PV112 Programování grafických aplikací

Jaro 2017

Výukový materiál

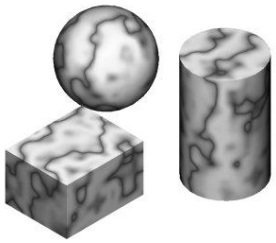
---

6. přednáška: Textury

Texturování (přesněji nanášení textur) je obecně způsob obarvení povrchu zobrazovaných těles různými obrazci. Důležité přitom je, že texturováním nijak neměníme geometrické vlastnosti těles, pouze jiným způsobem zobrazujeme jejich povrch. Obrazce, které se na povrch těles nanášejí, se označují jako **textury**. Ty jsou většinou představovány plošnými obrázky (2D textury). Některé grafické systémy však podporují í vykreslování 1D, ale i 3D textur (objemové textury), cubemaps, atd.

Obrazce pro textury je možné vytvářet několika způsoby. Mohou se buď použít klasické rastrové obrázky (namalované, fotka, scan) nebo se dá textura generovat pomocí různých algoritmů založených většinou na fraktálních technikách – takto vznikají tzv. **procedurální textury**. Ty lze použít pro výpočet rastrových obrázků před samotným vykreslováním (po tomto výpočtu se tato textura chová jako běžný rastrový obrázek se všemi jeho výhodami i nevýhodami) nebo je možné textury počítat v reálném čase až při vykreslování – parametry výpočtu se nastaví podle aktuální velikosti a orientace dané plošky, na kterou se textura nanáší. Tuto druhou variantu však OpenGL přímo nepodporuje. Výpočet procedurálních textur je tedy třeba provádět „ručně“.

3D procedurální textury poznáme podle toho, že na hranách je textura „hladce“ a realisticky napojena. Toho je téměř nemožné dosáhnout s použitím 2D textur.



Dále se tedy budeme téměř výhradně zabývat texturami ve formě rastrových obrázků bez ohledu na původ jejich vzniku. Přestože budeme hovořit převážně o 2D texturách, většina prezentovaných vlastností se vztahuje i na 1D a 3D textury. Pokud dojde k odlišnostem, zmíníme je.

Texel

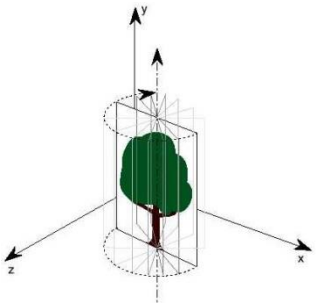
Textura složena z tzv. **texelů**. Pixel a texel mají stejné vlastnosti a velmi podobný či dokonce ekvivalentní způsob uložení v paměti. Přesto budeme tyto pojmy odlišovat. Tedy pixel je prvek vykreslovaný na obrazovce, zatímco texel je rastrový element většinou dvourozměrné textury. Proces texturování pak spočívá v nanášení texelů na vykreslovaný povrch.

Texturu můžeme použít ve všech případech, kdy je nutné vykreslovat tělesa se složitě strukturovanými povrchy, která však nevykazují velké změny v geometrii povrchu (tedy jeho tvaru). Typickým příkladem takového tělesa je zeď z cihel spojených maltou. Samozřejmě že takovou zeď můžeme namodelovat tak, že každou cihlu reprezentujeme kvádrem, ale pokud vykreslujeme „větší“ zeď, rychle roste počet vykreslovaných těles a přitom zbytečně plýtváme pamětí i výpočetním výkonem grafického subsystému. Ještě horší situace nastává například pro koberce, kdy bychom museli například ručně modelovat všechna jeho vlákna.

V této situaci je mnohem výhodnější danou zeď či koberec reprezentovat pouze jednou plochou (např. dva trojúhelníky nebo obdélníkový polygon) a na ni nanést předem vytvořený rastrový obrázek. Připravíme se tak sice o geometrické nerovnosti povrchu (což lze částečně, při vhodném natočení tělesa, simulovat pomocí např. bump-mappingu), které jsou ale významně vyváženy urychlením vykreslování. Při dostatečné velikosti (rozlišení) textury je pak výsledek jak rychlý, tak i kvalitní.

Billboarding

Textury se někdy používají i poněkud jiným způsobem – jejich pomocí se vytváří a následně vykreslují různé složité modely, například stromy. Máme samozřejmě možnost strom namodelovat jako těleso o až několika tisících polygonech nebo je možné vytvořit 2D obrázek stromu z několika směrů a strom vykreslit jako několik vzájemně se protínajících ploch s nanesenou texturou stromu (viz obrázek). V tomto případě je však nutné, aby byla textura v některých místech průhledná. Tato technika se nazývá **billboarding**.



Sprite

V minulosti se často používaly otexturované objekty (většinou obdélníky), které byly k pozorovateli natočeny vždy stejnou stranou. Tyto objekty se nazývají **sprites**. Tato technika byla

využívána hlavně v herním průmyslu (Doom apod.). Sprity jsou podporovány i v OpenGL, kde je vykreslujeme jako vhodně natočené obdélníky s texturou.

Výhody použití rastrových textur

Použití rastrových textur přináší řadu výhod i nevýhod. Jak již bylo řečeno, technika texturování se používá jako určitá náhrada při zobrazování složitých povrchů těles (zeď, omítka, dřevo, kámen, …) kdy se obecně nehomogenní povrch nahradí ploškou s nanesenou texturou. Velmi však záleží na vhodné volbě textury, velikosti objektu a nasvícení scény – to ovlivní, zda je náhrada dostatečná nebo zda uživatel pozoruje vizuální chyby ve scéně.

Velkou výhodou rastrových textur je jejich jednoduchá implementace ve vykreslovacím řetězci. Dnešní grafické akcelerátory navíc umožňují textury komprimovat, podporují tzv. mipmapping (textury ve více rozlišeních – viz dále), antialiasing, multitextury apod. Texturovací jednotka však stále patří k těm částem vizuálního systému, které mají velmi dobrý poměr složitost/vizuální efekt.

Další výhodou je, že pokud u textur použijeme průhlednost (alfa kanál), můžeme vizuálně měnit geometrii objektů (např. objekt se může na některých místech jevit jako děravý). Při použití průhlednosti je ale samozřejmě nutné setřídit průhledné a poloprůhledné plošky, aby Z-buffer pracoval korektně.

Nevýhody použití rastrových textur

Jednou z velkých nevýhod je předem dané rozlišení textur, tedy počet pixelů, ze kterých se textura skládá. Při volbě rozlišení textury se musí zvolit kompromis mezi dvěma extrémy – volba textury s malým rozlišením, která vede k viditelným chybám, nebo textura s neúměrně velkým rozlišením, kdy plýtváme pamětí na grafickém akcelerátoru.

Další nevýhoda se objeví v situaci, kdy zobrazujeme texturu na obrazovce a použije se zvětšení či zmenšení počtu zobrazovaných pixelů. To se stává při texturování téměř vždy, protože se na texturovaný povrch díváme z různých úhlů a vzdáleností. V tomto případě dochází k aliasu, který sice můžeme odstranit pomocí antialiasingu, ale dochází ke zpomalení vykreslování.

Třetí nevýhodou je, že textury zabírají poměrně velké množství paměti. Pokud je v paměti grafického akcelerátoru dostatečně dost místa, je možné textury nahrát do této paměti a vykreslování tím několikanásobně urychlíme. Pokud je však v této paměti místa málo, musí se textury při vykreslování nahrávat z hlavní paměti počítače, což zatěžuje sběrnici či port a hlavně zpomaluje vykreslování. Tuto nevýhodu částečně řeší různé metody komprimace textur.

Textury v OpenGL

OpenGL podporuje pouze rastrové textury, které mohou být 1D, 2D nebo 3D. V případě 1D textur se v podstatě jedná a pás pixelů, pomocí něhož můžeme realizovat různé barevné přechody.

2D textury (rastrové bitmapy a pixmapy) jsou nejpoužívanější a jsou podporovány naprostou většinou grafických akcelerátorů. 3D textury (objemové či voxelové textury) se používají především ve specializovaných aplikacích, kdy je potřeba zobrazovat objemová data (např. scientific visualization). Pro jejich uložení je potřeba poměrně velké množství paměti, která musí být dostatečně rychlá pro čtení.

V OpenGL je možné volit různé filtrace textur, režimy mapování textur na plošky, multitexturing a další grafické efekty, se kterými se seznámíme dále. Před jejich použitím je vhodné si zjistit, zda daný grafický akcelerátor tuto funkci podporuje. Jinak dochází k výraznému zpomalení vykreslování.

Postup při texturování

Vykreslování objektů s nanesenou texturou je poměrně složité na počet a vzájemnou provázanost kroků, které je potřeba v programu provést:

1. Vytvoření rastrové předlohy textury nebo její načtení ze souboru.

2. Vytvoření nového texturovacího objektu, přiřazení textury tomuto objektu a nastavení formátu textury.

3. Vykreslení scény se zadanými texturovacími souřadnicemi pro každý vrchol.

Ad 1: Vytvoření/načtení rastrové předlohy textury

Tento krok musí implementovat sám uživatel, protože OpenGL nenabízí téměř žádnou podporu pro vytvoření (výpočet) rastrových textur ani pro jejich načtení ze souboru.

Poměrně jednoduché řešení spočívá ve využití různých knihoven pro práci s grafickými formáty. Mezi tyto knihovny patří například *libtiff*, *libjpg* nebo *libpng*. Pomocí těchto knihoven je mimo jiné možné načíst soubory s rastrovou grafikou a ty dále použít pro texturu.

Další možností je provedení výpočtu procedurální textury přímo v aplikaci. Tento způsob sice poněkud zdrží start aplikace (pokud se pro výpočet nepoužije jiné vlákno), ale na druhou stranu se textury nemusí ukládat na disk, takže se může zjednodušit distribuce programu. Na internetu lze nalézt velké množství algoritmů pro vytváření procedurálních textur.

Texturu lze také vytvořit přečtením vykresleného obrázku přímo z framebufferu, což je umožněno příkazem *glCopyTexImage2D()*.

Složitejší problém nastane při použití *mipmappingu*, kdy je zapotřebí specifikovat textury v několika rozlišeních. O mipmapách se zmíníme později.

Ad 2: Vytvoření nového texturovacího objektu, přiřazení textury, nastavení formátu textury

Tento krok si můžeme představit na nejjednodušším případě, kdy pro všechna vykreslovaná tělesa používáme jen jednu texturu. V tomto případě musíme pouze nastavit základní parametry textury. Pokud by bylo nutné používat více textur (což je v reálných scénách pravděpodobné), musela by se aktivní textura vybírat podle čísla svého texturovacího objektu (ten se musí pro každou texturu nejprve vytvořit a teprve poté nastavovat formát textury).

OpenGL podporuje celou řadu formátů textur, které se podle dimenze textury dají specifikovat příkazy:

```
void glTexImage1D(GLenum target,
GLint level, GLint internalFormat, Lsizei
width, GLint border, GLenum format, GLenum
type, const GLvoid *pixels)
```

```
void glTexImage2D(GLenum target,
GLint level, GLint internalFormat, GLsizei
width, GLsizei height, GLint border, GLenum
format, GLenum type, const GLvoid *pixels)
```

Význam jednotlivých parametrů je následující:

Parametrem *target* určujeme dimenzi textury, pro jednodimenzionální texturu bude v tomto parametru předána hodnota *GL_TEXTURE_1D*, pro dvoudimenzionální *GL_TEXTURE_2D* atd. Pokud současně nastavíme parametry 1D a 2D textury, mají přednost parametry pro 2D texturu.

Parametr *level* má svůj význam zejména při používání mipmappingu, tj. textur s více úrovněmi detailu (viz dále). Pro běžnou texturu s jedním rozlišením tento parametr nastavujeme na nulovou hodnotu.

Pomocí parametru *internalFormat* se specifikuje vnitřní formát textury, specifikuje počet barevných komponent textury. Nabývá např. hodnot GL_RGB, GL_RGBA, atd. Formátem se zadává počet komponent, jejich typ a bitová hloubka.

Velikost (resp. rozlišení) textury se zadává v parametrech *width* a *height*. Pro 1D texturu se parametr *height* nezadává, protože je vždy jednotkový.

Parametrem *border* se zadává šířka okraje textury v pixelech a musí mít nulovou hodnotu.

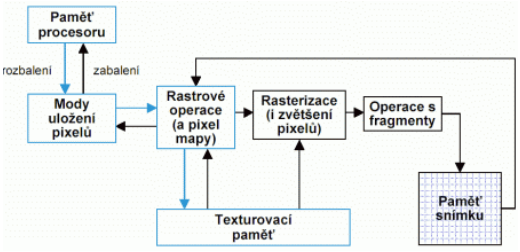
V parametru *format* se předává formát dat použitý pro jednotlivé pixely textury. Každá implementace by měla podporovat minimálně tyto formáty: GL_RED, GL_GREEN, GL_BLUE, GL_RGB, GL_RGBA,…

Parametr *type* udává typ dat každé barevné složky pixelu. Mezi běžně podporované typy patří GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT a GL_FLOAT. Jak je vidět, máme díky parametrům *format* a *type* značnou volnost při tvorbě a/nebo načítání textur ze souboru.

V posledním parametru *pixels* je předáván ukazatel na rastrová data textury, tj. ve většině případů na statické nebo dynamické pole.

Zápis do texturovací paměti

Po zadání příkazu glTexImage*() jsou data textury odeslány do texturovací paměti.

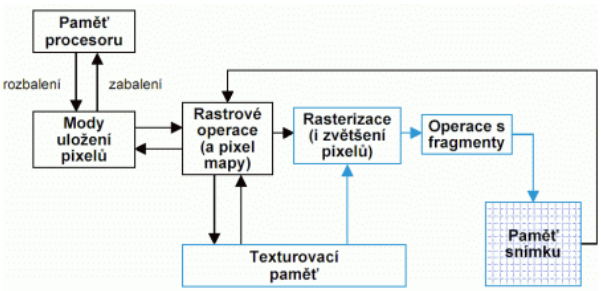


Ad 3: Vykreslení scény s texturami

Vykreslení scény s texturovanými ploškami probíhá podobně jako u barevných plošek nebo osvětlených plošek. Jedinou změnou je, že pro každý vrchol musíme k jeho 2D/3D/4D souřadnicím navíc specifikovat souřadnice v textuře. Každá textura má nezávisle na své velikosti souřadnice v rozsahu od 0.0 do 1.0, tj. 2D textura je chápána jako čtverec o straně délky 1. Souřadnice do textury však můžeme zadávat libovolně, protože díky opakování (pokud je zapnuto) se obrázek textury může na zobrazované ploše šachovnicově skládat.

Souřadnice v textuře se zadávají jako další parametry jednotlivých vrcholů.

Použití texturovací paměti při vykreslování



Transformace textury

Jakmile je textura přiřazena k objektu, může být transformována stejným způsobem, jako samotné objekty – např. rotací, scale, atd. Texturové souřadnice můžeme ještě před samotným mapováním textury vynásobit hodnotami v texturové matici 4x4.

Parametry textury (tj. například způsob, jakým je textura mapována na povrch vykreslovaného tělesa) lze nastavit voláním funkce:

```
void glTexParameter{fi}(GLenum target,
GLenum pname, TYPE value)
```

Alternativně lze také zavolat funkce, které jako svůj poslední parametr akceptují místo skalární hodnoty pole hodnot:

```
void glTexParameter{fi}v(GLenum target,
GLenum pname, const TYPE *values)
```

První parametr (*target*) může podle zvolené dimenze textury nabývat hodnot *GL_TEXTURE_1D* (jednodimenzionální texura, tj. barevný přechod), *GL_TEXTURE_2D* (nejběžnější, rastrová textura), *GL_TEXTURE_3D* (objemová textura) a další. Hodnoty dalších dvou parametrů *pname* a *value* jsou uvedeny v následujícím seznamu:

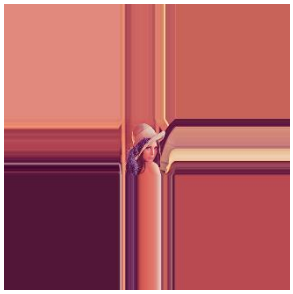
- **Parametr *GL_TEXTURE_WRAP_S*:**
  Určuje, zda se má při překročení rozsahu texturovací souřadnice ve směru osy *s* provést opakování motivu na textuře (*value* = **REPEAT**, **MIRRORED_REPEAT**) nebo protažení první či poslední hodnoty (*value* = **CLAMP_TO_EDGE**, **CLAMP_TO_BORDER**, **MIRROR_CLAMP_TO_EDGE**).
- **Parametr *GL_TEXTURE_WRAP_T*:**
  Tento parametr má podobný význam jako parametr předchozí s tím rozdílem, že se místo na souřadnici ve směru osy *s* vztahuje na souřadnici ve směru osy *t*. U 1D textur nemá hodnota tohoto parametru vliv na zobrazení textury, použitelný je pouze u 2D a 3D textur. Opakování resp. protažení textury lze pro souřadnice ve směrech *s*, *t* a *r* nastavovat nezávisle.
- **Parametr *GL_TEXTURE_WRAP_R*:**
  Tento parametr má opět podobný význam jako předchozí dva parametry, ale vztahuje se na třetí souřadnici *r*, která je použita u objemových textur.

  Význam GL_REPEAT, GL_CLAMP:

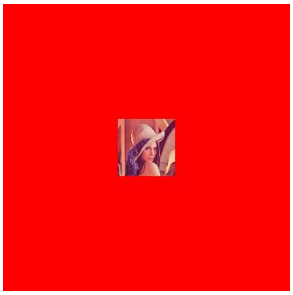
REPEAT

MIRRORED_REPEAT

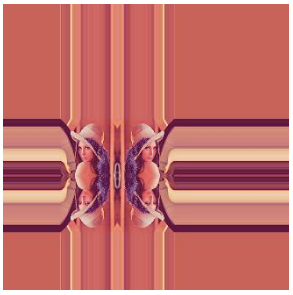
CLAMP_TO_EDGE



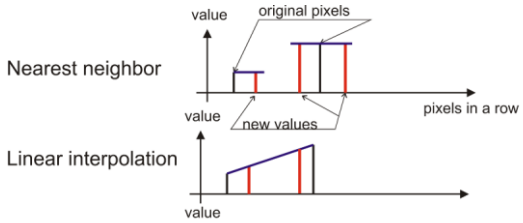
CLAMP_TO_BORDER



MIRROR_CLAMP_TO_EDGE



- **Parametr *GL_TEXTURE_MIN_FILTER*:**
  Tímto parametrem je možné zvolit filtr použitý při zmenšování textury, tj. tehdy, jestliže na plochu jednoho vykreslovaného pixelu musíme použít barvy několika sousedních texelů. Nejjednodušší a nejrychlejší filtr se volí hodnotou *GL_NEAREST*. U tohoto filtru se barva vykreslovaného pixelu vypočte z barvy texelu, jehož souřadnice nejpřesněji odpovídají souřadnicím zadaným do textury. Poněkud sofistikovanější filtr se zadává hodnotou *GL_LINEAR*, kdy se barva vykreslovaného pixelu spočítá pomocí bilineární interpolace z barev sousedních texelů. Další čtyři filtry (*GL_NEAREST_MIP-MAP_NEAREST*,*GL_NEAREST_MIPMAP_LINEAR*, *GL_LINEAR_MIPMAP_NEAREST* a *GL_LINEAR_MIPMAP_LINEAR*) využívají tazkvané *mipmapy*, tj. texturu uloženou ve formátu jakési pyramidy, kde jsou kromě základního rozlišení textury obsažena i rozlišení nižší (vždy s poloviční počtem pixelů v horizontální i vertikální ose) až do nejmenší velikosti 1×1 pixel (viz dále).



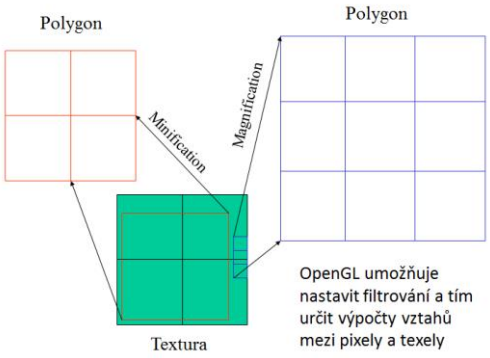
- **Parametr *GL_TEXTURE_MAG_FILTER*:**
  Tímto parametrem se volí filtr použitý při zvětšování textury, tj. v případě, že vykreslovaný pixel obsahuje pouze malou plochu texelu. Možné hodnoty jsou *GL_NEAREST* (použije se nejbližší texel) nebo *GL_LINEAR* (použije se lineární interpolace mezi barvami sousedních texelů).

- **Parametr *GL_TEXTURE_BORDER_COLOR*:**
  Tímto parametrem lze zvolit barvu rámečku okolo textury (pokud je rámeček použit, tj. má nenulovou šířku). V případě zadávání barvy je nutné použít příkazy *glTexParameteriv()* resp. *glTexParameterfv()*, protože barva se zadává jako vektor (pole) čtyř složek R, G, B a A. Implicitní barva rámečku je (0, 0, 0, 0), tedy černá. Jedná se o jediný způsob, jak zadat barvu rámečku.

Princip zvětšení/zmenšení textury:

Magnification – zvětší se význam texelu. Jednomu texelu odpovídá mnoho pixelů. Textura potřebuje být „natažena“ (zvětšena).

Minification – zmenší se význem texelu. Jednomu pixelů odpovídá více texelů.



Implicitní nastavení pro **GL_TEXTURE_MIN_FILTER** je **GL_NEAREST_MIPMAP_LINEAR.** V tomto okamžiku filtr požaduje pro výpočet správného výsledku plnou a konzistentní množinu mipmap. Pokud však není k dispozici, je návratová hodnota (0,0,0,1).
Pokud chceme použít glTexImage2D() bez mipmap, tak musíme nastavit buď:
GL_TEXTURE_MIN_FILTER, GL_LINEAR nebo **GL_TEXTURE_MIN_FILTER, GL_NEAREST**.

Rozšířená podpora formátů textur

Kromě základních formátů textur je možné používat v OpenGL verze 1.2 a vyšších i další „rozšířené“ formáty. Jejich využití má několik výhod. Především jde o to, že použitím některých formátů s malou nebo dostatečnou bitovou hloubkou můžeme ušetřit paměť alokovanou pro textury. Druhou výhodou je možnost použít textury načtené z různých externích souborů, které mají mnohdy „exotický“ formát (např. BGR). V následující tabulce budou ukázány rozšířené formáty textur spolu se základním formátem a typem dat pro každou barevnou složku:

Korespondence mezi rozšířenými a základními formáty	
Rozšířený formát	Základní formát
GL_R3_G3_B2	GL_RGB
GL_RGB4	GL_RGB
GL_RGB5	GL_RGB
GL_RGB8	GL_RGB
GL_RGB10	GL_RGB
GL_RGB12	GL_RGB
GL_RGB16	GL_RGB
GL_RGBA2	GL_RGBA
GL_RGBA4	GL_RGBA
GL_RGB5_A1	GL_RGBA
GL_RGBA8	GL_RGBA
GL_RGB10_A2	GL_RGBA
GL_RGBA12	GL_RGBA
GL_RGBA16	GL_RGBA

Rozšířený formát	Počet bitů na bar. složku					
	R	G	B	A	L	I
GL_R3_G3_B2	3	3	2			
GL_RGB4	4	4	4			
GL_RGB5	5	5	5			
GL_RGB8	8	8	8			
GL_RGB10	10	10	10			
GL_RGB12	12	12	12			
GL_RGB16	16	16	16			
GL_RGBA2	2	2	2	2		
GL_RGBA4	4	4	4	4		
GL_RGB5_A1	5	5	5	1		
GL_RGBA8	8	8	8	8		
GL_RGB10_A2	10	10	10	2		
GL_RGBA12	12	12	12	12		
GL_RGBA16	16	16	16	16		

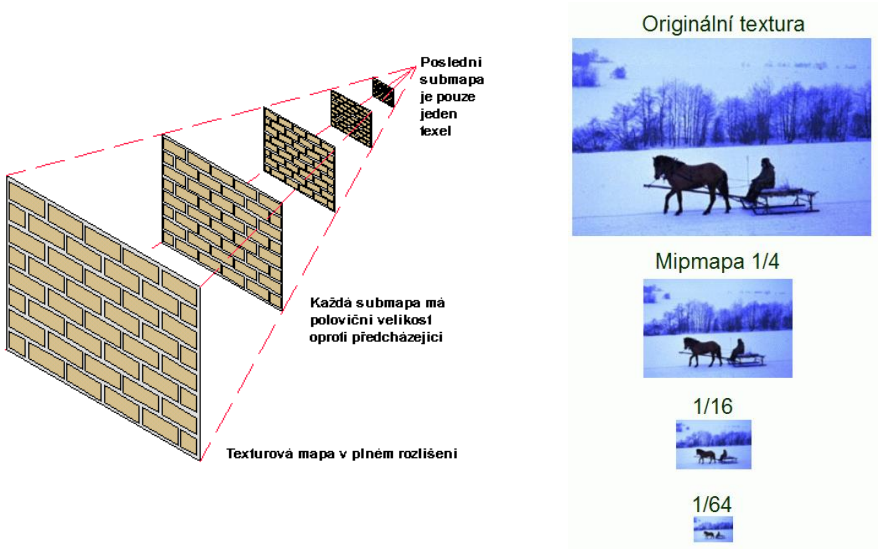
Pro šetření pamětí jsou zajímavé například formáty *GL_R3_G3_B2*, kdy paměť pro jeden texel zabírá pouze jeden byte, nebo *GL_RGB5_A1*, kde každá barevná složka zabírá 5 bitů (celkem je tedy pro barvu vyhrazeno 15 bitů) a alfa složka jeden bit (rozlišujeme tedy pouze průhlednost či neprůhlednost texelu).

Mipmapping

Mipmapping (Multum in Parvo) představuje techniku, která je často používaná pro odstranění či alespoň zmenšení některých vizuálních chyb (resp. nežádoucích rozdílů mezi jednotlivými snímky) vzniklých při pohybu těles s nanesenou texturou v trojrozměrné scéně nebo při pohybu celé scény (tj. změně orientace nebo pozice pozorovatele). Obraz těles s nanesenou texturou se na obrazovce při pohybu zmenšuje, zvětšuje či jinak deformuje, čímž také dochází k nutnosti zvětšování a zmenšování textury při nanášení *texelů* (rastrových elementů textury) na zobrazované *pixely*.

Pro zamezení problikávání se používají textury uložené ve více rozlišeních (tj. úrovních detailu). Při vykreslování otexturovaného povrchu se nejdříve zjistí relativní velikost povrchu vůči celé textuře a poté se vybere vhodné rozlišení textury, která se posléze nanese. Tento postup má nevýhodu v tom, že při postupném zmenšování objektu by docházelo ke skokové změně textury (přešlo by se k menšímu rozlišení textury). Proto se zavádí další úroveň interpolace, kdy se vybere nejbližší větší a nejbližší menší textura a barva pixelů se vypočte interpolací mezi těmito dvěma texturami.

Nyní tedy zbývá pouze volba vhodného rozlišení textur. Z hlediska implementace interpolátoru na grafickém akcelerátoru je nejvýhodnější, aby se rozlišení textury v horizontálním i vertikálním směru snižovalo vždy na polovinu. Počet texelů je v každé následující textuře zmenšen na čtvrtinu až do dosažení textury o velikosti 1×1 pixel.



Mipmapping v OpenGL

Pro nastavení mipmappingu v OpenGL je pro každé nastavované rozlišení textury zapotřebí zavolat již dříve popsanou funkci:

```
void glTexImage2D(GLenum target,
GLint level, GLint components, GLsizei width,
GLsizei height, GLint border, GLenum format,
GLenum type, const GLvoid *pixels)
```

kde se do parametru *level* zadává úroveň textury v hierarchii. Nulou specifikujeme texturu se základním (tj. nejvyšším) rozlišením, jedničkou texturu s rozlišením polovičním atd.
V texturovací paměti grafického akcelerátoru se vytvoří takzvaná *mipmapa*, tj. textura, ve které jsou uloženy všechny velikosti textur rozdělených na jednotlivé barevné složky RGB.

Příklad:

```
glTexImage2D(GL_TEXTURE_2D,0,GL_RGB,32,32,0,GL_RGB,
            GL_UNSIGNED_BYTE,&image32[0][0][0]);//32x32
glTexImage2D(GL_TEXTURE_2D,1,GL_RGB,16,16,0,GL_RGB,
            GL_UNSIGNED_BYTE,&image16[0][0][0]);//16x16
glTexImage2D(GL_TEXTURE_2D,2,GL_RGB,8,8,0,GL_RGB,
            GL_UNSIGNED_BYTE,image8[0][0][0]);  //8x8
glTexImage2D(GL_TEXTURE_2D,3,GL_RGB ,4,4,0,GL_RGB,
            GL_UNSIGNED_BYTE,&image4[0][0][0]); //4x4
glTexImage2D(GL_TEXTURE_2D,4,GL_RGB,2,2,0,GL_RGB,
            GL_UNSIGNED_BYTE,&image2[0][0][0]); //2x2
glTexImage2D(GL_TEXTURE_2D,5,GL_RGB,1,1,0,GL_RGB,
            GL_UNSIGNED_BYTE,&image1[0][0][0]); //1pixel
glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,
              GL_NEAREST);
glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,
             GL_NEAREST_MIPMAP_NEAREST);
```

Automatické generování mipmap

Textury s více úrovněmi detailu můžeme buď vytvořit programově s použitím různých filtrů, nebo je možné použít funkce pro automatické generování mipmap.

```
void glGenerateMipmap( GLenum target );
```

Tato funkce je v určitých situacích výhodná, protože můžeme generovat úrovně mipmapy bez toho, abychom je museli zpracovávat na CPU (například když kreslíme do textury). Způsob výpočtu generované textury závisí na implementaci (tedy na použitém driveru). Výpočet může být velmi rychlý, ale rovněž velmi nepřesný. Proto pokud si můžeme dovolit

vypočítat mipmapy na CPU, je stále lepší vygenerovat je na CPU a každou úroveň mipmapy načíst do GPU zvlášť.

Parametr *target* může být GL_TEXTURE_1D, GL_TEXTURE_2D, GL_TEXTURE_3D, GL_TEXTURE_1D_ARRAY, GL_TEXTURE_2D_ARRAY, GL_TEXTURE_CUBE_MAP, GL_TEXTURE_CUBE_MAP_ARRAY

Zmenšování/zvětšování textur

Při použití mipmappingu je také možné specifikovat další typy filtrů použitých při zmenšování textur. Kromě výběru nejbližšího souseda (*GL_NEAREST*) a interpolace nejbližších sousedů (*GL_LINEAR*) jsou k dispozici i interpolace prováděné mezi dvěma texturami (vybírá se nejbližší menší a nejbližší větší textura):

Při nastaveném filtru ***GL_NEAREST_MIPMAP_NEAREST*** je pro obarvení pixelu vybrán nejbližší texel z nejbližší větší nebo menší textury. Tento filtr poskytuje vizuálně nejhorší výsledky, vykreslování je však nejrychlejší.

GL_NEAREST_MIPMAP_LINEAR – vybere dva nejbližší texely z obou textur a provede mezi nimi lineární interpolaci. Tímto filtrem se dají jednoduše odstranit nepříjemné skokové změny v obraze, které nastávají v případě, že se zobrazované těleso příliš zvětší nebo zmenší a provede se tak výběr jiné dvojice textur z mipmapy.

GL_LINEAR_MIPMAP_NEAREST – provádí billineární interpolaci nejbližších texelů v jedné textuře. Při zmenšování nebo zvětšování tělesa mohou nastat skokové změny ve vykreslované textuře.

GL_LINEAR_MIPMAP_LINEAR – nejprve se použije interpolace pro výpočet barev texelů v obou texturách a poté se výsledná barva spočte další interpolací mezi dvěma předešlými (ve skutečnosti se tedy provádí trilineární interpolace). Tento filtr je sice nejpomalejší, ale poskytuje nejlepší vizuální výsledky.

Texturové objekty

Texturové objekty ukládají texturová data. Tato data jsou pak velmi rychle dostupná. Použití texturových objektů je obvykle nejrychlejším způsobem, jak aplikovat textury. Znovupoužití existující textury je totiž téměř vždy rychlejší než opětovné načtení obrázku textury použitím glTexImage*D().

Pokud chceme pro naše texturová data použít texturové objekty, musíme provést následující kroky:

1. Generování texturových jmen.
2. Počáteční vytvoření a navázání texturových objektů na texturová data.
3. Opětovné navázání texturových objektů, aby byly dostupné pro aktuálně renderované texturové objekty.

Poznámka: Texturové objekty byly do OpenGL zavedeny od verze 1.1.

Pojmenování texturových objektů

Jakákoliv nenulová unsigned integer hodnota může být použita jako jméno texturového objektu. Abychom se vyhnuli opakovanému použití stejných jmen, používáme funkci glGenTextures() poskytující nepoužitá jména pro textury.

```
void glGenTextures(Glsizei n, Gluint *textureNames);
```

Funkce vrátí n aktuálně nepoužívaných jmen pro texturové objekty v poli textureNames.
Jména obsažena v textureNames nemusí být spojitou sadou integer hodnot.
Hodnota 0 je rezervovaná, tudíž nebude nikdy návratovou hodnotou této funkce.
Poté, co je texturový objekt iniciálně navázán (tj. vytvořen), předpokládáme, že při inicializaci OpenGL máme nastavenu dimenzionalitu parametru target – GL_TEXTURE_1D, GL_TEXTURE_2D, GL_TEXTURE_3D,...

```
GLboolean glIsTexture(Gluint textureName);
```

Vytvoření a použití texturových objektů

```
void glBindTexture(Glenum target, Gluint textureName);
```

Funkce glBindTexture() vytvoří a naváže texturové objekty. Když je jméno textury iniciálně navázáno, je vytvořen nový texturový objekt obsahující defaultní hodnoty pro obrázek textury a jeho vlastnosti. Následné volání glTexImage*(), glTexSubImage*(), glCopyTexImage*()glCopyTexSubImage*(), glTexParameter*() a glPrioritizeTextures() ukládá data do texturových objektů. Texturové objekty mohou obsahovat obrázek textury s případnými asociovanými mipmap obrázky, včetně informací o šířce, výšce, šířce ohraničení, vnitřním formátu, rozlišení komponent a vlastností textury. Tyto vlastnosti zahrnují minification a magnification filtry, barvu ohraničení, prioritu textur atd.
Pokud je daný texturový objekt někdy znovu navázán, jeho data se stanou aktuálním stavem textury (stav předtím použité textury je nahrazen).

Funkce glBindTexture() má za úkol následující věci:

- Pokud navazujeme na dříve vytvořený texturový objekt, tento texturový objekt se stane aktivním.
- Pokud navazujeme na texturový objekt s hodnotou textureName 0, OpenGL přestane používat texturové objekty a vrací se k nepojmenované defaultní textuře.

Odstranění texturových objektů

Protože jsem texturové zdroje omezeny, musíme je občas uvolnit. Toho dosáhneme smazáním nepoužívaných textur.

```
void glDeleteTextures(Glsizei n, const Gluint *textureNames);
```

Funkce smaže *n* texturových objektů, jejichž jména jsou uložena v poli *textureNames*. Uvolněná jména textury mohou být následně použita znovu (např. pomocí **glGenTextures()**). Pokud je smazána aktuálně používaná textura, navázání se vrací k defaultní textuře. To odpovídá stejnému stavu, do jakého se dostaneme použitím **glBindTexture()** s hodnotou *textureName* 0. Pokus smazat texturu s neexistujícím jménem nebo texturu s *textureName* 0 je ignorován a není generována žádná chyba.

Příklad použití texturových objektů:

```
glGenTextures(2, texName);
glBindTexture(GL_TEXTURE_2D, texName[0]);
glTexParameteri( ...
...
glTexImage2D(...
glBindTexture(GL_TEXTURE_2D, texName[1]);
...
void display(void)
{
  glBindTexture(GL_TEXTURE_2D, texName[0]);
  ...
}
```

Další funkce

```
void glTexSubImage2D(GLenum target, GLint
level, GLint xoffset, GLint yoffset, GLsizei
width, GLsizei height, GLenum format, GLenum
type, const GLvoid *texels);
```

Funkce glTexSubImage2D definuje 2D texturový obrázek, který nahrazuje celou oblast nebo část spojité podoblasti (ve 2D jednoduše obdélník) aktuálního 2D texturového obrázku.
Parametr target musí být nastaven na hodnotu GL_TEXTURE_2D, ...
Parametry level, format a type jsou podobné parametrům funkce glTexImage2D().

```
void glCopyTexImage2D(GLenum target, GLint
level, GLint internalFormat, GLint x, GLint
y, GLsizei width, GLsizei height, GLint
border);
```

Funkce vytvoří 2D texturu, přičemž pro definici texelů využívá data z framebufferu.
Parametr target musí být nastaven na GL_TEXTURE_2D,... Parametry level, internalFormat a border mají stejný efekt jako ve funkci glTexImage2D().

```
void glCopyTexSubImage2D(GLenum target,
GLint level, GLint xoffset, GLint yoffset,
GLint x, GLint y, GLsizei width, GLsizei
height);
```

Funkce použije obrazová data z framebufferu pro nahrazení celé oblasti nebo spojitého podregionu aktuálního 2D texturového obrázku.
Parametr target musí být nastaven na GL_TEXTURE_2D,... Level je hodnota LOD pro mipmapu. Xoffset a yoffset určují offset texelu v ose x a y (dolní levý roh textury má hodnotu (0, 0)). Width a height určují velikost podregionu.

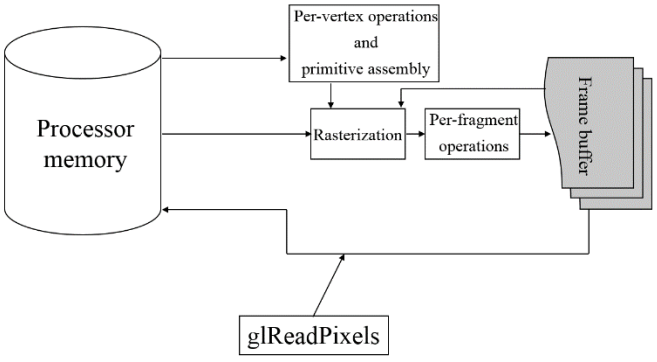
Obrázky

Dosud jsme se zabývali pouze vykreslováním geometrických dat – bodů, čar a polygonů. Pomocí OpenGL je ale možné vykreslovat i obrázky.
Obrázek (image) je matice sestávající z pixelů. S obrázky pracujeme prostřednictvím textur. Každý pixel může obsahovat kompletní (R, G, B, A) barvu. Obrázky běžně pocházejí z barevných bufferů.

```
void glReadPixels(GLint x,GLint y,
GLsizei w, GLsizei h, GLenum format,
GLenum type, GLvoid *dta)
```

Funkce glReadPixels() načte obdélníkové pole pixelů z framebufferu. Levý dolní roh tohoto pole je na pozici (x, y) a má rozměry definované parametry w a h. Tato data se uloží do pole, jehož ukazatel je posledním parametrem funkce, tedy dta. *format* může nabývat hodnot GL_STENCIL_INDEX, GL_DEPTH_COMPONENT, GL_DEPTH_STENCIL,GL_RED, GL_GREEN, GL_BLUE, GL_RGB, GL_BGR, GL_RGBA,GL_BGRA.

type může nabývat hodnot
GL_UNSIGNED_BYTE, GL_BYTE, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, GL_HALF_FLOAT, GL_FLOAT atd.



Módy uložení obrázku

Obrázek uložený v paměti má pro každý pixel možnost uložit jednu až čtyři „porce“ dat, nazývané elementy. Data se mohou skládat pouze z indexu barvy nebo jasu (luminance), který je určen jako vážená suma RGB hodnot, nebo mohou data obsahovat R, G, B, A

komponenty pro každý pixel. Možná rozmístění pixelových dat nebo jejich formát určují počet elementů, které budou pro každý pixel uložena, včetně jejich uspořádání.

Elementy mohou být v paměti uloženy jako různé datové typy – od 8-bitových hodnot typu byte po 32-bitové integery.

Funkce glPixelStore() slouží k nastavení módu pro uložení pixelů, které ovlivňuje operace glReadPixels(), glTexImage*D() a některé další.

Parameter Name	Type	Initial Value	Valid Range
GL_UNPACK_SWAP_BYTES, GL_PACK_SWAP_BYTES	GLboolean	FALSE	TRUE/FALSE
GL_UNPACK_LSB_FIRST, GL_PACK_LSB_FIRST	GLboolean	FALSE	TRUE/FALSE
GL_UNPACK_ROW_LENGTH, GL_PACK_ROW_LENGTH	GLint	0	any nonnegative integer
GL_UNPACK_SKIP_ROWS, GL_PACK_SKIP_ROWS	GLint	0	any nonnegative integer
GL_UNPACK_SKIP_PIXELS, GL_PACK_SKIP_PIXELS	GLint	0	any nonnegative integer
GL_UNPACK_ALIGNMENT, GL_PACK_ALIGNMENT	GLint	4	1, 2, 4, 8