Fragmentprogrammering: Felhantering

Se OH-242 för hur felutskrifter från t ex kompilering av ett vertex- eller fragmentprogram kodas.

De flesta fel upptäcks vid körningen av OpenGL-programmet. Några exempel (talet inom parentes anger radnummer):

Rad utan avslutande semikolon
`(9) : error C0000: syntax error, unexpected identifier, expecting ';'`
Deklaration utan typangivelse
`(9) : error C0501: type name expected at token "r1"`
Odeklarerad variabel ges värde
`(9) : error C1054: initialization of non-variable "r1"`
Odeklarerad (felstavad) variabel
`(13) : error C1008: undefined variable "gl_vertex"`
Typkonflikt vid addition
`(13) : error C1022: operands to "add" must be numeric`

Sedan kan naturligtvis OpenGL-programmet gå fel med den tråkiga utskriften
`Segmentation fault`
Då bör man tillkalla handledare om man inte är bevandrad i avlusaren *gdb*. Denna avlusare fungerar ev dåligt i Linux-miljö mot program länkade med de dynamiska grafikbiblioteken, vilket `OGL_COMPILE` gör. Man kan i stället använda `OGL_COMPILE_STAT`, som länkar mot ett statiskt GLUT-bibliotek.

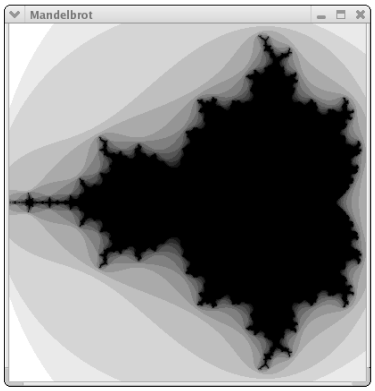
Fragmentprogrammering: Mandelbrot i GPU 1(3)

I OpenGL-programmet (`$DG/DEMOS/MANDELFRAG2005.c`; ej omskrivet för OpenGL 2.0) ritar vi en kvadrat motsvarande den del av komplexa talplanet vi är intresserade av.

Vertexprogrammet (`$DG/DEMOS/Mand.vert`)

```
varying vec2 pos;// för kommunikation med fragmentprogrammet
void main(void) {
     gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
     pos = vec2(gl_Vertex.x, gl_Vertex.y);
}
```

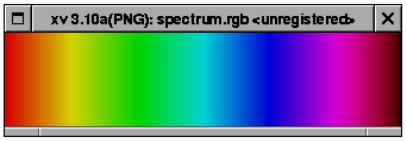
Fragmentprogrammet (`$DG/DEMOS/Mand.frag`) på nästa sida.



Fragmentprogrammering: Mandelbrot i GPU 2(3)

```
varying vec2 pos;
uniform sampler2D enhetsnr;
void main(void) {
     vec2 c = pos; // pos is read-only
     float size; int n = 0;
     vec2 z = vec2(0.0, 0.0);
     vec2 znew;
     float del;
     float iter;
     float max = 12.0;
     for (iter = 0.0; iter < max; iter++) {
          znew.x = z.x*z.x -z.y*z.y + c.x;
          znew.y = 2*z.x*z.y+c.y;
          z = znew;
          size = dot(z,z);
          if (size < 4.0) n = n + 1;
     }
     del = 1-n/max;
     // gl_FragColor = texture2D(enhetsnr, del,0.5);
     gl_FragColor = vec4(del,del,del,0);
     // gl_FragColor=noise1(z.x);
     // ger alltid 0 enligt info april 2005
     // gl_FragColor=vec4(sin(z.x),sin(z.y),0,0); snygg!!!
}
```

Den först bortkommenterade raden ger färg om vi använder texturen



Fragmentprogrammering: Mandelbrot i GPU 3(3)

Texturen är egentligen 1-dimensionell men läst som 2-dimensionell, varför jag använt `sampler2D` och `texture2D` i programmet. texture2D(...) använder den textur som f n är bunden till den texturenhet som anges av enhetsnr. Texturkoordinaterna anges som de två sista parametrarna.

Inläsning etc av textur kan vi i detta fall göra i *main* i OpenGL-programmet

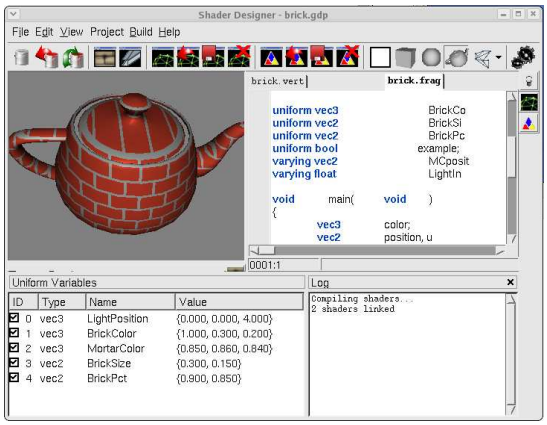
```
// texture global vektor
glGenTextures(1,texture);
// Vi lägger texturen i texturenhet 0
glActiveTexture(GL_TEXTURE0);
// Läsning av texturen och glBindTexture och glTexEnvf
//samt glTexParameteri
...
// Meddela fragmentprogrammet texturenhetens nummer
glUniform1i(glGetUniformLocation(
                        ProgramObject,"enhetsnr"),0);
```

Vertex- och fragmentprogrammering: Övrigt

Den kraft som grafikprocessorerna erbjuder kanske kan användas för allmänna beräkningar? Det finns grupper som sysslar med just detta, *General-Purpose computation on GPUs* (förkortat GPGPU). Till och med flera webbsidor, bl a `www.gpgpu.org`.
Det finns en OpenGL-emulator Mesa, som går att köra på en godtycklig dator oberoende av grafikkort. Den motsvarar OpenGL 1.5 och stöder därmed f n inte GLSL och inte heller de OpenGL-funktioner som används då. Däremot klarar den lågnivå-varianten av vertex/fragmentprogrammering. Arbete sägs pågå för att höja Mesa till OpenGL 2.0-nivån.

Utvecklingsmiljöer för vertex- och fragmentprogram 1(3)

Med detta begrepp avses program där man kan experimentera med vertex- och fragmentprogram och se hur de påverkar olika typer av objekt. Sådana finns för Microsofts HLSL och GLSL (och föregångare).
Vi intresserar oss här bara för GLSL. Jag har hittat en - kallad *ShaderDesigner* (`http://www.typhoonlabs.com`) - som finns för både Linux (dock kommersiell numera) och Windows. Min Linux-version av den är dålig (textvisning och -redigering är opålitlig), men belyser ideerna. När det gäller Windows-versionen tycks mina rättigheter inte räcka till för en korrekt installation under StuDAT. Så här ser det i alla fall ut under Linux:

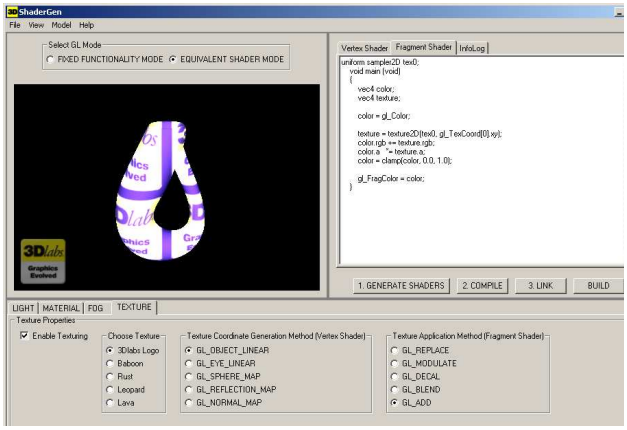


Utvecklingsmiljöer för vertex- och fragmentprogram 2(3)

I figurdelen kan man rotera och förflytta objektet liksom zooma med musknapparnas hjälp. Med verktygen till höger upptill kan vi växla mellan några olika objekt. I figuren är tekannan vald. Därunder kan vi inspektera antingen vertex- eller fragmentprogrammet och även i princip ändra i dem. Under figurdelen hittar vi en variabellista. Genom att klicka på en rad kan man med viss svårighet ändra värden.

Starta programmet med *shaderdesigner*. Med **File**/**Open Shader Project** öppnar du `$DG/SHADERDESIGNER/shaders/brick.gdb`.

För Windows har vi i stället ett liknande program kallat *ShaderGen* från 3DLabs (`http://www.3dlabs.com`).



Utvecklingsmiljöer för vertex- och fragmentprogram 3(3)

ShaderGen är tänkt att visa hur ett standardbeteende kan kodas med vertex- och fragmentprogram. Välj inställningar som i figuren och tryck på knappen **Build**. Du får då upp en textur applicerad på något objekt. Objekt kan bytas med **Model**-menyn (jag har valt Klein). Fragmentprogrammet kan studeras liksom vertexprogrammet. Man kan också ändra i dessa, vilket får genomslag efter **Compile** + **Link**. Ändringar av inställningarna i nedre halvan får ofta omedelbar effekt.

Gå till mappen `$DG/PC/SHADERGEN/bin` och starta `ShaderGen.exe`. Experimentera litet. Ändra t ex till gl_FragColor = vec4(1,0,0,0).

En Mac-ägabde kursdeltagare nämnde *OpenGL Shader Builder* för MacOS X. Ngot oklart om den klarar GLSL.

Det finns andra "shading"-språk. Det mest kända är **Renderman** (`http://www.renderman.org`), som har använts för filmframställning och är oberoende av grafikprocessor. Något fritt program för det språket känner jag just nu inte till.

Sh (http://libsh.org) "is a library that acts as a language embedded in C++, allowing you to program GPUs (Graphics Processing Units) and CPUs for graphical and general-purpose computations in novel ways."

Uppgift 7, laboration 3

a) **Linux-användare**: Starta programmet med *shaderdesigner*. Med **File**/**Open Shader Project** öppnar du `$DG/SHADERDESIGNER/shaders/brick.gdb`. Experimentera men försök inte ändra i programtexten.
Windows-användare: Gå till mappen `$DG/PC/SHADERGEN/bin` och starta `ShaderGen.exe`. Experimentera litet. Ändra t ex till `gl_FragColor = vec4(1,0,0,0)`.
b) Kopiera filerna `LIGHT.vert` och `LIGHT.frag` i `$DG/DEMOS` till din egen katalog. Kör programmet `$DG/DEMOS/LABVERTEXFRAG`. Det arbetar i Phong-moden med vertex- och fragmentprogrammet. Använd menyerna knutna till MK3. `LIGHT.frag` gör fragmentvis ljusberäkning men tar bara hänsyn till den diffusa reflektionen. Ändra i `LIGHT.frag` så att hänsyn till spegelreflektionen tas. Låt vektorn V mot betraktaren vara (0,0,1). Använd som exponent n talet 20.0. Beräkna det spekulära ljuset som $(V \bullet R)^n$, där R=2(L•N)N-L normerad (vår modells sätt). R kan beräknas med `reflect(-L,N)`. Alla vektorer skall vara normerade och alla beräkningar görs i vykoordinatsystemet. Använd `pow(float,n)` för upphöjt till.
OBS! Du skall bara utöka fragmentprogrammet, inte ändra i OpenGL-programmet. Uppgiften är därmed oberoende av om du hittills arbetat i C, C++ eller Java. **Redovisa b-delen av uppgiften**.
Anm. Källkoden `LABVERTEXPROG.c` finns i samma mapp, utan att jag gjort någon slutlig tillsnyggning av koden. Den har inte heller anpassats till OpenGL 2.0. Jag hade förra året märkliga problem med de knyckta läsfunktionerna och tvingats kommentera bort ett *fclose* och ett *free*. Därefter har allt gått bra.
Uppgift 8, lab 3 utgår men det hindrar ju inte att du försöker animera.