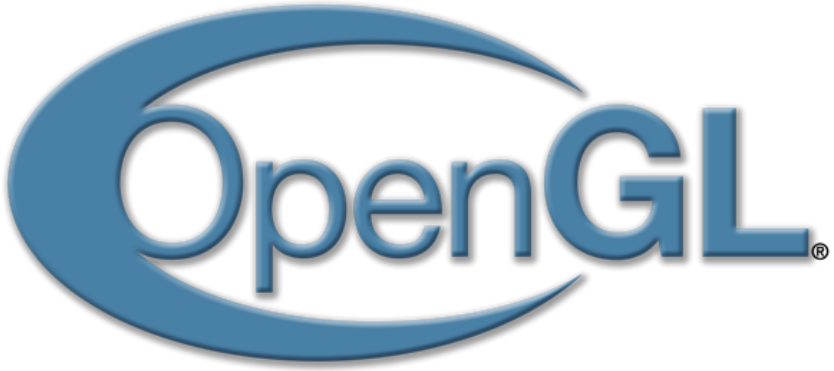


PV112 – Programování grafických aplikací

2. přednáška – nastavování uniform proměnných, druhy grafických primitiv, VBO, VAO, vykreslování

Nastavování uniform proměnných

- Zjištění umístění uniform proměnné

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

- Nastavení hodnoty uniform proměnné

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

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

  - Příznak *v* určuje, že jsou data uložena v poli

Nastavení uniform matic

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

- XXX určuje velikost matice
  - 2,3,4 = 2x2, 3x3, 4x4
  - 2x3, 3x2, 2x4, 4x2, 3x4, 4x3
- *location* = umístění uniform proměnné
- *count* = počet matic, které chceme nahrát
- *transpose* = zda se má matice transponovat při jejím načtení do uniform proměnné
- *value* = ukazatel na pole *count* matic

Uniform proměnné

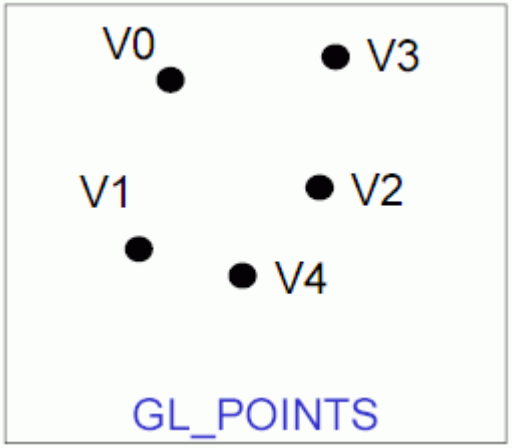
- Nastavují se aktuálnímu programu, je tedy před jejich nastavením a používáním nutné zavolat funkci *glUseProgram*

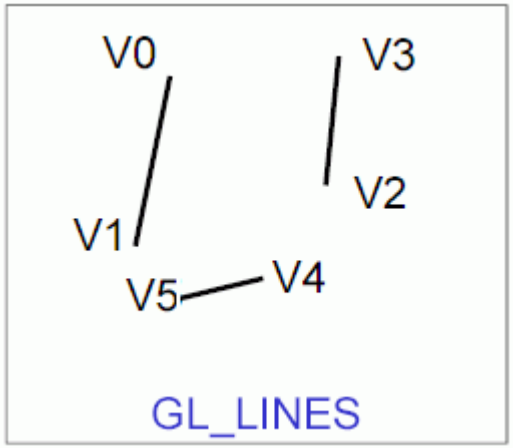
Druhy grafických primitiv

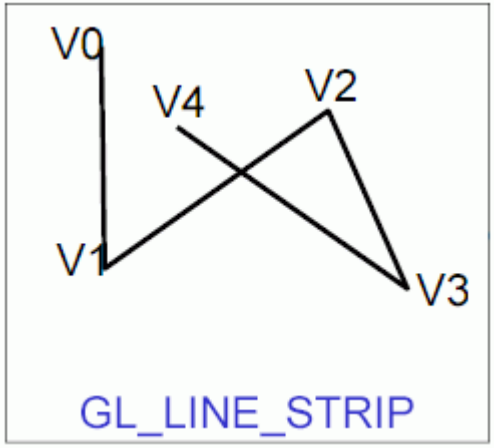
- GLSL nepodporuje vykreslování složitých geometrických objektů
- Základní grafická primitiva:
  - body
  - čáry
  - trojúhelníky

Grafická primitiva

GL_POINTS, GL_LINES, GL_LINE_STRIP, …

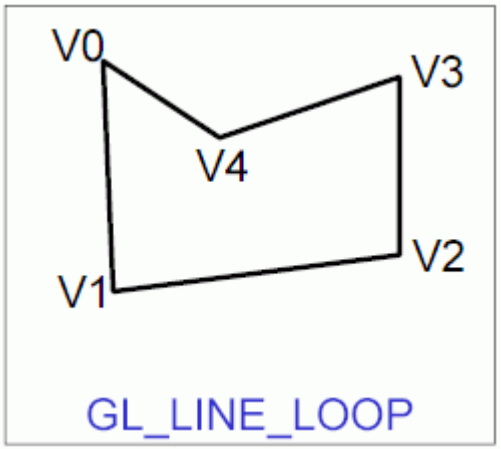


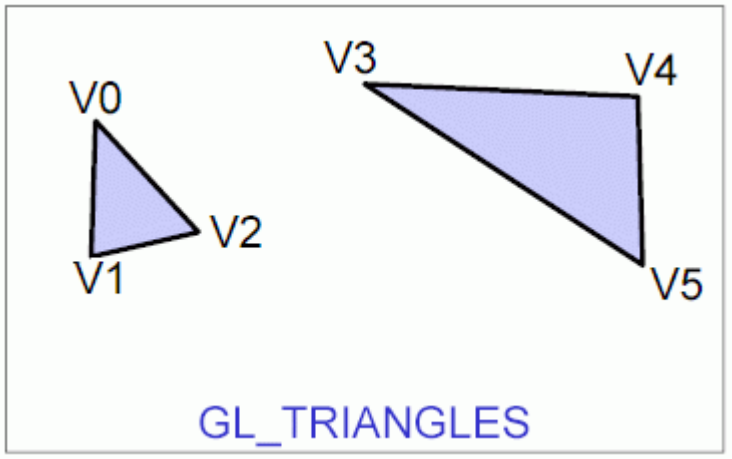




Grafická primitiva

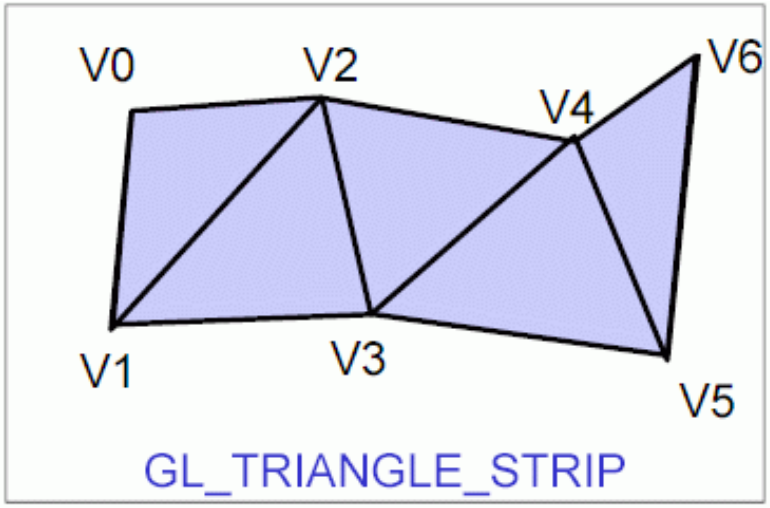
… GL_LINE_LOOP, GL_TRIANGLES, …

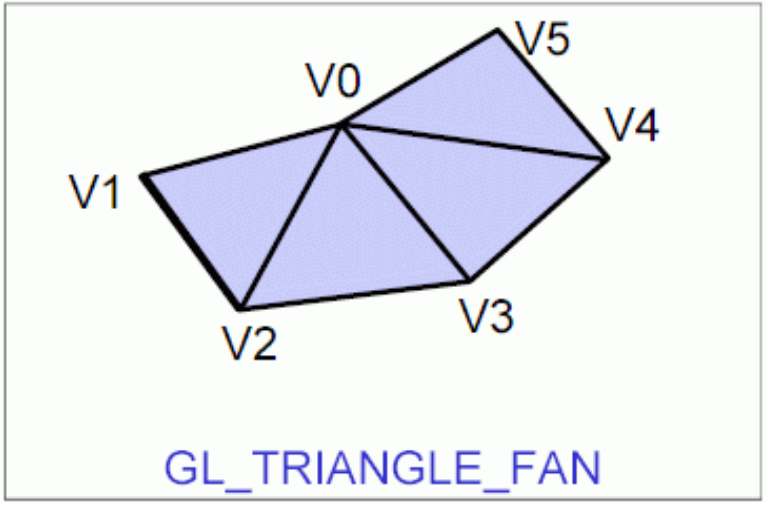




Grafická primitiva

GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN



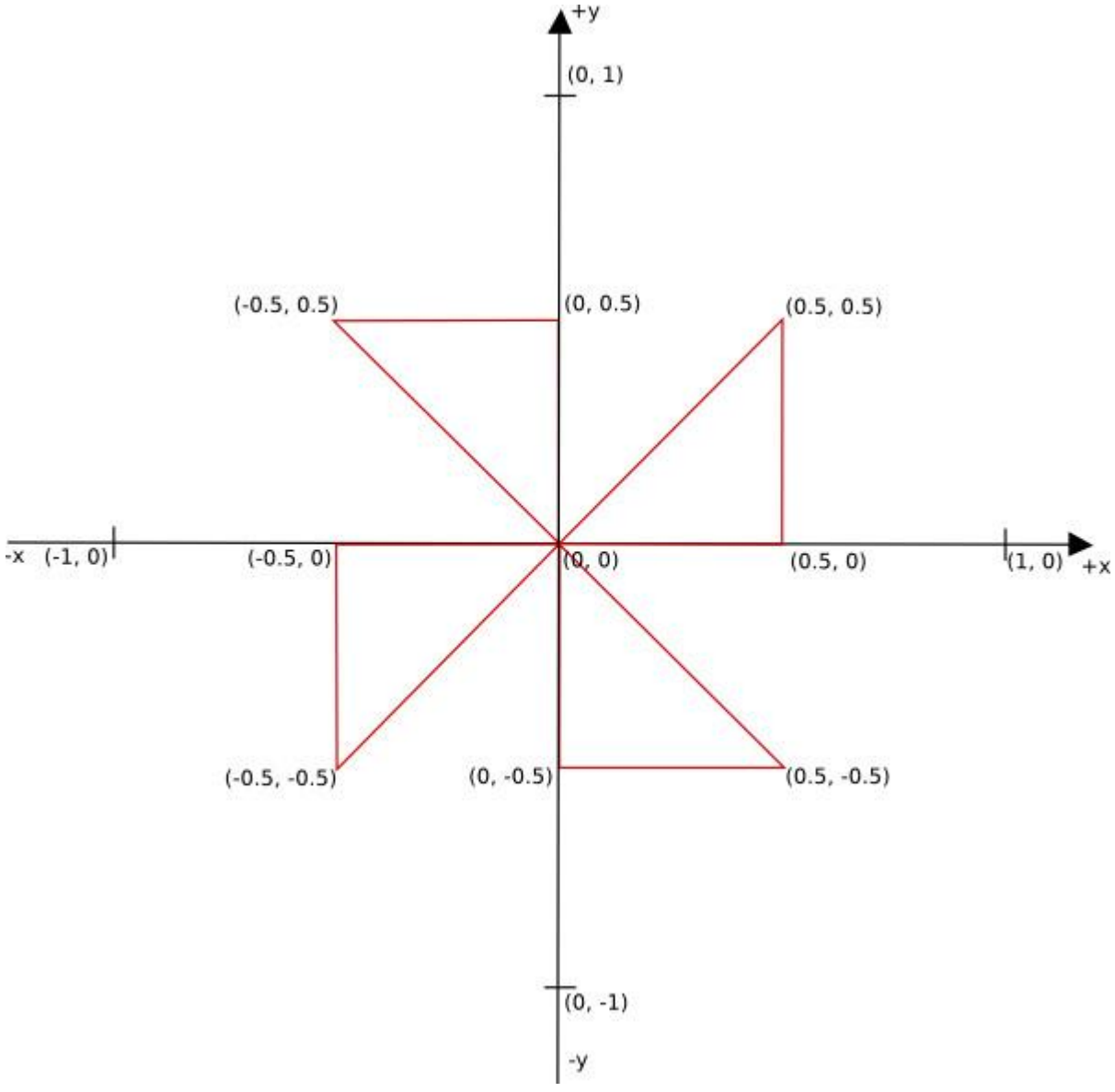


Další primitiva

- GL_LINES_ADJACENCY,
  GL_LINE_STRIP_ADJACENCY,
  GL_TRIANGLES_ADJACENCY,
  GL_TRIANGLE_STRIP_ADJACENCY a
  GL_PATCHES

- Nejsou používány vertex a fragment shadery

Příklad



Princip vykreslování

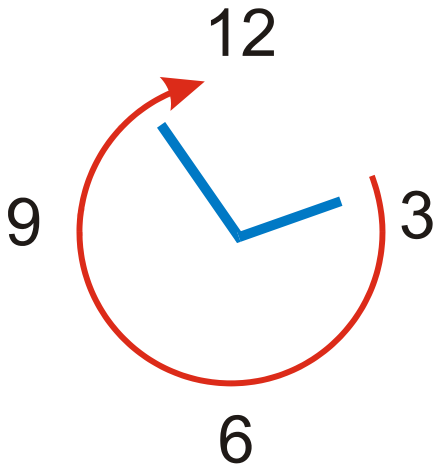
- Máme scénu se 4mi trojúhelníky, tedy 12ti vrcholy
- Pozice vrcholů můžeme uložit do pole

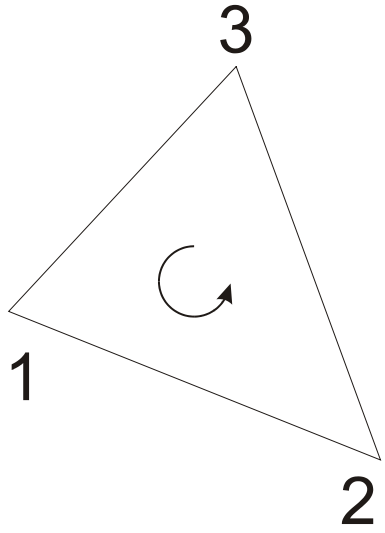
```
GLfloat vertices_position[24] = {
        0.0, 0.0,
        0.5, 0.0,
        0.5, 0.5,

        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,
};
```

Pořadí vrcholů

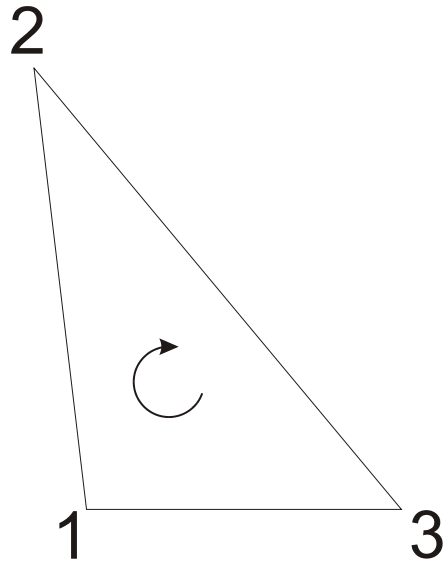
- Vrcholy trojúhelníka jsou zadávány **proti směru hodinových ručiček** = přední stěna
- Trojúhelníky mají dvě stěny (přední a zadní), které se mohou vykreslovat různými způsoby (nebo vůbec)
- Defaultně se vykresluje přední i zadní stěna

Orientace vrcholů





CCW
(FRONT)

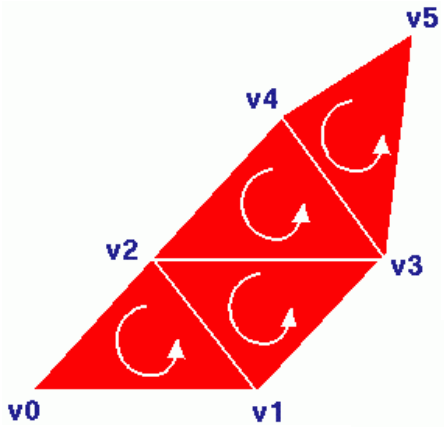


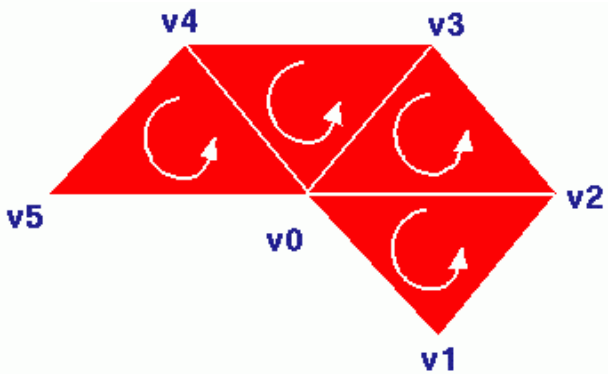
CW
(BACK)

glFrontFace(GL_CW/GL_CCW)

Orientace vrcholů

- Orientace GL_TRIANGLE_STRIP a GL_TRIANGLE_FAN je automaticky nastavena podle orientace prvního trojúhelníka





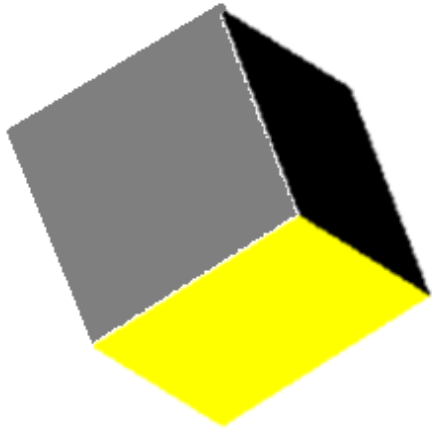
profs.sci.univr.it

Odstřel plošek

glEnable/glDisable(GL_CULL_FACE)
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);
```

Vykreslení

- Pole vrcholů je třeba převést na tzv. **Vertex Buffer Object (VBO)**
  - Kus paměti přímo na grafické kartě, který je zpracováván pomocí OpenGL
- VBO je třeba vytvořit, alokovat a naplnit daty

Vykreslování objektů

- Za použití **Vertex Array Objects** (VAO) a **Vertex Buffer Objects** (VBO) definujeme grafická primitiva pro vykreslení (složená z vrcholů)

Použití buffer objektů – 5 kroků

1. Vygenerování jména bufferu
2. Aktivace bufferu (bind)
3. Uložení dat do bufferu
4. Využití bufferu
5. Zrušení bufferu z paměti

Vytvoření BO (kroky 1, 2, 3)

- Vytvoření nového jména BO

```
void glGenBuffers(sizei n, uint *buffers);
```

- Navázání nového BO

```
void glBindBuffer(enum target, uint buffer);
```

- Přenesení (kopírování) dat do již vytvořeného BO

```
void glBufferData(enum target, GLsizeptr size,
                  const void *data, enum usage);
```

krok 1 - glGenBuffers()

- Parametry:
  - *n* – počet jmen buffer objektů, které mají být generovány
  - *buffers* – pole, ve kterém jsou jména generovaných buffer objektů uložena
- Dokud není zavolána funkce glBindBuffer(), nejsou s těmito jmény asociovány žádné buffer objekty

krok 1 - glGenBuffers()

- Příklad vygenerování jména pro jeden buffer

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

krok 2 - glBindBuffer()

- Funkce je určena k navázání bufferu ID jakožto aktuálního bufferu. Pokud je ID nula, navázaný buffer se přestane používat.

- Parametry:
  - *target* – typ bufferu, na který je buffer objekt navázán. Nabývá jednu z konstant.
  - *buffer* – specifikuje jméno buffer objektu (z kroku 1)

glBindBuffer() - target

- BO pracuje se dvěma typy bufferů:
- Array buffers (GL_ARRAY_BUFFER)
  - Tyto buffery obsahují atributy vrcholu (tedy souřadnice vrcholů, texturové souřadnice, normály, barvu vrcholů)
- Element array buffers (GL_ELEMENT_ARRAY_BUFFER)
  - Tyto buffery obsahují indexy na vrcholy

glBindBuffer() - target

- Další cíle:

Pixel Buffer Object (PBO): `GL_PIXEL_PACK_BUFFER`, `GL_PIXEL_UNPACK_BUFFER`

Kopírování dat mezi buffery:

`GL_COPY_READ_BUFFER`,
`GL_COPY_WRITE_BUFFER`

Texturová data uložená jako texturový buffer:

`GL_TEXTURE_BUFFER`

Použití v shaderech (GLSL):

`GL_TRANSFORM_FEEDBACK_BUFFER`

Použití v shaderech (GLSL):

`GL_UNIFORM_BUFFER`

krok 2 - glBindBuffer()

- Příklad:

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

krok 3 - glBufferData()

- Parametry:
  - *target* – specifikuje typ buffer objektu
  - *size* – množství paměti požadované pro uložení dat
  - *data* – buď ukazatel na data, která mají být zkopírována do nového datového uložiště, nebo NULL (paměť se pouze rezervuje)
  - *usage* – způsob použití bufferu - `GL_STREAM_*`, `GL_STATIC_*`, `GL_DYNAMIC_*`
- Dva způsoby použití:
  - Alokuje paměť, nastaví použití a data nastaví na NULL. Uživatel data objektu namapuje později.
  - Alokuje paměť, nastaví použití a zkopíruje data (především když pracujeme se statickým datovým modelem).

glBufferData - usage

- BO pracuje s těmito typy příznaků pro parametr *usage*:

  `GL_STREAM_*`

  `GL_STATIC_*`

  `GL_DYNAMIC_*`

Kde `*` může být:

`READ`, `DRAW` nebo `COPY`

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 vrcholů z pole triangles
- Zavolání glBufferData() na buffer, který již obsahuje data, smaže jeho uložená data

Další BO funkce

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

- Kopíruje data daného rozsahu do buffer objektu

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

- Obdrží podmnožinu dat daného rozsahu z aktuálního buffer objektu

Příklad přepsání celého bufferu

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

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

glMapBuffer(), glUnmapBuffer()

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

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

- Funkce pro „odemknutí a zamknutí“ bufferů – umožní natažení dat do bufferů nebo přenechání kontroly serveru
- *access* – přístup do paměti: `READ_ONLY`, `WRITE_ONLY`, `READ_WRITE`.

krok 4 – využití BO (vykreslení)

- Pro VBO nastavíme data pro vykreslení:
  - `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);
```

- Smazání či dotazování na identifikátor buffer objektu

Zpět k vykreslení trojúhelníků …

- Vytvoření VBO

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

- Alokace paměti a nahrání dat z našeho pole na GPU

```
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices_position),
vertices_position, GL_STATIC_DRAW);
```

Zpracování pomocí vertex shaderu

- Vytvoření globální proměnné typu vec4
- Po vložení hodnot x a y: (x,y,0,1)

```
in vec4 position;

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

Další atributy vrcholů a jejich propojení s vrcholy

- Jednotlivé vrcholy mohou mít další atributy – barva, souřadnice textury, normály, …

- Propojení atributů jednotlivých vrcholů s vrcholy samotnými – pomocí **Vertex Array Objects (VAO)**

Vertex Array Objects (VAO)

- Vertex arrays = sada uživatelem definovaných polí, které obsahují atributy jednotlivých vrcholů
- Vertex array objects = objekt, který obsahuje informaci o tom, ve kterých bufferech (a jak) jsou uložena data těchto atributů

Vytvoření a navázání VAO

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

- Generuje *n* aktuálně nepoužívaných jmen, která jsou uložena do pole *arrays*

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

- 2 funkce:
  - Sváže VAO se jménem v *array*
  - Navázání na objekt s hodnotou *array* 0 ukončí používání VAO (přestane používat aktivní VAO)

Smazání VAO

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

- Smaže *n* VAO specifikovaných v parametru *arrays*
- Uvolněná jména VAO je možné v budoucnu opět použít

Zjištění stavu VAO

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

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

Zpracování atributu

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

1. Získání indexu

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

- Vrací hodnotu >= 0 nebo -1, pokud tento atribut nebylo možné najít

2. Povolení atributu

- Povolení používání atributů

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

  - *idx* = index atributu vrcholů

- Zakázání užívání

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

3. Navázání bufferu
4. Nastavení způsobu uložení

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

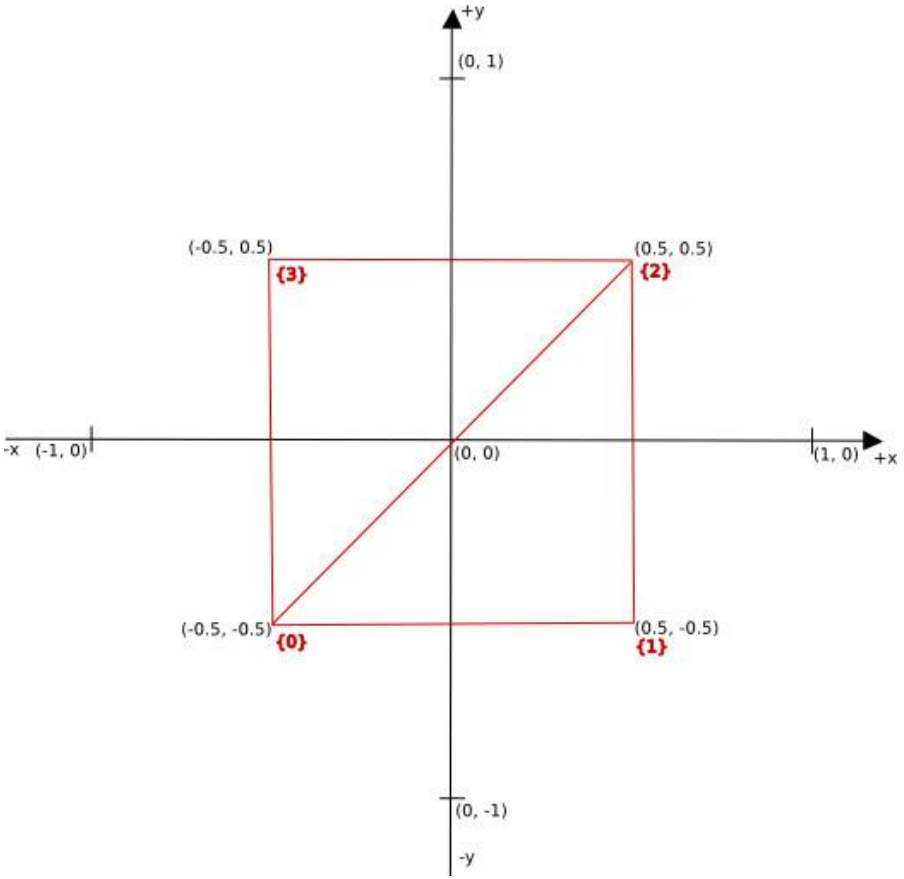
```
glVertexAttrib[IL]Pointer (GLuint index,
GLint size, GLenum type, GLboolean
normalized, GLsizei stride, const GLvoid *
pointer)
```

Význam parametrů

- *index* = index atributu vrcholů, který chceme měnit
- *size* = počet komponent, které jsou zapotřebí pro uložení daného atributu
- *type* = datový typ každé komponenty v poli
- *normalized* = specifikuje, jestli mají být datové hodnoty při přístupu k nim normalizovány (GL_TRUE) nebo ne (GL_FALSE)
- *stride* = offset mezi po sobě následujícími atributy vrcholů
- *pointer* = offset první komponenty prvního atributu vrcholů

Práce s indexy

- Problém s opakujícími se vrcholy v geometrii




```
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
};
```

Práce s indexy

- Indexy jsou uloženy v buffer objektu GL_ELEMENT_ARRAY_BUFFER

Nastavení bufferu obsahujícího indexy

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

- Nastavení bufferu, ze kterého bereme indexy
- 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 způsoby:
  - glDrawElements(GLenum mode, GLsizei count, GLenum type, const void *indices)
  - glDrawArrays(GLenum mode, GLint first, GLsizei count)

Vykreslení geometrie dat

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

- *mode* určuje druh vytvářených primitiv
- *count* počet vrcholů, které budu kreslit
- *type* je buď GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT nebo GL_UNSIGNED_INT a určuje datový typ indexového pole
- *indices* offset v bytech, který určuje místo v bufferu indexů, ze kterého indexy bereme

Vykreslení geometrie dat

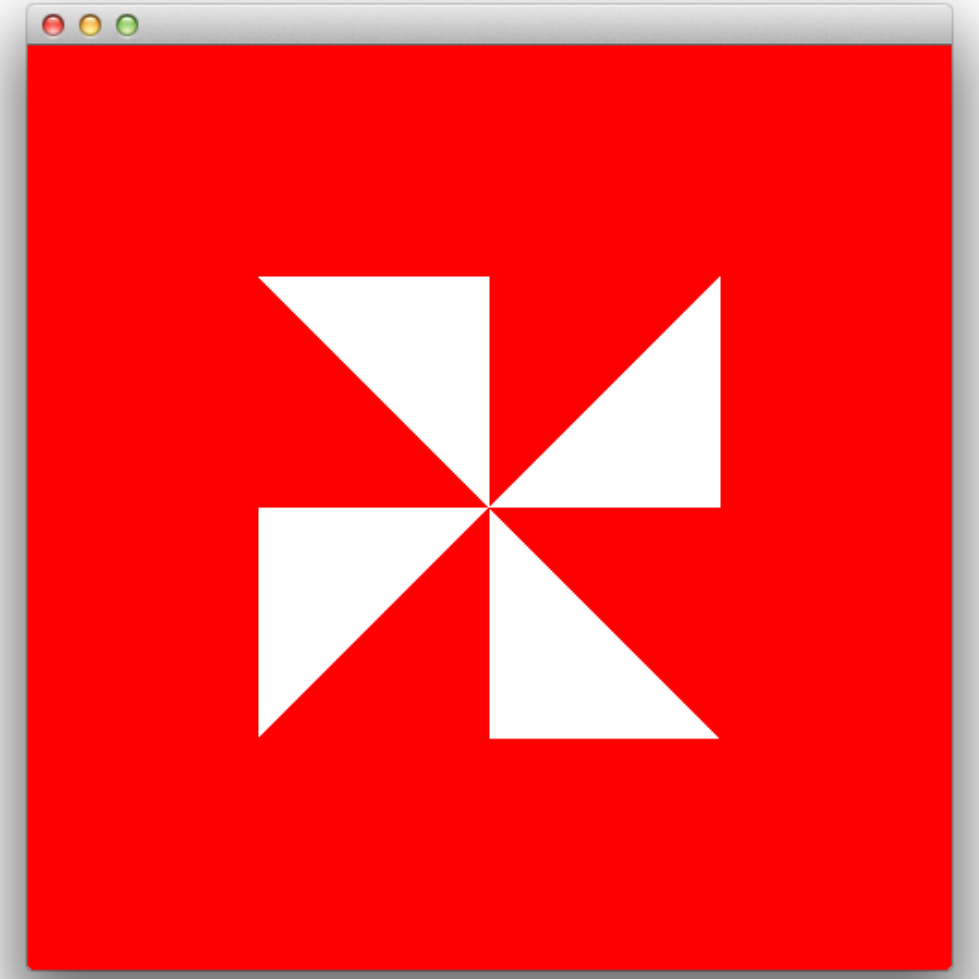
```
void glDrawArrays(GLenum mode, GLint first, GLsizei count);
```

- Dereference sekvenčního pole prvků
- Vytvoří sekvenci geometrických primitiv s použitím prvků polí počínaje *first* a konče *first + count –1*.
- Nepotřebuje použití index bufferu

Příklad použití VAO pro vykreslení našich trojúhelníků

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

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

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

Výsledek



Další úpravy

- Zobrazení bodů místo trojúhelníků

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

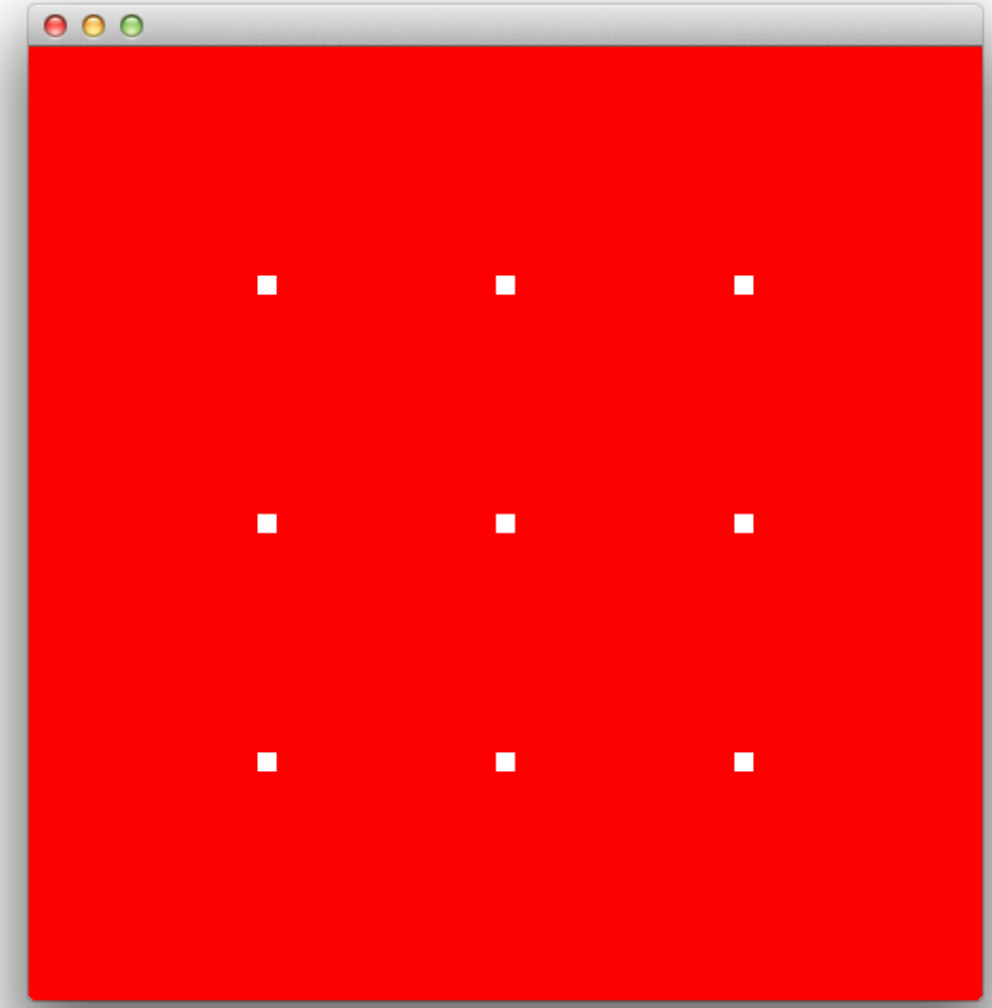
- Pro určení velikosti vykreslených bodů je třeba tuto funkci zpřístupnit při inicializaci

```
glEnable(GL_PROGRAM_POINT_SIZE);
```

- Poté je možné měnit velikost bodů ve vertex shaderu

```
gl_PointSize = 10.0;
```

Výsledek

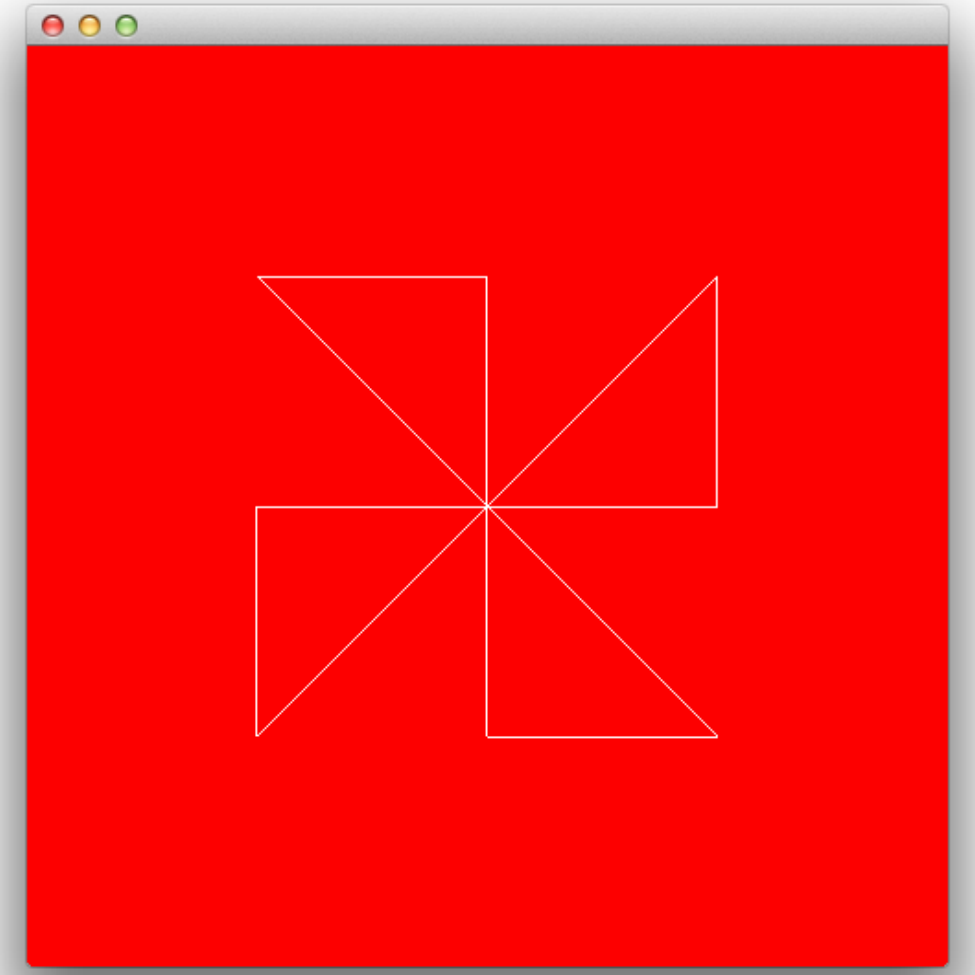


Další úpravy

- Defaultně jsou trojúhelníky vyplněny barvou, toto lze ale změnit pomocí

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

Výsledek



Změna kreslícího módu

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