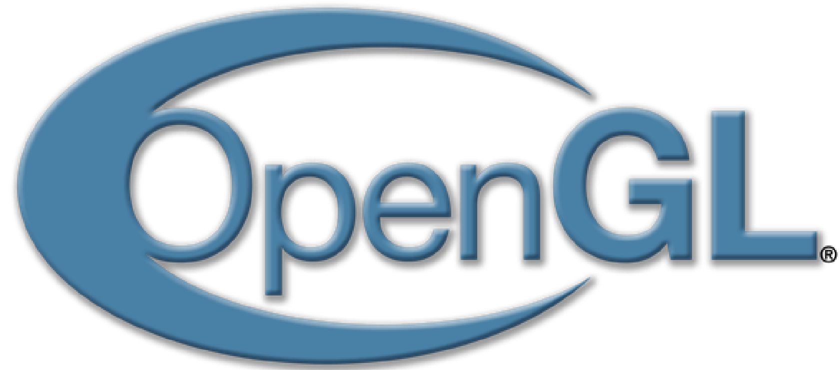
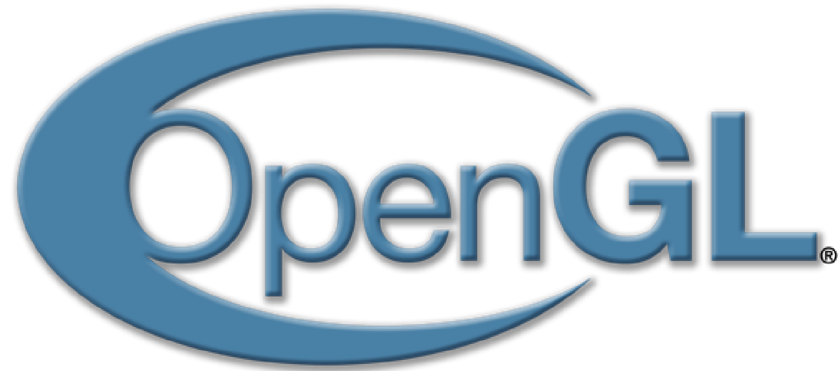


PV112 – Programování grafických aplikací

OpenGL (Open Graphics Library)

Barbora Kozlíková
xkozlik@fi.muni.cz



PV112 – Programování grafických aplikací

1. přednáška - Úvod

Struktura předmětu

- Přednášky – každé pondělí v 8:00 v D2
- Cvičení
  - seminární skupiny v Javě a C++
  - cvičící: Zuzana Ferková, Adam Matoušek, Martin Matouš, Adam Jurčík, Pavel Kouřil, Tomáš Kovanda
- Specifika jednotlivých jazyků a napojení OpenGL na ně budou probírána zejména na cvičeních

Požadavky na zápočet a zkoušku

- Zápočet
  - 2 projekty obsahující důležité prvky, které budou probírány na přednášce a procvičeny na cvičeních. Zadání bude dodáno během kurzu.
- Zkouška
  - Zápočet
  - Písemná forma

www.thegamergene.com

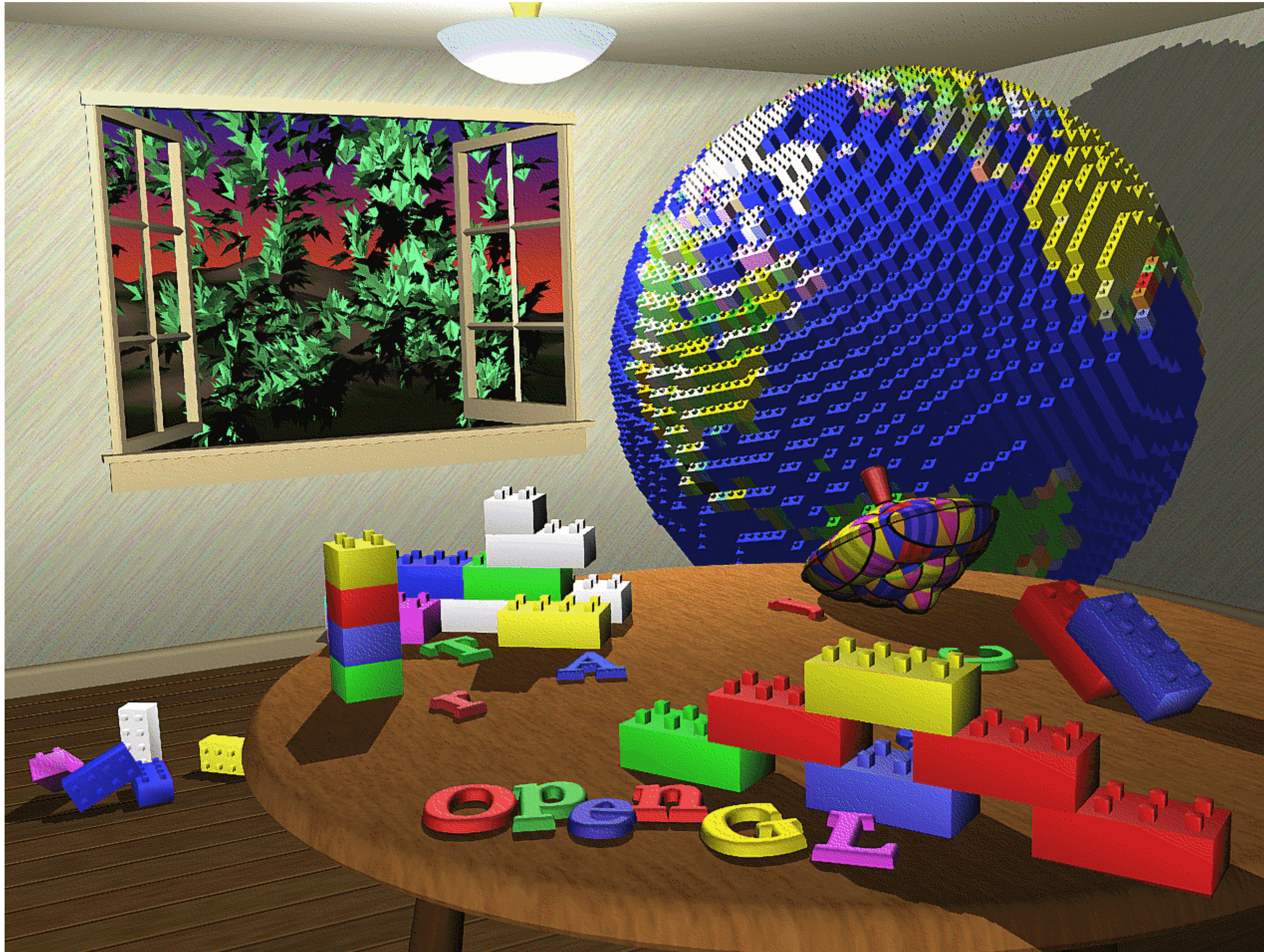
Studijní literatura

- OpenGL Shading Language (tzv. Orange Book)
  - http://wiki.labomedia.org/images/1/10/Orange_Book_-_OpenGL_Shading_Language_2nd_Edition.pdf
  - https://www.opengl.org/sdk/docs/man4/
- Doprovodné studijní materiály
- Tutoriály na webu

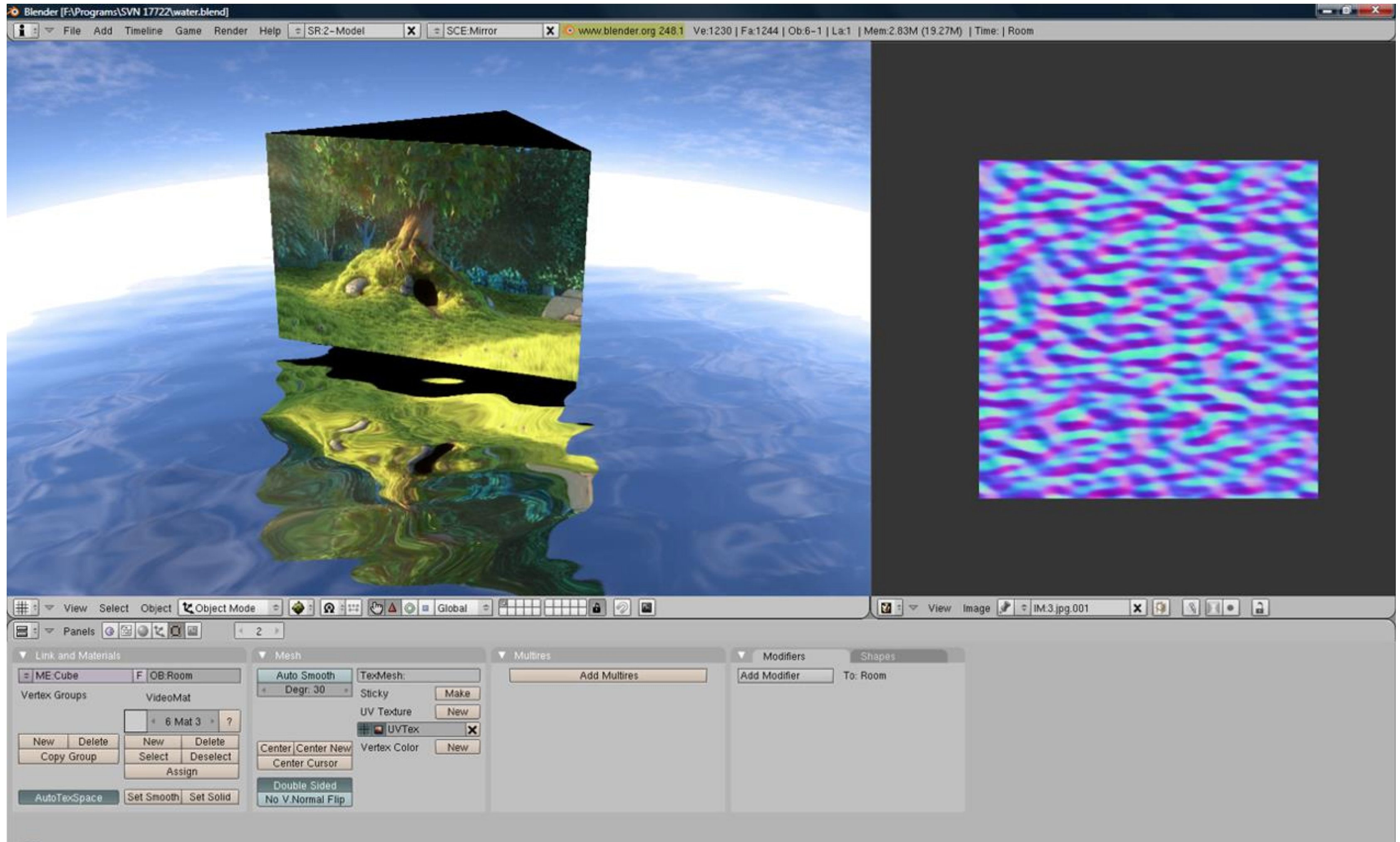
Co je OpenGL

- Aplikační programové rozhraní (API) ke grafickým subsystémům.
- Co nejvíce nezávislý na hardware – ten musí obsahovat pouze framebuffer.
- Nezávislost na operačním systému, grafických ovladačích a správcích oken.

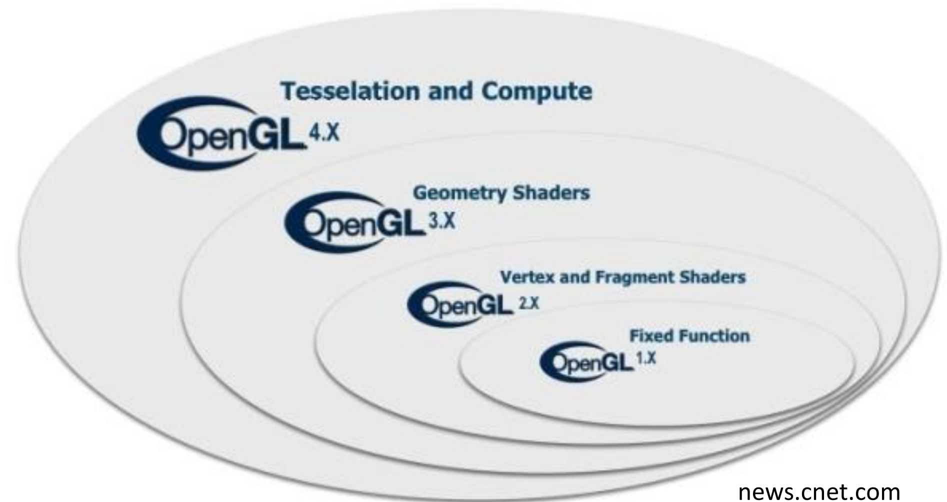
OpenGL před 25 lety



OpenGL 2009 – dnes



Verze OpenGL



news.cnet.com

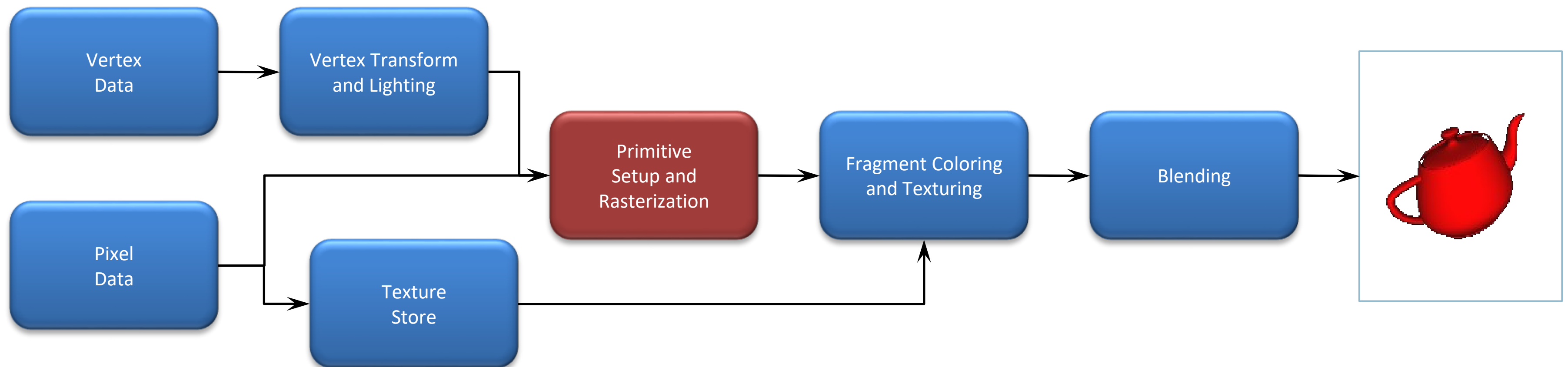
- OpenGL 1.0 – 1992 (Segal, Akeley)
- OpenGL 1.1 – 1997
- OpenGL 1.2, 1.2.1 – 1998
- OpenGL 1.3 – 2001
- OpenGL 1.4 – 2002
- OpenGL 1.5 – 2003
- OpenGL 2.0 – 2004 (zavedení GLSL)
- OpenGL 2.1 – 2006
- OpenGL 3.0 – 2008 (deprecation mechanismus)
- OpenGL 3.1 – 2009
- OpenGL 3.2 – 2009
- OpenGL 3.3 – 2010
- OpenGL 4.0 – 2010 (vydán společně s 3.3)
- OpenGL 4.1 – 2010
- OpenGL 4.2 – 2011
- OpenGL 4.3 – 2012
- OpenGL 4.4 – 2013
- OpenGL 4.5 – 2014

Vulkan

- Nová generace grafického a výpočetního API
- Vysoká efektivita, multiplatformní
- Vhodné pro PC, herní konzole, mobilní zařízení a embedded platformy
- Dostupné od 16. 2. 2016, vydalo konsorcium Khronos
- https://www.khronos.org/vulkan/

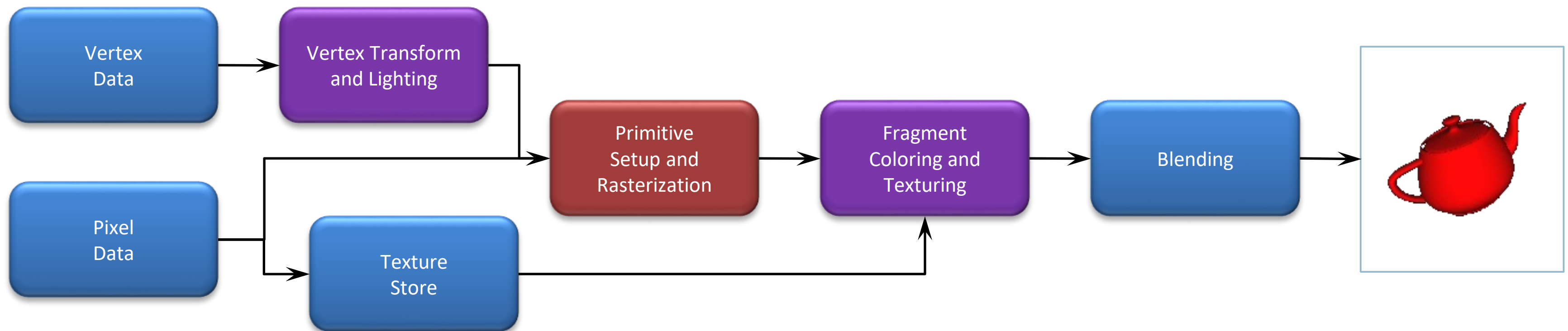
Verze OpenGL

- OpenGL 1.0 – fixní pipeline



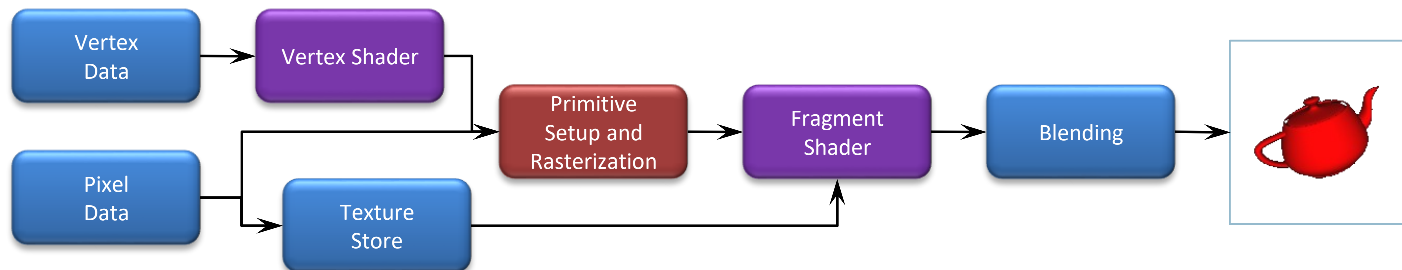
Verze OpenGL

- OpenGL 2.0 – přidání programovatelných shaderů (vertex, fragment), fixní pipeline stále dostupná



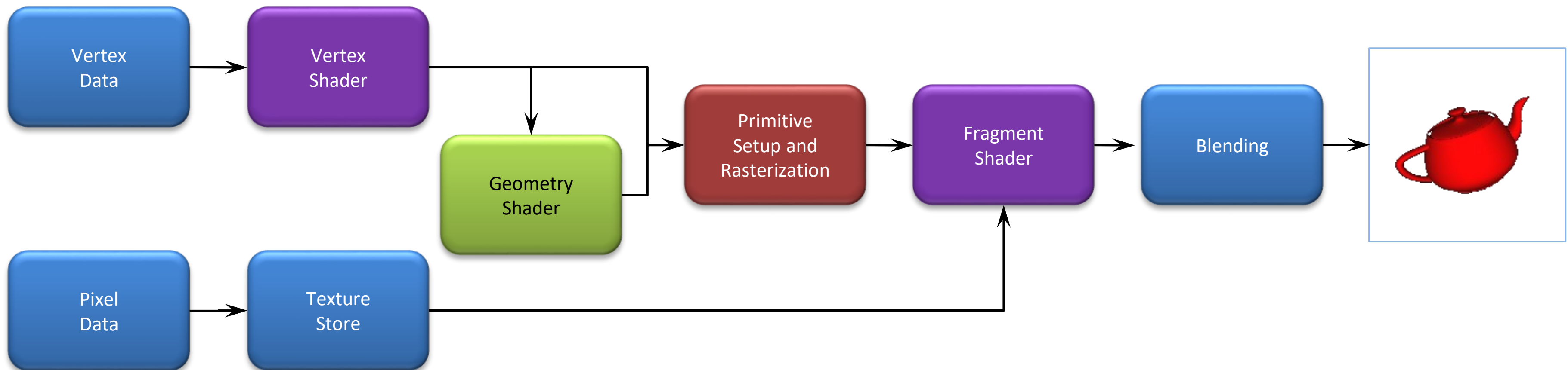
Verze OpenGL

- OpenGL 3.0 – zavedení *deprecation modelu*
  - požadavek na využívání shaderů – zakázání zpracování vertexů a fragmentů bez použití shaderů
- OpenGL 3.1 – odstranění deprecated funkcí, zrušení zpětné kompatibility



Verze OpenGL

- OpenGL 3.2 – přidání dalšího stupně shaderů – *geometry shaders*
  - Modifikace geometrických primitiv v grafické pipeline

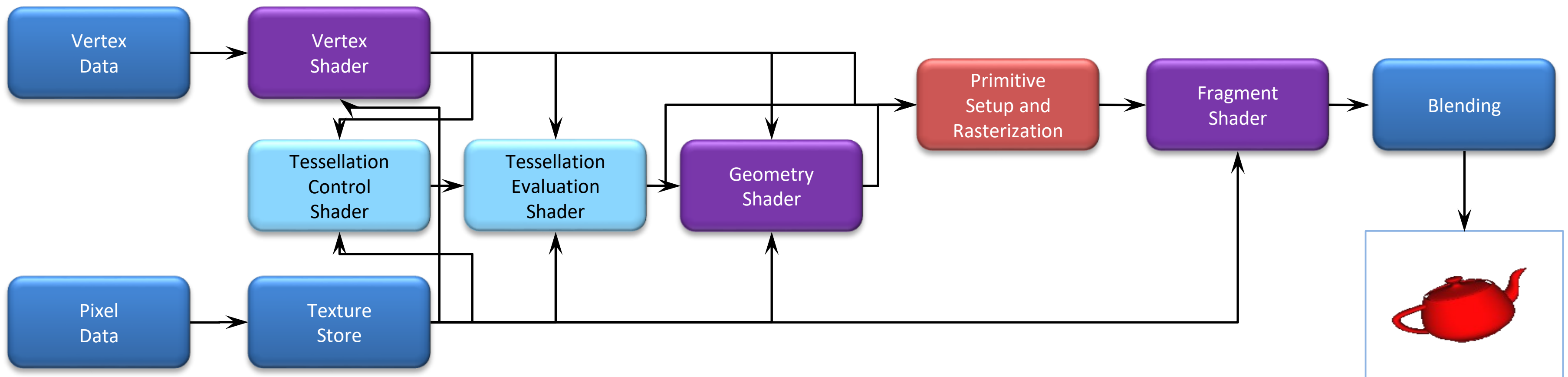


Verze OpenGL

- OpenGL 3.2 zavedla *context profiles*
  - Řeší zpětnou kompatibilitu verzí
  - Dva typy profilů:
    - **Core**
    - **Compatible**

Verze OpenGL

- OpenGL 4.0 zavádí další fáze:
  - *Tessellation-control*
  - *Tessellation-evaluation*



Verze OpenGL

- OpenGL ES 3.2
  - Navržen pro embedded a mobilní zařízení
  - Založen na OpenGL 3.1
  - Využívá shadery
- WebGL
  - JavaScript implementace ES 2.0
  - Podporován většinou současných prohlížečů
  - Dokáže běžet bez rekompilace na různých platformách
  - Jednoduše integrovatelný do webových aplikací

Kdo vytváří specifikaci OpenGL?

- ARB (Architecture Review Board)

- Knihovny vytvářejí libovolní poskytovatelé hardware – musí podstoupit *Conformance test,* jinak se produkt nesmí označovat jako součást OpenGL

První pohled na OpenGL

- OpenGL je procedurální interface = obsahuje výkonné instrukce na úrovni programovacího jazyka

- OpenGL není „pixelově exaktní“
  - Stejné sekvence příkazů mohou mít na různých platformách za výsledek nepatrně odlišné obrázky

Co OpenGL umí?

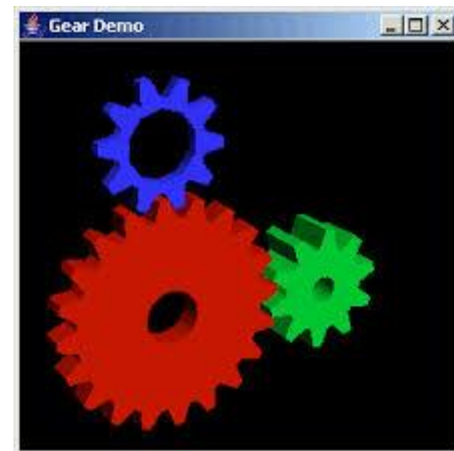
- vykreslování trojúhelníků, bodů
- texturování (texturing)
- osvětlení (lighting) – pomocí vlastní implementace
- stínování (shading)
- mlha (fog) – pomocí vlastní implementace
- výpočet viditelnosti
- alfa míchání (alpha blending)
- transformace (transformations) – pomocí vlastní implementace
- buffer šablony (stencil buffer)
- stíny (shadows) – pomocí vlastní implementace
- odrazy – pomocí vlastní implementace
- voxely – pomocí vlastní implementace

Co OpenGL neumí?

- práci s okny
- NURBS (lze přes Compute shadery nebo aproximací přes Tesselation shadery)
- reprezentaci scény (částečně lze přes Compute shadery)

Knihovny OpenGL

- Pro práci s okny, načítání obrázků, geometrií, animací, …
- FreeGLUT, GLEW, JOGL, …



- Detaily na cvičení

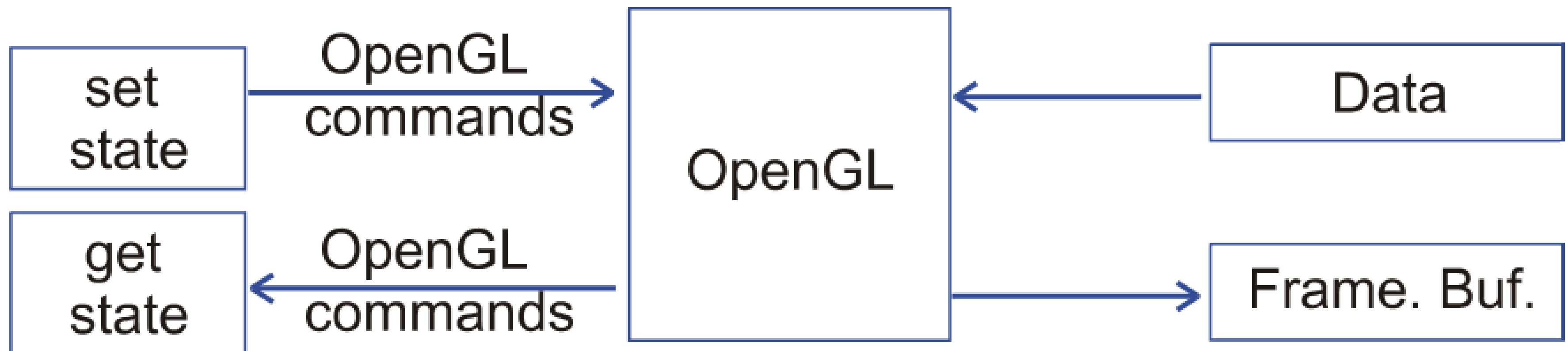
OpenGL a práce s okny

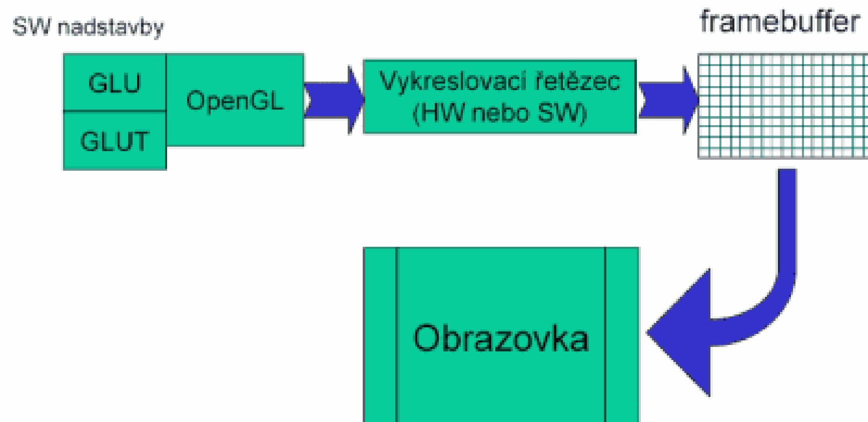
- V různých programovacích jazycích různé nástroje
- C/C++:
  - GLUT
  - SDL

- Java
  - JOGL + Swing

OpenGL jako stavový automat

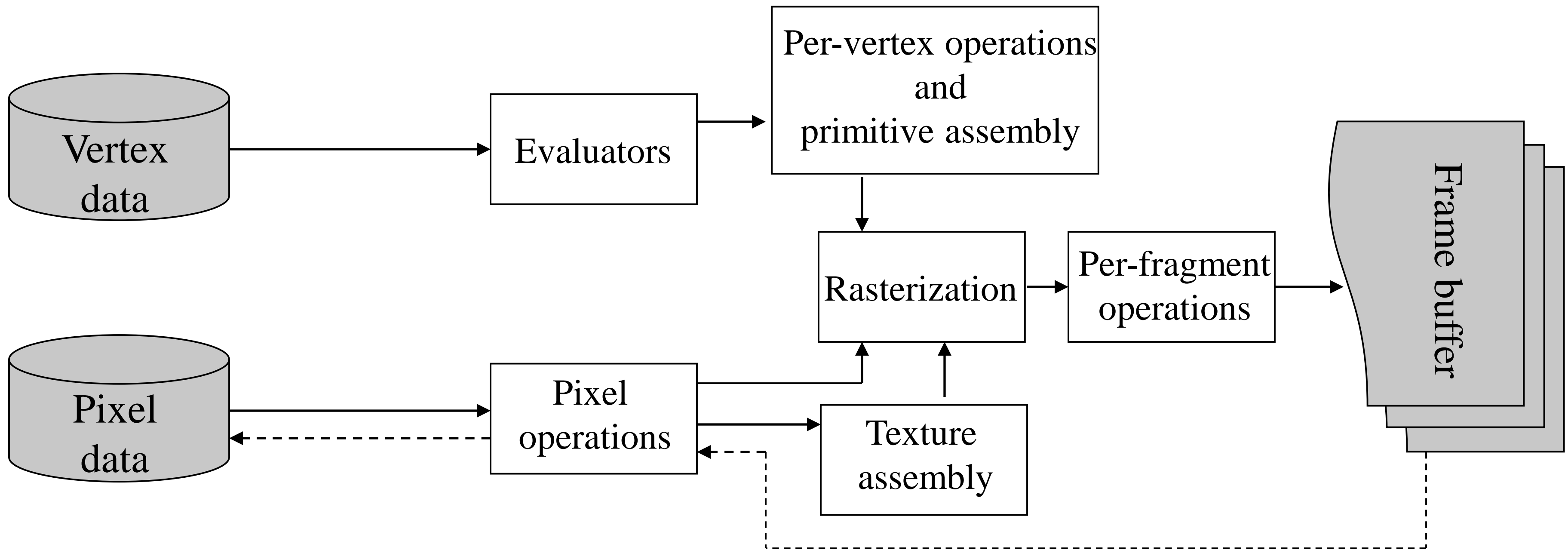
- Lze nastavit stav, zeptat se na aktuální stav, …



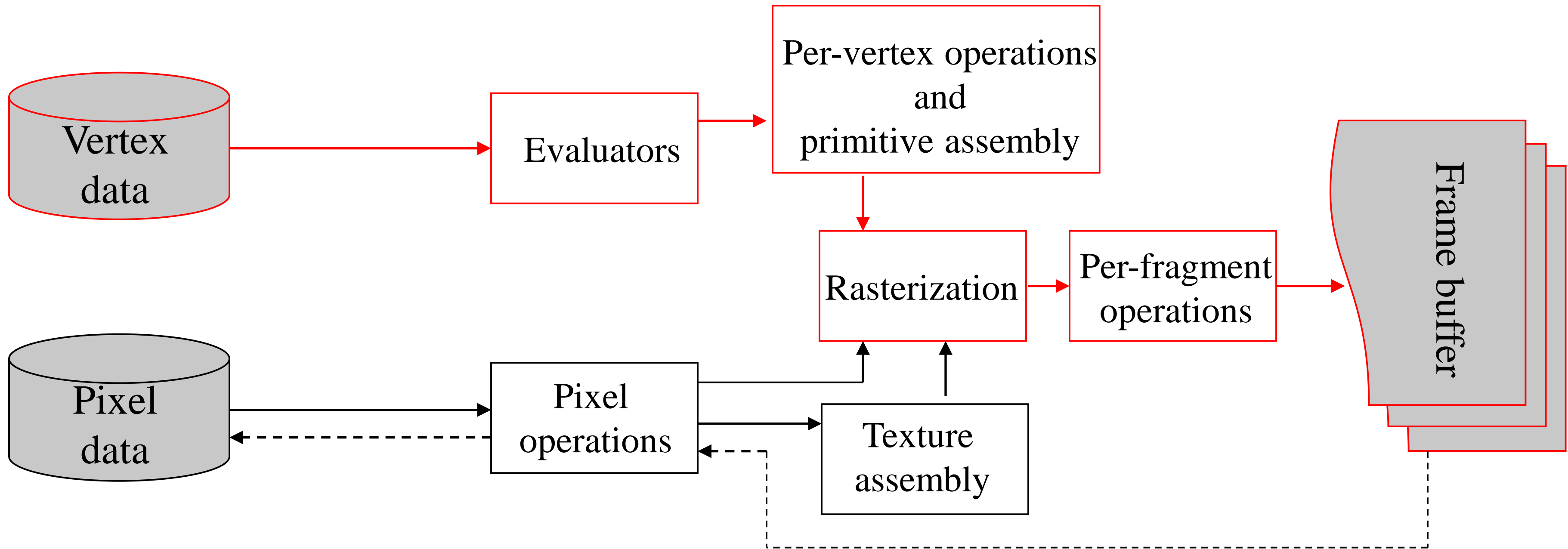


http://www.root.cz/

Blokový diagram OpenGL

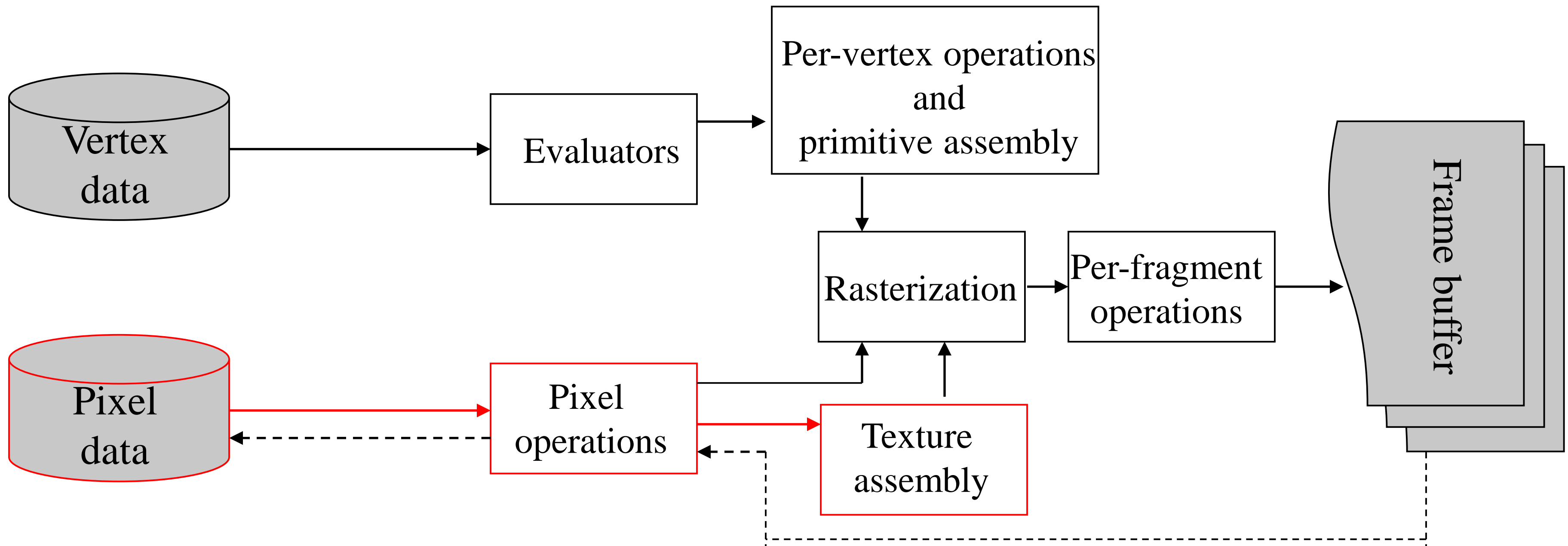


Vykreslování trojúhelníka



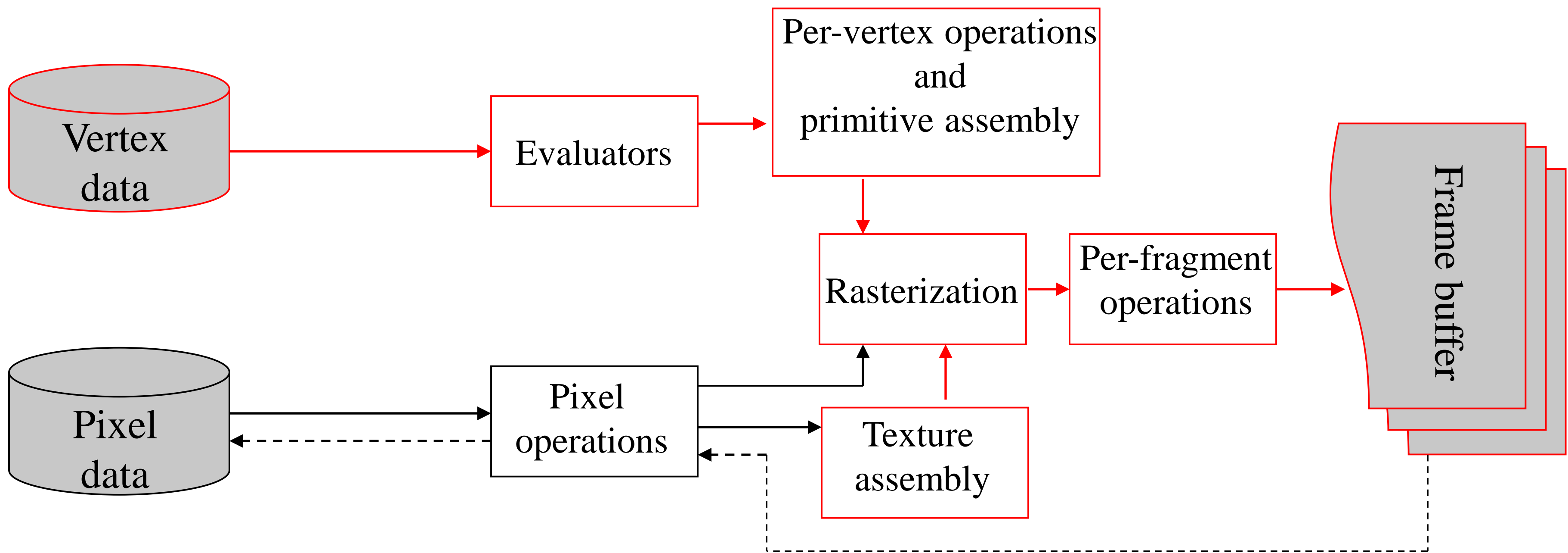
- Rasterizace je nejdražší operací

Vykreslování trojúhelníka s texturou – 1. krok



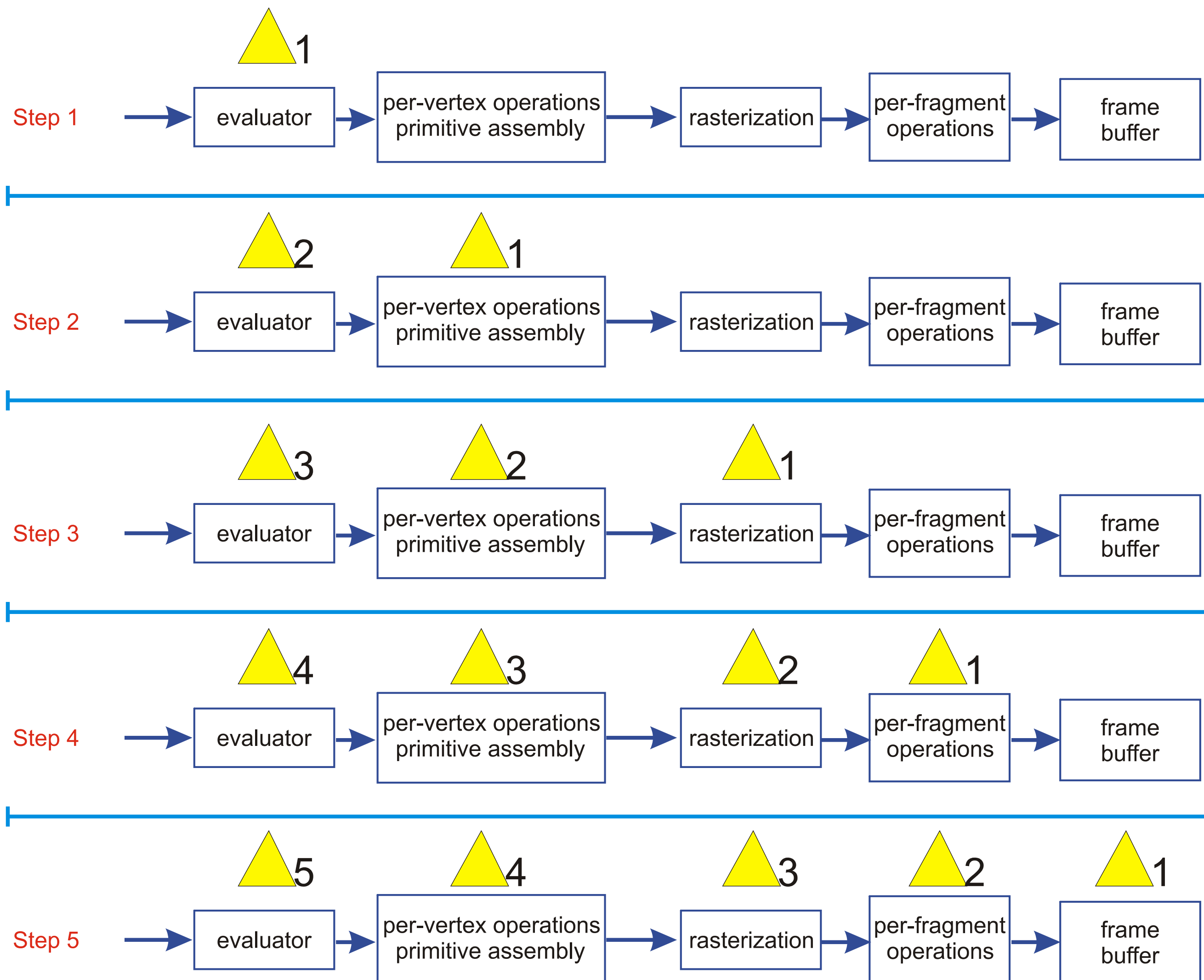
- Uložení textury v paměti – příprava na další použití

Vykreslování trojúhelníka s texturou – 2. krok



- Použití uložené textury

Pipelining – vykreslování množiny trojúhelníků



GLSL (Graphics Library Shading Language)

- Nahrazení některých částí fixní pipeline programovatelnými částmi
- Programovatelné části jsou psány pomocí shaderů
- Program je sadou shaderů, které jsou zkompilovány a „slinkovány“
- GLSL podobný C/C++
- Alternativy: Nvidia Cg, Microsoft HLSL

GLSL

- V současnosti existuje již několik shaderů
  - **Vertex shader**
  - **Fragment shader**
  - Tesselation shader
  - Geometry shader
  - Compute shader

- Budeme se zabývat vertex a fragment shadery

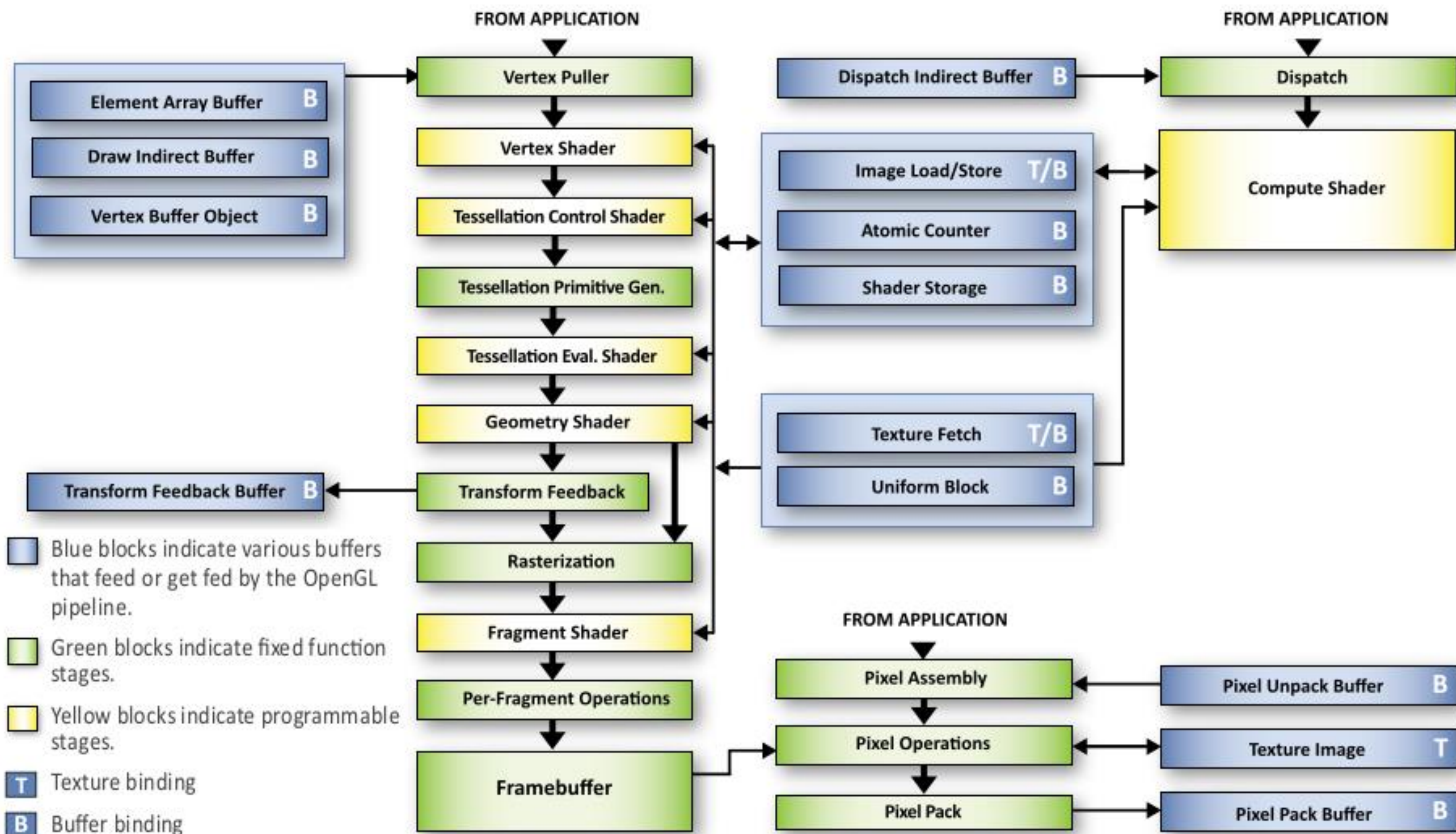
Vertex data
Evaluators
Per-vertex operations and primitive assembly
Rasterization
Per-fragment operations
Frame buffer
Pixel data
Pixel operations
Texture assembly

Fixed functionality vertex processing
Fixed functionality fragment processing
Vertex data
Per-vertex operations
Primitive assembly
Rasterization
Fragment processing
Frame buffer
Pixel data
Pixel operations
Texture memory

Shader program

- Malý program řídící určitou část grafické pipeline
- Kód, který je prováděn pro každý vrchol (vertex shader) nebo fragment (fragment shader) přímo na GPU
- Vertex a fragment shader navzájem komunikují
- Vertex shader:
  - Transformace vrcholů, barva vrcholů, osvětlení na úrovni vrcholů
- Fragment shader:
  - Texturování, barva a osvětlení na úrovni pixelů

Moderní grafická pipeline

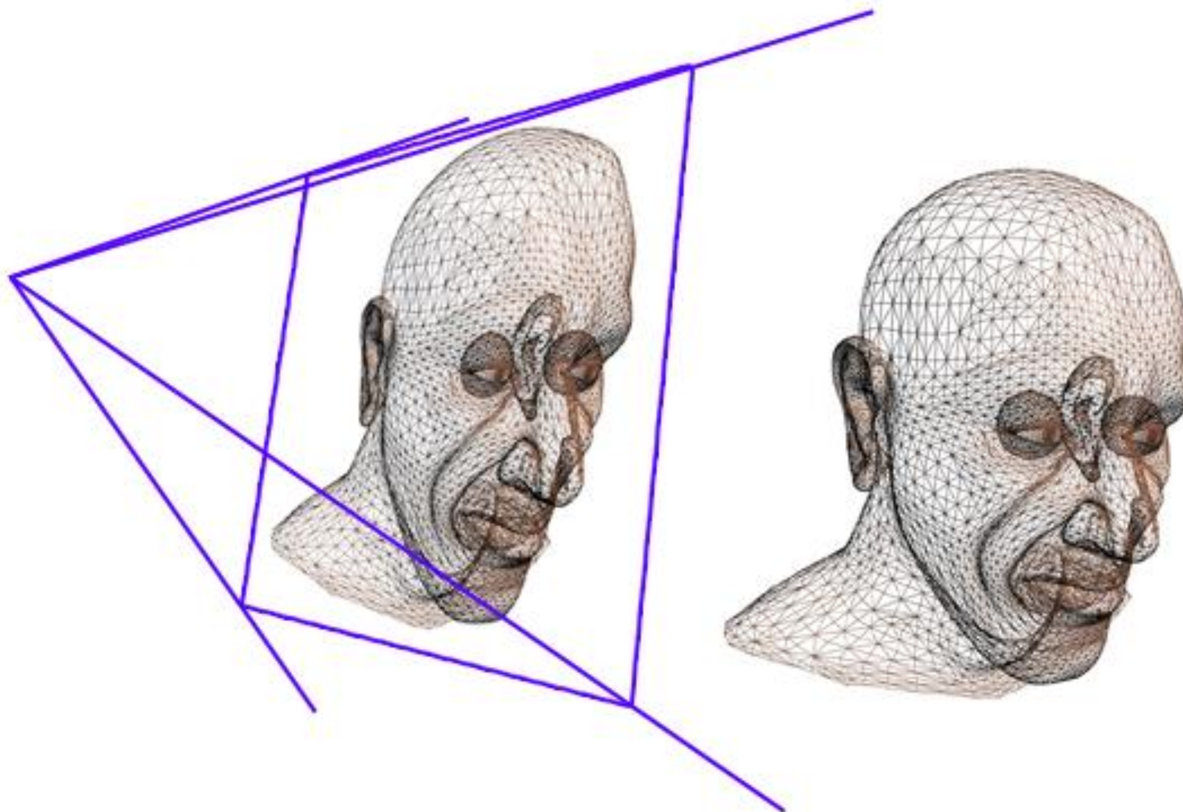


GLSL program

- Specifikuje, jak má OpenGL vykreslovat geometrii
- GLSL program je kolekcí shaderů (musí obsahovat alespoň jeden vertex shader a jeden fragment shader)
- Na GPU nemůže běžet více programů současně

Vertex shader

- Transformuje vrcholy z prostoru objektu do prostoru obrazovky

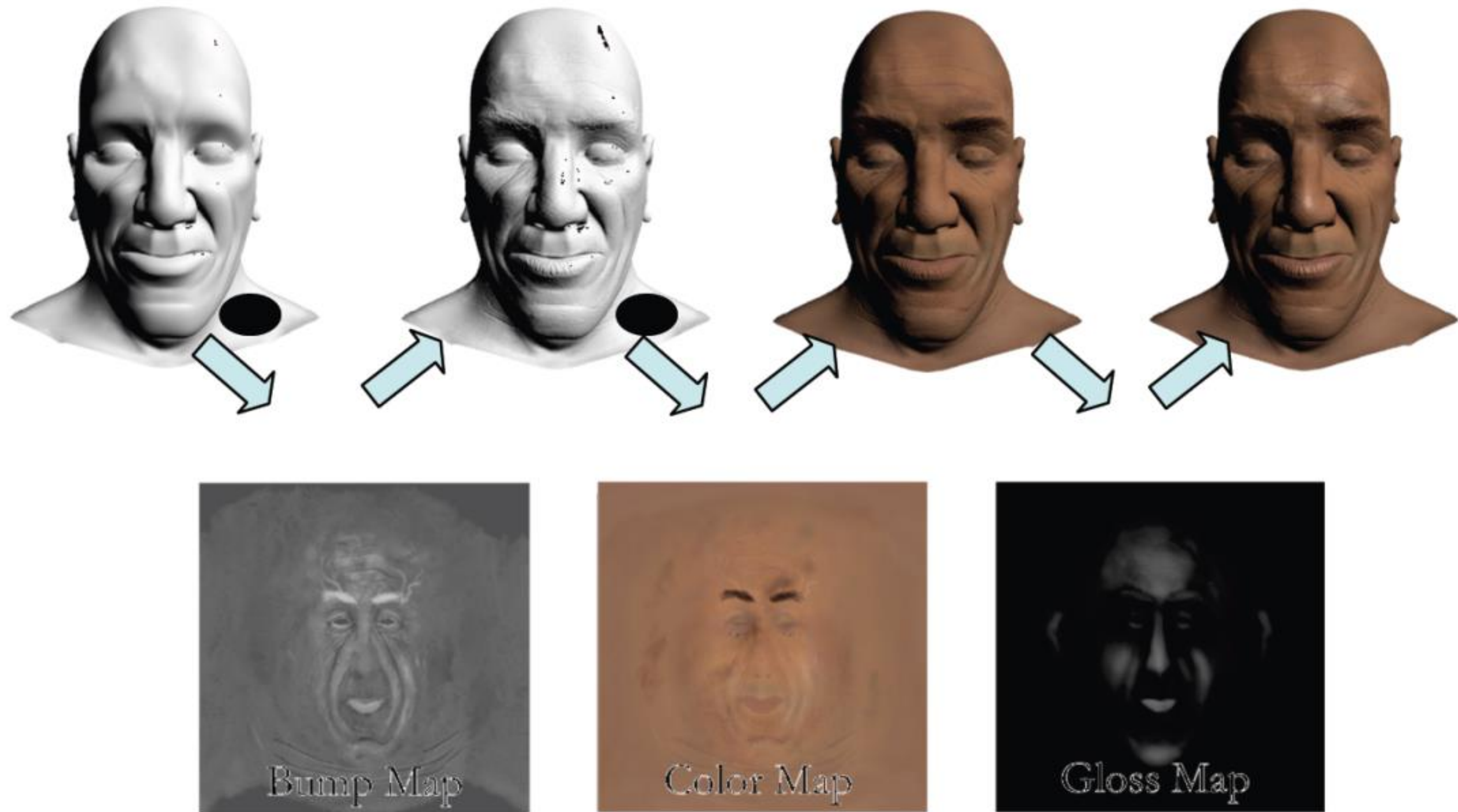


Vertex shader

- Transformace vrcholů, normál, texturových souřadnic
- Generování texturových souřadnic
- Počítání osvětlení vrcholů
- Nastavení hodnot pro interpolaci, která je poté provedena ve fragment shaderu
- …
- Vertex shader neví nic o organizaci scény (viewport, ořezání, …)

Fragment shader

- Počítá barvu fragmentu (pixelu)
- Na vstupu bere data z vertex shaderu



Fragment shader

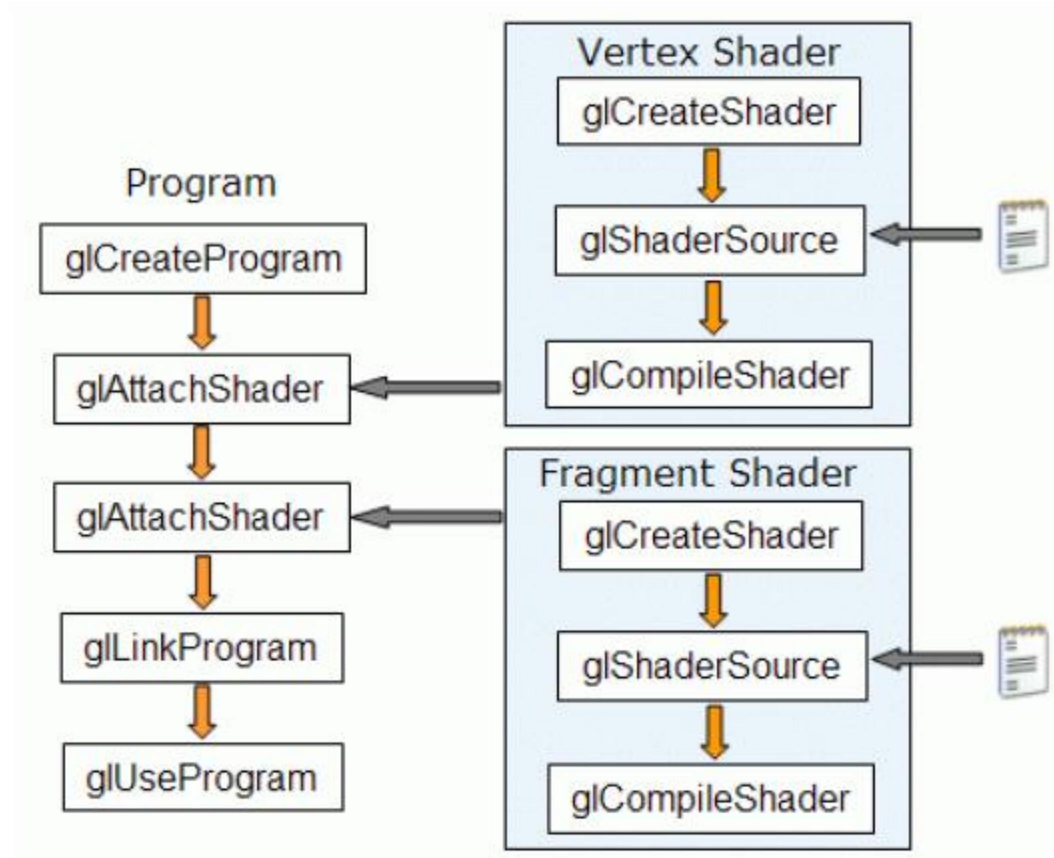
- Někdy označován i jako *pixel shader*
- Nastavení barvy fragmentu
- Získání hodnot barvy z textur
- Výpočet mlhy
- Využití interpolovaných dat z vertex shaderu

- Fragment shader neumí změnit souřadnice fragmentů, …

Jak vytvořit GLSL program

- 7 základních kroků:
  1. Vytvoření shader objektů.
  2. Načtení zdrojového kódu ze souboru a naplnění vytvořených shader objektů tímto kódem.
  3. Kompilace shaderu.
  4. Vytvoření objektu programu.
  5. Připojení (attach) shaderů k programu.
  6. Slinkování programu.
  7. Oznámení OpenGL, aby používalo tento program.

Nastavení shaderů

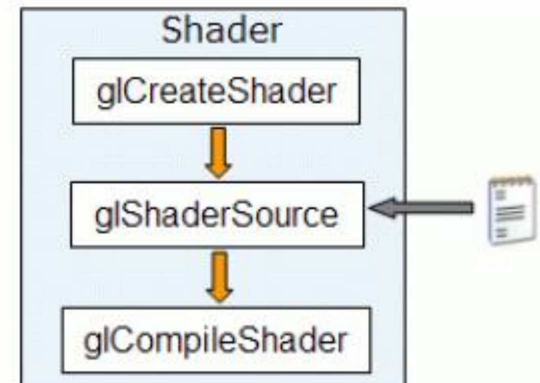


Vytvoření shaderu

```
Gluint glCreateShader(Glenum shaderType);
```

– *shaderType* = GL_{VERTEX|FRAGMENT|GEOMETRY|
TESS_CONTROL|TESS_EVALUATION|
COMPUTE}_SHADER

- Vytvoří shader objekt daného typu, který funguje jako kontejner
- Vrátí ID (jméno) tohoto kontejneru



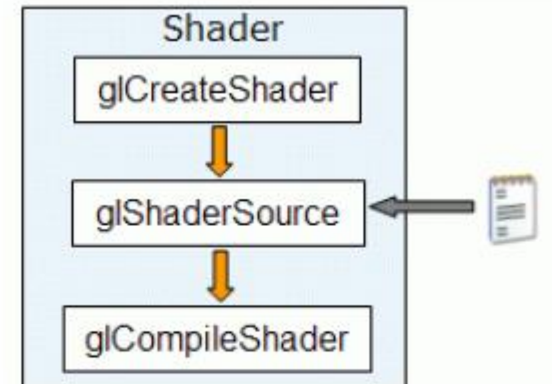
Vytvoření shaderu

- Lze vytvořit libovolné množství shaderů, které lze připojit k programu, ale pro jeden program může být pouze jedna *main* funkce pro sadu vertex shaderů a jedna *main* funkce pro sadu fragment shaderů

Vytvoření shaderu

```
void glShaderSource(Gluint shader,
Glsizei count, const Glchar **string,
const Glint *length);
```

- *shader* – handler shaderu
- *count* – počet řetězců v poli
- *string* – pole řetězů
- *length* – pole s délkami řetězců



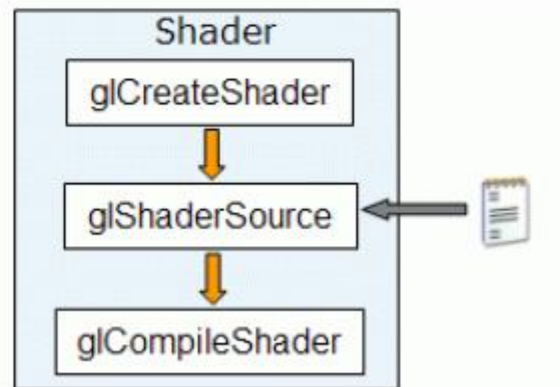
- Nahrazuje zdrojový kód shaderu
- Místo pole stringů může
  být použit jenom jeden string

Vytvoření shaderu

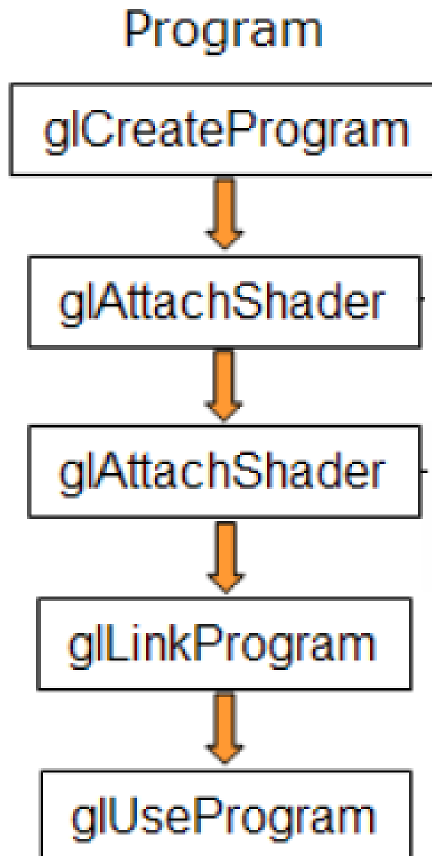
```
void glCompileShader(GLuint shader);
```

– *shader* – handler shaderu

- Zkompiluje shader a zkontroluje jeho validitu



Vytvoření programu



Vytvoření programu

```
GLuint glCreateProgram(void);
```

- Vytvoří program, který vystupuje jako kontejner
- Vrátí ID tohoto kontejneru
- V jednom frame může být vytvořeno a použito libovolné množství programů
- Mezi programy lze přepínat za běhu programu

Vytvoření programu

```
void glAttachShader(Gluint program, Gluint shader);
```

- *program* – handler programu
- *shader* – handler shaderu, který má být přiřazen

- Přiřadí shader k programu
- Shadery nemusí být zkompilované, dokonce nemusí mít zdrojový kód – musíme mít k dispozici alespoň kontejner shaderu
- Lze přiřadit libovolné množství shaderů, ale pro každý typ shaderu jenom jeden obsahuje *main* funkci
- Jeden shader může být přiřazen více programům

Vytvoření programu

```
void glLinkProgram(Gluint program);
```

– *program* – handler programu

- Slinkuje program
- Shadery už musí být v této fázi zkompilovány
- Následně mohou být shadery znovu modifikovány a překompilovány
- Jsou přiřazeny uniformní proměnné a nastaveny na nulu

Vytvoření programu

```
void glUseProgram(Gluint program);
```

- *program* – handler programu; pro použití fixní pipeline je tato hodnota nastavena na nulu

- Nastaví program, který má být využit při renderování

Vyčištění

```
void glDetachShader(Gluint program, Gluint shader);
```

- *program* – program, od kterého se má odpojit
- *shader* – shader pro odpojení

- Odpojí shader od programu

Vyčištění

```
void glDeleteShader(Gluint id);
```

```
void glDeleteProgram(Gluint id);
```

- *id* – handler shaderu/programu, který má být smazán

- Když je shader stále přiřazen k nějakému programu, je pouze „označen pro smazání“ a úplně smazán je až tehdy, když je odpojen od všech programů a není už tedy používán

GLSL nastavení – příklad

```
void setShaders()
{
  char *vs, *fs;

  // Setup
  v = glCreateShader(GL_VERTEX_SHADER);
  f = glCreateShader(GL_FRAGMENT_SHADER);

  vs = textFileRead("simple.vert");
  fs = textFileRead("simple.frag");

  const char * vv = vs;
  const char * ff = fs;

  glShaderSource(v, 1, &vv, NULL);
  glShaderSource(f, 1, &ff, NULL);

  free(vs);
  free(fs);

  glCompileShader(v);
  glCompileShader(f);
```

GLSL nastavení – pokračování příkladu

```
p = glCreateProgram();

glAttachShader(p, v);
glAttachShader(p, f);

glLinkProgram(p);
glUseProgram(p);

...

// Clean up
glDetachShader(p, v);
glDetachShader(p, f);

glDeleteShader(v);
glDeleteShader(f);

glUseProgram(0);
glDeleteProgram(p);
}
```

Nástroje pro psaní shaderů

- Shadery jsou pouze řetězce – je možné využít libovolný editor:
- RenderMonkey (http://developer.amd.com/tools-and-sdks/archive/legacy-cpu-gpu-tools/rendermonkey-toolsuite/)
- FX composer (https://developer.nvidia.com/fx-composer)
- OpenGL Shader Designer (http://www.opengl.org/sdk/tools/ShaderDesigner/)

Nástroje pro psaní shaderů

- NVIDIA Nsight
  - Pouze pro registrované vývojáře
- AMD CodeXL
- gDEBugger (http://www.gremedy.com/)
- Visual Studio
  - Zvýraznění syntaxe, IntelliSense
  - Používá se v PV227

GLSL datové typy

- float, int, bool
- vec2, vec3, vec4: vektory
- mat2, mat3, mat4: matice
- sampler1D, sampler2D, sample3D, samplerCube, …: textury

vec2

- Reprezentuje vektor ve 2D obsahující float hodnoty

```
vec2 a;
a.x = 0.0;
a.y = 1.0; // a = (0,1)

vec2 b;
b.s = 10.0;
b.t = 12.5; // b = (10,12.5)

vec2 c;
c[0] = 9.0;
c[1] = 8.0; // c= (9,8)
```

vec3

```
vec3 a;
a.x = 10.0; a.y = 20.0; a.z = 30.0;  // a = (10,  20,  30)
a.r = 0.1;  a.g = 0.2;  a.b = 0.3;   // a = (0.1, 0.2, 0.3)
a.s = 1.0,  a.t = 2.0;  a.p = 3.0;   // a = (1, 2, 3)

vec3 b = vec3(4.0, 5.0, 6.0);

vec3 c = a + b; // c = (5, 7, 9)
vec3 d = a - b; // d = (-3, -3, -3)
vec3 e = a * b; // e = (4, 10, 18)
vec3 f = a * 3; // e = (3, 6, 9)
float g = dot(a,b); // g = 32
vec3 h = cross(a,b); // h = (-5,6,-3)
float i = length(a); // i = 3.742
```

vec4

```
vec4 a;
a.x = 10.0; a.y = 20.0; a.z = 30.0; a.w = 40.0;
// a = (10, 20, 30, 40)
a.r = 0.1;  a.g = 0.2;  a.b = 0.3;  a.a = 0.4;
// a = (0.1, 0.2, 0.3, 0.4)
a.s = 1.0;  a.t = 2.0;  a.p = 3.0;  a.q = 4.0;
// a = (1, 2, 3, 4)

vec4 b = vec4(5, 6, 7, 8);

vec4 c = a + b; // c = (6, 8, 10, 12)
vec4 d = a - b; // d = (-4, -4, -4, -4)
vec4 e = a * b; // e = (5, 12, 21, 32)
vec4 f = a * 3; // f = (3, 6, 9, 12)
float g = length(a); // g = 5.477
```

mat2

- Matice 2x2 obsahující float čísla

```
mat2 A = mat2(1.0, 2.0, 3.0, 4.0); // in column-major order

vec2 x = vec2(1.0, 0.0);
vec2 y = vec2(0.0, 1.0);

vec2 a = A * x; // a = (1,2)
vec2 b = A * y; // b = (3,4)
```

mat3

```
mat3 A = mat3(1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0);
// in column-major order

vec3 x = vec3(1.0, 0.0, 0.0);
vec3 y = vec3(0.0, 1.0, 0.0);
vec3 z = vec3(0.0, 0.0, 1.0);

vec3 a = A * x; // a = (1,2,3)
vec3 b = A * y; // b = (4,5,6)
vec3 c = A * z; // c = (6,7,8)
```

mat4

- Matice 4x4, do které lze ukládat afinní transformace

```
mat4 A = mat4(1.0,  2.0,  3.0,  4.0,
              5.0,  6.0,  7.0,  8.0,
              9.0,  10.0, 11.0, 12.0
              13.0, 14.0, 15.0, 16.0); // in column-major order

vec4 x = vec4(1.0, 0.0, 0.0, 0.0);
vec4 y = vec4(0.0, 1.0, 0.0, 0.0);
vec4 z = vec4(0.0, 0.0, 1.0, 0.0);
vec4 w = vec4(0.0, 0.0, 0.0, 1.0);

vec4 a = A * x; // a = (1,2,3,4)
vec4 b = A * y; // b = (5,6,7,8)
vec4 c = A * z; // c = (9,10,11,12)
vec4 d = A * w; // d = (13,14,15,16)
```

Pole (array)

- Fixní velikost, notace C

```
float A[4];
A[0] = 5; A[3] = 10;

vec4 B[10];
B[3] = vec4(1,2,3,4);
B[8].y = 10.0;
```

Typy vstupů a výstupů

- Shadery mohou mít tři různé typy vstupů a výstupů:
  - **uniforms**
  - **attributes**
  - **varyings**
- **Uniforms**
  - hodnoty, které se během renderování nemění (například pozice světla či barva světla)
  - jsou dostupné ve vertex i fragment shaderech
  - jsou read-only

Typy vstupů a výstupů

- **Attributes**
  - Dostupné pouze ve vertex shaderu
  - Vstupní hodnoty, které mění každý vrchol (například pozice vrcholu nebo normály)
  - Jsou rovněž read-only
- **Varyings**
  - Používány pro transfer dat z vertex shaderu do fragment shaderu
  - Jsou read-only ve fragment shaderu, ale read a write ve vertex shaderu
  - Pokud chceme varyings používat, musíme deklarovat stejnou varying ve vertex i fragment shaderu
- Tato označení se používají spíše z historických důvodů. Dnes nahrazeno pojmy „vstupní“ (in) a „výstupní“ (out) proměnné shaderu.

Vestavěné typy

- Můžeme využít řadu vestavěných typů, které jsou používány pro výstup shaderu. My budeme používat:

gl_Position	4D vektor reprezentující finální pozici zpracovaného vrcholu. Dostupné pouze pro vertex shader.

Generické typy

- Můžeme si rovněž specifikovat své vlastní vstupní a vstupní proměnné či uniforms
- Attribute = in, varying = out
- Příklady:

```
uniform sampler2D my_color_texture;
uniform mat4 my_texture_matrix;

in vec3 tangent;
in vec3 binormal;

out vec3 vertex_to_light_vector;
out vec3 vertex_to_eye_vector;
```

Inicializace a konstruktory

- Proměnná shaderu může být inicializována v okamžiku její deklarace. Jakmile ji však změním pomocí C++/Java kódu, použije se tato nová hodnota.
- To neplatí pro proměnné typu **in**, **out** a **uniform**
- Existují konstruktory pro téměř všechny vestavěné typy a pro struktury

```
vec4 v = vec4(1.0, 0.5, 0.0, 1.0);
---

vec4 v;
v = vec4(1.0, 0.5, 0.0, 1.0);
mat2 m = (1.0, 2.0, 3.0, 4.0);
---
```

Struktury

- Uživatelem definované typy je možné vytvářet agregací předdefinovaných typů do struktur uvozených klíčovým slovem **struct**. Příklad:

```
struct light
{
    float intensity;
    vec3 position;
} lightVar;
```

kde *light* je jméno nového typu a *lightVar* je proměnnou typu *light*. Pro deklaraci proměnné tohoto nového typu stačí použít její jméno:

```
light lightVar2;
```

Inicializace a konstruktory

```
vec3 color = vec3(0.2, 0.5, 0.6);
vec4 v = vec4(20.0, 12.4, 2.1, 1.0);
struct light
{
  vec4 position;
  struct lightColor
  {
    vec3 color;
    float intensity;
  }
} light1 = light(v, lightColor(color, 0.9));
```

Typová konverze

- Explicitní typová konverze se rovněž provádí pomocí konstruktorů – GLSL totiž neposkytuje syntaxi pro převod typů

```
float f = 2.3;
bool b = bool(f);
float f = float(3); // convert integer 3 to 3.0
float g = float(b); // convert Boolean b to
                       floating point
vec4 v = vec4(2)    // all components of v are
                       set to 2.0
```

Přístup ke komponentám

- Ke komponentám jednotlivých proměnných je možné přistupovat pomocí tečkové notace:

```
vec2 pos;
pos.x  // správně
pos.z  // špatně
```

- Syntaxe povoluje rovněž výběr několika komponent najednou – uvedením jejich jména (označení) za tečkou:

```
vec4 v4;
v4.rgba; // stejný výsledek jako při použití pouze v4
v4.rgb;  // je vec3
v4.b;    // je float
v4.xy;   // je vec2
v4.xgba; // špatně - jména komponent nepocházejí ze
            stejné sady
```

Swizzling

- Pořadí jednotlivých komponent může být změněno nebo replikováno pomocí tzv. swizzlingu:

```
vec4 pos = vec4(1.0, 2.0, 3.0, 4.0);
vec4 swiz = pos.wzyx; // swiz = (4.0, 3.0,
                              2.0, 1.0)
vec4 dup = pos.xxyy;  // dup = (1.0, 1.0,
                              2.0, 2.0)
```

Operace po komponentách

- Kromě několika málo výjimek se operátory chovají, jako by byly aplikovány na každou komponentu vektoru zvlášť.

```
vec3 u, v, w;
float f;

v = u + f; ~ v.x = u.x + f; v.y = u.y + f;
             v.z = u.z + f;

w = u + v; ~ w.x = u.x + v.x; w.y = u.y +
             v.y; w.z = u.z + v.z;
```

Operace po komponentách

- Logické operátory !, &&, ||, ^^ fungují pouze na výrazech, které jsou označeny jako scalar booleans a jejich výsledkem je opět scalar boolean.
- Relační operátory <, >, <=, >= fungují pouze pro floating-point a integer skaláry a výsledkem je scalar boolean.
- Operátory rovnosti ==, != fungují pro všechny typy s výjimkou polí. Výsledkem je scalar boolean.

Výrazy a funkce

- Můžeme používat řadu výrazů, které jsou velmi podobné jazyku C:

```
if (bool expression)
        ...
else
        ...

for (initialization; bool expression; loop expression) ...

while (bool expression) ...

do
         ...

while (bool expression)
```

Výrazy a funkce

- Skoky jsou rovněž definovány:
  - **continue** – dostupné ve smyčkách, způsobí skok na další iteraci smyčky
  - **break** – dostupné ve smyčkách, ukončí zpracování smyčky
  - **discard** – dostupné pouze ve fragment shaderech. Způsobí ukončení shaderu pro aktuální fragment bez jeho zápisu do framebufferu.

Výrazy a funkce

- Shader je strukturován do funkcí
- Každý shader musí obsahovat minimálně main funkci:

```
void main()
```

- Můžeme definovat svoje vlastní funkce
- Ty mohou být typu void nebo mohou mít svoji návratovou hodnotu
- Návratový typ může být libovolný z definovaných – ale nemůže to být pole

Výrazy a funkce

- Parametry funkce mají následující dostupné kvalifikátory:
  - **in** – pro vstupní parametry.
  - **out** – pro výstupy funkce. Jiným způsobem předání návratové hodnoty je využití právě výše zmíněného return statement.
  - **inout** – pro parametry, které jsou zároveň vstupem i výstupem dané funkce.
- Pokud není specifikován žádný kvalifikátor, je defaultně brán kvalifikátor *in*.

Výrazy a funkce

Další poznámky:

- Funkce může být přetížena – obsahuje jiný seznam parametrů.
- Rekurzivní chování není ve specifikaci definováno.

Příklad funkce

```
vec4 toonify(in float intensity) {
    vec4 color;
    if (intensity > 0.98)
        color = vec4(0.8,0.8,0.8,1.0);
    else if (intensity > 0.5)
        color = vec4(0.4,0.4,0.8,1.0);
    else if (intensity > 0.25)
        color = vec4(0.2,0.2,0.4,1.0);
    else
        color = vec4(0.1,0.1,0.1,1.0);
    return(color);
}
```

Detaily jazyka

- GLSL obsahuje následující vlastnosti/omezení:
  - **GLSL je 100% type-safe. Nelze tedy přiřadit integer hodnotu do float proměnné bez přetypování.**

```
float my_float = 1;           // Nefunguje! 1 je
                                 integer
float my_new_float = 1.0;     // Funguje!
```

Detaily jazyka

- Přetypování musí být provedeno v konstruktoru

```
vec2 my_vec;
ivec2 my_int_vec;
my_vec2 = (vec2)my_int_vec; // Nefunguje,
          protože není použit konstruktor!
my_vec2 = vec2(my_int_vec); // Funguje!
```

Detaily jazyka

- Vektory a matice mohou být naplněny daty jen pomocí konstruktoru:

```
vec3 my_vec = vec3(1.0, 1.0, 1.0);
mat3 my_mat = mat3(1.0, 1.0, 1.0,
0.0, 0.0, 0.0, 1.0, 1.0, 1.0);
```

Detaily jazyka

- Násobení vektorů je prováděno po složkách:

```
vec3 my_vec1 = vec3(5.0, 1.0, 0.0);
vec3 my_vec2 = vec3(1.0, 3.0, 4.0);
vec3 product = my_vec1 * my_vec2;   // Výsledný
                               vektor: (5.0, 3.0, 0.0)
```

- Násobení vektoru a matice probíhá následovně:
- *Matice * vektor* – vektor je brán jako sloupcový
- *Vektor * matice* – vektor je brán jako řádkový

Detaily jazyka

- Můžeme (a měli bychom) využívat i řadu vestavěných funkcí, například:

dot	jednoduchý skalární součin
cross	jednoduchý vektorový součin
texture	používána pro sampling textury
normalize	normalizace vektoru
clamp	clamping vektoru na minimum a maximum