PV112 Programování grafických aplikací

Jaro 2017

Výukový materiál

---

1. **přednáška: Nastavování uniform proměnných, druhy grafických primitiv, Vertex Buffer Objects, Vertex Array Objects, vykreslování**

Nastavování uniform proměnných

Než se seznámíme s principem vykreslování grafických primitiv v OpenGL, doplníme ještě k předchozí přednášce informace o tom, jakým způsobem nastavujeme uniform proměnné. Nejdříve musíme zjistit umístění uniform proměnné. To se provádí pomocí funkce glGetUniformLocation.

```
GLint glGetUniformLocation(GLuint program, const GLchar *name);
```

Funkce vrací integer hodnotu, která reprezentuje umístění dané uniform proměnné v objektu programu. Parametr *name* musí být string bez mezer a musí odpovídat aktivní uniform proměnné programu *program*, která není strukturou, polem struktur nebo subkomponentou vektoru nebo matice. Funkce vrátí hodnotu -1, pokud *name* neodpovídá žádné aktivní uniform proměnné programu *program*, pokud *name* začíná rezervovaným prefixem „gl_“ nebo pokud je *name* asociováno s nějakým atomickým čítačem nebo pojmenovaným blokem.

Samotné nastavení hodnoty uniform proměnné se provádí pomocí funkcí

```
void glUniform[1234][f|i|ui](location, jednotlivé vrcholy);
```

```
void glUniform[1234][fv|iv|uiv](location, count, value);
```

kde příznak *v* určuje, že data jsou uložena v poli.

Uniform proměnné mohou nabývat i podoby matice. V tomto případě se pro nastavení hodnot používá funkce

```
void glUniformMatrix[XXX]fv(location, count, transpose, const GLfloat *value);
```

kde XXX určuje velikost matice. Hodnoty 2, 3, 4 odpovídají maticím o velikostech 2x2, 3x3 a 4x4. Pokud chceme použít matici o různém počtu řádků a sloupců, použijeme následující zápis: 2x3, 3x2, 2x4, 4x2, 3x4, 4x3.

Význam dalších parametrů je následující:

- *location* = umístění uniform proměnné
- *count* = počet prvků v poli nebo počet matic, které chceme modifikovat
- *transpose* = zda se má matice transponovat při jejím načtení do uniform proměnné
- *value* = ukazatel na pole hodnot *count*

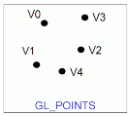
Uniform proměnné se nastavují aktuálnímu programu, je tedy před jejich nastavením a používáním nutné zavolat funkci *glUseProgram.*

Druhy grafických primitiv

OpenGL je nízkoúrovňové aplikační programové rozhraní, což způsobuje to, že programátor nemá k dispozici nástroje pro vykreslování složitých geometrických objektů. Je tedy úkolem programátora zkombinovat dostupná geometrická primitiva poskytovaná OpenGL a vytvořit tak komplexní tvary. Základní grafická primitiva dostupná v v core OpenGL profilu jsou body, čáry a trojúhelníky.

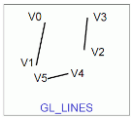
Pro jednoduchost budeme jejich použití demonstrovat ve 2D, nicméně vše funguje i ve 3D.

GL_POINTS



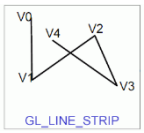
Izolované body patří mezi nejjednodušší grafická primitiva v OpenGL.

GL_LINES



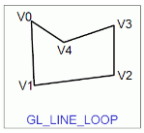
Úsečka je popsána dvěma koncovými body (vrcholy). Přitom se berou vždy dva po sobě jdoucí vrcholy, které vytvoří úsečku. Máme-li lichý počet vrcholů, je poslední vrchol ignorován.

GL_LINE_STRIP



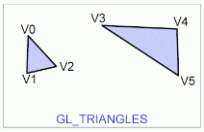
Umožňuje vykreslit lomenou čáru (polyline). Představuje vlastně rozšíření jednotlivých úseček. První dva vrcholy definují počáteční segment (úsečku) polyčáry a každý další vrchol přidává do polyčáry nový segment (počátek segmentu tvoří koncový vrchol aktuální polyčáry a konec segmentu definuje právě nový vrchol). Obecně tedy *n* vrcholů definuje lomenou čáru o *n-1* úsecích (segmentech). Tento způsob vykreslení je velmi efektivní, protože pokud kreslíme sadu úseček, které na sebe navazují, pak oproti tradičnímu vykreslování po jedné úsečce ušetříme téměř polovinu velikosti pole s uloženými vrcholy.

GL_LINE_LOOP



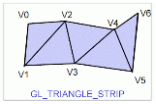
Smyčka tvořená úsečkami je drobnou modifikací předchozího případu. Opět se jedná o lomenou čáru, jejíž poslední vrchol je spojen s prvním. Pro *n* vrcholů tedy tento útvar obsahuje *n* segmentů. Pozor, vytvořená oblast není polygonem, je to pouze hraniční reprezentace bez výplně.

GL_TRIANGLES



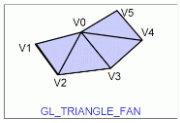
Trojúhelník je nejjednodušší ploška, jakou lze v OpenGL vykreslit. Je zadaný třemi vrcholy, které by neměly ležet v jedné přímce. Tři po sobě jdoucí vrcholy definují jeden trojúhelník. Pokud nemáme počet vrcholů dělitelný třemi, je zbytek vrcholů ignorován. Použití trojúhelníků je jednoduché, protože každý trojúhelník je konvexní a jeho vrcholy leží v jedné rovině (kterou definují). Díky tomu je velmi časté využívání trojúhelníkových sítí pro definici složitějších objektů. Na druhou stranu, pro každý trojúhelník je třeba zadat tři vrcholy, což představuje značný datový tok mezi procesorem a grafickou kartou. Proto OpenGL podporuje vytváření dalších primitiv popsaných dále, která tento problém částečně eliminují.

GL_TRIANGLE_STRIP



Dalším primitivem, které je složeno z trojúhelníků, ke tzv. pás trojúhelníků (triangle strip). Lze použít například při vykreslování stěn složitějších těles. První tři vrcholy definují první trojúhelník. Každý další vrchol pak definuje nový trojúhelník mající s posledním přidaným společnou hranu.

GL_TRIANGLE_FAN

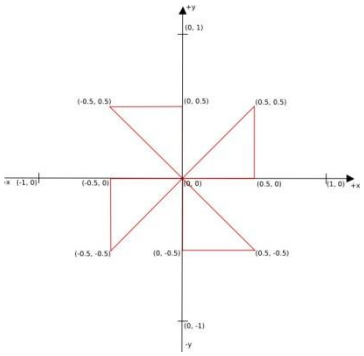


Trojúhelníky tvoří základ dalšího primitiva, trsu trojúhelníků (triangle fan). Cílem je snížení datového toku při zadávání vrcholů. Typickým využitím je vykreslování vrchlíku koule. Všechny trojúhelníky mají jeden vrchol společný.

Mezi další primitiva podporovaná v jazyce GLSL patří ještě GL_LINES_ADJACENCY, GL_LINE_STRIP_ADJACENCY, GL_TRIANGLES_ADJACENCY, GL_TRIANGLE_STRIP_ADJACENCY a GL_PATCHES. Tyto však nejsou ve vertex a fragment shaderech podporovány (pouze novějšími typy shaderů), proto se jimi nebudeme dále zabývat.

Princip vykreslování

Celý proces si demonstrujeme na příkladě vykreslení čtyř trojúhelníků na obrázku.

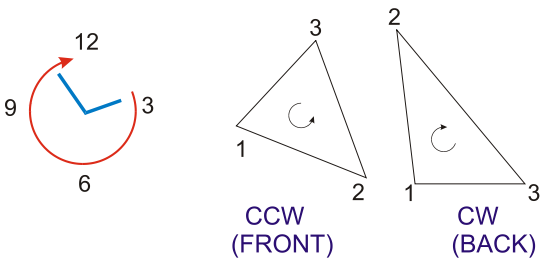


Z geometrického pohledu je trojúhelník definován pozicí tří jeho vrcholů. Vrchol je v terminologii OpenGL složen ze sady atributů, jako jsou jeho pozice, barva, texturové souřadnice, atd. Obrázek ukazuje situaci, ve které máme 4 trojúhelníky a 12 vrcholů. Předpokládejme, že každý z těchto vrcholů má definovánu pouze pozici (ignorujeme zatím barvu a další atributy). Pak můžeme tyto vrcholy uložit do následujícího pole:

```
GLfloat vertices_position[24] = {
        0.0, 0.0,
        0.5, 0.0,
        0.5, 0.5,

        0.0, 0.0,
        0.0, 0.5,
        -0.5, 0.5,

        0.0, 0.0,
        -0.5, 0.0,
        -0.5, -0.5,

        0.0, 0.0,
        0.0, -0.5,
        0.5, -0.5,
};
```

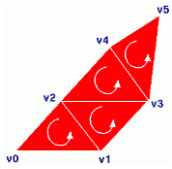
Všimněte si důležité věci, že vrcholy jsou do pole vkládány proti směru hodinových ručiček. Tímto způsobem jsou v OpenGL defaultně definovány přední stěny objektů. Trojúhelník má totiž definovánu přední a zadní stěnu a pro každou z nich můžeme použít jiný typ vykreslení, případně jednu ze stěn nemusíme vykreslit vůbec. Defaultně se však vykreslují obě stěny.

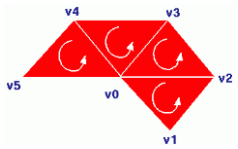
Defaultní nastavení orientace stěn trojúhelníka je následující: přední stěna je orientována proti směru chodu hodinových ručiček a zadní stěna je orientována naopak.



Orientaci vrcholů můžeme změnit. To se provádí pomocí příkazu glFrontFace(GLenum mode), kde parametr mode může mít hodnoty GL_CCW (proti směru hodinových ručiček - implicitně) nebo GL_CW (po směru).

U primitiv GL_TRIANGLE_STRIP a GL_TRIANGLE_FAN je orientace jednotlivých částí nastavena automaticky podle orientace prvního vloženého trojúhelníka.

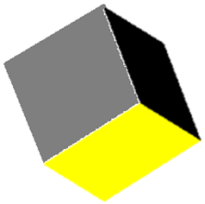




Pro urychlení vykreslování se velmi často používá tzv. back face culling, který odstraní neviditelné plošky (zadní – odvrácené od pozorovatele) ještě před rasterizací. Implicitně je renderování obou stěn polygonů zapnuto, což je velmi časově náročné. Použití je následující. Nejdříve musíme odstřel plošek povolit pomocí glEnable(GL_CULL_FACE) (analogicky glDisable). Samotný odstřel provedeme pomocí funkce glCullFace(GL_FRONT/GL_BACK/GL_FRONT_AND_BACK).

```
glDisable(GL_CULL_FACE);
```

```
glEnable(GL_CULL_FACE);
glCullFace(GL_BACK);
```



```
glEnable(GL_CULL_FACE);
glCullFace(GL_FRONT);
```

Nyní pokračujme v popisu principu vykreslení našich trojúhelníků. Jak tedy teď vykreslíme trojúhelník definovaný pomocí výše uvedeného pole? Prvním krokem je převedení obsahu tohoto pole do tzv. **Vertex Buffer Object** (VBO). VBO je v podstatě kus paměti (buffer), který je spravován pomocí OpenGL a je umístěn přímo na grafické kartě. VBO je třeba vytvořit, alokovat a naplnit daty. Následně za použití kombinace **Vertex Array Objects** (VAO) a VBO definujeme grafická primitiva pro vykreslení (složená z vrcholů). Nyní se na tento proces podíváme detailněji.

VBO jsou jedním z druhů buffer objektů, které OpenGL používá. Obecně s buffer objekty se pracuje následujícím způsobem. V prvním kroku musíme vygenerovat jméno nově vytvářeného bufferu. Poté dojde k jeho aktivaci (tzv. binding) a uložení samotných dat do tohoto bufferu. Ve čtvrtém kroku data z bufferu použijeme (např. vykreslíme) a nakonec buffer zrušíme z paměti.

Vytvoření buffer objektu

Pro vytvoření BO využíváme následující funkce:

- void glGenBuffers(sizei *n*, uint **buffers);* - funkce, která vytvoří nový BO
- void glBindBuffer(enum *target*, uint *buffer);* - funkce pro navázání nového BO

- void glBufferData(enum *target*, GLsizeptr *size,* const void **data,* enum *usage);* - funkce pro zkopírování dat do již vytvořeného BO

Krok 1 - glGenBuffers()
Funkce generuje jména buffer objektů. Má dva parametry:

- *n* – počet jmen buffer objektů, které mají být generovány
- *buffers* – pole, ve kterém jsou jména generovaných buffer objektů uložena. Pole pak obsahuje sekvenci celočíselných názvů.

Funkce vrací *n* jmen buffer objektů uložených v *buffers*. Není garantováno, že jména vytváří spojitou sadu integer hodnot. Je ale zaručeno, že žádné z těchto jmen nebylo použito bezprostředně před voláním této funkce. Jméno tedy může sloužit jako jednoznačný identifikátor bufferu. Dokud není zavolána funkce glBindBuffer(), nejsou s těmito jmény asociovány žádné buffer objekty.
Příklad vygenerování jména pro jeden buffer:

```
GLuint bufferID;
glGenBuffers(1, &bufferID);
```

Krok 2 - glBindBuffer()
Tato funkce slouží k navázání bufferu ID jakožto aktuálního bufferu. Pokud je ID nula, navázaný buffer se přestane používat. Funkce má dva parametry:

- *target* – specifikuje cílový buffer, na který je buffer objekt navázán. Tato symbolická konstanta nabývá jedné z těchto hodnot: GL_ARRAY_BUFFER, GL_ATOMIC_COUNTER_BUFFER, GL_COPY_READ_BUFFER, GL_COPY_WRITE_BUFFER, GL_DRAW_INDIRECT_BUFFER, GL_DISPATCH_INDIRECT_BUFFER, GL_ELEMENT_ARRAY_BUFFER, GL_PIXEL_PACK_BUFFER, GL_PIXEL_UNPACK_BUFFER, GL_SHADER_STORAGE_BUFFER, GL_TEXTURE_BUFFER, GL_TRANSFORM_FEEDBACK_BUFFER, GL_UNIFORM_BUFFER.
- *buffer* – specifikuje jméno buffer objektu (z kroku 1).

BO pracuje se dvěma možnými typy cílů:

- **Array buffers** (ARRAY_BUFFER) – slouží pro ukládání vrcholů objektů. Tyto buffery obsahují atributy vrcholu, jako jsou souřadnice, texturové souřadnice, barva vrcholů a normály. Mohou být prokládané nebo sekvenční.
- **Element array buffers** (ELEMENT_ARRAY_BUFFER) – tyto buffery jsou určeny zejména pro uložení indexů na vrcholy.

Příklad:

```
glBindBuffer(GL_ARRAY_BUFFER, bufferID);
```

Krok 3 - glBufferData()
Funkce glBufferData() může být použita dvěma způsoby:

- Alokuje paměť a nastaví použití pro aktuální buffer objekt a nastaví mu data na NULL. Uživatel může data objektu mapovat později.
- Alokuje paměť, nastaví použití a hned zkopíruje data. Tento přístup se využívá, pokud pracujeme se statickým datovým modelem.

Parametry funkce jsou následující:

- *target* – specifikuje typ buffer objektu. Nabývá jedné z konstant GL_ARRAY_BUFFER, GL_ATOMIC_COUNTER_BUFFER, GL_COPY_READ_BUFFER, GL_COPY_WRITE_BUFFER, GL_DRAW_INDIRECT_BUFFER, GL_DISPATCH_INDIRECT_BUFFER, GL_ELEMENT_ARRAY_BUFFER, GL_PIXEL_PACK_BUFFER, GL_PIXEL_UNPACK_BUFFER, GL_SHADER_STORAGE_BUFFER, GL_TEXTURE_BUFFER, GL_TRANSFORM_FEEDBACK_BUFFER, GL_UNIFORM_BUFFER.
- *size* – definuje množství paměti požadované pro uložení dat
- *data* – buď ukazatel na data, která budou zkopírována do nově vytvořeného datového uložiště, nebo NULL, pokud se nemají žádná data kopírovat (paměť se pouze rezervuje)
- *usage* – specifikuje očekávaný „vzor“ použití nového uložiště. Symbolická konstanta musí být GL_STREAM_DRAW, GL_STREAM_READ, GL_STREAM_COPY, GL_STATIC_DRAW, GL_STATIC_READ, GL_STATIC_COPY, GL_DYNAMIC_DRAW, GL_DYNAMIC_READ nebo GL_DYNAMIC_COPY. Pokud je parametr *usage* zvolen nevhodně, nebude to mít vliv na funkčnost aplikace. Bude ale ovlivněn její výkon.

Jednotlivé flagy mají následující význam:

- STREAM – obsah datového uložiště bude modifikován pouze jednou a použit (čten) nejvýše několikrát.
- STATIC - obsah datového uložiště bude modifikován pouze jednou a použit (čten) mnohokrát.
- DYNAMIC – obsah datového uložiště bude opakovaně modifikován (zapisován) a použit (čten) mnohokrát.

Typy přístupu jsou následující:

- DRAW – obsah datového uložiště je modifikován aplikací a používán pro renderování pomocí OpenGL.
- READ – obsah datového uložiště je modifikován čtením dat z GL a data jsou vrácena, pokud jsou požadována aplikací.
- COPY – obsah datového uložiště je modifikován čtením dat z GL a používán pro renderování pomocí OpenGL.

Je třeba poznamenat, že v nejnovějších verzích OpenGL (od 4.3 dále) se již parametry *usage* nepoužívají a tento problém je řešen jiným způsobem. Ukázalo se totiž, že tyto parametry moc neodpovídají tomu, jakou paměť bude muset OpenGL opravdu použít.

Nyní si uveďme příklad naplnění bufferu:

```
glBufferData (GL_ARRAY_BUFFER, //typ bufferu
     sizeof (triangles), // kolik dat ukládám
     triangles, // odkud
     GL_STATIC_DRAW); // použití bufferu
```

Buffer je naplněn souřadnicemi vrcholu z pole *triangles*. Zavolání *glBufferData()* na buffer, který již obsahuje data, smaže jeho uložená data.

Buffer objekty mají k dispozici i další funkce:

```
void glBufferSubData(enum target, GLint offset,
     GLsizei size, const void *data);
```

Funkce **glBufferSubData()** kopíruje data daného rozsahu do buffer objektu. Zkopíruje data do již existujícího buffer objektu počínajíce daným offsetem. Celková velikost bufferu musí být předem dána pomocí volání funkce glBufferData().

Tato funkce se využívá například v případě, že buffer inicializujeme jako prázdný a poté jej plníme daty. Některé zdroje tento způsob označují jako efektivnější.

První parametr funkce udává typ bufferu, parametr offset definuje, odkud se má do bufferu provádět zápis (nula znamená od počátku bufferu). Parametr size určuje velikost zapisovaných dat a poslední parametr je odkaz na data.

```
void glGetBufferSubData(enum target, GLintptr offset,
     GLsizeptr size, void *data);
```

Funkce **glGetBufferSubData()** získá podmnožinu dat o daném rozsahu z aktuálního buffer objektu.

Jako příklad použití si uvedeme dva způsoby přepsání celého bufferu:

```
glBufferData(GL_ARRAY_BUFFER, sizeof(triangles), triangles,
     GL_STATIC_DRAW);
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(triangles),
triangles);
```

Mezi další podporované funkce patři glMapBuffer a glUnmapBuffer. Tyto funkce „odemykají a zamykají“ buffery, což umožní natažení dat do těchto bufferů nebo přenechání kontroly serveru. Jako vstup na začátek bufferu je využit dočasný ukazatel, který rovněž mapuje buffer do paměti klienta. OpenGL je zodpovědné za to, jakým způsobem dochází k tomuto mapování.

```
void *glMapBuffer(enum target, enum access);
```

- **glMapBuffer()** – tato funkce mapuje buffer objekt do paměti klienta, pokud je to možné. Vrácený ukazatel může být využit pro čtení nebo zápis přímo CPU, přičemž povoluje libovolné updaty. Pokud nemůže být buffer namapován, implementace vrací ukazatel NULL.

```
boolean glUnmapBuffer(enum target);
```

- **glUnmapBuffer()** – tato funkce odmapuje buffer objekt z paměti klienta. V případě úspěchu vrací kód, který potvrzuje správně provedený update. Pokud funkce vrátí chybu, obsahy bufferů se mohou stát nedefinovanými díky neočekávané události. V tomto případě by data měly být aplikací poslána znovu.

Krok 4 – využití BO (vykreslení)
Samotné vykreslování probíhá téměř identicky jako při použití Vertex Arrays, které budou následovat. Např. při vykreslování použijeme VBO a data pro vykreslení nastavíme pomocí `glVertexAttribPointer(position_loc, 3, GL_FLOAT, GL_FALSE, 0, (char *) NULL);`

Krok 5 – zrušení bufferu z paměti

```
void glDeleteBuffers(sizei n, const uint *buffers);
```

```
boolean glIsBuffer(uint buffer);
```

Funkce glDeleteBuffers() smaže dané buffer objekty a glIsBuffer() se dotazuje na identifikátor buffer objektu.

Pokud se nyní vrátíme k dříve uvedenému příkladu se čtyřmi trojúhelníky, bude postup následující. Pomocí následujících příkazů nejdříve vytvoříme VBO a poté alokujeme místo v paměti GPU, do kterého z CPU zkopírujeme data z našeho pole.

```
GLuint vbo;
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices_position),
vertices_position, GL_STATIC_DRAW);
```

Funkce glGenBuffers() vytvoří ukazatel na náš nově vytvořený VBO. Tato funkce dokáže v případě potřeby vytvořit celé pole ukazatelů. Po vytvoření VBO je třeba jej navázat, což zajistí funkce glBindBuffer(). Poslední funkce glBufferData() alokuje potřebný prostor pro VBO a naplní jej obsahem pole *vertices_position*. Nyní je už možné data poslat do OpenGL pipeline.

Pozice načtené do VBO se následně zpracují pomocí vertex shaderu. Uvádíme jednoduchý příklad, ve kterém vytváříme globální proměnnou typu *vec4*, která obdrží pozici vrcholu uloženého ve VBO. Pokud nebudeme v bufferu specifikovat proměnné x, y, z a w, tak tyto složky budou mít v shaderu hodnoty (0,0,0,1). Pokud do takovéhoto vektoru pošleme pouze hodnoty x a y, zůstanou poslední dvě hodnoty vektoru nezměněné (x, y,0,1). Shader musí obsahovat main() funkci, ve které v tomto případě nastaví hodnotu interní GLSL proměnné *gl_Position* na hodnotu souřadnic vrcholu.

```
in vec4 position;

void main() {
     gl_Position = position;
}
```

Vzhledem k tomu, že jednotlivé vrcholy mohou mít další atributy, jako například barvu nebo souřadnice textury (definované ve fragment shaderu), je třeba tuto informaci propojit s jednotlivými vrcholy uloženými ve VBO. OpenGL k tomuto účelu používá speciální proměnnou nazývanou **Vertex Array Object** (VAO). Následující příklad fragment shaderu vytváří a naplňuje proměnnou udávající barvu každého fragmentu (bílá).

Když je VAO vytvořen a navázán, je do něj uložena informace o umístění dat. Z toho vyplývá, že VAO musí být navázán ještě předtím, než přesouváme data z CPU do VBO nebo musíme vytvořit shader program, který propojí vstupy a výstupy sám.

Vertex Array Objects

Vertex arrays = sada uživatelem definovaných polí, které obsahují atributy jednotlivých vrcholů.

Vertex array objects = objekty, které obsahují informaci o tom, ve kterých bufferech (a jak) jsou uložena data těchto atributů.

Vertex array objects je třeba vytvořit a navázat. K tomu slouží funkce:

```
void glGenVertexArrays(GLsizei n, Gluint *arrays);
```

Tato funkce generuje *n* aktuálně nepoužívaných jmen, která jsou uložena do pole *arrays.*

```
GLvoid glBindVertexArray(GLuint array);
```

Tato funkce má dva způsoby použití. Sváže VAO se jménem v *array* nebo při použití hodnoty *array* 0 dojde k ukončení používání VAO (přestane se používat aktivní VAO).

Nepoužívaný VAO je možné následně smazat:

```
void glDeleteVertexArrays(GLsizei n, GLuint *arrays);
```

Funkce smaže *n* VAO specifikovaných v parametru *arrays.* Uvolněná jména VAO je možné v budoucnu opět použít.

Funkce **glIsVertexArray()** se používá v případě, kdy potřebujeme zjistit, zda daná hodnota reprezentuje alokovaný VAO (ne nutně inicializovaný).

```
GLboolean glIsVertexArray(GLuint array);
```

Funkce vrací GL_TRUE, pokud je *array* jméno VAO, který byl vygenerován pomocí glGenVertexArrays() a nebyl ještě smazán. GL_FALSE vrací v případě, pokud je v *array* nulová nebo nenulová hodnota, která není jménem žádného VAO.

Zpracování atributů

Pro každý atribut vrcholů musíme provést následující:

1. Získání indexu atributu ze shaderu
2. Povolení příslušného atributu
3. Nastavení bufferu, ze kterého se data pro tento atribut budou brát
4. Nastavení způsobu, jak jsou tato data v bufferu uložena

1. **krok** – získání indexu

```
GLint glGetAttribLocation(GLuint program, const GLchar *name);
```

Funkce vrací hodnotu >= 0 nebo -1, pokud tento atribut nebylo možné najít.

2. **krok** – povolení atributu

Před prvním použitím pole vrcholů v programu je zapotřebí příslušný atribut povolit zavoláním funkce glEnableVertexAttribArray(int idx), kde parametr *idx* určuje index příslušného atributu vrcholů.Pokud některé z těchto polí nebudeme chtít použít, je možné zavolat funkci glDisableVertexAttribArray(int idx), kde parametr *idx* nabývá stejných hodnot jako u předchozí funkce.

```
void glEnableVertexAttribArray(int idx)
```

```
void glDisableVertexAttribArray(int idx)
```

3. **krok** – navázání bufferu

```
glBindBuffer(GL_ARRAY_BUFFER, bufferID);
```

4. **krok** – nastavení způsobu uložení

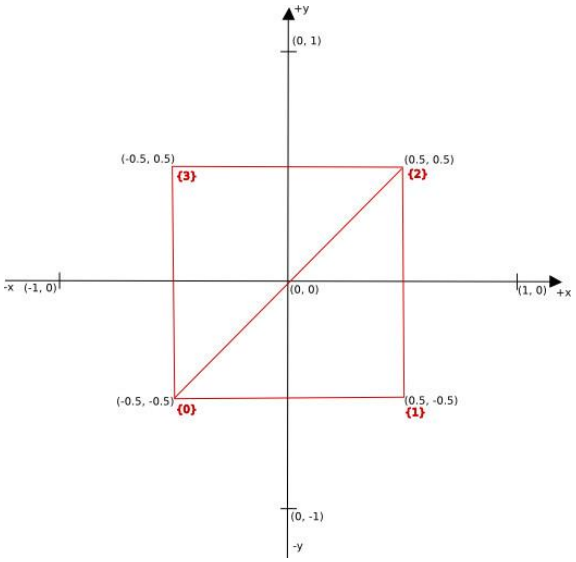
```
glVertexAttrib[IL]Pointer (GLuint index, GLint
size, GLenum type, GLboolean normalized, GLsizei
stride, const GLvoid * pointer)
```

Jednotlivé parametry mají následující význam:

- *index* – specifikuje index atributu vrcholů, který chceme měnit
- *size* – specifikuje počet komponent, které jsou zapotřebí pro uložení daného atributu vrcholů. Hodnota musí být 1, 2, 3 nebo 4.
- *type* – specifikuje datový typ každé komponenty v poli. Funkce glVertexAttribPointer a glVertexAttribIPointer akceptují tyto symbolické konstanty: GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, a GL_UNSIGNED_INT. Iniciální hodnota je GL_FLOAT. Další možnosti viz specifikace.
- *normalized* – pro funkci glVertexAttribPointer specifikuje, jestli mají být datové hodnoty při přístupu k nim normalizovány (GL_TRUE) nebo ne (GL_FALSE). To znamená, že pokud jsou např. hodnoty uloženy ve formátu signed integer, budou namapovány do rozsahu [-1, 1]. Pro unsigned integer to bude rozsah [0,1]. K mapování dojde v případě, že se k těmto datům přistupuje a konvertují se do floating point formátu. Při nastavení GL_FALSE jsou hodnoty přímo konvertovány na float formát bez normalizace.
- *stride* – specifikuje offset mezi po sobě následujícími atributy vrcholů. Pokud je hodnota stride nastavena na 0, následují atributy ve Vertex Array hned po sobě.
- *pointer* – specifikuje offset první komponenty prvního atributu vrcholů v poli odpovídajícímu bufferu aktuálně navázaného k cíli GL_ARRAY_BUFFER.

Práce s indexy

V mnoha případech se vrcholy v geometrii opakují (viz obrázek). V takovém případě je vhodné použít místo opakování vrcholů indexy. Tyto indexy ukazují na vrcholy v poli, které máme použít.




```
GLfloat vertices_position[8] = {
        -0.5, -0.5,
        0.5, -0.5,
        0.5, 0.5,
        -0.5, 0.5,
};

GLuint indices[6] = {
        0, 1, 2,
        2, 3, 0
};
```

Indexy jsou uloženy v buffer objektu GL_ELEMENT_ARRAY_BUFFER.

Buffer, ze kterého indexy bereme, je třeba navázat pomocí funkce

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, bufferID);
```

Pokud indexy nemáme nebo nechceme používat, funkce se nevolá.

Příklad:

```
int position_loc = glGetAttribLocation(program, "position");
int color_loc = glGetAttribLocation(program, "color");
glGenVertexArrays(1, &cube_geometry);
glBindVertexArray(cube_geometry);
glBindBuffer(GL_ARRAY_BUFFER, cube_positions_VBO);
glEnableVertexAttribArray(position_loc);
glVertexAttribPointer(position_loc, 2, GL_FLOAT, GL_FALSE, 0, 0);
glBindBuffer(GL_ARRAY_BUFFER, cube_colors_VBO);
glEnableVertexAttribArray(color_loc);
glVertexAttribPointer(color_loc, 3, GL_UNSIGNED_BYTE, GL_TRUE, 0, 0);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, cube_indices_VBO);

// a další geometrie
glGenVertexArrays(1, &sphere_geometry);
…
```

Vykreslení geometrie dat

Prvky pole lze vybrat dvěma základními způsoby (podle toho, jakým způsobem máme uložena data):

```
void glDrawElements(GLenum mode, GLsizei count,
        GLenum type, const void *indices);
```

Prvky pole lze vybrat zadáním seznamu několika vrcholů pomocí funkce void glDrawElements(GLenum mode, GLsizei count, GLenum type, void *indices). Tato funkce vybere *count* vrcholů ze všech zapnutých polí, přičemž posloupnost indexů vybíraných vrcholů je uložena v poli *indices*. Typ tohoto pole pak určuje parametr *type*, který může nabývat hodnot GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT nebo GL_UNSIGNED_INT. Parametr *mode* určuje, jaká geometrická primitiva mají být z vybíraných vrcholů konstruována.

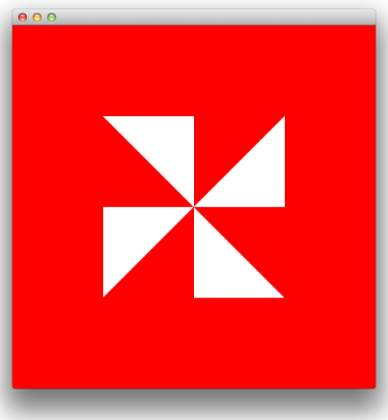
```
void glDrawArrays(GLenum mode, GLint first, GLsizei count);
```

Druhou možností, jak můžeme prvky pole vybrat, je zadáním seznamu několika po sobě jdoucích vrcholů pomocí funkce void glDrawArrays(GLenum mode, GLint first, GLsizei count). Tato funkce vybere *count* vrcholů, které jsou uloženy v poli za sebou. Index prvního vrcholu je určen parametrem *first*. Tato funkce nepotřebuje použití index bufferu.

Uveďme následující jednoduchý příklad, který opět využívá data o trojúhelnících vytvořených na začátku přednášky. VAO se použije ve funkci display pro výsledné zobrazení dat. Funkce display na začátku vyčistí barevný buffer (zdůvodnění v dalších přednáškách) a poté naváže Vertex Array předaný jako parametr funkce display. Vykreslení se realizuje pomocí funkce glDrawArrays s prvním parametrem definujícím, jaký typ primitiva se má z vrcholů vykreslit.

```
void display(GLuint &vao) {
    glClear(GL_COLOR_BUFFER_BIT);

    glBindVertexArray(vao);
    glDrawArrays(GL_TRIANGLES, 0, 12);

    // Swap front and back buffers
    SwapBuffers();
}
```

Obrázek ukazuje výsledné zobrazení vstupních trojúhelníků.



Místo trojúhelníků můžeme zobrazit jednotlivé body:

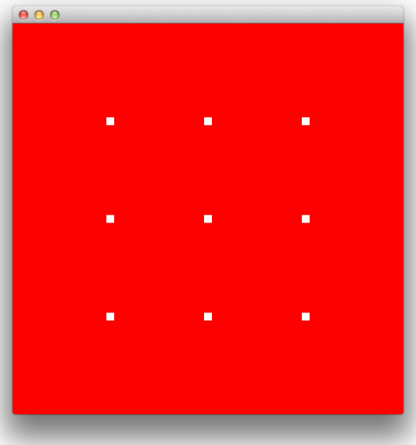
```
glDrawArrays(GL_POINTS, 0, 12);
```

Pro určení velikosti vykreslených bodů je třeba tuto funkci zpřístupnit při inicializaci:

```
glEnable(GL_PROGRAM_POINT_SIZE);
```

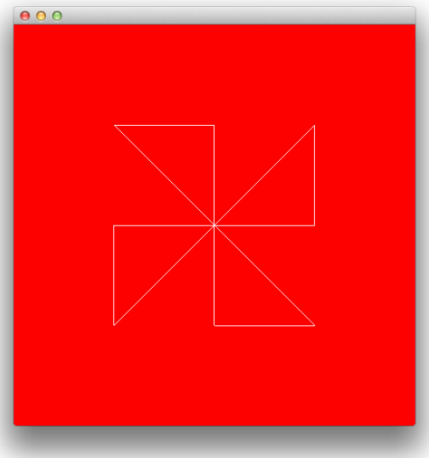
Poté je možné měnit velikost bodů ve vertex shaderu:

```
gl_PointSize = 10.0;
```



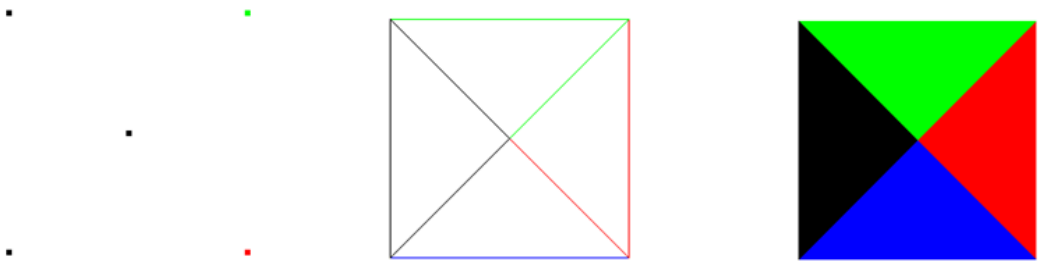
Trojúhelníky vykreslené v prvním příkladě jsou defaultně vyplněny barvou. Toto je však možné změnit použitím funkce glPolygonMode v inicializační funkci. Pro vykreslení pouze obrysu trojúhelníků použijeme tento příkaz:

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
```



Pro změnu kreslícího módu můžeme využít následující modifikace tohoto příkazu:

```
glPolygonMode(GL_FRONT_AND_BACK, GL_POINT);
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
```



Tyto módy můžeme aplikovat na stany trojúhelníka odděleně (zvlášť na přední a zadní stěnu). Toho dosáhneme použitím parametru GL_FRONT, resp. GL_BACK.