

Základy OpenGL

© 2003-2019 Josef Pelikán
CGG MFF UK Praha

pepca@cgg.mff.cuni.cz
<https://cgg.mff.cuni.cz/~pepca/>



Pokroky v hardware

3D akcelerace běžná i v konzumním sektoru

Hry, multimedia i mobilní app. (OpenGL ES)

Vzhled – kvalita prezentace

- velmi důmyslné techniky texturování
- kombinace mnoha textur, modularita zpracování

Vysoký výkon

- nejmodernější čipové technologie pro výrobu GPU (NVIDIA Volta, Turing: 12 nm), **masivní paralelismus**
- velmi rychlé **paměti** (vícecestný přístup, GDDR6, HBM2)
- výjimečné sběrnice mezi GPU a CPU (dnes PCI-E)



Pokroky v software

Dvě hlavní knihovny pro 3D grafiku

- **OpenGL/Vulkan** (SGI, open standard) a **Direct3D** (Microsoft)
- přístup je podobný, API je velmi ovlivněno hardwarem

Nastavení parametrů a **úsporný přenos dat** do GPU

- maximální sdílení společných datových polí

Programování grafického řetězce!

- revoluce v programování 3D grafiky
- „*vertex shader*“ – zpracování vrcholů
- „*geometry/tessellation/mesh sh.*“ – generování dalších elementů
- „*fragment shader*“ („*pixel shader*“) – zpracování jednotlivých fragmentů/pixelů před vykreslením



Vývojové nástroje

Příjemné pro programátory i „kreativce“

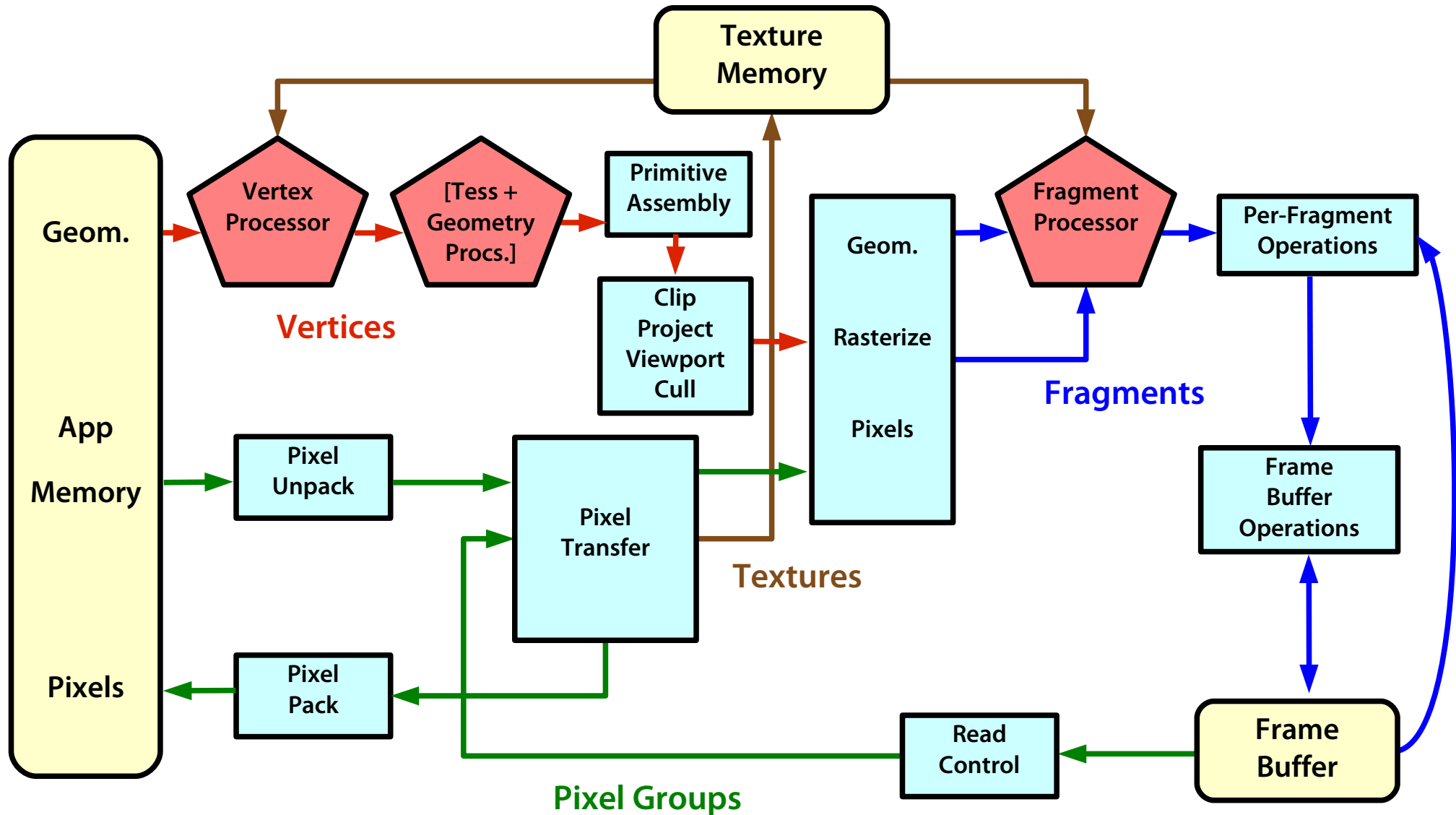
Vyšší jazyky pro programování GPU

- GLSL (OpenGL), HLSL (DirectX), ~~Cg (NVIDIA)~~
- ~~Cg~~ a HLSL jsou téměř shodné

Kompozice grafických efektů

- kompaktní popis celého efektu (GPU programy, odkazy na data) v jednom souboru
- DirectX **.FX** formát, NVIDIA **CgFX** formát
- nástroje: Effect Browser (Microsoft), **FX Composer** (NVIDIA), **RenderMonkey** (ATI)

Schéma OpenGL GPU





OpenGL – geometrická primitiva

Typy geometrických primitiv

- bod, úsečka, lomená čára, smyčka
- polygon, **trojúhelník**, proužek trojúhelníků, vějíř trojúhelníků, čtyřúhelník, proužek čtyřúhelníků...

Zpracování jednotlivých vrcholů

- glVertex, glColor, glNormal, glTexCoord, ...
- neefektivní (mnoho volání gl* funkcí)

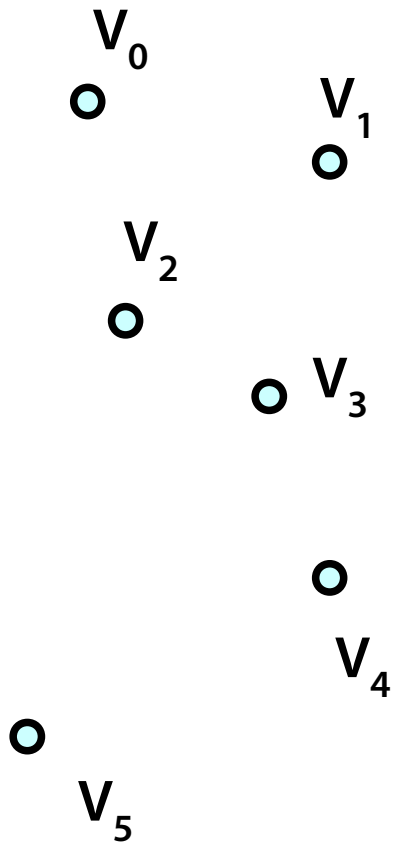
Pole vrcholů

- glDrawArrays, glMultiDrawArrays, glDrawElements ...
- glColorPointer, glVertexPointer... nebo **prokládání**

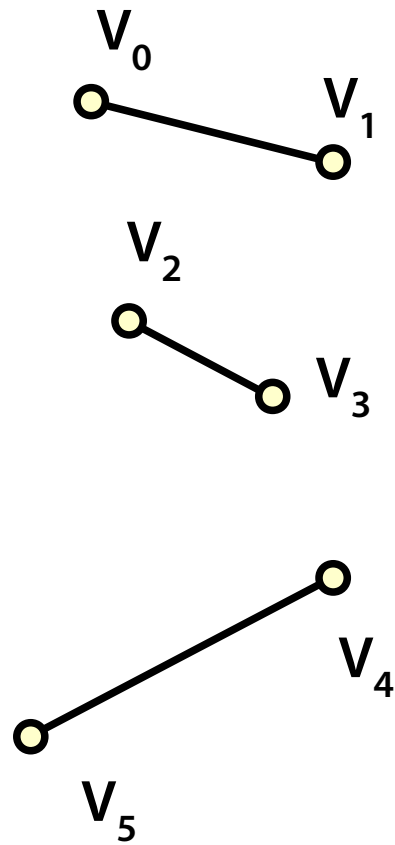


Geometrická primitiva I

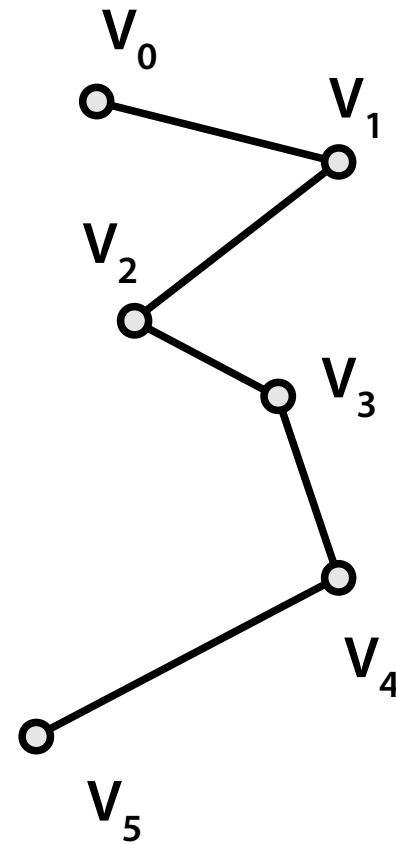
GL_POINTS



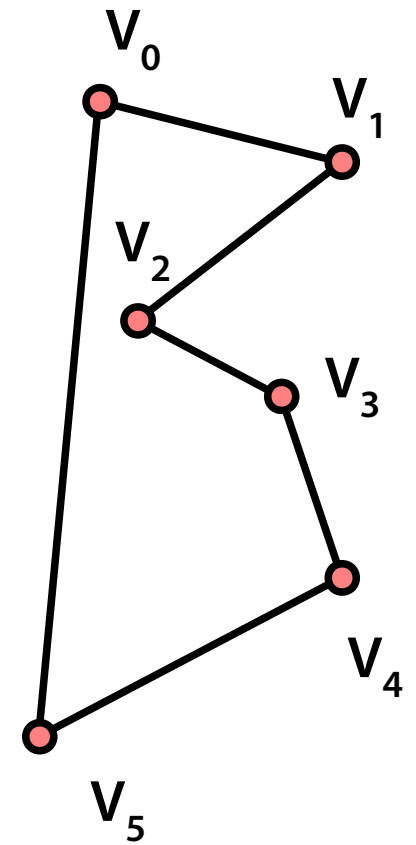
GL_LINES



GL_LINE_STRIP



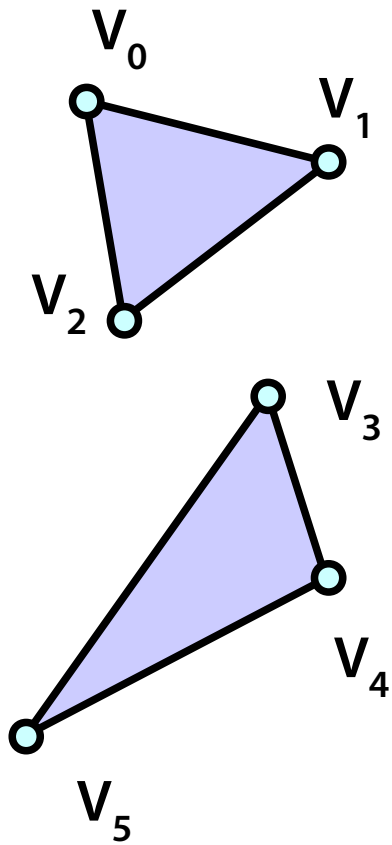
GL_LINE_LOOP



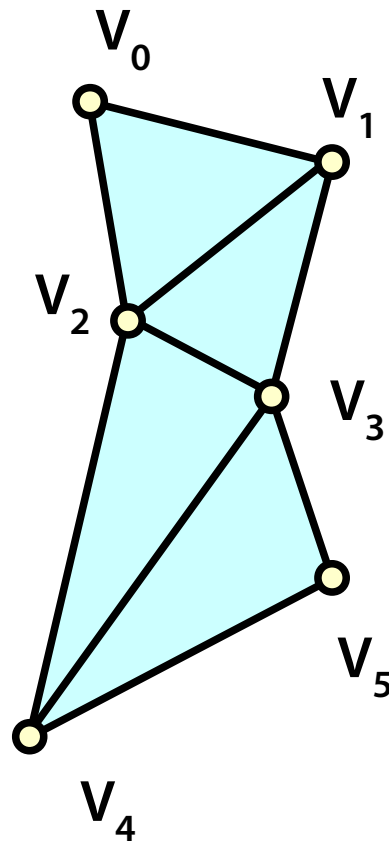


Geometrická primitiva II

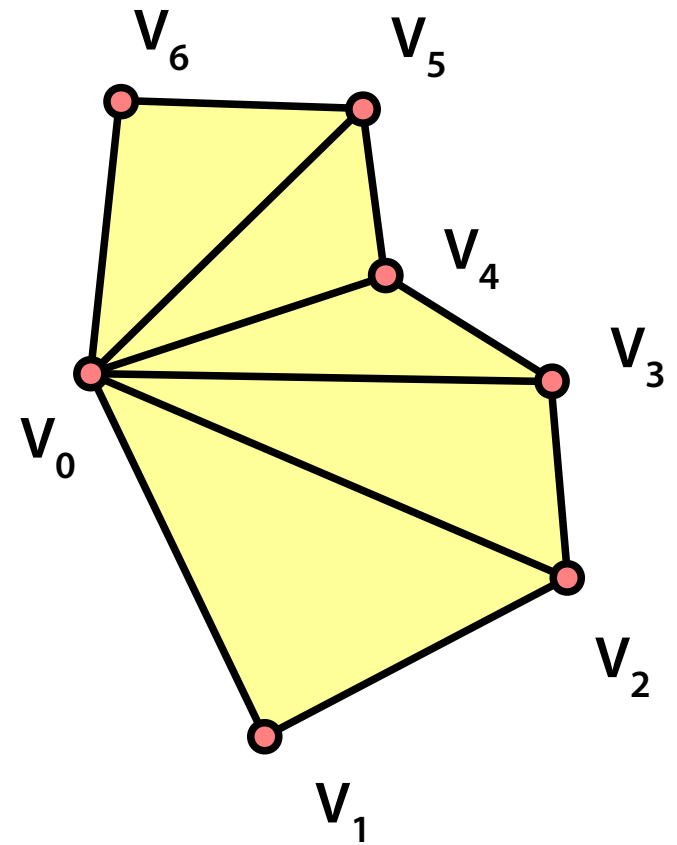
GL_TRIANGLES



GL_TRIANGLE_STRIP



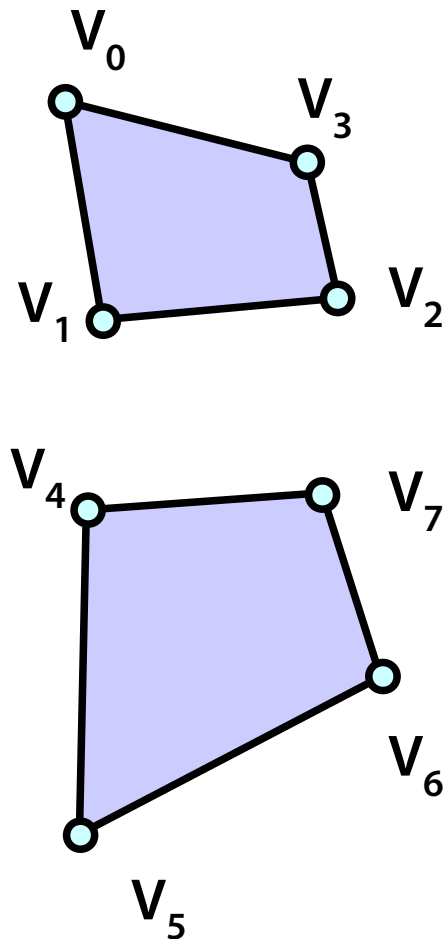
GL_TRIANGLE_FAN



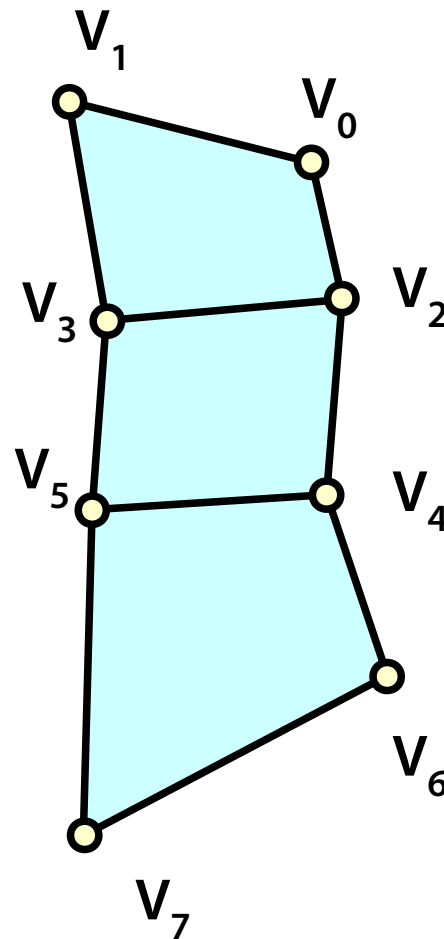


Geometrická primitiva III

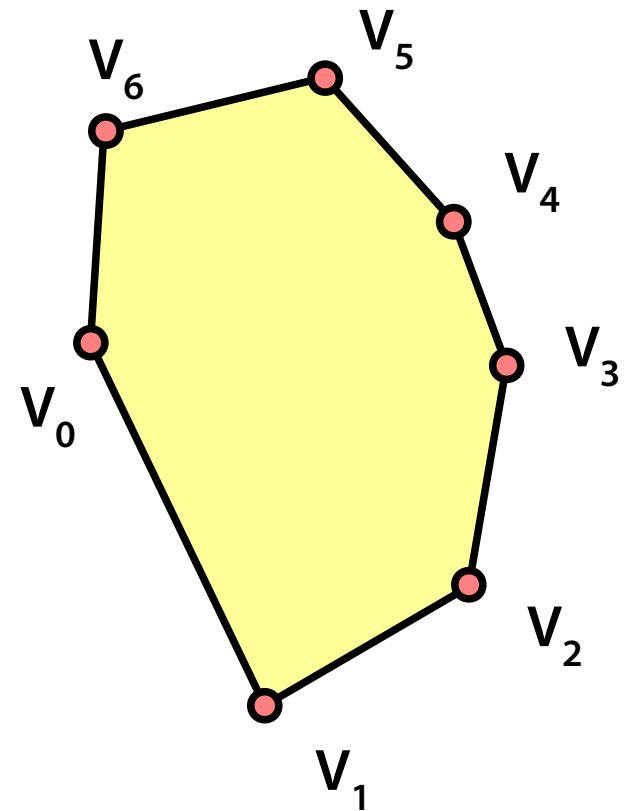
GL_QUADS



GL_QUAD_STRIP

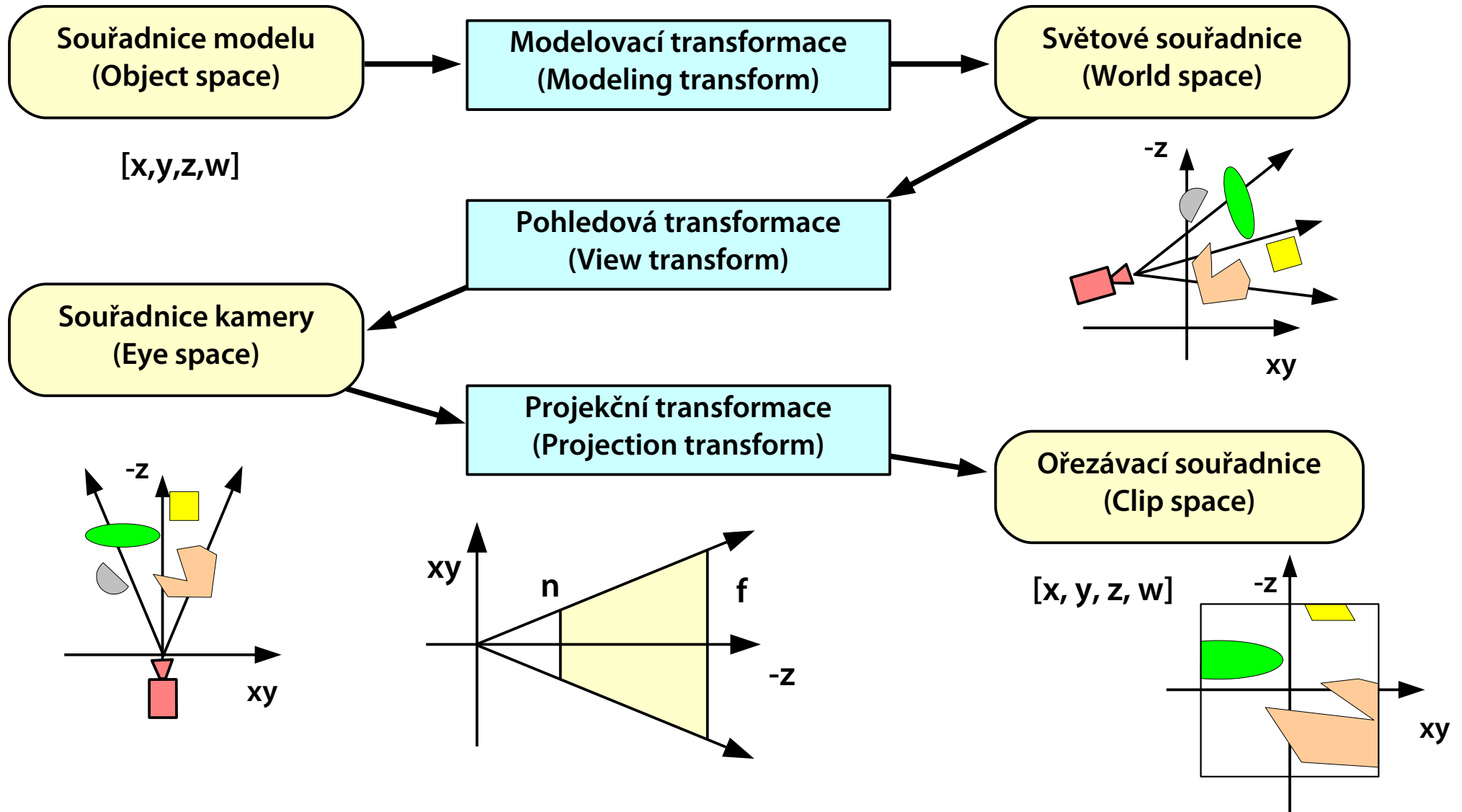


GL_POLYGON

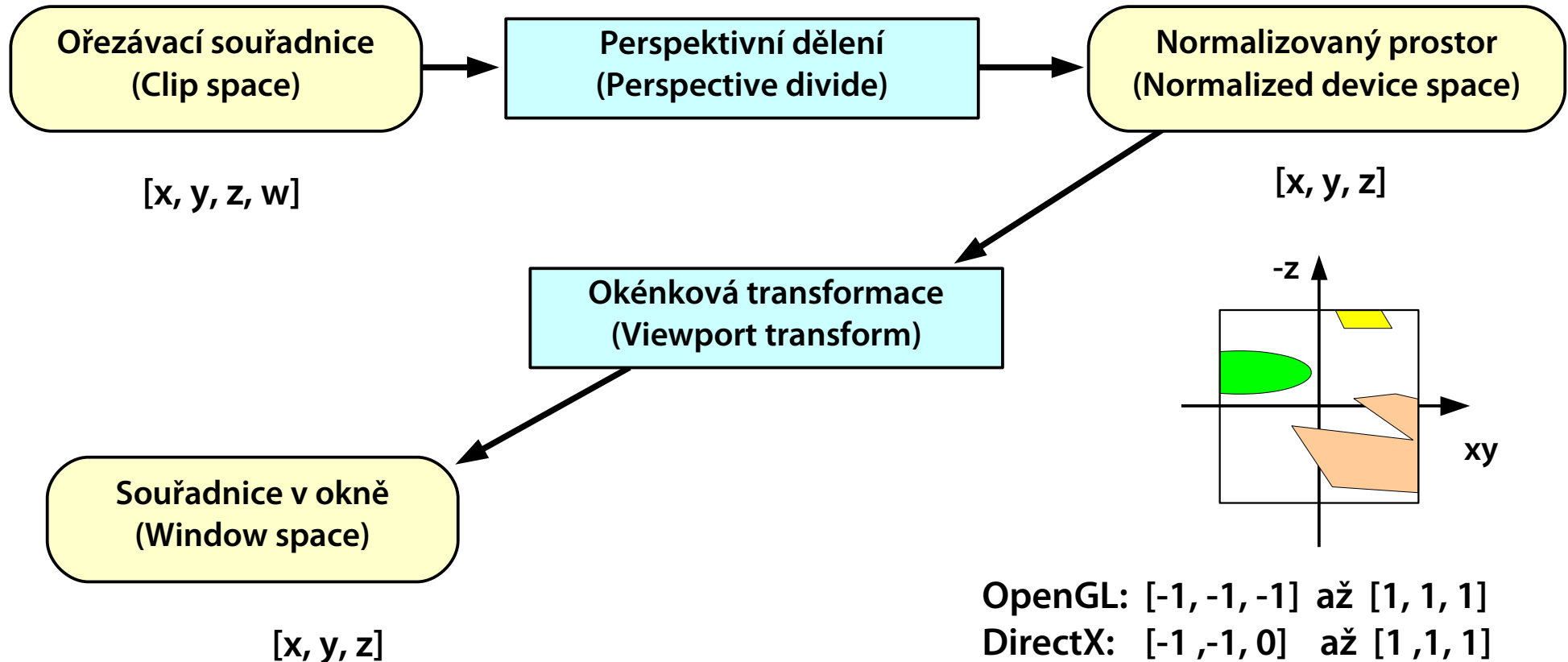




Souřadné soustavy



Souřadné soustavy II



$[x, y]$ skutečná velikost v pixelech na obrazovce (fragmenty)
 z hloubka kompatibilní s z-bufferem



Souřadné soustavy III

Souřadnice modelu

- databáze objektů, ze kterých se skládá scéna
- 3D modelovací programy (3DS Max, Blender, Maya...)

Světové souřadnice

- absolutní souřadnice virtuálního 3D světa
- vzájemná poloha jednotlivých **instancí objektů**

Souřadnice kamery

- 3D svět se transformuje do relativních souřadnic kamery
- střed projekce: **počátek**, směr pohledu: **-z** (nebo **z**)



Souřadné soustavy a transformace

Transformace model → kamera

- společná transformační matice „model-view“
- světové souřadnice nejsou moc důležité

Projekční transformace

- definuje zorný objem = „frustum“ [l, r, b, t, n, f]
- přední a zadní ořezávací vzdálenost: n, f
- výsledkem je homogenní souřadnice (před ořezáním)

Ořezávací souřadnice („clip space“)

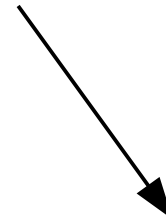
- výstupní souřadnice vertex shaderu!



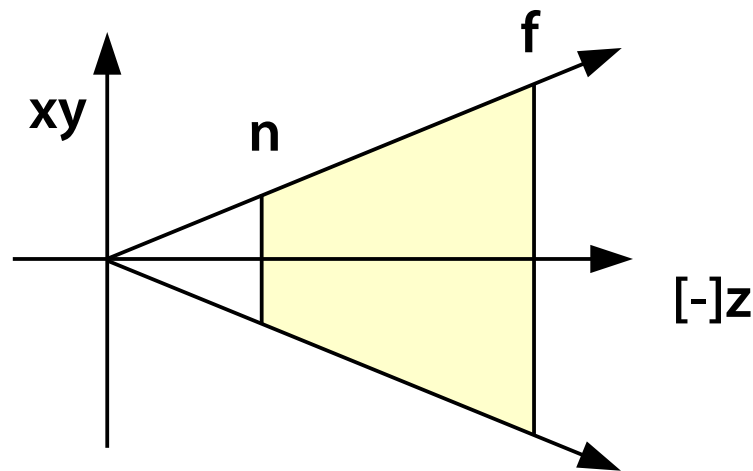
Projekční transformace (perspektiva)

Vzdálený bod f může být i v nekonečnu

$$\begin{bmatrix} \frac{2n}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2n}{t-b} & 0 & 0 \\ -\frac{r+l}{r-l} & -\frac{t+b}{t-b} & \frac{f+n}{f-n} & 1 \\ 0 & 0 & -\frac{2fn}{f-n} & 0 \end{bmatrix}$$



$$\begin{bmatrix} \frac{2n}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2n}{t-b} & 0 & 0 \\ -\frac{r+l}{r-l} & -\frac{t+b}{t-b} & 1 & 1 \\ 0 & 0 & -2n & 0 \end{bmatrix}$$





Souřadné soustavy a transformace

Perspektivní dělení

- pouze převádí **homogenní** souřadnice do **kartézských**

Normalizované souřadnice zařízení („NDC“)

- kvádr standardní velikosti
- OpenGL: $[-1, -1, -1]$ až $[1, 1, 1]$
- DirectX: $[-1, -1, 0]$ až $[1, 1, 1]$

Souřadnice v okně („window space“)

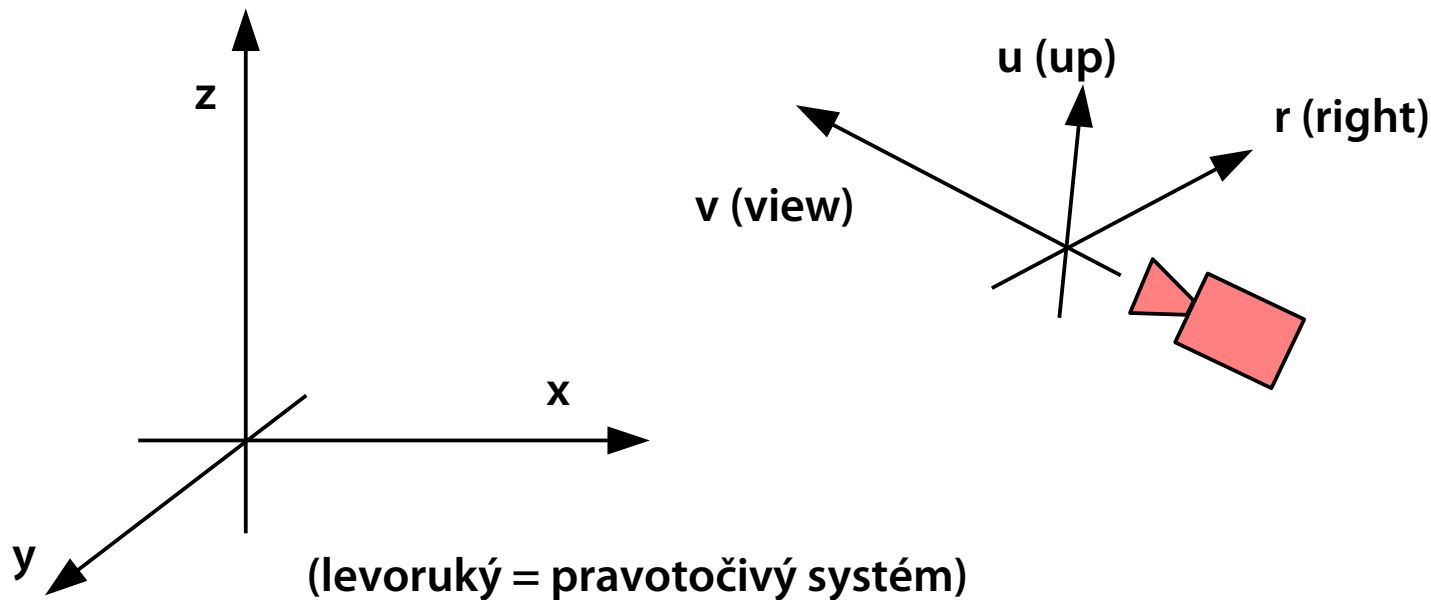
- výsledkem okénkové transformace a transformace hloubky
- používají se při **rasterizaci** a práci s **fragmenty**



Transformace tuhého tělesa

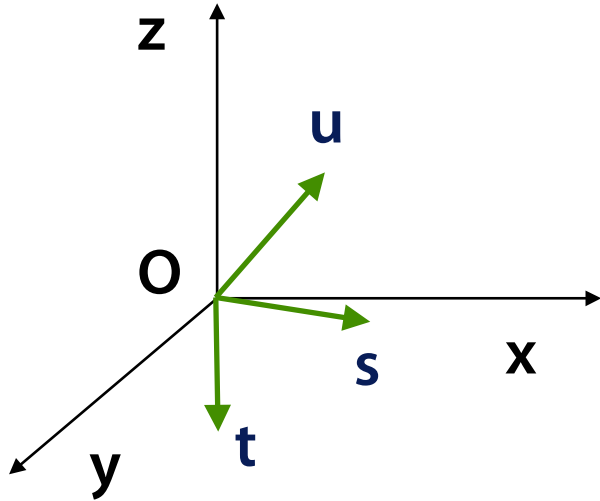
Zachovává **tvár těles**, mění pouze jejich umístění

- skládá se jenom z **posunutí** a **otočení**
- často se používá k **převodu mezi souřadnicovými systémy** (např. mezi světovými souřadnicemi a systémem spojeným s pozorovatelem)





Převod mezi dvěma orientacemi



Souřadný systém má počátek v **O**
a je zadán trojicí jednotkových
vektorů **[s, t, u]**

$$[1, 0, 0] \cdot M_{stu \rightarrow xyz} = s$$

$$[0, 1, 0] \cdot M_{stu \rightarrow xyz} = t$$

$$[0, 0, 1] \cdot M_{stu \rightarrow xyz} = u$$

$$M_{stu \rightarrow xyz} = \begin{bmatrix} s_x & s_y & s_z \\ t_x & t_y & t_z \\ u_x & u_y & u_z \end{bmatrix}$$

$$M_{xyz \rightarrow stu} = M_{stu \rightarrow xyz}^T$$



Geometrická data na GPU

VBO, VAO, počátky od OpenGL 1.5 (2003)

- pro .NET nutností (klientská paměť není fixována)

Buffer na straně GPU (grafického serveru) obsahující geometrická data

- založení bufferu: `glBindBuffer`
- zadání dat z pole: `glBufferData`, `glBufferSubData`
- mapování do paměti aplikace: `glMapBuffer`, `glUnmap...`

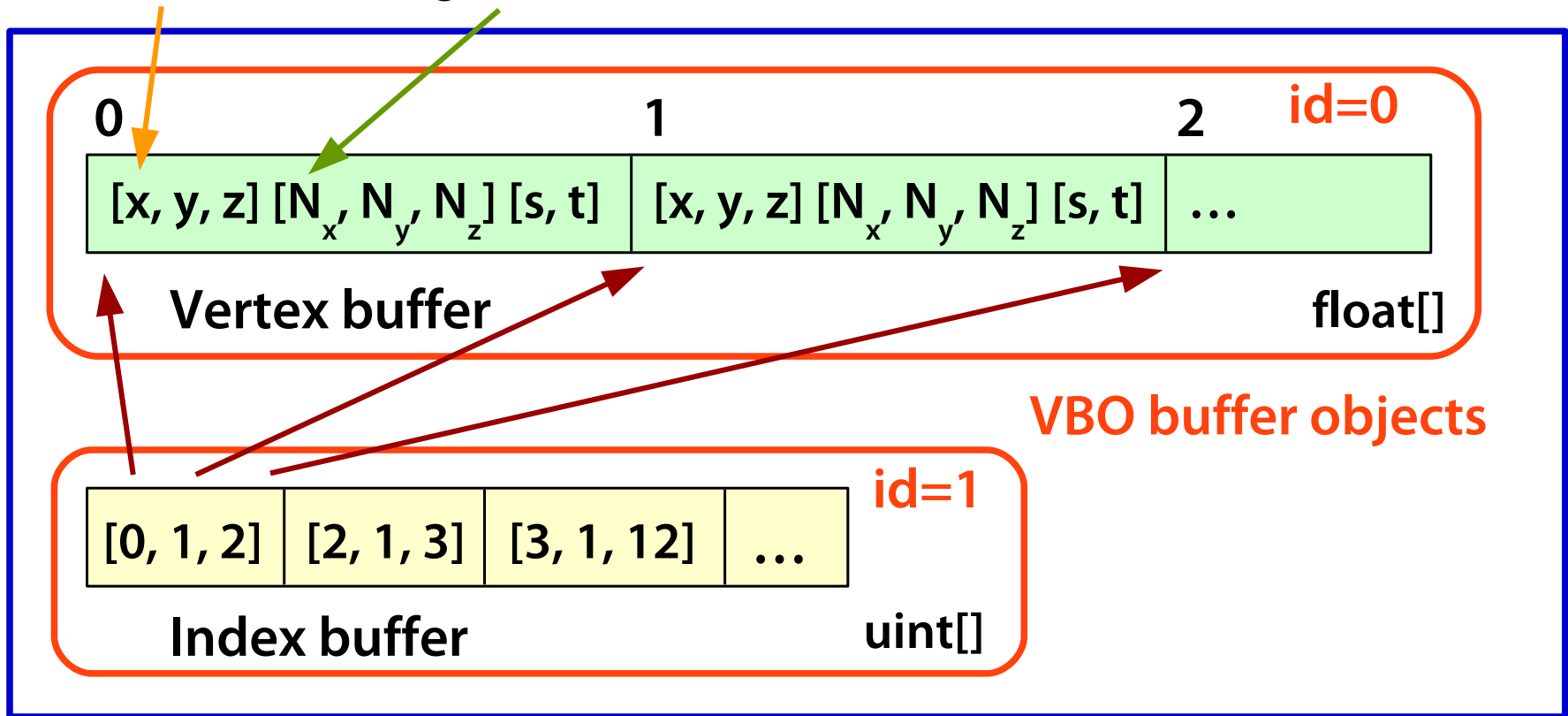
Práce s klientskou pamětí nebo s bufferem

- `glColorPointer`, `glNormalPointer`, `glVertexPointer...`



Vertex Buffer Objects (Buffers)

```
glBindBuffer(GL_ARRAY_BUFFER, 0);  
glVertexPointer(...); glNormalPointer(...); ...
```



```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 1);  
glDrawElements(GL_TRIANGLES, ...);
```

GPU memory



Zpracování vrcholu (shaders)

Povinný „Vertex shader“

Transformace vrcholů modelovacími a projekčními maticemi

- `glMatrixMode`
- `glLoadIdentity`, `glLoadMatrix`, [`glMultMatrix`]
- [`glRotate`, `glScale`, `glTranslate...`]



Sestavení a zpracování primitiv

Volitelné „Tesselation“ a/nebo „Geometry shader“

Sestavení (Assembly)

- určení, kolik vrcholů primitivum potřebuje
- shromáždění balíčku dat a odeslání dál

Zpracování primitiv

- ořezávání („clipping“)
- projekce do zorného objemu („frustum“) – dělení „w“
- projekce a ořezání do 2D okénka („viewport“)
- odstranění odvrácených stěn („culling“)
 - » jednostranná vs. oboustranná primitiva



Rasterizace, fragmenty

Rasterizace = vykreslení vektorových primitiv

- rozklad geometrických objektů na **fragmenty**
- geometrické objekty: body, úsečky, trojúhelníky, bitmapy

Fragment

- **rastrový element**, který **potenciálně** přispívá k barvě nějakého px.
- velikost: stejná nebo menší než u pixelu (anti-aliasing)
- „balíček dat“ procházející rasterizační jednotkou GPU
 - » vstup/výstup: **x, y, z** (pouze hloubku lze měnit!)
 - » texturovací souřadnice **t_0** až **t_n**
 - » lesklá a difusní barva, koeficient mlhy, uživatelská data...
 - » výstupní barva **RGB** a neprůhlednost **α** (frame-buffer op.)



Interpolace ve fragmentech

Atributy fragmentů se automaticky **interpolují z hodnot ve vrcholech**

- hloubka (**z** nebo **w**)
- texturové souřadnice
- barvy (lesklá a difusní složka)
- uživatelské atributy...

Rychlé HW interpolátory

Perspektivně korektní interpolace

- jen **[x, y]** se mění lineárně
- ostatní veličiny vyžadují jedno dělení na každý fragment



Zpracování fragmentů (shader)

Povinný „Fragment shader“

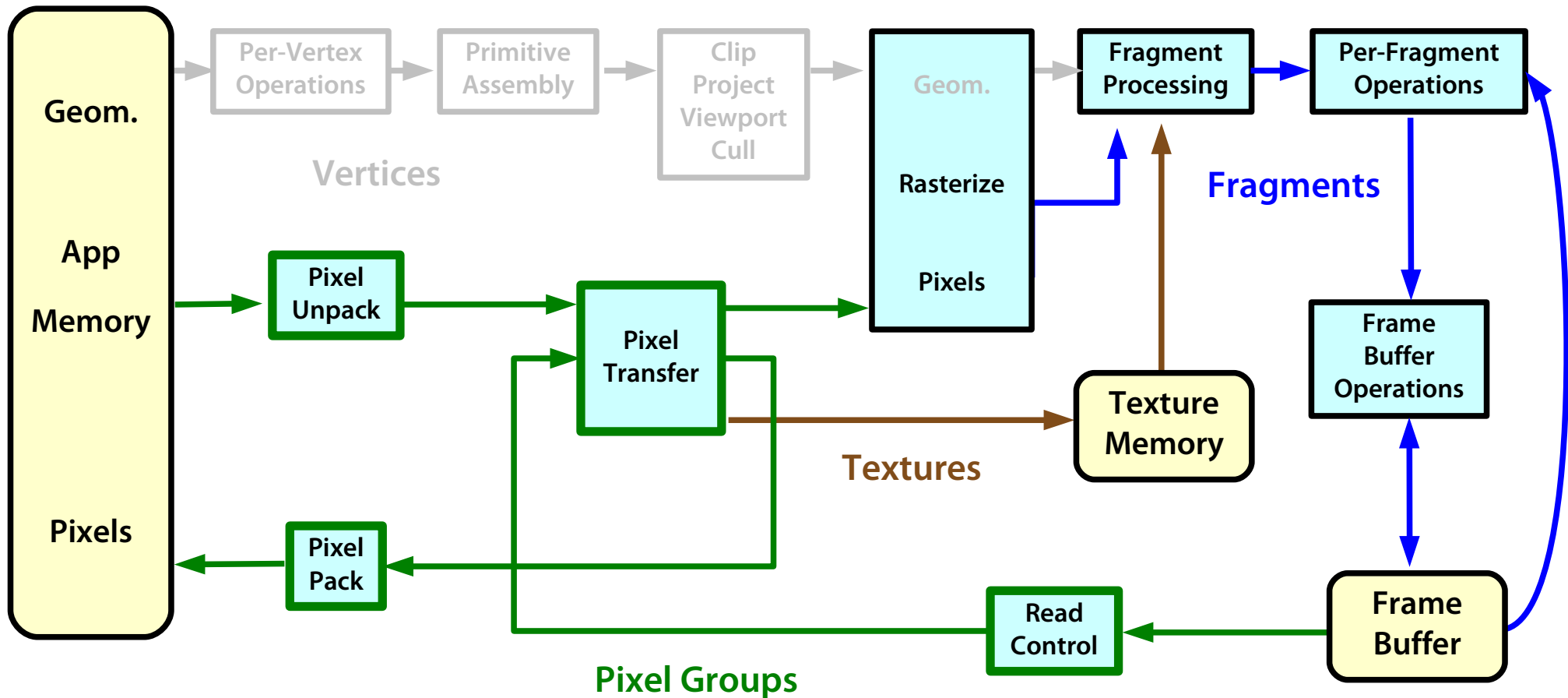
Texturovací operace

- maximálně **optimalizované** operace
- výběr barvy z texturovací paměti
- interpolace texelů
 - » mip-mapping, anisotropic filtering...
- kombinace několika textur (výběr z mnoha operací)
- zvláštní efekty (bump-mapping, environment mapping)

Kombinace primární a sekundární barvy (diffuse, specular)



Rastrové obrázky v OpenGL





Vertex shader

Modul zpracování vrcholů

- transformace vrcholů
- transformace a normalizace normálových vektorů
- výpočet/transformace texturovacích souřadnic
- výpočet osvětlovacích vektorů
- nastavení materiálových konstant do vrcholů

Nemůže ovlivnit

- **počet vrcholů!** (nelze přidat ani ubrat vrchol*)
 - » částečné řešení: degenerace primitivu
- typ / topologii geometrických primitiv
 - » řešení: geometry shader



Fragment shader

Modul zpracování fragmentů

- aritmetické operace s interpolovanými hodnotami
- čtení dat z textur
- aplikace textur
- výpočet mlhy
- závěrečná syntéza barvy fragmentu
- možnost modifikace hloubky fragmentu

Nemůže ovlivnit

- **počet fragmentů!** (nelze přidat fragment*)
 - » zrušení fragmentu: discard
- **polohu fragmentu** na obrazovce [x, y]



Shader uniforms (constants)

Hodnoty, které se nemění příliš často

- jsou **konstantní během jedné kreslicí dávky** (batch)
- aplikace je předává do GPU jednotlivě nebo přes buffery
- omezený objem dat (až 1024 vektorů)
 - » data většího rozsahu se už musí předávat přes textury/buffery

Typické uniforms/konstanty

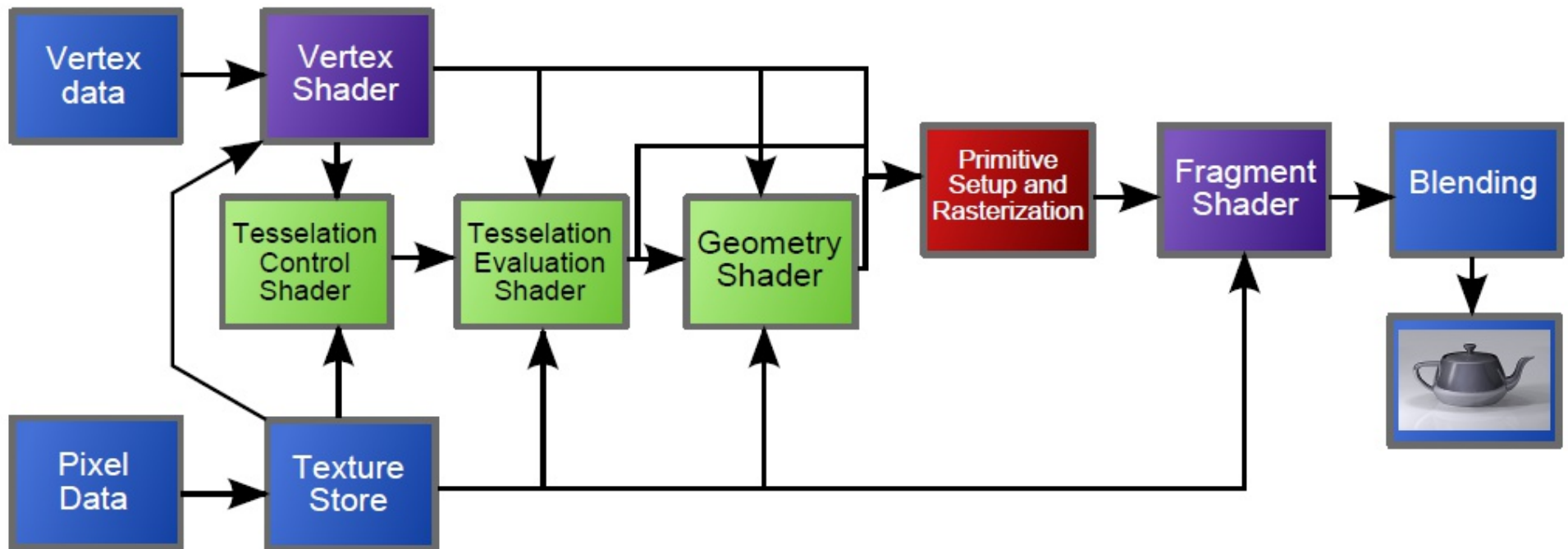
- **transformační matice** (matrix stack)
 - » **model matrix** (mění se s každou instancí objektu)
 - » **view-projection matrix** (mění se jednou za snímek)
- mody vykreslování
- poloha světel, kamery a další hodnoty pro stínování



„Novinky“ v OpenGL 3.x+ (2009-)

Dva další kroky zpracování geometrie na GPU

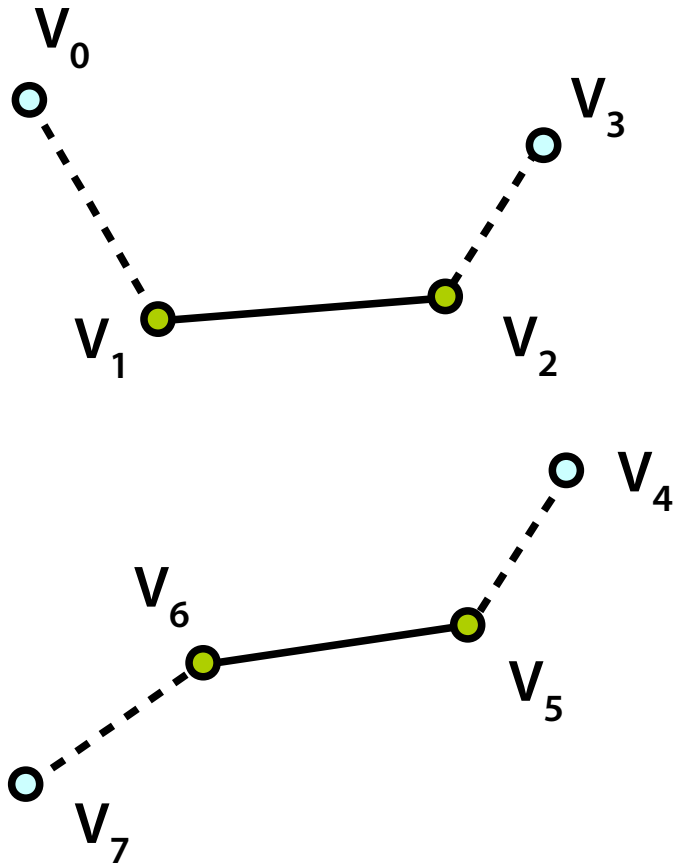
- Geometry shader (OpenGL 3.2+)
- Tessellation shaders (OpenGL 4.0+)



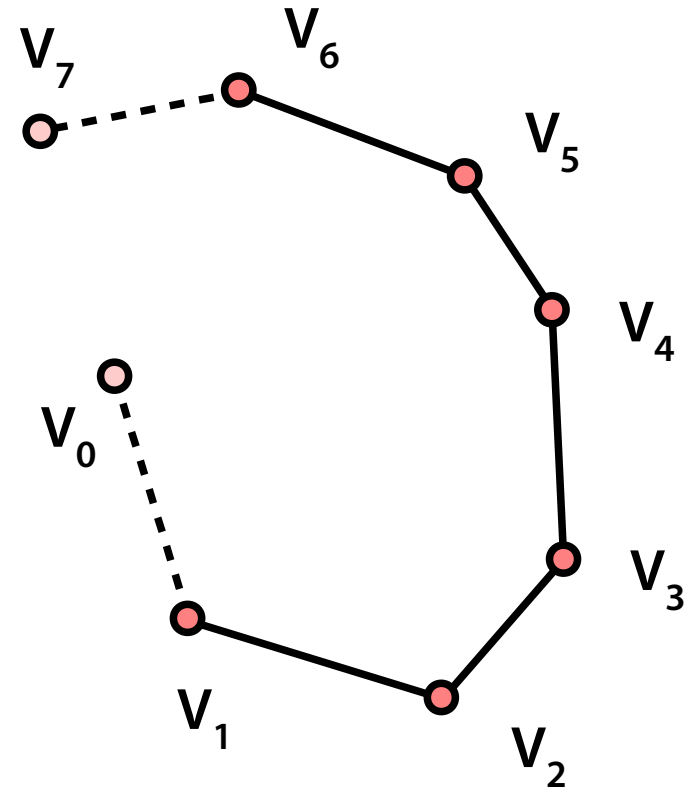


Geometrická primitiva IV

GL_LINES_ADJACENCY



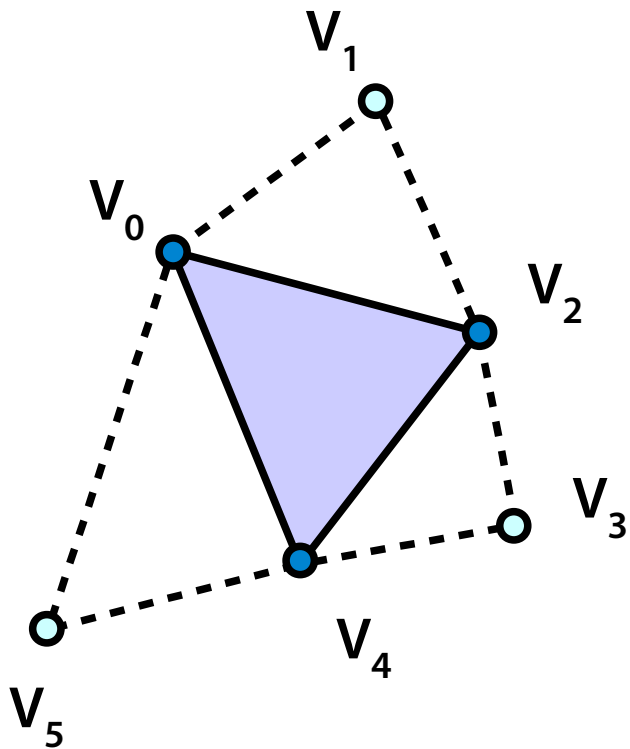
GL_LINE_STRIP_ADJACENCY



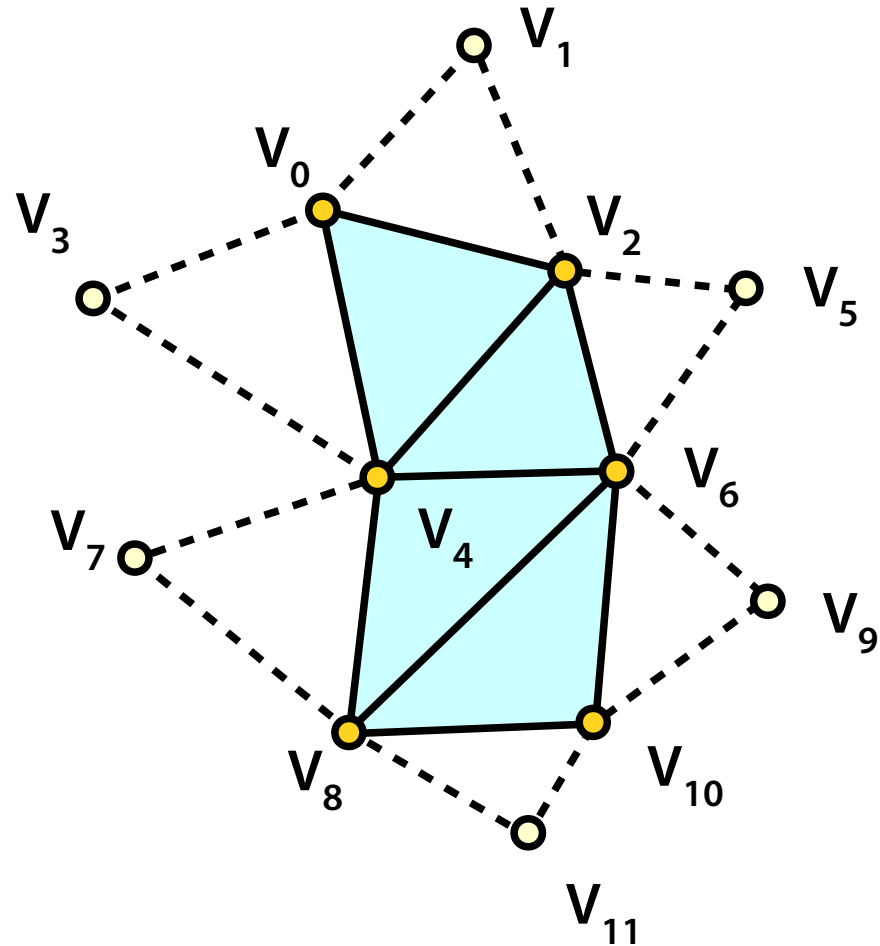


Geometrická primitiva V

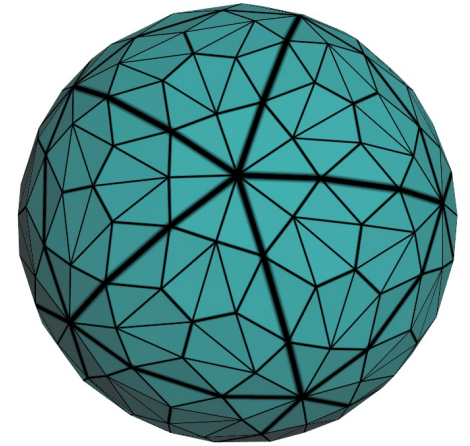
GL_TRIANGLES_ADJACENCY



GL_TRIANGLE_STRIP_ADJACENCY



Geometrické procesory



Tessellation shaders

- nově v OpenGL 4.0
- HW podporované dělení ploch, subdivision (spline pláty...)
- dva shadery: „tessellation control“ a „tessellation evaluation“
- první definuje topologii, druhý počítá geometrii (koeficienty)

Geometry shader

- od OpenGL 3.2
- těsně před rasterizační jednotkou
- možnost pracovat s celými primitivy
- obecnější než TS, avšak pomalejší (na jednoduchá schémata dělení se nehodí)






Programování procesorů v GPU

Vertex shader, Fragment shader...

- kód zavedený ve vrcholovém, fragmentovém... procesoru

Aplikační programátor může tyto kódy měnit!

- **HW nezávislé*** programovací jazyky
- **strojový kód pro GPU** se kompiluje až v době běhu aplikace (RT optimalizace, různé profily/verze)
- low-level instrukce (jazyk se podobá assembleru)
- nebo **vyšší jazyky** ~~Cg~~, HLSL, GLSL (podobné)

 ~~Cg~~  HLSL  GLSL
NVIDIA Microsoft OpenGL



GLSL example (vertex shader)

```
#version 130

in vec4 position;
in vec3 normal;
in vec2 texCoords;
in vec3 color;

out vec2 varTexCoords;
out vec3 varNormal;
out vec3 varWorld;
out vec3 varColor;
flat out vec3 flatColor;

uniform mat4 matrixModelView;
uniform mat4 matrixProjection;

void main ()
{
    gl_Position = matrixProjection * matrixModelView * position;
    // Propagated quantities.
    varTexCoords = texCoords;
    varNormal     = normal;
    varWorld      = position.xyz;
    varColor      = flatColor = color;
}
```



GLSL example (Phong shader + texture)

```
#version 130

in vec2  varTexCoords;           // [s, t]
in vec3  varNormal;             // world coordinate system
in vec3  varWorld;
in vec3  varColor;             // Gouraud color
flat in  vec3 flatColor;

uniform bool useShading;
uniform vec3 globalAmbient;
uniform vec3 lightColor;
uniform vec3 lightPosition;    // world coordinate system
uniform vec3 eyePosition;      // world coordinate system
uniform vec3 Ks;
uniform float shininess;
uniform bool useTexture;
uniform sampler2D texSurface;

out vec3 fragColor;            // output = fragment color

...
```



GLSL example (Phong shader + texture)

```
void main ()
{
    if (useShading)
    {
        vec3 P = varWorld;
        vec3 N = normalize(varNormal);
        vec3 L = normalize(lightPosition - P);
        vec3 V = normalize(eyePosition - P);
        vec3 H = normalize(L + V);

        float cosb = 0.0;
        float cosa = dot(N, L);
        if ( cosa > 0.0 )
            cosb = pow(max(dot(N, H), 0.0), shininess);

        vec3 kkd;
        if (useTexture)
            kkd = vec3(texture2D(texSurface, varTexCoords));
        else
            kkd = varColor;

        fragColor = kkd * globalAmbient +
                    kkd * lightColor * cosa +
                    ks    * lightColor * cosb;
    }
    else
        fragColor = flatColor;
}
```



Typický main() v OpenGL

Real-time simulátor (videohra, 3D editor/prohlížeč, apod.)

- vykreslování 3D scény plnou rychlostí (maximální fps)

Nadstavba nad OpenGL (okno + „**swap-chain**“, interakce)

- SDL, OpenTK, freeglut, Qt, wxWidgets...

ColdStart – inicializace nadstavby i OpenGL

- globální start a nastavení OpenGL, mody grafiky (color, depth-buffer, swap-chain), kompilace shaderů, [načtení textur]...

WindowResize – reakce na změnu velikosti okna

- přenastavení **viewport** a **projekční matice**
- změna dalších souvisejících hodnot (např. pro UnProject())



Vnitřní smyčka (pozor na synchronizaci)

ProcessEvents – čtení a zpracování událostí

- interaktivní: **myš, klávesnice, pad**, ostatní: **síťová komunikace...**
- lze implementovat **asynchronně** (UWP, pozor na ochranu dat!)

Update (Simulate) – simulace scény

- logika simulátoru/hry, simulace fyziky, přehrávání animací...
- komponenta **masivně měnící aktuální stav** simulovaného světa
- lze implementovat **asynchronně** (i ve více vláknech), ale pozor...!

Render – vykreslení scény

- **nakreslení aktuálního stavu** 3D reprezentace a předání výsledku do swap-chainu
- zde je potřeba **OpenGL**



Typický main() v OpenGL I

```
world world(..);           // simulated world
Swapchain swapchain(..);   // object presenting result 2D graphics to the window
bool done = false;        // app-exit request

void main (int argc, char **argv)
{
    CommandLineArguments(argc, argv);

    ColdStart("Test app", swapchain, PIX_FORMAT_R8G8B8A8, DOUBLE_BUFFERING, ..);

    WindowResize();

    world.InitSimulation();

    while (!done)
    {
        // simulate & render one frame.

        ProcessEvents();

        Update(GetSystemTime());

        if (WindowVisible())
            Render();
    }
}
```



Typický main() v OpenGL II

```
void ProcessEvents ()           // process all available events
{
    Event ev;
    while (PollEvent(ev))
        switch (ev.type):
        {
            case WINDOW_RESIZE:
                windowResize();
                break;
            case WINDOW_CLOSE:
                done = true;
                break;
            case KEY_DOWN:
                ...
            case MOUSE_BUTTON_DOWN:
                ...
            case MOUSE_MOVE:
                ...
        }
}

void Update (double time)       // simulate the world (to the given target time)
{
    world.UpdateForces(time);    // application-related code
    world.UpdatePositions(time);

    world.PlayAnimations(time); // scripted animations
}
```




Typický main() v OpenGL III

```
void Render ()           // Render current world's state using OpenGL.
{
    swapchain.FrameStart();

    // Clear frame-buffer & depth-buffer.
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    SetViewProjMatrices(); // camera is constant for the whole frame..

    for (const auto& section : world.GetSections()) // R/O copy of world's data (?)
    {
        section.SetModelMatrix(); // every section has its own model transform
        section.SetShaders(); // change shaders only if necessary
        section.SetTextures(); // change textures only if necessary
        section.SetUniforms(); // either here or before every batch

        // One scene section can have multiple rendering batches.
        for (const auto& batch : section.batches)
        {
            batch.SetBuffers(); // change VB/IB only if necessary (glBindBuffer..)
            batch.Render(); // actual rendering command[s] (glDrawElements,
                            // glDrawArrays, glDrawElementsInstanced, ..)
        }
    }

    swapchain.Present(); // "SwapBuffers()" ..
}
```



Literatura

Tomas Akenine-Möller, Eric Haines et al.: *Real-time rendering*, 4th edition, A K Peters, 2018, ISBN: 9781138627000

OpenGL ARB: *OpenGL Programming Guide*, 8th edition, Addison-Wesley, 2013, ISBN: 0321773039