället

```
void main(void) {
    // Den normala (fixa) transformation
    gl_Position = ftransform();
    gl_FrontColor = gl_Color;
}
```

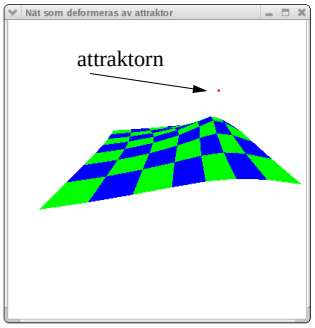
Om man vill krångla till det för sig kan man även skriva

```
void main(void) {
    // OBS! Eftersom m[i] avser en kolumn måste vi
    // använda den transponerade matrisen
    mat4 mvp = gl_ModelViewProjectionMatrixTranspose;
    //En rad i taget. dot = skalärprodukt.
    gl_Position.x  = dot(mvp[0],gl_Vertex);
    gl_Position.y  = dot(mvp[1],gl_Vertex);
    gl_Position.z  = dot(mvp[2],gl_Vertex);
    gl_Position.w  = dot(mvp[3],gl_Vertex);
    gl_FrontColor = gl_Color;
}
```

Vi kan alternativt skriva `gl_FrontColor = vec4(1.0, 0.0, 0.0, 0.0);` så blir allt ritat rött och vi ser hur man kan skriva en konstantvektor. Vill vi att färgläggningen skall göras utifrån modellkoordinaterna, kan det ske med (värden under 0 ger svart, över 1 ger vitt): `gl_FrontColor = sin(gl_Vertex);` Vi får alltså en färgsättning som beror på modellkoordinaterna utan att använda något *glColor*.

Vertexprogrammering 4(9)

Ett exempel hämtat från Mesa 4.1 modifierat för GLSL. Ett rutnät deformeras av en roterande attraktorn ovanför rutnätet. Attraktorns dragningskraft avtar omvänt proportionellt mot avståndet mellan dess position och aktuellt hörn. Vi flyttar en del beräkningar som skulle ha kunnat utföras av värddatorn till grafikprocessorn.



Vertexprogrammet (filen `$DG/DEMOS/Warp.vert;` kan köras med *VertexWarp2005* i samma mapp).

```
// Dessa får alltså värden i vår tillämpning
uniform vec4 Strength; // Attraktorns styrka
uniform vec4 Pos;      //  Dess position
void main(void) {
     vec4 r1, nytt_vertex; float r2;
     // Avståndet från attraktor till hörn beräknas
     // Vektor från hörn till attraktor
     r1 = Pos - gl_Vertex;
     // Avståndet i kvadrat
     r2 = dot(r1,r1);
     // Verkan skall vara omvänt proportionell mot avståndet
     r2 = inversesqrt(r2);
     r2 = r2*Strength.x;
     nytt_vertex = gl_Vertex + r2*r1;
     gl_Position = gl_ModelViewProjectionMatrix*nytt_vertex;
     gl_FrontColor = gl_Color;
}
```

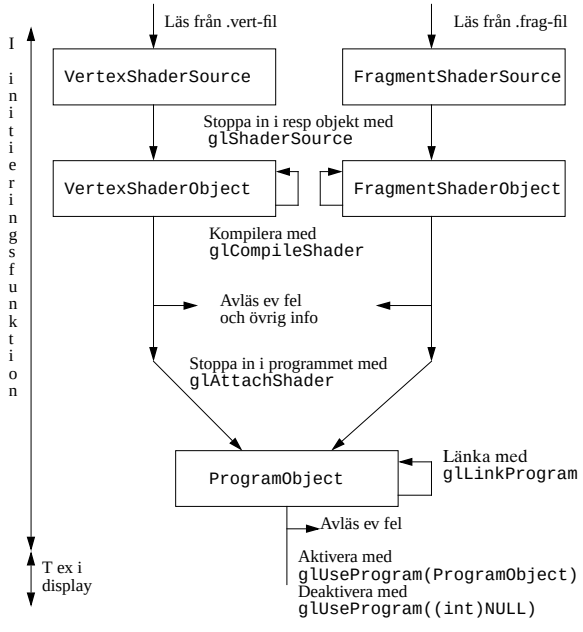

Vertexprogrammering 5(9)

Följande bild visar översiktligt vad som görs i OpenGL-programmet. Vi arbetar med tre globala s k objekt

`GLuint ProgramObject, VertexShaderObject, FragmentShaderObject`

samt två lokala (i *main*) strängar

`GLchar *VertexShaderSource, *FragmentShaderSource.`



Vertexprogrammering 6-9(9)

Över till OpenGL-programmet mest för fullständighetens skull (en del konstigheter förklaras av att man annars får kompileringsfel). Se `$DG/DEMOS/VertexWarp2005.c` för alla detaljer. PC-koden skiljer sig; se sen

```
...
#define GL_GLEXT_PROTOTYPES
#include <GL/glut.h>
GLuint ProgramObject;
GLuint VertexShaderObject;
GLuint FragmentShaderObject;

GLboolean glversion2() {...}
int shaderSize(char *fileName) {...}
int readShader(char *fileName, char *shaderText, int size)
{...}
int  readShaderSource(char  *fileName,  GLchar  **ourShader)
{..}
GLboolean initGLSL(const GLchar *vertexShader,
                   const GLchar *fragmentShader ) {
    GLchar    *pInfoLog;
    GLint compiled = GL_FALSE; //GLboolean ger typfel senare
    GLint linked   = GL_FALSE;
    GLint length, maxLength;
    // Skapa programobjekten.
    ProgramObject      = glCreateProgram();
    VertexShaderObject =
           glCreateShader(GL_VERTEX_SHADER);
    FragmentShaderObject =
           glCreateShader(GL_FRAGMENT_SHADER);
    length = strlen(vertexShader);
    if (vertexShader) {
      // Lägg in vertexprogramsträngen i objektet
      glShaderSource(VertexShaderObject, 1,
                  &vertexShader, NULL); //&length);
      // Man kan frisläppa utrymmet för strängen
      free((char*)vertexShader);
      // Kompilera vertexprogrammet och skriv ut ev info
      glCompileShader(VertexShaderObject);
```



```
      glGetProgramiv(VertexShaderObject,
                GL_COMPILE_STATUS, &compiled);
      glGetProgramiv(VertexShaderObject,
               GL_INFO_LOG_LENGTH, &maxLength);
      pInfoLog=(GLchar *)malloc(
                maxLength*sizeof(GLchar));
      glGetProgramInfoLog(VertexShaderObject, maxLength,
                &length, pInfoLog);
      printf("%s", pInfoLog);
      free(pInfoLog);
      if (!compiled) {
         printf("Misslyckad kompilering av VertexShader\n");
         return GL_FALSE;
      }
      printf("Lyckad kompilering av VertexShader\n");
      // Stoppa in i programobjektet
      glAttachShader(ProgramObject,VertexShaderObject);
      // Det är bra att ta bort shaderobjektet nu        /
      glDeleteShader(VertexShaderObject);
    } // Slut på vertexshaderdelen
    // Motsv för fragmentshader
    // Länka och skriv ut ev info
    glLinkProgram(ProgramObject);
    glGetProgramiv(ProgramObject,
              GL_LINK_STATUS, &linked);
    glGetProgramiv(ProgramObject,
              GL_INFO_LOG_LENGTH, &maxLength);
    pInfoLog=(GLchar *)malloc(
              maxLength*sizeof(GLchar));
    glGetProgramInfoLog(ProgramObject, maxLength, NULL,
              pInfoLog);
    printf("%s\n", pInfoLog);
    free(pInfoLog);
    // Aktivera programobjektet
    if (linked) {
          glUseProgram(ProgramObject);
          return GL_TRUE;
    } else { return GL_FALSE; }
}
```



```
void DrawMesh( int rows, int cols ) {
   static const GLfloat colorA[3] = { 0, 1, 0 };
   static const GLfloat colorB[3] = { 0, 0, 1 };
   const float dx = 2.0 / (cols - 1);
   const float dy = 2.0 / (rows - 1);
   float x, y;
   int i, j;
   y = -1.0;
   for (i = 0; i < rows - 1; i++) {
      glBegin(GL_QUAD_STRIP);
         x = -1.0;
         for (j = 0; j < cols; j++) {
            if ((i + j) & 1)  glColor3fv(colorA);
            else glColor3fv(colorB);
            glVertex2f(x, y); glVertex2f(x, y + dy);
            x = x + dx;
         }
      glEnd();
      y = y + dy;
   }
}
GLfloat Phi = 0.0, X=1.5;
// Snurrar på attraktorn
void idle( void ) {
   Phi += 0.01;  glutPostRedisplay();
}
void reshape( int width, int height ) {
   glViewport( 0, 0, width, height );
   glMatrixMode( GL_PROJECTION );
   glLoadIdentity();
   gluPerspective(90,1.0,0.1,25.0);
}
void display( void ) {
   GLfloat x, y, z, r = 0.5;
   glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
   glMatrixMode( GL_MODELVIEW );
   glLoadIdentity();
   gluLookAt(0,0,X, 0,0,0, 0,1,0);
   glRotatef(-60.0, 1, 0, 0);
```

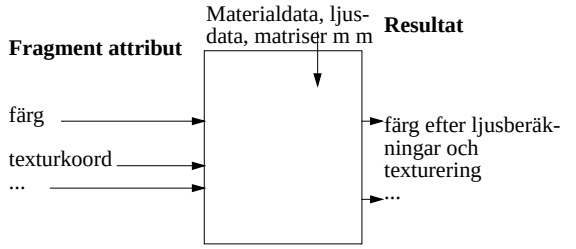


```
   // Position the gravity source
   x = r * cos(Phi); y = r * sin(Phi); z = 0.7;
   glUniform4f(glGetUniformLocation(
                     ProgramObject,"Pos"), x, y, z, 1);
   // Rita ljuskällan röd
   glUseProgram(NULL);//0 i st f NULL hindrar varning
   glBegin(GL_POINTS);
         glColor3f(1,0,0); glVertex3f(x, y, z);
   glEnd();
   glUseProgram(ProgramObject);
   DrawMesh(8, 8);
   printf("glGetError=%s\n",
          gluErrorString((int)glGetError()));
   glutSwapBuffers();
}

int main(int argc, char **argv) {
    GLchar *VertexShaderSource;
    glutInitDisplayMode(GLUT_RGBA|GLUT_DEPTH|GLUT_DOUBLE );
    glutInitWindowSize(400, 400);
    glutCreateWindow("Nät som deformeras av attraktor");
    if (!glversion2()) {
       printf("GLSL stöds inte av denna dator.\n");
       exit(1);
    }
    glutDisplayFunc(display);
    glutIdleFunc(idle);
    glutReshapeFunc(reshape);
    readShaderSource("Warp.vert",&VertexShaderSource);
    initGLSL(VertexShaderSource, NULL);
    glUniform4f(glGetUniformLocation(
              ProgramObject,"Strength"), 0.5,0,0,0);
    glEnable(GL_DEPTH_TEST);
    glClearColor(1.0, 1.0, 1.0, 1);
    glShadeModel(GL_FLAT);
    glPointSize(3);
    glutMainLoop();
    return 0;
}
```

Fragmentprogrammering 1(4)

I den vanliga rörledningen behandlas fragmentet enligt figuren nedan. Ljusberäkningarna är redan gjorda på vertexnivå (om vi inte använt ett vertexprogram). Väsentligen handlar det därför om texturering i ett eller flera steg (och dimma som vi inte tagit upp).



Denna etapp i den grafiska rörledningen är från och med NVIDIAs GeForce3 programmerbar. Det viktigaste användningsområdet är mer realistiska ljusberäkningaar, t ex Phong-toning, som ju måste göras per bildpunkt. Det praktiska maskineriet är mycket likt det vid vertexprogrammering. I själva OpenGL-programmet läser man in och kompilerar m m fragmentprogrammet på samma sätt som när det gällde vertexprogrammet. Det är bara när man med *glCreateShader* skapar ett fragmentshader-objekt som parametern skall vara annorlunda.

Fragmentprogrammering 2(4)

När det gäller fragmentprogrammets variabler (se tidigare tabell) avser nu

- `gl_Color` det färgvärde som givits `gl_FrontColor` resp `gl_BackColor` i vertexprogrammet om sådant används, annars det värde som beräknats av standardmaskineriet. Fragmentet kommer i förekommande fall både som ett framsides- och ett baksidesfragment.
- `gl_TexCoord[i]` det värde som tilldelats `gl_TexCoord[i]` i vertexprogrammet eller om sådant inte ingår det värde som hör ihop med den i:te texturenheten. Värdet är med interpolation framräknat från hörnvärdena. Detta gäller även t ex `gl_Color`.
- `gl_FrontMaterial` och `gl_BackMaterial` med komponenterna `.emission`, `.ambient`, `.diffuse`, `.specular` och `.shininess` materialvärden som satts med *glMaterial* i OpenGL-programmet. Den sista komponenten är `float` medan övriga är `vec4`.
- `gl_LightSource[i]` med bl a komponenterna `.ambient`, `.diffuse`, `.specular`, `.position` och `.halfVector` ljusvärden som satts med *glLight* (den sista är automatiskt framräknad). De nämnda komponenterna är alla av typen `vec4`.
- `gl_FragColor` utgående färg. Denna variabel kan även läsas och kan ges värde flera gånger, vilket ibland är bekvämt.

Fragmentprogrammering 3(4)

Två enkla fragmentprogram ser ut så här:

```
void main(void) {
    gl_FragColor = gl_Color;
    //gl_FragColor = vec4(1.0, 1.0, 0.0, 0.0);
}
```

Om vi flyttar kommentartecknen en rad uppåt blir fragmentet gult oberoende av vad som hänt tidigare.
Ett något intressantare fragmentprogram är

```
uniform sampler2D enhetsnr;
void main(void) {
    gl_FragColor = texture2D(enhetsnr, gl_TexCoord[3].st);
}
```

som sätter utgångsfärgen till det värde i texturkartan hörande till texturenheten *enhetsnr* som pekas ut av texturkoordinaterna angivna med `glMultiTexCoord2f(GL_TEXTURE3,...)`. Variabeln *enhetsnr* måste ha tilldelats ett värde med

```
glUniformi(glGetUniformLocation(
ProgramObject,"enhetsnr"),heltal);
```

i OpenGL-programmet. I det måste också texturen bindas till en plats i texturregistret (och blir då ett texturobjekt), men `glEnable(GL_TEXTURE_2D)` och *glTexEnvf*-anropen behövs nu inte.

I ett glUniform-anrop tar vi först med `glGetUniformLocation(ProgramObject,"variabelnamn")` reda på platsen (registernummer, heltal 0 och uppåt) för variabeln och ger den sedan ett värde. Detta kan verka primitivt, men tanken är att man effektivt skall kunna ta reda på platsen en gång för alla.

Utan fragmentprogram utförs ju ett antal operationer per fragment. Med utökar vi bara antalet operationer, vilket gör att korta fragmentprogram inte kostar märkbart i tid.

Fragmentprogrammering 4(4)

Nu en kombination av vertex- och fragmentprogram som gör Phongtoning (enbart av det diffusa ljuset här och vi låtsas som om ljuskälla och betraktare är långt bort, vilket gör att vektorn L mot ljuskällan är konstant över objektet). Vi antar att ljuset är maximalt vitt, dvs {1.0, 1.0, 1.0, 1.0}. Först `LIGHT.vert`:

```
void main(void) {
    // Transformation från modellkoord till projkoord
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    // Transformera normalen till vykoordinater, se "Från .."
    // Lägg som texturkoordinat 0 för interpolatio
    // gl_Normal är vec3 och gl_NormalMatrix mat3
    gl_TexCoord[0].stp = gl_NormalMatrix * gl_Normal;
    // Borde vara likvärdigt men namnet finns ej hos oss
    //gl_TexCoord[0] = gl_ModelViewMatrixInverseTranspose
                        * (vec4(gl_Normal,0.0));
    // Nu kommer vi åt den interpolerade normalen
    // i fragmentprogrammet
}
```

Över till fragmentprogrammet `LIGHT.frag`:

```
uniform vec3 LightPos;
void main(void) {
   vec4 diffuse = gl_FrontMaterial.diffuse;
   vec4 specular = gl_FrontMaterial.specular;
   vec3 L, N; float dotProd;
   // Vi har en ljusriktning, denna har transformerats och
   // avser vykoordinater. Normalisera den
   // Tycks inte uppdateras
   // L = normalize(gl_LightSource[0].position.xyz);
   L = normalize(LightPos);
   // Normalisera den interpolerade normalen
   N = normalize(gl_TexCoord[0].stp);
   // Beräkna skalärprodukten LN
   dotProd = dot(L,N); gl_FragColor = diffuse * dotProd;
}
```