Základy počítačové grafiky
Přednáška 2

Martin Němec

VŠB-TU Ostrava

2024

OpenGL

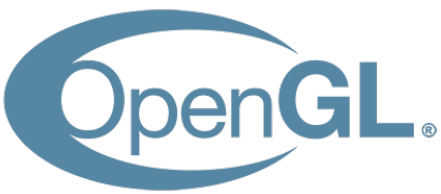
Open Graphics Library (OpenGL) je multiplatformní rozhraní (API) pro tvorbu grafických aplikací. Od roku 2006 spadá pod Kronos Group.

- první vydání v červnu 1992 (Silicon Graphics);
- zaměřen čistě na vykreslování (neřeší ovládání, okna atd.);
- během vývoje postupně upravován (fixní a programovatený);
- OpenGL je API, rozhranní pro práci s grafickou kartou (přes 120 funkcí);
- Opengl není programovací jazyk;
- aktuální verze 4.6 (31.7.2017).

OpenGL

Mimo OpenGL existují další:

- **OpenGL** - nedokáže plně využít výkon nejnovějších grafických karet.
- **Vulkán** - nízkoúrovňové API, využití výkonu a paralelizace.
- **DirectX** - Microsoft od roku 1995, Windows, XBOX, aktuálně DirectX 12.
- **Metal** - grafické API pro Apple.



Vývoj OpenGL

První verze **OpenGL 1.0** (1994) měla fixní vykreslovací řetězec (Fixed Function Pipeline), uživatel může spouštět nebo nastavovat předem vytvořené funkce ale nemůže programově zasahovat do vykreslování.

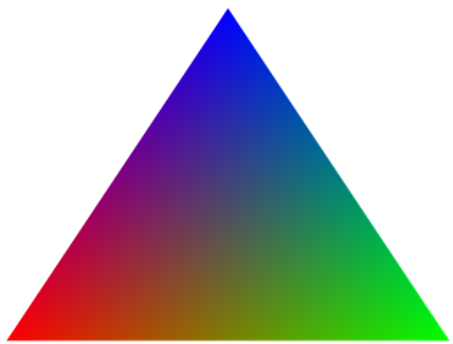
- Vychází z rozhraní IrisiGL;
- činnost se řídí voláním funkcí a procedur;
- S3 Graphics - S3 KCZ-S3805.

OpenGL 1.0

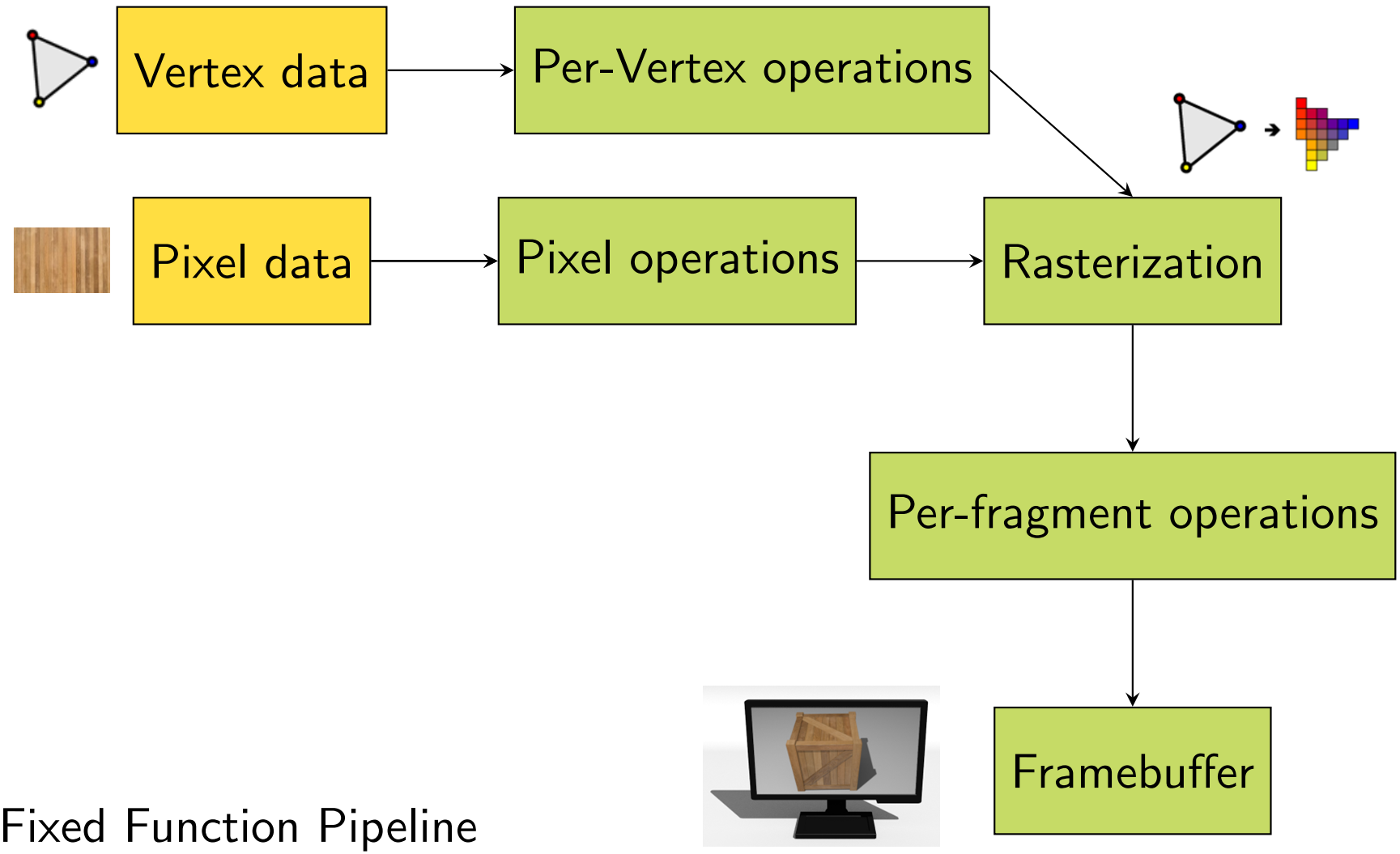
Fixní řetězec obsahuje pevné kroky (transformace, osvětlení, textury, rasterizace, viditelnost, míchání atd.).

```
glClear(GL_COLOR_BUFFER_BIT);

glBegin(GL_TRIANGLES);
  glColor3f(1.f, 0.f, 0.f);
  glVertex3f(-0.6f, -0.4f, 0.f);
  glColor3f(0.f, 1.f, 0.f);
  glVertex3f(0.6f, -0.4f, 0.f);
  glColor3f(0.f, 0.f, 1.f);
  glVertex3f(0.f, 0.6f, 0.f);
glEnd();
glFlush();
```



OpenGL 1.0



Fixed Function Pipeline

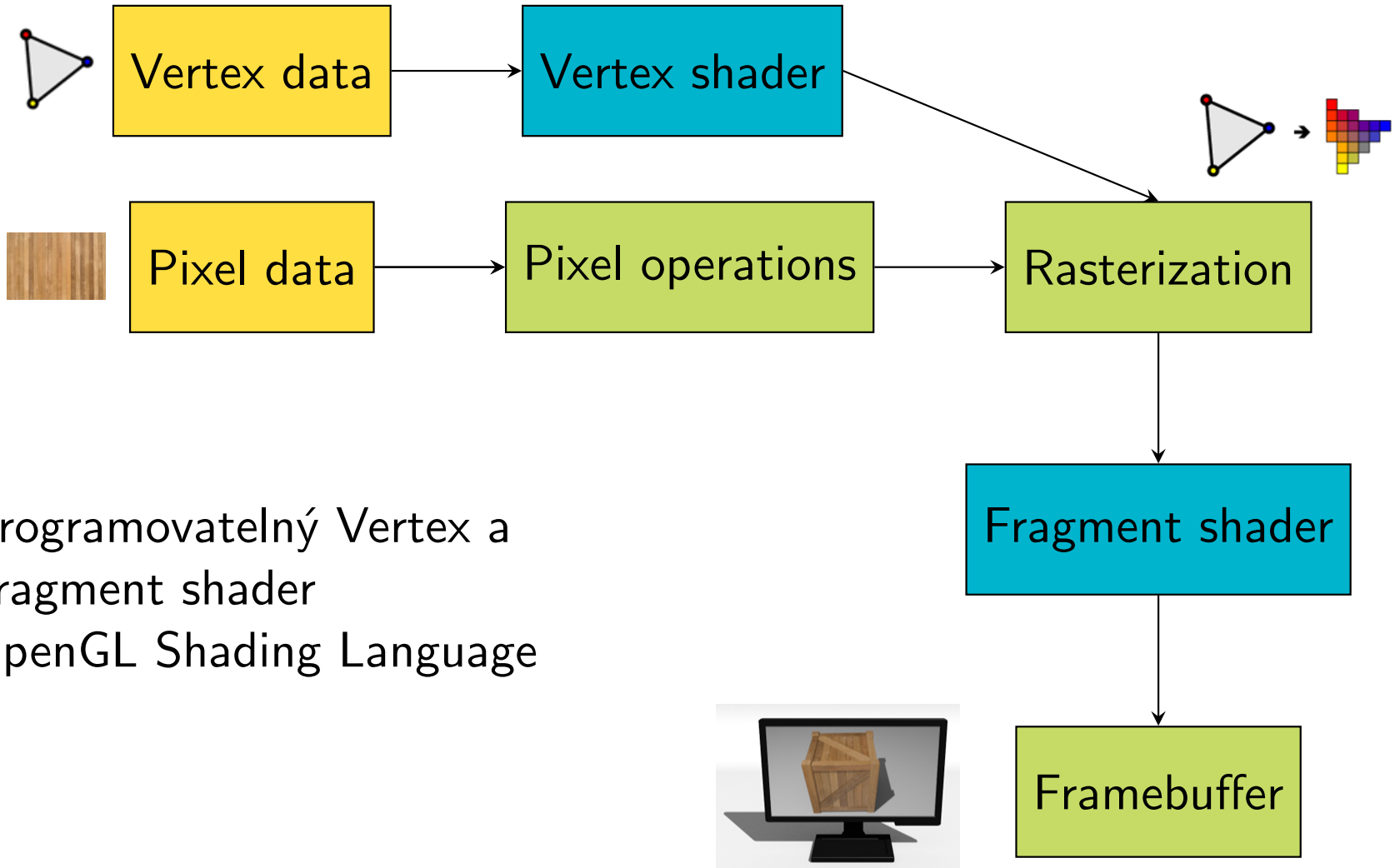
OpenGL 2.0 (2004)

V této verzi dochází k rozšíření o programovatelnou část - možnost používat shadery. **Shader** - počítačový program určený pro řízení konkrétní části vykreslovacího řetězce (GPU).
Dva typy shaderu:

- (Ne)unifikované shadery - pevně daný počet shaderu. (GeForce 7800 GTX: 8 vertex a 24 fragment jednotek).
- Unifikované shadery – možnost měnit dynamicky typ shaderu podle potřeby. Nemohou být tak efektivní.

Zdrojový kód pro vykreslovací řetězec lze vytvářet a upravovat (použijeme jazyk GLSL - OpenGL Shading Language).

OpenGL 2.0 (2004)



Programovatelný Vertex a
Fragment shader
OpenGL Shading Language

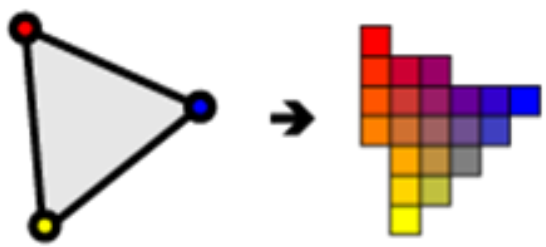
OpenGL 2.0 (2004)

Vertex shader (vrcholový procesor) - typicky provádí transformace (vrcholů, normál, texturovacích souřadnic atd.).

- Mění polohu vrcholu, nemůže ani přidávat ani mazat vrcholy.
- Nemá informaci o tělese (nezná sousední vrcholy).

Fragment shader (fragmentový procesor) typicky výpočet výsledné barvy pro jednotlivé fragmenty.

- Čtení z texturovacích jednotek.
- Výpočet osvětlení, mlhy atd.
- Nelze změnit pozici fragmentu (pouze barvu).



Vertex Shader

- Provádí se zde obvykle transformace jednotlivých vrcholů vstupního tělesa (např. posun, rotaci, scale atd.).
- Používá se zde homogenní souřadný systém (více na další přednášce), proto čtyři souřadnice.
- **gl_Position** - speciální proměnná, kde nastavujeme výslednou pozici vrcholu.

```
#version 330
layout(location = 0)in vec3 position;

void main(void)
{
    //nastavuje promennou gl_position
    gl_Position = vec4(position, 1.0);
}
```

Fragment Shader

- Výpočet výsledné barvy daného fragmentu (pixelu).
- Fragment je objekt vzniklý po rasterizaci.
- Může obsahovat pozici, barvu, normálu atd.
- Pixel se může skládat z více fragmentu (anti-aliasing).
- Později budeme počítat barvu pomocí osvětlovacího modelu.

```
#version 330

void main(void)
{
    //musi nastavovat promennou gl_FragColor
    gl_FragColor = vec4(0.0,1.0,0.0,1.0);
}
```

OpenGL 3.0 (2008)

Zaveden deprecation model (potřeba odstranit zastaralé funkce OpenGL).
OpenGL verze 3.1 odstraňuje fixní vykreslovací řetězec (nutno použít shadery).
Data se ukládají přes Buffer objekty na grafickou kartu.

```
glBegin(); glEnd();
glVertex();
glLoadIdentity();
glTranslatef(0.0f, 0.0f, -10.0f);
glRotatef(20.0f, 0.0f, 1.0f, 0.0f);
gluPerspective(70, w/h, 0.1f, 100.0f);
gluLookAt(); gluOrtho2D();
//vestavene zasobniky na matice
glMatrixMode(GL_PROJECTION);
glMatrixMode(GL_MODELVIEW);
```

OpenGL 3.2 (2009)

Rozšiřuje OpenGL o geometrický shader (Geometry shader), který umožňuje měnit geometrii v přímo během vykresování.

- Umožňuje využívat informace o sousedních vrcholech.
- Umožňuje vytvářet, měnit nebo mazat základní vrcholy nebo objekty.

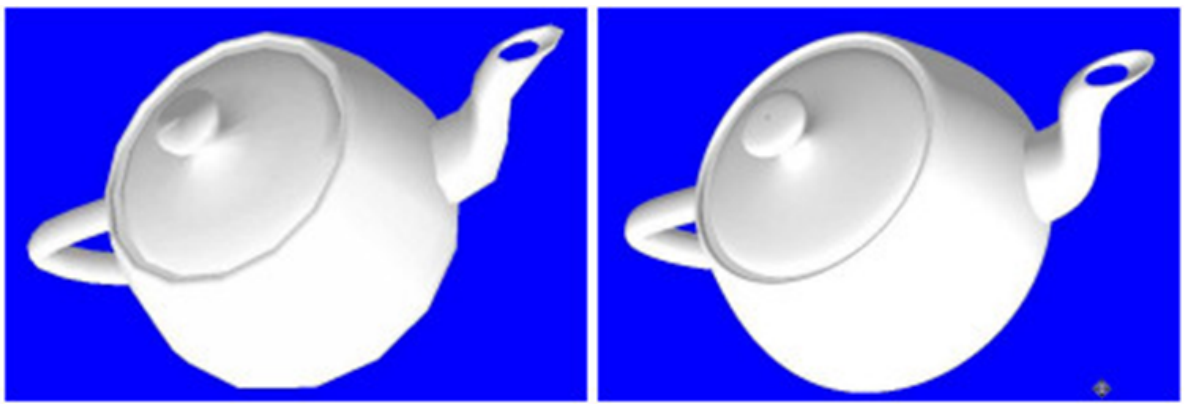
```
#version 330 core
layout (points) in;
layout (triangle_strip, max_vertices = 3) out;

void main() {
    gl_Position = in[0].gl_Position + vec4(-1,0,0,0);
    EmitVertex(); // prida vrchol
    gl_Position = in[0].gl_Position + vec4( 1,0,0,0);
    EmitVertex(); // prida vrchol
    gl_Position = in[0].gl_Position + vec4( 0,1,0,0);
    EmitVertex(); // prida vrchol
    EndPrimitive();
}
```

OpenGL 4.0 (2010)

Rozšiřuje OpenGL o teselační shader (Tesselation shader) - zjemnění (zaoblení) geometrického modelu (nezaměňovat s displacement mappingem).

- Umožňuje rozdělovat polygony na menší a zaoblit tak výsledný tvar.
- Možnost použití Beziérových plátů.



Posílání proměnných mezi shadery

```
#version 330
layout(location = 0)in vec3 in_Position;
layout(location = 1)in vec3 in_Color;
out vec3 color; //vystupni promenna
uniform mat4 MVP;

void main(void)
{
    gl_Position = MVP * vec4(in_Position, 1.0);
    color = in_Color;
}
```

```
#version 330
out vec4 frag_colour;
in vec3 color; //vstupni promenna
void main(void)
{
  gl_FragColor = vec4(color,1.0);
}
```

Shader program

Vytvoření vertex a fragment shaderu a kompilace výsledného shader programu.

```
//create and compile shaders

GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, &vertex_shader, NULL);
glCompileShader(vertexShader);

GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fragment_shader, NULL);
glCompileShader(fragmentShader);

GLuint shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, fragmentShader);
glAttachShader(shaderProgram, vertexShader);
glLinkProgram(shaderProgram);
```

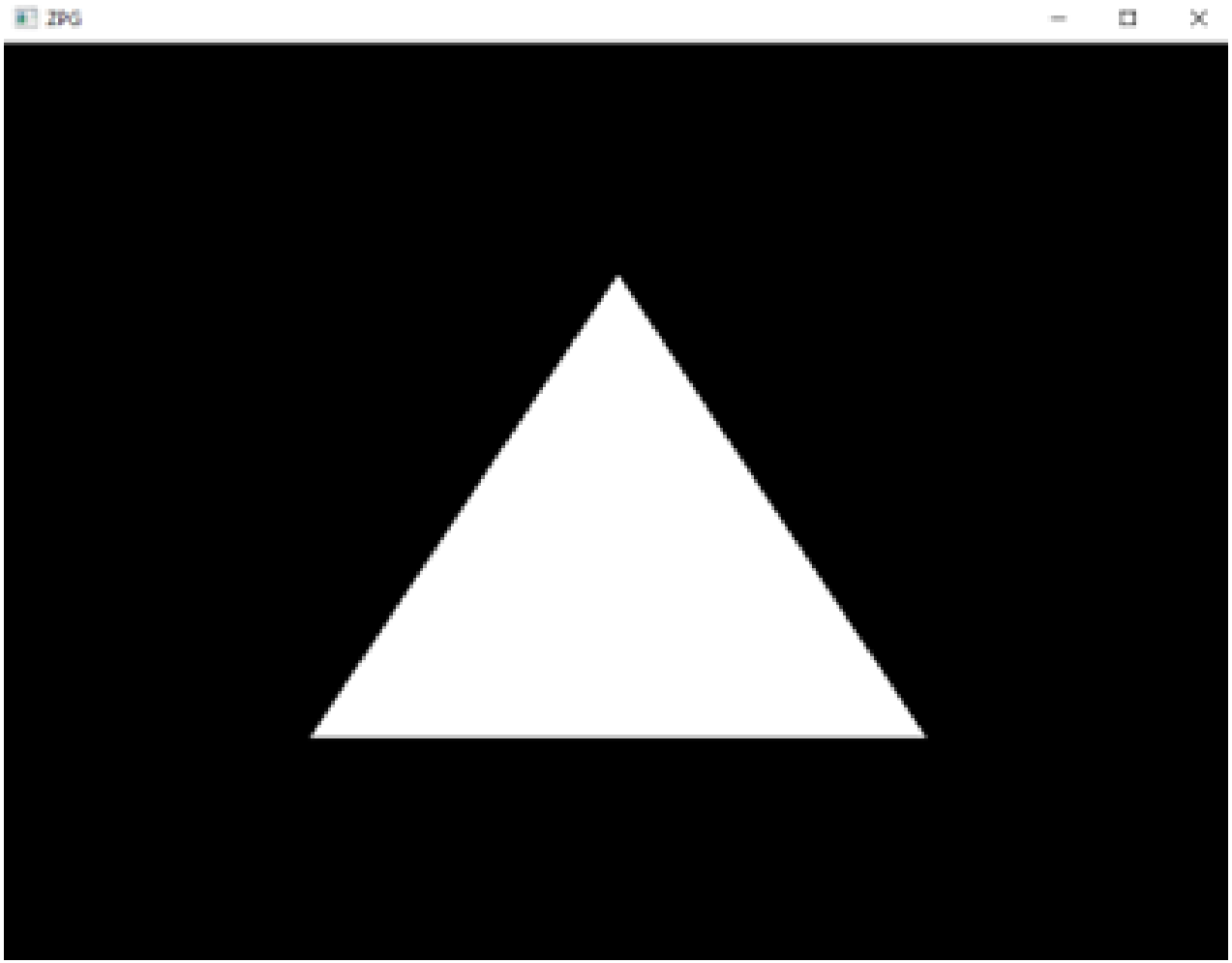
Shader program

Program doplňte o kontrolu kompilace a linkování program shaderu.

```
GLint status;
glGetProgramiv(shaderProgram, GL_LINK_STATUS, &status);
if (status == GL_FALSE)
{
  GLint info;
  glGetProgramiv(shaderProgram,GL_INFO_LOG_LENGTH,&info);
  GLchar *strInfoLog = new GLchar[infoLogLength + 1];
  glGetProgramInfoLog(shaderProgram,info,NULL,strInfoLog);
  fprintf(stderr,"Linker␣failure:␣%s\n",strInfoLog);
  delete[] strInfoLog;
}
```

Shader program

V případě chyby lze vidět třeba toto.



Možnost nastavení konkrétní verze OpenGL a závislosti na deprecated funkcích.

- **Core Profile** - není zpětně kompatibilní (bez fixed pipeline);
- **Compatibility Profile** - zachovává zpětnou kompatibilitu se staršími verzemi OpenGL.

```
//inicializace konkretni verze pomoci GLFW
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
glfwWindowHint(GLFW_OPENGL_PROFILE,
    GLFW_OPENGL_CORE_PROFILE);
```

Pokud nechceme vybrat konkrétní verzi, inicializuje se maximální verze.

Vykreslování objektů v OpenGL

Vývoj grafických karet (např. rostoucí paměť) ovlivnil i možnosti vykreslování objektů v OpenGL.

- Konstrukce glBegin-glEnd
- Display list
- Vertex Arrays
- Vertex Buffer Object

konstrukce glBegin-glEnd

Lze použít možnost redukce dalšími primitivy (TRIANGLE FAN, TRIANGLE STRIP atd.) nebylo nutné přenášet duplicitní vrcholy.

- Velmi časově drahé při vykreslování, hodnoty posílány opakovaně.
- Od verze 3.1 patří mezi deprecated příkazy.

```
glBegin(GL_TRIANGLES);
  glColor3f(1.f, 0.f, 0.f);
  glVertex3f(-0.6f, -0.4f, 0.f);
  glColor3f(0.f, 1.f, 0.f);
  glVertex3f(0.6f, -0.4f, 0.f);
  glColor3f(0.f, 0.f, 1.f);
  glVertex3f(0.f, 0.6f, 0.f);
glEnd();
```

Display List

Objekty lze zapsat do display listu a v případě potřeby vykreslit jedním příkazem.

- V případě úpravy objektu bylo nutno vytvořit nový display list.
- Od verze 3.1 patří mezi deprecated příkazy.

```
GLuint index = glGenLists(1); // create display list

glNewList(index, GL_COMPILE);
   glBegin(GL_TRIANGLES);
     glVertex3f(-0.6f, -0.4f, 0.f);
     glVertex3f(0.6f, -0.4f, 0.f);
     glVertex3f(0.f, 0.6f, 0.f);
   glEnd();
glEndList();
...
glCallList(index); // draw
```

Vertex Array

Data se ukládají do polí, k dispozici bylo šest polí:

- Vrcholy, normály, barvy, indexy barev, uv, nastavení hrany.
- glVertexPointer, glColorPointer, glTexCoordPointer atd.

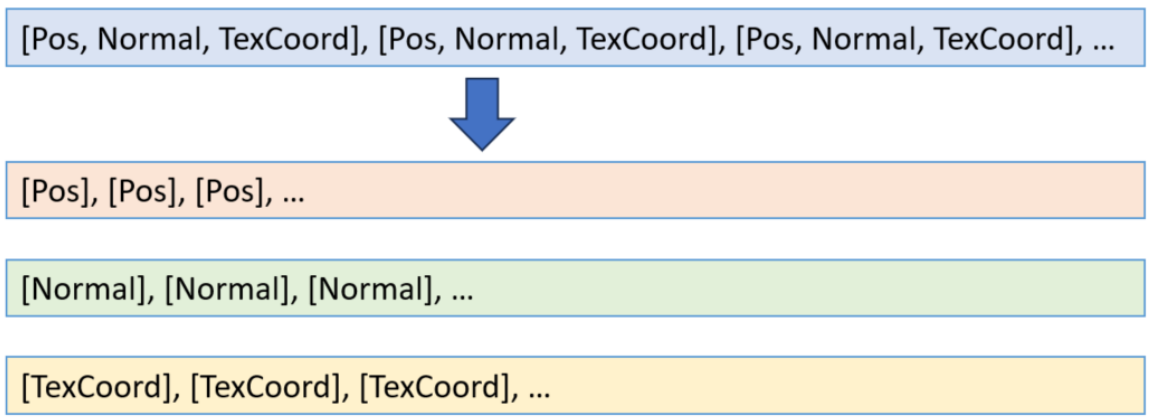
Při každém vykreslení se musí data poslat na grafickou kartu. Od verze 3.1 patří mezi deprecated příkazy.

Vertex Buffer Objects + Vertex Array Object

Data se ukládají do pole (dynamické objekty) VBO (Vertex Buffer Object). Lze je vykreslit jedním příkazem. Nutné definovat jak pole dělit VAO (Vertex Array Object).

```
float points[] = {
    0.0f, 0.5f, 0.0f,
    0.5f, -0.5f, 0.0f,
   -0.5f, -0.5f, 0.0f};
...
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
//(index, pocet, typ, normalized, posun, pocatek)

glDrawArrays(GL_TRIANGLES, ...);
```



Vertex Buffer Objects + Vertex Array Object

Data objektu se mohou skládat z více částí (pozice, normála, barva, uv souřadnice atd.), pak budeme muset pro každý atribut vytvořit nový objekt VertexAttribArray.

```
const a[] ={
    -.5f,-.5f, .5f, 1, 1, 1, 0, 1,
    -.5f, .5f, .5f, 1, 1, 0, 0, 1,
     .5f, .5f, .5f, 1, 0, 0, 0, 1,
     .5f,-.5f, .5f, 1, 0, 1, 0, 1,
};
...
glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);
glVertexAttribPointer(
    0,4,GL_FLOAT,GL_FALSE,sizeof(float)*8,(GLvoid*)0);
glVertexAttribPointer(
    1,4,GL_FLOAT,GL_FALSE,sizeof(float)*8,(GLvoid*)size...);
...
glDrawArrays(GL_TRIANGLES, ...)
```

Vertex Buffer Objects + Vertex Array Object

- Dnes se používá kombinace VBO (Vertex Buffer Object) a VAO (Vertex Array Object).
- Data uložena přímo v paměti grafické karty (snížení času potřebného pro rendering).
- Data obvykle obsahují jednotlivé vrcholy, normály, texturovací souřadnice apod.
- Existuje i tzv. index buffer s indexy určujícími jednotlivé plochy (trojúhelníky). Vhodné při opakování vrcholů.

Vykreslení bez IBO: **glDrawArrays**(...);
Vykreslení s IBO: **glDrawElements**(...);

Vertex Buffer Objects

V moderním OpenGL se modely předávají mezi CPU a GPU pomocí VBO+VAO.
Vertex buffer objekty (VBO)

- pojmenovaný blok v paměti (varianty VBO, EBO atd.);
- obecně jednorozměrné pole vrcholů, které může mít více informací (pozici, normály, barvu atd.).

Vertex array objekty (VAO)

- Určuje, jak se bude s daty ve VBO zacházet.
- Může obsahovat jeden nebo více VBO.

VAO+VBO

```
float points[] = { \\VBO
    0.0f, 0.5f, 0.0f,
    0.5f, -0.5f, 0.0f,
   -0.5f, -0.5f, 0.0f};

//Vertex Buffer Object (VBO)
GLuint VBO = 0;
glGenBuffers(1, &VBO); // generate the VBO
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER,sizeof(points),
             points,GL_STATIC_DRAW);

//Vertex Array Object (VAO)
GLuint VAO = 0;
glGenVertexArrays(1, &VAO); //generate the VAO
glBindVertexArray(VAO); //bind the VAO
glEnableVertexAttribArray(0); //enable vertex attributes
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
```

Čtyřúhelník

Další typy modelů, vrcholy, normály, texturovací souřadnice atd.

```
//6
//glDrawArrays(GL_TRIANGLES, 0, 6);

const float plain[36] = {
     1.0f, 0.0f,  1.0f, 0.0f, 1.0f, 0.0f,
     1.0f, 0.0f, -1.0f, 0.0f, 1.0f, 0.0f,
    -1.0f, 0.0f, -1.0f, 0.0f, 1.0f, 0.0f,
    -1.0f, 0.0f,  1.0f, 0.0f, 1.0f, 0.0f,
     1.0f, 0.0f,  1.0f, 0.0f, 1.0f, 0.0f,
    -1.0f, 0.0f, -1.0f, 0.0f, 1.0f, 0.0f
};
```

Koule

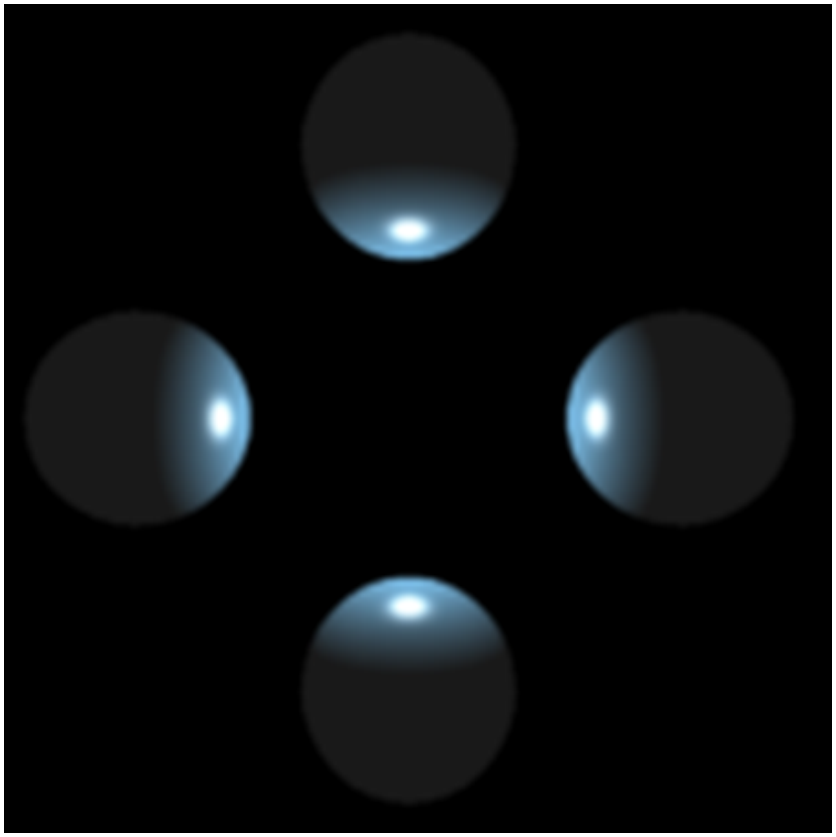
Další typy modelů, vrcholy, normály, texturovací souřadnice atd.

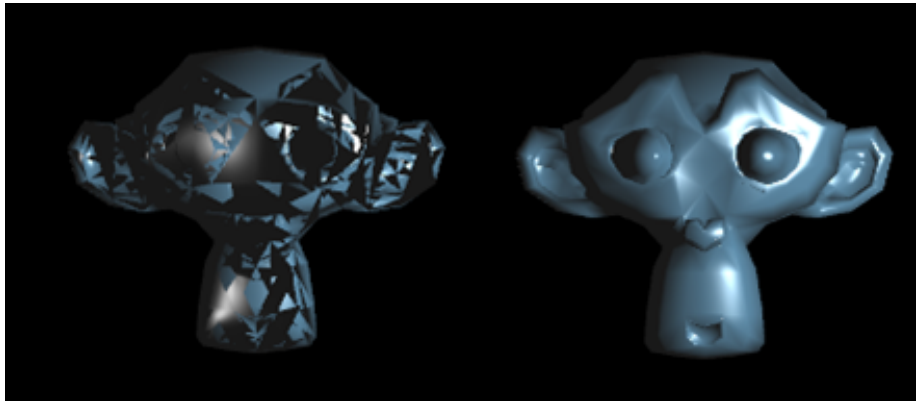
```
//2880
//glDrawArrays(GL_TRIANGLES, 0, 2880);


const float sphere[17280] = {
-0.83147,-0.55557, 0.00000,-0.83333,-0.55275, 0.00000,
-0.92388,-0.38268, 0.00000,-0.92474,-0.38053, 0.00000,
-0.81549,-0.55557,-0.16221,-0.81731,-0.55275,-0.16257,
-0.92388, 0.38268, 0.00000,-0.92474, 0.38053, 0.00000,
-0.83147, 0.55557, 0.00000,-0.83333, 0.55275, 0.00000,
...
-0.55557, 0.83147, 0.00000,-0.55977, 0.82863, 0.00000,
-0.38268,-0.92388, 0.00000,-0.38786,-0.92169, 0.00000,
-0.37533,-0.92388, 0.07465,-0.38041,-0.92169, 0.07565,
-0.55557,-0.83147, 0.00000,-0.55977,-0.82863, 0.00000
};
```

Testovací modely

Více modelů, jak vytvořit kód?





Vykreslovací smyčka while

```
while (!glfwWindowShouldClose(window))
{
    // clear color and depth buffer
     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
     glUseProgram(shaderProgram);
     glBindVertexArray(VAO);
     // draw triangles
     glDrawArrays(GL_TRIANGLES, 0, 3); //mode,first,count
     // update other events like input handling
     glfwPollEvents();
     glfwSwapBuffers(window);
}
                        //multiple models
                        glUseProgram(shaderProgram1);
                        glBindVertexArray(VAO1);
                        glDrawArrays(GL_TRIANGLES, 0, 3);

                        glUseProgram(shaderProgram2);
                        glBindVertexArray(VAO2);
                        glDrawArrays(GL_TRIANGLES, 0, 3);
```

Shrnutí

```
float points[] = {
        0.0f, 0.5f, 0.0f,
        0.5f, -0.5f, 0.0f,
   -0.5f, -0.5f, 0.0f
};

const char* vertex_shader =
    #version 330
    layout(location=0) in vec3 vp;
    void main () {
        gl_Position = vec4 (vp, 1.0);
};

const char* fragment_shader =
    #version 330
    out vec4 frag_color;
    void main () {
        frag_color = vec4 (1.0, 1.0, 0.0, 1.0);
};
```

Shrnutí

```
/* //inicializace konkretni verze
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
glfwWindowHint(GLFW_OPENGL_PROFILE,
GLFW_OPENGL_CORE_PROFILE);  //*/

window = glfwCreateWindow(800, 600, "ZPG", NULL, NULL);
if (!window){
    glfwTerminate();
    exit(EXIT_FAILURE);
}

glfwMakeContextCurrent(window);
glfwSwapInterval(1);

// start GLEW extension handler
glewExperimental = GL_TRUE;
glewInit();
```

Shrnutí

```
//vertex buffer object (VBO)
GLuint VBO = 0;
glGenBuffers(1, &VBO); // generate the VBO
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(points), points,
    GL_STATIC_DRAW);

//Vertex Array Object (VAO)
GLuint VAO = 0;
glGenVertexArrays(1, &VAO); //generate the VAO
glBindVertexArray(VAO); //bind the VAO
glEnableVertexAttribArray(0); //enable vertex attributes
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
```

Shrnutí

```
//create and compile shaders
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, &vertex_shader, NULL);
glCompileShader(vertexShader);
GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fragment_shader, NULL);
glCompileShader(fragmentShader);
GLuint shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, fragmentShader);
glAttachShader(shaderProgram, vertexShader);
glLinkProgram(shaderProgram);

//kontrola spravnosti shaderu
```

Shrnutí

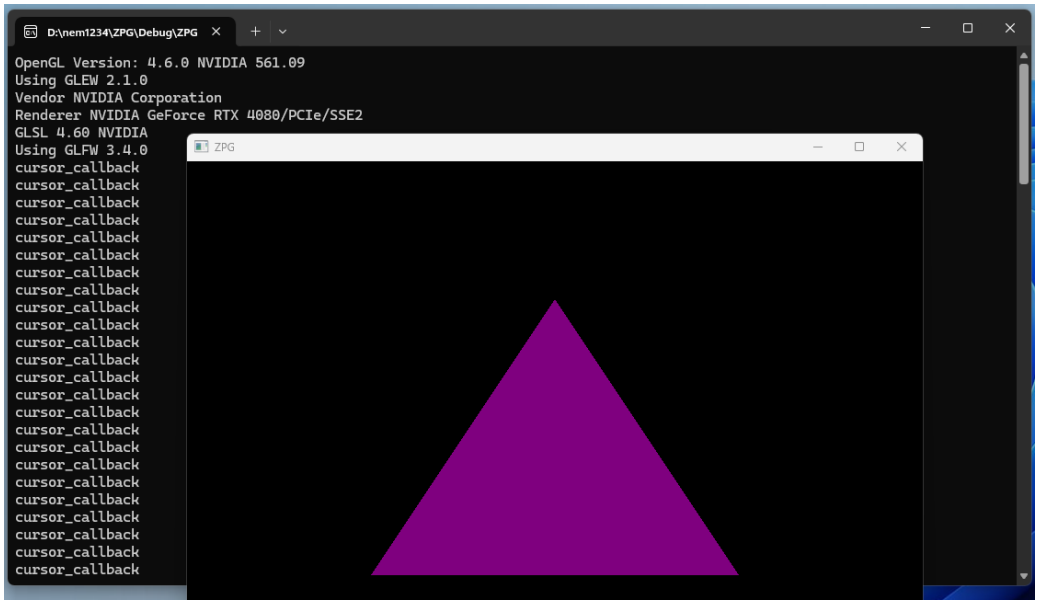
```
while (!glfwWindowShouldClose(window)) {

// clear color and depth buffer
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
// draw triangles
glDrawArrays(GL_TRIANGLES, 0, 3); //mode,first,count
// update other events like input handling
glfwPollEvents();
// put the stuff  w e ve  been drawing onto the display
glfwSwapBuffers(window);

}
```

Úkoly na cvičení

- Upravte projekt na moderní OpenGL.
- Upravte model na složitější (více modelů, více shaderu).
- Vytvořte objektový kód.
- Jaké třídy by jste očekávali?
- Zopakujte si základní návrhové vzory (Singleton, Observer, Factory atd.).



Závěr

Dotazy?