PV112 Programování grafických aplikací

Jaro 2017

Výukový materiál

---

8. přednáška: Uniform buffer objekt, redukce počtu glDraw*, multisampling

Uniform buffer objekt

Než začneme mluvit o uniform buffer objektech, musíme si popsat, co to je tzv. **interface blok**. Interface blok nám umožňuje seskupit několik proměnných dohromady a pracovat s nimi dohromady. Používá se pro seskupení vstupních a výstupních proměnných shaderu a pro seskupení uniform proměnných.

Syntaxe interface bloku je následující. *qualifier* je in/out/uniform, *block_name* je název interface bloku. Ve složených závorkách uvádíme jednotlivé proměnné bloku a za blokem můžeme uvést nepovinně jméno instance, kterou se budeme odkazovat na položky bloku.

Syntaxe:

```
qualifier block_name
{
   members
   ...
   ...
   ...
} [instance_name];
```

Příklad:

```
uniform LightData
{
   vec4 position;
   vec3 ambient;
   vec3 diffuse;
   vec3 specular;
} light;
```

Instance name je nepovinná. Pokud ji neuvedeme, pracujeme s proměnnými přímo, jsou to jakoby globální proměnné. To také znamená, že nesmí existovat jiná globální proměnná, která by se jmenovala stejně. Pokud *instance name* uvedeme, přistupujeme k proměnným přes tuto *instance name*.

```
out VertexData
{
   vec3 pos_ws;
};
...
pos_ws = M_matrix * pos;
```

```
out VertexData
{
   vec3 pos_ws;
} output;
...
output.pos_ws = M_matrix * pos;
```

Jak již bylo řečeno, interface blok je možné použít pro vstupní a výstupní proměnné shaderu, ale pouze pro proměnné mezi shadery. Nelze je použít pro vstup vertex shaderu nebo výstup fragment shaderu.

Je vyžadováno, aby daný interface blok měl stejný název (*block_name*) a stejně definované proměnné. Nemusí mít ovšem stejně pojmenované *instance_name*. To nám umožňuje alespoň trochu odlišit jménem výstupní proměnné vertex shaderu a vstupní proměnné fragment shaderu (například jako outData a inData).

Pokud budeme používat pouze vertex a fragment shadery, tak této výhody asi příliš nevyužijeme. Velkou výhodu představují interface bloky, pokud používáme i ostatní druhy shaderů (geometry shader a teselační shadery), ty si ovšem v tomto kurzu představovat nebudeme. Způsob podobný interface blokům také používá Direct3D (resp. HLSL).

Další možností, kde použít interface bloky, je seskupit uniformní proměnné. Takovýto blok se liší především tím, že k takto seskupeným proměnným nepřistupujeme přímo, ale přes buffer.

```
uniform LightData
{
   vec4 position;
   vec3 ambient;
   vec3 diffuse;
   vec3 specular;
} light;
```

Data proměnných jsou uložena v bufferu (uniform buffer objektu – UBO).

Uniform buffer objekty

Uniform buffer objekty jsou buffery jako každé jiné (setkali jsme se už s GL_ARRAY_BUFFER a GL_ELEMENT_ARRAY_BUFFER), používá se pro ně konstanta GL_UNIFORM_BUFFER. Můžeme pro ně použít stejné funkce jako pro jiné buffery, tedy *glGenBuffers*, *glBufferData*, *glMapBuffer* atd.

```
GLuint light_ubo;
glGenBuffers(1, &light_ubo);
glBindBuffer(GL_UNIFORM_BUFFER, light_ubo);
glBufferData(GL_UNIFORM_BUFFER, ...);
...
glBufferSubData(...);
```

Abychom mohli buffer svázat s interface blokem v shaderu, musíme nejprve získat index tohoto bloku. K tomu slouží funkce *glGetUniformBlockIndex*, které kromě programu zadáváme jméno bloku, na který se dotazujeme. Pozor, předáváme jméno bloku (block_name), nikoliv jméno instance.

```
GLuint glGetUniformBlockIndex(GLuint program, const char *name)
```

Funkce vrací speciální konstantu GL_INVALID_INDEX, pokud zadaný blok neexistuje, nebo není aktivní (není v shaderu použit).

Uvedeme si několik příkladů na to, na co se dotazovat a co získáme. Vždy se dotazujeme na block_name, což je LightData, nikdy ne na jméno instance, která může nebo nemusí být zadána. Pokud je instance zadána jako pole, vytvoříme tak několik uniform bloků, které obsahují data v separátních uniform buffer objektech. Indexy těchto bloků získáváme stále stejně, tedy tak, že použijeme jméno bloku. Funkce *glGetUniformBlockIndex* nám vrátí index prvního prvku pole, následující prvky mají indexy vždy o jedna větší.

Nyní víme, jak vytvořit uniform buffer a víme, jak získat index bloku, ke kterému bychom chtěli tento vytvořený buffer navázat. Buffery se ovšem nepřiřazují k indexům přímo, ale přes „binding point“ (někdy se to označuje i jako „binding slot“). Ke svázání binding slotu, ke kterému vážeme buffer, a indexu bloku, který jsme získali ze shaderu, slouží funkce *glUniformBlockBinding*.

```
void glUniformBlockBinding(GLuint program,
  GLuint uniformBlockIndex,
  GLuint uniformBlockBinding)
```

Tento postup má jednu velkou výhodu. Je možné takto zadat, že všechny bloky ve všech shaderech, které obsahují data osvětlení, mají brát data z uniform bufferu navázaného na binding point 0. Při kreslení potom navážeme příslušný buffer na tento binding point jen jednou a nemusíme to měnit při kreslení každého objektu.

Pro navázání bufferu na daný binding point slouží funkce *glBindBufferBase*. Je to funkce podobná funkci *glBindBuffer*, která také navazuje buffer, ale tato funkce neumí specifikovat binding index, pro to musíme použít speciální funkci.

```
void glBindBufferBase(GLenum target,
  GLuint index, GLuint buffer)
```

Po navázání bufferu funkcí *glBindBufferBase* se tento buffer také zpřístupňuje funkcím, které pracují s aktuálně navázaným bufferem, jako jsou funkce *glBufferData*, *glMapBuffer* apod. V tomto ohledu duplikuje funkci jednodušší *glBindBuffer*.

Na následujícím jednoduchém příkladu vidíme použití a výhody UBO. Po načtení a kompilaci shaderů získáme index bloku „LightData“ ve všech shader programech, které máme, a svážeme všechny indexy s bodem 0. To stačí provést jen jednou. Poté při kreslení funkcí *glBindBufferBase* navážeme uniform buffer light_ubo na bod 0 a kreslíme postupně všechny objekty všemi shadery. Vzhledem k tomu, že všechny shadery získávají data z uniform bufferu ze stejného binding point (bodu 0), není nutné uniform buffer znovu nastavovat.

Obdobně kdybychom aktualizovali data světel, stačí nám aktualizovat data v uniform bufferu a nemusíme aktualizovat data uniform proměnných každého shaderu zvlášť.

Toto představuje velkou výhodu použití UBO. Při dobrém rozvržení dat (například blok s maticemi kamery, blok s maticemi objektu, blok s daty světel, blok s daty materiálu) se

značně zpřehledňuje a zjednodušuje kód. Některá data stačí nastavit jen jednou na začátku vykreslování (kamera, světla), a ta, která je nutné nastavit s každým objektem zvlášť (matice objektu, materiál), stačí navázat jedním příkazem. Taktéž aktualizace dat se zjednoduší – stačí nám aktualizovat data bufferu a nemusíme nastavovat uniformní proměnné všem shader programům.

```
LD_loc1 = glGetUniformBlockIndex(program1, "LightData");
LD_loc2 = glGetUniformBlockIndex(program2, "LightData");
LD_loc3 = glGetUniformBlockIndex(program3, "LightData");
glUniformBlockBinding(program1, LD_loc1, 0);
glUniformBlockBinding(program2, LD_loc2, 0);
glUniformBlockBinding(program3, LD_loc3, 0);
...
glBindBufferBase(GL_UNIFORM_BUFFER, 0, light_ubo);
glUseProgram(program1);
glDraw*
glUseProgram(program2);
glDraw*
glUseProgram(program3);
glDraw*
```

Pro úplnost dodejme, že je možné navázat i část bufferu pomocí funkce *glBindBufferRange*. Můžeme tak mít data pro několik uniform bloků ve stejném uniform bufferu.

```
void glBindBufferRange(GLenum target,
  GLuint index, GLuint buffer,
  GLintptr offset, GLsizeiptr size)
```

Při práci s uniform bloky a s UBO musíme mít na paměti jednu důležitou věc: data musí být v bufferu umístěna přesně tak, jak je bude OpenGL v shaderu číst. Problém je v tom, že OpenGL a náš programovací jazyk může mít (a často má) jiné požadavky na zarovnání dat. Vzhledem k tomu, že jednou z priorit OpenGL a shaderů je to, aby výpočet běžel co nejrychleji (příp. neběžel zbytečně pomalu), OpenGL zarovnává data tak, aby s jimi hardware lépe pracoval.

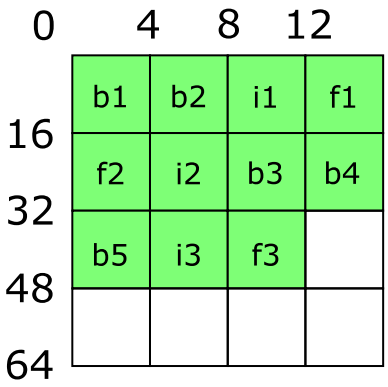
OpenGL definuje několik způsobů, jakým zarovnává data, a je na nás, který zvolíme. Vybraný způsob se definuje pomocí vlastnosti layout. My si zde podrobněji popíšeme pouze způsob označený jako std140, protože se s ním nejjednodušeji pracuje.

```
layout (std140) uniform LightData
{
   ...
} light;
```

Pravidla pro layout std140 vysvětlíme na jednoduchých příkladech. Začneme základními typy a poté přejdeme na složitější typy. U všech typů si musíme dávat pozor na velikost typů a na zarovnání. Nejjednodušší jsou základní datové typy bool, int, uint a float. Tyto typy mají velikost 4 byty a jsou zarovnávány také na 4 byty. Pro následující uniform blok je rozložení

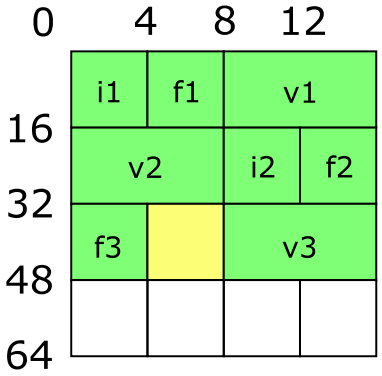
v paměti ilustrováno na obrázku. Zde jednotlivé čtverečky znázorňují 4B paměti a data v paměti jdou po řádcích.

```
layout (std140) uniform U
{
   bool b1, b2;
   int i1;
   float f1, f2;
   int i2;
   bool b3, b4, b5;
   int i3;
   float f3;
};
```



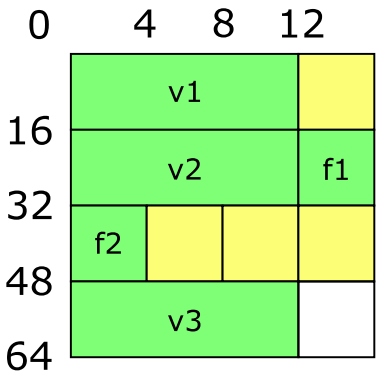
Vektor dvou čísel (int, float, apod.) má velikost 8 bytů, ale je nutné ho také zarovnat na 8 bytů. Na ilustraci vidíme, že za proměnnou f3 budou 4 byty nevyužité, protože vektor v3 musí být zarovnán.

```
layout (std140) uniform U
{
   int i1;
   float f1;
   vec2 v1, v2;
   int i2;
   float f2, f3;
   vec2 v3;
};
```



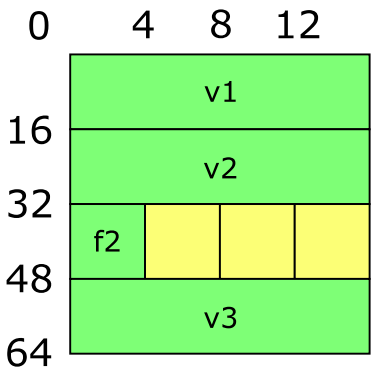
Vektor tří čísel má velikost 12 bytů, ale je nutné ho zarovnat na 16 bytů (!). Na ilustraci vidíme jednak prázdné místo za vektorem v1 a za proměnnou f2, které nebylo využito, protože následující vektor musel být zarovnán. Místo za vektorem v2 ale využito být mohlo, protože proměnná f1 může být umístěna na tomto místě.

```
layout (std140) uniform U
{
   vec3 v1, v2;
   float f1, f2;
   vec3 v3;
};
```



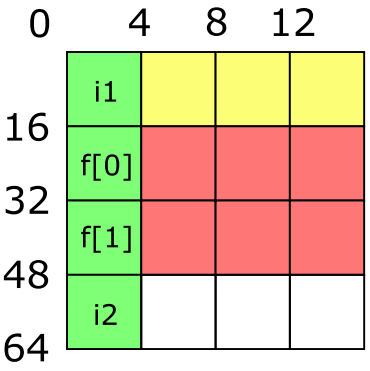
Vektor 4 čísel má velikost i zarovnání 16 bytů. Na našem příkladu se to projeví tak, že za proměnnou f2 je 12 bytů nevyužitých, kvůli zarovnání.

```
layout (std140) uniform U
{
   vec4 v1, v2;
   float f2;
   vec4 v3;
};
```



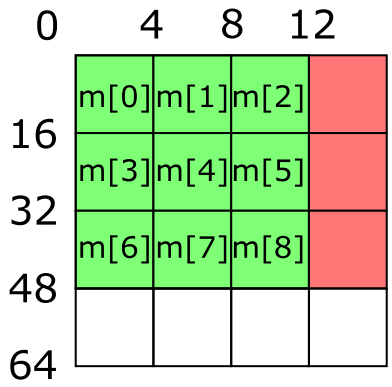
Práce s poli je už poněkud zajímavější. Zarovnání pole je 16 bytů. Velikost jednoho prvku pole je ale zaokrouhlena na nejbližší vyšší násobek 16, což zejména znamená, že třeba velikost jednoho prvku v poli floatů je 16 bytů, nikoliv pouze 4 byty. Toto místo nelze využít, jako to bylo možné u vec3. Na příkladu vidíme, že 12 bytů za i1 bychom mohli využít, nejsou ale využity kvůli zarovnání na 16 bytů, které potřebuje pole f. Za jednotlivými prvky pole f je vždy 12 bytů, které ovšem nemůžeme využít, jsou jakoby součástí pole f.

```
layout (std140) uniform U
{
   int i1;
   float f[2];
   int i2;
};
```



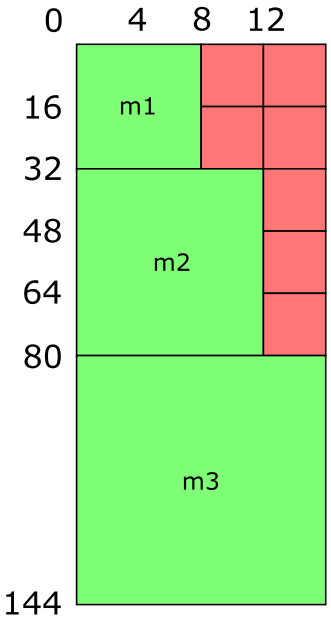
Matice se chová stejně, jako by to bylo pole vektorů. Pořadí elementů v matici je stejné jako pořadí, které bychom potřebovali, kdybychom při volání funkce *glUniformMatrix** použili hodnotu GL_FALSE u parametru transposed. Jinými slovy, nejprve máme v paměti první sloupec, poté druhý sloupec atd.

```
layout (std140) uniform U
{
   mat3 m;
};
```



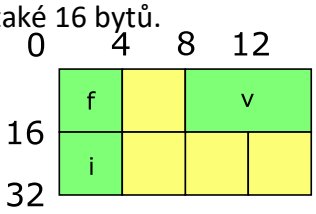
Zde vidíme, jak vypadá uspořádání paměti pro mat2, mat3 a mat4.

```
layout (std140) uniform U
{
   mat2 m1;
   mat3 m2;
   mat4 m3;
};
```

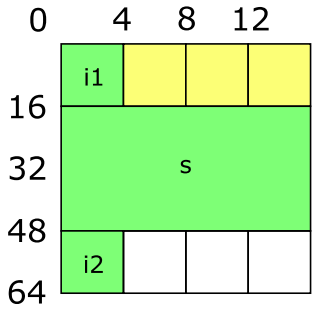


A na závěr struktura. Prvky struktury se seskupují stejně, jako by to byl samostatný blok (začínající od 0 bytů). Velikost celé struktury je pak zaokrouhlena na nejbližší vyšší násobek 16, podobně jako u pole. Zarovnání struktury je také 16 bytů.

```
struct S
{
   float f;
   vec2 v;
   int i;
};
```




```
layout (std140) uniform U
{
   int i1;
   S s;
   int i2;
};
```



Layout std140 není složitý, ale může být zdrojem velmi těžko nalezitelných chyb. Proto jedna rada na závěr, dokud si nejste jistí, jak se proměnné zarovnávají, používejte pro všechny proměnné typy vec4 a mat4, ty se chovají „nejpřirozeněji“. Sice budete plýtvat pamětí, ale pro začátek to není špatný krok (ale jen pro začátek!).

Atribut layout se původně používal pouze pro specifikaci rozložení dat, postupem času se ovšem jeho možnosti rozšířily. Uvedeme si tady z nich několik asi nejužitečnějších.

Od OpenGL 3.3 je možné (a velmi užitečné) pomocí layout(location = N) specifikovat index vstupních proměnných vertex shaderu a výstupních proměnných fragment shaderu. Takto si

zajistíme, že tyto proměnné mají vždy stejný index i bez toho, abychom museli volat *glBindAttribLocation*/*glBindFragDataLocation*.

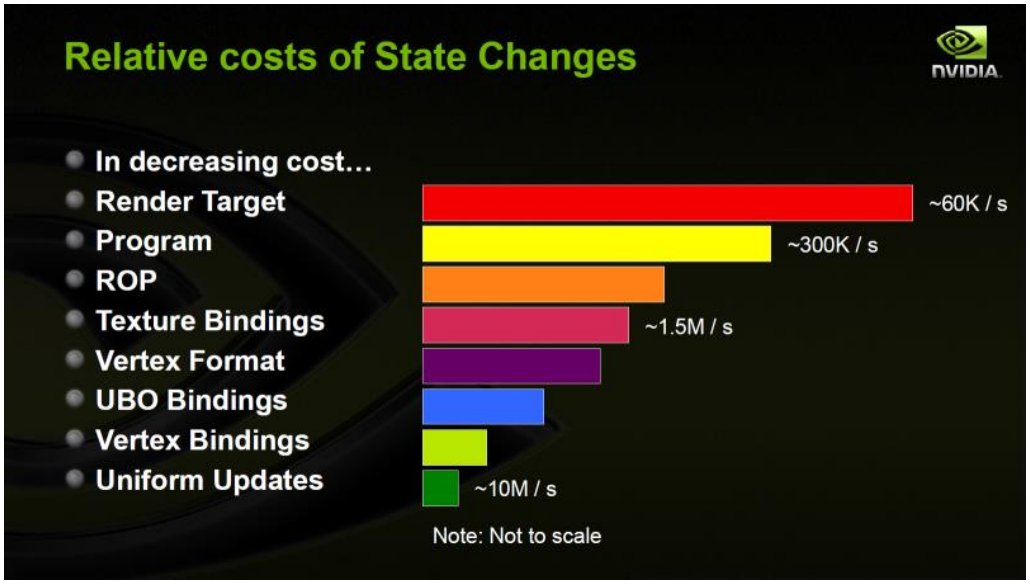
Od OpenGL 4.2 je také možné pomocí layout specifikovat binding point uniform bloku, a tím si ušetříme volání funkcí *glGetUniformBlockIndex* a *glUniformBlockBinding*.

Od OpenGL 4.3 je také možné specifikovat index samostatných uniform proměnných mimo uniform blok. Nicméně tento postup má malou nevýhodu. OpenGL nedovoluje takto použít stejný index pro dvě uniformní proměnné, ani když ty proměnné mají stejný typ a jméno. Z tohoto důvodu není možné určit index uniformních proměnných používaných ve vertex shaderu a fragment shaderu zároveň, v takovém případě musíme nechat OpenGL, aby zvolilo index samo.

Urychlování vykreslování

V následující části se budeme zabývat tím, jak urychlit vykreslování scény v OpenGL.

Rychlost vykreslování v OpenGL nezávisí pouze na tom, jak máme složitou scénu, geometrie či shadery, ale také závisí hodně na tom, jak OpenGL využíváme. Mezi nejnáročnější operace v OpenGL patří změny stavů a volání kreslících funkcí. I jednotlivé ceny různých změn stavů se liší, jak lze vidět na následujícím obrázku (pro jistotu „přeložíme“ některé zkratky: Render Target jsou změny framebuffer objektů, ROP jsou změny míchání barev, testu hloubky, stencil testu apod., a vertex format a vertex binding jsou oba ukryty ve VAO).



My se nebudeme tolik zabývat tím, jak ušetřit změny stavů, ty lze ušetřit výhradně lepším návrhem aplikace. My se ale zaměříme na to, jak zmenšit počet volání vykreslovacích funkcí.

S každým voláním *glDraw** musí OpenGL zkontrolovat mnoho stavů, což znamená nějaký overhead. Je proto lepší vykreslit 1 milion trojúhelníků pomocí jednoho volání *glDraw**, než

kreslit 1000 x 1000 trojúhelníků. My se zde podíváme zejména na instancování a poté také na další techniky, kterými lze seskupit vykreslovací příkazy dohromady.

Nejprve tedy instancování. Instancování je technika, kdy najednou kreslíme stovky téměř totožných objektů, které se liší jen v několika drobnostech (pozici, barvě apod.).

Pro instancované kreslení máme funkce *glDrawArraysInstanced* a *glDrawElementsInstanced*, které fungují stejně jako jejich ne-instanced protějšky, ale mají navíc parametr primcount, kterým zadáme počet instancí, které chceme vykreslit.

```
void glDrawArraysInstanced(GLenum mode,
   GLint first, GLsizei count, GLsizei primcount)
```

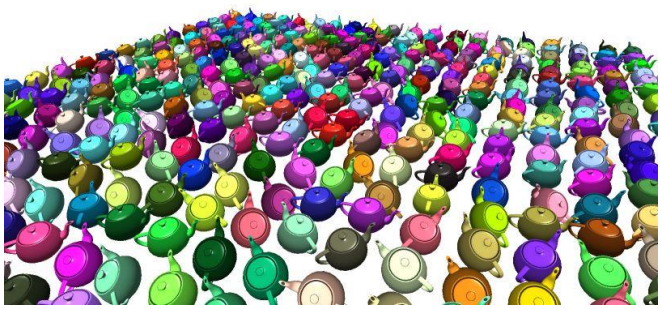
```
void glDrawElementsInstanced(GLenum mode,
   GLsizei count, GLenum type, const void *indices, GLsizei
primcount)
```

Když kreslíme instancovaně, máme ve vertex shaderu k dispozici proměnnou gl_InstanceID, která obsahuje index instance, kterou vertex shader zrovna zpracovává. Tento index ve vertex shaderu použijeme k tomu, abychom získali data specifická pro onu instanci, například z pole dat uložených v UBO.

Nyní si uvedeme příklad takového vertex shaderu. Vertex shader obsahuje uniform blok ObjectData, který obsahuje data všech instancí, a ve funkci *main* používá gl_InstanceID pro to, aby získal data dané instance.

```
uniform ObjectData
{
   mat4 model_matrices[400];
   vec4 model_colors[400];
};
out vec4 color;
void main()
{
   ...
   mat4 model_matrix = model_matrices[gl_InstanceID];
   color = model_colors[gl_InstanceID];
   ...
}
```

Výsledekem je 400 objektů, každý s jinou barvou a jinou pozicí, ale vykresleny jedním voláním *glDraw**.



Existuje ještě druhá možnost, jak poslat data instance, a to přes vstupní proměnné vertex shaderu. Takový vertex shader bude mít mezi svými vstupními proměnnými i modelovou matici objektu a barvu objektu.

```
in mat4 model_matrix;
in vec4 model_color;
out vec4 color;
void main()
{
   ...
   // transform object using the model_matrix
   color = model_color;
   ...
}
```

Jak se s takovými vstupními proměnnými pracuje? Stejně jako s jakýmikoliv jinými: získáme si jejich index, navážeme příslušný GL_ARRAY_BUFFER, povolíme atribut a použijeme glVertexAttribPointer pro nastavení dat. Jediná věc navíc, kterou musíme udělat, je zavolat funkci *glVertexAttribDivisor*. Touto funkcí řekneme, že data pro daný vstupní atribut nejsou určena pro každý vrchol, ale pro každou instanci.

```
void glVertexAttribDivisor(GLuint index, GLuint divisor)
```

Funkce *glVertexAttribDivisor* změní stav aktuálně navázaného VAO, je tedy možné ho zavolat jen jednou při zadávání parametrů VAO.

A pro úplnost, jak zadáváme vstupní proměnnou typu matice? Matice předáváme jako několik vektorů, které odpovídají sloupcům matice. Index, který získáme funkcí *glGetAttribLocation*, je index, který odpovídá indexu prvního vektoru. Další vektory mají index vždy o 1 větší.

Následuje část kódu, která ukazuje, jak se nastavuje taková matice. Musíme povolit index všech atributu model_matrix_loc – model_matrix_loc+3, nastavit data všech atributů (zde můžeme využít parametrů stride a offset), a funkcí *glVertexAttribDivisor* řekneme, že se jedná o data instancí.

```
glBindBuffer(GL_ARRAY_BUFFER, model_matrices_vbo);
glEnableVertexAttribArray(model_matrix_loc);
glEnableVertexAttribArray(model_matrix_loc+1);
glEnableVertexAttribArray(model_matrix_loc+2);
glEnableVertexAttribArray(model_matrix_loc+3);
glVertexAttribPointer(model_matrix_loc, 4, GL_FLOAT,
   GL_FALSE, 16*sizeof(float), nullptr);
glVertexAttribPointer(model_matrix_loc+1, 4, GL_FLOAT,
   GL_FALSE, 16*sizeof(float), (void *)(4*sizeof(float)));
glVertexAttribPointer(model_matrix_loc+2, 4, GL_FLOAT,
   GL_FALSE, 16*sizeof(float), (void *)(8*sizeof(float)));
glVertexAttribPointer(model_matrix_loc+3, 4, GL_FLOAT,
   GL_FALSE, 16*sizeof(float), (void *)(12*sizeof(float)));
glVertexAttribDivisor(model_matrix_loc, 1);
glVertexAttribDivisor(model_matrix_loc+1, 1);
glVertexAttribDivisor(model_matrix_loc+2, 1);
glVertexAttribDivisor(model_matrix_loc+3, 1);
```

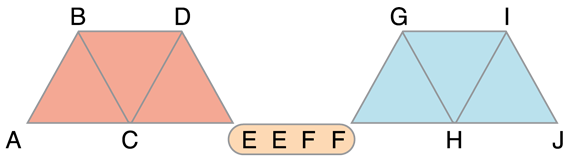
A kde lze instancování najít? Zde například bylo instancování použito pro kreslení publika ve hře FIFA.

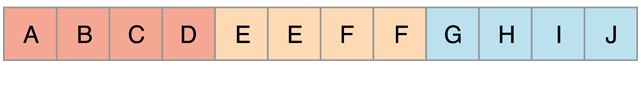
Instancování lze také použít na kreslení terénu – trávy, stromů, keřů apod. Instancování je také možné použít i pro mnohem menší objekty, například částice částicového systému, nebo jednotlivé znaky textu.

Další techniky

Nyní stručně zmíníme další používané techniky pro zmenšení počtu volání. První z nich jsou **degenerované trojúhelníky**. Tato technika využívá toho, že OpenGL nekreslí trojúhelníky s nulovým obsahem (ani jako úsečky), a tak je možné je využít pro spojení dvou trojúhelníkových pásů vložením dvou správných vrcholů do seznamu indexů.

Kde toho lze využít? Příkladem můžou být objekty, které jsou složeny z několika pásů trojúhelníků, například teselovaný terén.





Další technikou, která řeší prakticky totéž, je **primitive restart**. Tímto způsobem je možné vložit speciální index do seznamu indexů, který je interpretován jako konec jednoho primitiva a začátek druhého primitiva, jako bychom zavolali další *glDraw**.

Pokud toto chceme použít, musíme to povolit pomocí *glEnable(GL_PRIMITIVE_RESTART)* a nastavit index pomocí *glPrimitiveRestartIndex*. Nejčastěji se jako index používá nejvyšší možná hodnota (0xffff pro unsigned short a 0xffffffff pro unsigned int). Direct3D a Vulkan například ani jinou možnost nedávají.

Další možností je použít funkce ***glMultiDrawArrays*** a ***glMultiDrawElements***. Tyto funkce jsou opět podobné svým ne-multi protějškům, pouze namísto parametrů first/count a count/indices chtějí pole těchto parametrů o velikosti drawcount. Fungují stejně jako bychom několikrát zavolali funkce *glDrawArrays*/*glDrawElements*. Všimněte si, že parametr mode není pole, neboli tyto funkce kreslí vždy stejný druh primitiv.

```
void glMultiDrawArrays(GLenum mode, const GLint
*first, const GLsizei *count, GLsizei drawcount)
```

```
void glMultiDrawElements(GLenum mode, const GLsizei *count, GLenum type,
 const GLvoid * const * indices, GLsizei drawcount);
```

Poslední technikou, kterou zmíníme, spočívá ve funkci *glDrawElementsBaseVertex*. Tato funkce funguje stejně jako *glDrawElements*, jen ke všem zpracovávaným indexům přičte hodnotu basevertex.

Tato technika neslouží ke snížení počtu volání *glDraw**, ale pomáhá, pokud máme více geometrií ve stejných bufferech (a snižujeme tak množství nutných přepnutí bufferů). Pokud data geometrií jen jednoduše zkopírujeme za sebe, musíme čelit problému, že indexy druhé geometrie (a i dalších) začínají od 0, ale tím referencují na data první geometrie. S využitím basevertex přičteme ke všem indexům druhé geometrie počet vrcholů předešlých geometrií, a tak nemusíme měnit ani data geometrie, ani navázané buffery.

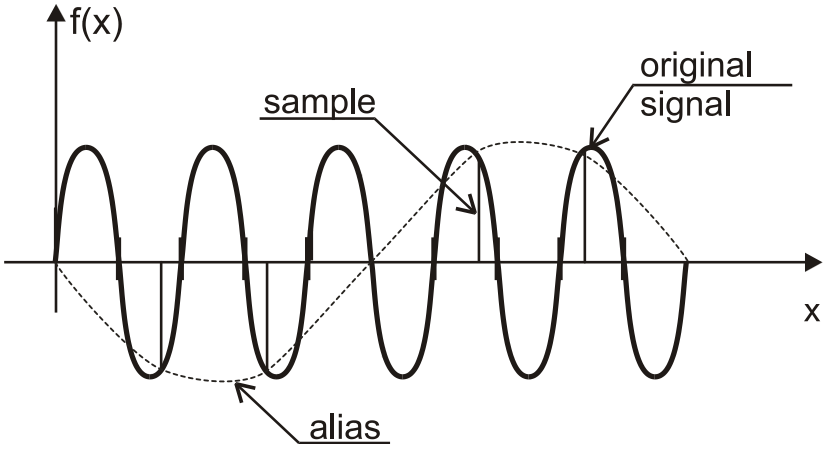
```
void glDrawElementsBaseVertex(
   GLenum mode, GLsizei count, GLenum type,
   const GLvoid *indices, GLint basevertex)
```

Několik slov závěrem k části o optimalizaci kreslení. Instancování je jednoduchá a elegantní technika, kterou možná ve svých projektech využijete. Ostatní techniky ale ve svých projektech pravděpodobně potřebovat nebudete. Moderní hardware není až tak slabý, aby

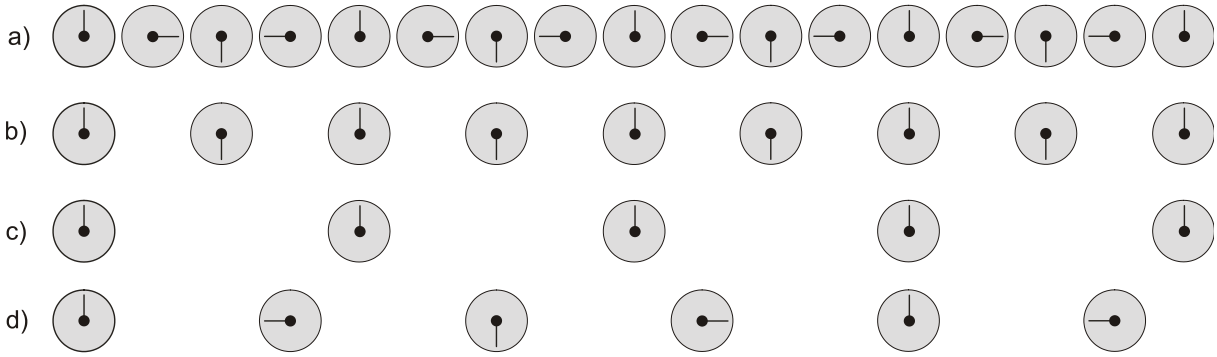
nedokázal kreslit vaše scény rychle. Pokud se setkáte s tím, že se vám scéna kreslí až podezřele pomalu, chyba je většinou jinde.

Aliasing

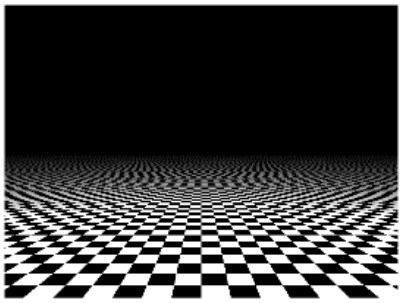
V poslední části se budeme bavit o aliasingu, antialiasingu a multisamplingu. Alias je artefakt ve vykreslených obrázcích, který je způsoben vzorkováním vysokofrekvenční informace nízkou frekvencí. Výsledkem jsou „zubaté“ okraje.

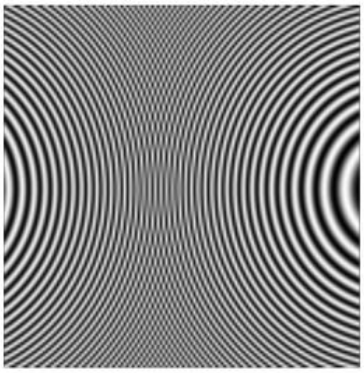


Důsledky aliasingu mohou být různé – od již zmíněných zubatých okrajů (zejména u téměř horizontálních či vertikálních čar), přes třepotání (části objektů jakoby střídavě mizely) až po tzv. efekt kol vagónu, který představuje problém s aliasingem v časové doméně. Jedná se o známý problém, kdy vzorkování rotace kol může vést až k tomu, že se kola jeví jako nehybná (viz obrázek c)).



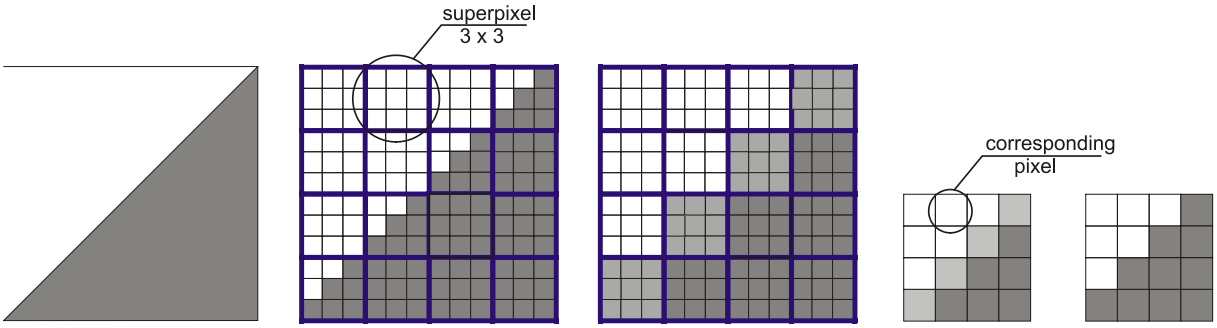
Následující obrázky ilustrují efekty aliasingu, kde vlivem „nízké“ vzorkovací frekvence se objevují vzory, které v původním objektu neexistují.





Řešením je tzv. **antialiasing**, což jsou obecně techniky odstranění aliasingu. Možností, jak odstranit aliasing, je hned několik. Jednou z možností je provést rozmazání obrázku, či přesněji detekci a rozmazání zubatých částí v rámci postprocessingu (práce s finálním obrázkem). Zde existuje mnoho technik, které se skrývají za různými zkratkami, možná jste se setkali se zkratkami, jako jsou TXAA (temporal antialiasing), FXAA (fast approximate anti-aliasing) nebo MLAA (morphological antialiasing).

Další možností je použít tzv. supersampling, neboli zpracovávat v rámci každého pixelu několik fragmentů, které se nakonec zprůměrují.

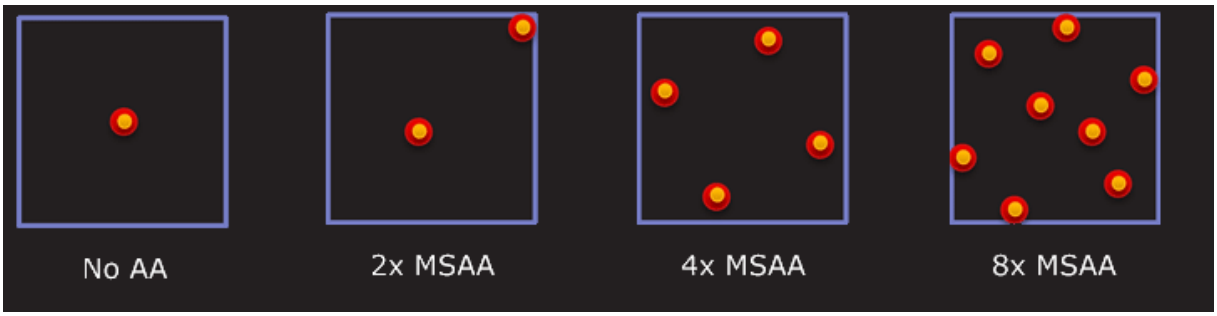


Supersampling má několik výhod. Je jednoduchý na implementaci (stačí kreslit v n-krát vyšším rozlišení) a dává velmi dobré výsledky. Bohužel má i několik nevýhod. Vyžaduje velké množství paměti, například FullHD obsahuje cca 2 mil. pixelů, pokud má každý pixel 4 byty barvu a 4 byty hloubku/stencil, máme 16 MB paměti bez supersamplingu, 9x supersampling znamená 150 MB paměti. S tím souvisí i požadavek na vysokou bandwidth paměti, a samozřejmě velký výpočetní výkon, protože musíme vyhodnotit několikanásobně více fragmentů.

I přesto se můžeme se supersamplingem setkat. Například NVIDIA pod názvem DSR (Dynamic Super Resolution) představuje technologii, kdy kreslí aplikaci ve vyšším rozlišení než je rozlišení obrazovky a poté provádí zmenšení, což je přesně supersampling.

My se ale více zaměříme na **multisampling**, což je zjednodušený supersampling. Označuje se často jako MSAA a můžeme ho najít v nastavení aplikací jako například 8x MSAA, neboli multisampling, který používá 8 vzorků na pixel. Rozmístění těchto vzorků je různé pro různé grafické karty a snaží se co nejvíce zamezit vzniku rušivých vzorů. Všechna primitiva

(trojúhelníky, body a úsečky) jsou rasterizována nikoliv ve středu pixelu, ale v místech těchto vzorků.



A v čem se tedy liší multisampling a supersampling? Multisampling se snaží řešit problém s výpočetním výkonem, nikoliv s pamětí. Dělá to tak, že spustí pouze jeden fragment shader a výsledek použije pro všechny fragmenty daného pixelu. Výpočet a porovnání hloubky a stencil operace se provedou pro každý fragment zvlášť. Tímto způsobem multisampling vyhladí okraje trojúhelníku a místa, kde se trojúhelníky setkávají. Multisampling nevyhladí ovšem vnitřek trojúhelníku. Pokud tedy máme zubaté hrany v důsledku špatných textur, multisampling nám nepomůže, i když by supersampling pomohl. I přesto je dobré jej používat.

Co všechno je nutné provést, abychom začali vyhlazovat za pomoci multisamplingu? Nejprve musíme vytvořit okno nebo framebuffer objekt, který bude mít podporu multisamplingu. V případě okna jsme odkázáni na prostředí, které nám okno vytváří, toto není záležitost OpenGL. Pokud pracujeme s GLUTem, okno podporující multisampling si vyžádáme přidáním bitu GLUT_MULTISAMPLE do parametrů funkce glutInitDisplayMode. V případě JOGLu zatím podpora neexistuje.

Druhou možností, jak si zajistit podporu multisamplingu, je vytvořit framebuffer objekt a k němu připojit multisample textury. Multisample textury jsou textury, které obsahují více vzorků na každý texel.

Pro 2D multisample textury používá OpenGL target GL_TEXTURE_2D_MULTISAMPLE (ve funkcích *glBindTexture* apod.). Data pro tyto textury se alokují pomocí funkce *glTexImage2DMultisample*.

```
void glTexImage2DMultisample(GLenum target,
  GLsizei samples, GLenum internalformat,
  GLsizei width, GLsizei height,
  GLboolean fixedsamplelocations)
```

Tato funkce se v mnohém podobá funkci *glTexImage2D* (parametry target, internalFormat, width, height). Parametr samples určuje počet vzorků textury. Toto číslo může nabývat hodnoty mezi 0 a maximem, na které se lze dotázat pomocí *glGetIntegerv* a parametrů GL_MAX_COLOR_TEXTURE_SAMPLES a GL_MAX_DEPTH_TEXTURE_SAMPLES. Poslední

parametr určuje, zdali mají mít všechny samply stejnou pozici ve všech texelech, nebo zdali dovolíme implementaci rozložení samplů v každém texelu jinak (může vést k lepším výsledkům). Nám zde bude stačit GL_FALSE.

Při práci s multisample FBO a multisample texturami musíme mít na paměti několik pravidel. Jednak musíme mít všechny multisample textury napojené na FBO stejný počet samplů, jinak máme chybu a nekompletní FBO. Dále, multisample textury nelze vzorkovat (nebo přesněji lze, ale speciálními funkcemi), nemají u nich smysl wrap módy či filtrování a tak nemají ani mipmapy. Pro nás nejjednodušší práce s multisample texturami a multisample FBO bude vždy použít *glBlitFramebuffer* a pomocí ní vytvořit z multisample textury běžnou texturu, se kterou umíme pracovat.

Na závěr pár věcí, které je dobré vědět při práci s multisamplingem. Použití multisamplingu lze povolit a zakázat pomocí funkcí *glEnable*/*glDisable* s parametrem GL_MULTISAMPLE. Pokud multisampling zakážeme a přesto máme framebuffer s více vzorky na pixel, jsou ovlivněny buď všechny vzorky stejně, nebo žádný vzorek, prostě jako bychom multisampling neměli. Počet multisample vzorků framebufferů můžeme zjistit zavoláním funkce *glGetIntegerv* s parametrem GL_SAMPLES.

S multisamplingem se pojí i technika nazvaná jako **alpha to coverage**. Pokud alpha to coverage povolíme, tak OpenGL bude brát alphu fragmentu, kterou dáme na výstup fragment shaderu, jako pokrytí daného pixelu. Pixely s alphou rovou nule tedy nemají žádné pokrytí a nezakryjí pixely za jimi, ostatní pixely jen z částí (nebo úplně, záleží na hodnotě alpha) zakryjí ty ostatní. Tato technika se hodí pro kreslení tenkých objektů, jako jsou trávy, listy stromů nebo vlasy.