Dr. Mileff Péter

GRAFIKA PROGRAMOZÁSA

3D GRAFIKA II. RÉSZ

PROGRAMOZHATÓ GRAFIKUS CSŐVEZETÉK

Miskolci Egyetem
Általános Informatikai Tanszék

PROGRAMOZHATÓ GRAFIKUS CSŐVEZETÉK (ALAPOK)…

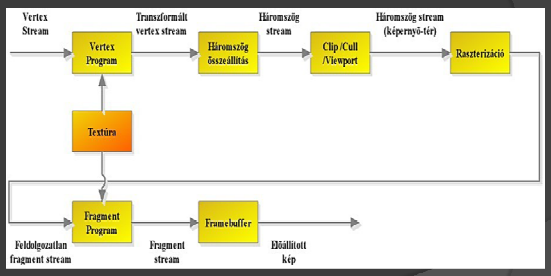
Grafikus csővezeték
(Általános áttekintés)

- **A grafikus csővezeték(graphics pipeline):**
  - feldolgozási szakaszok egy elméleti modellje
  - ezen keresztül küldjük a grafikai adatokat, hogy megkapjuk a várt eredményt
    - Végeredmény: pixelek összessége a képernyőn
- **Megvalósításai:**
  - 1. Szoftveres: CPU segítségével
    - Korai korszak
  - 2. Hardveres: a GPU-ban
    - Mai korszak
  - 3. A kettő kombinációjaként
    - Átmeneti időszak (mai korszak)

Grafikus csővezeték
(Általános áttekintés)

- **A feldolgozási szakaszok:**
  - nem mások, mint adat átalakítási folyamatok
  - térbeli koordinátákkal kezdünk és rasztergrafikus képet kapunk eredményképpen.
  - A valóságban összetett folyamatok együttese
- A csővezeték modell folyamatosan változik:
  - Az adott technológiai fejlődés indukálja
    - Pl. a GPU-k fejlődésének iránya nagyban befolyásolja, hogy mely feladatokat kell szétbontani, melyeket pedig összevonni
- **Kezdetben:**
  - csakis szoftveres (fix funkcionalitású, **fixed function pipeline**) futószalag támogatás volt a jellemző - CPU
  - A GPU-k megjelenésével és fejlődésével ez megváltozott

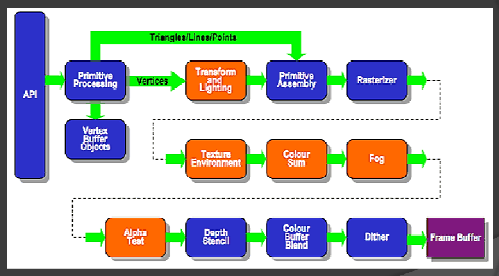
Grafikus csővezeték
(Általános áttekintés)



A csővezeték általános modellje

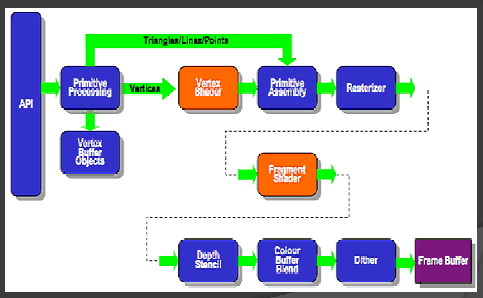
5

OpenGL ES 1.0
fixed pipeline



6

OpenGL ES 2.0
programozható csővezeték



7

Grafikus csővezeték
(Általános áttekintés)

- **A fix funkciós csővezeték:**
  - nagy hátránya az volt, hogy a GPU-val csakis rögzített művelet végrehajtása volt lehetőség.
- **A GPU-k egyre gyorsabbak lettek:**
  - nőtt az igény arra, hogy a futószalagot valamilyen módon programozni is lehessen
  - A programozhatóság mértéke kezdetben minimális volt
  - A lehetőségek az évek során folyamatosan bővültek:
    - Kialakultak az **árnyaló nyelvek**
    - a csővezeték így egyre inkább programozhatóvá vált.

8

Grafikus csővezeték
(Általános áttekintés)

- A folyamatos fejlődés miatt a csővezetékek is átalakultak
  - A teljes programozhatóság irányába
- **A döntés a miénk:**
  - A készítendő grafikus alkalmazás esetén ki kell választani, hogy mely grafikus API-ra akarunk építeni
  - és annak milyen verzióit szeretnénk támogatni.
  - A csővezeték és a funkciók hardveres támogatása ugyanis ettől függ.

Grafikus csővezeték
(Általános áttekintés)

Példa 1: szeretnénk *Vertex Buffer Object* (VBO) típusú adattárolást alkalmazni

- **Asztali gépek:** az OpenGL 1.5-től magasabb verziószámot támogató videokártyák esetén tehetjük meg.
- **Mobileszközök:** az OpenGL ES 1.1-től magasabb chipek támogatják csak.

Példa 2:

- Vertex és fragment programokat csak az OpenGL ES 2.0-tól tudunk futtatni.

OPENGL 3.0 ÚJÍTÁSAI…

OpenGL 3.0 újításai

- A hardverek erőteljes fejlődése miatt igény merült fel az API átalakítására
  - olyan módú átalakításra, hogy a hardver változásokat jobban tudja követni
  - közelebb kerülni a hardverhez.
    - **Getting „back to the bare metal for performance”** - Michael Gold az Nvidia részéről.
- **Megoldás:**
  - El kell tüntetni azokat a funkciókat, amelyeket már alig használnak,
  - csak a kompatibilitás rétegben meghagyni.

OpenGL 3.0 újításai

- **A 3.0-tól szakítottak a fix funkciós csővezetékkel**
- Mit jelent ez?
  - a fix futószalag bizonyos részeit nem használhatjuk
  - ezen részek implementációját a programozónak kell elvégeznie az **árnyaló nyelv** segítségével.
- Új **objektum modell** került az OpenGL 3.0 középpontjába.
- **Két fő oka:**
  - az egész rendszer és a driver teljesítményének növelése,
  - a kliensoldali objektum kezelés könnyítése és robusztussá tétele.

13

OpenGL 3.0 újításai

- **A teljesítmény növelésének eszközei:**
  - egyazon objektumhoz tartozó összes tulajdonságot magában foglalja
    - Pl. VAO – Vertex Array Object – Opengl 3.2
  - ezeket atomi egységként adja át az API-nak.
  - Így biztosítható az átadott objektumok teljessége,
    - elkerülve a banális, ám nehezen felderíthető programozói hibákat.
- Az általános cél:
  - csökkentsük a driver által keltett többletterheléseket,
  - egységesítve és egyszerűsítve az objektumok kezelését a programozó szemszögéből.

14

OpenGL 3.0 újításai

- Néhány példa az elhagyott funkciókból:
  - *glBegin/glEnd,*
  - *glTranslate,*
  - *glRotate,*
  - *glIdentity,*
  - *glColor3f,*
  - stb.
  - Ezek pontos leírását az OpenGL aktuális specifikációjából nyerhetjük ki.

15

PROGRAMOZHATÓ PÁRHUZAMOS PROCESSZOROK...

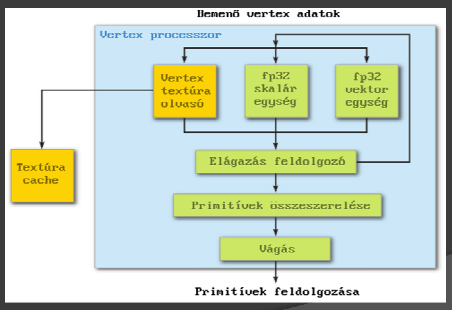
18

Programozható párhuzamos processzorok

- Egy GPU kétféle programozható processzorral rendelkezik
  - vertex és fragment processzorok
- **Vertex processzor:**
  - a vertex streameken hajtják végre a műveleteiket
    - Pl. eltolás, forgatás, stb
  - A processzor egy vertex programot hajt végre,
    - vertex shader – csúcspont árnyaló
  - Egy fragment stream outputot állít elő,
  - amit a fragment processzor egy fragment program segítségével dolgoz fel,
    - fragment shader – pixel árnyaló
  - és állítja elő az egyes pixelek végleges színét.

17

Vertex processzor általános modellje



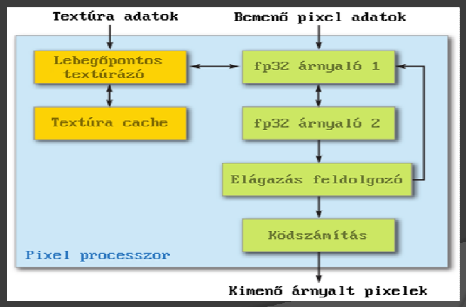
18

Programozható párhuzamos processzorok

- **Fragment processzor (Pixel processzor):**
  - olyan programozható egység, amely a textúrázáshoz előkészített képtöredékek feldolgozására szolgál.
  - Általánosan elterjedt, hogy több fragment processzor foglal helyet a GPU-ban.
  - **Tipikusan feladata:**
    - a geometriailag előkészített potenciális képpontok színének manipulálása,
    - minden pixelre végrehajt egy felhasználói programot.

19

Fregment processzor általános modellje



20

MAGAS SZINTŰ ÁRNYALÓ NYELVEK…

21

Árnyaló nyelvek

- Segítségükkel programozhatjuk a GPU vertex és fragment processzorait
- Lehetővé a fix funkciós csővezeték „kihagyását” a programozó számára.
- **Nyelvek jellemzői:**
  - Kezdetben nem voltak magas szintű nyelvek,
  - csakis Assembly nyelvel lehetett shader programot írni.
  - A mai kialakult nyelvek sokévnyi fejlődés eredményei,
  - három fő meghatározó irányból fejlődtek ki:
    1. Általános programozási nyelvek
    2. A grafikus interfész nyelvek
    3. Maguk az árnyalók

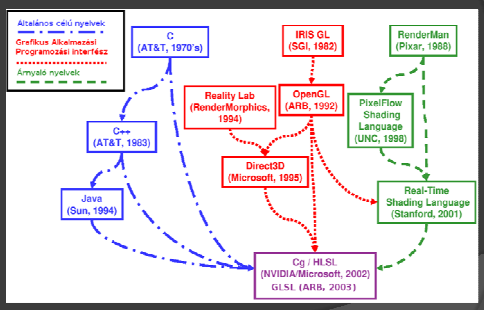
22

Árnyaló nyelvek

- A programok közös nyelvi alapja a C nyelv
  - A szintaktika és szemantika szempontjából
  - a mai legfőbb árnyaló nyelvek is ezen alapulnak
- **Ismertebb nyelvek:**
  - CG (C for Graphics) – NVIDIA
    - Első GPU árnyaló nyelvek egyike
  - GLSL (GLslang) – OPENGL
  - HLSL (High Level Shading Language) – MICROSOFT
    - XBOX – XNA – DirectX

23

Árnyaló nyelvek fejlődése



24

OpenGL Assembly Language
(Példa pregment árnyalóra)

```
!!ARBfp1.0
TEMP color;
MUL color, fragment.texcoord[0].y, 2.0;
ADD color, 1.0, -color;
ABS color, color;
ADD result.color, 1.0, -color;
MOV result.color.a, 1.0;
END
```

25

GLSL Language
(Példa pregment árnyalóra)

```
#version 120
void main(void)
{
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

26

GLSL (RÖVID) ÁTTEKINTÉS…

27

GLSL nyelv

- OpenGL Shading Language (gyakran **glslang**)
- Az OpenGL ARB szervezet hozta létre az OpenGL 1.4 kiterjesztéseként
- Később az OpenGL 2.0-nél már teljes mértékben a szabvány részévé vált.
- **Jellemzője:**
  - magas szintű árnyaló nyelv,
  - a C nyelv szintaktikáján alapszik
  - A fejlesztőknek lehetővé teszi, hogy közvetlen módon vezéreljék a grafikus csővezetéket
  - anélkül, hogy assembly-t vagy valamilyen hardver közeli nyelvet keljen használniuk.

28

GLSL nyelv

- **Jellemzője (folytatás):**
  - Platformfüggetlen, támogatja többek között a GNU/Linux, Windows és Mac OS X operációs rendszereket.
  - A GLSL-ben írt shaderek bármely grafikus kártyán használhatóak, amelyek támogatják a GLSL-t.
  - Minden grafikus kártya driver tartalmazza a GLSL fordítót,
    - így a grafikus kártya gyártók optimalizálhatják a fordító által előállított kódot a kártya architektúrájának megfelelően.
- A shader programok alapvetően az adat-párhuzamosságra épülnek,
  - a párhuzamosság módja driver implementáció függő.
    - Különbözően lehetnek optimalizálva
  - a párhuzamosan futó programok közti kommunikációt nem támogatják.

29

GLSL árnyalók alkalmazása

- A programok reprezentációja szöveges nyelven történik.
  - C nyelv szintaktikáján alapul
- A kódolás során csúcs- és pixelárnyalókat kell írnunk
- Akár külön fájlban, akár a programkódban karakterláncként tárolva.
- **Általában azonban az árnyalókat külön fájlban helyezik el**
  - Például: akarmi.vert, akarmi.frag.
  - A fájl kiterjesztés nem releváns
    - tartalma fogja eldönteni, hogy melyik típusú árnyalóról van szó.
    - Enne ellenére célszerű a kiterjesztésben is jelezni ezt

30

GLSL árnyalók alkalmazása

- Az árnyalók forrásszövege:
  - az UTF-8 kódolású Unicode karakterek egy részhalmazát tartalmazhatják.
  - A forráskód ezen jelekből alkotott string
    - több sorban helyezkedhet el.
- Tartalmazhatnak direktívákat a fordítónak, változó deklarációkat, stb a fájl elején.
- A változókat különböző minősítőkkel is elláthatjuk
  - Pl. tárolás, memória, precizitás, layout
  - jelezve annak egyéb tulajdonságait
- Minden árnyalónak rendelkezni kell egy **main** függvénnyel,
  - jelzi a program belépési pontját

31

GLSL árnyalók alkalmazása

Árnyalók használata:

- Az árnyaló fájlok betöltését a felhasználónak saját kezűleg kell elvégeznie
- Az OpenGL erre közvetlen lehetőséget nem biztosít.
- **Jelentése:**
  - a fájlok memóriába való olvasását kell manuálisan megtenni
  - az OpenGL innen már biztosít lehetőséget a tényleges shader objektum létrehozására.

32

GLSL MINTAPÉLDA…

GLSL Mintapélda

Mire lesz szükség?

- 1. Egy egyszerű shader osztály
  - Képes eltárolni a betöltött shaderek azonosítóit és az objektumokat.
- 2. Árnyaló fájlok betöltése a hdd-ről
- 3. OpenGL objektumok létrehozása a megfelelő árnyalóknak
  - hibakezeléssel
- 4. Árnyalók alkalmazása egy példában
  - Árnyalók aktiválása és deaktiválása

Általános árnyaló osztály

```
/// Simple Shader class
class CShader {
   /** Global ID for Shader */
   unsigned int m_iShaderID;

   /** Fragment Shader name */
   string m_pFragmentShader;

   /** Vertex Shader name */
   string m_pVertexShader;

   /** OpenGL Shader handler */
   GLuint m_pShaderHandler;
public:
...
}
```

Shader osztály (C++)

Általános árnyaló osztály

Mit célszerű tárolni?

- a grafikus motor számára az osztályt egyedileg azonosító id-t (*m_iShaderID*),
- a shaderek fájlneveit,
- az OpenGL belső shader azonosítóját,
  - egy előjel nélküli szám.

Általános árnyaló osztály
(árnyalók betöltése)

```
bool CShader::setShaders(string fragment_shader, string vertex_shader){
   GLuint v,f;
   /** Loading vertex shader source file */
   m_pVertexShader = LoadShaderSource(vertex_shader.c_str());
    if (m_pVertexShader.c_str() == NULL) {
      cout << "\nError: Cannot Load vertex shader: " << vertex_shader.c_str() << endl;
      return false;
   }

   /** Loading fragment shader source file */
   m_pFragmentShader = LoadShaderSource(fragment_shader.c_str());
   if (m_pFragmentShader.c_str() == NULL){
      cout << "\nError: Cannot Load fragment shader: " << fragment_shader.c_str() <<
endl;
      return false;
   }
```


Általános árnyaló osztály
(árnyalók betöltése)

```
/** If everything was good, we initialize shader */
  const char *vv = m_pVertexShader.c_str();
  const char *ff = m_pFragmentShader.c_str();

  v = glCreateShader(GL_VERTEX_SHADER);
  f = glCreateShader(GL_FRAGMENT_SHADER);
```

- A shader források betöltése után az OpenGL shader objektumok létrehozását a *glCreateShader* függvénnyel tehetjük meg.
- A függvény paramétere a létrehozandó shader objektum típusa (jelen példában vertex és fragment).

Általános árnyaló osztály
(árnyalók betöltése)

```
glShaderSource(v, 1, &vv, NULL);
glShaderSource(f, 1, &ff, NULL);
```

- A betöltött szövegek shader objektumokhoz való társítása történik a **glShaderSource** függvény segítségével.
- **Paraméterek:**
  - A shader objektum, amelyhez a forrást társítani szeretnénk,
  - A társítani kívánt szövegforrások száma.
  - Ezt követi a forrás szöveg,
  - Majd egy NULL érték,
    - azt jelenti, hogy az OpenGL a forrás szöveget NULL karakterrel zárt.

Általános árnyaló osztály
(árnyalók betöltése)

```
glCompileShader(v);
```

- A fenti sorral pedig megtörténik a fordítása a kódnak. Majd pedig a hibakezelés következik:

```
GLint status;
glGetShaderiv(v, GL_COMPILE_STATUS, &status);
if (status == GL_FALSE) {
    GLint infoLogLength;
    glGetShaderiv(v, GL_INFO_LOG_LENGTH, &infoLogLength);
    GLchar *strInfoLog = new GLchar[infoLogLength + 1];
    glGetShaderInfoLog(v, infoLogLength, NULL, strInfoLog);
     printf("\nError: Compile failure in %s shader: %s ", vertex_shader.c_str(), strInfoLog);
     delete[] strInfoLog;
    return false;
}
```

Általános árnyaló osztály
(árnyalók betöltése)

- **Ugyanez a fregment shader -re is elvégezve:**

```
glCompileShader(f);
glGetShaderiv(f, GL_COMPILE_STATUS, &status);

if (status == GL_FALSE){
   GLint infoLogLength;
   glGetShaderiv(f, GL_INFO_LOG_LENGTH, &infoLogLength);

   GLchar *strInfoLog = new GLchar[infoLogLength + 1];
   glGetShaderInfoLog(f, infoLogLength, NULL, strInfoLog);

   printf("\nError: Compile failure in %s shader: %s",fragment_shader.c_str() ,strInfoLog);

   delete[] strInfoLog;
   return false;
}
```

41

Általános árnyaló osztály
(árnyalók betöltése)

- **A hibakezelés után már csak a shader program létrehozása maradt hátra:**

```
m_pShaderHandler = glCreateProgram();
glAttachShader(m_pShaderHandler,v);
glAttachShader(m_pShaderHandler,f);
glLinkProgram(m_pShaderHandler);
```

- **Első lépésként egy üres program objektumot hozunk létre,**
- **majd a korábban létrehozott shader objektumokat társítjuk a programhoz.**

42

Általános árnyaló osztály
(árnyalók betöltése)

- **Végezetül a program linkelése következik és státuszának ellenőrzése:**

```
glGetProgramiv(m_pShaderHandler, GL_LINK_STATUS, &status);
if (status == GL_FALSE){
   GLint infoLogLength;
   glGetProgramiv(m_pShaderHandler, GL_INFO_LOG_LENGTH, &infoLogLength);
    GLchar *strInfoLog = new GLchar[infoLogLength + 1];
   glGetProgramInfoLog(m_pShaderHandler, infoLogLength, NULL, strInfoLog);
    printf("\nError: Linker failure in %s shader: %s", fragment_shader.c_str(), strInfoLog);
    delete[] strInfoLog;
   return false;
}
 return true;
}
```

43

GLSL MINTAPÉLDA…
(ÁRNYALÓK ALKALMAZÁSA)

Mintapélda:

2D textúrázott négyzet kirajzolása
Textúra szín (RGB) állítása árnyaló segítségével
Régi glBegin/glEnd páros alkalmazása

44

Árnyalók alkalmazása

- Az alkalmazás során három fontos lépést kell követni:
  - **Árnyaló aktiválása, paraméterek átadása:** kiválasztjuk a megfelelő árnyalót és átadjuk a szükséges paramétereket
  - **Objektumok rajzolása:** kirajzolunk minden olyan objektumot, amelyre érvényes lesz az árnyaló hatása
  - **Árnyaló kikapcsolása:** amennyiben az árnyaló már nem szükséges, ki kell kapcsolni.

45

Árnyalók alkalmazása

- **Az árnyalók kikapcsolása kulcsfontosságú:**
  - ennek hiányában a később kirajzolt elemek megjelenítésében ez gondot okozhat.
  - Tipikusan ilyen hiba:
    - a kirajzolt objektum(ok) után a grafikus user interface más színű lesz.
    - Mivel a user interface-t mindig legkésőbb kell kirajzolni, mert az kerül legfölülre, így általában itt szokott jelentkezni a kérdéses probléma.
- **Mi történik akkor ha az árnyaló hibás?**
  - A vázolt betöltő jelzi a fordítási hibát, de ennek ellenére a szoftver futni fog.
  - Legtöbb esetben a megjelenítés ilyenkor „visszakapcsol” a grafikus API beépített funkcióira.
  - Például fények esetében a fix funkciós árnyalásra.

46

Árnyalók alkalmazása

```
void DrawQuad() {

glActiveTexture(GL_TEXTURE0);                 // Bind to Texture Unit 0
glBindTexture(GL_TEXTURE_2D, textureID);  // Bind texture
```

- A GPU első textúrázó egységéhez rendeljük a korábban betöltött textúra azonosítóját.

47

Árnyalók alkalmazása

- `//glColor4f(red,green,blue,alpha);  // Fixed pipeline!!!`
- A fenti sor jelentené a fix funkciós csővezeték használatát.
- Aktiválva és a shader kódot kikommentezve ugyanazt az eredményt kell kapjuk.
  - Egyszerre nincs értembe használni, mert úgyis a shader program lesz az elsődleges.

```
// Using shaders
Gluint sHandler = shader->m_pShaderHandler;
glUseProgram(sHandler);
glUniform4f(glGetUniformLocation(sHandler, "color"),red,green,blue,alpha);
```

48

Árnyalók alkalmazása

- Az árnyalót ***glUseProgram*** segítségével tudjuk használatba venni
  - megadva a használni kívánt shader azonosítóját.
- Az utasítás jelzi az OpenGL-nek
  - hogy a következő renderelési utasításokra az árnyaló lesz alkalmazva a fix funkciós csővezeték helyett.
- **A textúra színeinek módosítása:**
  - az árnyalónak átadjuk az aktuális RGBA változók értékét a ***glUniform4f*** függvény segítségével.
  - A programsor az aktuális árnyalóban megkeresi a „color” változó „helyét”, és áttölti a megadott színértékeket.

49

Árnyalók alkalmazása

- Ezt követően kirajzoljuk a négyzetet a megszokott módon:

```
  glBegin(GL_QUADS);
  glTexCoord2f(0, 1);
  glVertex2f(0, 256);
  glTexCoord2f(1, 1);
  glVertex2f(256, 256);
  glTexCoord2f(1, 0);
  glVertex2f(256, 0);
  glTexCoord2f(0, 0);
  glVertex2f(0, 0);
  glEnd();

  // Disable shader effect
 glUseProgram(0);
}
```

50

GLSL MINTAPÉLDA…

(ÁRNYALÓK BEMUTATÁSA)

51

Árnyalók alkalmazása
(Vertex árnyaló)

- **Vertex árnyaló:** feladata nagyon egyszerű
- Szeretnénk a textúra koordinátákat elérni a pixel árnyalóban is:
  - így ezt a *v_texCoord* változóban tároljuk lekérve a 0-ás textúrázó egységtől.
  - Azért a nullástól, mert a kirajzoláskor ehhez az egységhez kötöttük a textúrát
  - A változó varying minősítéssel van ellátva.
- Az *ftransform()* függvény szerepe pedig a vertex-ek transzformálása a megfelelő 3D koordinátákra.
  - Lényegében mátrixok szorzását rejti el, a következő kód azonos vele:
  - *gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;*
  - *Vagy: gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix * gl_Vertex;*

```
varying vec2 v_texCoord;
void main(){
    v_texCoord = gl_MultiTexCoord0.xy;
    gl_Position = ftransform();  // Deprecated from GLSL 1.40
}
```

52

Árnyalók alkalmazása
(Pixel árnyaló)

- **Pixel árnyaló:** feladata nagyon egyszerű
  - feladata az, hogy a textúra képpontjait és a beállított színt kombinálja egymással.
  - Ehhez nem kell mást tenni, mint a összeszorozni a két értéket.
  - A kódban a texcolor változó fogja tárolni az aktuális textúra pixel színét.
  - A fájl elején uniform minősítéssel ellátott color változó a shader-nek a felhasználói programból átadott szín.
  - A végleges pixel színt a *gl_FragColor*-nak adva adhatjuk meg.

```
uniform sampler2D diffuseTex;
uniform vec4 color;
varying vec2 v_texCoord;

void main(){
    vec4 texcolor = texture2D(diffuseTex,v_texCoord);
    gl_FragColor = texcolor *color;
}
```

53

SHADER PROGRAMOK OPTIMALIZÁLÁSA…

54

Shader programok optimalizálása

- A magas teljesítmény érdekében az árnyalókkal a megfelelő módon kell bánni.
- **Az árnyalók gyakori váltása költséges a GPU számára!**
  - A túl sok árnyaló váltás megöli a grafikai rajzolás teljesítményét
- **Váltás:**
  - például az egyik objektum az X shader-tt alkalmazza a kirajzoláshoz, a másik pedig Y-t.
- Egy komplex grafikai alkalmazás (pl. játék) esetén ezeket célszerű optimalizálni

55

Shader programok optimalizálása

- Az optimalizálás alapja:
  - ha az előzőleg használt shader megegyezik azzal az shader-rel, amivel most akarok rajzolni, akkor nem kell a shader-eket váltani.
- **Megvalósítása:**
  - egy **logikai rajzolási csoport**ba kell gyűjteni azon objektumokat, amelyek azonos árnyalót igényelnek.
  - A rajzolás folyamatát az egyes listákra külön végezzük el:
    - 1. a lista elején aktiváljuk az árnyalót (*glUseProgram*),
    - 2. majd kirajzoljuk az összes objektumot,
    - 3. végül pedig lezárjuk az árnyalót.
- Az így kialakított rendszerben az árnyaló váltások száma minimális lesz

56

GAME OVER