Úvod do OpenGL

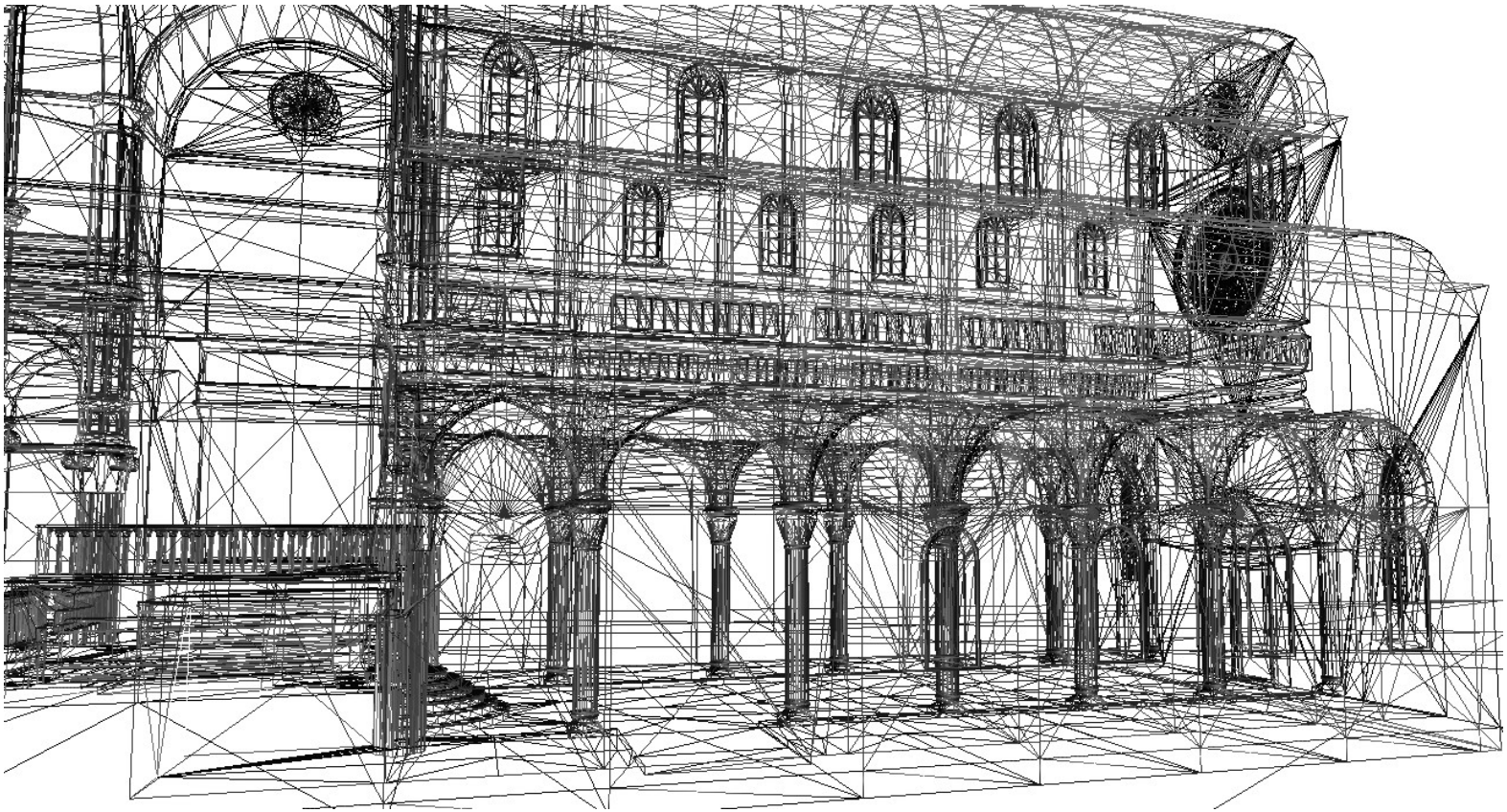
Tomáš Milet

Brno University of Technology, Faculty of Information Technology
Božetěchova 1/2. 612 66 Brno - Královo Pole
login@fit.vutbr.cz

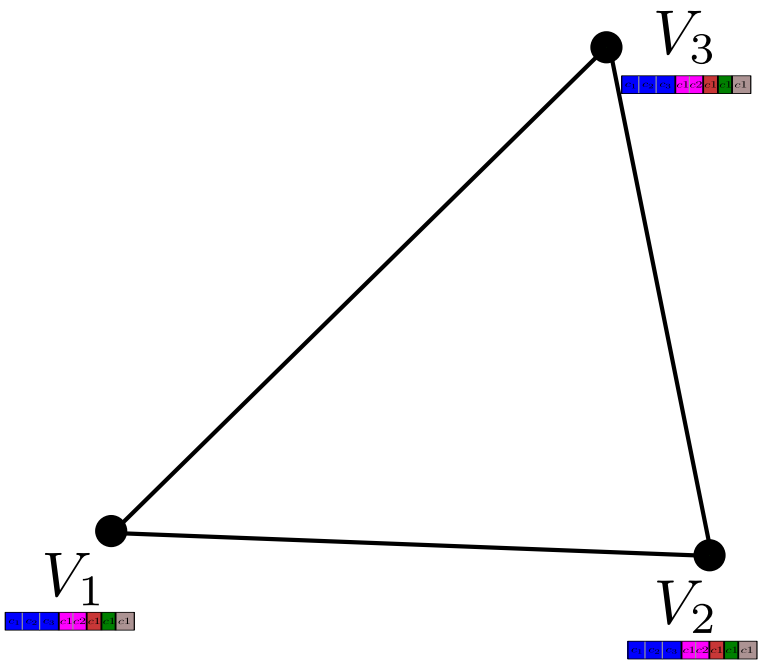
14. dubna 2016

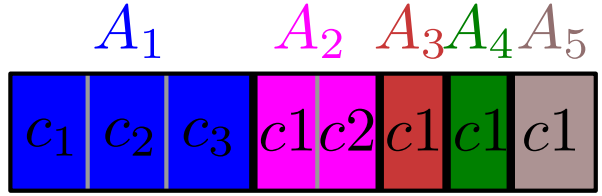
Základní postup kreslení/zobrazovací pipeline

- Povrchová reprezentace - vektorová data.

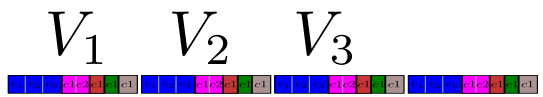


- OpenGL pracuje s vrcholy - Vertexy
- Jeden Vertex může obsahovat několik různých atributů (pozice, barva, čas, hmotnost, texturovací koordináty,...)
- Několik Vertexů tvoří jedno primitivum - bod, úsečka, trojúhelník,...



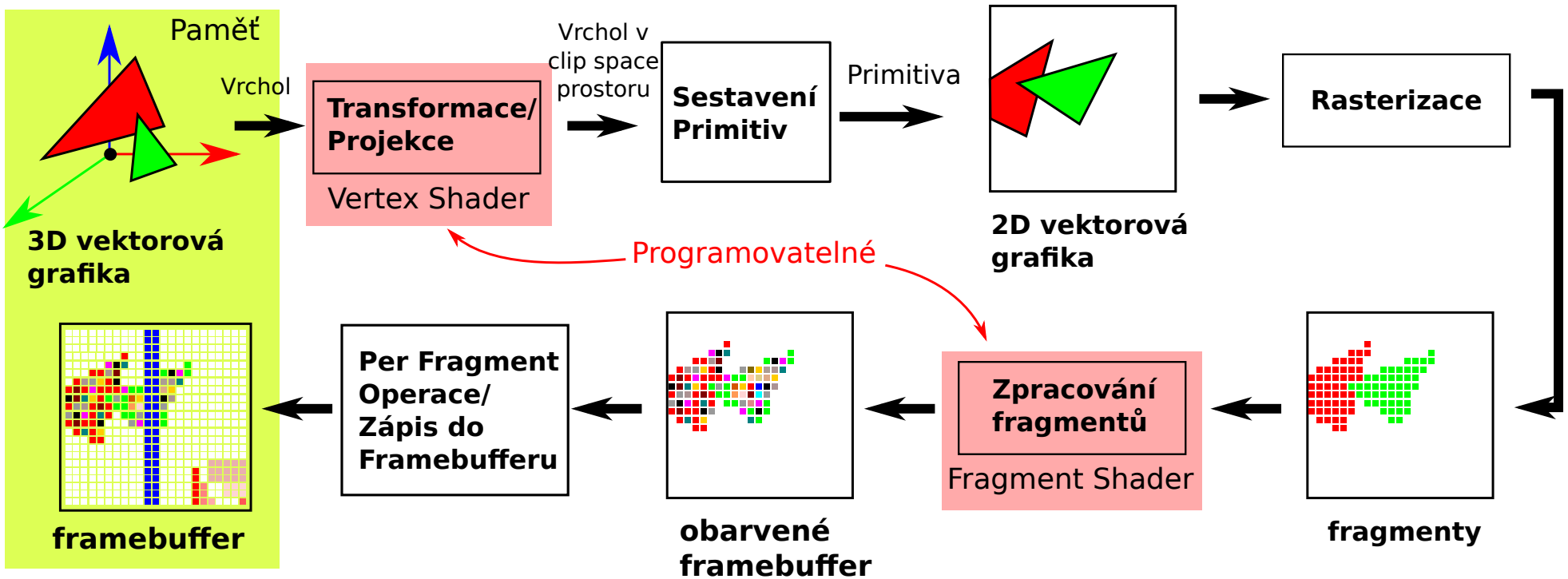


Jeden Vertex obsahuje 5 atributu
1. atribut je složen ze 3 složek



Vertex Buffer Object (VBO) obsahuje seznam Vertexu

- Vektorová data se posílají do pipeline a vrací se v podobě rastru

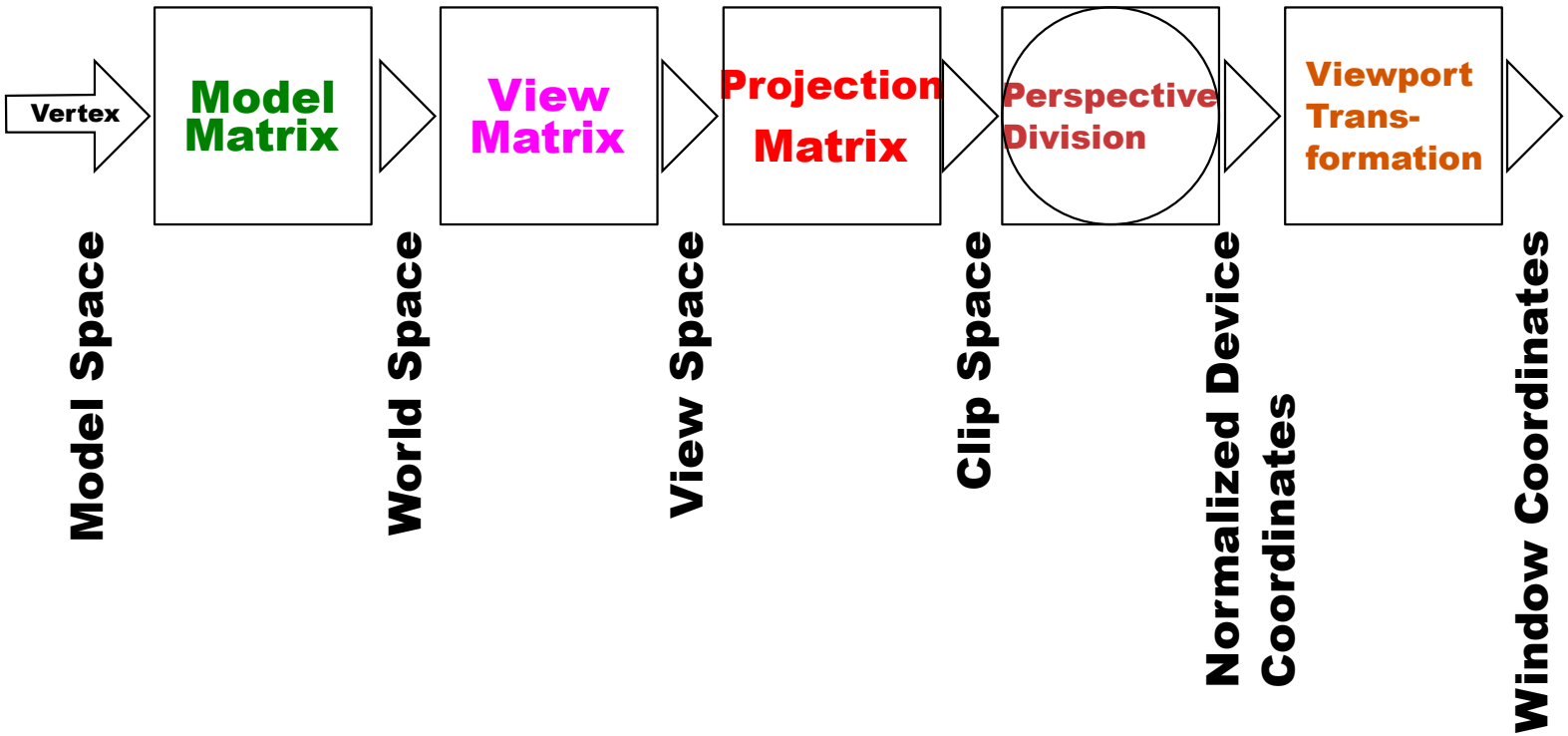


Programovatelné bloky pipeline

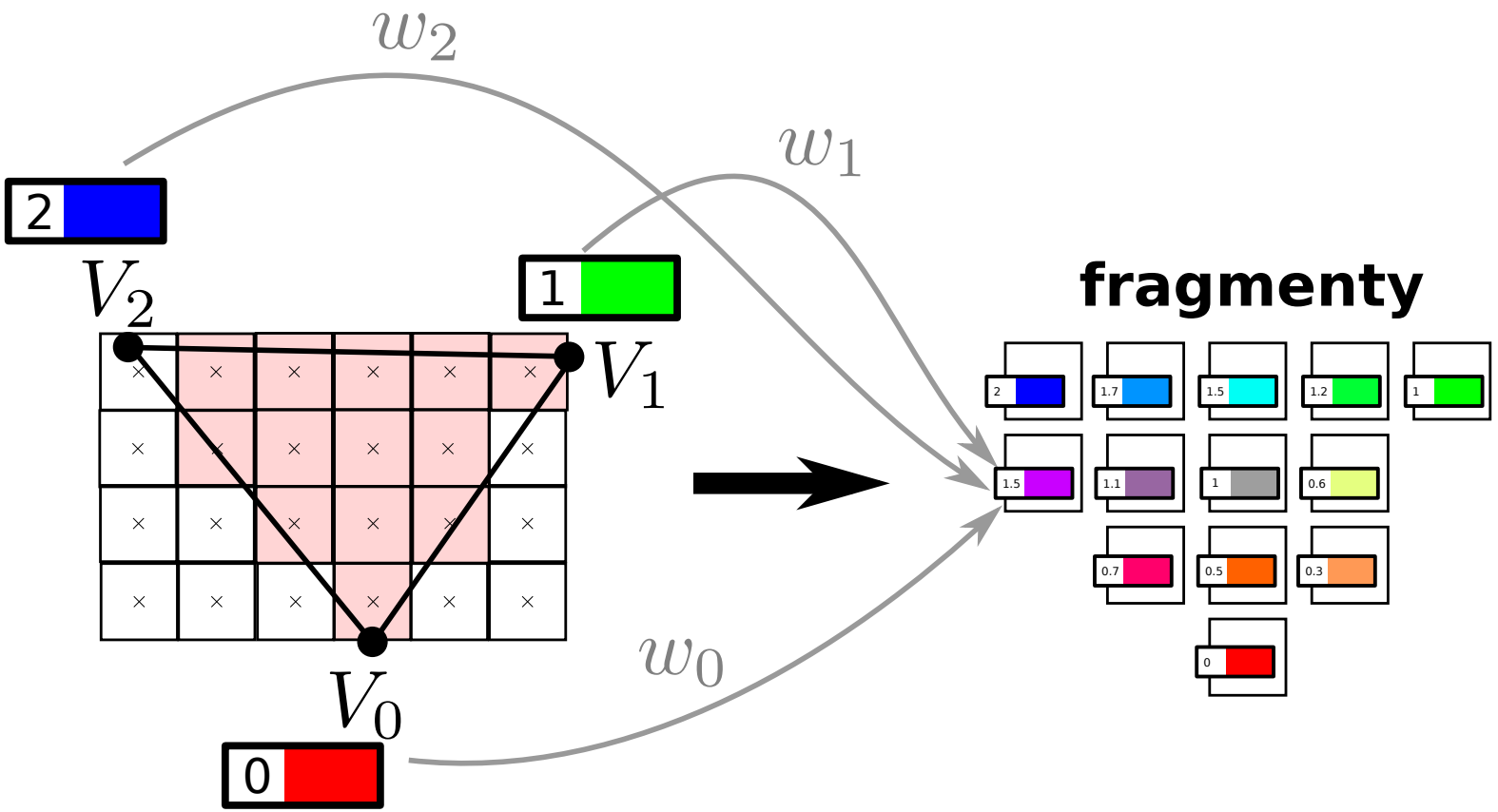
- V zobrazovací pipeline je několik programovatelných bloků
- Vertex Shader blok obsahuje program, který transformuje (přesouvá, promíta vrcholy geometrie)
- Vertex Shader často násobí vrcholy maticemi a přesouvá vrcholy mezi několika prostory.
- Je pouštěn pro každý vrchol a to paralelně
- Fragment Shader blok obsahuje program, který pracuje s fragmenty - vyrasterizovanými úlomky primitiv. Ty často obarvuje.
- Fragment Shader je spouštěn alespoň pro každý vyrasterizovaný fragment.
- Fragment Shader se také spouští v mnoha instancích, invokacích paralelně.

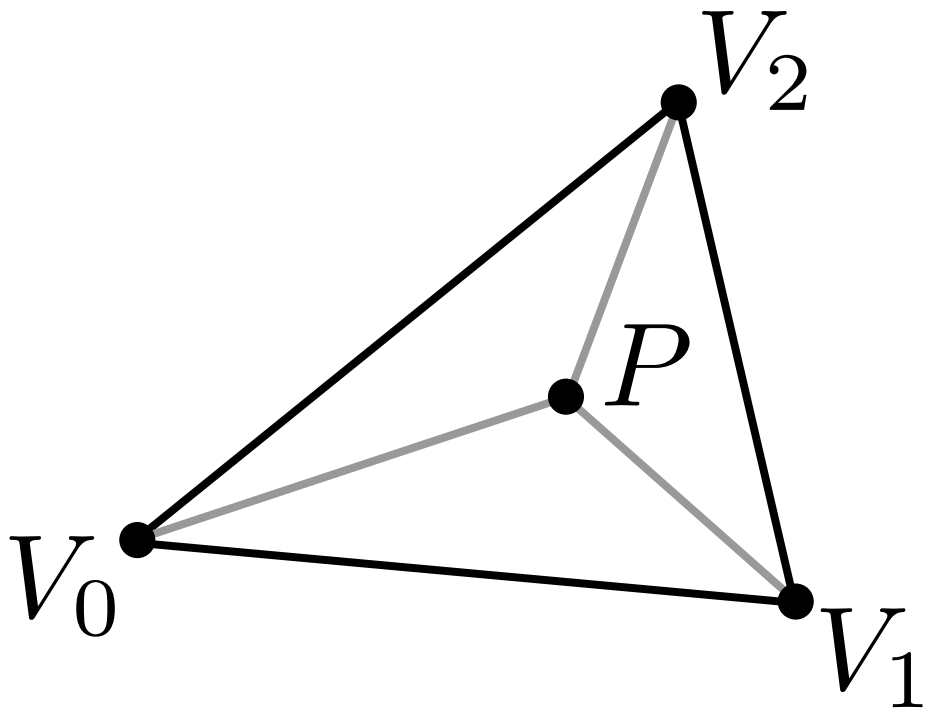
Matice a prostory

- Model je namodelovaný v tzn. model space prostoru
- Vrcholy modelu se po vynásobení modelovou maticí přesunou do world space prostoru - prostoru scény
- Celá scéna se poté transformuje pomocí view matice tak, aby to simulovalo pohled z kamery
- Následuje projekce do clip space pomocí projekční matice
- Výstup vertex shaderu by měl být v clip space



- Vertexy jsou před rasterizací popsány pomocí n-tice atributů
- Rasterizace produkuje fragmenty, pokud jejich střed leží uvnitř primitiva
- Po rasterizaci jsou tyto atributy vloženy do fragmetů pomocí interpolace



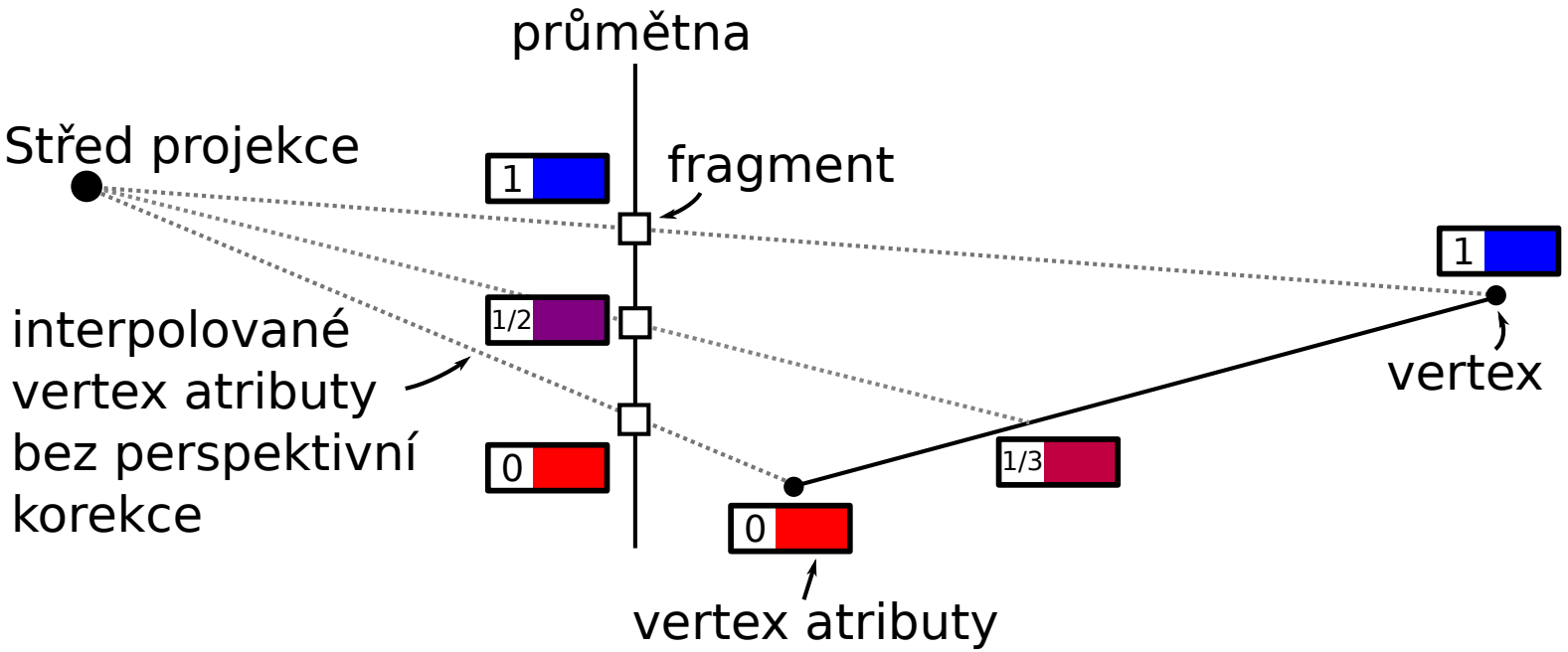


$$P = V_0 \cdot w_0 + V_1 \cdot w_1 + V_2 \cdot w_2$$

$$w_0, w_1, w_2 \in \langle 0, 1 \rangle$$

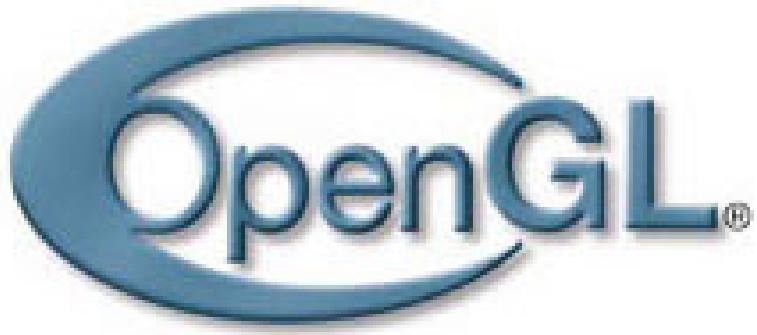
$$w_0 + w_1 + w_2 = 1$$

- Vertex atributy se mohou interpolovat v rovině průmětny nebo v prostoru scény
- Aby se mohlo interpolovat v prostoru scény, musí se provést perspektivní korekce (v OpenGL automaticky/lze vypnout)



OpenGL

- OpenGL Open Graphics Language (Library)
- OpenGL je API pro 3D grafiku
- Vychází z IrisGL od SGI
- Platformně nezávislé
- Použitelné skoro z každého jazyka
- Slouží pro převod scény popsané primitivy (body, čáry trojúhelníky) na 2D rastr obrazovky.
- V novější verzi (4.3) i pro GPGPU
- Obsahuje vlastní jazyk GLSL pro programování GPU



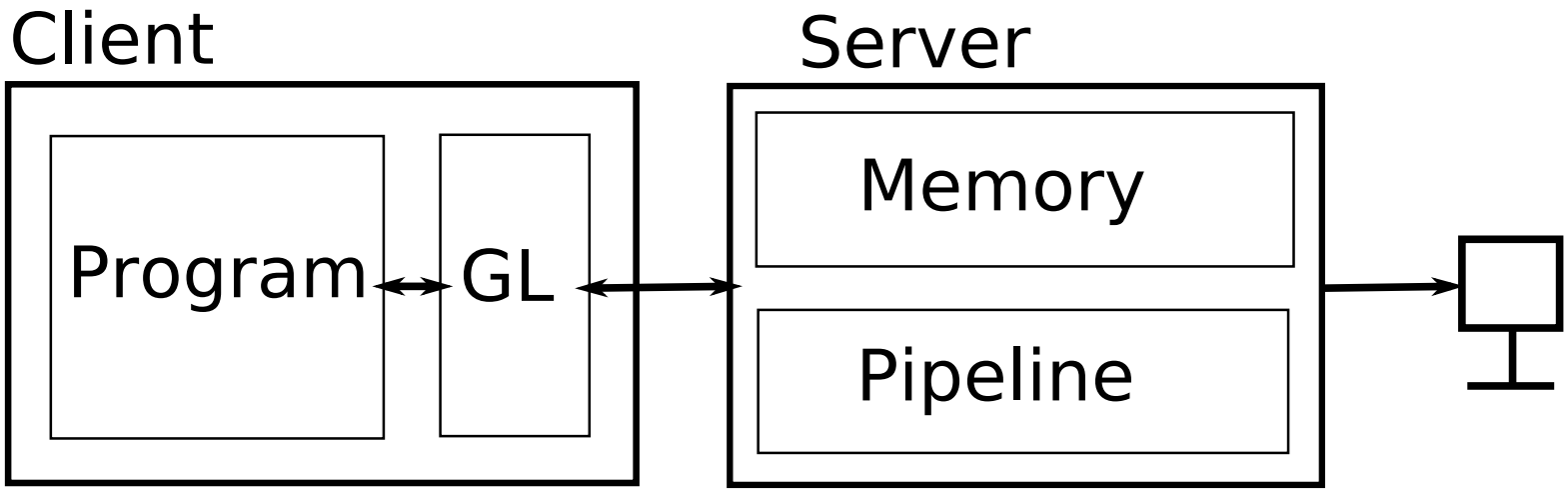
OpenGL - proč používat?

- OpenGL je multiplatformní - Linux, Window, Mac Os X, Android,...
- OpenGL lze použít téměř z každého jazyka - C, C++, Python, Java, Javascript, ...
- OpenGL je zpětně kompatibilní
- OpenGL je nízkoúrovňové
- OpenGL má jednoduché API
- OpenGL je rychlé
- OpenGL je otevřený industriální standard
- WebGL

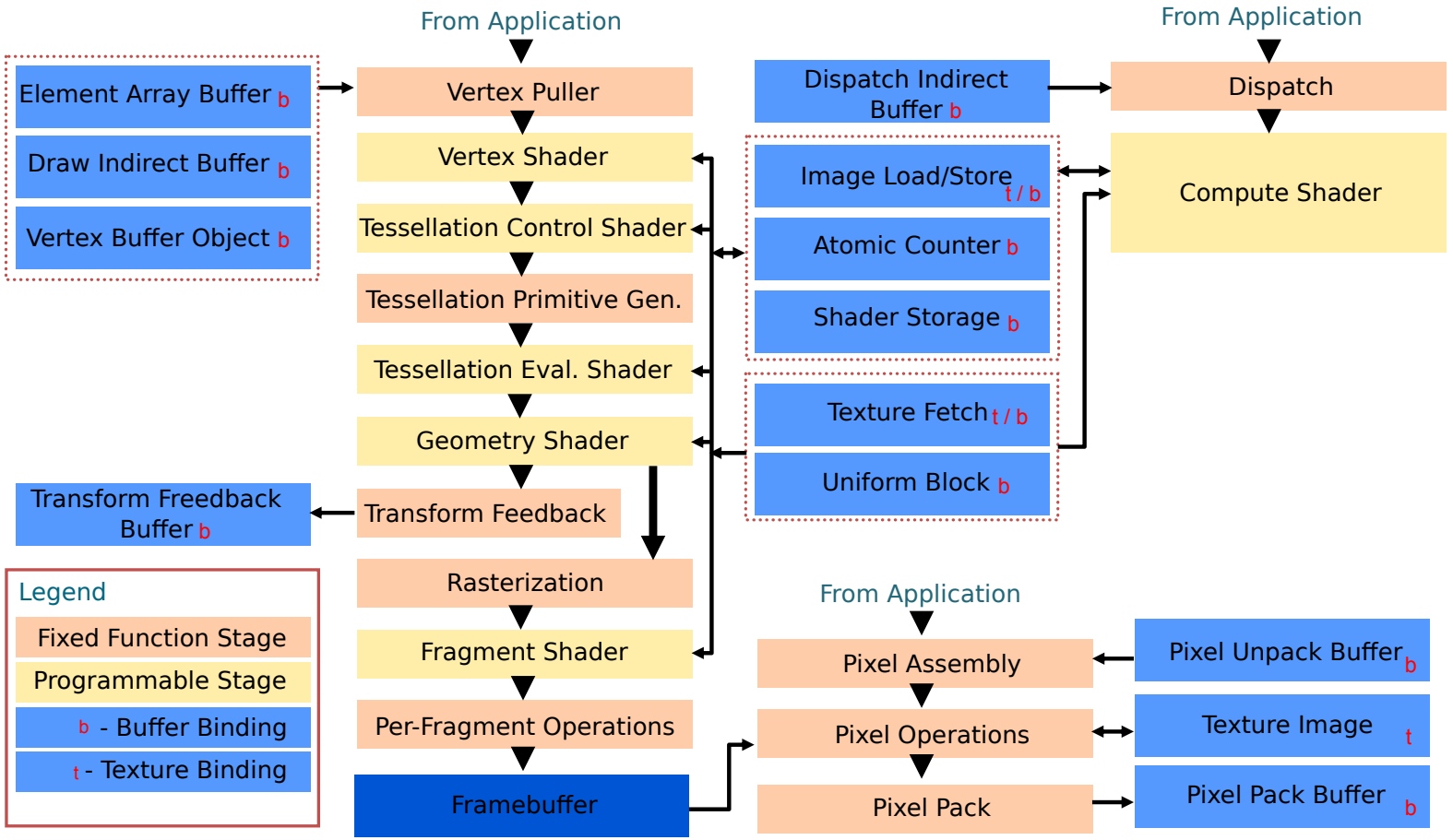
Verze OpenGL

- OpenGL
  - 1.x - fixní pipeline
  - **2.x** - programovatelná pipeline
  - 3.x - geometry shader, **Deprecation**
  - 4.x - Hardwarová tesselace, dvojitá přesnost
  - 4.3 - Compute shadery
  - 4.5 - Direct State Access
- OpenGL ES
  - Vestavěné systémy, mobily, tablety
  - 1.x - fixní pipeline
  - **2.x** - programovatelná pipeline
  - 3.x - Occlusion queries, 3D textury, transform feedback
- WebGL
  - OpenGL ve webovém prohlížeči
  - **Velmi podobné OpenGL ES**

- OpenGL je architektura klient server
- Aplikace běží na CPU a využívá OpenGL pro přístup k GPU



OpenGL 4.3 pipeline



- Jednoduché rozhraní
  - Pouze C funkce
  - Data jsou jen čísla a pole
  - Žádné struct, class
- Stavový stroj
  - Většina příkazů nastavuje stav pipeline
  - Stav se sám nemění
- OpenGL (Rendering) Context
  - Hlavní objekt OGL
  - Mimo OpenGL (WGL/GLX)
  - Zapouzdřuje data, stav, napojení na výstup

glName*NT*(...)

- N - počet parametrů
- T - typ parametrů

b	8b integer	signed char	GLbyte
s	16b integer	short	GLshort
i	32b integer	long	GLint,GLsizei
f	32b float	float	GLfloat,GLclampf
d	64b float	double	GLdouble,GLclampd
ub	8b unsigned	unsigned char	GLubyte,GLboolean
us	16b unsigned	unsigned short	GLushort
ui	32b unsigned	unsigned long	GLuint,GLenum,GLbitfield
*v	Ukazatel a *		

GLvoid glUniform2f(GLuint,GLfloat,GLfloat); GLvoid glUniform2fv(GLuint,GLfloat*);

OpenGL příkazy lze rozdělit do několika skupin

- **Příkazy pro správu OpenGL objektů (10 hlavních OpenGL objektů)**
- **Exekuční příkazy (kreslící a výpočetní příkazy)**
- **Stavové příkazy (nastavují globální stav OpenGL, příkazy pro zjištění stavu)**
- Debugovací příkazy
- Operace s framebufferem
- Příkazy pro synchronizaci (glFinish)
- Utilitní příkazy

OpenGL Objekty

GLvoid glGen*Objects*(GLsizei n,GLuint * objects);
GLvoid glDelete*Objects*(GLsizei n,const GLuint * objects);

- Jméno objektu - GLuint, všechny objekty jsou v API reprezentovány integerem
- 0 rezervována pro prázdný objekt

Objekty:

- **Program**
- **Shader**
- **Buffer**
- **Vertex Array Object**
- Texture
- Framebuffer
- Renderbuffer
- Sampler
- Asynchronous Query
- ProgramPipeline

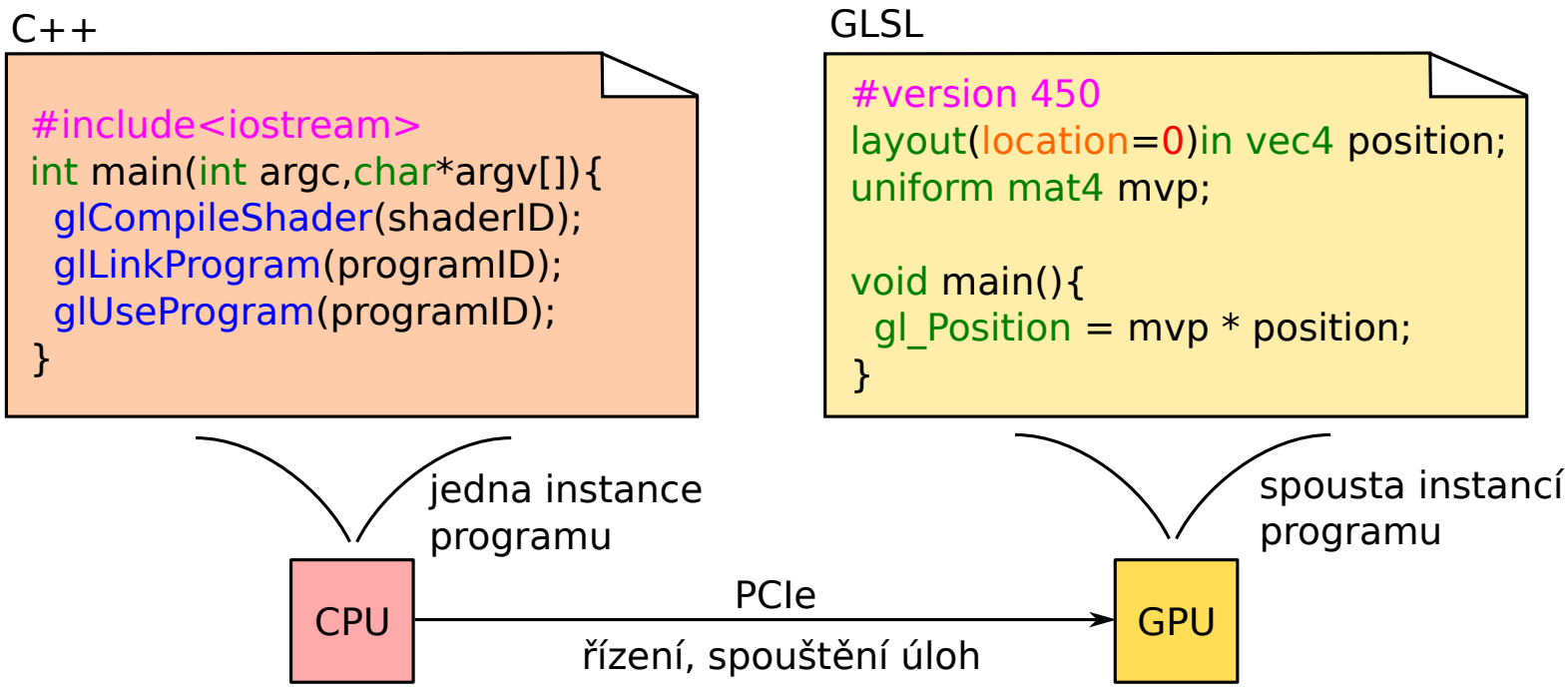
Způsob využití OpenGL pro kreslení

- Pro vykreslení grafiky pomocí OpenGL je potřeba inicializovat několik objektů
- Shader Program(y), Buffer(y), Vertex Array Object(y)
- Inicializace spočívá v kompilaci a likování programů
- Alokaci a kopírování dat na GPU
- Konfigurace stavů OpenGL a konfigurace čtení z GPU paměti
- Spuštění kreslení pomocí vykreslovacích příkazů

Příprava programů a shaderů pro GPU

Dva druhy programovacích jazyků

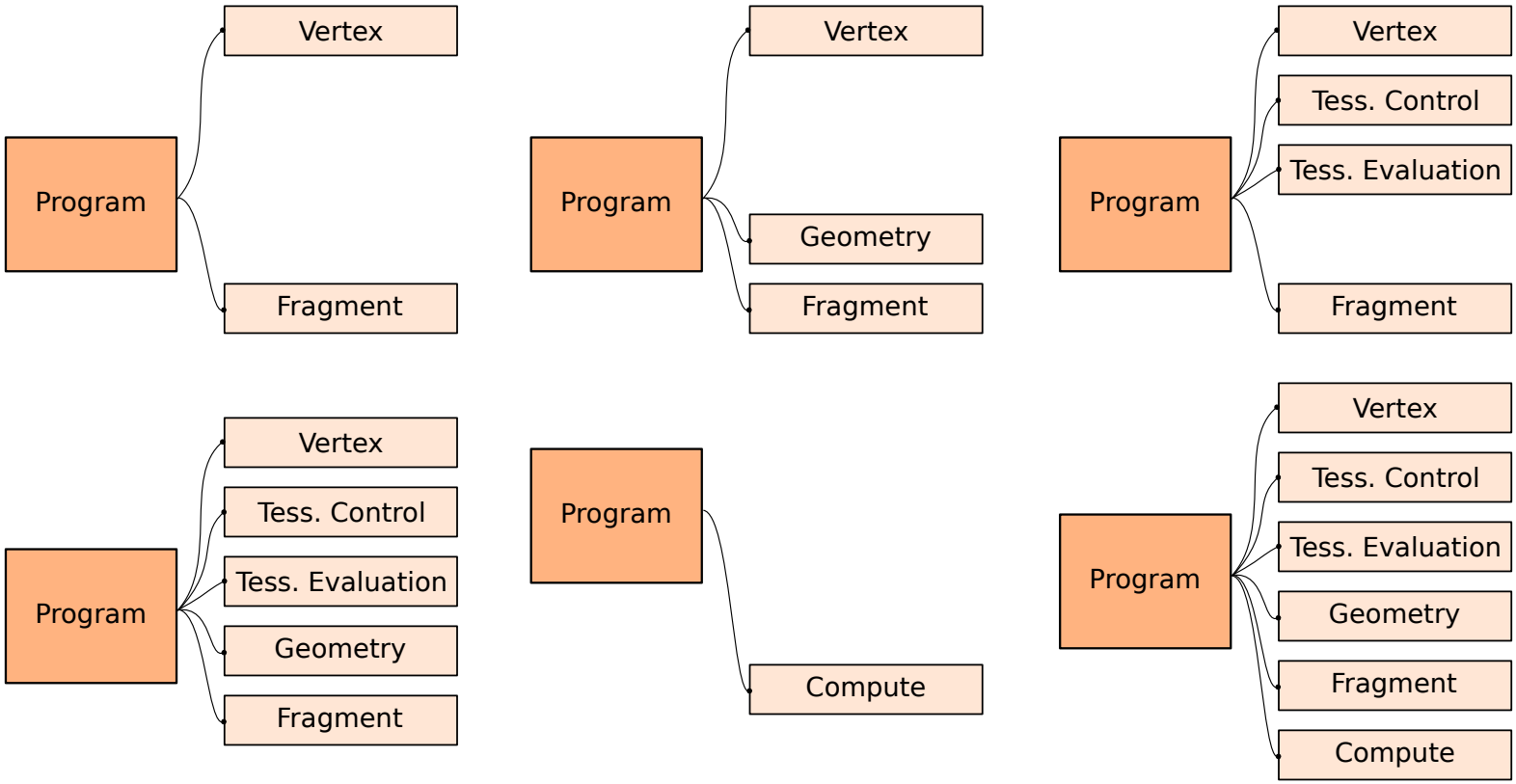
- OpenGL standard popisuje i jazyk GLSL
- Jazyk GLSL popisuje programy, které běží na GPU
- Programátor 3D grafiky píše aplikaci vždy ve 2 jazycích



Programy a Shadery

- Program, který běží na GPU se v OpenGL označuje jako Shader Program
- Shader Program je složen z několika částí (stages), které se nazývají Shader
- Existuje 6 typů shaderů: **vertex**, **fragment**, geometry, tesselation control, tesselation evaluation a compute shader.
- Program nemusí obsahovat všechny typy shaderů.
- Jednotlivé shadery lze sdílet mezi vícero programy.
- Shadery se kompilují (za běhu aplikace)
- Programy se linkují (za běhu aplikace)

Validní a často používané kombinace shaderů:



Kompilace shaderů

```
template<typename...ARGS>
GLuint compileShader(GLenum type,ARGS... sources){
  std::vector<std::string>str = {std::string(sources)...};
  std::vector<const GLchar*>ptr;
  for(auto x:str)ptr.push_back(x.c_str());

  //reserve shader id
  GLuint id = glCreateShader(type);

  //set shader sources
  glShaderSource(id,(GLsizei)ptr.size(),ptr.data(),nullptr);

  //compile shader
  glCompileShader(id);

  //get compilation log
  GLint bufferLen;
  glGetShaderiv(id,GL_INFO_LOG_LENGTH,&bufferLen);
  if(bufferLen>0){
    char*buffer = new char[bufferLen];
    glGetShaderInfoLog(id,bufferLen,nullptr,buffer);
    std::cerr<<buffer<<std::endl;
    delete[]buffer;
    return 0;
  }
  return id;
}
```

```
template<typename...ARGS>
GLuint createProgram(ARGS...args){
  //reserver program id
  GLuint id = glCreateProgram();

  //attach all shaders
  auto dummy0 = {(glAttachShader(id,args),0)...};
  (void)dummy0;

  //link program
  glLinkProgram(id);

  //get linking log
  GLint bufferLen;
  glGetProgramiv(id,GL_INFO_LOG_LENGTH,&bufferLen);
  if(bufferLen>0){
    char*buffer = new char[bufferLen];
    glGetProgramInfoLog(id,bufferLen,nullptr,buffer);
    std::cerr<<buffer<<std::endl;
    delete[]buffer;
    glDeleteProgram(id);
    id = 0;
  }
  //mark shaders for deletion
  auto dummy1 = {(glDeleteShader(args),0)...};
  (void)dummy1;
  return id;
}
```

Ukázka volání/sestavení programu

```
#include<iostream>
#include<fstream>

std::string loadFile(std::string fileName){
  std::ifstream f(fileName.c_str());
  if(!f.is_open()){
    std::cerr<<"file: "<<fileName<<" does not exist!"<<std::endl;
    return 0;
  }
  std::string str((std::istreambuf_iterator<char>(f)),
      std::istreambuf_iterator<char>());
  f.close();
  return str;
}

int main(int32_t argc,char*argv[]){
  ...
  GLuint program = createProgram(
      compileShader(GL_VERTEX_SHADER  ,loadFile("flag.vp")),
      compileShader(GL_FRAGMENT_SHADER,loadFile("usefulFunctions.fp"),
                                       loadFile("flag.fp")));
  ...
}
```

- GLSL - OpenGL Shading Language
- Slouží pro popis programů, které běží na GPU
- Je odvozený od C
- Neobsahuje rekurzi, třídy, výjimky, std knihovny
- Obsahuje vektorové a maticové typy, vestavěné funkce, vestavěné proměnné, synchronizační funkce, typové kvalifikátory, rozšířené adresování vektorů
- Každý shader musí obsahovat main funkci
- Každá main funkce je vykonávána v mnoha instancích v dané části pipeline
- Některé části pipeline mají speciální nastavení

Jazyk GLSL - příklad typy

```
#version 450

void main(){
  //32 bit integer
  int a;
  //32 bit unsigned integer
  uint b;
  //32 bit float
  float c;
  //vector of 4 ints
  ivec4 d;
  //vector of 3 floats
  vec3 e = vec3(1,2,3);
  //matrix 3x3 of floats
  mat3 m;
  //zeroth element of e
  e[0] == e.x == e.r;
  //swizzling
  vec2 f = e.xy; // (1,2)
  f = e.zz; // (3,3)
  //matrix vector multiplication
  e = m*e;
  //constructing ivec4 from vec3 and scalar
  d = ivec4(c,4);
  d = ivec4(c.xx,c.yy);
}
```

Jazyk GLSL - vestavěné funkce

```
abs acos acosh asin asinh atan atanh ceil cos cosh degrees exp exp2 floor fract inversesqrt
log log2 max min mod modf pow radians round roundEven sign sin sinh sqrt tan tanh trunc
clamp cross distance dot floatBitsToInt floatBitsToUint fma frexp intBitsToFloat isinf
isnan ldexp length mix normalize smoothstep step

packDouble2x32 packSnorm4x8 packUnorm2x16 packSnorm2x16
packUnorm4x8 uintBitsToFloat unpackDouble2x32 unpackSnorm4x8 unpackUnorm2x16
unpackSnorm2x16 unpackUnorm4x8 packHalf2x16 unpackHalf2x16

all any bitCount bitfieldExtract bitfieldInsert bitfieldReverse determinant equal
faceforward findLSB findMSB greaterThan greaterThanEqual imulExtended inverse lessThan
lessThanEqual matrixCompMult not notEqual outerProduct reflect refract transpose uaddCarry
umulExtended usubBorrow

textureSize textureQueryLod texture textureProj textureLod
textureOffset texelFetch texelFetchOffset textureProjOffset textureLodOffset textureProjLod
textureProjLodOffset textureGrad textureGradOffset textureProjGrad textureProjGradOffset
textureGather textureGatherOffset textureGatherOffsets textureQueryLevels

dFdx dFdy fwidth
interpolateAtCentroid interpolateAtOffset interpolateAtSample

noise1 noise2 noise3 noise4

EmitStreamVertex EndStreamPrimitive EmitVertex EndPrimitive

barrier memoryBarrier
memoryBarrierAtomicCounter memoryBarrierBuffer memoryBarrierImage memoryBarrierShared
groupMemoryBarrier
imageSize

atomicAdd atomicMin atomicMax atomicAnd atomicOr atomicXor atomicExchange atomicCompSwap

imageSize imageLoad imageStore imageAtomicAdd imageAtomicMin imageAtomicMax
imageAtomicAnd imageAtomicOr imageAtomicXor imageAtomicExchange imageAtomicCompSwap
```

```
#version 450

//promenna a je plnena predchazejici shader stage nebo
//z bufferu GL_ARRAY_BUFFER
//read only
in vec4 a;

//hodnota b bude viditelna v dalsi shader stage nebo
//ve framebufferu
//write only
out vec4 b;

//hodnoty promennych a,b se meni s kazdou invokaci shaderu

//promenna m je ulozena v konstantni pameti, lze ji vycist
//ve vsech shader stage
//read only
uniform mat4 m;

//hodnota m je nemenna pro vsechny invokace shaderu

//promenna d je typu textury (obrazek), je to opaque type, ktery
//lze cist jen pomoci specialnich funkci
//read only
uniform sampler2D d;

//promenna e je lokalni, je ulozena v registru, kazda
//invokace shaderu ma svoji vlastni
vec4 e = vec4(0,1,2,3);

void main(){
  b = e + m * texture(d,a.xy);
}
```

Vertex Shader obsahuje několik důležitých vestavěných proměnných

```
#version 450

void main(){
  //vystupni promenna, sem se zapisuje pozice vrcholu
  //po perspektivni projekci
  vec4 gl_Position;

  //cislo vrcholu
  in int gl_VertexID;
  //cislo instsance
  in int gl_InstanceID;
  //cislo draw callu
  in int gl_DrawID;
  //velikost primitiva typu bod
  out float gl_PointSize;
  out float gl_ClipDistance[];
  out float gl_CullDistance[];
}
```

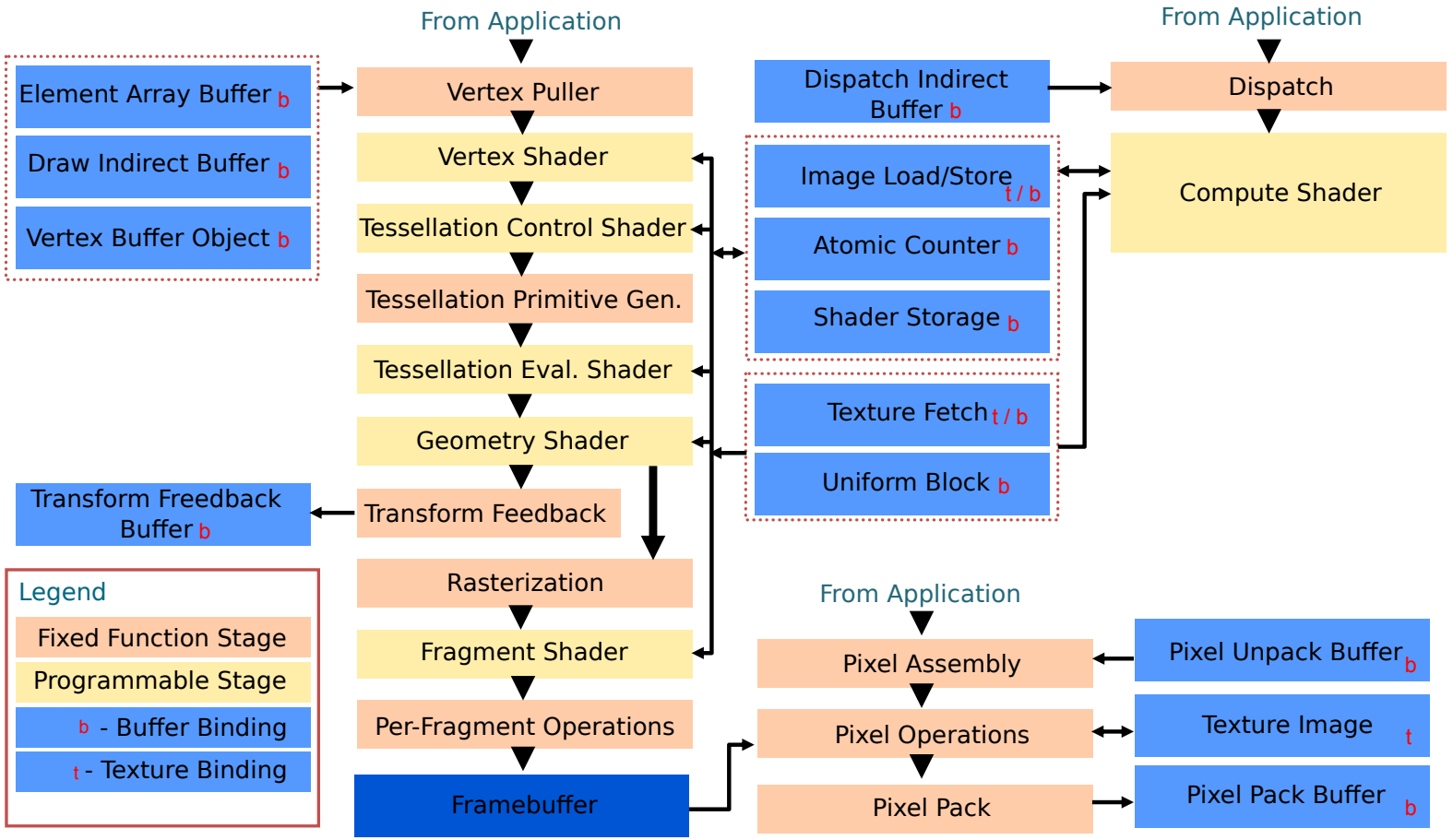
Fragment Shader obsahuje několik důležitých vestavěných proměnných. Výstup fragment shaderu si specifikuje programátor pomocí vlastní výstupní proměnné.

```
#version 450

//vlastni vystup, namapuje se na 0. barevny framebuffer
layout(location=0)out vec4 fColor;

void main(){
  //coordinaty fragmentu ve viewportu
  in vec4 gl_FragCoord;

  //vznikl fragment z privracene strany primitiva
  in bool gl_FrontFacing;

  //pro modifikaci hloubky fragmentu
  //pri zapisu vypina brzky depth test
  out float gl_FragDepth;

  in float gl_ClipDistance[];
  in float gl_CullDistance[];
  in int gl_PrimitiveID;

}
```

Příprava bufferů

- Buffer je objekt zastřešující lineární paměť na GPU
- Může obsahovat jakákoliv data
- Nejčastěji se používá pro uložení vrcholů geometrie (a jejich vlastností), indexů na vrcholy a materiálů
- Buffer lze připojit na několik přípojných míst OpenGL pipeline (binding points)
- Binding point udává sémantiku bufferu
- Pro vrcholy se používá GL_ARRAY_BUFFER, buffer se pak nazývá Vertex Buffer Object (VBO)
- Pro indexy se používá GL_ELEMENT_ARRAY_BUFFER, Element Buffer Object (EBO)
- Pro obecná data se používá GL_SHADER_STORAGE_BUFFER

OpenGL 4.3 pipeline



Vytvoření buffer, nahrání dat

```
float data[]={1,2};//data, ktera budeme vkladat do bufferu
GLuint vbo;//identifikator VBO
glGenBuffers(1,&vbo);//vygenerujeme si identifikator
glBindBuffer(GL_ARRAY_BUFFER,vbo);//aktivujeme a vytvorime VBO
//alokujeme buffer a nahrajeme do nej data
glBufferData(GL_ARRAY_BUFFER,sizeof(data),data,GL_STATIC_DRAW);
```

Změna dat ve VBO.

```
float*ptr;//ukazatel na data
glBindBuffer(GL_ARRAY_BUFFER,vbo);
ptr=(float*)glMapBuffer(GL_ARRAY_BUFFER,GL_READ_WRITE);//namapujeme buffer
ptr[0]=0.5;//nastavime hodnotu prvniho prvku
glUnmapBuffer(GL_ARRAY_BUFFER);//odmapujeme buffer, komitujeme zmeny do GPU
```

Nebo pomoci glBufferSubData.

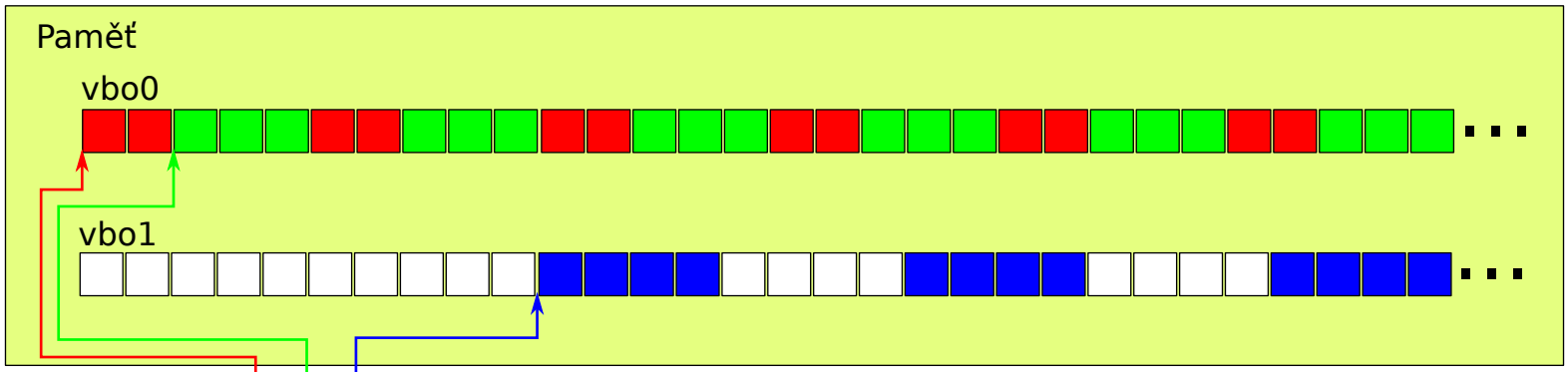
```
glBufferSubData(GL_ARRAY_BUFFER,
  sizeof(float),//nahrajeme nova s offsetem jeden float
  sizeof(float),//nahrajeme jen jeden float
  data);//data
```

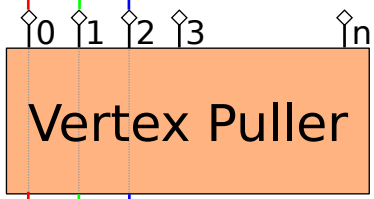
Konfigurace Vertex Pulleru/Vertex Array Object

Vertex Array Object

- Vertex Array Object (VAO) obsahuje konfiguraci Vertex Puller jednoty.
- Vertex Puller čte data z bufferů a plní je do vstupních proměnných v prvním shaderu (vertex shader)
- VAO obsahuje nastavení propojení Shader Programu a Bufferů
- V novějších verzích OpenGL je povinný
- Obsahuje sadu nastavení pro každý Vertex Attribut a nastavení pro indexový buffer
- Jeden Vertex Attribut je napojen na jednu vstupní proměnnou ve Vertex Shaderu
- Mezi nastavení Vertex Attributu patří: číslo bufferu, velikost a typ datové položky, prokládání (stride), offset

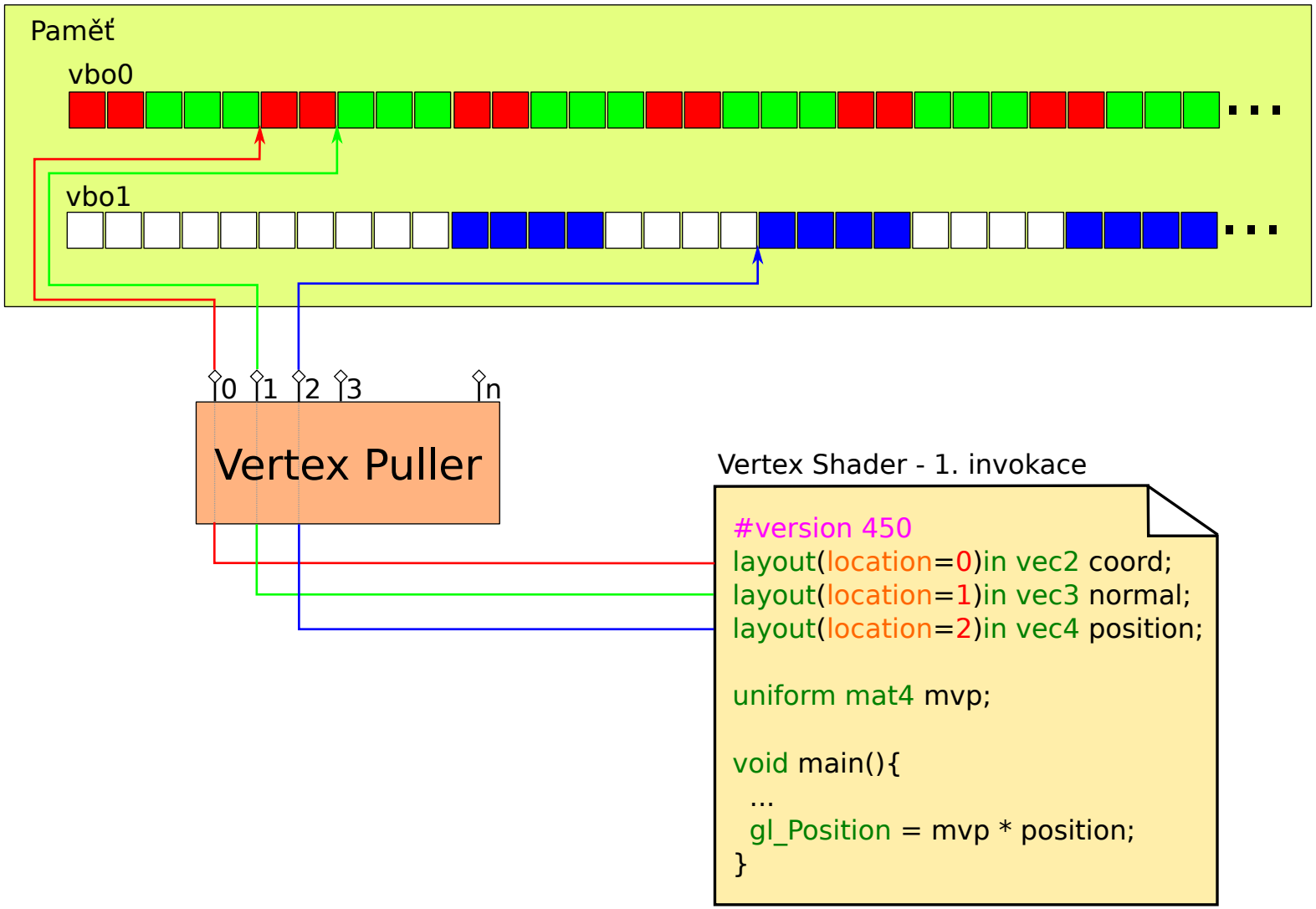
Příklad - ilustrace 0. invokace vertex shaderu





Vertex Shader - 0. invokace

```
#version 450
layout(location=0)in vec2 coord;
layout(location=1)in vec3 normal;
layout(location=2)in vec4 position;

uniform mat4 mvp;

void main(){
  ...
  gl_Position = mvp * position;
}
```


Paměť
vbo0
vbo1
0
1
2
3
n
Vertex Puller
Vertex Shader - 1. invokace
#version 450
layout(location=0)in vec2 coord;
layout(location=1)in vec3 normal;
layout(location=2)in vec4 position;
uniform mat4 mvp;
void main(){
...
gl_Position = mvp * position;
}

VAO - příklad

```
GLuint vao;
glGenVertexArrays(1,&vao);//vygenerovani jmena VAO
glBindVertexArray(vao);//aktivovani VAO
//nyni nastavime buffery a atributy

glBindBuffer(GL_ARRAY_BUFFER,vbo0);

glEnableVertexAttribArray(0);
glVertexAttribPointer(
    0,//cislo vertex attributu
    2,//pocet polozek pro cteni (vec2)
    GL_FLOAT,//typ polozek
    GL_FALSE,//normalizace
    sizeof(float)*5,//stride
    (GLvoid*)(sizeof(float)*0));//offset

glEnableVertexAttribArray(1);
glVertexAttribPointer(1,3,GL_FLOAT,GL_FALSE,sizeof(float)*5,
  (GLvoid*)(sizeof(float)*2));


glBindBuffer(GL_ARRAY_BUFFER,vbo1);

glEnableVertexAttribArray(2);
glVertexAttribPointer(2,4,GL_FLOAT,GL_FALSE,sizeof(float)*8,
  (GLvoid*)(sizeof(float)*10));

glBindVertexArray(0);//deaktivujeme VAO po tomto bode uz jej neovlivnujeme
```

Kreslení a uniformní proměnné

Uniformní proměnné

- Uniformní proměnné jsou uloženy v konstantní paměti
- Narozdíl od vertex atributů se v průběhu kreslení nemění
- Každá invokace shaderu adresuje stejnou hodnotu
- Uniformní proměnné lze využít ve všech shader stage
- Uniformní proměnné jsou vhodné například pro uložení matic, barvy, světla
- Stejně jako objekty v OpenGL zastupuje integerová hodnota, tak i každá uniformní proměnná má svoje integerové jméno
- Toto jméno lze získat z Shader Programu pomocí specializovaných funkcí

Způsob využívání uniformních proměnných

1. Vytvoření Shader Programu
2. Získání integerového jména pomocí jména proměnné v shaderu
3. Aktivování Shader Programu
4. Nahrání dat pomocí vhodné OpenGL funkce

```
GLuint program = 0;
GLint  colorUniform = -1;
void init(){
  //sestaveni programu
  program = createProgram(
      compileShader(GL_VERTEX_SHADER  ,loadFile("flag.vp")),
      compileShader(GL_FRAGMENT_SHADER,loadFile("flag.fp")));

  //ziskani integeroveho jmena z promenne "color" v shaderu
  colorUniform = glGetUniformLocation(program,"color");
}
```

```
#verion 450

layout(location=0)out vec4 fColor;
uniform vec3 color;//uniformni promenna

void main(){
  fColor = vec4(color,1);
}
```

Vykreslení dat

1. Aktivování programu
2. Nastavení uniformních proměnných
3. Aktivování Vertex Array Objectu
4. Zavolání vykreslovacího příkazu

```
void draw(){
  //vymazani barevneho framebuffer
  glClear(GL_COLOR_BUFFER_BIT);

  //aktivovani program
  glUseProgram(program);

  //nastaveni uniformni promenne
  glUniform3fv(colorUniform,1,0,0);

  //aktivovani vao
  glBindVertexArray(vao);

  glDrawArrays(//vykresleni
    GL_TRIANGLES,//typ primitiva
    0,//prvni Vertex
    3);//pocet Vertexu pro vykresleni 3 -> 1 trojuhelnik

  //deaktivovani vao
  glBindVertexArray(0);
```

Per fragment operace

Per fragment operace

- Po fragment shaderu jsou na fragmenty aplikovány Per fragment operace
- Nejzákladnější operací je řešení viditelnosti pomocí depth testu
- Viditelnost se v OpenGL řeší pomocí Depth Buffer (paměť hloubky)
- Další testy jsou Stencil test a Scissor test
- Mezi jiné operace patří Blending, který se využívá pro průhledné objekty

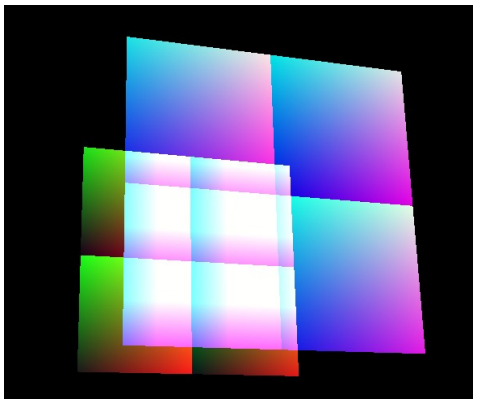
Depth Test a Depth Buffer

- Depth test řeší viditelnost pomocí Depth Bufferu a to na úrovni fragmentů
- Různé způsoby nastavení
- Přesnost depth bufferu není nekonečná (většinou 24 bitů)
- Častý problem depth bufferu je jev známý jako depth Fight
- Brzký depth test - depth test může předběhnout vykonávání fragment shaderu
- Brzký depth test se provádí tehdy, pokud se ve fragment shaderu nemodifikuje hloubka fragmetu
- Brzký depth test může značně urychlit kreslení

```
glEnable(GL_DEPTH_TEST);//zapneme depth test - nastavi se stav pipeline
glDepthFunc(GL_LEQUAL);//fragment s mensi nebo stejnou hloubkou projde
glDepthMask(GL_TRUE);//maskovani zapisu do depth bufferu
```

Bleding - průhledné objekty

- Blending umožňuje kombinovat novou barvu s již zapsanou barvou ve framebufferu
- Blending je řízen pomocí blendovací operace, source a destination faktorů
- Blendovací operace jsou sčítání, odčítání, min, max, ...
- Source faktor se aplikuje na barvu fragmentu
- Destination faktor se aplikuje na barvu již uloženou ve framebufferu
- Nutnost kreslit ve správném pořadí



```
glEnable(GL_BLEND);//zapneme blending
glBlendEquation(GL_FUNC_ADD);//barvy se budou scitat
glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);//nastavime zpusob michani
```

OpenGL kontext, knihovny

SDL - Simple Directmedia Layer

- IO, okna, zvuky, vlákna, ...

GLM - GL mathematics

- Práce s vektory a maticemi

GLEW - OpenGL Extension Wrangler

- OpenGL rozšíření

GLUT - GL Utility Toolkit (IO, okna, ...)
GLEE - GL Easy Extension library
GLAUX - GL Auxiliary Library (IO, okna, ...)
GLFW - (IO, okna, ...)

- glext.h
- void*wglGetProcAddress(const char*name);
- void*glXGetProcAddress(const char*name);
- PFNGL*NECO*PROC glNeco=(PFN...)wglGetProcAddress("glNeco");
- glGetString(GL_EXTENSIONS);
- GL_XYZ_name
- GL_ARB_multisample
- GL_EXT_blend_func_separate
- IBM,NV,ATI,SGIS,...
- GLEE/GLEW

OpenGL kontext

- OpenGL kontext zapouzdřuje věškeré nastavení OpenGL
- Zastřešuje všechny objekty (buffer, programy, textury, ...)
- Po jeho uvolnění, nejsou již data na GPU přístupná
- OpenGL kontext není popsán v OpenGL specifikaci a lze jej vytvořit z externích knihoven
- Kontext má svoji verzi (vertex OpenGL), možnost aktivovat debugging, a různe profily

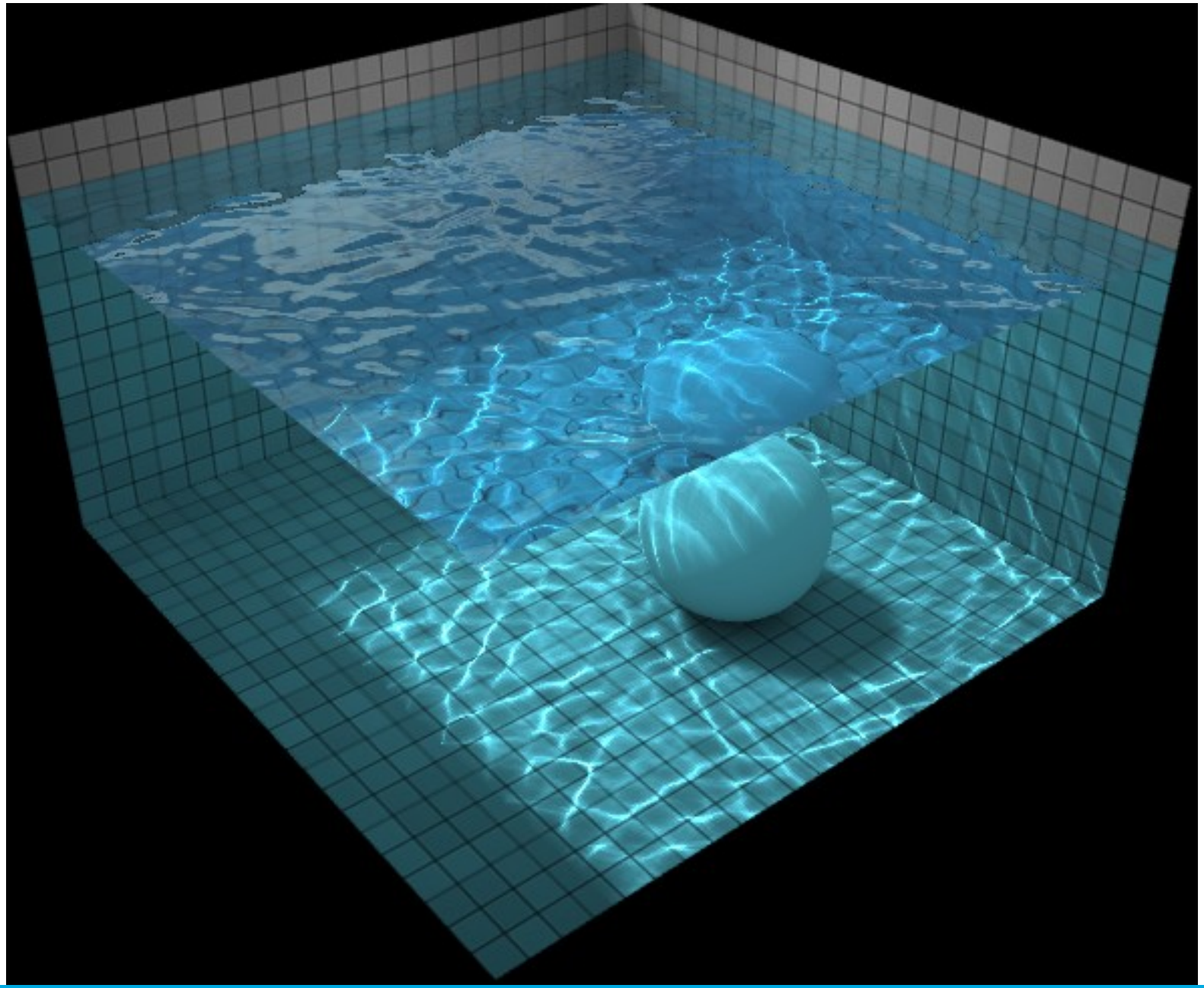
Vytvoření kontextu - SDL2

```
//vytvoreni okna v SDL2
unsigned version = 450;//context version
unsigned profile = SDL_GL_CONTEXT_PROFILE_CORE;//context profile
unsigned flags   = SDL_GL_CONTEXT_DEBUG_FLAG;//context flags
SDL_Init(SDL_INIT_VIDEO);//init. video
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, version/100    );
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION,(version%100)/10);
SDL_GL_SetAttribute(SDL_GL_CONTEXT_PROFILE_MASK ,profile         );
SDL_GL_SetAttribute(SDL_GL_CONTEXT_FLAGS        ,flags           );

SDL_Window*window = SDL_CreateWindow("sdl2",0,0,1024,768,
  SDL_WINDOW_OPENGL|SDL_WINDOW_SHOWN);
SDL_GLContext context = SDL_GL_CreateContext(window);//create context

glewExperimental=GL_TRUE;
glewInit();//initialisation of gl* functions
```

WebGL

- OpenGL aplikace v prohlížeči
- API pro 3D grafiku v prohlížeči
- Programuje se v Javascriptu
- OpenGL ES

WebGL - Voda



WebGL - Modely z doom 3

- http://www.chromeexperiments.com/webgl/
- http://madebyevan.com/
- https://www.shadertoy.com/
- http://playwebgl.com/
- http://media.tojicode.com/q3bsp/
- http://alteredqualia.com/three/examples/webgl_animation_skinning_doom3.html
- ...

Soubor webgl.js

```
var gl;//objekt gl
var program;//shader program
var VBO;//vertex buffer object

//funkce pro inicializaci WebGL
function initGL(canvas){
  try{
    gl=canvas.getContext("experimental-webgl");//kontext
    gl.viewportWidth=canvas.width;//sirka view portu
    gl.viewportHeight=canvas.height;//vyska view portu
  }catch(e){
  }
  if(!gl){
    alert("Could not init WebGL");
  }
}
```

Soubor webgl.js

```
//funkce vytvori shader program
function CreateShaderProgramVSFS(vertexSource,fragmentSource){
  var vshader=gl.createShader(gl.VERTEX_SHADER);//vertex shader
  var fshader=gl.createShader(gl.FRAGMENT_SHADER);//fragment shader
  gl.shaderSource(vshader,vertexSource);//nahrani zdrojaku v. shaderu
  gl.shaderSource(fshader,fragmentSource);//nahrani zdrojaku f. shaderu
  gl.compileShader(vshader);//kompilace vertex shaderu
  gl.compileShader(fshader);//kompilace fragment shaderu
  if(!gl.getShaderParameter(vshader,gl.COMPILE_STATUS))
    alert(gl.getShaderInfoLog(vshader));//chyba
  if(!gl.getShaderParameter(fshader,gl.COMPILE_STATUS))
    alert(gl.getShaderInfoLog(fshader));//chyba

  var programID=gl.createProgram();//vytvorime novy shader program
  gl.attachShader(programID,vshader);//pripojime vertex shader
  gl.attachShader(programID,fshader);//pripojime fragment shader
  gl.linkProgram(programID);//slinkujeme shader dohromady
  if(!gl.getProgramParameter(programID,gl.LINK_STATUS))
    alert("LINK ERROR");//chyba

  return programID;//navrat shader programu
}
```

Soubor webgl.js

```
function WebGLStart(){
  var canvas=document.getElementById("test");//ziskani platna
  initGL(canvas);//inicializace WebGL
  program=CreateShaderProgramVSFS(//vytvoreni programu
      "attribute vec2 Pos;void main(){gl_Position=vec4(Pos,0,1);}",
      "void main(){gl_FragColor=vec4(1,0,0,0);}");
  gl.useProgram(program);//pouzijeme shader program
  program.posatt=gl.getAttribLocation(program,"Pos");//atribut pozice
  gl.enableVertexAttribArray(program.posatt);//povolime atribut pozice
  VBO=gl.createBuffer();//vytvorime VBO
  gl.bindBuffer(gl.ARRAY_BUFFER,VBO);//navazeme vytvorime VBO
  var Data=[0,0,.3,.3,-.3,-.3,.3,-.3,-.3,.3];//data
  gl.bufferData(gl.ARRAY_BUFFER,new Float32Array(Data),gl.STATIC_DRAW);
  gl.clearColor(0,0,0,1);//cerna barva pozadi
  gl.enable(gl.DEPTH_TEST);//depth test
  gl.viewport(0,0,gl.viewportWidth,gl.viewportHeight);//nastaveni viewportu
  gl.clear(gl.COLOR_BUFFER_BIT|gl.DEPTH_BUFFER_BIT);//vycistime framebuffer
  gl.vertexAttribPointer(program.posatt,2,gl.FLOAT,false,0,0);//attibut
  gl.drawArrays(gl.POINTS,0,5);//vykreslime 5 bodu
}
```

Soubor index.html

```
<!-- Soubor s WebGL !-->
<script type="text/javascript" src="webgl.js">
</script>

<html>
  <body onload="WebGLStart();">
    <canvas id="test" style="border: none;" width="500" height="500"/>
  </body>
</html>
```

Budoucnost OpenGL

- Zaměří se vývojáři více na OpenGL?
- Herní společnost Valve vytvořila Linuxovou verzi svého klienta Steam
- Začala vydávat hry pod Linux - OpenGL
- Port hry Left4Dead běží rychleji pod Linuxem - pod OpenGL
- `http://blogs.valvesoftware.com/linux/faster-zombies/`

OpenGL - Left4Dead - před třemi roky

OpenGL dnes

- Kolem 2000 her portovaných pod Linux na Steamu
- Portování většinou znamená využít SDL + OpenGL
- Unity Engine pod Linuxem - OpenGL
- Unreal Engine 4 pod Linuxem - OpenGL
- Cry Engine 3 pod linuxem - OpenGL

OpenGL budoucnost?

- Problém spočívá hlavně ve složitosti driverů
- Přes 2 milioný řádků kódu pro drivery
- OpenGL je poměrně vysokoúrovňové (buffery, schovaná fronta příkazů, synchronizace, ...)
- OpenGL je rozsáhlé - přes 800 stran specifikace bez popisu GLSL
- Nejednotná mezivrstva jazyka
- Kvůli složitost obsahují drivery spousty bugů
- Kvůli obecnosti a vysokoúrovňovosti není OpenGL tak rychlé jak by mohlo být (driver overhead)
- Vznikla nová specifikace Vulkan a Spir-V

- Nízkoúrovňové api
- Mnohem jednodušší drivery
- Jednotná mezivrstva Spir-V, do které se překládají programy (GLSL, OpenCL, ...)
- Fronty, command buffer, render passy, shader pipeline, více vláken
- Více práce pro aplikační programátory (podobně jako tomu bylo při přechodu na programovatelnou pipeline)
- Je pravděpodobné, že bude implementace OpenGL pomocí Vulkan API

Zdroje informací

- `http://www.opengl.org/sdk/docs/`
- `http://www.opengl.org/documentation/glsl/`

Děkuji za pozornost!