

# PGR - GPU, Rendering pipeline, OpenGL

Tomáš Milet

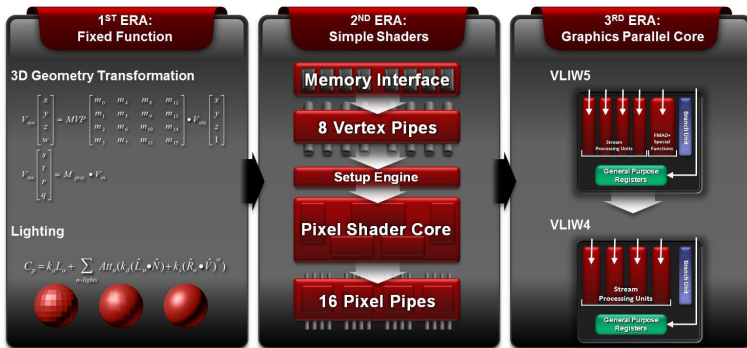
Brno University of Technology, Faculty of Information Technology  
Božetěchova 1/2. 612 66 Brno - Královo Pole  
[imilet@fit.vutbr.cz](mailto:imilet@fit.vutbr.cz)



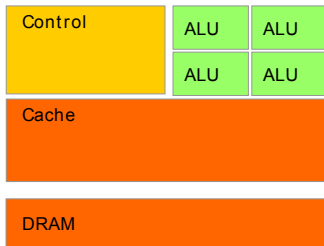
21. prosince 2022

## Overview of GPU architecture / Přehled architektury GPU

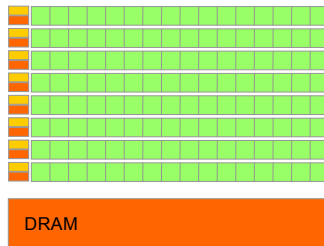
- At first, everything was fixed in HW / just 2D acceleration.
- 3D acceleration next, specialized HW.
- Next, partially programable pipeline.
- Now, large number of cores for general computation, (2D emulated).
- Nejprve pouze 2D akcelerace
- Poté 3D akcelerace, specializovaný HW
- Dál částečně programovatelná pipeline
- Nyní velké množství výpočetních jednotek pro obecné výpočty (2D akcelerace emulovaná pomocí 3D)



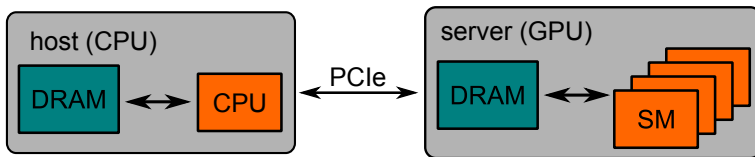
- CPU - small number of very fast computing units.
- Large caches, large control, out-of-order instruction execution, ...
- GPU - large number of not that fast computing units, simpler units.
- Smaller caches, small control, more transistors allocated for computation, ...
- GPU - data parallel, semi task parallel
- CPU - task parallel, semi data parallel
- CPU - málo, velmi výkonných výpočetních jednotek
- Velká keš, velké řízení, vykonávání instrukcí mimo pořadí, ...
- GPU - velké množství méně výkonných, jednodušších výpočetních jednotek
- Malé keše, malé řízení, víc tranzistorů pro výpočty, ...
- GPU - data parallel, náznak task parallel
- CPU - task parallel



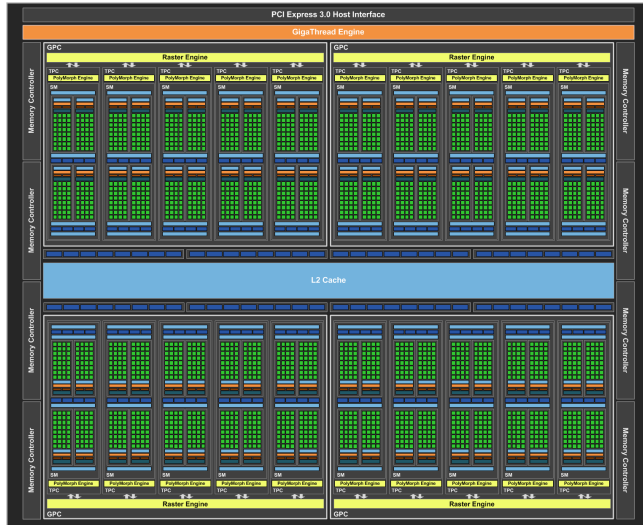
CPU



GPU



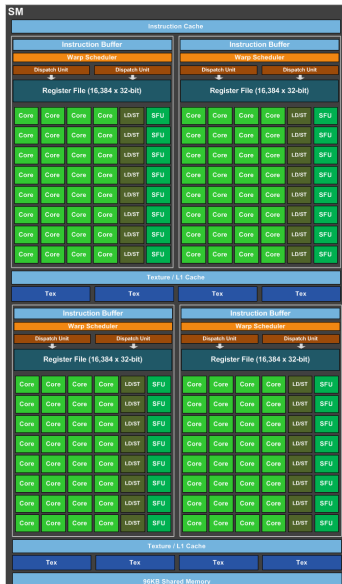
- GPU is composed from multi-processors and device memory.
- Nvidia - streaming multi processor (SM).
- AMD - compute unit (CU).
- Multi-processor is composed of SIMD units and various kinds of memory.
- Computation executed on one multi-processor is independent from computation on other multi-processors.
- SIMD cores are commonly referred as shader unit (AMD Radeon HD7970M 20xCU, 64 shader unit per CU, = 1280 shader unit).
- Grafická karta je složena z několika multiprocesorů a grafické paměti.
- Nvidia - streaming multi processor (SM).
- AMD - compute unit (CU).
- Tento multiprocessor je dále složen z velkého množství jader a různých druhů paměti.
- Akce, která běží na jednom multiprocesoru je nezávislá na akci, která běží na jiném multiprocesoru.
- Jádra bývají označována jako shader unit (AMD Radeon HD7970M 20xCU, 64 shader unit na CU, = 1280 shader unit).

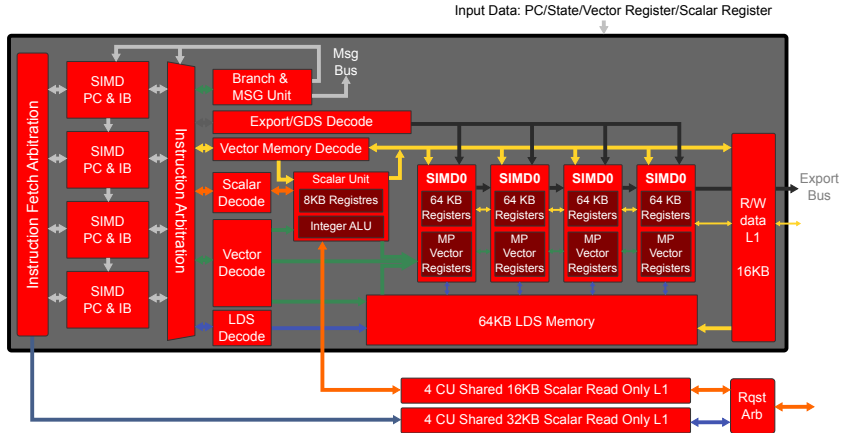




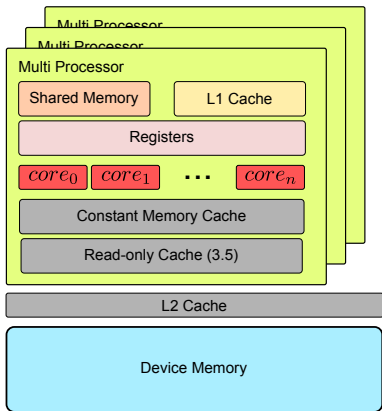








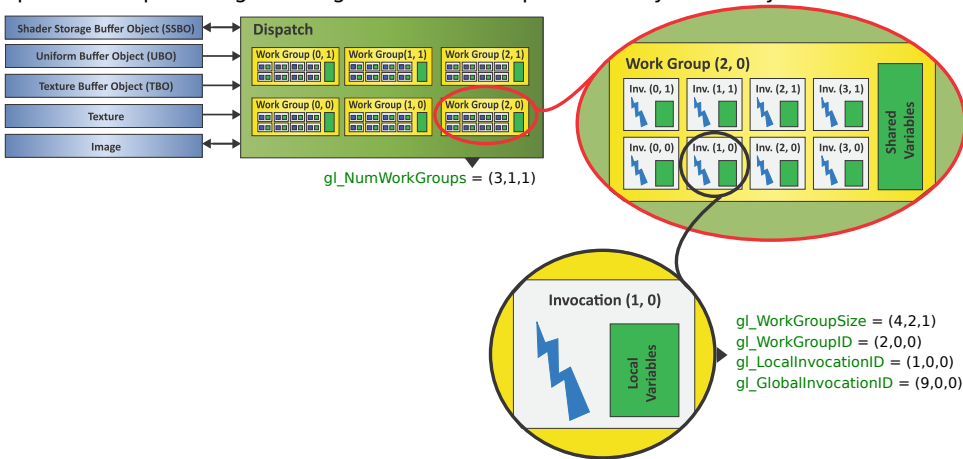
- Today's GPUs are highly programmable.
- Few parts remain fixed for configurable.
- Rasterization - converting vector primitives to fragments.
- Tessellation - splitting polygons into many sub-polygons.
- Texture units - filtering, wrapping, compression, ...
- Per-fragment-operation, depth buffer, stencil buffer, ...
- Ray-tracing, ...
- And more
- Some parts are inaccessible from some APIs (CUDA cannot use rasterization).
  
- Dnešní GPU jsou značně programovatelné
- Některé části stále zůstávají hardwarově zadrátované
- Rasterizace - převod trojúhelníků na fragmenty
- Tessellace - rozřezání polygonů na mnoho podpolygonů
- Texturovací jednoty - filtrování, opakování na okrajích, ...
- Depth buffer, Stencil buffer, ...
- A další
- Ke většině lze přistoupit pouze z některých API (OpenGL, Vulkan, DirectX)

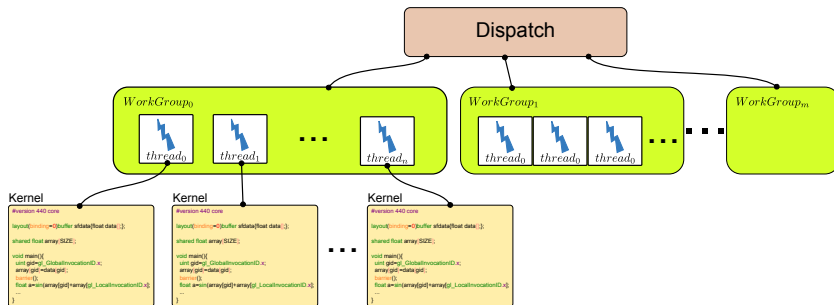


- Different memory types with different size and speed.
- General rule: closer to core, faster, smaller memory.
- Registers are fastest (Ada contains 256KB of registers per SM).
- Shared/Local memory is used for thread communication.
- Device memory/global memory is cached using L2 cache. Large, big latency (100s of clocs).
- Spousta různých pamětí s různou velikostí a rychlostí.
- Obecně platí, čím blíže k jádru, tím rychlejší a tím menší.
- Registry jsou nejrychlejší (na Ada je 256KB registrů na SM).
- Sdílená paměť slouží pro komunikaci mezi vlákny.
- Device memory (global memory) je velká (16GB i více), ale pomalá paměť.

- Thread is kernel instantiation with unique index.
- Kernel/Shader is program composed of instructions.
- User launches 1000s of threads - Dispatch.
- Dispatch is divided into work-groups. Whole work-group runs on one SM.
- WG are further divided into Warps/Wavefronts (32/64 threads sub-groups).
- Warps/Wavefronts are executed on SIMD unit. (One warp per one SIMD).
- Threads in WG can be synchronized and can use shader memory.
- WG are 1D, 2D, 3D (dictates thread ordering and indices).
- Terminology differs (OpenCL, Cuda, Compute Shader, ...).
- Na GPU použijeme instance kernelů - vlákna.
- Kernel je program složený z instrukcí (podobně jako shader) a běží ve vláknu.
- Instrukce v kernelu jsou spouštěny v mnoha instancích na jádrech SM.
- Vlákna (thread, invocation, work-item) jsou seskupovány do pracovních skupin (work-group).
- Vlákna v pracovních skupinách mohou být na sobě závislá.
- Skupiny mohou být 1D, 2D, 3D (určuje pořadí vláken a jejich index).
- Mnoho work-group tvoří dispatch (taký může být 1D, 2D, 3D).
- Terminologie se liší (OpenCL, Cuda, Compute Shader, ...).

## OpenGL Compute Programming Model and Compute Memory Hierarchy





- Threads are grouped to work-groups.
- Work-group can be 1D, 2D, 3D.
- WG are grouped into dispatch.
- Dispatch can also be 1D, 2D, 3D.
- One SM can launch multiple work-groups, if there is enough resources (registers, shader memory).
- Vlákna jsou seskupena do skupin work-group.
- Work-group může být 1D, 2D, 3D.
- Work-groupy jsou seskupeny do dispatch.
- Dispatch může být také 1D, 2D, 3D.
- Na jeden SM se může pustit vícero Work-group, pokud na to vystačí zdroje (registry, sdílená paměť)!

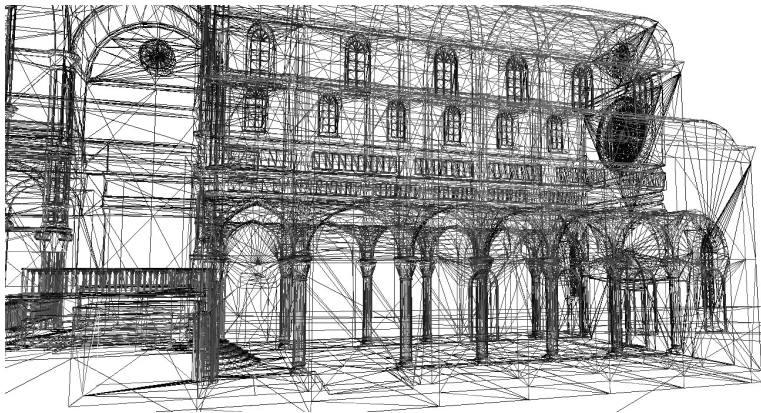
Model representation / Repräsentace modelu



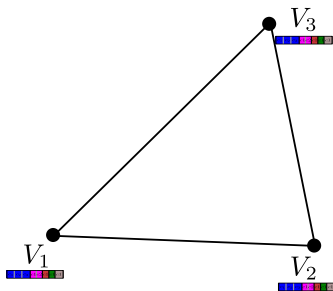
- Main goal of the GPU is to convert 3D vector graphics to raster graphics (framebuffer)
- Hlavním úkolem grafické karty je převod 3D vektorové grafiky na rastrový obrázek (framebuffer).



- Boundary representation (B-rep) - vector graphics.
- Povrchová reprezentace - vektorová data.



- 3D Vector graphics is not just points, lines and polygons.
- A GPU renders geometry based on vertices.
- A vertex is not just point in 3D space.
- A vertex is structure of user specific data.
- Jeden Vertex může obsahovat několik různých atributů (pozice, barva, čas, hmotnost, texturovací koordináty,...)
- Několik Vertexů tvoří jedno primitivum - bod, úsečka, trojúhelník,...
- 3D Vektorová grafika nejsou jen body, hrany a polygony.
- GPU kreslí geometrii založenou na vertexech.
- Vertex není jen bod v 3D prostoru.
- Vertex je struktura uživatelských dat.
- Jeden Vertex může obsahovat několik různých atributů (pozice, barva, čas, hmotnost, texturovací koordináty,...)
- Několik Vertexů tvoří jedno primitivum - bod, úsečka, trojúhelník,...



**Example: 1 vertex  
contains 5 attributes.**

**1. attribute is composed of  
3 elements.**



**Vertex Buffer Object (VBO)  
1 or more VBO contains  
list of vertices.**

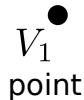
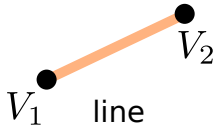
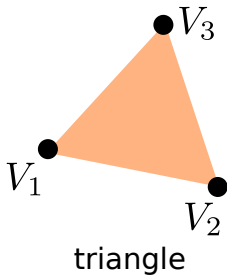
The scene is usually composed using:

- A scene graph - user friendly, hierarchical structure of the scene.
- A model - one object of the scene. Can be instanced.
- A mesh - one piece of model. 1 material, 1 kind of geometry.
- A primitive - geometric piece, 2 categories: base primitives (point, line, triangle) and primitive.
- A vertex - structure of user specific vertex attributes (data).
- A vertex attribute - 1-4D vector of floats or integers.
- A vertex buffer - memory containing vertices.
- A element/index buffer - memory containing indices to vertices.

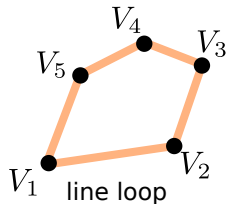
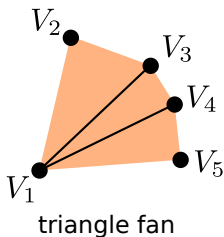
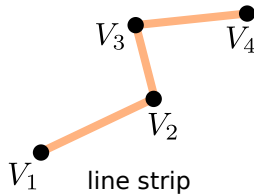
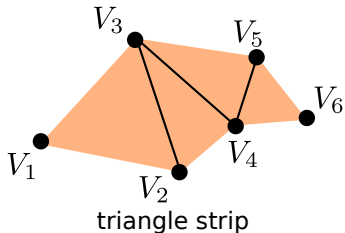
Scéna je obvykle složena z:

- Graf scény - uživatelsky přívětivá hierarchická struktura scény.
- Model - jeden objekt scény, který je možné instanciovat.
- Mesh - jeden kousek modelu, obvykle má jen jeden materiál a jeden typ geometrie.
- Primitivum - geometrický kousek, dvě kategorie: základní primitiva (bod, úsečka, trojúhelník) a primitiva.
- Vertex - struktura uživatelských vertex atributů (data).
- Vertex atribut - 1-4 dimenzionální vektor floatů nebo integerů.
- Vertex Buffer - paměť obsahující vrcholy.
- Index/Element buffer - paměť obsahující indexy na vrcholy.

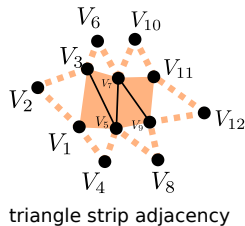
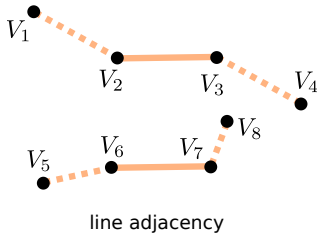
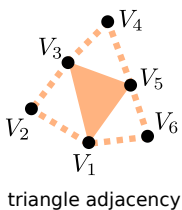
- The base primitive is subset of all primitives. There are only 3 types - triangles, lines, points. There is specialized rasterization hardware for each type. More complex primitives are composed of these base primitives.
- Základní primitiva tvoří podmnožinu všech primitiv. Jsou jen tři typy (trojúhelník, úsečka, bod). Pro každý je rasterizační hardware. Složitější primitiva jsou z nich složena.



- Base primitives together with these primitives form commonly used primitives. Main purpose is to reduce memory footprint and processing time.
- Kromě základních primitiv se ještě běžně používají tato primitiva. Hlavním účelem je ušetřit paměť a čas zpracování.



- There are also primitives with adjacency (mainly for geometry shader).
- Existují také primitiva se sousedností (hlavně pro geometry shader).



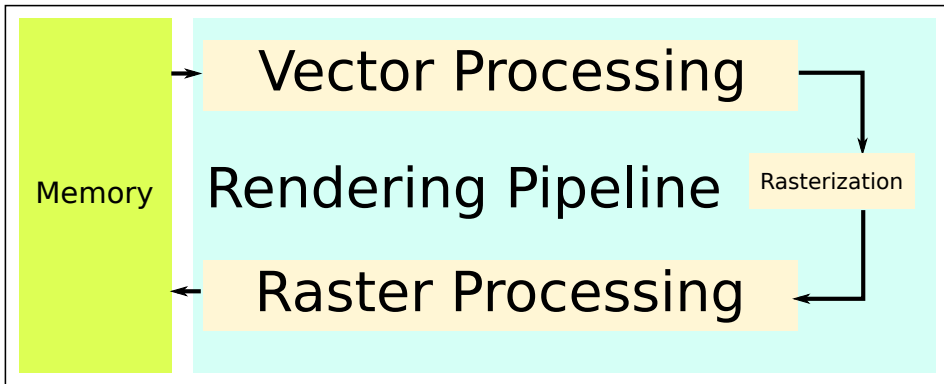
## Rendering Pipeline / Zobrazovací řetězec



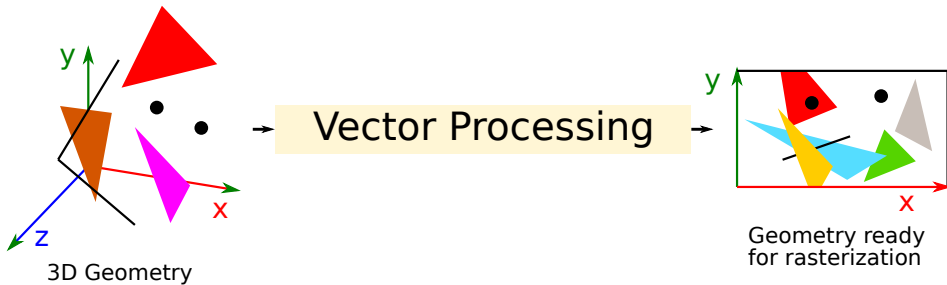
- GPU is divided into memory and rendering pipeline.
- GPU je rozděleno na paměť a vykreslovací řetězec.



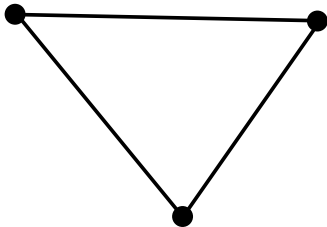
- Rendering Pipeline is divided into 2 parts: vector and raster.
- Splitting element is rasterization.
- Zobrazovací řetězec je rozdělen na 2 části - vektorovou a rastrovou.
- Dělicím prvkem je rasterizace.



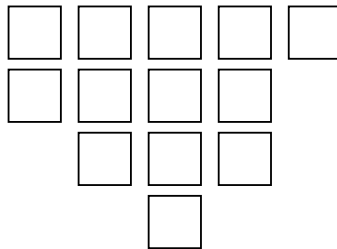
- The main goal of vector part is to transform geometry.
- It processes vertices, primitives, it performs transformations, clipping, culling, tessellation, projection,...
- Hlavním úkolem vektorové části je transformace geometrie.
- Počítá vertex, primitiva, provádí transformace, ořez, teselaci, projekci, ...



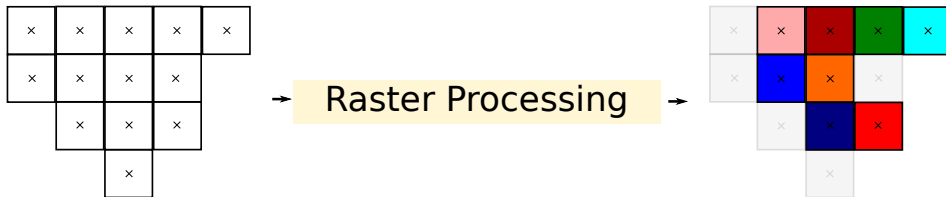
- The goal of rasterization is to convert vector graphics elements (triangles, lines, points) to fragments.
- Cílem rasterizace je převod vektorových grafických elementů (trojúhelníky, čáry, body) na fragmenty.



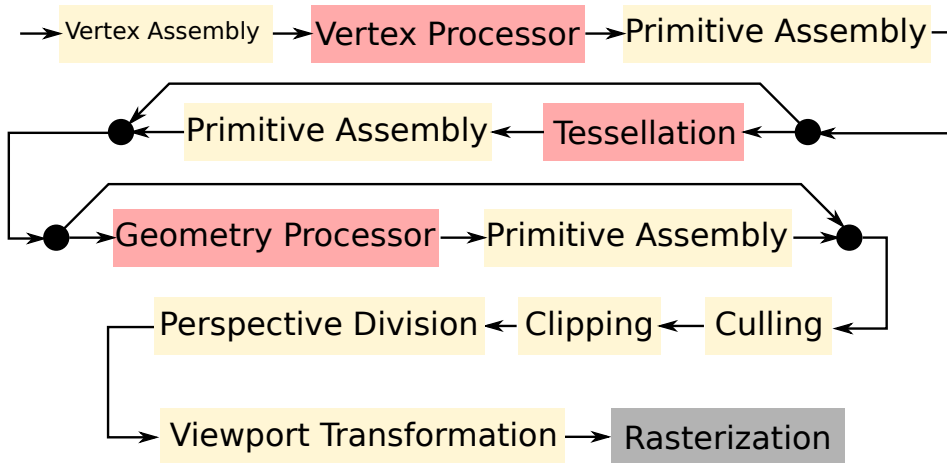
**fragments**



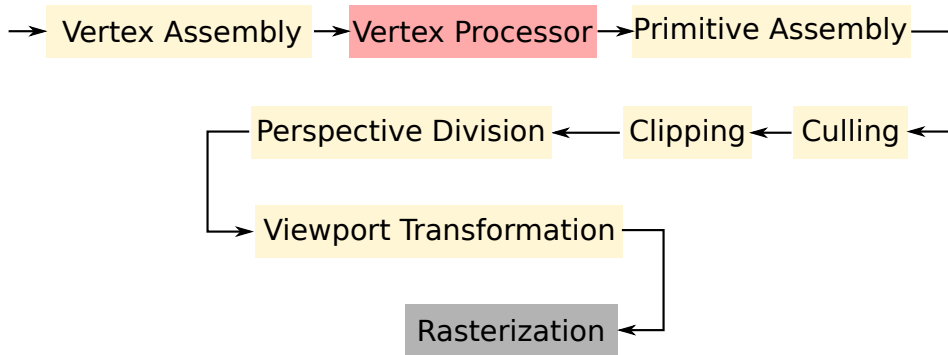
- The main goal of raster part is to process fragments.
- It colors fragment, performs per fragment operations, depth test, stencil test, blending, ...
- Hlavním úkolem rastrové části je výpočet fragmentů.
- Obarvuje fragment, provádí per fragment operace, hloubkový test, stencil test, blending, ...



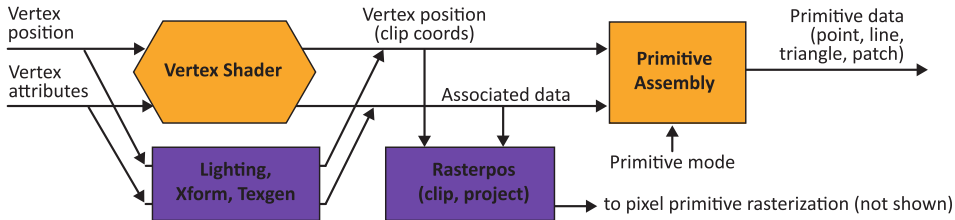
- Vector Part is composed of many blocks.
- Some of the blocks are programmable, some can be skipped.
- Vektorová část řetězce je složena z mnoha bloků.
- Některé bloky jsou programovatelné a některé vynechatelné.



- If we remove optional blocks, the pipeline looks like this.
- Pokud vynecháme volitelné bloky, zůstane zjednodušená vektorová část řetězce.

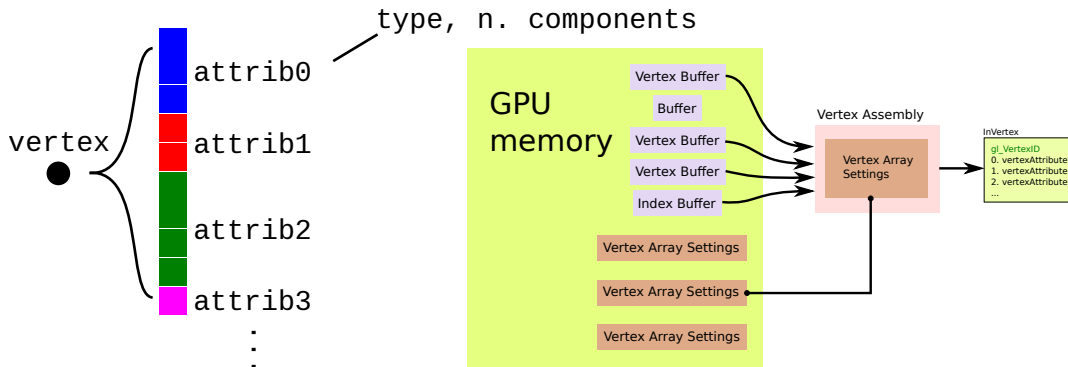


- Vertex Assembly creates vertices.
- Vertex processor transforms vertices.
- Primitive assembly creates base primitives from transformed vertices.
- Vertex Assembly sestavuje vrcholy
- Vertex processor transformuje vrcholy
- Primitive assembly sestavuje základní primitiva.



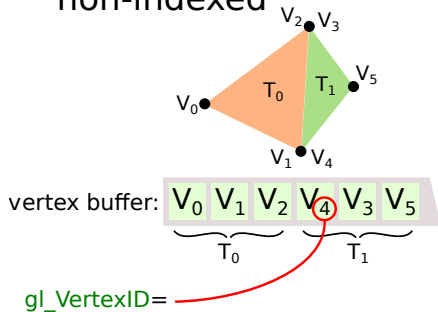


- Vertex is a structure of vertex attributes.
- Vertex Assembly unit reads data from memory and forms vertices.
- Vertex je struktura vertex atributů
- Vertex Assembly jednotka se stará o sestavování vrcholů z bufferů.

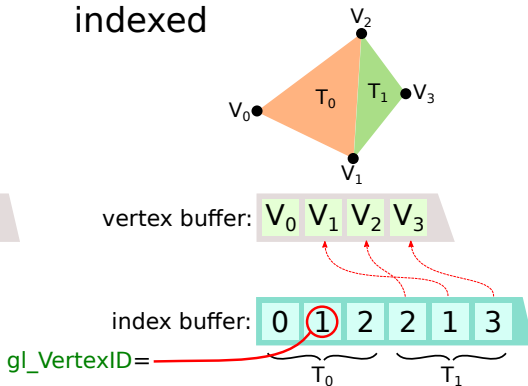


- Vertex Assembly can utilize indexing
- Vertex Cache - already processed vertices do not have to be processed again
- Vertex Assembly může využít indexování
- Vertex Cache - jednou propočítané vrcholy není třeba počítat znovu

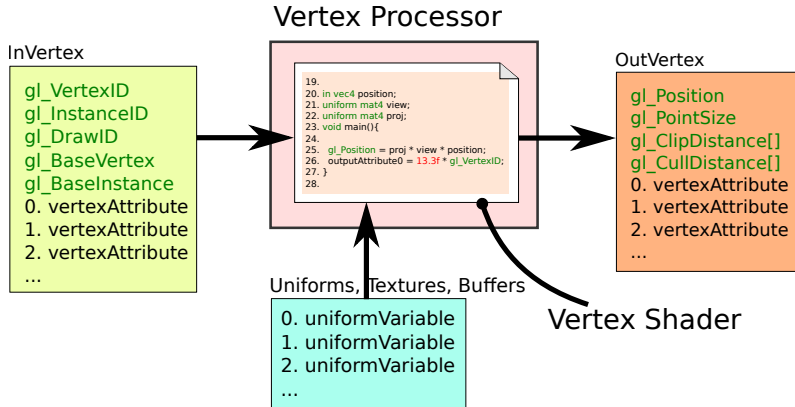
non-indexed



indexed

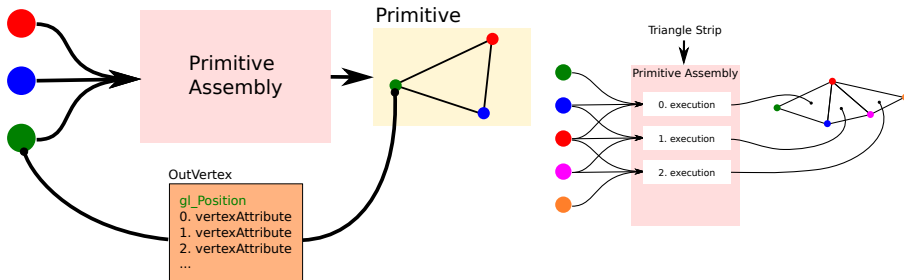


- Vertex Processor executes vertex shader.
- Vertex Shader is user program.
- The goal is to transform input vertex structure to output vertex structure.
- Ve Vertex Processoru běží vertex shader.
- Vertex Shader je uživatelem specifikovaný program.
- Cílem je transformovat vstupní vrchol na výstupní vrchol.

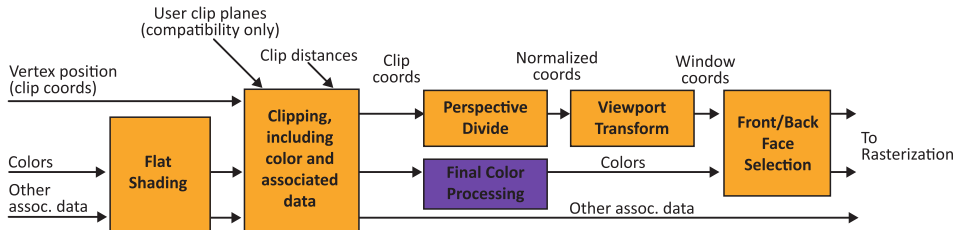


- Primitive Assembly unit creates base primitives.
- Output of the unit is one of the three base primitives (triangles, lines, points).
- The unit is guided by the primitive type (Triangles, Triangle Strip, Triangle Fan, Line Strip, Adjacency, Patch, ...)
- Primitive Assembly jednotka sestavuje primitiva.
- Cílem je podle nastavení sestavovat trojúhelníky, úsečky, vrcholy.
- Základní primitiva (trojúhelník, úsečka) mohou být součástí složitějších primitive.
- Triangle Strip, Triangle Fan, Line Strip, Triangle Adjacency, Patch, ...

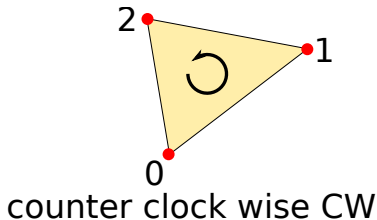
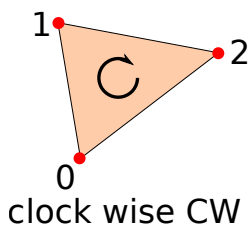
## Out Vertices



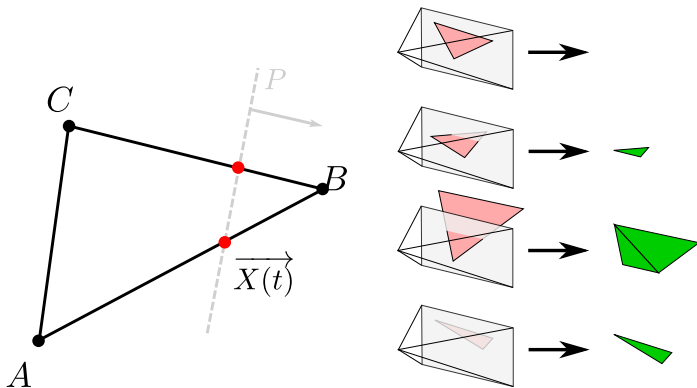
- Mezi rasterizací a sestavením primitiv se provádí ořez, zahazování odvrácených trojúhelníků, perspektivní dělení a viewport transformace.
- Clipping ořezává primitiva, která jsou jen částečně v pohledovém jehlanu.
- Perspektivní dělení zmenšuje objekty, které jsou dál od kamery.
- Viewport transformace převádí objekty na rozměr obrazovky.



- Culling removes triangles that are backfacing or front facing the camera.
- The state of triangle orientation is decided by vertex winding.
- It is possible to discard backfacing or frontfacing triangles.
- It is also possible to specify what is backfacing side (clock wise/counter clock wise vertices).
- Just for 2 manifold objects or scenes with restricted camera movement.
- Performance can be doubled with Culling.
- Culling se stará o zahazování odvrácených trojúhelníků.
- Odvrácenost je rozhodnuta na základě pořadí vrcholů.
- Je možné nastavit, jestli je trojúhelník přivrácený nebo odvrácený, pokud jsou vrcholy po nebo proti směru hodinových ručiček.
- Výkon vykreslování může být zdvojnásoben díky Cullingu.



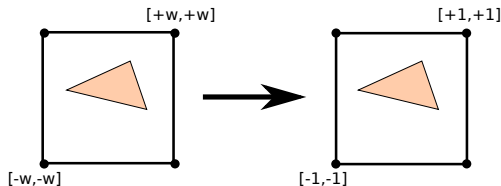
- If a primitive is only partially inside view-frustum, it needs to be clipped.
- Only near-plane clipping is necessary.
- If the result is quadrangle, it can be replaced with two sub-triangles.
- Pokud je primitivum jen částečně uvnitř pohledového jehlanu, je potřeba jej oříznout.
- Obvykle stačí oříznout pomocí blízké ořezové roviny (near-plane).
- Pokud vznikne čtyřúhelník, je možné jej nahradit dvěma trojúhelníky.



- Perspective Division unit converts homogeneous coordinates to cartesian coordinates.
- It divides  $x, y, z$  using  $w$ . The  $w$  contains distance from camera.
- After division,  $x, y, z \in [-1, +1]$  which is called normalized device coordinates (NDC).
- Division is expensive operation, specialized HW performs it faster.
- Blok perspektivního dělení převádí homogenní souřadnice na kartézské.
- Dělí se pomocí  $W$ , ve kterém je uložena hloubka. Tím se zmenšují objekty, které jsou dál od kamera.
- Po perspektivním dělení všechny vrcholy leží v rozsahu  $[-1, +1]$  - normalized device coordinates.
- Dělení je drahá operace, specializovaný HW ji provede rychleji.

Clip-Space

NDC



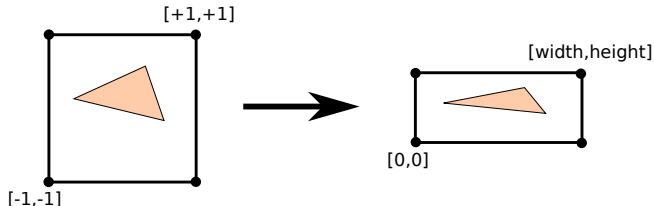
$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \rightarrow \begin{bmatrix} x/w \\ y/w \\ z/w \end{bmatrix}$$



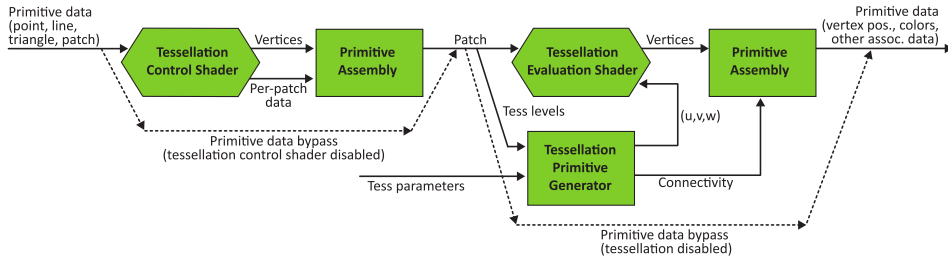
- Viewport transformation converts NDC to screen resolution.
- Blok viewport transformace transformuje normalizované souřadnice na rozlišení plátna.

NDC

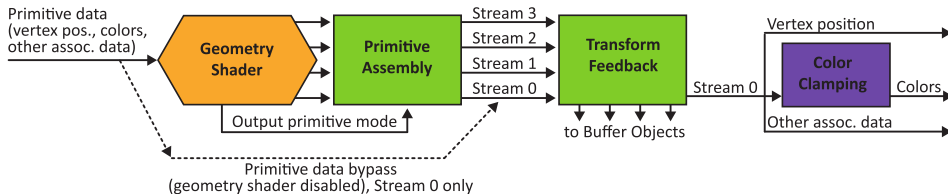
Screen-Space



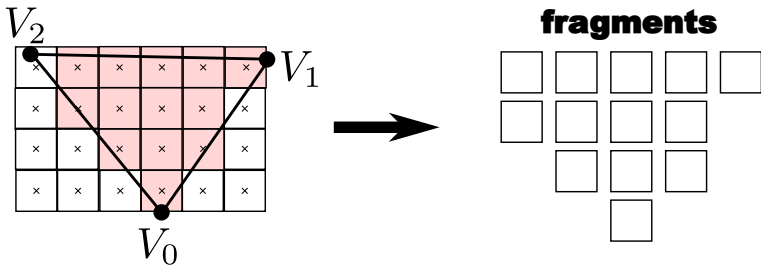
- Tessellation is composed of 2 programable parts and hardware primitive generator.
- The goal of tessellation is to add geometric details.
- Tesselace je složená zde 2 programovatelných processorů a hardwarového generátoru primitiv.
- Cíl teselace je přidání geometrických detailů.



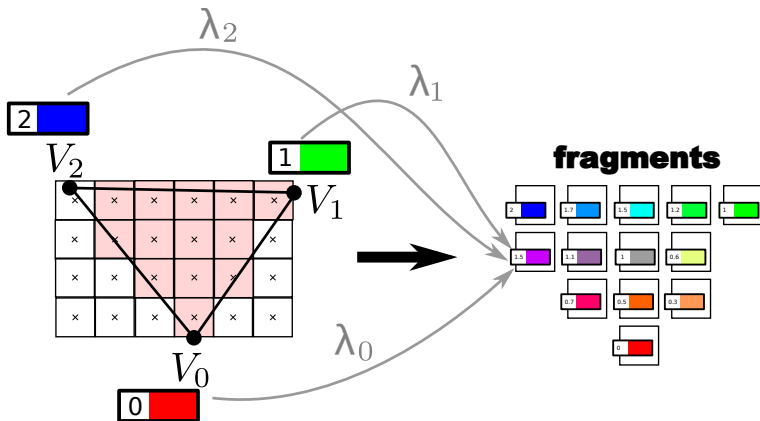
- Geometry processor inputs and outputs are primitives.
- The goal is to change, generate primitives.
- The stream of primitives can be sent to buffer.
- Geometry processor transformuje primitiva.
- Transform feedback může primitiva přeposlat zpět do bufferu.

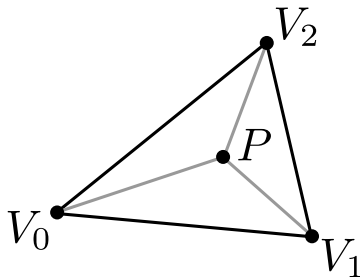


- Rasterization produces fragments.
- Fragment is data structure that is created in the location of sampling point.
- Sampling point is usually in the center of pixel.
- The situation is more complicated with the use of multisampling.
- If a sampling point is located inside a triangle, a fragment is created for corresponding pixel.
- Rasterizace produkuje fragmenty.
- Fragment je datová struktura, která vznikne na pozici vzorku (sampling point).
- Vzorkovací bod je obvykle uprostřed pixelu.
- Situace je komplikovanější při využití multi-samplingu.
- Pokud leží vzorkovací bod uvnitř trojúhelníku, vznikne fragment pro daný pixel.



- Out Vertices are composed of n vertex attributes.
- Rasterization produces fragments – data structure – which is composed of fragment attributes with the same user specific fragment attributes.
- Barycentric interpolation is used to transform three vertex data structures to many fragment data structures.
- 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 fragmentů pomocí interpolace



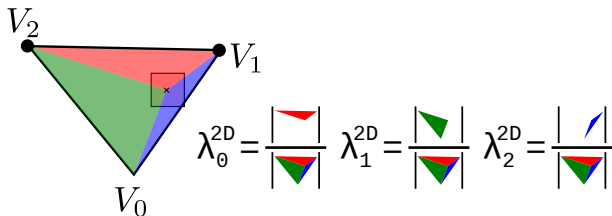


$$P = V_0 \cdot w_0 + V_1 \cdot w_1 + V_2 \cdot w_2$$

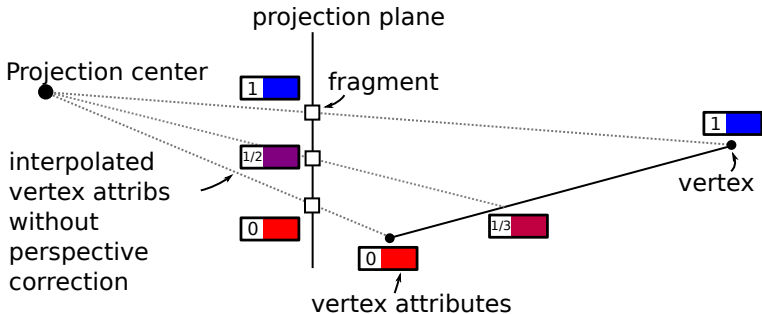
$$w_0, w_1, w_2 \in \langle 0, 1 \rangle$$

$$w_0 + w_1 + w_2 = 1$$

- 2D barycentrics are computed as ratio between the area of the triangle and the areas of its sub-triangles.
- Barycentrické koordináty ve 2D jsou spočítány jako poměr obsahů.

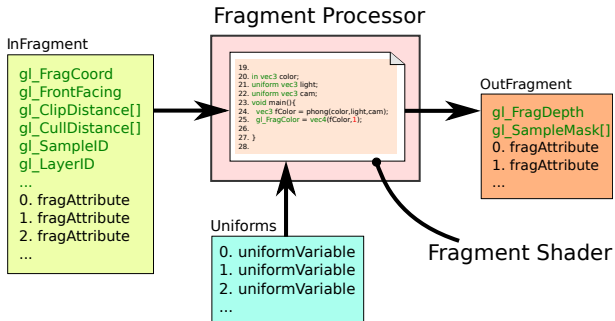


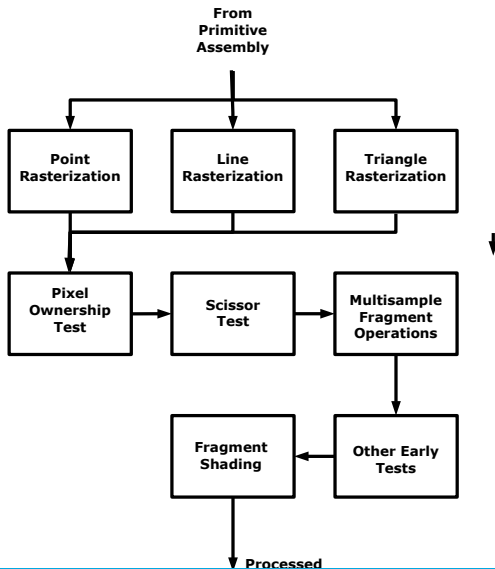
- Vertex attributes can be interpolated in the projection plane or in the 3D space.
- Interpolation in the 3D space is correct.
- In order to convert 2D interpolation to 3D interpolation, the perspective correction is required (OpenGL supports this natively / can be turned off)
- Vertex atributy se mohou 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)

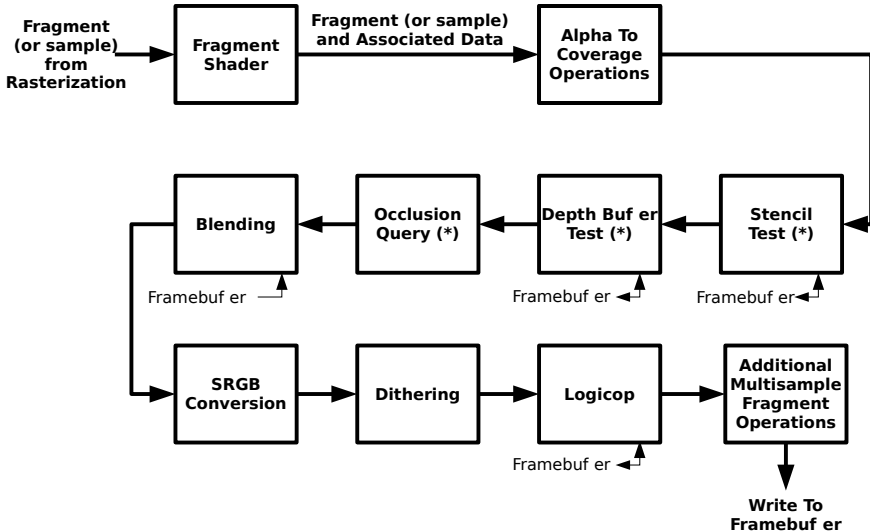


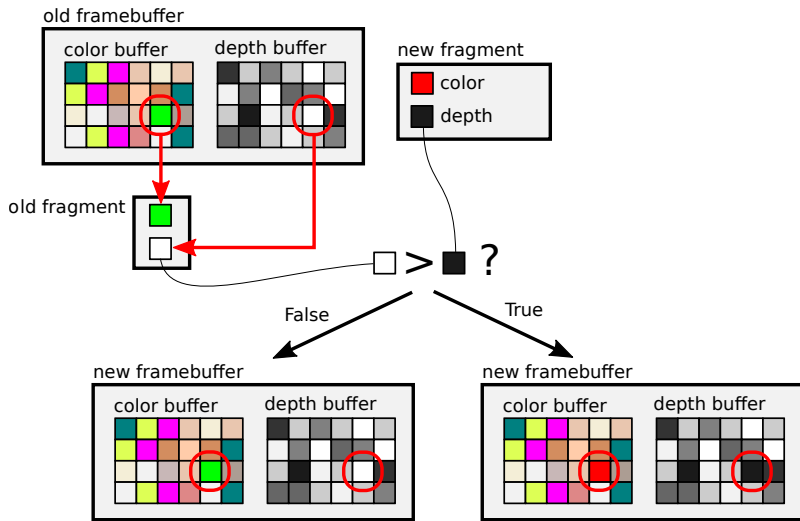


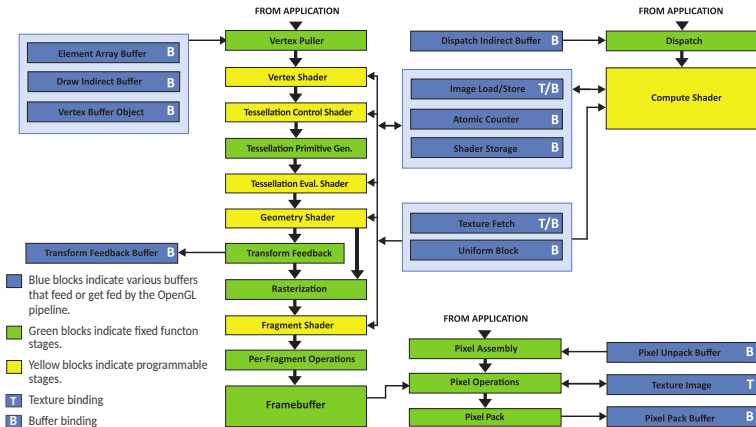
- Fragment shader is executed inside fragment processor.
- Fragment Shader is user program.
- The goal of the shader is to transform input fragment to output fragments.
- Multiple Render Target - support for rendering into multiple 2D arrays (renderbuffers / textures).
- Ve Fragment Processoru běží fragment shader.
- Fragment Shader je uživatelem specifikovaný program.
- Cílem je transformovat vstupní fragment na výstupní fragment.
- Multiple Render Target.





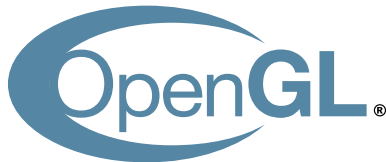






OpenGL

- OpenGL Open Graphics Language (Library)
- OpenGL is API for 3D graphics and general purpose computing (GPGPU).
- It's origin is IrisGL by SGI
- Multiplatform
- Can be used from almost any language .
- OpenGL shading language
- OpenGL Open Graphics Language (Library)
- OpenGL je API pro 3D grafiku a obecné výpočty na GPU.
- Vychází z IrisGL od SGI
- Platformně nezávislé
- Použitelné skoro z každého jazyka
- Obsahuje vlastní jazyk GLSL pro programování GPU

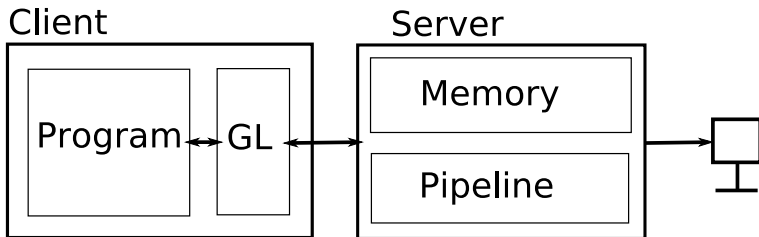


- Linux, Window, Mac Os X, Android,...
  - C, C++, Python, Java, Javascript, ...
  - OpenGL is backward compatible
  - OpenGL is low levels API (not as low as Vulkan)
  - OpenGL has simple API
  - OpenGL is fast (if used properly)
  - OpenGL is open industrial standard
  - WebGL
- 
- OpenGL je multiplatformní - 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 (Vulkan je nízko úrovnější)
  - OpenGL má jednoduché API
  - OpenGL je rychlé (když se správně použije)
  - OpenGL je otevřený industriální standard
  - WebGL



- OpenGL
  - 1.x - Fixed pipeline / fixní pipeline
  - **2.x** - Programable pipeline / programovatelná pipeline
  - 3.x - geometry shader, **Deprecation**
  - 4.x - Hardware tessellation, double precision / Hardwarová tessellace, dvojitá přesnost
  - 4.3 - Compute shaders / Compute shadery
  - 4.5 - Direct State Access
  - 4.6 - SPIRv
- OpenGL ES
  - Embedded systems, mobiles, tables / Vestavěné systémy, mobility, tablety
  - 1.x - Fixed pipeline / fixní pipeline
  - **2.x** - programable pipeline / programovatelná pipeline
  - 3.x - Occlusion queries, 3D textury, transform feedback
- WebGL
  - OpenGL in internet browser / OpenGL ve webovém prohlížeči
  - **Very similar to OpenGL ES / Velmi podobné OpenGL ES**

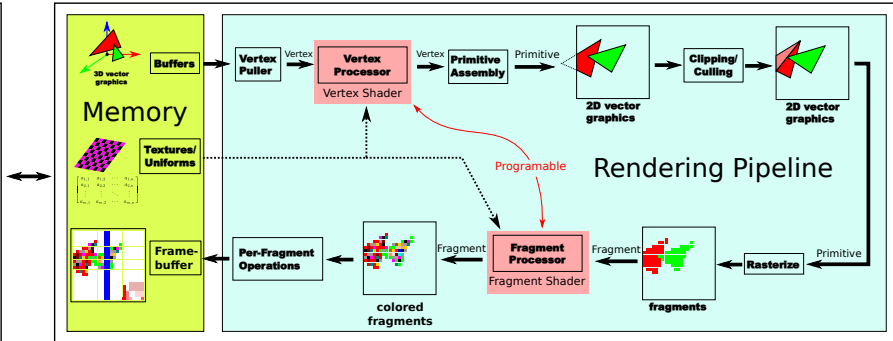
- OpenGL architecture is client-server
- Application runs on CPU and calls OpenGL function to access GPU
- Command Queue is hidden
- OpenGL je architektura klient server
- Aplikace běží na CPU a využívá OpenGL pro přístup k GPU
- Front příkazů mezi CPU a GPU je skryta



## CPU

1. upload data to buffers
2. prepare shaders
3. setup vertex puller
4. setup uniforms
5. execute draw call

## GPU



- Simple interface / Jednoduché rozhraní
  - Only C functions / Pouze C funkce
  - Data in form of numbers and arrays / Data jsou jen čísla a pole
  - No structures, classes / Žádné struct, class
- State machine / Stavový stroj
  - Most of the commands set the state of pipeline / Většina příkazů nastavuje stav pipeline
  - State cannot change on its own / Stav se sám nemění
- OpenGL (Rendering) Context
  - Main object of OpenGL / Hlavní objekt OpenGL
  - Must be created outside of OpenGL / Mimo OpenGL (WGL/GLX)
  - Encapsulates data, state, connection to the output / Zapouzdřuje data, stav, napojení na výstup

glNameNT(...)

- N - number of parameters / počet parametrů
- T - type of parameters / 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	pointer *		

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

OpenGL commands can be divided into several groups.

- **Commands for OpenGL object management (10 main OpenGL objects)**
- **Execution commands (draw commands, compute commands)**
- **State commands (set OpenGL state)**
- Debug commands
- Framebuffer commands
- Synchronization commands (glFinish)
- Util commands

OpenGL příkazy lze rozdělit do několika skupin

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

```
GLvoid glCreateObjects(GLsizei n, GLuint * objects);  
GLvoid glDeleteObjects(GLsizei n, const GLuint * objects);
```

- Object name - GLuint, all OpenGL object instances are represented with integer
- 0 is reserved for empty object
- Jméno objektu - GLuint, všechny objekty jsou v API reprezentovány integerem
- 0 rezervována pro prázdný objekt

Objects / Objekty:

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

For drawing, it is necessary to initialize several OpenGL objects (Shaders, Shader Programs, Buffers, Vertex Arrays), set OpenGL state and execute draw command.

- 1 Compile and link shader programs from string
- 2 Allocate buffers and copy data from CPU
- 3 Setup vertex array object to tell OpenGL about the layout of the data in buffers
- 4 Setup OpenGL state
- 5 Execute draw call

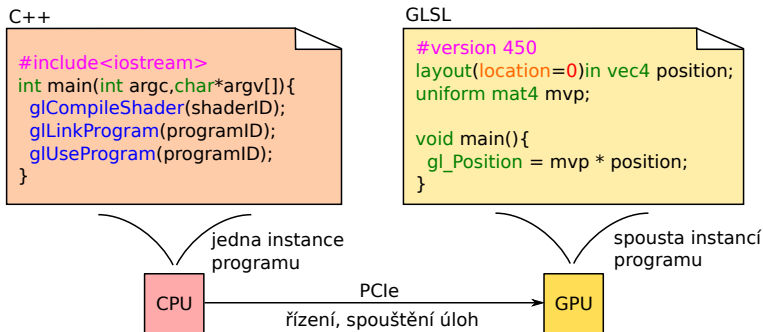
Pro kreslení je potřeba inicializovat několik OpenGL objektů (Shadery, Shader Programy, Buffery, Vertex Array objekty), nastavit OpenGL stav a zavolat kreslicí příkazy.

- 1 Zkompilování a slikování shader programů z řetězce
- 2 Alokace bufferu a nakopírování dat z CPU
- 3 Nastavení vertex array tak, aby popisoval formát dat v bufferech
- 4 Nastavení OpenGL stavu
- 5 Spuštění kreslicího příkazu



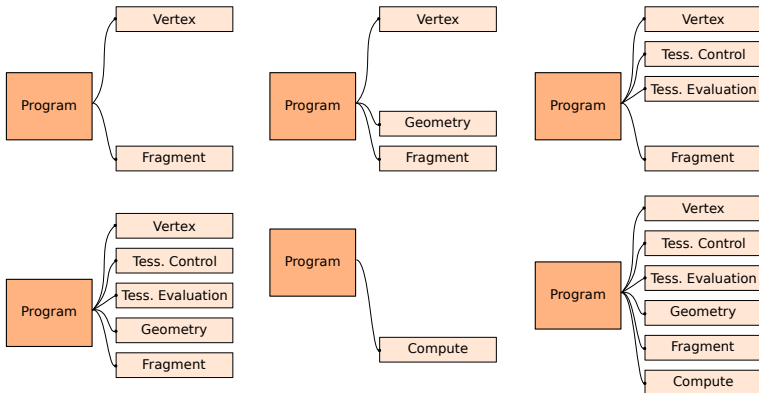
Příprava programů a shaderů pro GPU

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



- 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, tessellation control, tessellation 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:



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

```
#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



```
#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);
}
```

```
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 vseh 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, který
//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 instance
    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(){
    //koordinaty fragmentu ve viewportu
    in vec4 gl_FragCoord;

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

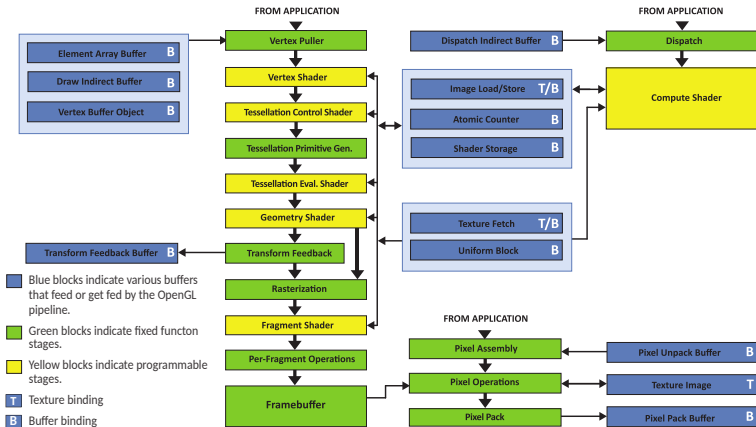
    //pro modifikaci hloubky fragmentu
    //pri zapisu vypina brzky depth test
    out float gl_FragDepth;

    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`





Vytvoření bufferu.

```
float data[]={1,2};//data, která budeme vkladat do bufferu  
GLuint vbo;//identifikator VBO  
glCreateBuffers(1,&vbo);  
//alokujeme buffer a nahrajeme do něj data  
glNamedBufferData(vbo, sizeof(data), data, GL_STATIC_DRAW);
```

Změna dat ve VBO.

```
float*ptr;//ukazatel na data  
ptr=(float*)glMapNamedBuffer(vbo, GL_READ_WRITE);//namapujeme buffer  
ptr[0]=0.5;//nastavíme hodnotu prvního prvku  
glUnmapNamedBuffer(vbo);//odmapujeme buffer, komitujeme změny do GPU
```

Nebo pomocí `glNamedBufferSubData`.

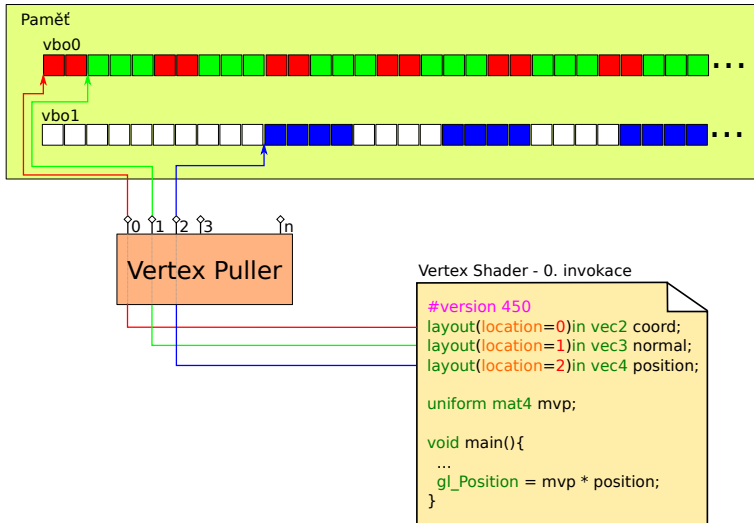
```
glNamedBufferSubData(vbo,  
    sizeof(float),//nahrajeme nová s offsetem jeden float  
    sizeof(float),//nahrajeme jen jeden float  
    data);//data
```

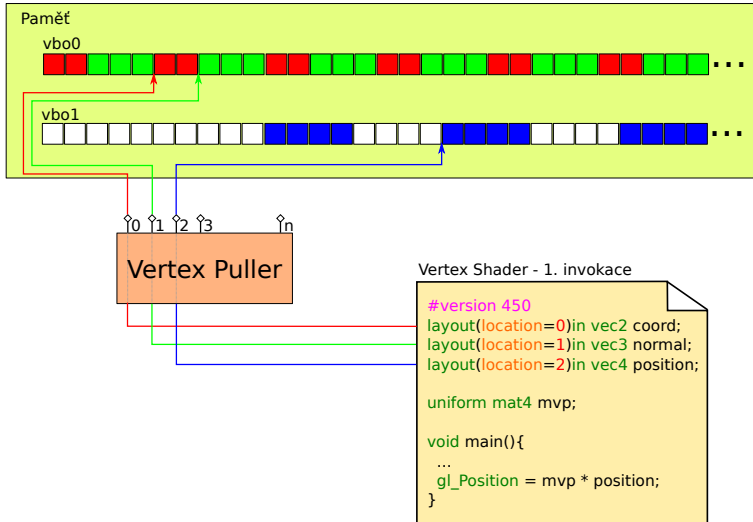
Nabindování bufferu.

```
glBindBuffer(GL_ARRAY_BUFFER, vbo);
```

Konfigurace Vertex Pulleru/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





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

glVertexArrayAttribBinding(vao,0,0);
glEnableVertexArrayAttrib(vao,0);
glVertexArrayAttribFormat(vao,
    0,//cislo vertex atributu
    2,//pocet polozek pro cteni (vec2)
    GL_FLOAT,//typ polozek
    GL_FALSE,//normalizace
    0);//relativni offset
glVertexArrayVertexBuffer(vao,0,
    vbo,
    sizeof(float)*5,//stride
    (GLvoid*)(sizeof(float)*0));//offset

glVertexArrayAttribBinding(vao,1,1);
glEnableVertexArrayAttrib(vao,1);
glVertexArrayAttribFormat(vao,1,3,GL_FLOAT,GL_FALSE,0);
glVertexArrayVertexBuffer(vao,1,vbo,sizeof(float)*5,
    (GLvoid*)(sizeof(float)*2));

glVertexArrayAttribBinding(vao,2,2);
glEnableVertexArrayAttrib(2);
glVertexArrayAttribFormat(2,4,GL_FLOAT,GL_FALSE,0);
glVertexArrayVertexBuffer(vao,2,vbo,sizeof(float)*8,
```

```
glVertexArrayAttribBinding(GLuint vao, GLuint attribIndex, GLuint bindingIndex);  
glVertexArrayAttribFormat(GLuint attribIndex, GLint size, GLenum type, GLboolean normalized, GLuint relativeoffset);
```

Vertex Shader

```
layout(location = 0) vec4 position;  
layout(location = 2) ivec2 gridId;  
layout(location = 7) float velocity;  
  
void main(){  
  ...  
}
```

Vertex Array Object

*BindingIndex<sub>0</sub>*  
*BindingIndex<sub>1</sub>*  
⋮  
*BindingIndex<sub>n</sub>*  
*ElementBuffer*

States

Buffer, Offset, Stride  
Buffer, Offset, Stride  
⋮  
Buffer, Offset, Stride

```
glBindVertexBuffer(GLuint bindingIndex, GLuint buffer, GLintptr offset, GLsizei stride);  
...
```

Kreslení a 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

- 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");
}
```

```
#version 450

layout(location=0) out vec4 fColor;
uniform vec3 color; //uniformni promenna
void main() {
    fColor = vec4(color, 1);
}
```

- 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
    glProgramUniform3fv(program, colorUniform, 1, 0, 0);

    //aktivovani vao
    glBindVertexArray(vao);

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

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

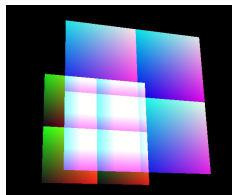
Per fragment operace

- 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 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 fragmentu
- Brzký depth test může značně urychlit kreslení

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

- 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, ...)

- `glxext.h`
- `void* wglGetProcAddress(const char* name);`
- `void* glXGetProcAddress(const char* name);`
- `PFNGLNECOPROC 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 zapouzdřuje vškeré 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 různé profily

```
//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
```

- <http://www.opengl.org/sdk/docs/>
- <http://www.opengl.org/documentation/glsl/>
- <http://www.opengl.org/registry/>
- <http://www.khronos.org/opencl/>
- <https://wiki.libsdl.org/FrontPage>
- [http://developer.amd.com/wordpress/media/2013/07/AMD\\_Accelerated\\_Parallel\\_Processing\\_OpenCL\\_Programming\\_Guide-rev-2.7.pdf](http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf)
- <http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/Asynchronous-Shaders-White-Paper-FINAL.pdf>

Thank you for your attention! Questions?