Hardware pro počítačovou grafiku NPGR019

Novinky v OpenGL 4

Jan Horáček

http://cgg.mff.cuni.cz/
MFF UK Praha

2012

Obsah

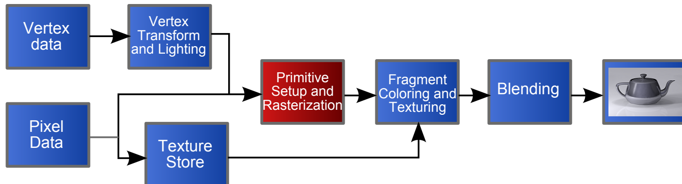
1. Úvod
2. Shadery
3. Texturování v OpenGL 4.0
4. Geometrické objekty v OpenGL
5. Přehled dalších novinek
6. Literatura

Předpoklady

- umíte programovat v OpenGL
  - alespoň základy - vykreslení objektu, texturování
- umíte vertex shadery
  - jak by se napsala fixní pipeline pomocí *vertex shaderu*
- umíte fragment shadery
  - co vše zhruba *fragment shader* zvládne

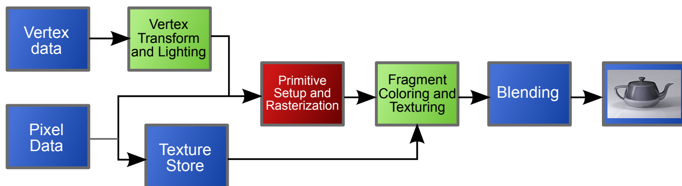
Historie OpenGL ve zkratce

- OpenGL 1.0 vydáno v roce 1994
- *fixed-function* pipeline
  - tzn. veškeré zpracování pevně dané implementací
- pipeline se vyvíjela a zůstala hlavní součástí OpenGL až do verze OpenGL 2.0 (2004)



Počátky programovatelné pipeline

- OpenGL 2.0 oficiálně přidalo programovatelné **shadery**
    - **vertex shader** nahrazoval fixed-function transformace a světlo
    - **fragment shader** měnil přiřazování barvy fragmentu
    - fixed-function pipeline ovšem zůstala a byla oficiálně k dispozici
- téměř beze změny až do OpenGL 3.1 (2009)



Změna principu vylepšování OpenGL

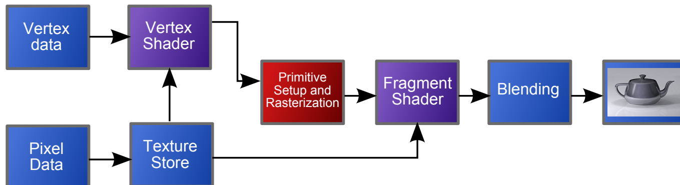
- OpenGL 3.0 zavedl tzn. **deprecation model**
  - metoda pro **odstraňování** zastaralých částí OpenGL
- změna i v tom, jak se používají OpenGL *kontexty*
  - OpenGL **context** je datová struktura v driveru, která ukládá stavová data (textury, shadery, atd.)

Typ kontextu	**Popis**
Full	obsahuje vše (včetně *deprecated*)
Forward Compatible	obsahuje pouze *non-deprecated*, tzn. je *podobný* příští verzi OpenGL

- typ kontextu se vybírá při vytváření

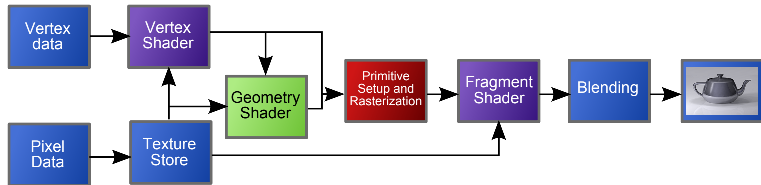
Exkluzivně programovatelné pipeline

- OpenGL 3.1 **odstranilo** fixní pipeline
  - programy musely použít shadery
  - rozšíření `GL_ARB_compatibility` zpřístupňovalo starou funkcionalitu
- skoro všechna data **na GPU**
- veškerá **vertex** data posílána přes **buffer** objekty



Více programovatelnosti

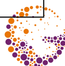
- OpenGL 3.2 (2009)přidalo novou shader stage - **geometry shader**



Kontextové profily

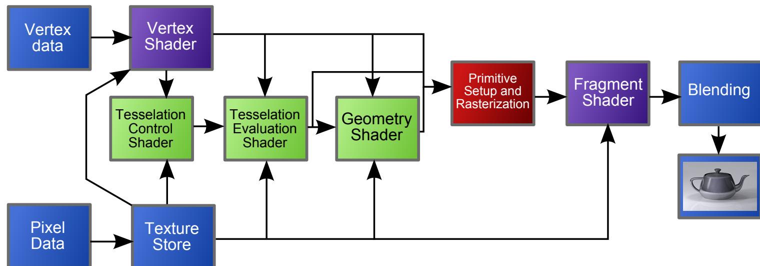
- OpenGL 3.2 také zavedlo tzv. **context profiles**
- profily kontrolují, které funkce OpenGL jsou k dispozici
  - podobně jako `GL_ARB_compatibility`, jen lépe zakomponované
- momentálně dva typy profilů: **core** a **compatible**

Typ kontextu	**Profil**	**Popis**
Full	core	vše z aktuální verze
	compatible	vše z celé historie OpenGL
Forward Compatible	core	vše *non-deprecated*
	compatible	není podporováno



Nejnovější pipeline

- OpenGL 4.0 (březen 2010) přidalo další nové shader stage - **tesselation control** a **tesselation evaluation** shaders
    - OpenGL 4.1 (červenec 2010) nepřidalo žádnou novou stage



Moderní programování ve zkratce

1. vytvoř *shader* programy
2. vytvoř a načti buffer objekty pro ukládání dat
3. propoj umístění dat s proměnnými v shaderech
4. vykresli scénu

Obsah

1 Úvod

2 Shadery

3 Texturování v OpenGL 4.0

4 Geometrické objekty v OpenGL

5 Přehled dalších novinek

6 Literatura

Shadery v OpenGL 4.0

- OpenGL 4.0 aplikace **musí** používat shadery
- verze Shader language pro OpenGL 4.0 je 400
  - `#version 400 [core|compatibility]`

Typ shaderu	**Vstupní data**
GL_VERTEX_SHADER	jednotlivé vertexy
GL_FRAGMENT_SHADER	jednotlivé fragmenty
GL_TESSELATION_CONTROL_SHADER	vstupní vertexy každé *patch*
GL_TESSELATION_EVALUATION_SHADER	výstupní vertexy každé *patch*; teselační koordináty
GL_GEOMETRY_SHADER	vstupní vertexy primitiv

Nahrání shaderů do OpenGL

- shadery musí být zkompilované a slinkované aby vytvořily spustitelný shader program
- OpenGL **driver** poskytuje kompilátor a linker
- program musí obsahovat
  - vertex a fragment shadery
  - ostatní shadery jsou volitelné

Vytvoř program	`glCreateProgram()`
Vytvoř shader	`glCreateShader()`
Načti zdrojový kód shaderu	`glShaderSource()`
Zkompiluj shader	`glCompileShader()`
Připoj shader k programu	`glAttachShader()`
Linkování programu	`glLinkProgram()`
Použij program	`glUseProgram()`

Vytvoření shader programu

- podobné kompilování **C** programu
  - editace, kompilování a slinkování
- provádí se následující kroky
  1. vytvoření a kompilace shader objektů
  2. připojení shader objektů k programu
  3. slinkování objektů do spustitelného programu

Kompilování shaderů - část 1

vytvoř a zkompiluj shader

```
GLuint shader = glCreateShader( shaderType );
const char* str = "void main() {...}";
glShaderSource( shader, 1, &str, NULL );
glCompileShader( shader );
```

- *shaderType* je jeden z dostupných typů shaderu

Kompilování shaderů - část 2

zkontroluj, jestli je kompilace v pořádku

```
GLint compiled;
glGetShaderiv( shader, GL_COMPILE_STATUS, &compiled );
if( !compiled ) {
   GLint len;
   glGetShaderiv( shader, GL_INFO_LOG_LENGTH, &len );
   std::string msgs( ' ', len );
   glGetShaderInfoLog( shader, len, &len, &msgs[0]);
   std::cerr << msgs << std::endl;
   throw shader_compile_error;
}
```

Linkování shaderů - část 1

vytvoř prázdný *program object*

```
GLuint program = glCreateProgram();
```

propoj shader objekty s programem

```
glAttachShader( program, vertexShader );
glAttachShader( program, fragmentShader );
```

slinkuj program

```
glLinkProgram( program );
```

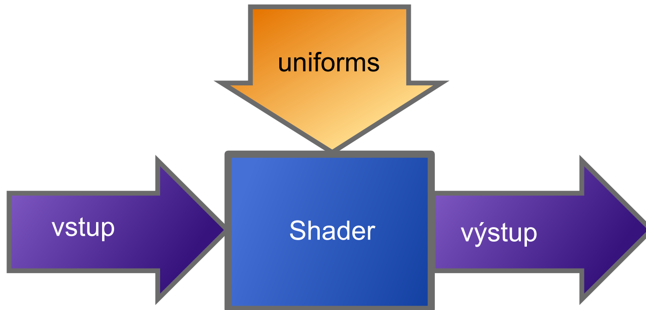
Linkování shaderů - část 2

zkontroluj, jestli je vše v pořádku

```
GLint linked;
glGetProgramiv( program, GL_LINK_STATUS, &linked );
if( !linked ) {
   GLint len;
   glGetProgramiv( program, GL_INFO_LOG_LENGTH, &len );
   std::string msgs( ' ', len );
   glGetProgramInfoLog( program, len, &len, &msgs[0]);
   std::cerr << msgs << std::endl;
   throw shader_link_error;
}
```

Spojování shaderů

- musíme specifikovat, jak putují data mezi shader stages v celém shader programu
- v angličtině označováno také jako *plumbing*



Základy spojování - vstup

- tok dat
  - do shaderových **in** proměnných
  - zpracování instrukcemi v shaderu
  - výstup do shaderových **out** proměnných

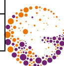
příklad

```
out vec4 color;      →   in vec4 color;
out vec2 texCoord;   →   in vec2 texCoord;
```

- shadery se odmítnout slinkovat, pokud není v pořádku pospojování
  - výstupy neodpovídají vstupům

Vstupy do shaderů

Shader stage	Zdroj dat	Vstupní data
vertex	kreslící příkazy	vertexové atributy pro jeden vertex
tesselation control	**out** proměnné předešlé stage	vstupní *patch* vertex atributy z vertex shaderu
tesselation evaluation	**out** proměnné předešlé stage	výstupní *patch* vertex atributy z tesselation control shaderu
geometry	**out** proměnné předešlé stage	vstupní vertexové atributy primitiva z vertex/tesselation evaluation shaderu
fragment	rasterizer	*varying* proměnné pro fragment



Uniform proměnné

- **uniform** proměnné jsou během volání vykreslení *konstantní*
- jejich hodnoty
  - mohou být změněny **pouze** mezi voláním vykreslování
  - jsou stejné pro všechny primitiva
- deklarují se pomocí klíčového slova `uniform`
- nastavení hodnoty
  - volání funkcí: `glUniform*()`, `glUniformMatrix*()`, `glProgramUniform*()`
  - uniform buffer object

před použitím se musí zjistit umístění

```
GLuint loc = glGetUniformLocation( program, "name" );
```

Vestavěné GLSL proměnné

- před OpenGL 3.1 - **mnoho vestavěných** proměnných
  - většinou se odkazovaly na něco ve *fixed-function pipeline*
- **skoro všechny** odstraněny (spolu s ff-pipeline) pomocí deprecation režimu
- několik nejdůležitějších zůstalo
- například `gl_Position`
  - implicitní výstup všech shaderů zpracovávajících vertexy
  - může být částí pole - to závisí na konkrétní shader stage

Jednoduchý vertex a fragment shader

Vertex shader

```
#version 400 core

layout(location = 0)
   in vec4 vPos;
layout(location = 1)
   in vec3 vColor;

out vec3 color;
uniform mat4 MVP;

void main()
{
   color = vColor;
   gl_Position = MVP * vPos;
}
```

Fragment shader

```
#version 400 core

in vec3 color;
layout(location = 0)
   out vec4 fragColor;

void main()
{
   fragColor = color;
}
```

Program Objects - problém?

- *program object* může nyní obsahovat až **pět** shaderů
- často bude rozdíl mezi dvěma programy pouze **jeden** shader
  - například jiný fragment shader
- toto způsobí hodně **duplicit** shaderů v grafické paměti
  - každý program obsahuje kopii **identického** shaderu

Oddělené shader programy a řetězce

- v OpenGL 4.0 je možné vytvořit **nezávislý** shader program pouze s **jednou** shader stage
- je třeba takový program **označit**, že ho chceme mít nezávislý
  - před zavoláním `glLinkProgram()`

nastavení parametru

```
glProgramParameteri( program, GL_PROGRAM_SEPARABLE, GL_TRUE );
```

- pomocná funkce `glCreateShaderProgram` toto dělá automaticky
- takto oddělené programy se mohou kombinovat v *shader pipeline*

Shader pipelines

- **Shader pipelines** jsou kolekce oddělených program objektů
- inicializace *shader pipeline*
  1. vygeneruj shader pipeline objekt
  2. nastav shader pipeline objekt jako aktivní
  3. připoj oddělené programové objekty do pipeline a nastav, které shader stage mají být použity

Shader pipelines - příklad vytvoření

vytvoření pipeline

```
// nejprve vytvoř několik shader objektů
GLuint vertPgm = glCreateShaderProgramv( GL_VERTEX_SHADER, 1, &src );
GLuint fragPgm1 = glCreateShaderProgramv( GL_FRAGMENT_SHADER, 1, &src );
GLuint fragPgm2 = glCreateShaderProgramv( GL_FRAGMENT_SHADER, 1, &src );

ShaderEntry shaders[] = {
   { GL_VERTEX_SHADER, "shader.vert" },
   { GL_TESS_CONTROL_SHADER, "shader.cont" },
   { GL_TESS_EVALUATION_SHADER, "shader.eval" },
   { GL_NONE, NULL }
};
GLuint multiPgm = LoadShaders ( shaders );
```

Shader pipelines - příklad vytvoření

vytvoření pipeline

```
// alokuj id pro shader pipeline
enum { VF, VTF, NumPipelines };
GLuint pipelines[NumPipelines];
glGenProgramPipelines( NumPipelines, pipelines );

// zaktivuj pipeline a nastav stage z oddělených program objektů
glBindShaderPipeline( pipelines[VF] );
glUseProgramStages( pipelines[VF], GL_VERTEX_SHADER_BIT, vertPgm );
glUseProgramStages( pipelines[VF], GL_FRAGMENT_SHADER_BIT, fragPgm1 );

glBindShaderPipeline( pipelines[VTF] );
glUseProgramStages( pipelines[VTF],
   GL_VERTEX_SHADER_BIT|GL_TESS_EVALUATION_SHADER_BIT, multiPgms);
glUseProgramStages( pipelines[VTF], GL_FRAGMENT_SHADER_BIT, fragPgm2 );
```

Program nebo pipeline - kdo má prioritu?

- aktivní shadery se dají nastavit dvěma způsoby
  - `glUseProgram()` pro *zapouzdřené* programy
  - `glBindProgramPipeline()` pro konfigurovatelné shader pipeline
- platí následující pravidla
  - programy nastavené pomocí `glUseProgram()` mají nejvyšší priroitu, i když je nastavená pipeline
  - pipeline se použije v případě, že není aktivní žádný program (`glUseProgram()` nebyl zavolán nebo byl volán s nulou)

Volby ve shaderech

- často se dva shadery stejného typu liší jen trochu

Fragment shader

```
vec4 AmbientColor( vec3 n ) {
   return Materials.ambient;
}

vec4 DiffuseColor( vec3 n ) {
   return Materials.diffuse *
      max(dot(normalize(n),
         LightVec.xyz), 0.0 );
}
```

```
uniform int mode;
in vec3 n; // lighting normal
out vec4 fragColor;

struct Materials {...};

void main()
{
   if( mode == 0 )
      fragColor = AmbientColor(n);
   else
      fragColor = DiffuseColor(n);
}
```

Problémy s tímto přístupem

- **branching**!!! - stále to není GPU-friendly
- extra instrukce, které nedělají užitečnou práci
  - toto vadí obzvláště ve fragment shaderech, kde se každá instrukce počítá

Řešení: shader podprogramy

- efektivně se to chová jako ukazatele na funkci v C
- dovoluje optimalizovanější shader
  - odstraněny nepotřebné `if-else` výrazy
  - dovoluje to driveru optimalizovat spuštění shaderu
- **shader subroutines** dovolují aplikaci specifikovat, který podprogram z dané množiny se má spustit v shaderu

Shader podprogramy

Shader

```
#version 400 core
// deklaruj typ podprogramu
subroutine vec4 LightFunc( vec3 n );

// specifikuj, které funkce mají typ daného podprogramu
subroutine (LightFunc) vec4 AmbientColor( vec3 n ) {...}
subroutine (LightFunc) vec4 DiffuseColor( vec3 n ) {...}

// deklaruj uniform proměnnou pro volbu podprogramu
subroutine uniform LightFunc materialShader;

void main()
{
   materialShader( n );
}
```

Shader podprogramy

Aplikace

```
// zjisti indexy podprogramů
GLuint ambient = glGetSubroutineIndex( program,
   GL_FRAGMENT_SHADER, "AmbientColor" );
GLuint diffuse = glGetSubroutineIndex( program,
   GL_FRAGMENT_SHADER, "DiffuseColor" );

// zjisti umístění uniform proměnné pro volbu podprogramu
GLuint materialShaderLoc =
   glGetSubroutineUniform( program, "materialShader");

// nastav shader uniform na příslušný index podprogramu
GLint numSubroutines;
glGetIntegerv( GL_ACTIVE_SUBROUTINE_UNIFORM_LOCATIONS, &numSubroutines );

GLuint* indices = new GLuint[numSubroutines];
indices[materialShaderLoc] = ambient;
glUniformSubroutinesuiv( GL_FRAGMENT_SHADER, numSubroutines, indices );
```

Obsah

1. Úvod
2. Shadery
3. Texturování v OpenGL 4.0
4. Geometrické objekty v OpenGL
5. Přehled dalších novinek
6. Literatura

Texture gather

- uživatelsky specifikovaný výběr texelů od aktuální texturové souřadnice
- v podstatě se jedná o zobecněný bilineární filtering
- vrací původní hodnoty texelů, můžete provést vlastní filtrování
- GLSL varianty
  1. `textureGather`
  2. `textureGatherOffset`
  3. `textureGatherOffsets`

Texture gather 2

příklad texture gather

```
vec2 offsets[4];
vec4 texels = textureGatherOffsets( texture,
              texCoord, offsets, [comp] );
```

- funkce `textureQueryLod` vrací parametry mip-mappingu
  - `textureGather*` spolu s `textureQueryLod` umožňuje v podstatě vlastní (nejen) anisotropní filtrování

Obsah

1. Úvod
2. Shadery
3. Texturování v OpenGL 4.0
4. Geometrické objekty v OpenGL
5. Přehled dalších novinek
6. Literatura

Reprezentace geometrických objektů

- geometrické objekty jsou reprezentovány **vertexy**
- vertex je sada obecných atributů
  - pozice v prostoru
  - parametry barev
  - texturové koordináty
  - jakákoliv další data přímo spojená s tímto konkrétním bodem v prostoru
- vertexová data musí být uložena ve **vertex buffer objektu** (VBOs)
- VBOs uloženy ve **vertex array objektu** (VAOs)

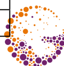
Datové objekty v OpenGL

- téměř všechna data posílaná do OpenGL musí být uložena na straně serveru (GPU)
- všechny objekty používají stejný postup
  1. generuj *jméno* objektu - `glGenBuffers()`
  2. nastav *jméno* objektu jako aktivní - `glBindBuffer()`
  3. inicializuj nebo updatuj data objektu - `glBufferData()`
  4. nastav *jméno* objektu jako aktivní pro použití jeho dat - `glBindBuffer()`

Typy datových bufferů v OpenGL

- existuje řada typů bufferových objektů v OpenGL

Typ bufferu	Popis uložených dat
GL_ARRAY_BUFFER	vertexové atributy
GL_ELEMENT_ARRAY_BUFFER	vertexové indexy
GL_DRAW_INDIRECT_BUFFER	struktury pro *indirect* rendering
GL_COPY_READ_BUFFER	data pro čtení určená pro kopírování do jiných bufferů
GL_COPY_WRITE_BUFFER	zapisovatelná data pro kopírování z jiných bufferů
GL_PIXEL_PACK_BUFFER	zapisovatelná data pro čtení pixelů
GL_PIXEL_UNPACK_BUFFER	data pro čtení na inicializaci textur
GL_TRANSFORM_FEEDBACK_BUFFER	transformované vertexové atributy
GL_TEXTURE_BUFFER	texturový buffer pro použití v shaderu
GL_UNIFORM_BUFFER	uniform proměnné pro shader



Vertex Array Objects (VAOs)

- VAOs souhrnně ukládají data geometrického primitiva
- kroky pro použití VAO
  1. generuj *jméno* VAO objektu - `glGenVertexArrays()`
  2. nastav VAO jako aktivní pro inicializaci - `glBindVertexArray()`
  3. updatuj všechny VBO asociované s tímto VAO objektem
  4. nastav VAO jako aktivní pro kreslení
- toto umožňuje nastavit všechny data nutné pro vykreslení objekty pomocí **jednoho** volání fuknce
- dříve bylo třeba mnoha volání pro nastavení všech dat jako aktivní

VAOs - příklad

použití Vertex Array Object

```
enum{ Cube, Teapot, NumVAOs };
GLuint VAO[NumVAOs];

glGenVertexArrays( NumVAOs, VAO );
glBindVertexArray( VAO[Teapot] );
// nastav VBO data asociovaná s touto konvicí
...

// vykreslení
glBindVertexArray( VAO[Teapot] );
```

Vertex Buffer Objects (VBOs)

- vertexová data musí být uložena ve VBO a asociovaná s VAO
- podobné jako inicializace VAO
  1. generuj *jméno* VBO objektu - `glGenBuffers()`
  2. nastav VBO jako aktivní - `glBindBuffer( GL_ARRAY_BUFFER, ... )`
  3. nahraj data do VBO - `glBufferData( GL_ARRAY_BUFFER, ... )`
  4. nastav VAO jako aktivní pro vykreslení - `glBindVertexArray()`

VBOs - příklad

použití Vertex Buffer Object

```
GLfloat vertices[][3] = { {0.0, 0.0, 0.0},
  {0.0, 0.0, 1.0}, {0.0, 1.0, 0.0}, ... };

glBindVertexArray( VAO[Cube] );

enum{ Array, NumBuffers };
GLuint buffers[NumBuffers];

glGenBuffers( NumBuffers, buffers );
glBindBuffer( GL_ARRAY_BUFFER, buffers[Array] );
glBufferData( GL_ARRAY_BUFFER, sizeof(vertices),
   vertices, GL_STATIC_DRAW );
```

Ukládání vertexových atributů

- *vertex arrays* jsou velmi flexibilní

uložení dat spojitě jako pole

```
GLsizeiptr size = sizeof(v) + sizeof(c) + sizeof(tc);
glBufferData( GL_ARRAY_BUFFER, size, NULL, GL_STATIC_DRAW );
GLsizei offset = 0;
glBufferSubData( GL_ARRAY_BUFFER, offset, sizeof(v), v);
offset += sizeof(v);
glBufferSubData( GL_ARRAY_BUFFER, offset, sizeof(c), c);
offset += sizeof(c);
glBufferSubData( GL_ARRAY_BUFFER, offset, sizeof(tc), tc);
```

Ukládání vertexových atributů 2

uložení dat jako spojité pole struktur

```
struct VertexData {
   GLfloat tc[2];
   GLubyte c[4];
   GLfloat v[3];
};

VertexData verts[];

glBufferData( GL_ARRAY_BUFFER, sizeof(verts),
   verts, GL_STATIC_DRAW );
```

Přehled

- `double` (4.0, 4.1) - výpočty s využitím double přesnosti, není běžné mimo specializované výpočetní karty, na mnohých kartách přepočet s využítím single-precision jednotky
- `GL_ARB_debug_output` (4.1) - alternativa ke `glGetError` pomocí callback funkcí
- `GL_ARB_get_program_binary` (4.1) - možnost získání binárního kódu shaderu, aktuálně dobré pouze pro cachování (nepřenositelné)

Přehled 2

- atomický čítač (4.2)
- `GL_ARB_shader_image_load_store` (4.2) - dovoluje **načítání i ukládání** do textury a atomické operace na ní

Obsah

1 Úvod

2 Shadery

3 Texturování v OpenGL 4.0

4 Geometrické objekty v OpenGL

5 Přehled dalších novinek

6 Literatura

Literatura

- OpenGL Architecture Review Board: **OpenGL Programming Guide: The Official Guide to Learning OpenGL**, Addison-Wesley, nejnovější vydání (aktuálně 8. vydání pro OpenGL 4.1)
- Randi J. Rost, Bill Licea-Kane: **OpenGL Shading Language, 3rd Edition**, Addison-Wesley,
- The Khronos Group: **The OpenGL Graphics System: A Specification (Core/Compatibility profile)**, `http://www.opengl.org/registry/`
- Christophe Riccino: **OpenGL reviews**, `http://www.g-truc.net/post-opengl-review.html`
- Wikipedia: `http://en.wikipedia.org/wiki/OpenGL`
- OpenGL tutorials: `http://nehe.gamedev.net`