PV112 Programování grafických aplikací

Jaro 2017

Výukový materiál

---

7. přednáška: Textury pokračování, framebuffer objekty

Textury

K texturám se v shaderech přistupuje přes speciální datový typ *sampler*.

```
sampler1D
sampler2D
...
```

Taková proměnná musí být typu uniform. Je to tzv. „opaque type“, samplery nelze sčítat, násobit, interpolovat, nemůžeme si vytvořit lokální proměnnou, nemůžeme do těchto proměnných zapsat, nemůžeme tyto proměnné vracet z funkcí. Do funkcí je můžeme předat pouze jako in, nelze jako out nebo inout. Nelze je inicializovat ze shaderů jako lze ostatní uniform proměnné.

Lze vytvořit strukturu obsahující sampler nebo vytvořit pole samplerů, ale taková struktura či pole musí být uniform proměnná.

OpenGL API k samplerům přistupuje jako k jakékoliv jiné uniformní proměnné. Pomocí funkce glGetUniformLocation získáme její ID:

```
my_tex_loc = glGetUniformLocation(my_program, ”my_tex”);
```

Příklad přístupu ze shaderu:

```
uniform sampler1D my_colors;
uniform sampler2D my_tex;
```

Texturovací jednotky

Textury nepředáváme do shaderů přímo, ale přes texturovací jednotky (*Texture Unit*). Texturovací jednotky si lze představit i jako kus hardwaru, který dělá interpolaci, wrap atd. K dispozici máme minimálně 16 texturovacích jednotek. Celý proces funguje tak, že textury nejprve navážeme na texturovací jednotku a proměnné v shaderech pak vzorkují z těchto texturovacích jednotek.

Funkce glActiveTexture nastavuje aktuální texturovací jednotku. Mění stav OpenGL – nastavuje aktuální texturovací jednotku a ovlivňuje tak další funkce jako je glBindTexture, která váže texturu na právě aktivní texturovací jednotku.

```
void glActiveTexture(GLenum texture)
```

Vzhledem k tomu, že můžeme mít několik navázaných textur zároveň (každou na jiné texturovací jednotce), musíme si dát pozor na to, že i funkce jako jsou glTexParameter*, glTexImage*, glGenerateMipmap a další, které upravují aktuálně navázanou texturu, ovlivňují pouze texturu navázanou na právě aktivní texturovací jednotku.

Pozor, parametrem je 0, nikoliv GL_TEXTURE0. (A naopak, parametrem glActiveTexture je GL_TEXTURE0, nikoliv 0).

Pomocí *glUniform1i* nastavím číslo texturovací jednotky:

```
glUniform1i(my_tex_loc, 0);
```

Příklad:

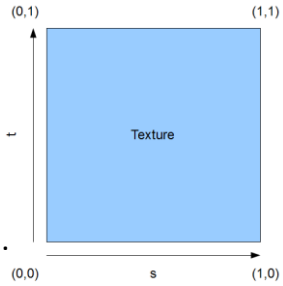
```
uniform sampler2D my_tex1;
uniform sampler2D my_tex2;
...
my_tex1_loc = glGetUniformLocation(my_program, "my_tex1");
my_tex2_loc = glGetUniformLocation(my_program, "my_tex2");
...
glUniform1i(my_tex1_loc, 0);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, wood_tex);
glUniform1i(my_tex2_loc, 1);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, rocks_tex);
...
glDraw*(...);
```

```
uniform sampler2D my_tex1;
uniform sampler2D my_tex2;
...
my_tex1_loc = glGetUniformLocation(my_program, "my_tex1");
my_tex2_loc = glGetUniformLocation(my_program, "my_tex2");
glUniform1i(my_tex1_loc, 0);
glUniform1i(my_tex2_loc, 1);
...
glActiveTexture(GL_TEXTURE0);
```

```
glBindTexture(GL_TEXTURE_2D, wood_tex);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, rocks_tex);
...
glDraw*(...);
```

Texturovací souřadnice

Texturovací souřadnice určuje, na kterém místě máme texturu navzorkovat. Je normalizovaná do intervalu [0,1], je tedy nezávislá na rozlišení textury. Na souřadnice mimo interval [0,1] se aplikuje wrap mode nastavený pomocí GL_TEXTURE_WRAP_S/T/R. Pokud se netrefíme přímo do pixelu, použije se filtrování (MIN/MAG filter). Texturovací souřadnice se označují jako:



- *s*, *t*, *r*, *q* v OpenGL API
- *s*, *t*, *p*, *q* v GLSL
- *u*, *v*, *w* v jiných knihovnách či softwarech

UVW se používá třeba v Direct3D, Blender, Cinema4D (UV coordinates).

Vzorkování textury

Vzorkování textury se provádí pomocí funkce:

```
vec4 texture(sampler* sampler, vec* coord)
```

Parametry této funkce jsou následující:

- *sampler* = objekt reprezentující texturu
- *coord* = souřadnice, na které texturu vzorkovat (dimenze závisí na sampleru)

Funkce navzorkuje texturu na dané texturovací souřadnici, vrací objekt typu *vec4*, tj. včetně hodnoty alpha.

Příklad:

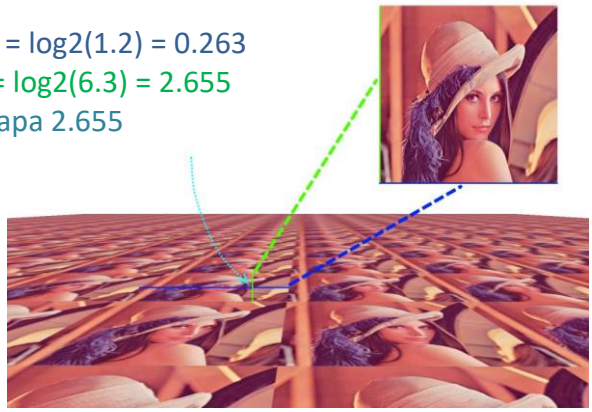
```
in vec2 tex_coord;
uniform sampler2D my_tex;
out vec4 final_color;
void main()
{
    vec3 color = texture(my_tex, tex_coord).rgb;
    final_color = vec4(color, 1.0);
}
```

Problémy při vzorkování textury:

- Při zavolání funkce *texture* OpenGL automaticky spočítá úroveň mipmapy.
- Mipmapa se počítá na základě derivací v každém směru.
- Derivace se nejčastěji počítá numericky pomocí vzorců pro dopřednou/zpětnou derivaci. Ačkoliv se jednotlivé fragmenty při běhu fragment shaderu nemohou ovlivňovat, jak jsme si uvedli na první přednášce, implementace může využít toho, že všechny pixely vzorkují texturu, hodnoty texturovacích souřadnic tak lze použít pro výpočet derivace. Proto je ale potřeba, aby sousední fragmenty tuto texturu vzorkovaly. To není tak jednoduché zařídit, zejména v případech, kdy vzorkování textury je možné přeskočit podmínkou, discard, returnem či cyklem, nebo třeba ve vertex shaderu, kdy vůbec nepracuji s fragmenty.

Následující příklad ilustruje, jakým způsobem OpenGL získá level mipmapy. OpenGL počítá, kolikrát je třeba texturu roztáhnout ve směru s a t a na základě toho volí mipmapu, která odpovídá tomuto roztažení.

dx : log2(512/426) = log2(1.2) = 0.263
dy : log2(512/81) = log2(6.3) = 2.655
Mipmapa 2.655



Příklad problému a jeho možného řešení:

V některých případech lze přeskládat příkazy, aby texturování probíhalo před podmínkami.

- Problém:

```
float my_alpha = texture(my_alpha_tex,
tex_coord).a; if (my_alpha < 0.5)
    discard;
vec3 my_color = texture(my_color_tex,
tex_coord).rgb;
...
```

- Řešení:

```
float my_alpha = texture(my_alpha_tex, tex_coord).a;
vec3 my_color = texture(my_color_tex, tex_coord).rgb;
if (my_alpha < 0.5)
    discard;
...
```

V jiných situacích toto přeskládání provést nelze, jako například v následujícím příkladu, kdy počítáme počet potřebných kroků, než dostaneme z textury správnou hodnotu alpha. Nebo když chceme získat nějakou hodnotu ve vertex shaderu.

- Problém:

```
int count = 0;
while (texture(my_tex, coord).a < 0.5)
{
    coord += step;
    count++;
}
...
```

Řešení těchto problémů spočítá v použití jiných funkcí pro texturování.

Funkce *textureLod* dělá totéž jako texture, ale umožňuje nám explicitně zadat LOD. Lze ji tak použít jak v cyklech, tak zejména ve vertex shaderu.

```
vec4 textureLod(sampler* sampler, vec* coord, float lod)
```

Parametry jsou následující:

- *sampler, coord* = stejné jako u *texture*
- *lod* = úroveň mipmapy, kterou použít

Mezi další funkce, které GLSL obsahuje pro texturování, zmíníme některé.

```
vec4 texture(sampler* sampler, vec* coord [, float bias])
```

Funkce *texture* má ve skutečnosti ještě jeden nepovinný parametr bias, který když zadáme, tak se přičte k vypočtenému LOD (k levelu používané mipmapy).

```
vec4 textureProj(sampler* sampler, vec* coord [, float bias])
```

Funkce *textureProj* provede dělení poslední souřadnicí před vzorkováním. Lze tak docílit projekce textury, například kdybychom chtěli udělat projektor. Počet souřadnic je o jednu větší než u funkce *texture* (2 souřadnice u 1D textury, 3 souřadnice u 2D textury atd.)

```
vec4 textureProjLod(sampler* sampler, vec* coord, float lod)
```

Jedná se o kombinaci *textureLod* a *textureProj.*

Existují další funkce, které pracují s texturami, například pro získání texelu zadáním nenormalizované souřadnice v intervalu [0, width/height] (texelFetch), pro získání velikost textury (textureSize), atd. Ty zde ale nebudeme přesně popisovat, protože je pravděpodobně v rámci tohoto předmětu nevyužijete. Pokud byste je potřebovali, jejich popis lze najít na internetu nebo ve specifikaci.

Využití texturování

Jakým způsobem nyní využít to, co jsme získali z textury? Barvu z textury nejčastěji použijeme jako ambientní a difusní složku materiálu, nebo jako průhlednost objektu.

- Ambientní a difusní složka

```
color =
  light_ambient * tex_color +
  light_diffuse * tex_color * Idiff +
  light_specular * mat_specular * Ispec;
```

- Alpha

```
final_color = vec4(color, tex_alpha);
```

- Děravé objekty

```
if (tex_alpha < 0.5)
    discard;
```

Dnes je možné použít textury „prakticky kdekoliv“. Textury jsou používány pro změnu normály na povrchu objektu, nebo obsahují kompletní okolí objektu. Textury také můžou ovlivňovat výšku vrcholů a tak vytvářet boule, můžou obsahovat výchylku listů trávy či stromů, nebo obsahovat vzdálenost objektů, které jsou nejblíže světlu, a umožnit tak výpočet stínů. Vzhledem k tomu, že interpretace dat v textuře je na nás, jejich použití je opravdu rozmanité.

Anisotropické vzorkování

Jedná se o rozšíření v OpenGL, ale je dostupné prakticky všude. Řeší problém vzorkování textur, které jsou v jednom směru mnohem víc natažené než ve druhém směru. Mipmapping vybere menší texturu, a tak jakoby „rozmazává“. Problém je nejčastěji vidět u objektů, které jdou téměř rovnoběžně s pohledem, jako jsou cesty, silnice apod.

Všimněme si, že je textura v zadních částech prakticky celá rozmazaná, například „oranžový sloupec“ v levé části textury Lenny už ke konci nejde prakticky rozpoznat. Důvodem je to, že OpenGL ke konci používá mipmapy, které mají velikost v jednotkách pixelů.

Anisotropické vzorkování nastavujeme pomocí funkce *glTexParameterf* a parametru *GL_TEXTURE_MAX_ANISOTROPY_EXT.* Hodnoty se nacházejí v intervalu [1, 16], kde 1 je „žádné anisotropické vzorkování“ a 16 je maximální hodnota, kterou většina grafických karet podporuje.

```
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAX_ANISOTROPY_EXT, 8.0f);
```

Rozšíření OpenGL

Setkali jsme se zde s rozšířeními v OpenGL, proto na chvilku odbočíme od textur a podíváme se podrobněji na to, jak se s nimi pracuje.

Funkce a konstanty, které dané rozšíření přidává, mají většinou příponu podle toho, kdo je podporuje. Některé rozšíření jsou určené jen pro některé grafické karty (NV – NVidia, AMD, INTEL), i když i přesto je lze najít na grafických kartách jiných výrobců. Rozšíření, která se od počátku vymýšlí tak, aby byly implementovatelné na více grafických kartách, mají příponu EXT. Rozšíření, které vytvoří skupina ARB (Architecture Review Board), mají koncovku ARB a jsou psány „s ohledem na všechny“ – jsou to například části nových verzí OpenGL, aby bylo možné je použít i tam, kde není možné podporovat kompletní novou verzi OpenGL.

Popis jednotlivých rozšíření lze najít na internetu na stránce www.opengl.org/registry nebo na stránkách daného výrobce grafických karet.

Počet dostupných rozšíření lze získat pomocí volání funkce *glGetIntegerv* s parametrem GL_NUM_EXTENSIONS a jména jednotlivých rozšíření iterativně voláním funkce *glGetStringi* s parametrem GL_EXTENSIONS přes indexy všech rozšíření.

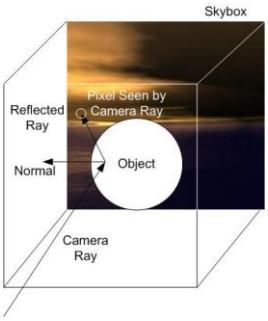
A nyní zpět k texturám.

Texturovací souřadnice není nutné zadávat spolu s objektem. V některých případech je výhodnější (nebo přímo nutné) je odvodit z ostatních dat vrcholů, jako je jejich pozice, normála atd.

Při odvozování texturovacích souřadnic se používají data před transformováním (v souřadnicovém systému objektu), nebo se transformují do jiného souřadnicového systému. Pokud nebudeme nic transformovat, textura se bude držet na objektu, což využijeme například při vytváření materiálu textury (například dřeva, mramoru atd.). Transformace použijeme například tehdy, když máme texturu fixovanou na nějaký objekt ve scéně (textura intenzity světla) a to, kde se na objektu zobrazí, závisí na tom, kde se tento objekt ve scéně nachází.

Generování texturovacích souřadnic si ukážeme na tzv. environment mappingu. To je jednoduchá, ale rychlá technika pro odlesky okolních objektů.

Princip je velmi jednoduchý. Vypočteme směr, ve kterém se pozorovatel dívá na daný pixel (I_{in}) a tento směr odrazíme podle normály (N). Pro výpočet odraženého směru (I_{out}) lze použít následující vzorec, nebo funkci reflect, která je přítomná v GLSL.



$$I_{out} = I_{in} - 2(N \cdot I_{in})N$$

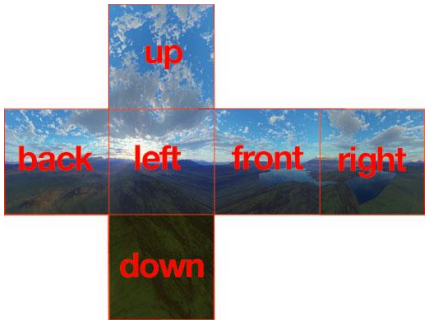
I_{out} = reflect(I_{in}, N)

Odražený směr poté použijeme k vzorkování cube textury, která obsahuje okolí objektu.

Cube textury

Cube textura je šestice textur, která reprezentuje šest stěn kostky. Není to tedy 3D textura (ta obsahuje objem této kostky), a není to ani jedna 2D textura, která vypadá jako kříž.

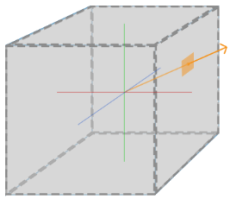
Pro označení cube textur OpenGL používá konstantu GL_TEXTURE_CUBE_MAP (podobně jako je třeba GL_TEXTURE_2D). Jednotlivé obrázky jsou načítány pomocí funkce *glTexImage2D*, prvním parametrem určujeme, kterou stěnu kostky načítáme.



Každá stěna cube mapy musí být čtverec (výška = šířka) a všechny stěny cube mapy musí mít stejnou velikost a stejný internal_format. Orientaci jednotlivých textur lze najít ve specifikaci OpenGL nebo na internetu.

```
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X, 0, GL_RGB,
    512, 512, 0, GL_RGB, GL_UNSIGNED_BYTE, data_px);
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Y, 0, GL_RGB,
    512, 512, 0, GL_RGB, GL_UNSIGNED_BYTE, data_py);
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Z, 0, GL_RGB,
    512, 512, 0, GL_RGB, GL_UNSIGNED_BYTE, data_pz);
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_X, 0, GL_RGB,
    512, 512, 0, GL_RGB, GL_UNSIGNED_BYTE, data_nx);
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, 0, GL_RGB,
    512, 512, 0, GL_RGB, GL_UNSIGNED_BYTE, data_ny);
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_Z, 0, GL_RGB,
    512, 512, 0, GL_RGB, GL_UNSIGNED_BYTE, data_nz);
```

Ke vzorkování se používají 3 souřadnice, které reprezentují směr, kterým se díváme, jako kdybych byl uprostřed kostky v bodě (0,0,0) a kostka byla okolo mě. Daný směr nemusí být normalizovaný.



Jak tedy vypadá kód environment mappingu? Ve fragment shaderu vypočteme odražený směr a tento směr použijeme pro vzorkování textury. V C++/Java kódu navážeme texturu přes GL_TEXTURE_CUBE_MAP.

```
uniform samplerCube my_env;
...
vec3 reflected = reflect(incoming, normal);
vec3 color = texture(my_env, reflected);
...
...
glUniform1i(my_env_loc, 0);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_CUBE_MAP, skybox);
glDraw*
...
```

Jaké jsou výhody a nevýhody této techniky? Mezi výhody jistě patří, že je jednoduchá, rychlá a dává hezké výsledky. Má ovšem několik nevýhod, které musíme mít na paměti. Mezi dvě největší patří to, že vzorkování vychází pouze z odraženého směru a nebere v potaz místo, odkud byl paprsek odražen. Pokud jsou ostatní objekty daleko, tak to tolik nevadí, ale pokud jsou objekty blízko, výsledek může vypadat nepřirozeně. Druhým problémem je to, že objekt samotný v cube mapě pravděpodobně nebude přítomen, a proto sám sebe neodráží.

V souvislosti s cube texturami si uvedeme trochu víc o kompletnosti textur. Když pracujeme s cube texturami (a také s mipmapami), musíme si dát pozor na to, že máme kompletní sadu textur. Není to nic složitého. Cube textura musí mít zadány všechny stěny. Textura, u které používáme pro filtrování mipmapping, musí mít definovány všechny levely až do úrovně 1x1x1. Pokud použijeme nekompletní texturu, získáme hodnotu (0,0,0,1).

V případě mipmappingu nám OpenGL umožňuje omezit úrovně mipmap, které jsme schopni zadat, a tak si zajistíme, že můžeme použít mipmapping i v případech, kdy nemáme textury až do velikosti 1x1x1. Slouží k tomu parametry GL_TEXTURE_BASE_LEVEL a GL_TEXTURE_MAX_LEVEL. Díky nim nemusíme definovat všechny mipmap úrovně (například když je nemáme připraveny) a přesto použít mipmapping. Například zde nám stačí definovat obrázky pro mipmap úroveň 1, 2 a 3.

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_BASE_LEVEL, 1);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, 3);
```

Další z věcí, kterou si vysvětlíme, budou komprimované textury. Komprimované textury jsou textury, které jsou uloženy v paměti grafické karty ve zkomprimované podobě, a tak šetří množství paměti, kterou zabírají. Využívají se zejména v počítačových hrách, kde umožňují mít dostupné větší množství textur. Dekomprimace probíhá až při vzorkování. Vzorkování je tedy náročnější na výpočet, ale na druhou stranu vzhledem k tomu, že textury jsou menší, dokáží lépe využít cache.

Pro kompresi se používají speciální algoritmy vyvinuté pro tyto účely (žádný zip (LZW), PNG, JPG apod.). Komprimační algoritmy mají spoustu společných prvků. Obrázky jsou rozděleny do malých bloků (např. 4x4) a tyto bloky jsou samostatně komprimovány na stejnou velikost (64 bitů, 128 bitů apod.). Liší se kvalitou a efektivitou komprese. Pro různé textury jsou určeny různé kompresní algoritmy, jsou jiné algoritmy pro RGB textury, pro průhledné textury, pro textury s normálovou mapou, pro textury s HDR daty atd.

V OpenGL jsou komprimované textury reprezentovány pomocí speciálních konstant pro internal_format. Lze je načíst pomocí funkcí *glTexImage**, které načítají nezkomprimovaná data a kompresi provede OpenGL (což stojí čas při každém načítání), nebo je možné provést kompresi textury během přípravy dat v externím programu a načíst rovnou zkomprimovanou texturu pomocí funkce:

```
void glCompressedTexImage2D(GLenum target,
  GLint level, GLenum internal_format,
  GLsizei width, GLsizei height,
  GLint border, GLsizei imageSize,
  const GLvoid *data)
```

Parametry této funkce jsou prakticky stejné jako parametry funkce *glTexImage2D*, liší se pouze tím, že nezadáváme formát a typ dat. Formát a typ (a vlastně i velikost) dat je určena parametrem internal_format.

Nyní si uvedeme pár formátů komprimovaných textur. Některé formáty komprimace textury musí být přítomny ve všech implementacích OpenGL, některé jsou dostupné pouze v podobě rozšíření.

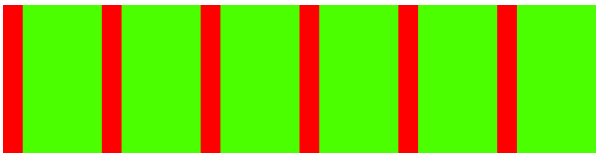
Formáty S3TC jsou dostupné pouze přes rozšíření, hlavně z toho důvodu, že algoritmus komprese a dekomprese podléhá patentu (patentu ale nepodléhá uchovávání zkomprimovaných dat). I přesto je toto rozšíření dostupné na většině implementací OpenGL.

Některé formáty jsou povinně dostupné ve všech verzích OpenGL. Jsou to formáty skupiny RGTC (výhradně pro textury obsahující pouze jednu či dvě barvy), které jsou dostupné od OpenGL 3.0, formáty skupiny BPTC, dostupné od OpenGL verze 4.2, a formáty skupiny ETC, dostupné od OpenGL 4.3. Mezi nejnovější formáty komprimovaných textur patří formát ASTC, dostupný přes rozšíření. Jeho cílem je nahradit ostatní formáty, které jsou určeny jen pro některé druhy textur. Formát ASTC má různé varianty lišící se v počtu bitů na pixel (8, 3.56, 2).

Procedurální textury

Nyní se trochu víc zaměříme na procedurální textury. Procedurální textury jsou textury, které barvu v daném místě nezískávají z paměti, ale výpočtem. Ilustrovat to můžeme následujícím kódem, který nám vytvoří jednoduchou texturu s červeno-zelenými pruhy.

```
vec3 textureStripes(vec2 coord)
{
    if (fract(coord.s) < 0.2)
        return vec3(1.0, 0.0, 0.0);
    else
        return vec3(0.0, 1.0, 0.0);
}
```



Lze najít různé procedury: pro generování textury dřeva, pro generování textury mramoru, a spoustu dalších. Algoritmy často používají funkce pro generování náhodných čísel a šumové funkce (např. Perlinův šum).

Procedurální textury mají mnoho výhod i nevýhod. Mezi výhody jistě patří to, že nezabírají místo v paměti, jsou spojité (lze udělat hladké přechody), netrpí aliasingem, a tak je lze libovolně zvětšovat. Mezi nevýhody patří to, že je nutné provádět nějaký výpočet (který nemusí být vždy triviální), což ovlivní dobu kreslení.

S texturami souvisí i sampler objekty. Pozor, nemyslíme tím datový typ sampler v GLSL, o kterém jsme mluvili na začátku přednášky! Toto je objekt, podobně jako textura nebo buffer.

Sampler objekty byly přidány do OpenGL ve verzi 3.3 a najdeme je v jiných API, jako je třeba Direct3D. Obsahují některé z parametrů textur, z těch, které jsme zmiňovali. To jsou WRAP mód, border color, MIN/MAG filter a hodnota anisotropického filtrování. Podobně jako buffery nebo textury obsahuje OpenGL funkce pro vytváření, rušení, a dotázání se na existenci sampler objektu.

```
void glGenSamplers(GLsizei n, GLuint *samplers)
```

Funkce vytvoří *n* sampler objektů.

```
void glDeleteSamplers(GLsizei n, const GLuint * samplers)
```

Zruší sampler objekty.

```
GLboolean glIsSampler(GLuint id)
```

Vrátí *true,* jestli daný sampler objekt existuje.

```
void glSamplerParameter[if][v](
  GLuint sampler, GLenum pname, T param)
```

K nastavení parametrů sampleru slouží funkce *glSamplerParameter**. Tato funkce je prakticky totožná s funkcí *glTexParameter**. Jediný rozdíl je ten, že nastavuje parametry přímo danému sampleru, a nikoliv tomu, který je aktuálně navázán, jako to je u funkce *glTexParameter**.

```
void glBindSampler(GLuint unit, GLuint sampler)
```

Pro navázání sampleru na texturovací jednotku slouží funkce *glBindSampler*. Jakmile je sampler navázán, OpenGL přestane používat odpovídající parametry textury a začne používat parametry, které definuje sampler (včetně například použití mipmap, takže se může stát, že získáme kompletní/nekompletní texturu). Pokud navážeme sampler 0, OpenGL opět začne používat parametry, které jsou definovány společně s texturou.

Příklad:

```
GLuint my_sampler;
glGenSamplers(1, &my_sampler);
glSamplerParameteri(my_sampler,
    GL_TEXTURE_WRAP_S, GL_REPEAT);
glSamplerParameteri(my_sampler, set other parameters);
...
glUniform1i(my_tex_loc, 0);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, my_tex);
glBindSampler(0, my_sampler);
glDraw*
```

V souvislosti se vstupními atributy jsme ještě zapomněli popsat funkce *glBindAttribLocation*.

```
void glBindAttribLocation(GLuint program,
  GLuint index, const GLchar * name)
```

Tato funkce slouží k tomu, abychom byli schopni vynutit index vstupní proměnné, který jinak získáváme funkcí *glGetAttribLocation*. Toto vynucení je nutné provést ještě předtím, než se program slinkuje pomocí funkce *glLinkProgram*.

Pomocí této funkce si lze vynutit to, že například atributy, které reprezentují pozice vrcholů, mají stejný index u všech shader programů. Potom nám stačí vytvořit jen jeden VAO, který váže pozice vrcholů na daný index. Pokud bychom to neudělali, OpenGL si může při linkování programu zvolit indexy jakkoli. My bychom tak potřebovali několik VAO každý podle toho, na který index musíme data s pozicemi (a normálami a texturovacími souřadnicemi) navázat.

Framebuffer objekty

Nyní se pustíme do další části OpenGL a tou jsou framebuffer objekty. Jak již víme, framebuffer je to, do čeho kreslíme frame (snímek). Framebuffer objekt je objekt, do kterého taktéž kreslíme, ale nekreslíme do hlavního okna aplikace, ale do obrázků, které nám zůstanou v paměti jako textury. Tomuto postupu, kdy kreslíme jinam než do okna aplikace, se říká offscreen rendering. Jejich použití je rozsáhlé, stejně jako jsou rozsáhlé možnosti, co lze udělat se získaným obrázkem - například postprocessing, kreslení do textury, kreslení ve vyšším rozlišení (pro screenshoty) atd.

Framebuffer objekty jsou objekty jako třeba buffer, textury či samplery, a proto se i u nich setkáme s trojicí známých funkcí pro vytváření objektů:

```
void glGenFramebuffers(GLsizei n, GLuint *framebuffers)
```

Funkce vytvoří *n* framebuffer objektů.

```
void glDeleteFramebuffers(GLsizei n, const GLuint *framebuffers s)
```

Funkce zruší framebuffer objekty.

```
GLboolean glIsFramebuffer(GLuint id)
```

Funkce vrátí *true* jestli daný framebuffer objekt existuje.

```
void glBindFramebuffer(GLenum target, GLuint framebuffer)
```

Pro navázání framebufferu se používá funkce *glBindFramebuffer*, která nám může připomínat například funkci *glBindTexture*. Jako první parametr target můžeme použít tři konstanty, GL_READ_FRAMEBUFFER, GL_DRAW_FRAMEBUFFER a GL_FRAMEBUFFER. První konstantou navážeme framebuffer, ze kterého se číst data funkce, jako jsou *glReadPixels*, *glCopyTexImage* apod. Druhou konstantou navážeme buffer, do kterého jde výstup z kreslení. Třetí konstantou navážeme buffer na čtení i na kreslení, jakoby ušetříme jedno volání této funkce.

Druhým parametrem je objekt, který chceme navázat. Pokud použijeme hodnotu 0, OpenGL pro čtení/kreslení začne používat defaultní framebuffer, který odpovídá oknu aplikace.

Framebuffer objekt jako takový neobsahuje žádná data ani žádnou paměť. Je to jen jakoby kontejner, do kterého připojíme textury, do kterých se kreslí. OpenGL vyžaduje minimálně 8 míst pro připojení barvy. Pod touto barvou se myslí data, která jsou z fragment shaderu, ale vzhledem k tomu, že z fragment shaderu můžeme dát na výstup prakticky libovolnou informaci, označení „barva“ je příliš omezené. Ale používá se. Framebuffer nemusí mít připojenou žádnou barvu, když ji nepotřebujeme (například se zajímáme jen o hloubku).

Framebuffer dále obsahuje jedno místo pro připojení textury hloubky a stencil textury. Opět, framebuffer nemusí mít připojenou ani hloubku ani stencil, ale v takovém případě samozřejmě nebude fungovat test hloubky a stencil test. Obecně můžeme k framebuffer objektu připojit textury libovolných velikostí a není to chyba. OpenGL bude vždy kreslit jen do velikosti odpovídající průniku všech textur.

```
void glFramebufferTexture2D(GLenum target,
  GLenum attachment, GLenum textarget,
  GLuint texture, GLint level)
```

Texturu připojujeme k framebuffer objektu pomocí funkce *glFramebufferTexture2D*. První parametr funkce určuje, ke kterému framebufferu texturu připojuji (k tomu, který byl naposledy navázán na dané místo). Nemůžu navázat texturu k defaultnímu framebufferu, ale můžu ji mít navázanou na více framebuffer objektů.

Druhým parametrem je attachment, který určuje místo, kam texturu navazuju. Jedná se o jednu z konstant určující color/depth/stencil attachment. Vzhledem k tomu, že spousta vnitřních formátů textur obsahuje hloubku a stencil zároveň, existuje i parametr GL_DEPTH_STENCIL_ATTACHMENT, kterým nastavím jednu texturu na oba attachments naráz.

Třetím parametrem je GL_TEXTURE_2D, pokud navazujeme 2D texturu, nebo konstanta určující jednu ze stěn cube mapy GL_TEXTURE_CUBE_MAP_POSITIVE_X, GL_TEXTURE_CUBE_MAP_NEGATIVE_X atd.

Čtvrtým parametrem je objekt textury, kterou navazuji. Pokud zadám 0, aktuálně připnutou texturu odepnu od daného attachmentu.

Posledním parametrem je level mipmapy té textury, kterou chci připnout.

Ačkoliv je framebuffer vždy 2D, můžeme připnout 1D i 3D texturu. V případě 1D textury se OpenGL chová tak, jako bychom připnuli 2D texturu s výškou 1. V případě 3D textury musíme zadat vrstvu, do které kreslíme.

```
void glFramebufferTexture1D(GLenum target,
  GLenum attachment, GLenum textarget,
  GLuint texture, GLint level)
```

```
void glFramebufferTexture3D(GLenum target,
  GLenum attachment, GLenum textarget,
  GLuint texture, GLint level, GLint layer)
```

Příklad:

```
GLuint my_tex[3];
glGenTextures(3, my_tex);
glBindTexture(GL_TEXTURE_2D, my_tex[0]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8,
  2048, 2048, 0, GL_RGBA, GL_UNSIGNED_BYTE, nullptr);
glBindTexture(GL_TEXTURE_2D, my_tex[1]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_R32F,
  2048, 2048, 0, GL_RED, GL_FLOAT, nullptr);
glBindTexture(GL_TEXTURE_2D, my_tex[2]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH24_STENCIL8,
  2048, 2048, 0, GL_DEPTH_STENCIL, GL_UNSIGNED_INT_24_8,
  nullptr);
```

Vytvoříme si tři textury o velikost 2048x2048. První bude obsahovat barvu RGBA 8 bitů na kanál a bude reprezentovat barvu daného pixelu. Druhá bude jeden 32 bitový float a bude reprezentovat jas pixelu. Třetí obsahuje hloubku (24 bitů) a stencil (8 bitů). Ani do jedné z nich nepředávám data, textury se tedy pouze alokují v paměti OpenGL.

```
GLuint my_fbo;
glGenFramebuffers(1, &my_fbo);
glBindFramebuffer(GL_FRAMEBUFFER, my_fbo);
glFramebufferTexture2D(GL_FRAMEBUFFER,
  GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, my_tex[0], 0);
glFramebufferTexture2D(GL_FRAMEBUFFER,
  GL_COLOR_ATTACHMENT1, GL_TEXTURE_2D, my_tex[1], 0);
glFramebufferTexture2D(GL_FRAMEBUFFER,
  GL_DEPTH_STENCIL_ATTACHMENT, GL_TEXTURE_2D,
  my_tex[2], 0);
```

Poté si vytvoříme framebuffer objekt a všechny tři textury k němu připneme.

Nastavení výstupu ze shaderů

Nyní si ukážeme, jak zadat, která výstupní proměnná fragment shaderu se má zapisovat do kterého color attachmentu. To se provádí ve dvou krocích. V prvním kroku musíme získat index, který byl přiřazen dané výstupní proměnní fragment shaderu. K tomu použijeme funkci *glGetFragDataLocation*, která je podobná funkcím *glGetAttribLocation* a *glGetUniformLocation*. Pozor, vrácený index neodpovídá color attachmentu. Jedná se o index draw bufferu (jednoho z bufferů framebufferu, do kterého kreslíme).

```
GLint glGetFragDataLocation(GLuint program, const char *name)
```

Pro úplnost zmíníme, že stejně jako index atributu můžeme vynutit zavoláním *glBindAttribLocation*, i index výstupní proměnné můžeme vynutit funkcí *glBindFragDataLocation*.

```
void glBindFragDataLocation(GLuint program,
  GLuint index, const char *name)
```

Nyní nám ještě chybí ke každému indexu, ke kterému máme navázány výstupní proměnné fragment shaderu, přiřadit color attachmenty. To provedeme funkcí *glDrawBuffers*.

```
void glDrawBuffers(GLsizei n, const GLenum *bufs)
```

Parametrem této funkce je pole konstant GL_COLOR_ATTACHMENTi, které reprezentují color attachmenty, nebo GL_NONE, pokud daný výstup nechceme nikam zapisovat. Tato informace, který attachment použít pro který index výstupní proměnné, je svázána s framebuffer objektem, který je aktuálně navázán na GL_DRAW_FRAMEBUFFER.

Pro úplnost, na 4. přednášce byla zmíněna funkce glDrawBuffer (bez ‚s'). Tato funkce nastavuje color attachment jen pro první index.

Příklad:
Výstupní proměnné final_color a final_brightness navážeme na indexy 0 a 1 a při vytváření FBO na attachmenty 0 a 1. Poté kreslíme. Jakmile vykreslíme, můžeme výsledek použít například pro postprocessing (třeba zvýraznit velmi jasné pixely) a uložit obrázek v rozlišení 2048 x 2048.

```
out vec4 final_color;
out float final_brightness;
...
glBindFragDataLocation(my_program, 0, ”final_color”);
glBindFragDataLocation(my_program, 1, ” final_brightness”);
...
glBindFramebuffer(GL_FRAMEBUFFER, my_fbo);
GLenum bufs[] = {GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1};
glDrawBuffers(2, bufs);
...
glDraw*
...
// use textures my_tex for postprocessing
```

Nyní už víme to nejpodstatnější o framebufferech, proto se teď zaměříme na některé detaily, které je vhodné při práci s FBO znát. Pro získání informací o některém attachmentu framebuffer objektu slouží funkce *glGetFramebufferAttachmentParameteriv*.

```
void glGetFramebufferAttachmentParameteriv(
  GLenum target, GLenum attachment,
  GLenum pname, GLint *params)
```

První parametr má stejný význam jako u funkce *glFramebufferTexture2D* a určuje framebuffer, o který se zajímáme. Pokud je aktuálně navázaný framebuffer defaultně navázaný framebuffer (který odpovídá oknu, do kterého kreslíme), získáme informace o něm. Druhý parametr attachment určuje, o který attachment se zajímáme. Pokud se dotazujeme na framebuffer objekt, parametrem je některá z konstant specifikující attachment. Pokud se dotazujeme na defaultní framebuffer, parametrem je některá z

konstant určující jeden z bufferů defaultního framebufferu (GL_FRONT/BACK/LEFT/RIGHT/DEPTH/STENCIL, jako například u funkce *glDrawBuffer*).

Parametr pname určuje to, na co se ptáme. Možností je spousta, všechny začínají s GL_FRAMEBUFFER_ATTACHMENT_*. Zmíníme pouze dotazy na bitovou hloubku jednotlivých kanálů *_RED/GREEN/BLUE/ALPHA/DEPTH/STENCIL_SIZE, dotaz na id textury navázané na daný attachment (*_OBJECT_NAME), dotaz na level navázané textury (*_TEXTURE_LEVEL) a dotaz na navázanou stěnu cube mapy (*_CUBE_MAP_FACE).

Poslední parametrem předáme proměnnou, do které chceme uložit výstupní informaci.

Renderbuffer objekty

Starší hardware, pro který jsou psány starší verze OpenGL, nebyl vždy schopen vytvořit texturu některých formátů, ale i přesto byl schopen kreslit do těchto formátů. Příkladem mohou být textury pro multisampling nebo textury pro stencil.

Pro tyto účely byly přidány do OpenGL renderbuffer objekty. Tyto renderbuffer objekty jsou jako textury v tom ohledu, že reprezentují 2D obrázek, lze do nich ale pouze kreslit, nelze je ve fragment shaderu vzorkovat. Pokud chceme data z renderbufferu použít, musíme zavolat funkce pro kopírování dat (do CPU nebo do textury).

Dnes už je stále méně a méně situací, kdy je nutné použít renderbuffer objekt namísto textury, proto se renderbuffer objektům zde nebudeme více věnovat. Pokud byste na ně narazili a potřebovali je použít, popis funkcí, které s nimi pracují, lze najít na internetu nebo ve specifikaci OpenGL.

Při práci s framebuffer objekty se nám může stát, že budeme číst z textur, do kterých ve stejný okamžik zapisujeme. To se někdy označuje jako **feedback loops**. Nejčastěji k tomu dochází, když v shaderu vzorkujeme texturu, kterou máme připnutou na framebuffer, do kterého kreslíme, nebo když kopírujeme data (mezi framebufferem a texturou či mezi dvěma framebuffery). To je chyba a pokud se tak stane, OpenGL nedefinuje, jak se má aplikace zachovat.

Abychom mohli framebuffer používat, musíme vše dobře nastavit, pak bude „kompletní“. Ze všech věcí, které musíme dodržet, zmíníme dvě nejdůležitější.

Musíme zajistit, že interní formáty textur připnutých ke color attachmentu jsou některé z formátů pro barvu a lze tyto formáty použít pro kreslení, a obdobně u hloubkové textury a u stencil textury. To není problém zajistit. Z barevných formátů nemůžeme použít formáty pro komprimované textury, ale jinak lze použít prakticky všechny formáty. Nejbezpečnější je použít formáty pro R/RG/RGBA (tedy ne RGB), které jsou typu 8/16/32 nic/I/UI/F (například GL_RGBA8, GL_R32F apod.).

Bohužel některé kombinace textur nemusí některé implementace podporovat. OpenGL je v tomto velmi benevolentní, a proto se doporučuje vždy volat následující funkci pro kontrolu, zda je všechno v pořádku.

```
GLenum glCheckFramebufferStatus(GLenum target)
```

Funkce *glCheckFramebufferStatus* nám řekne, jestli je daný framebuffer kompletní. Pokud vrátí GL_FRAMEBUFFER_COMPLETE, je vše v pořádku. Pokud vrátí GL_FRAMEBUFFER_INCOMPLETE_ATTACHMENT, máme chybu v některém z attachmentů. Pokud vrátí GL_FRAMEBUFFER_UNSUPPORTED, implementace nepodporuje použitou kombinaci internal formátů. Framebuffer může být nekompletní ještě z několika dalších příčin. Chyby zmíněné zde jsou ty nejčastější, kompletní popis lze najít na internetu nebo ve specifikaci OpenGL.

Pro zkopírování oblasti mezi dvěma framebuffer objekty slouží funkce *glBlitFramebuffer*. Její parametry určují oblast, která bude kopírována, buffery, které budou kopírovány, a filter, který bude použit, pokud oblasti nebudou mít stejnou velikost.

```
void glBlitFramebuffer(
 GLint srcX0, GLint srcY0, GLint srcX1, GLint srcY1,
 GLint dstX0, GLint dstY0, GLint dstX1, GLint dstY1,
 GLbitfield mask, GLenum filter)
```

Funkce má několik omezení (nelze filtrovat a tedy ani kopírovat různě velké oblasti) depth nebo stencil buffery apod.). Asi nejlepší její použití je kopírování dat z framebufferu, který obsahuje multisample textury, do framebufferu, který multisample není.

Podobně jako když kreslíme do framebufferu a potřebujeme určit, co kterého color attachmentu kreslíme, stejně tak potřebujeme při čtení dat určit, ze kterého color attachmentu budeme číst data. K tomu slouží funkce *glReadBuffer*. Jejím parametrem určíme, který attachment použijeme.

```
void glReadBuffer(GLenum mode)
```

Toto nastavení ovlivňuje funkce jako jsou *glCopyTexImage**, *glReadPixels*, *glBlitFramebuffer* apod.

Na závěr zmíníme to, že provádění některých operací OpenGL lze nastavit separátně pro každý draw buffer zvlášť. Pozor, tyto operace se vztahují na indexy draw bufferů, tedy na to, na co se navazují výstupní proměnné fragment shaderu. Nejedná se o color attachmenty, a proto tato nastavení nejsou součástí parametrů framebuffer objektů. Zvlášť lze nastavit maskování zápisu do color bufferů a míchání barev.

Pro nastavení masky zápisu do color bufferu použijeme funkce *glColorMaski*. Ta se od funkce *glColorMask*, kterou jsme si už popsali, liší písmenem ‚i' na konci a prvním parametrem, kterým je index, který ovlivňujeme. Pokud použijeme funkci *glColorMask* (bez ‚i'), nastavíme masku zápisu do všech draw bufferů.

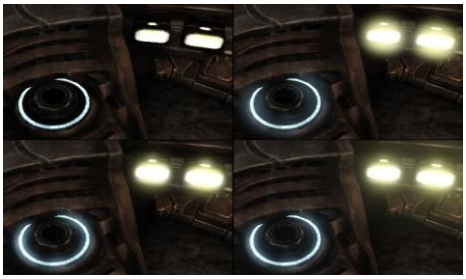
```
void glColorMaski(GLuint buf,
  GLboolean red, GLboolean green,
  GLboolean blue, GLboolean alpha)
```

Obdobně pro nastavení blendingu použijeme funkci *glEnablei*/*glDisablei* (opět s ‚i' na konci). Pokud zavoláme tyto funkce s prvním parametrem GL_BLEND, tak povolíme/zakážeme míchání barev při zápisu do daného bufferu. Funkce *glEnable*/*glDisable* (bez ‚i') opět ovlivňují všechny buffery. Od OpenGL 4.0 lze navíc nastavit pro každý buffer i rozdílné parametry míchání.

```
void glEnablei(GLenum target, GLuint index)
void glDisablei(GLenum target, GLuint index)
```

Na závěr si na několika příkladech ukážeme, k čemu a jak se framebuffer objekty používají. Začneme několika efekty, které jsou založené a postprocessingu, neboli upravují vykreslený obrázek.

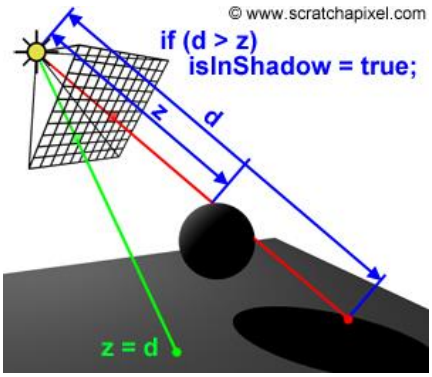
Jako první efekt si ukážeme tzv. bloom efekt. Je to efekt, kdy vybereme z obrázku nejjasnější pixely, obrázek rozmažeme, a smícháme s původním obrázkem. Výsledek vypadá, jakoby jasné objekty byly tak jasné, že přesvítí okolní objekty.



Jako druhou techniku si ukážeme tzv. screen space ambient occlusion. Tato technika simuluje útlum světla, který vzniká v rozích objektů. Princip spočívá v tom, že po vykreslení scény zpracujeme hloubku každého pixelu a pokud detekujeme roh, zabarvíme odpovídající pixel do tmava.

Výsledek kreslení do framebufferu můžeme použít rovnou při kreslení současného snímku na jiných objektech. Velmi často se této metody používá při aktualizaci cube map tak, aby objekty na sobě odrážely aktuální prostředí.

Jedna z dnes asi nejvíce používaných technik pro stíny v real-time aplikacích využívá framebufferu a kreslení do textury, ale na rozdíl od toho, aby využívala vykreslenou barvu, tak využívá zapsanou hloubku. Celá scéna je tak kreslena dvakrát, nejprve s pomocí framebuffer objektu do hloubkové textury, a poté s využitím této textury. Při druhém průchodu je spočítána vzdálenost každého kresleného pixelu od světla a s pomocí hloubkové textury je ověřeno, zdali je to ten nejbližší pixel. Grafický hardware má pro tyto testy podporu v podobě speciálních parametrů pro textury a vzorkování.



Posledním z příkladů, který si ukážeme, je tzv. deferred shading, neboli odložené stínování. Při použití této techniky nepočítáme osvětlení ihned při kreslení daného objektu, ale ukládáme data potřebná pro výpočet osvětlení do několika textur framebuffer objektu (zde máme barvu, pozici a normálu, a také hloubkovou texturu, která ovšem s deferred shadingem přímo nesouvisí). Následně, až je vykreslená celá scéna, projdeme všechny pixely a spočítáme pro ně osvětlení. Vzhledem k tomu, že osvětlení je počítáno pro všechny pixely jen jednou, ušetříme velké množství operací, což nám umožní kreslit scény, které obsahují stovky až tisíce světel.