Számítógépes grafika

XX. rész

A GPU programozása – a GLSL nyelv

Az OpenGL árnyaló nyelve a GLSL *(OpenGL Shading Language)*, amely segítségével *vertex-* és *pixel-* (fragment) *shader*ek által programozhatjuk a GPU-t. A vertex-shader lefut minden vertexre, a pixel-shader lefut minden egyes képernyőre kerülő pixelre.

A GLSL egyszerű *C* szintaxisra épülő nyelv, amely a következő adattípusokkal rendelkezik:

- `float`, `int`, `bool`, `void`: *C*-szerű típusok
- `vec2`, `vec3`, `vec4`: 2, 3 és 4 lebegőpontos elemű vektorok
- `ivec2`, `ivec3`, `ivec4`: 2, 3 és 4 egész elemű vektorok
- `bvec2`, `bvec3`, `bvec4`: 2, 3 és 4 boolean elemű vektorok
- `mat2`, `mat3`, `mat4`: 2×2, 3×3, 4×4-es lebegőpontos mátrixok
- `mat2x2`, `mat2x3`, `mat2x4`, `mat3x2`, `mat3x3`, `mat3x4`, `mat4x2`, `mat4x3`, `mat4x4`: a nevüknek megfelelő méretű mátrixok
- `sampler1D`, `sampler2D`, `sampler3D`: 1D, 2D és 3D textúra
- `samplerCube`: „Cube Map” textúra
- `sampler1Dshadow`, `sampler2Dshadow`: 1D és 2D „depth-component” textúra

Felhasználó által definiálható típusként léteznek a struktúrák (`struct`) és a tömbök (`[]`).

A programozást számos beépített változó segíti, a fontosabbak a következők:

- `gl_Vertex`: egy 4D vektor, a vertex helyzetvektora;
- `gl_Normal`: 3D vektor, a vertexhez tartozó normális;
- `gl_Color`: 4D vektor, a vertex RGBA színe;
- `gl_MultiTexCoordX`: 4D vektor, az *X*. textúra-elem koordinátái;
- `gl_ModelViewMatrix`: 4×4-es mátrix, a modell-nézet mátrix;
- `gl_ModelViewProjectionMatrix`: 4×4-es mátrix, a modell-nézet és vetítési mátrix;
- `gl_NormalMatrix`: 3×3-as mátrix, amelyet transzformációkhoz használ a rendszer;
- `gl_FrontColor`: 4D vektor, az előtér színe;
- `gl_BackColor`: 4D vektor a háttér színe;
- `gl_TexCoord[X]`: 4D vektor, az *X*-edik textúra koordinátái;
- `gl_Position`: 4D vektor, az utoljára feldolgozott vertex koordinátái vertex-shader esetén;
- `gl_FragColor`: 4D vektor, az utoljára írt pixel színe pixel-shader esetén;
- `gl_FragDepth`: valós érték, a mélységbufferbe utoljára beírt mélységérték pixel-shader esetén.

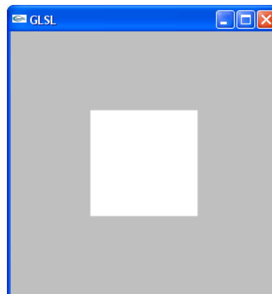
A nyelv más elemei (pl. operátorok, függvények stb.) megegyeznek a *C*-vel, azzal a különbséggel, hogy léteznek tömbműveletek, például összeadást (+), szorzást (*) stb. tömbökkel is végezhetünk.

A számításokat számos függvény is segíti a szögkonverziós és trigonometriai függvényektől a különböző analitikus mértan képleteken át a vektor- és mátrixműveletekig. Álljon itt egy pár példa:

- `genType` **`radians`**`(genType degrees)`: a fokban mért szöget radiánná alakítja: $\pi/180 \cdot$ `degrees`;
- `genType` **`sin`**`(genType angle)`: szinusz függvény;
- `genType` **`pow`**`(genType x, genType y)`: hatványok: x^y;
- `genType` **`sqrt`**`(genType x)`: négyzetgyök: $\sqrt{x}$;
- `genType` **`min`**`(genType x, genType y)`: minimum függvény: x, ha $x < y$, különben y;
- `genType` **`clamp`**`(genType x, genType minVal, genType maxVal)`: szorító függvény: `min(max(x, minVal), maxVal)`;
- `genType` **`mix`**`(genType x, genType y, genType a)`: az `x` és `y` lineáris keveréke: $x \cdot (1-a) + y \cdot a$;
- `float` **`length`**`(genType x)`: az `x` vektor hossza: $\sqrt{\mathrm{x}[0]^2 + \mathrm{x}[1]^2 + \ldots}$;
- `float` **`dot`**`(genType x, genType y)`: két vektor skaláris szorzata: $\mathrm{x}[0] \cdot \mathrm{y}[0] + \mathrm{x}[1] \cdot \mathrm{y}[1] + \ldots$;
- `vec3` **`cross`**`(vec3 x, vec3 y)`: két 3D vektor vektoriális szorzata: $\begin{bmatrix} \mathrm{x}[1] \cdot \mathrm{y}[2] - \mathrm{y}[1] \cdot \mathrm{x}[2] \\ \mathrm{x}[2] \cdot \mathrm{y}[0] - \mathrm{y}[2] \cdot \mathrm{x}[0] \\ \mathrm{x}[0] \cdot \mathrm{y}[1] - \mathrm{y}[0] \cdot \mathrm{x}[1] \end{bmatrix}$;
- `genType` **`normalize`**`(genType x)`: normalizálja a vektort, egy azonos irányú, de 1-es hosszúságú vektort térít vissza;

A következőkben a http://www.lcg.ufrj.br/Cursos/GPUProg/GLSLfirst: *Starting with GLSL and OpenGL* alapján egy egyszerű példát mutatunk be a GLSL alkalmazására.

A feladat: rajzoljunk ki OpenGL-ben egy fehér négyzetet, majd vertex-shadert használva forgassuk el, valamint pixel-shadert használva színezzük sárgára!



1. ábra. *Fehér négyzet kirajzolása OpenGL-ben*

Az OpenGL program a fehér négyzet kirajzolására egyszerű:

```
1.  #include <iostream>
2.  #include <math.h>
3.  #include <stdio.h>
4.  #include <GL/glut.h>
5.
6.  void init()
7.  {
8.    glClearColor(0.75, 0.75, 0.75, 0.0);
9.  }
10.
11. void render()
12. {
13.   glClear(GL_COLOR_BUFFER_BIT |
14.           GL_DEPTH_BUFFER_BIT);
15.   glLoadIdentity();
16.   gluLookAt(0.0, 0.0, 3.0, 0.0, 0.0,
17.             0.0, 0.0, 1.0, 0.0);
18.   glBegin(GL_QUADS); // a négyzet kirajzolása
19.     glVertex3f(-0.5, -0.5, 0.0);
20.     glVertex3f(0.5, -0.5, 0.0);
21.     glVertex3f(0.5, 0.5, 0.0);
22.     glVertex3f(-0.5, 0.5, 0.0);
23.   glEnd();
24.   glutSwapBuffers( );
25. }
26.
27. void reshape(int w, int h)
28. {
29.   glViewport(0, 0, w, h);
30.   glMatrixMode(GL_PROJECTION);
31.   glLoadIdentity();
```

```
32.   if (h == 0) h = 1;
33.   gluPerspective(45, (float)w/(float)h,
34.                  0.1, 100.0);
35.   glMatrixMode(GL_MODELVIEW);
36.   glLoadIdentity();
37. }
38.
39. void keyboard(unsigned char key, int x, int y)
40. {
41.   switch (key)
42.   {
43.     case 27:
44.       exit(0);
45.       break;
46.     default:
47.       break;
48.   }
49. }
50.
51. int main(int argc, char** argv)
52. {
53.   glutInit(&argc, argv);
54.   glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE |
55.                       GLUT_DEPTH);
56.   glutCreateWindow("GLSL");
57.   init();
58.   glutDisplayFunc(render);
59.   glutReshapeFunc(reshape);
60.   glutKeyboardFunc(keyboard);
61.   glutMainLoop();
62.   return 0;
63. }
```

Az elforgatás és a színezés megvalósítására GLSL kódrészleteket használunk, ezért az OpenGL-t fel kell készíteni az árnyaló nyelv felismerésére, a shader programok fordítására, futtatására. Ehhez a 2006-ban Ben Woodhouse által kifejlesztett *GLee*-t (*OpenGL Easy Extension Library*) használjuk, amely letölthető a *http://elf-stone.com/glee.php* honlapról.

Először is, a `#include <GL/glut.h>` elé írjuk be:

```
1. #include "GLee.h"
```

Ezután (az *include*-ok után) globális változókként deklaráljuk a shader-programok azonosítóit, valamint string konstansokként deklaráljuk a GLSL programokat:

```
1.  GLuint program_object;   // a GLSL program azonosítója
2.  GLuint vertex_shader;    // a vertex-shader azonosítója
3.  GLuint fragment_shader;  // a pixel-shader azonosítója
4.
5.  // a vertex-shader forráskódja, 45°-os szögben elforgat egy
    vertexet
6.  static const char *vertex_source =
7.  {
8.   "void main()"
9.   "{"
10.  "  float PI = 3.14159265358979323846264;"
11.  "  float angle = 45.0;"
12.  "  float rad_angle = angle*PI/180.0;"
13.  "  vec4 a = gl_Vertex;"
14.  "  vec4 b = a;"
15.  "  b.x = a.x*cos(rad_angle) –
    a.y*sin(rad_angle);"
16.  "  b.y = a.y*cos(rad_angle) +
    a.x*sin(rad_angle);"
17.  "gl_Position = gl_ModelViewProjectionMatrix*b;"
18.  "}"
19. };
20.
21. // a pixel-shader forráskódja, sárgára színezi a pixelt
22. static const char *fragment_source =
23. {
24.  "void main(void)"
25.  "{"
26.  "  gl_FragColor = vec4(1.0, 1.0, 0.0, 1.0);"
27.  "}"
28. };
```

A shader-programokról szóló esetleges fordítási információkat a következő függvénnyel írhatjuk ki:

```
1.  static void printProgramInfoLog(GLuint obj)
2.  {
3.    GLint infologLength = 0, charsWritten = 0;
4.    glGetProgramiv(obj, GL_INFO_LOG_LENGTH,
5.                   &infologLength);
6.    if(infologLength > 2)
7.    {
8.      GLchar* infoLog = new GLchar[infologLength];
9.      glGetProgramInfoLog(obj, infologLength,
```

```
10.                         &charsWritten, infoLog);
11.     std::cerr << infoLog << std::endl;
12.     delete infoLog;
13.   }
14. }
```

Az `init` eljárást az aktuális törlőszín definiálása után a következőkkel bővítjük ki:

```
1.  void init()
2.  {
3.    glClearColor(0.75, 0.75, 0.75, 0.0);
4.
5.    // létrehozzuk a program objektumot
6.    program_object  = glCreateProgram();
7.    // létrehozzuk a vertex-shadert
8.    vertex_shader =
    glCreateShader(GL_VERTEX_SHADER);
9.    // létrehozzuk a pixel-shadert
10.   fragment_shader = glCreateShader(
11.                     GL_FRAGMENT_SHADER);
12.   printProgramInfoLog(program_object);
13.
14.   // hozzárendeljük a vertex-shader forráskódot
15.   glShaderSource(vertex_shader, 1,
16.                  &vertex_source, NULL);
17.   // hozzárendeljük a pixel-shader forráskódot
18.   glShaderSource(fragment_shader, 1,
19.                  &fragment_source, NULL);
20.   printProgramInfoLog(program_object);
21.
22.   // lefordítjuk a vertex-shadert és hozzárendeljük
23.   //a program objektumhoz
24.   glCompileShader(vertex_shader);
25.   glAttachShader(program_object, vertex_shader);
26.   printProgramInfoLog(program_object);
27.
28.   // lefordítjuk a pixel-shadert és hozzárendeljük
29.   //a program objektumhoz
30.   glCompileShader(fragment_shader);
31.   glAttachShader(program_object,
    fragment_shader);
32.   printProgramInfoLog(program_object);
33.
```

```
34.   // összeszerkesztjük a teljes GLSL programot
35.   glLinkProgram(program_object);
36.   printProgramInfoLog(program_object);
37.
38.   // minden hibát leellenőrzünk
39.   GLint prog_link_success;
40.   glGetObjectParameterivARB(program_object,
41.   GL_OBJECT_LINK_STATUS_ARB, &prog_link_success);
42.   if(!prog_link_success)
43.   {
44.     fprintf(stderr, "Hiba a szerkesztésnél\n");
45.     exit(1);
46.   }
47. }
```

Most már megvan a teljesen lefordított, összeszerkesztett, működő GLSL programunk, nem maradt más hátra, mint használjuk ezt.

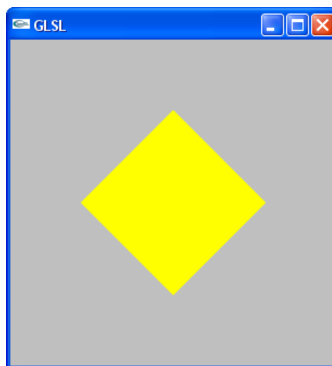
A `render` függvénybe, a négyzet effektív kirajzolása elé (`glBegin (GL_QUADS)`) írjuk be:

```
1.    glUseProgram(program_object);
```

utána pedig (`glEnd()` után) szüntessük meg a GLSL program használatát:

```
1.    glUseProgram(0);
```

A kód többi része változatlan marad.



2. ábra. *A négyzet elforgatása és kiszínezése GLSL-t használva*

Kovács Lehel