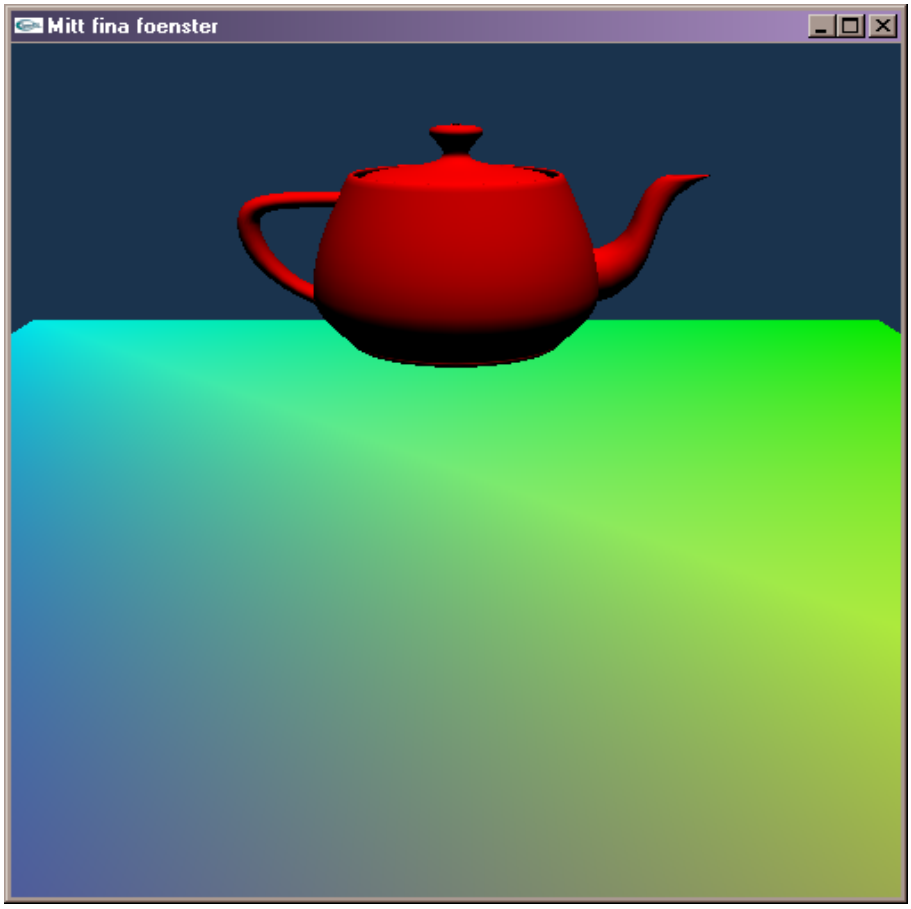


En enkel Cg-shader

Gustav Taxén, 2007-04-11

Översikt

I `main` initieras GLUT precis som vanligt. Där initieras också Cg och en s.k. *kontext* skapas där Cg-programmen (vertex- och fragmentprogram) sedan kommer att lagras. Nästa steg är att läsa in programmen (”vertex.cg” och ”fragment.cg”) och kompilera dem. Sedan lämnas kontrollen över till GLUT som vanligt.

`display`-funktionen börjar med att aktivera Cg-programmen. Därefter ser koden ut ungefär som vanligt, med skillnaden att funktionen `passMatricesToVertexProgram` anropas innan varje objekt ritas ut. ***Detta är viktigt***. Eftersom vi själva är ansvariga för transformationer är vi också ansvariga för att se till att de matriser vi använder är rätt. Det är enklast att låta OpenGL skapa och multiplicera ihop transformationsmatriserna (med `gluLookAt`, `glTranslate`, etc.), så vi ”hämtar” dem från OpenGL och skickar dem till vertexprogrammet innan vi ritar geometrin. *Det är viktigt att inse att det inte finns någon direktkoppling mellan uppdateringarna av OpenGLs interna Modelview-matris och de matriser som finns i vertex-programmet.* Matriserna i Cg-programmet ändras bara när vi explicit uppdaterar dem. Vi drar helt enkelt bara nytta av att OpenGL redan har bra matrisfunktioner inbyggda. Vi skulle lika gärna ha kunnat använda ett annat matrispaket med funktioner för matrismultiplikation, invertering och transponering, etc.

Vi behöver skicka två matriser till vertexprogrammet:

1) Resultatet av att multiplicera Modelview- och Projektionsmatriserna. Denna matris konverterar från *objektkoordinater* till *klippkoordinater* (som rastreraren behöver).
2) $(\text{Modelview}^{-1})^{T}$. Denna matris konverterar normaler från *objektkoordinater* till *ögonkoordinater*. Vi behöver normalerna i ögonkoordinater eftersom vi vill använda dem för ljussättningsberäkningar.

I `passMatricesToVertexProgram` skickar vi också riktningen mot ljuskällan till vertexprogrammet. Ljusriktningen antas vara i ögonkoordinater. ***Det är viktigt att förstå skillnaden mellan objekt-, ögon- och klippkoordinater, och varför man gör ljussättning i ögonkoordinater***. Läs i kursboken om du är osäker!

Vertexprogrammet, som exekveras för varje hörn, tar följande indata:

- Hörnets position i objektkoordinater. Det är dessa koordinater vi anger med `glVertex`.
- Hörnets normal i objektkoordinater. Dessa anges med `glNormal`.
- Hörnets färg, anges med `glColor`.

Utöver detta har vi också matriserna, som alltså uppdateras separat. Utdata från vertexprogrammet är:

- Hörnets position i klippkoordinater. Det är dessa positioner som rastreraren ”konverterar” till fragment.
- Hörnets normal i *ögonkoordinater*. Dessa ögonkoordinater kommer automatiskt att interpoleras över polygonerna av rastreraren.
- Hörnets färg, interpoleras också av rastreraren.

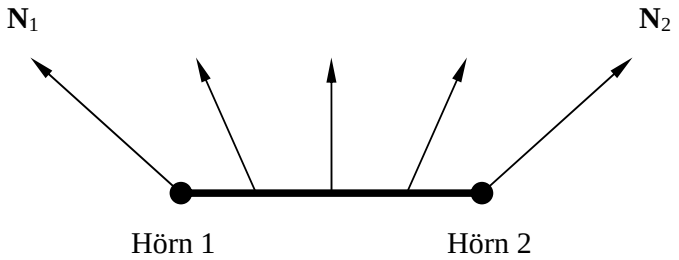
Notera att normalerna och färgerna har `TEXCOORD` som *binding semantics*, d.v.s., de betecknas som texturkoordinater. Texturkoordinater är det som oftast används för att skicka data mellan vertex- och fragmentprogram. Anledningen till att man säger `TEXCOORD0, TEXCOORD1,` etc., trots att man inte skickar texturkoordinater har att göra med hur grafikkorten utvecklats funktionsmässigt[1]. Värdena som skickas interpoleras på exakt samma sätt av rastreraren oavsett vilken binding semantics de har. Däremot är det givetvis viktigt att indata till fragmentprogrammet har samma binding semantics som utdata från vertexprogrammet, annars ”tappar” du data mellan programmen!

Fragmentprogrammet, som anropas för varje fragment i polygonerna, tar följande indata:

- Fragmentets (interpolerade) normal i ögonkoordinater.
- Fragmentets (interpolerade) färg.

Utöver detta har vi också ljusriktningen (i ögonkoordinater).

Fragmentprogrammet normerar först ljusriktningen – det kan hända att användaren inte skickat in en normerad riktning, så det är bäst att vara på den säkra sidan. Sedan normeras normalen också. Anledningen till detta är att rastreraren kan producera normaler av annan längd än 1 när de interpoleras, även om användaren angett normaler i hörnen som har längden 1. I bilden nedan har normalerna $\mathbf{N}_1$ och $\mathbf{N}_2$ längden 1, men de interpolerade normalerna mellan hörn 1 och 2 får kortare längd.



Lamberts reflektionsmodell används för att ljussätta fragmentet.

[1] Binding semantics anger vilket *register* på GPUn som ska användas för att skicka data mellan programmen.

main.c

```
#include <windows.h>
#include <GL/glut.h>
#include <Cg/cg.h>
#include <Cg/cgGL.h>
#include <stdio.h> /* For printf() */

/* Variable to hold the Cg context that we're storing our programs
   in, as well as handles to the vertex and fragment program used
   in this demo. */
CGcontext context;
CGprogram vertexProgram, fragmentProgram;
CGprofile vertexProfile, fragmentProfile;

/* Cg error handler */
void handleCgError(void) {
  const char *err = cgGetErrorString(cgGetError());
  printf("Cg error %s\n", err);
  exit(1);
}

/* Select an appropriate cg profile */
void ChooseProfiles(void) {
  if (cgGLIsProfileSupported(CG_PROFILE_ARBVP1))
    vertexProfile = CG_PROFILE_ARBVP1;
  else {
    if (cgGLIsProfileSupported(CG_PROFILE_VP30))
      vertexProfile = CG_PROFILE_VP30;
    else {
      fprintf(stderr,
              "Vertex profiles not supported on this system.\n");
      exit(1);
    }
  }
  if (cgGLIsProfileSupported(CG_PROFILE_ARBFP1))
    fragmentProfile = CG_PROFILE_ARBFP1;
  else {
    if (cgGLIsProfileSupported(CG_PROFILE_FP30))
      fragmentProfile = CG_PROFILE_FP30;
    else {
      fprintf(stderr,
              "Fragment profiles not supported on this system.\n");
      exit(1);
    }
  }
}
```

```
/* Load and compile the vertex and fragment program */
void LoadCgPrograms(void) {
  vertexProgram = cgCreateProgramFromFile(context,
                                          CG_SOURCE,
                                          "vertex.cg",
                                           vertexProfile,
                                          NULL, NULL);
  if (!cgIsProgramCompiled(vertexProgram))
    cgCompileProgram(vertexProgram);
  cgGLEnableProfile(vertexProfile);
  cgGLLoadProgram(vertexProgram);

  fragmentProgram = cgCreateProgramFromFile(context,
                                            CG_SOURCE,
                                            "fragment.cg",
                                            fragmentProfile,
                                            NULL, NULL);
  if (!cgIsProgramCompiled(fragmentProgram))
    cgCompileProgram(fragmentProgram);
  cgGLEnableProfile(fragmentProfile);
  cgGLLoadProgram(fragmentProgram);
}

/* Called when window changes size */
void reshape(int w, int h) {
  /* Set a perspective projection */
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(45.0f, (float)w / (float)h, 0.1f, 50.0f);

  /* Make sure the OpenGL drawing fills the entire window */
  glViewport(0, 0, w, h);
}


/* Pass the modelview and projection matrices to the
   vertex program */
/* Also pass the light position (in eye coordinates) to the
   fragment program */
void passMatricesToVertexProgram(void) {
  cgGLSetStateMatrixParameter(cgGetNamedParameter(vertexProgram,
                                                  "ModelViewProj"),
                              CG_GL_MODELVIEW_PROJECTION_MATRIX,
                              CG_GL_MATRIX_IDENTITY);
  cgGLSetStateMatrixParameter(cgGetNamedParameter(vertexProgram,
                                                  "ModelViewInvT"),
                              CG_GL_MODELVIEW_MATRIX,
                              CG_GL_MATRIX_INVERSE_TRANSPOSE);

  cgGLSetParameter3f(cgGetNamedParameter(fragmentProgram,
                                         "LightDirEye"),
                     0.1f, 1.0f, 1.0f);
}
```

```
/* Callback for repainting the window */
void display(void) {
  /* Make sure that the vertex and fragment programs are bound. */
  cgGLBindProgram(vertexProgram);
  cgGLBindProgram(fragmentProgram);

  /* Clear the window and the depth buffer */
  glClearColor(0.1, 0.2, 0.3, 1.0);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  /* Enable the depth test */
  glEnable(GL_DEPTH_TEST);

  /* Make sure normals are always of length 1, even if we're
     scaling */
  glEnable(GL_NORMALIZE);

  /* Set the modelview matrix */
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt(0, 6, 15,
            0, 0, 0,
            0, 1, 0);

  /* Draw the ground polygon, a quadrilateral */

  passMatricesToVertexProgram();

  glBegin(GL_QUADS);
      glNormal3f(0.0f, 1.0f, 0.0f);
      glColor3f(0.0f, 0.0f, 1.0f);
      glTexCoord2f(0.0f, 0.0f);
      glVertex3f(-10.0f, 0.0f, 10.0f);

      glColor3f(1.0f, 1.0f, 0.0f);
      glTexCoord2f(1.0f, 0.0f);
      glVertex3f(10.0f, 0.0f, 10.0f);

      glColor3f(0.0f, 1.0f, 0.0f);
      glTexCoord2f(1.0f, 1.0f);
      glVertex3f(10.0f, 0.0f, -10.0f);

      glColor3f(0.0f, 1.0f, 1.0f);
      glTexCoord2f(0.0f, 1.0f);
      glVertex3f(-10.0f, 0.0f, -10.0f);
  glEnd();

  /* Draw a teapot */

  glTranslatef(0.0f, 2.0f, 0.0f);
  glRotatef(-90.0f, 1.0f, 0.0f, 0.0f);

  passMatricesToVertexProgram();

  glColor3f(1.0f, 0.0f, 0.0f);
  glutSolidTeapot(1.0f);

  /* Show the back buffer */
  glutSwapBuffers();
}
```

```
/* Called when user presses a key */
void keyboard(unsigned char key, int x, int y) {
  if (key == 'q' || key == 27) { /* 27 = Escape (on PCs, at least) */
    exit(0);
  }
}

int main(int argc, char *argv[]) {
  /* Initialize GLUT */
  glutInit(&argc, argv);

  /* Tell GLUT we want a depth buffer and a back buffer */
  glutInitDisplayMode(GLUT_RGB | GLUT_DEPTH | GLUT_DOUBLE);

  /* Suggest a window size */
  glutInitWindowSize(500, 500);

  /* Create a window */
  glutCreateWindow("Mitt fina foenster");

  /* Set the redisplay callback */
  glutDisplayFunc(display);

  /* Set the reshape callback */
  glutReshapeFunc(reshape);

  /* Set the keyboard callback */
  glutKeyboardFunc(keyboard);

  /* Initialize Cg */
  cgSetErrorCallback(handleCgError);
  context = cgCreateContext();
  ChooseProfiles();
  LoadCgPrograms();

  /* Leave control to GLUT */
  glutMainLoop();

  return 0;
}
```

vertex.cg

```
// Vertex program

void main(float4 Pobject : POSITION, // Vertex in obj coords
          float4 Nobject : NORMAL,   // Vertex normal in obj coords
          float4 Color   : COLOR,    // Vertex color
          uniform float4x4 ModelViewProj, // Modelview * Projection
          uniform float4x4 ModelViewInvT, // Modelview inverse transp
          out float4 HPosition : POSITION, // Vertex in eye coords
          out float4 Neye : TEXCOORD0,     // Normal in eye coords
          out float4 ColorOut : TEXCOORD1) // Vertex color
{
  // transform position into clip coords and pass it to rasterizer
  HPosition = mul(ModelViewProj, Pobject);

  // transform normal into eye coords and pass it to rasterizer
  Neye = mul(ModelViewInvT, Nobject);

  // pass through color to rasterizer
  ColorOut = Color;
}
```

fragment.cg

```
// Fragment program

float4 main(float4 Neye: TEXCOORD0,   // Normal of this fragment
            float4 Color : TEXCOORD1, // Color of this fragment
            uniform float3 LightDirEye) : COLOR  // Light direction
{
  // Make sure light direction is normalized
  LightDirEye = normalize(LightDirEye);

  // Normalize the normal (the interpolation may
  // produce normals that aren't of length 1, even
  // if the user inputs normals of length 1)
  float3 n = Neye.xyz;  // Extract x, y, and z from 4D vector
  n = normalize(n);

  // Lighting = L dot N
  float ln = saturate(dot(LightDirEye, n));

  // Return the final color of this fragment
  return float4(ln, ln, ln, 1.0f) * Color;
}
```