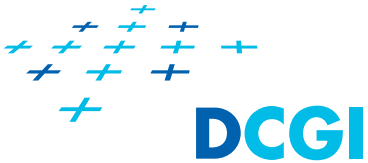


Zobrazovací řetězec a obrazová paměť, operace s fragmenty

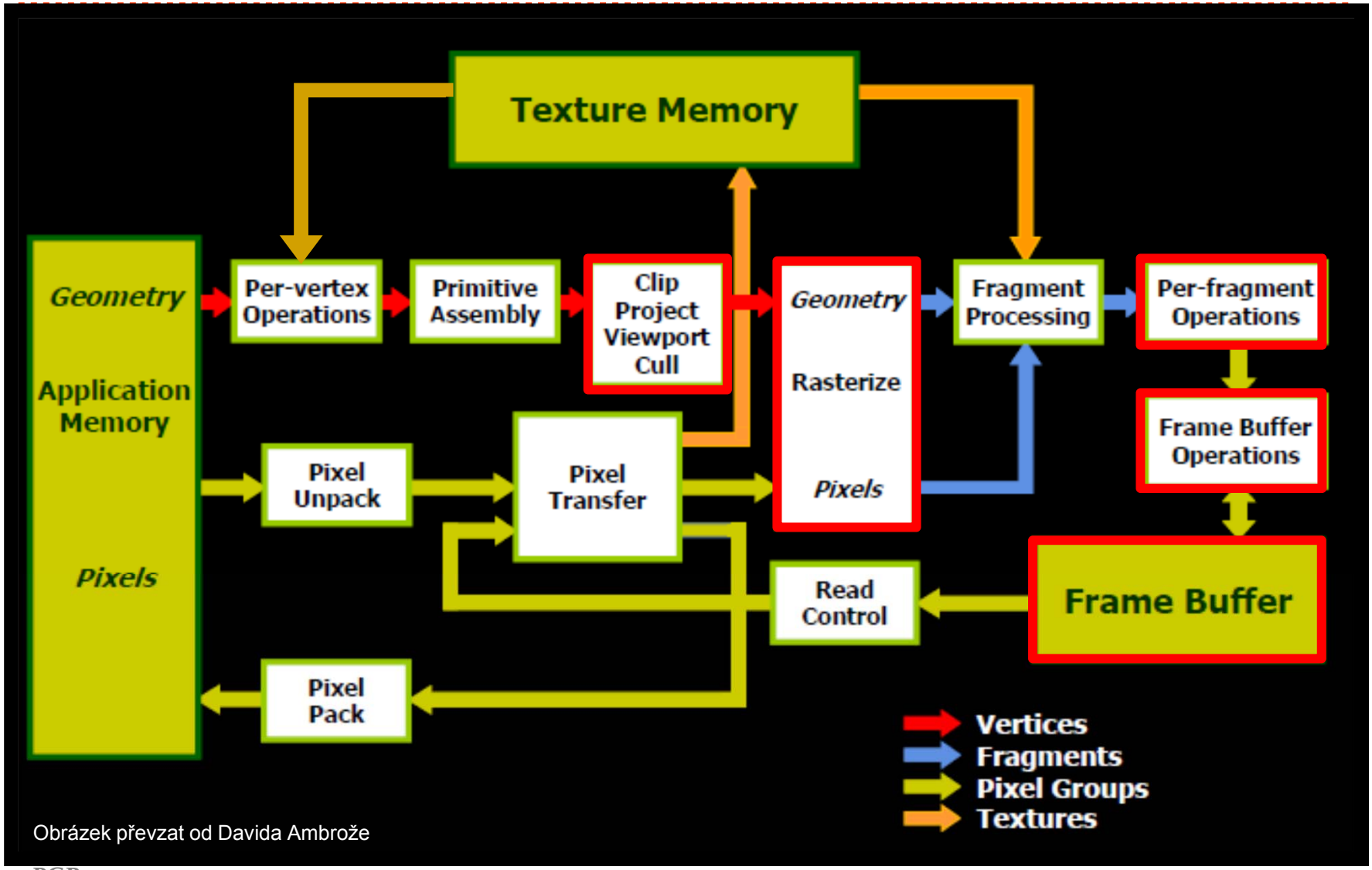
Petr Felkel

Katedra počítačové grafiky a interakce, ČVUT FEL
místnost KN:E-413 na Karlově náměstí
E-mail: felkel@fel.cvut.cz

S použitím materiálů Bohuslava Hudce, Jaroslava Sloupa a úprav Vlastimila Havrana

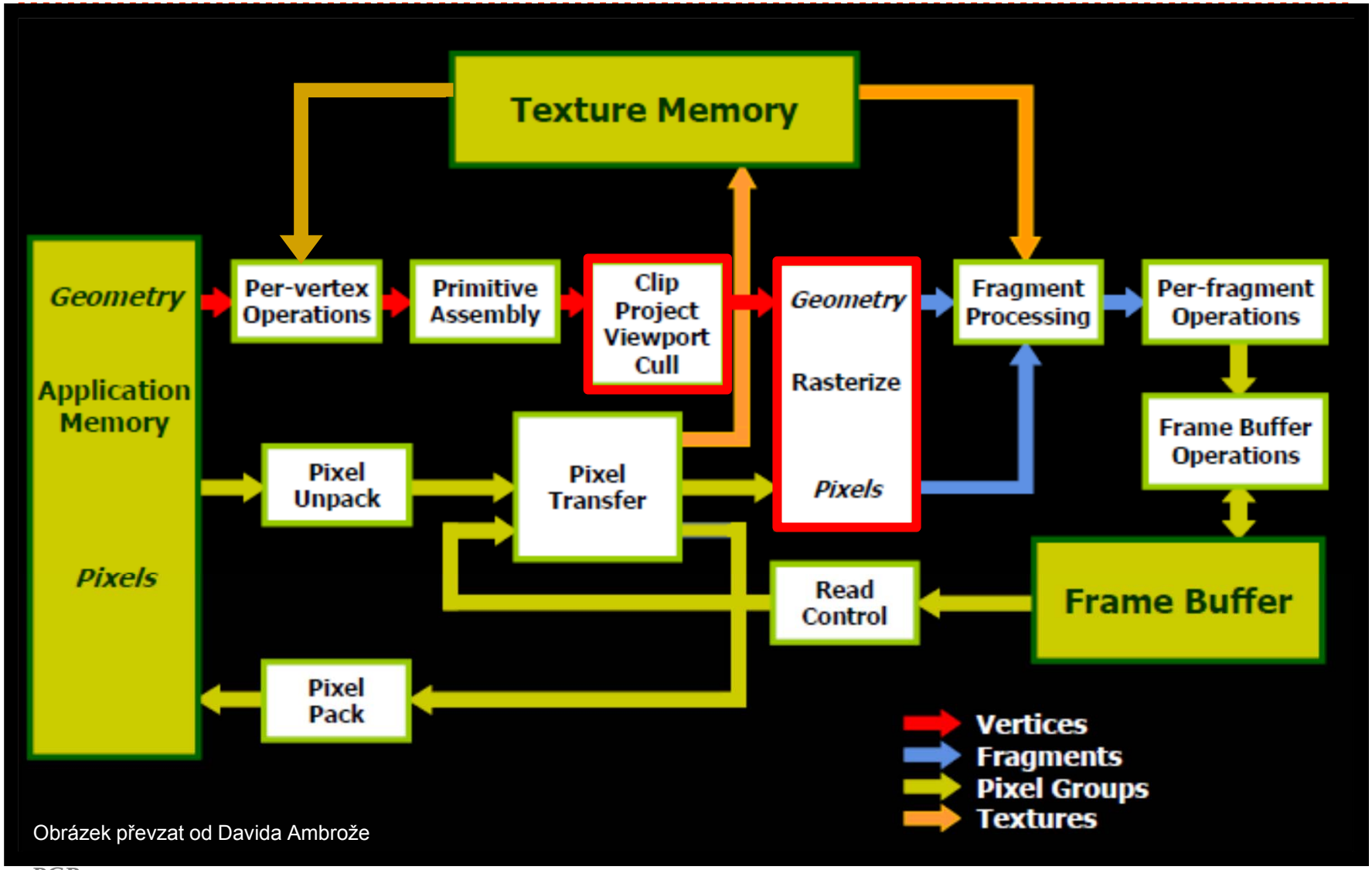
Poslední změna: 16.6.2016

Zbývající bloky zobrazovacího řetězce



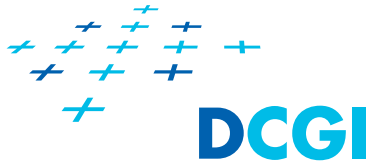
Obrázek převzat od Davida Ambrože

Od vrcholu k fragmentu



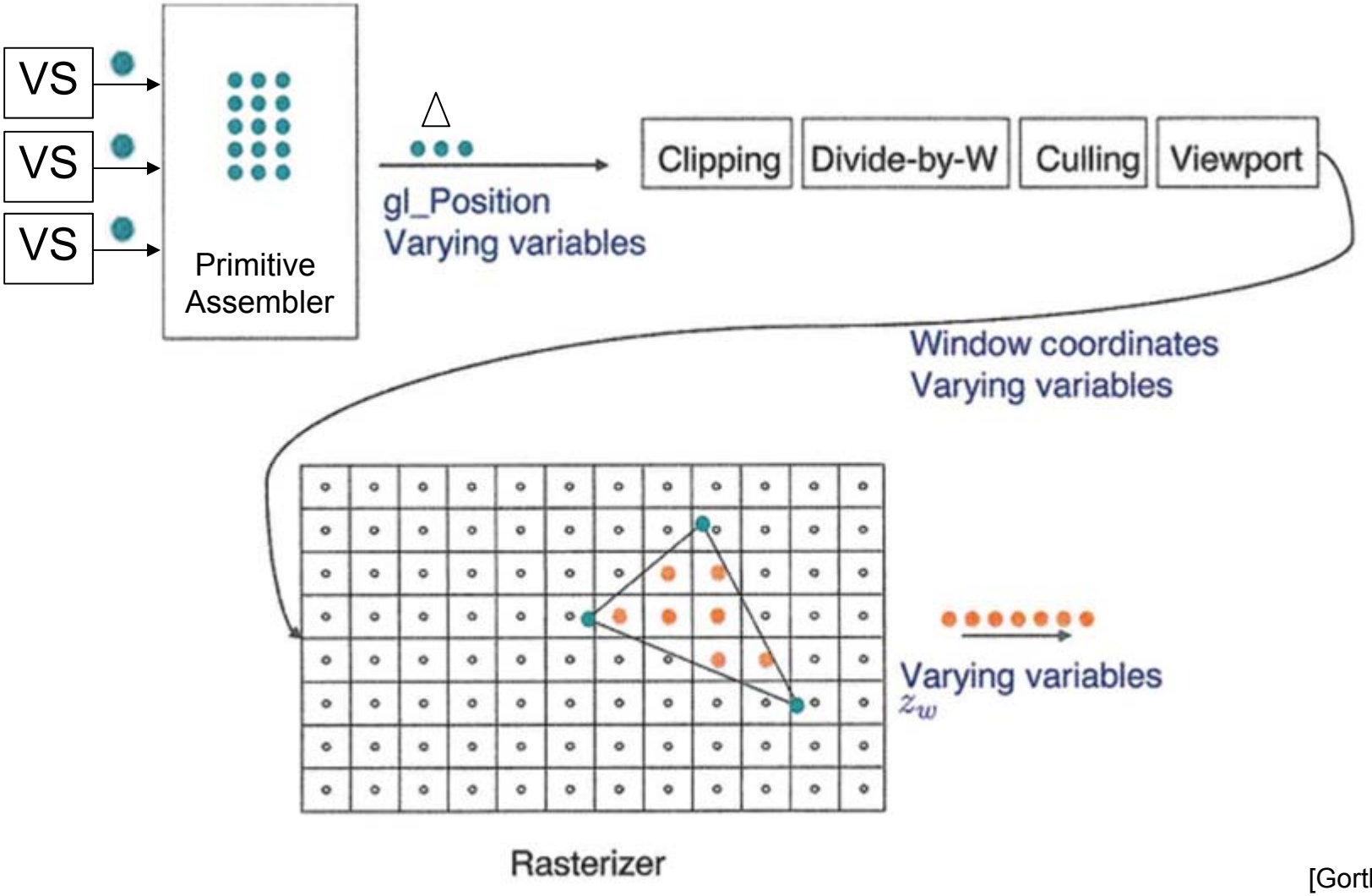
Obrázek převzat od Davida Ambrože

Od vrcholu k fragmentu



- Po průchodu vertex shaderem vstupují vrcholy do fixní části zobrazovacího řetězce
  - sestavení primitiva (*primitive assembly*)
  - ořezání do pohledového jehlanu (*clipping*)
  - převod z homogenních souřadnic (*Divide-by-W*)
  - umístění na obrazovku (*Viewport*)
  - rasterizace a interpolace (*Rasterizer*)
- Pak následuje fragment shader
  - V něm se spočítá výsledná barva fragmentu
  - **Fragment** jsou všechna data adresovaná souřadnicemi jednoho konkrétního pixelu, viz dále.

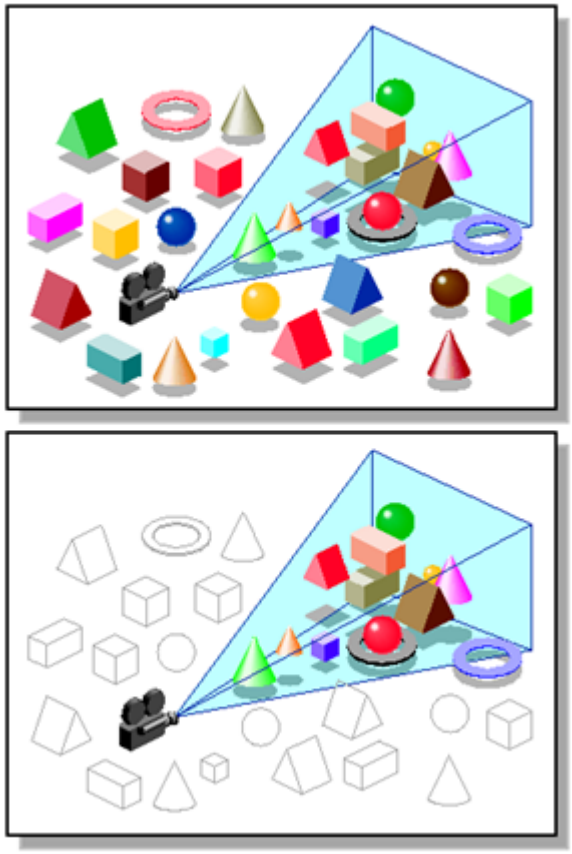
Od vrcholu k fragmentu

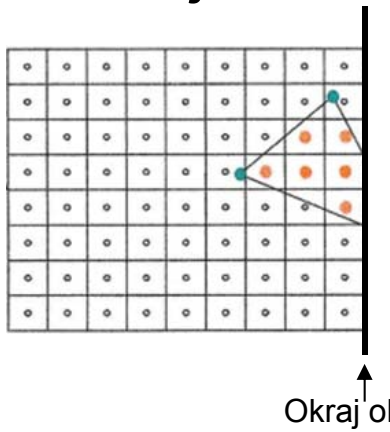


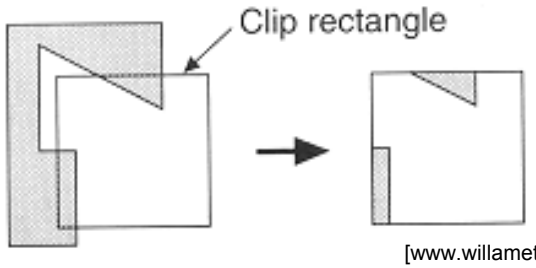
[Gortler]

Ořezání (*clipping*)

- Ořezání odstraní geometrii mimo pohledový jehlan (*viewing frustrum*) - *top, bottom, left, right, near, far*
- Tedy i části primitiv za kamerou
- Šetří se tím rasterizace a výpočty (nepočítá se to, co není vidět)
- Vznikají nové vrcholy







[www.willamette.edu]

[http://techpubs.sgi.com]

Clipping

Ve kterých souřadnicích ořezávat?

DCGI

- V souřadnicích kamery?

  => Ne, ještě neproběhla projekce.

- V normalizovaných souřadnicích (po dělení w_c)?

  => Ne, již došlo k překlopení vrcholů za kamerou

- V ořezových souřadnicích *(clip-space)?*

  => ANO – ještě nedošlo k dělení w_c, trojúhelník je ve 4D

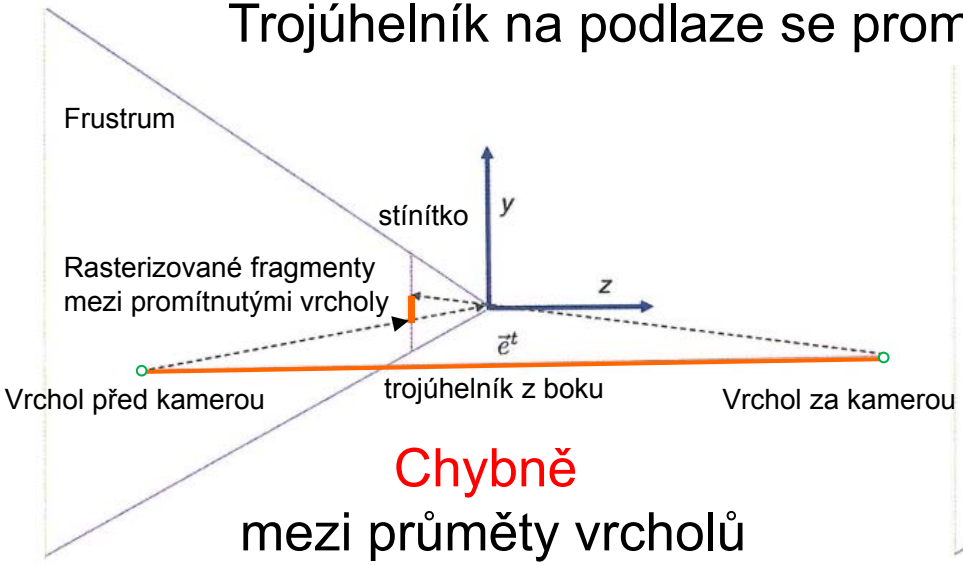
  $$-w_c < x_c < w_c$$
  $$-w_c < y_c < w_c$$
  $$-w_c < z_c < w_c$$

Závěr: Pozice i atributy nových vrcholů se počítají v ořezových souřadnicích – a v nich se lineárně interpolují

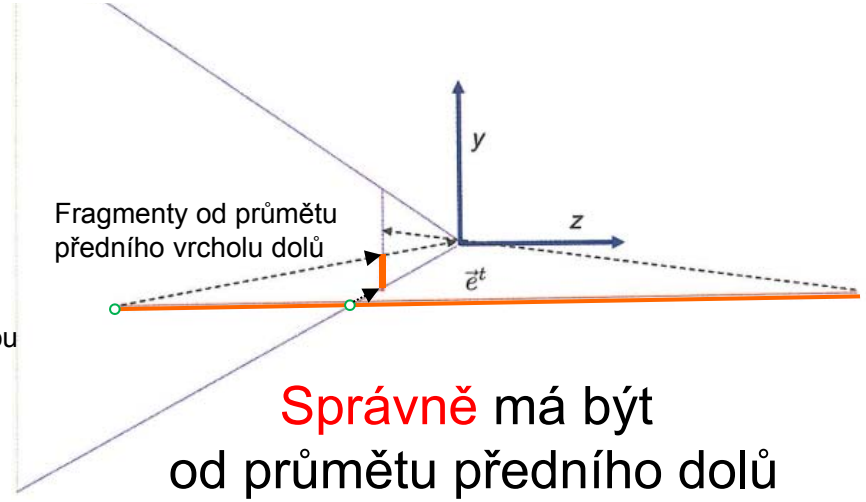
Divide-by-W (až pak se násobí interpolovaným $\frac{1}{w}$ => perspektivně správná interpolace)

Odstranění části primitiv za kamerou

Trojúhelník na podlaze se promítne na stínítko jako úsečka

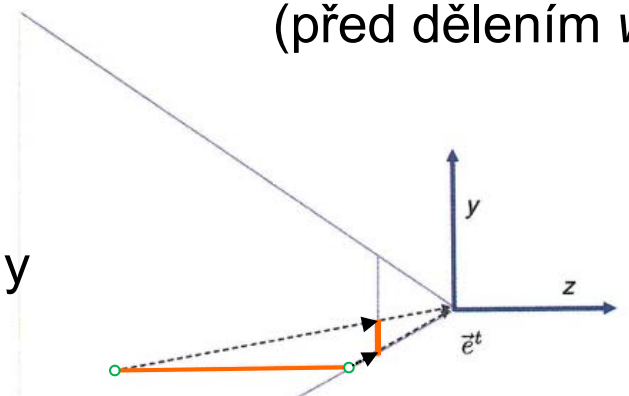


Chybně
mezi průměty vrcholů
(po dělení *w*)



Správně má být
od průmětu předního dolů
(před dělením *w*)

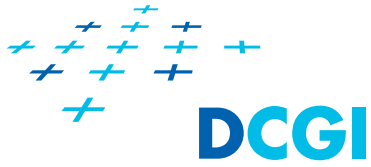
Vykreslujeme totiž fragmenty
mezi průměty koncových bodů
a dělení $w_c = -z$ překlopí oba vrcholy
do stejné průmětny



[Gortler]

tedy průmět po ořezání

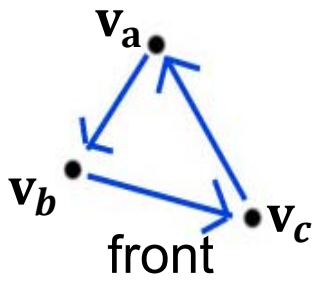
Eliminace odvrácených plošek

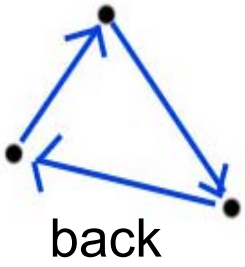


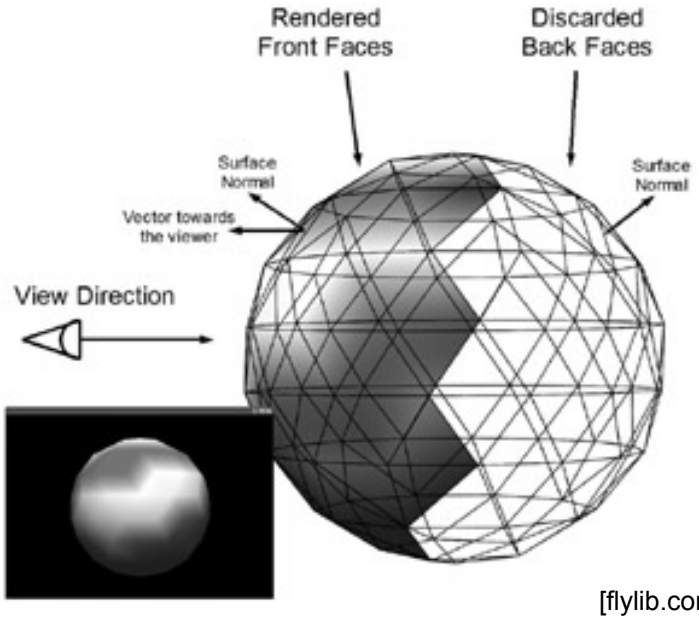
- U uzavřených těles *(watertight)* nikdy nevidíme odvrácenou stěnu
- Vrcholy zadáváme proti směru hodinových ručiček *(ccw)*
- Pak stačí vyřadit plošky s odvrácenou normálou

$$\mathbf{n} = (\mathbf{v_b} - \mathbf{v_a}) \times (\mathbf{v_c} - \mathbf{v_b})$$

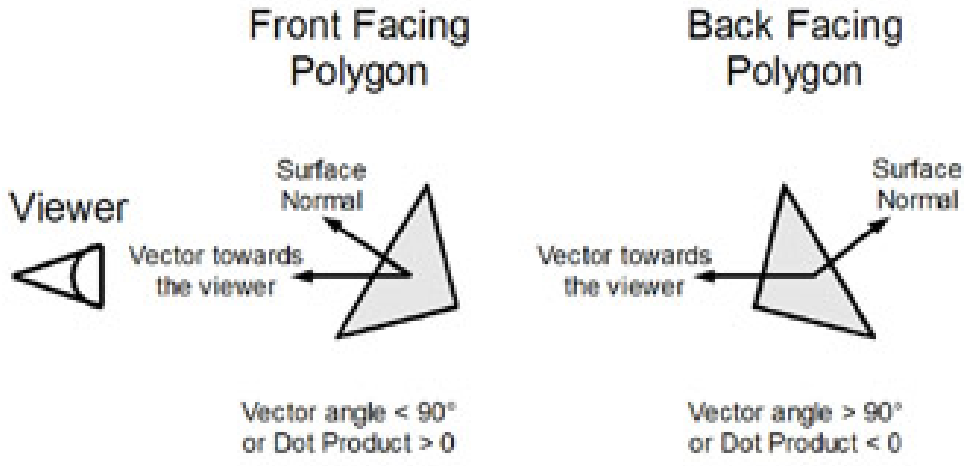
back face = if$((\mathbf{e} \,.\, \mathbf{n}) < \mathbf{0})$







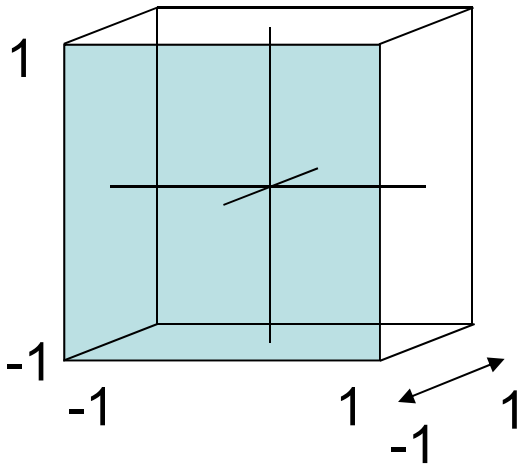
[flylib.com]



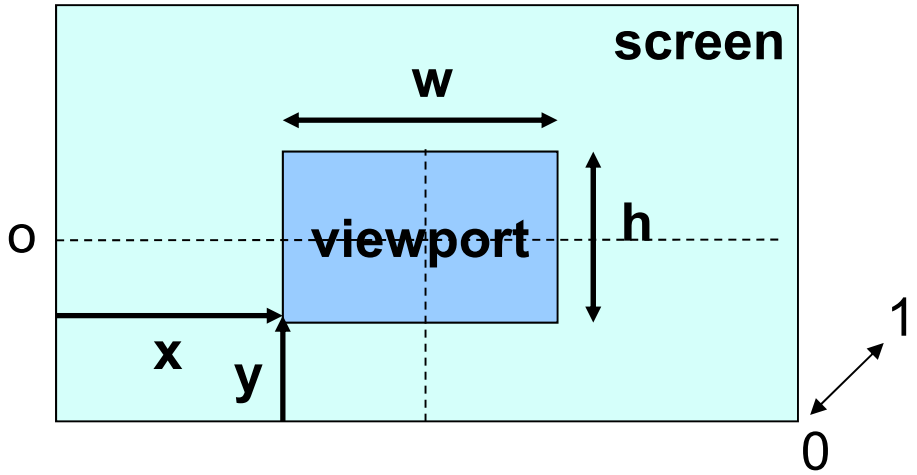
Viewport

- Transformuje pozici vrcholu
  z normalizovaných souřadnic zařízení $\langle -1, 1 \rangle^3$
  - x, y do souřadnic na obrazovce (pozice fragmentu na stínítku)
  - z do intervalu $\langle 0, 1 \rangle$, reprezentující vzdálenost od stínítka
    - zNear => 0, zFar => **1** !!!
    - používá se k určování viditelnosti vykreslovaných objektů

Normalizované souřadnice zařízení $[x_d\ y_d\ z_d\ ]^t$



Souřadnice formátu na obrazovce $[x_w\ y_w\ z_w\ ]^t$

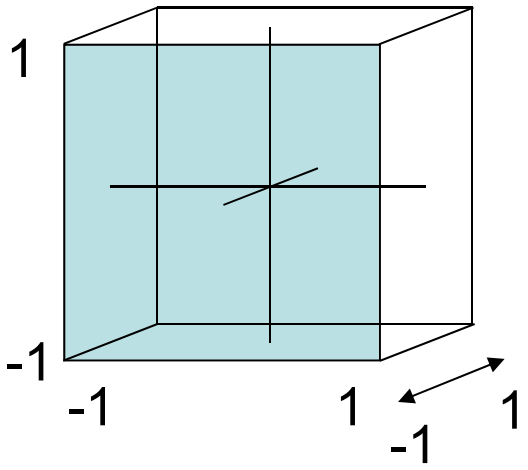


Viewport – zvlášť posun a scale

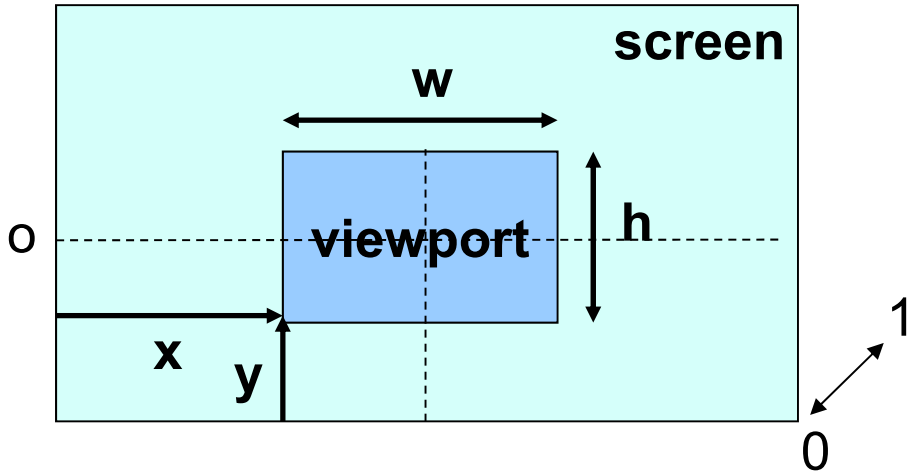
- Matice transformace záběru (pracoviště) …viz předn. 05

$$\begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & x + {}^{w}/_{2} \\ 0 & 1 & 0 & y + {}^{h}/_{2} \\ 0 & 0 & 1 & {}^{1}/_{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} {}^{w}/_{2} & 0 & 0 & 0 \\ 0 & {}^{h}/_{2} & 0 & 0 \\ 0 & 0 & {}^{1}/_{2} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_d \\ y_d \\ z_d \\ 1 \end{bmatrix}$$

Normalizované souřadnice zařízení $[x_d\ y_d\ z_d\ ]^t$



Souřadnice formátu na obrazovce $[x_w\ y_w\ z_w\ ]^t$

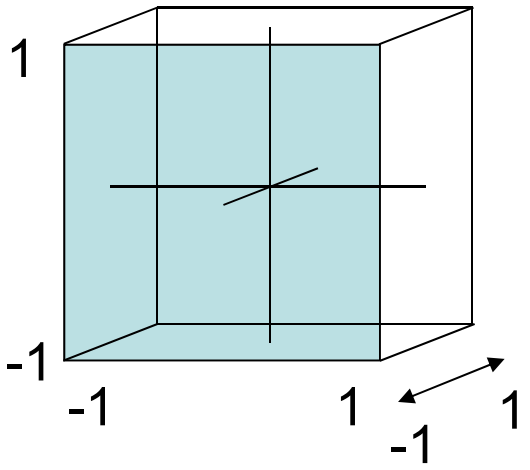


Viewport

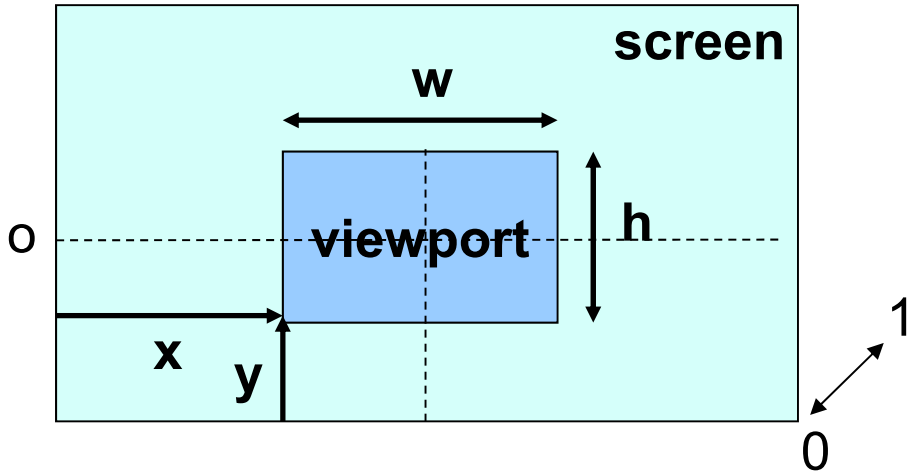
- Matice transformace záběru (pracoviště) …viz předn. 05

$$\begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} = \begin{bmatrix} w/2 & 0 & 0 & x + w/2 \\ 0 & h/2 & 0 & y + h/2 \\ 0 & 0 & 1/2 & 1/2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_d \\ y_d \\ z_d \\ 1 \end{bmatrix}$$

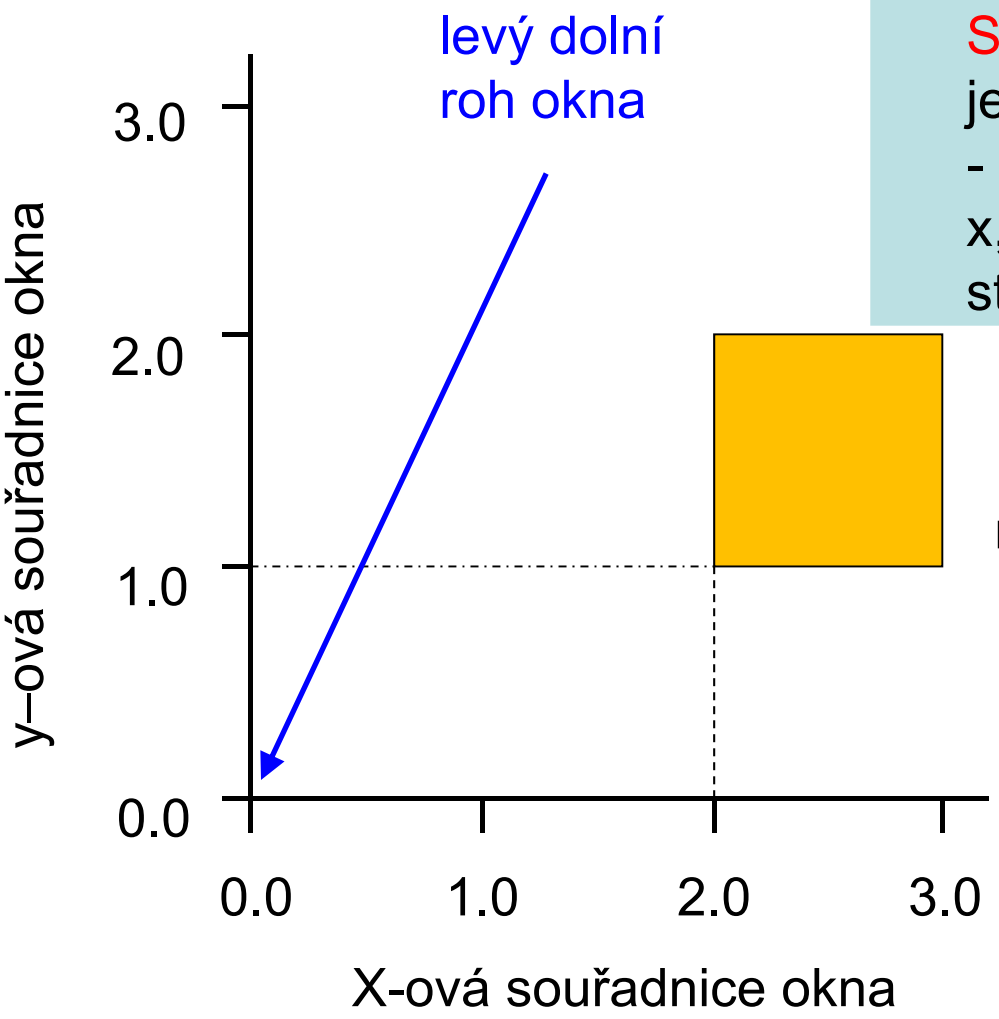
Normalizované souřadnice zařízení $[x_d\ y_d\ z_d\ ]^t$



Souřadnice formátu na obrazovce $[x_w\ y_w\ z_w\ ]^t$



Pixel coordinates

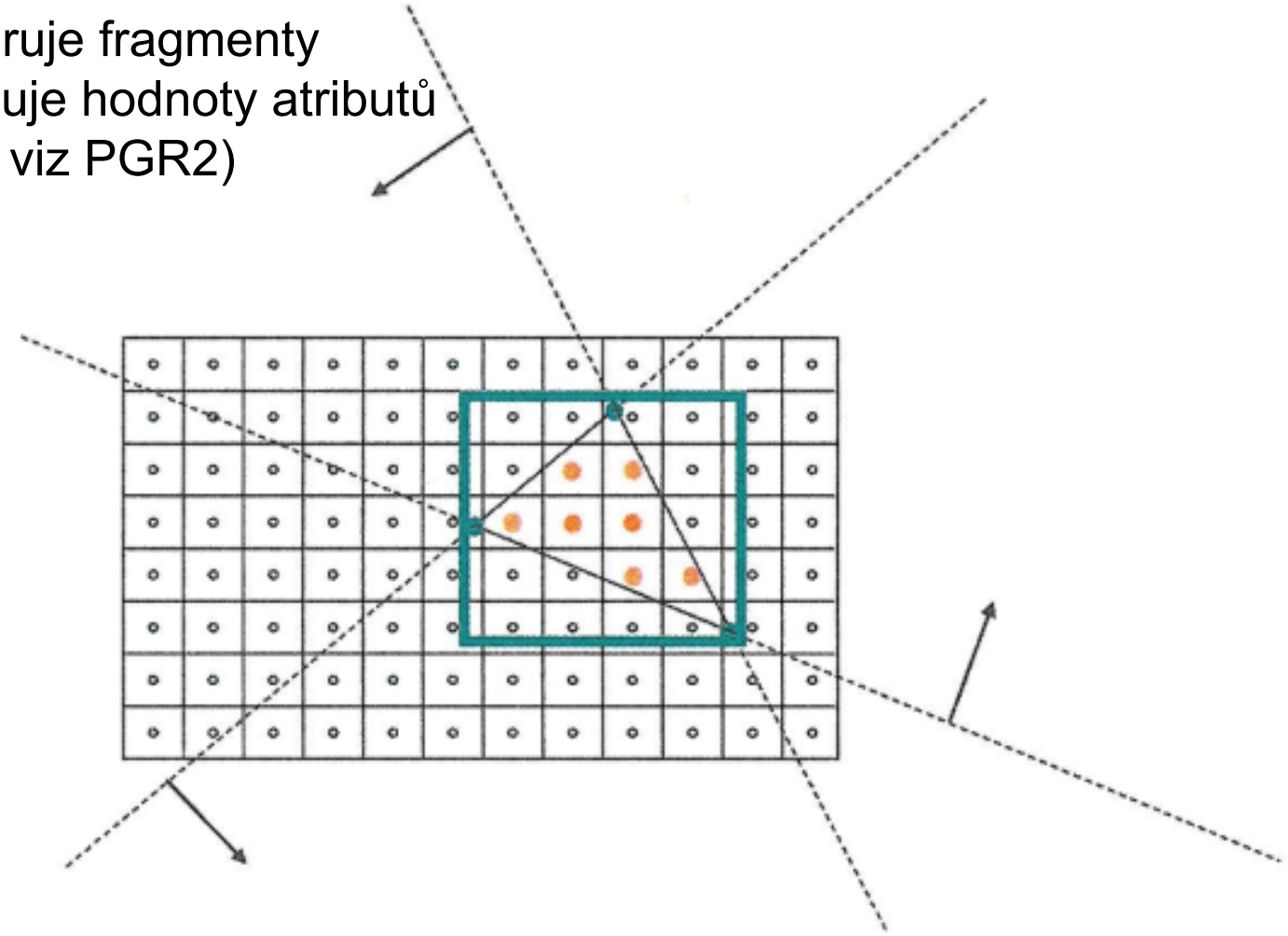


Souřadnice pixelu = souřadnice jeho levého dolního rohu
- pixel (x, y) pokrývá plochu mezi x,y a x+1, y+1
střed tedy má v bodě [x+0,5, y+0,5]

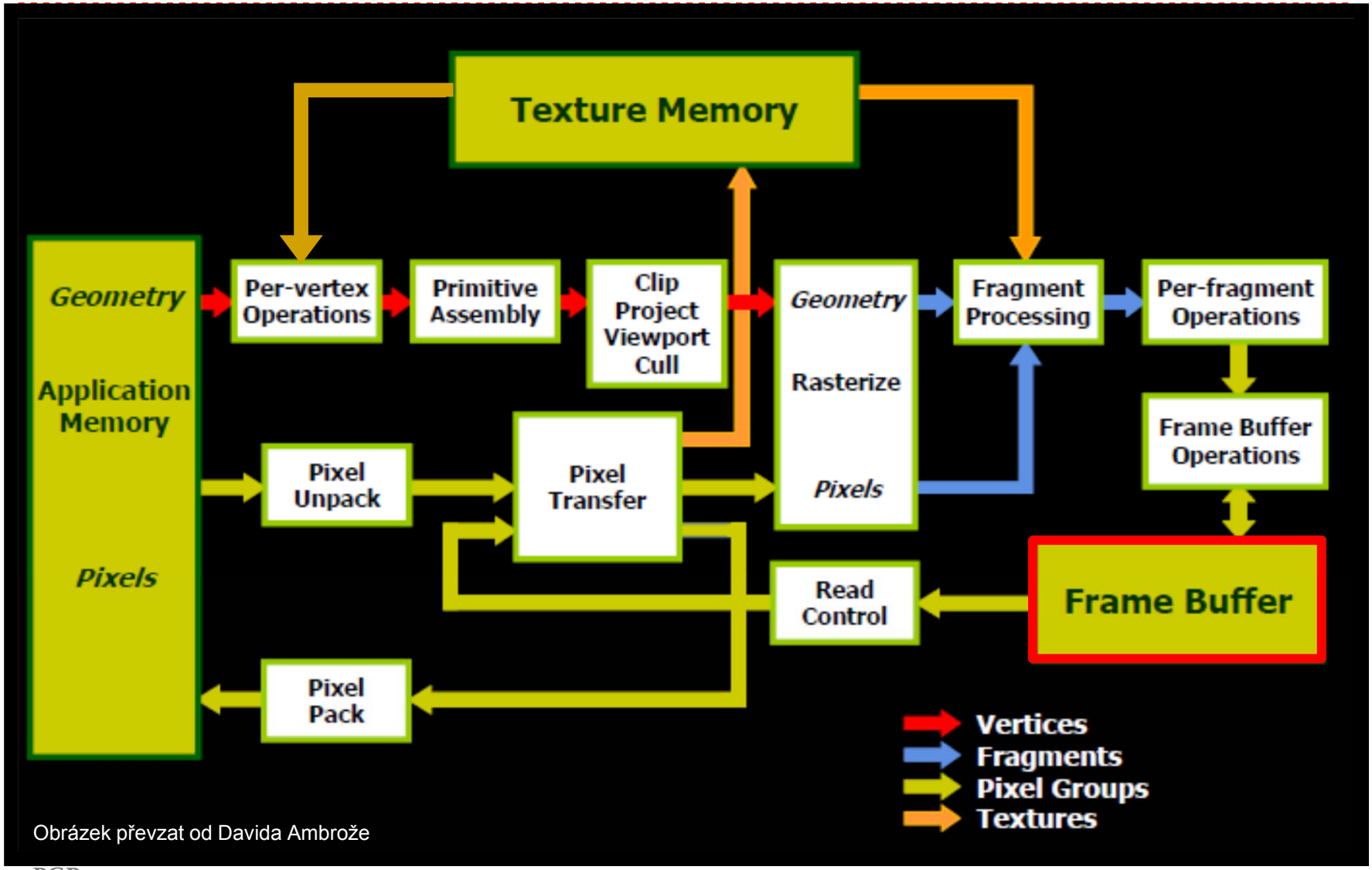
region occupied by a pixel (2, 1)

Rasterizace

Vygeneruje fragmenty
Interpoluje hodnoty atributů
(detaily viz PGR2)



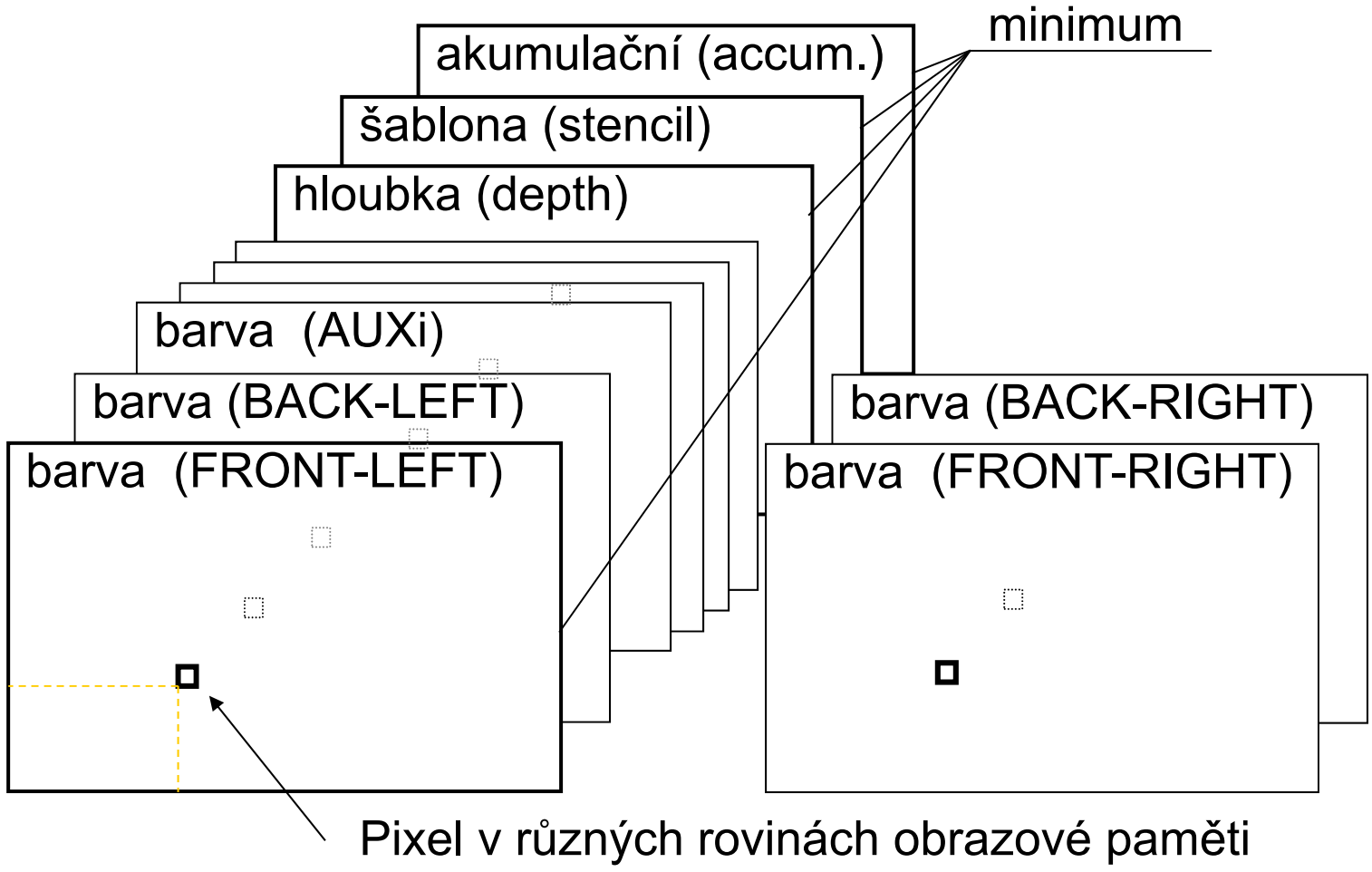
Framebuffer – vrstvy obrazové paměti



Obrázek převzat od Davida Ambrože

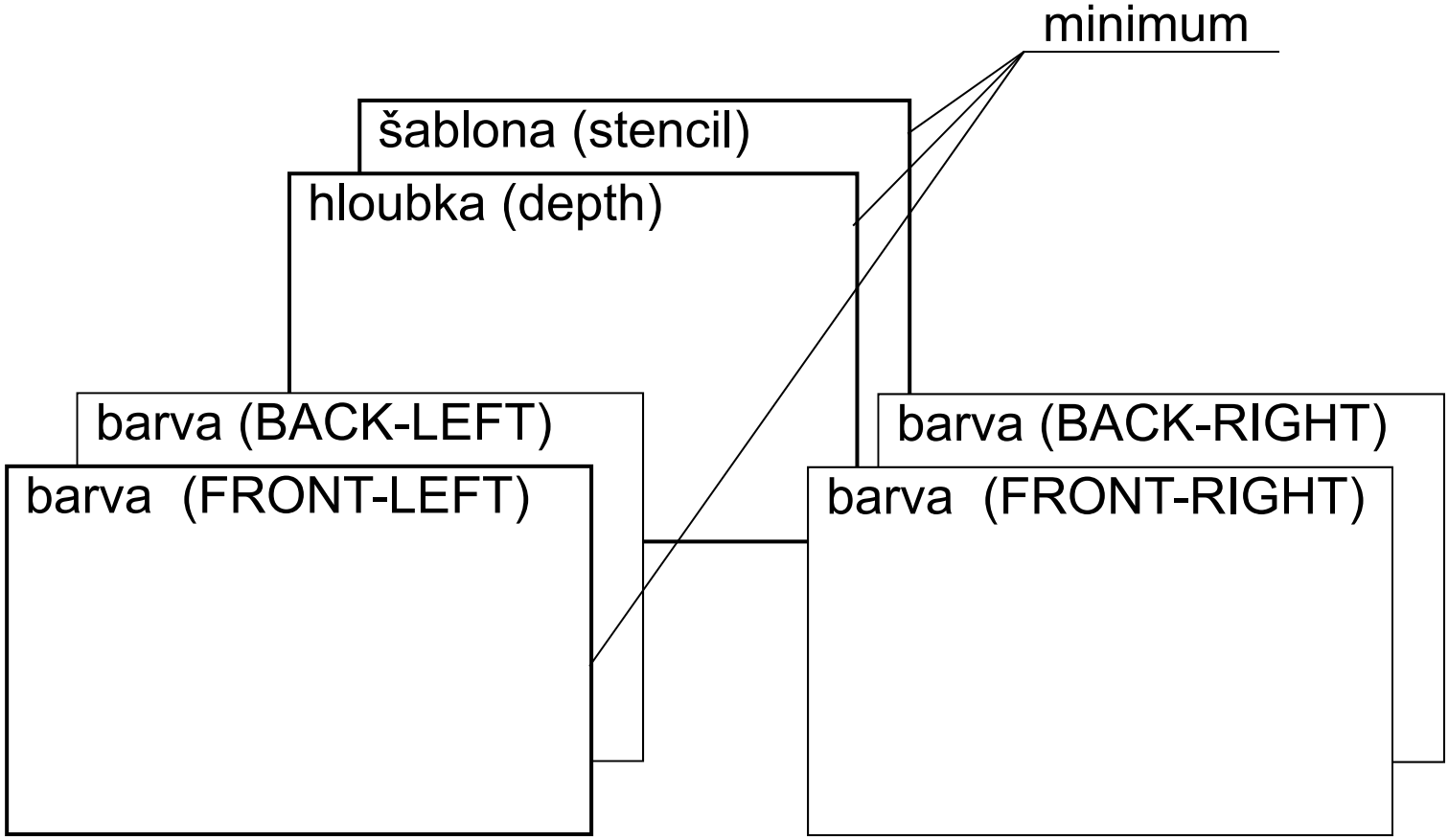
Obrazová paměť – framebuffer dříve ≤ 3.3

Součásti (vrstvy, roviny) obrazové paměti



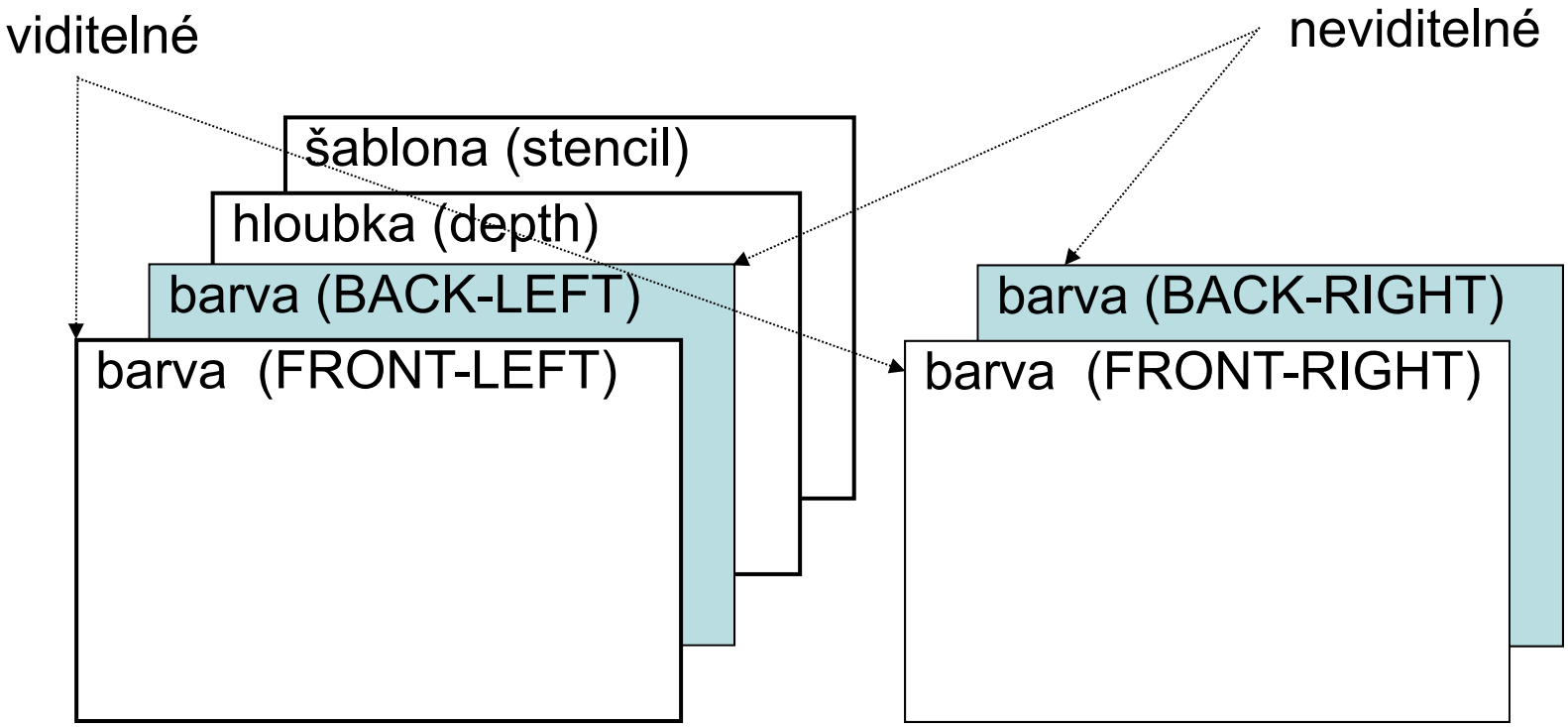
Obrazová paměť – framebuffer nyní ≥ 4.0

Součásti (vrstvy, roviny) obrazové paměti

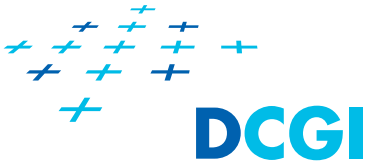


Paměť barvy - color buffer (1)

Barva
- RGB(A)
- Kreslí se do ní



Paměť barvy - color buffer (2)



- Minimální konfigurace OpenGL - FRONT_LEFT
- Stereo, double buffer - podporován?

```
glGetBooleanv( GL_STEREO, &b );
glGetBooleanv( GL_DOUBLEBUFFER, &b );
```

- Výběr, kam se kreslí barva fragmentů

```
glDrawBuffer( GLenum mode );                    // … jedna paměť
```

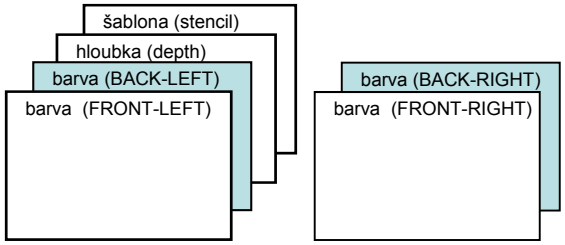
  - GL_NONE, … „nekreslí se“ nic
  - GL_FRONT_LEFT, GL_FRONT_RIGHT, GL_BACK_LEFT, GL_BACK_RIGHT, … jedna paměť

```
glDrawBuffers(GLsizei  n,  const GLenum * bufs);
// n – pamětí – multiple rendering targets (viz PGR2)
```

  - GL_COLOR_ATTACHMENTn … pro FBO (framebuffer object)

Paměť barvy - color buffer (3)

Barva v BACK bufferu



- RGB(A),
- Kreslí se do ní na pozadí (není vidět)
- Přepíná se s FRONT bufferem

```
glutSwapBuffers();
```

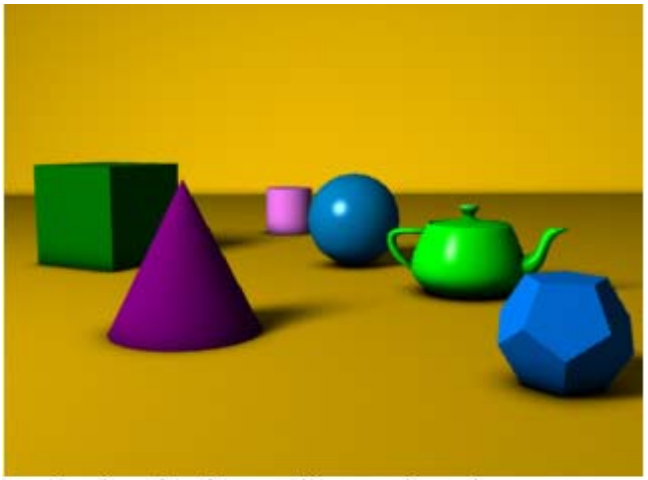
- Má typicky 8+8+8+8 bitů (RGBA)

- Výběr roviny na kreslení
  - **Mono:**
    ```
    glDrawBuffer(GL_BACK_LEFT); // GL_BACK
    ```
  - **Stereo:**
    ```
    glDrawBuffer(GL_BACK_LEFT); cameraL();draw();
    glDrawBuffer(GL_BACK_RIGHT);cameraR();draw();
    ```
- Výběr zdroje pro čtení `glCopyPixels();`
  - ```
    glReadBuffer(GL_BACK_LEFT);
    ```
    ```

Paměť hloubky – depth buffer (1)

Hloubka (depth-, Z-buffer Br [zed], Am [zee])

- vzdálenost oko-pixel
- typické použití: viditelnost
  (vzdálenější pixel je přepsán bližším)



A simple three dimensional scene

Blízká tělesa zakrývají vzdálená tělesa

- Bližší fragmenty překreslí ty vzdálenější
- Fragmenty, které jsou dále se zahodí (stejně by nebyly vidět)

Z-buffer representation

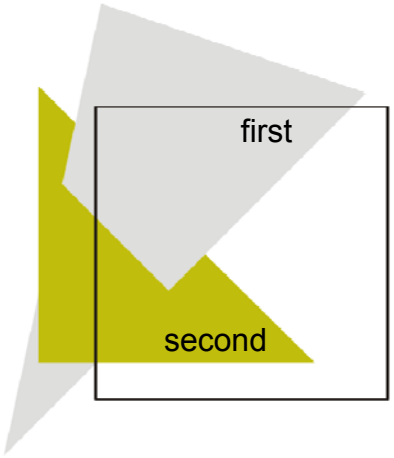
Lze řešit viditelnost bez paměti hloubky?

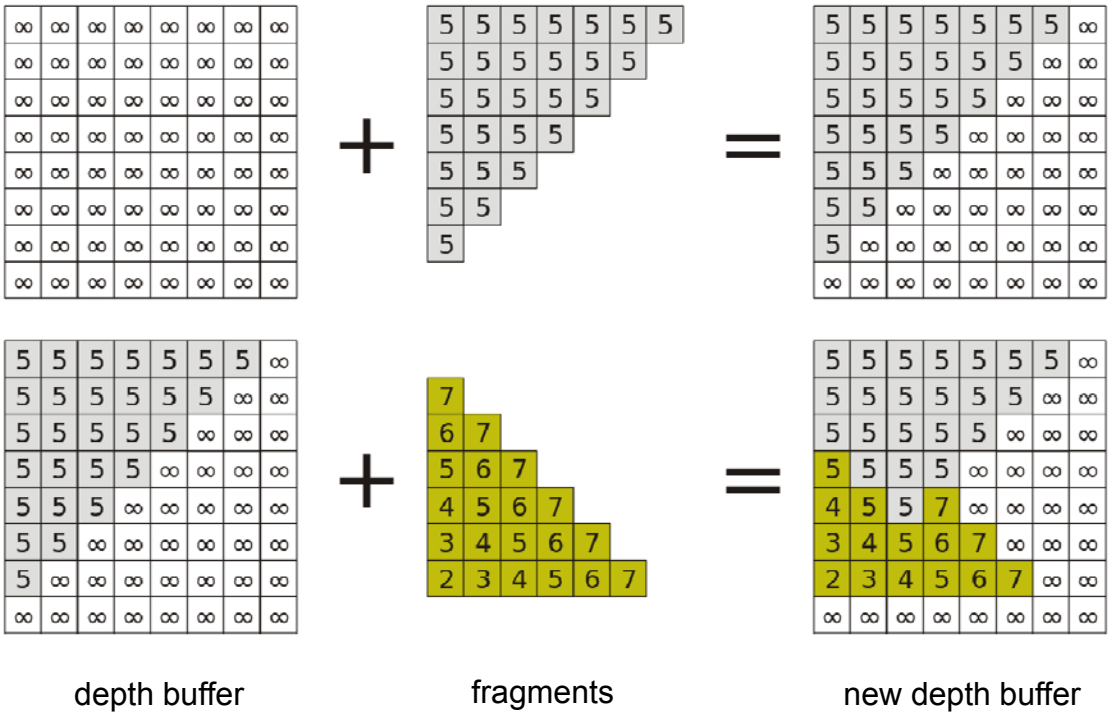
https://de.wikipedia.org/wiki/Z-Buffer

Princip paměti hloubky (2)

- Pro každý pixel je uložena hloubka naposledy nakresleného fragmentu
- Fragment se nakreslí, jen když je blíž
- Pro přehlednost s celočíselnou hloubkou (namísto 0.0 .. 1.0)

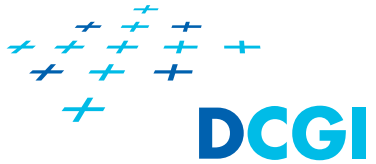
Rendering order of triangles





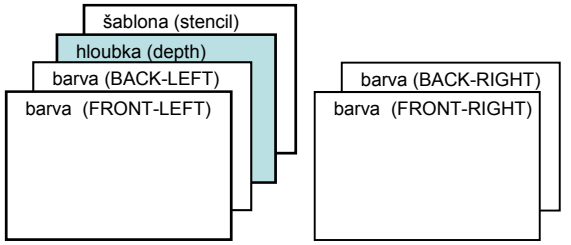
https://de.wikipedia.org/wiki/Z-Buffer

Paměť hloubky – depth buffer (3)



Hloubka (depth)

- Vzdálenost pixelu od stínítka <0,1>
- Near je 0.0, far je 1.0
- Má obvykle 24 bitů na pixel



- Nastavení v programu při výběru kontextu

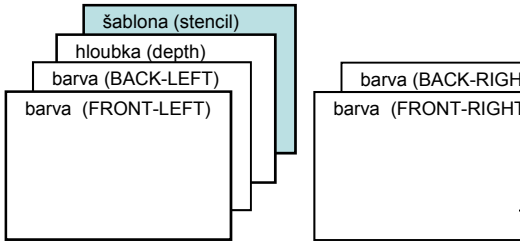
```
glutInitDisplayMode(…  | GLUT_DEPTH | …);
glEnable( GL_DEPTH_TEST ); // zapnutí hloubkového testu
glDepthMask( GL_TRUE );      // povolení aktualizace
```

- Použití – viz dále

```
glClear( ….| GL_DEPTH_BUFFER_BIT | … ); // smazání
DrawObjectsInTheScene();
…
```

Paměť šablony – stencil buffer

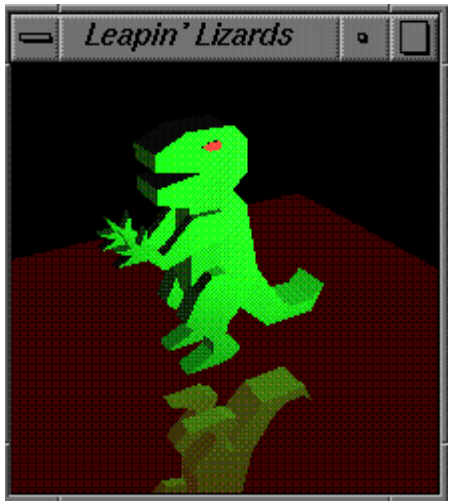
Šablona (stencil)



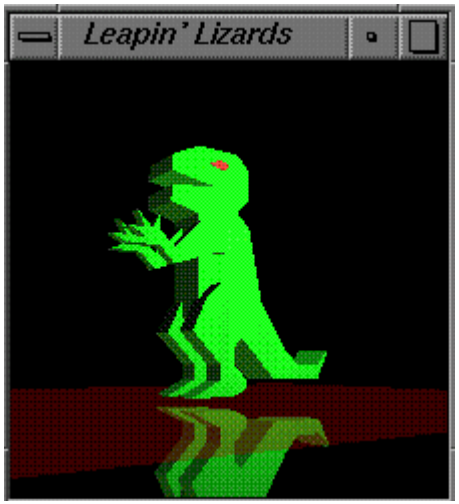
- lze označit pixely, kam se smí / nesmí kreslit
- jedna či více bitových rovin (bitplane)
- pomocí masky se zvolí bitová rovina
- přímo se nekreslí (dle paměti barvy)
- používá se ve víceprůchodových algoritmech pro speciální efekty:
  - obtisky (decals),
  - získání obrysu (outlining),
  - odrazy (v zrcadle, ve vodě),
  - a zobrazování CSG modelů (constructive solid geometry).
- Typicky 8 bitů na pixel

Šablona – stencil a zrcadlový odraz

- Nakreslí se
  1. Zrcadlo do šablony
  2. Převrácená scéna (opačný test hloubky, CW front triangles)
  3. Původní scéna



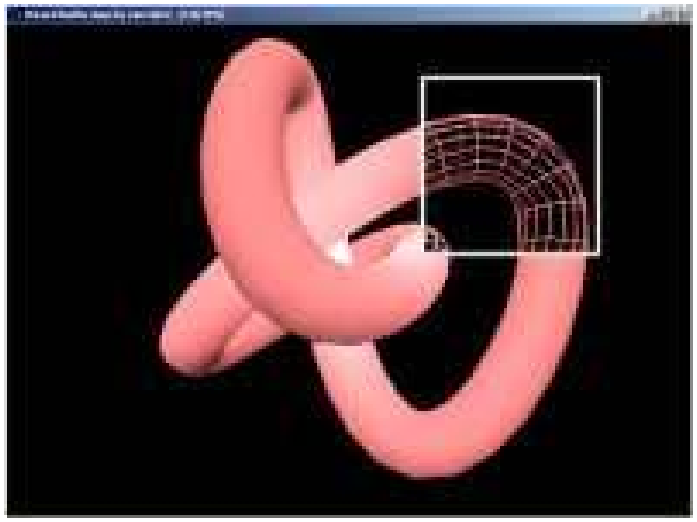
Převrácená scéna



Bez šablony

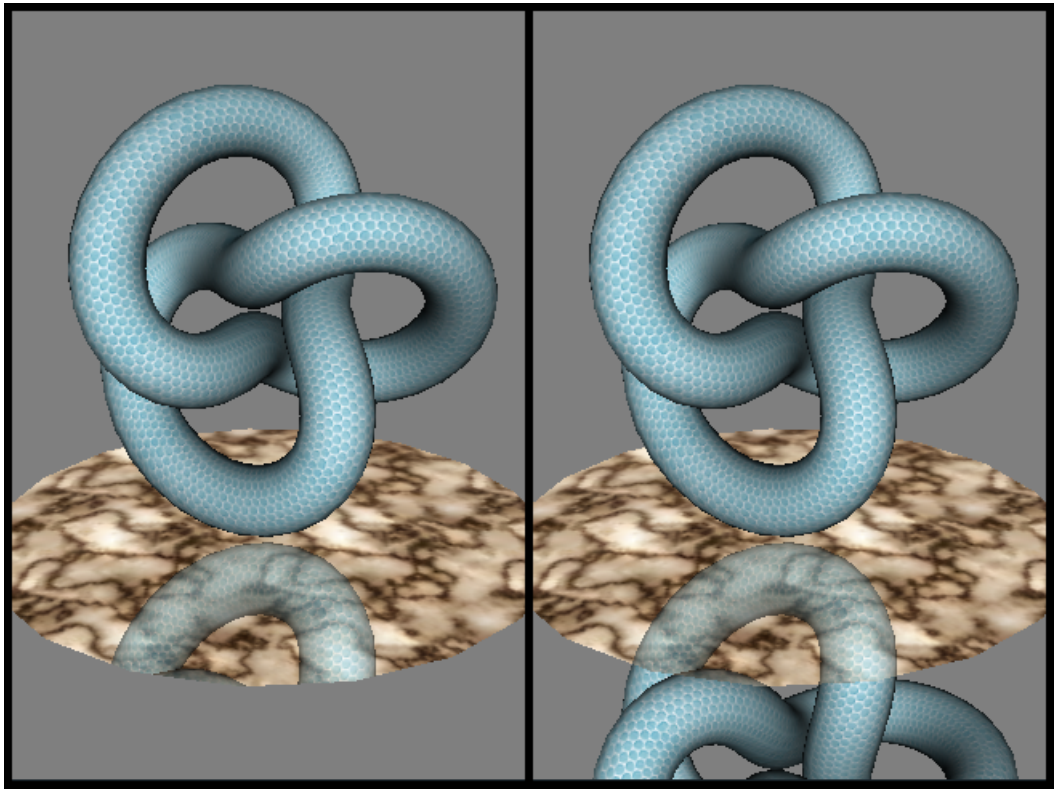
Omezeno šablonou

Šablona - stencil



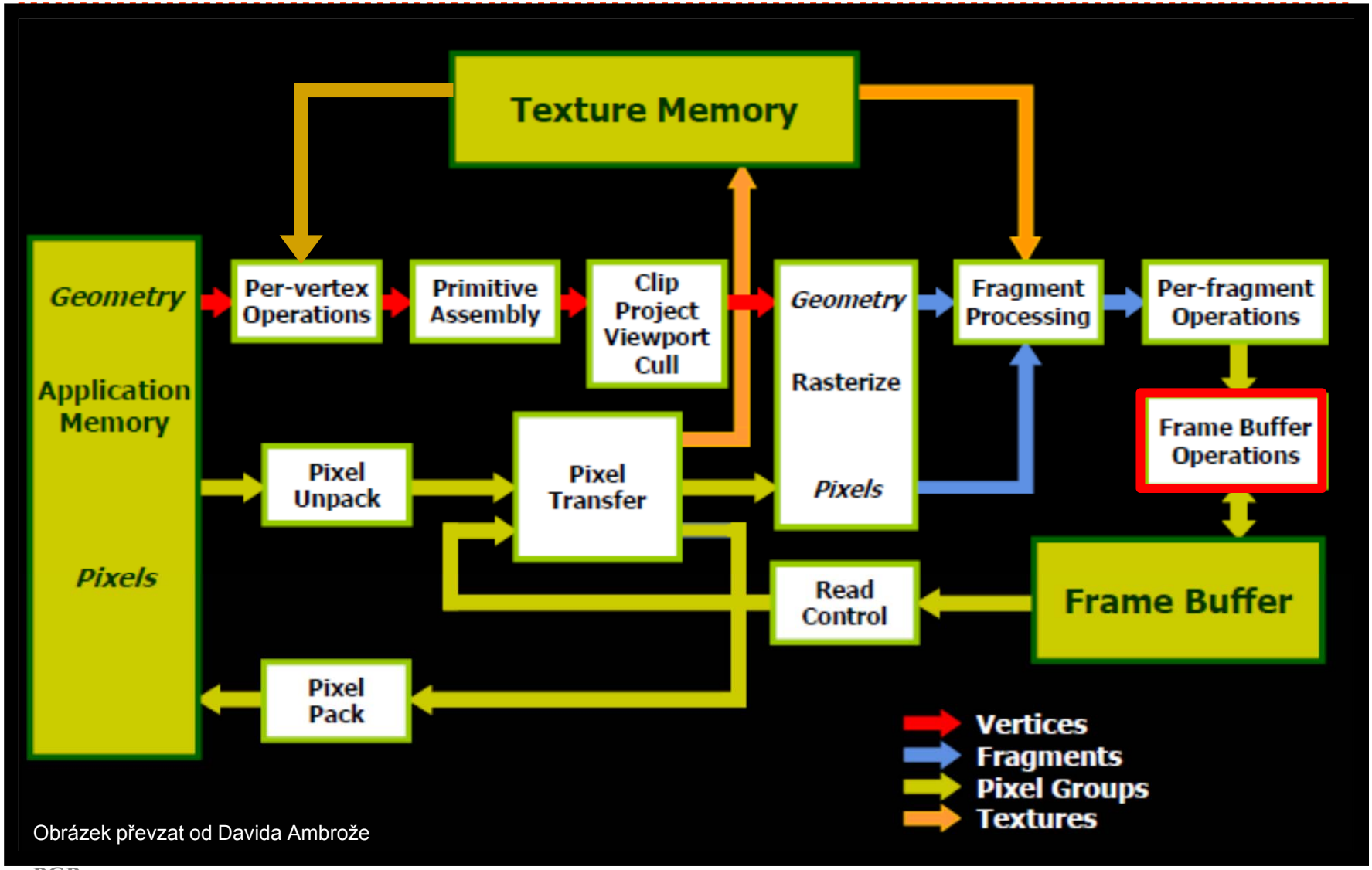
[Sulaco]

Se šablonou bez šablony



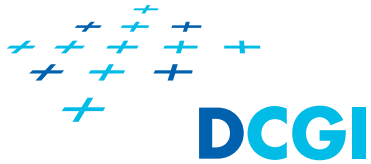
[O'Reilly]

Operace s celou pamětí obrazu



Obrázek převzat od Davida Ambrože

Počet bitových rovin - bitplanes



Inicializace paměti obrazu () v GLUTu

```
glutInitDisplayMode(  GLUT_RGB     // only RGB, no alpha
                    | GLUT_DEPTH   // depth buffer
                    | GLUT_STENCIL // stencil buffer
                    | GLUT_DOUBLE  // double buffer(front + back)
                   );
```

```
Příklad na notebooku:
Color buffer:          [r,g,b,a] = [8, 8, 8, 8] bits
Depth buffer:          depth = 24 bits
Stencil buffer:        8 bits
Auxiliary color buffers: 4
Double buffer:           supported
Stereo:                  not supported
```

Mazání pamětí (buffers) pro vykreslování

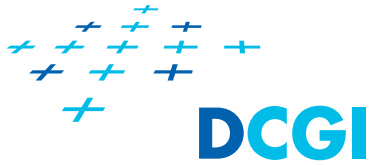
Smazání vybraných pamětí

```
void glClear( GLbitfield mask );
```

mask – které paměti smazat

- GL_COLOR_BUFFER_BIT,
- GL_DEPTH_BUFFER_BIT,
- GL_STENCIL_BUFFER_BIT,

Mazání pamětí



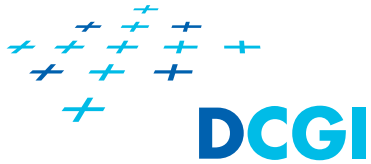
Nastavení „mazací“ hodnoty pro jednotlivé roviny

```
void glClearColor( GLclampf red,
                   GLclampf green,
                   GLclampf blue,
                   GLclampf alpha );
void glClearDepth( GLclampd h );
void glClearStencil( GLint s );
```

Hloubka ***h*** je z intervalu <0,1>. Implicitní hodnota ***h*** = 1.0 ~ zFar

Ostatní parametry mají implicitní hodnotu 0, resp 0.0f.

Maskování zápisu do obrazové paměti



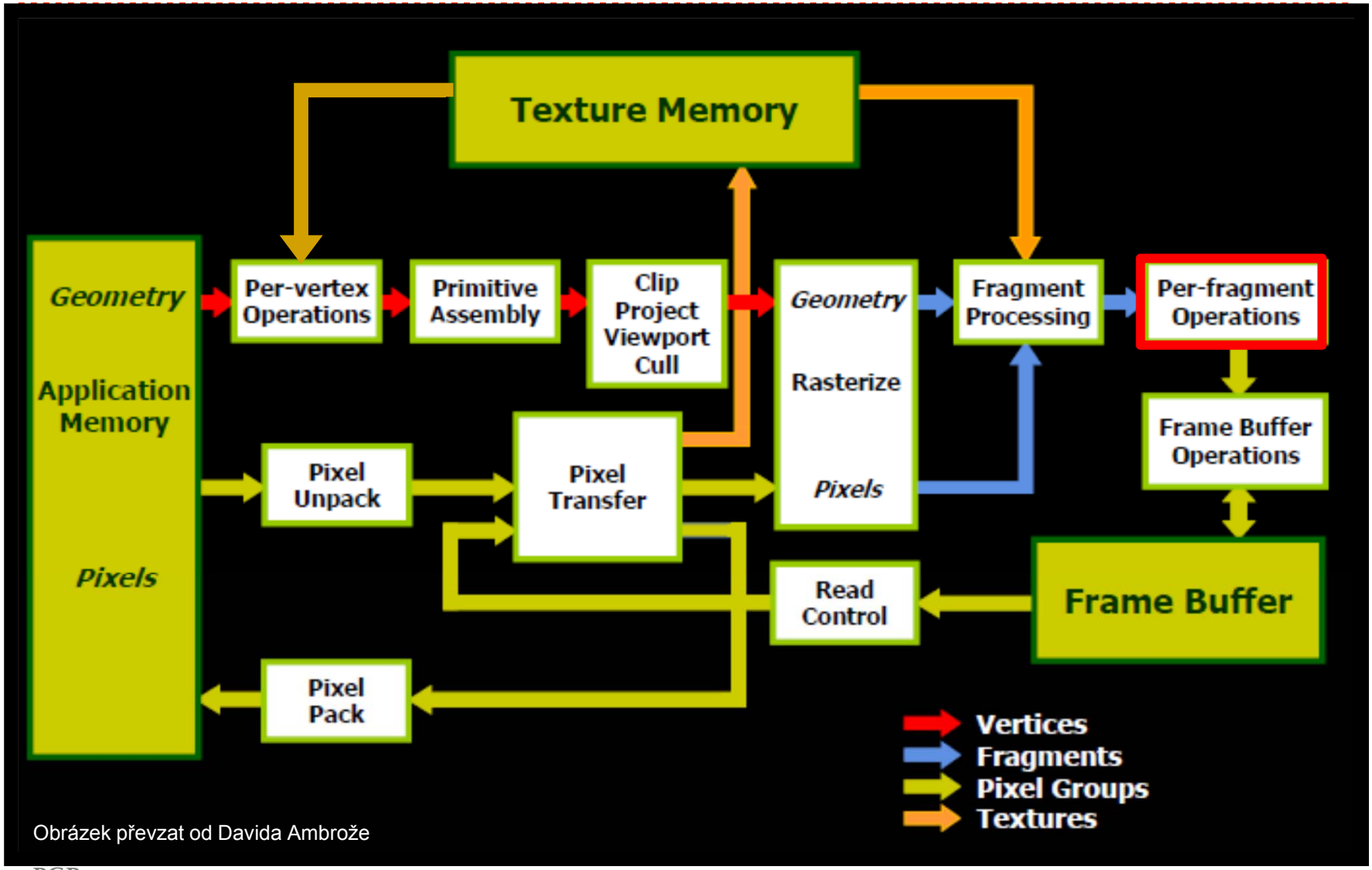
Do paměti se zapíše, jen pokud je to povoleno maskou

Nastavení masky (true zapisuje, false ne)

```
void glColorMask( GLboolean red,
                  GLboolean green,
                  GLboolean blue,
                  GLboolean alpha );
void glDepthMask( GLboolean flag );
void glStencilMask( GLuint mask );
```

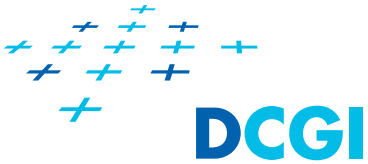
Implicitně všechny na GL_TRUE a GLuint na samé jedničky

Testy a operace s jednotlivými fragmenty



Obrázek převzat od Davida Ambrože

Testování fragmentů



Po rasterizaci víme, které **fragmenty** vznikly, jakou mají barvu, hloubku, případně i další informace - normálu, vektor ke světlu,…

(fragmenty = možné budoucí pixely s hloubkou, barvou,…)

Fragment musí projít ještě řadu testů a operací

…. na konci je obrazová paměť (color, depth, stencil)

Test

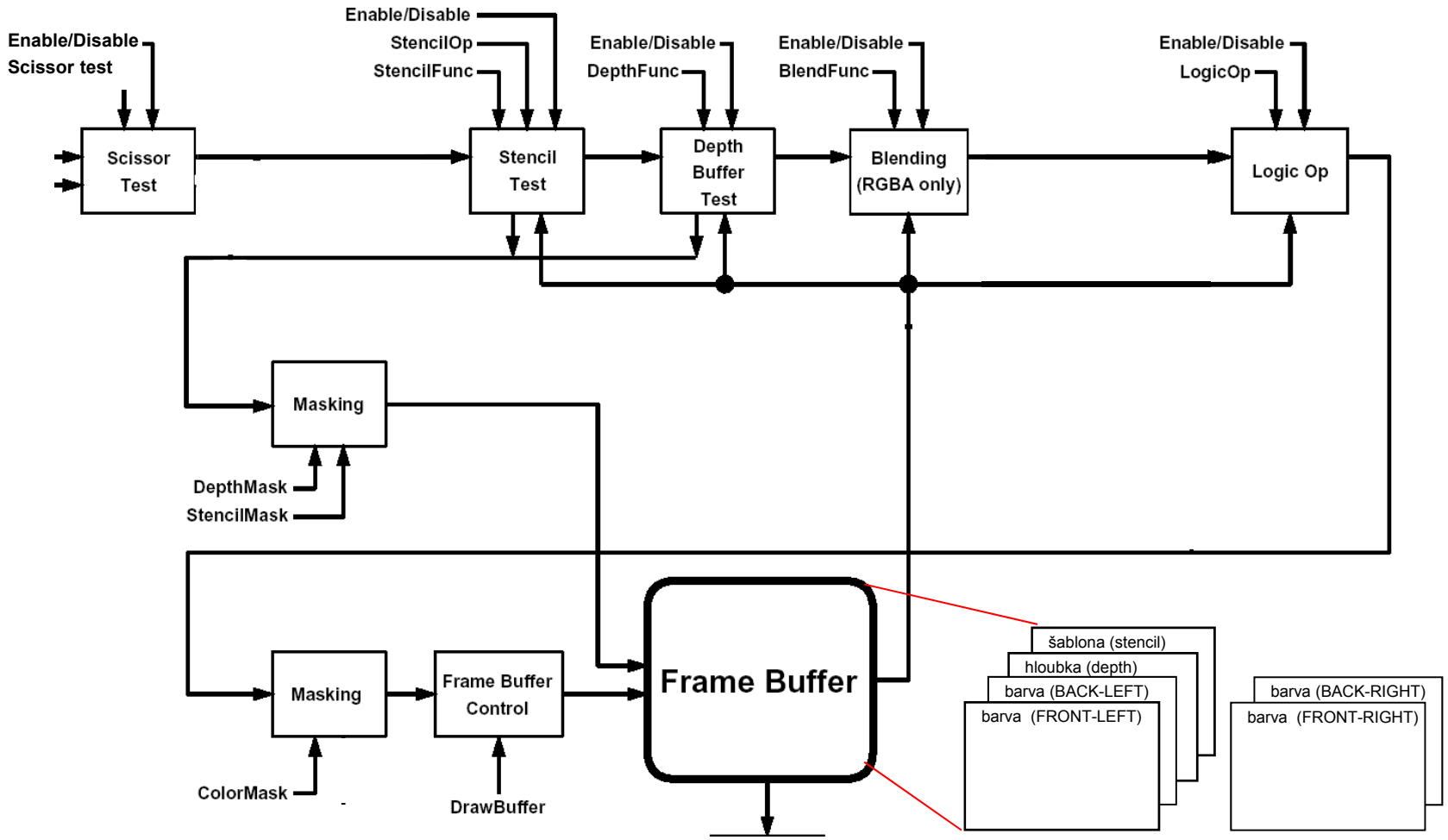
– povolí či zakáže daný fragment

– může změnit hodnotu příslušné paměti

test hloubky – aktualizuje hloubku pixelu v paměti hloubky

test šablony – změní hodnotu v šabloně

Testování fragmentů a operace s fragmenty



Testování fragmentů a operace s fragmenty

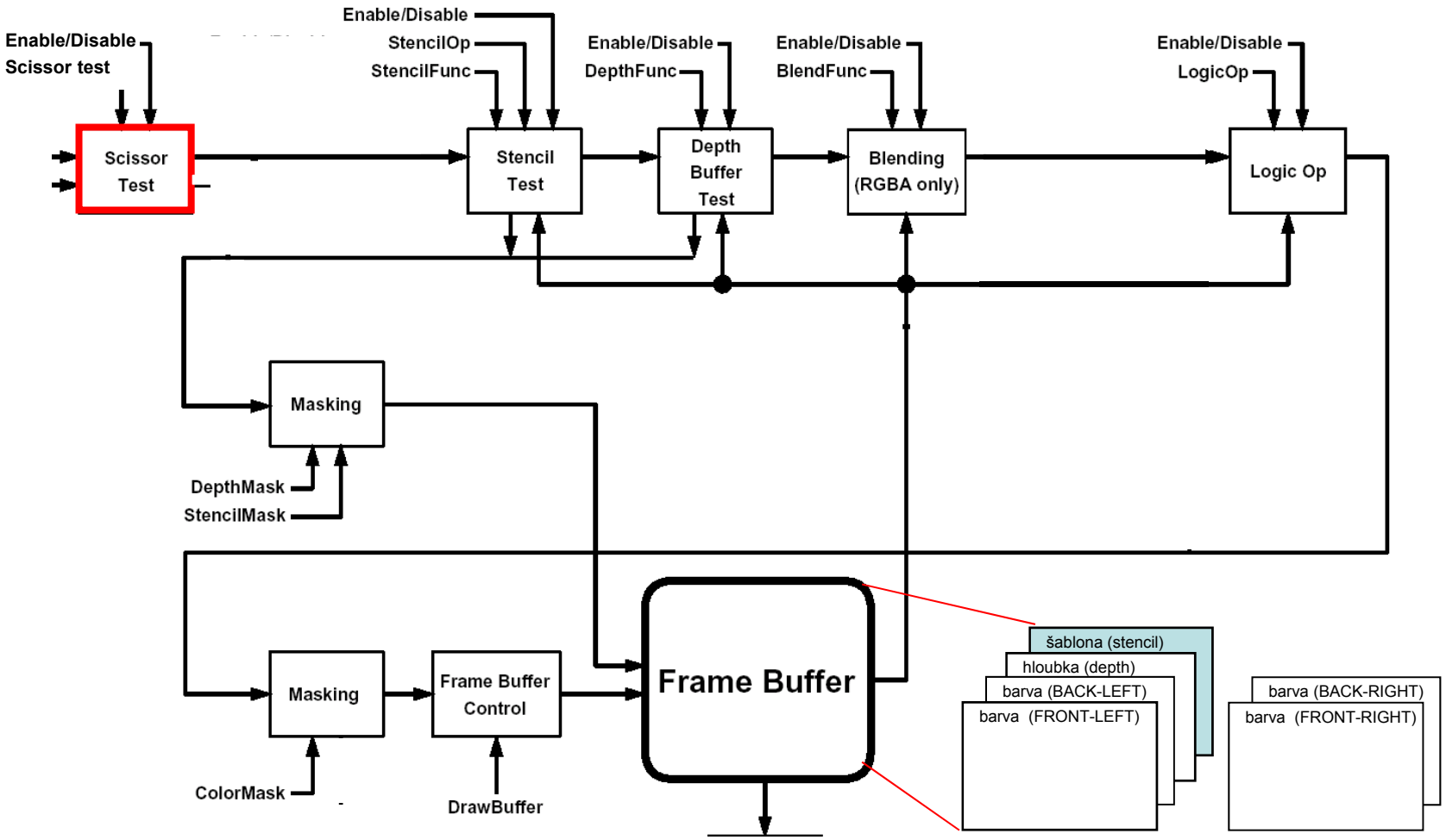
1. Výstřižek (Scissor test)
2. Test šablony (Stencil test)
3. Test hloubky (Depth test)

} 3 testy

4. Míchání (Blending)
5. Logické operace

} 2 operace

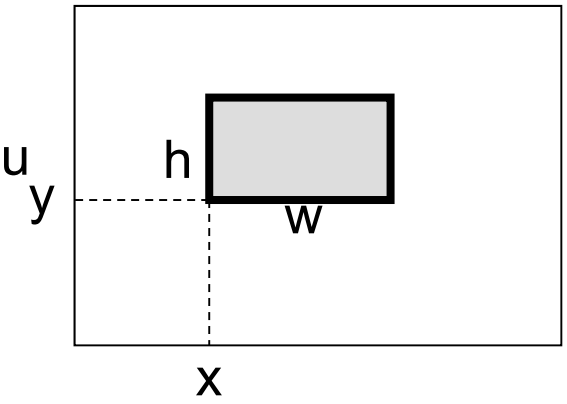
1. test: Výstřižek (scissor test) (1)



1. test: Výstřižek (scissor test) (2)

Kreslení pouze do obdélníkové oblasti okna

- rychlá verze šablony (stencil)
- vhodná k aktualizaci změněné části obrazu



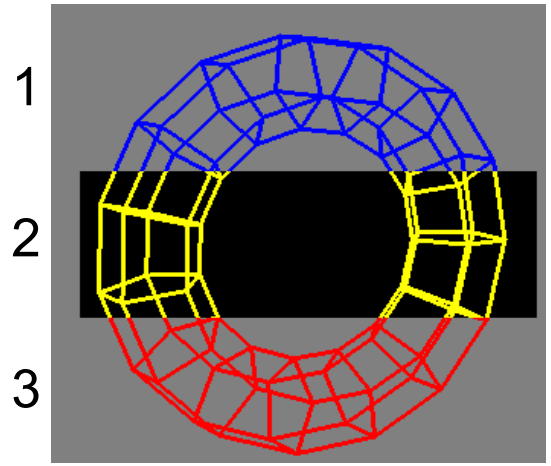

```
void glScissor( GLint x,   GLint y,
                GLsizei w,   GLsizei h );
```

x, y dolní levý roh

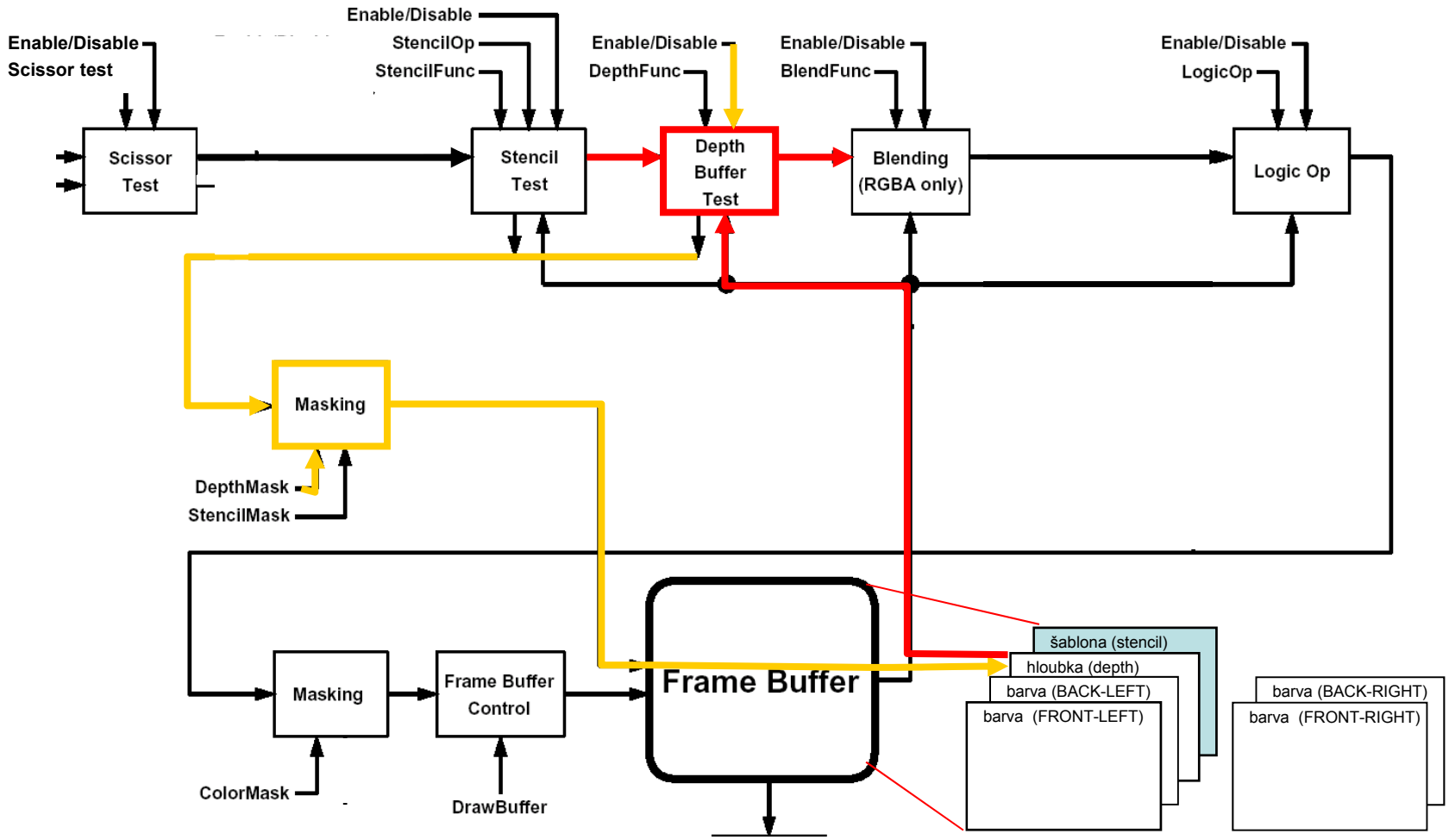
w, h šířka a výška

```
glEnable( GL_SCISSOR );
```

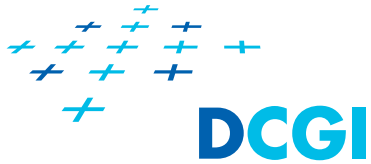
Tři různé výstřižky



3. test – Hloubkový test (depth test) (1)



3. test – Hloubkový test (depth test) (2)



Pro každý pixel uložena vzdálenost ke stínítku

- rozsah jako složky barvy ⟨**0, 1**⟩
- zNear => 0, zFar => **1** !!!

- Hloubkový test určí viditelnost fragmentu

```
if( depth_test_passes )
        propusť pixel a nahraď hloubku novou hodnotou
```

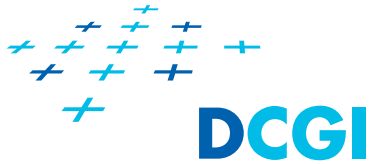
- Nastavení testovací funkce

void **glDepthFunc**(GLenum *func*);

func – funkce pro porovnávání (z fragmentu s uloženou z pixelu)

GL_NEVER, **GL_LESS**, GL_EQUAL, GL_LEQUAL, GL_GREATER, GL_NOTEQUAL, GL_GEQUAL, a GL_ALWAYS

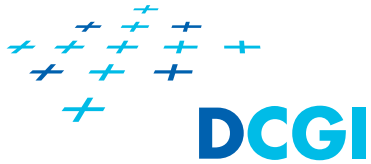
3. test – Hloubkový test (depth test) (3)



Režimy paměti hloubky (Z-buffer)

- Test vypnut `glDisable( GL_DEPTH_TEST );`
- Test zapnut `glEnable( GL_DEPTH_TEST );`
  - povolena aktualizace zBufferu `glDepthMask( GL_TRUE );`
  - zakázána aktualizace zBufferu `glDepthMask( GL_FALSE );`

3. test – Hloubkový test (depth test) (4)

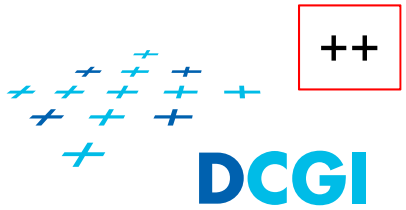


Pro každý pixel uložena vzdálenost k oku (ke kameře)

```
if(enabled)
    if( test depth passed )
        pass the fragment
        if( mask )
           update Z-buffer
    else // depth failed
       discard the fragment

else // disabled
     pass the fragment
     do not update Z-buffer  // ignore the depth mask
```

3. test – Hloubkový test (depth test) (5)

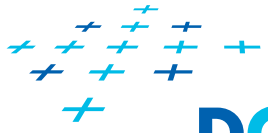


void **glDepthRange**(GLclampd zNear, GLclampd zFar);

- defines the linear mapping of NDC with window space z
  - normalized device coordinates NDC z $\in\langle -1, 1\rangle$
  - window space coordinates $z_{window} \in\langle \mathbf{0, 1}\rangle$ – range as colors
- Standard settings:
  - zNear => 0, zFar => **1** !!!
- Special usage:
  - When using the GL_EQUAL and GL_NOTEQUAL depth comparisons in order to reduce the number of available values
  - Selectively clear the depth buffer behind object (e.g. mirror)

```
gLDepthRange(1,1);              // selective depth clear
gLDepthFunc(GL_ALWAYS);  // pass all – ignore depth
drawObject();
```

Example: Selectively clear the depth buffer (e.g. in place of mirror)



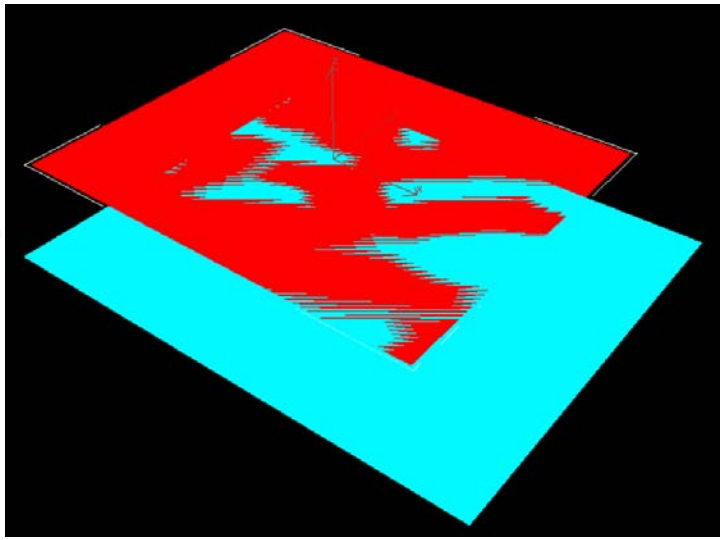
DCGI

++

1. Set mask values to 1 in place of visible parts of the mirror
2. Selectively clear Z behind reflections - where (stencil == 1)

```
gLColorMask(0,0,0,0); // draw no color
gLDepthRange(1,1);    // selective depth clear
gLDepthFunc(GL_ALWAYS);  // pass all fragments – ignore depth
gLStencilFunc(GL_EQUAL, 1, ~0);  // visible fragments of the mirror
glStencilOp(GL_KEEP, GL_KEEP,GL_KEEP);  // keep the stencil
drawReflections();  // just clear the depth behind
```

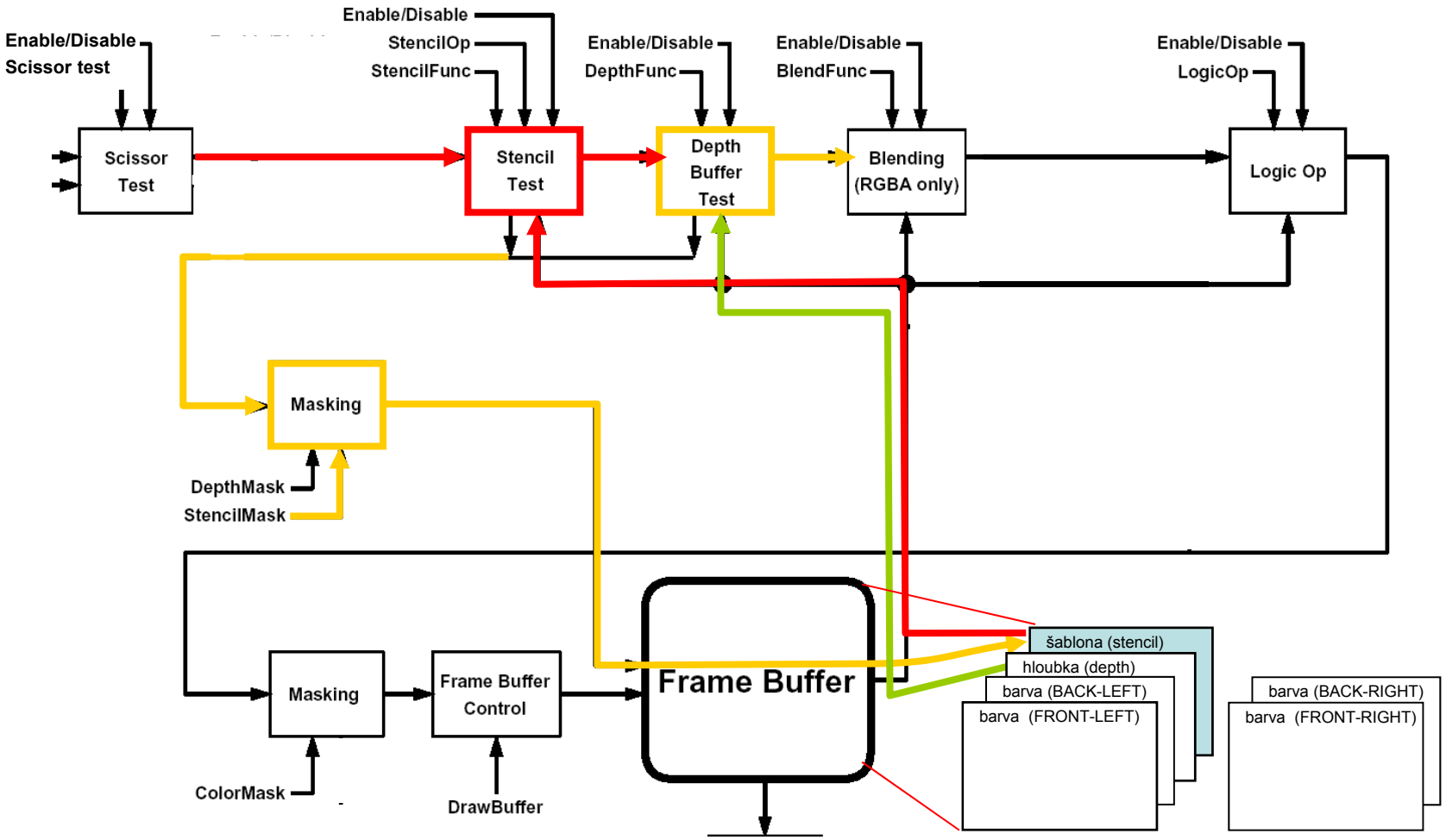
Z-fighting



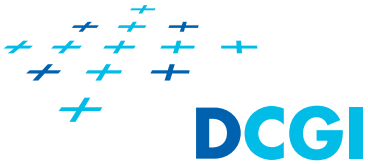
- Two ore more primitives have similar values in Z-buffer (precision of n-bits)
- Typical for coplanar polygons
- Fragments are rendered randomly from one or another – who wins the z test on that particular position
- Overall effect
  - Flickering when moving
  - Noisy rasterization – polygons “fight” to color the pixel
- Reduction
  - Set the near plane farther away from camera
  - Increase size of z-buffer ~~8-bit~~, (16-bit), 24-bit or 32-bit Z-buffer
  - Z-buffer offset in screen space (after transformations)
  - Stencil buffer (do not draw 2nd polygon to the fragment of the 1st one)

https://en.wikipedia.org/wiki/Z-fighting

2. test: Šablona (stencil test) (1)



2. test: Šablona (stencil test) (2)



Stencil test

- Porovná *referenční hodnotu ref s hodnotou pixelu v šabloně* s
  Podle výsledku porovnání (true / false)
  - propustí či nepropustí fragment - **glStencilFunc()**
  - a modifikuje obsah šablony (3) - **glStencilOp()**
  - porovnává jen bity které mají 1 v *masce mask*

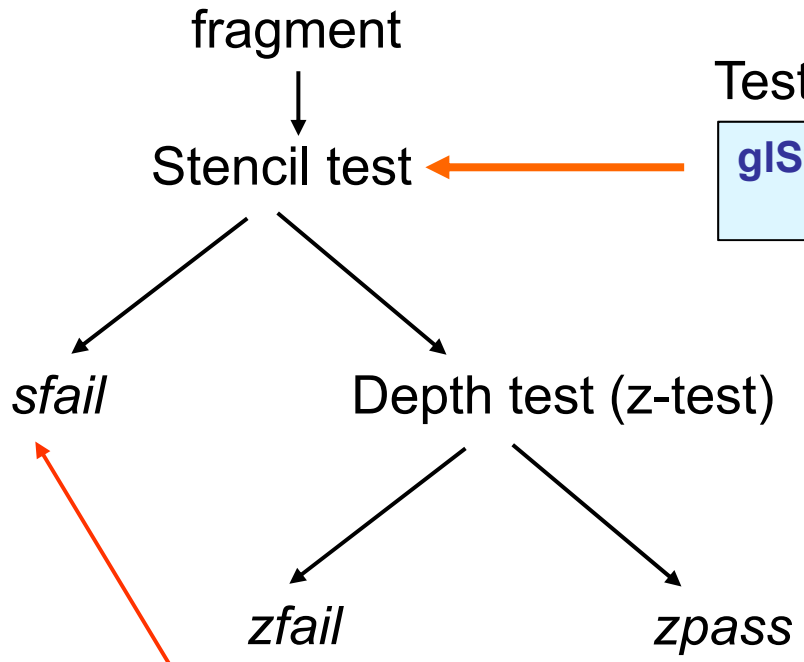
- Modifikace závisí i na testu hloubky (*depth test*)

  Tři různé kombinace -> tři modifikující operace
  - *fail* porovnání šablony neuspělo
  - *zfail* porovnání šablony uspělo, neuspěl test hloubky (vzadu, zakryto)
  - *zpass* porovnání šablony uspělo, uspěl test hloubky (nebo je zakázán) (vpředu)

- **glEnable(GL_STENCIL_TEST);**
  - Povolí test i modifikaci šablony
- **glStencilMask(GLuint mask);**
  - Maskování jen některých bitů

2. test: Šablona (stencil test) (3)

- Funkcionalita nad fragmenty je následující:



Testovací funkce a maska:

```
glStencilFunc(GLenum func,
              GLint ref, GLuint mask);
```

Aktualizace stencil bufferu:

```
glStencilOp(sfail, zfail, zpass);
```

2. test: Šablona (stencil test) (4)

void glStencilFunc(GLenum *func*, GLint *ref*, GLuint *mask*);

Nastavuje porovnávání

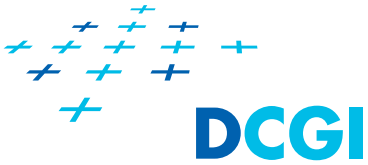
- porovnávací funkci ***func***
  - GL_NEVER, GL_LESS, GL_LEQUAL, GL_GREATER, GL_GEQUAL, GL_EQUAL, GL_NOTEQUAL a **GL_ALWAYS**
  - např.: GL_LESS => if(***ref*** < pixelStencilValue) then true
- referenční hodnotu ***ref*** *(default 0) = konstanta*
  - S ní se porovnává, nebo se zapíše do masky
- masku ***mask*** (samé 1)
  - dolních *s* bitů odpovídá *s* bitům šablony
  - bitový AND (&) určí bitové roviny k porovnání
  - např.: GL_LESS =>

    if(***ref*** & ***mask***) < (stencil & ***mask***) then true

přesněji

Porovnávací funkce GL_LESS, GL_GREATER atd.

2. test: Šablona (stencil test) (5)

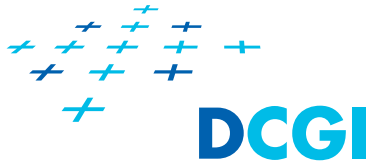


void glStencilOp(GLenum *fail*, GLenum *zfail*, GLenum *zpass*)

Nastavuje modifikující operace pro hodnoty ve stencil buffer

- funkce modifikující data ve stencil bufferu
  - ***fail*** porovnání šablony neuspělo
  - ***zfail*** porovnání šablony uspělo, neuspěl test hloubky
  - ***zpass*** porovnání šablony uspělo, uspěl test hloubky (nebo je zakázán)
- **GL_KEEP** (nemění šablonu)
  GL_ZERO (nastaví na 0)
  GL_REPLACE (nastaví na ***ref*** zadané **glStencilFunc**)
  GL_INCR, GL_INCR_WRAP (inkrementace šablony)
  GL_DECR, GL_DECR_WRAP (dekrementace šablony)
  GL_INVERT (inverze bitů)

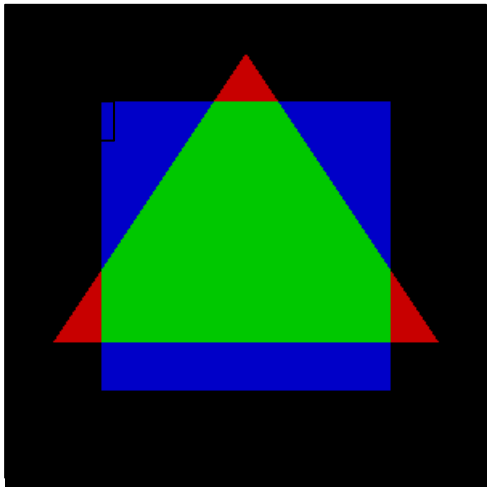
2. test: Šablona (stencil test) (6)



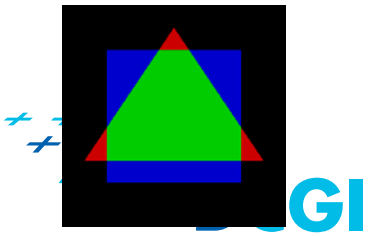
Příklad kreslení vymezené trojúhelníkem

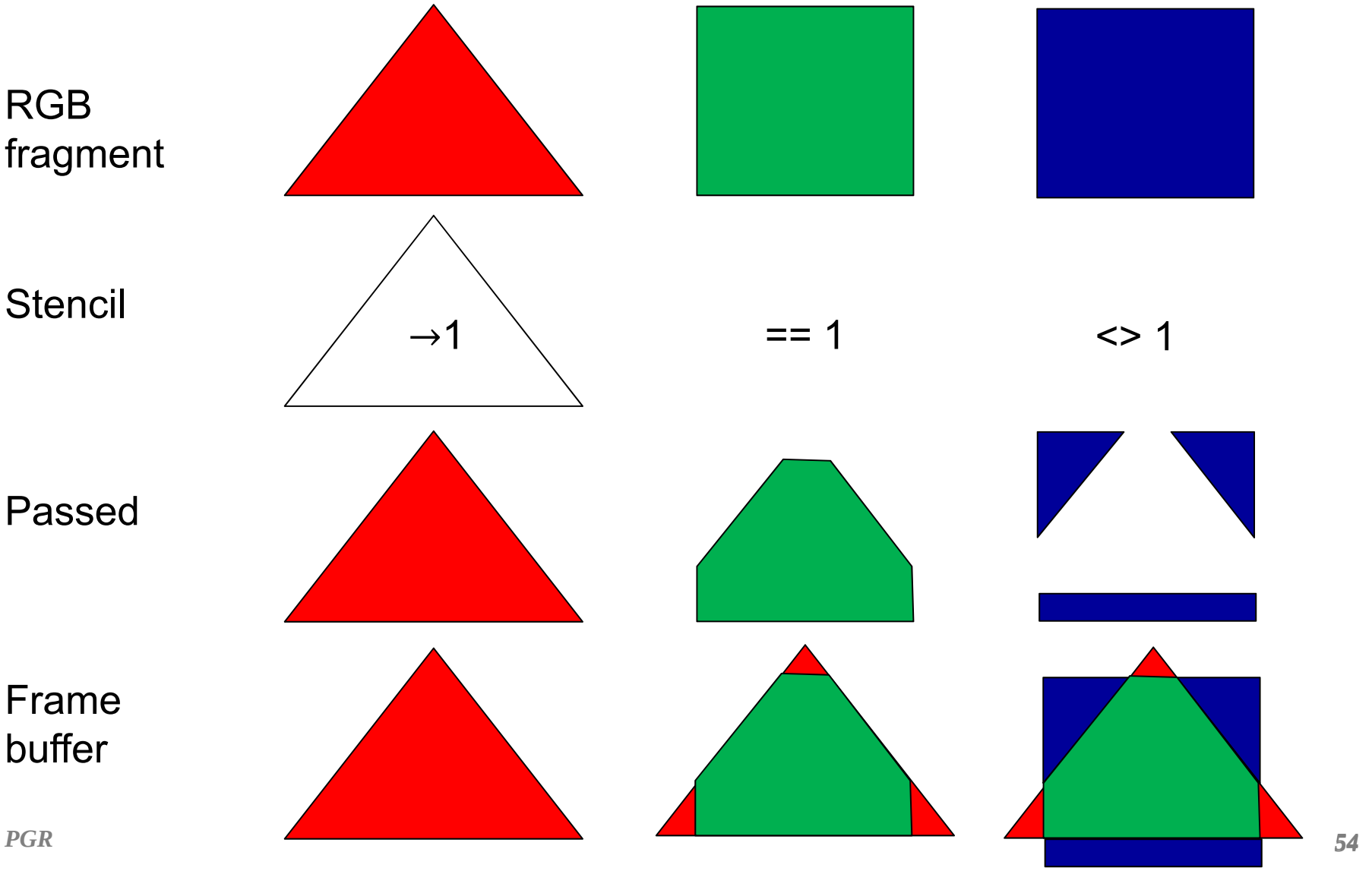
- Červený trojúhelník se nakreslí a přitom naplní šablonu 1
- Pak se kreslí zelený čtverec pouze tam, kde je v šabloně 1 (zbyde z něj jen zelená část bez rohů)
- Pak se kreslí modrý čtverec pouze tam, kde je v šabloně 0 (zbydou z něj jen modré rohy)

```
glStencilFunc(GL_ALWAYS, 1, 1);          zpass
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
  drawRedTriangle();          ref  mask
glStencilFunc(GL_EQUAL, 1, 1);          zpass
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
  drawGreenPolygon();
glStencilFunc(GL_NOTEQUAL, 1, 1);
  drawBluePolygon();
```



2. test: Šablona (stencil test) (6a)





2. test: Šablona (stencil test) (7)

Příklad – **kreslení do kosočtverce a mimo něj**

hodnota 1 pro replace
rovina č. 1 (maska)

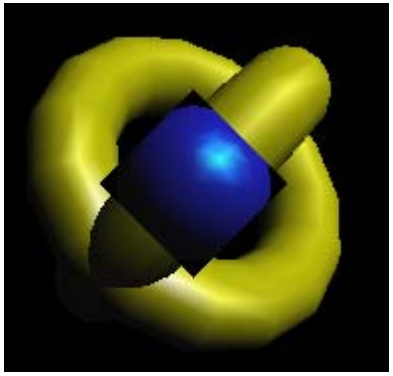
1. inicializovat

```
glClearStencil(0x0);
glEnable(GL_STENCIL_TEST);
```

2. naplnit šablonu jedničkami (kosočtverec na obrázku)
   - vynulovat
   - nastavit
   - nakreslit tvar

```
glClear(GL_STENCIL_BUFFER_BIT);
glStencilFunc (GL_ALWAYS, 0x1, 0x1)
glStencilOp(…,GL_REPLACE,GL_REPLACE);
drawStencil(); // kosočtverec, tvar šablony
```

3. Kreslit scénu



```
glStencilOp(GL_KEEP, GL_KEEP,GL_KEEP);
glStencilFunc(GL_EQUAL, 0x1, 0x1);  // 1
drawBlueSphere() // uvnitř šablony

glStencilFunc(GL_NOTEQUAL, 0x1, 0x1);  // 0
drawToriScene(); // mimo šablonu
```

glStencilOp(GLenum *fail*, GLenum *zfail*, GLenum *zpass*)

2. test: Šablona (stencil test) (8)

Vyplnění děr v uzavřených tělesech po oříznutí rovinou
(„uzavření tělesa“, aby nebylo vidět dovnitř)

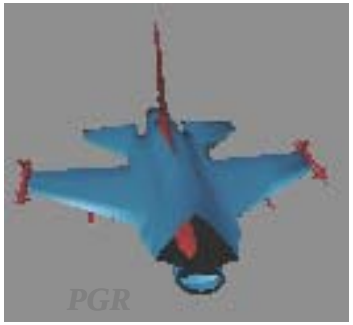
- musí mít sudý počet přivrácených a odvrácených stěn -> XOR
  - vynulovat šablonu

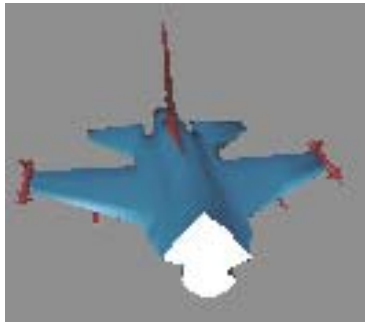
```
glClearStencil(0x0);
glClear(GL_STENCIL_BUFFER_BIT);
```

  - povolit update, invertovat (1x bude 1, 2x bude 0, 3x bude 1,…)

```
glEnable(GL_STENCIL_TEST);
glStencilFunc (GL_ALWAYS, …, 0x1);
glStencilOp (…, GL_INVERT, GL_INVERT);
```

  - dokreslit „čepice“ (všechny najednou – 1 velký bílý obdélník)





```
glStencilFunc (GL_NOTEQUAL, 0x0, 0x1);
glStencilOp( GL_KEEP, GL_KEEP,GL_KEEP);
drawLargeRectangleCap();
```

glStencilOp(GLenum *fail*, GLenum *zfail*, GLenum *zpass*)

4. operace: Míchání barev (blending)

- míchání nové barvy s barvou v obrazové paměti
- v režimu RGBA
- Následuje
  - Nastavení míchací funkce
  - Pořadí vykreslování průhledných a neprůhledných objektů
  - Jak nastavit hloubkový test?

Aplikace míchání barev

sklo

Průhledný ukazatel

DP: Pavel Nemec

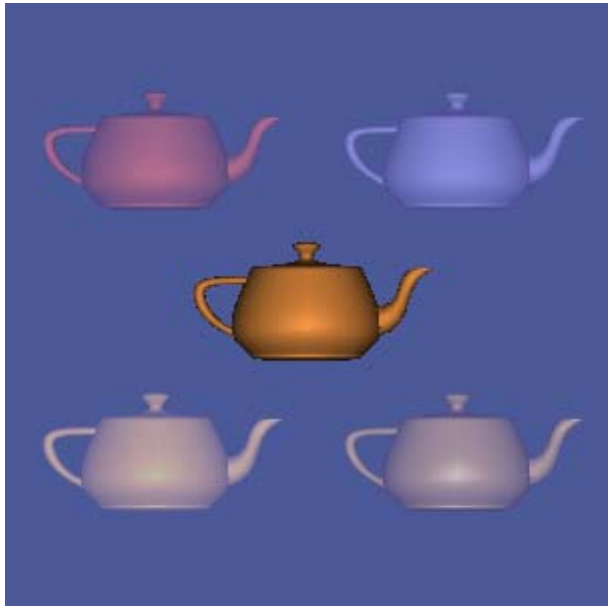
Míchání barev (Blending)

- Při **míchání barev** (α-míchání, *blending*) se kombinuje barva aktuálně vykreslovaných fragmentů (*src*) s barvou pixelů uložených v obrazové paměti (*dst*)

  $\Rightarrow$ používá se na zobrazování poloprůhledných objektů přes již vykreslenou scénu, prolínání obrazů, klíčování na barvu, apod.

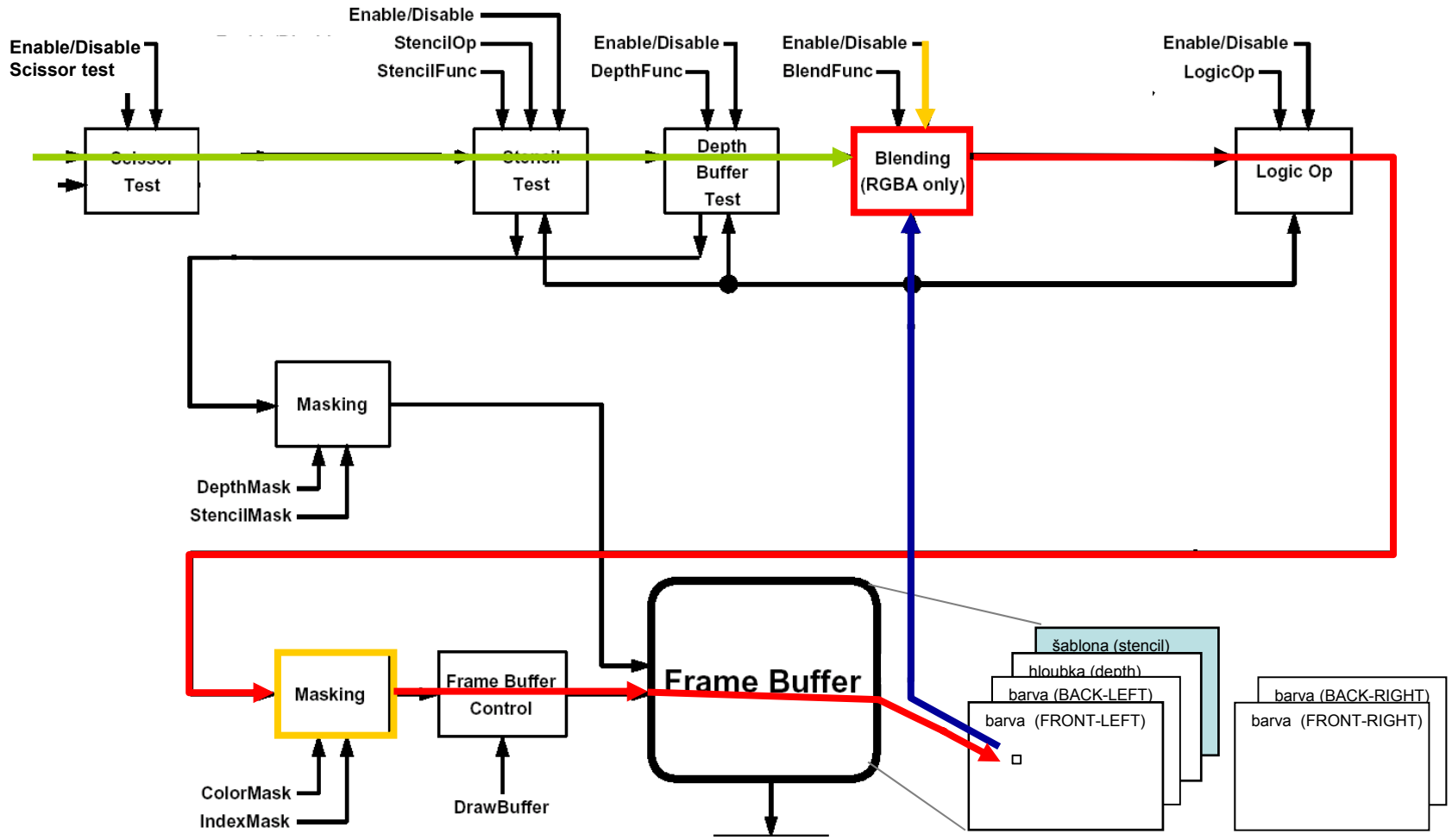
  Příklad: Pohled přes modré sklo.

  Všechny barvy se zabarvují do modra a my vidíme část původních barev a část modré

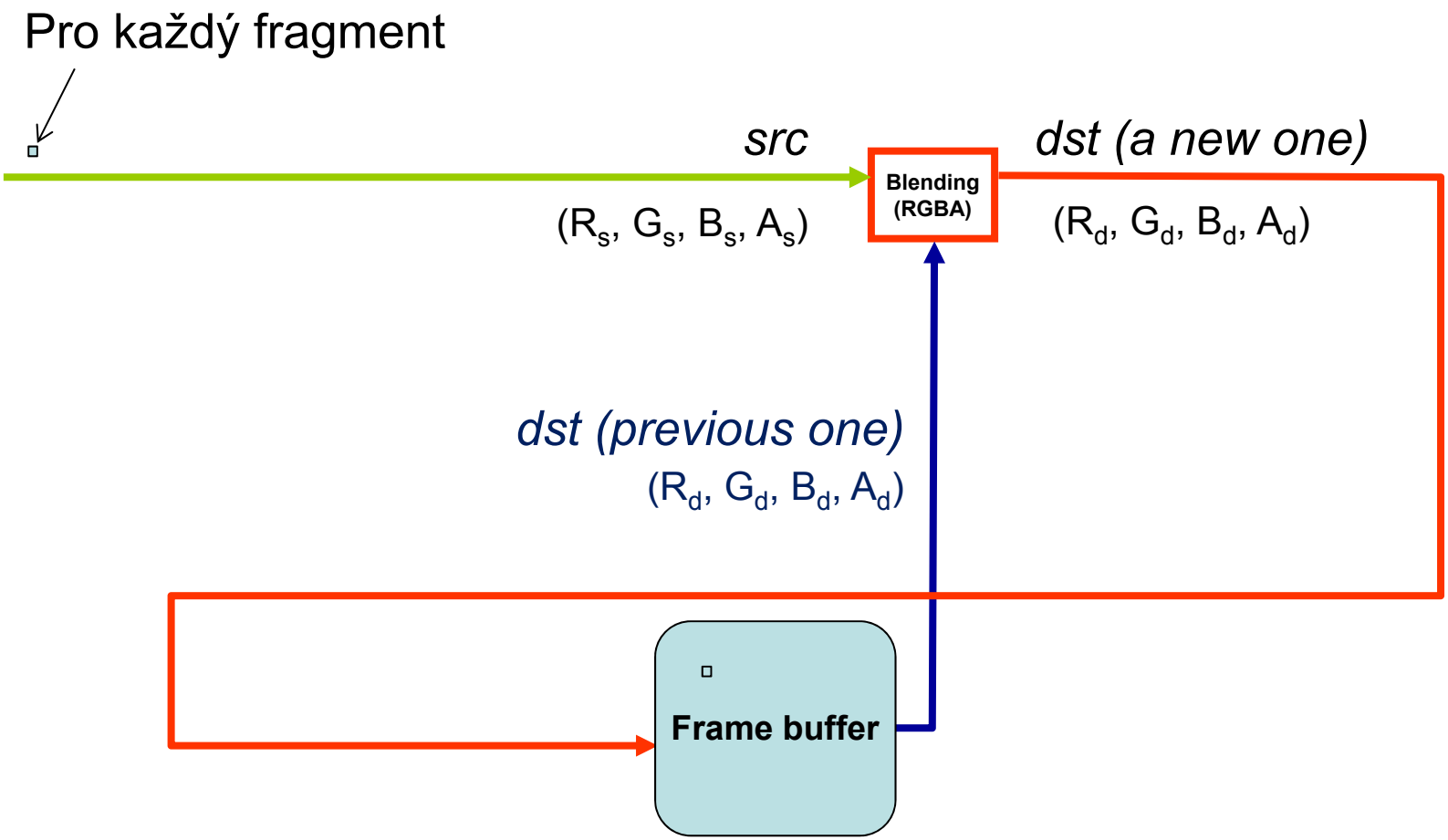


- **Zapnutí míchání** `glEnable(GL_BLEND);`
- **Vypnutí míchání** `glDisable(GL_BLEND);`

Operace míchání – 4. operace s fragmenty



Míchání barev – toky hodnot



Blok míchání barev podrobně

Do míchání vstupuje:

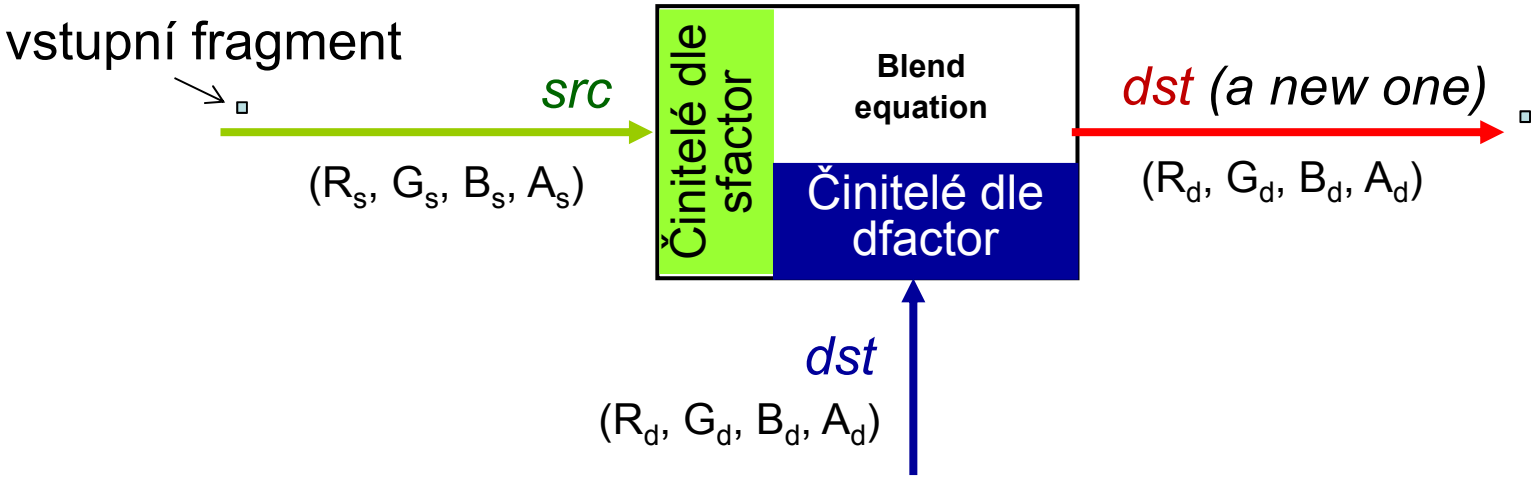
- **vykreslovaná barva** fragmentu – (R_s, G_s, B_s, A_s) – *src*
- **barva uložená v obrazové paměti** – (R_d, G_d, B_d, A_d) – *dst*

Složky vstupu se vynásobí míchacími činiteli – *sfactor* a *dfactor*

- **glBlendFunc**

A spočítá se míchací rovnice -> nová hodnota dst – ořízne se na $\langle 0, 1 \rangle$

- **glBlendEquation**

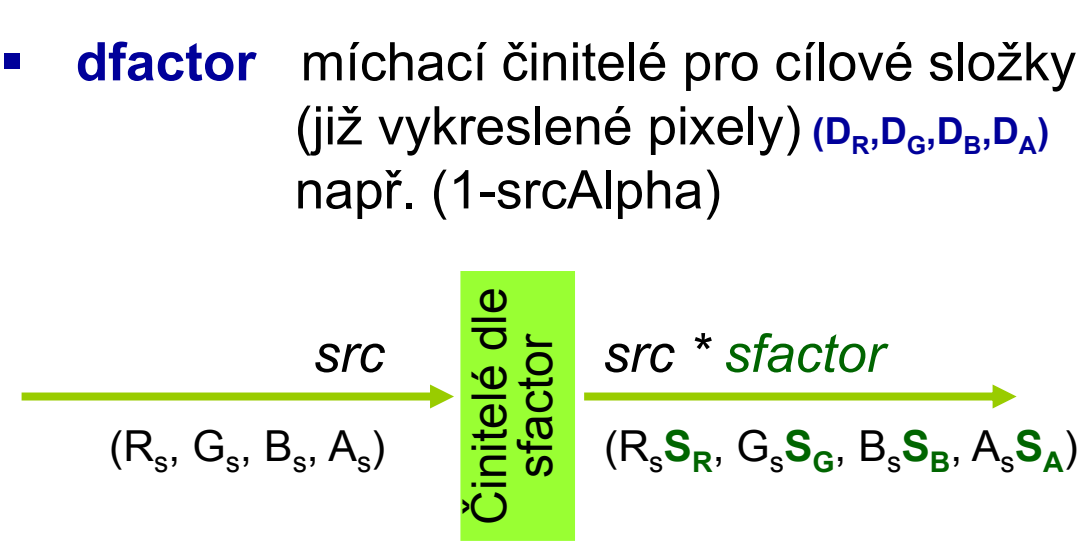


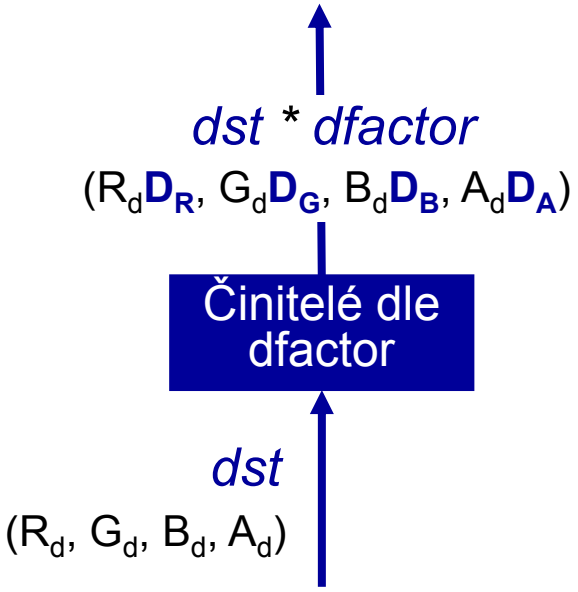
Nastavení míchacích činitelů

glBlendFunc(GLenum **sfactor**, GLenum **dfactor**);

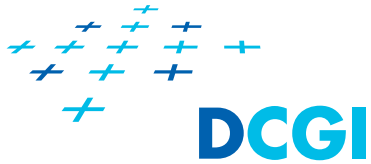
Příkaz pro „hromadné nastavení“ míchacích činitelů S a D
(platí pro všechny následně vykreslené fragmenty)

- **sfactor** míchací činitelé pro zdrojové složky - např. srcAlpha
  (právě vykreslované fragmenty) (S_R,S_G,S_B,S_A)
- **dfactor** míchací činitelé pro cílové složky
  (již vykreslené pixely) (D_R,D_G,D_B,D_A)
  např. (1-srcAlpha)





Některé míchací činitele



Příklady míchacích činitelů (S_R,S_G,S_B,S_A), (D_R,D_G,D_B,D_A):

míchací funkce	míchací činitelé (f_R, f_G, f_B, f_A)
GL_ZERO	(0, 0, 0, 0)
GL_ONE	(1, 1, 1, 1)
GL_DST_COLOR	(R_d, G_d , B_d, A_d)
GL_SRC_COLOR	(R_S, G_S, B_S, A_S)
GL_ONE_MINUS_DST_COLOR	(1, 1, 1, 1) – (R_d, G_d, B_d, A_d)
GL_ONE_MINUS_SRC_COLOR	(1, 1, 1, 1) – (R_S, B_S, B_S, A_S)
GL_SRC_ALPHA	(A_S, A_S, A_S, A_S)
GL_ONE_MINUS_SRC_ALPHA	(1, 1, 1, 1) – (A_S, A_S, A_S, A_S)
GL_DST_ALPHA	(A_d, A_d, A_d, A_d)
GL_ONE_MINUS_DST_ALPHA	(1, 1, 1, 1) – (A_d, A_d, A_d, A_d)
GL_SRC_ALPHA_SATURATE	(f, f, f, 1); f = min(A_s, 1 – A_d)

Těchto míchacích funkcí existuje ve vyšších verzích OpenGL cekem 19, navíc lze nastavit výpočet barvy RGB a Alpha odděleně

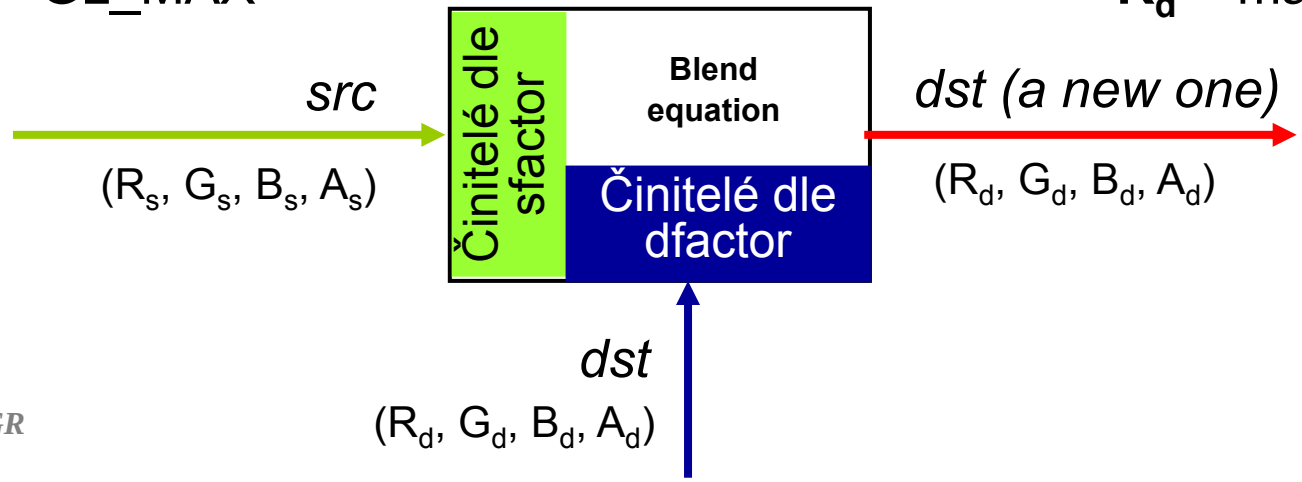
```
glBlendFuncSeparate( srcRGB, dstRGB,  srcALpha, dstAlpha);
```

Nastavení míchací rovnice

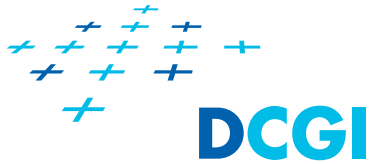
glBlendEquation(Glenum **mode);**

Příkaz pro nastavení míchací rovnice (*blend equation*)

mode	**Rovnice pro složku R**
▪ GL_FUNC_ADD (implicitně)	$R_d = R_s S_R + R_d D_R$
▪ GL_FUNC_SUBTRACT	$R_d = R_s S_R - R_d D_R$
▪ GL_FUNC_REVERSE_SUBTRACT	$R_d = R_d D_R - R_s S_R$
▪ GL_MIN	$R_d = \min(R_s, R_d)$
▪ GL_MAX	$R_d = \max(R_s, R_d)$



Nastavení míchání v různých situacích



- **glBlendEquation()**
  - Implicitně nastaveno na GL_FUNC_ADD
  - GL_FUNC_ADD vhodné na antiliasing a průhledné objekty
  - Analýza obrazových dat (prahování vůči konstantní barvě) GL_MIN a GL_MAX

- **glBlendFunc()**
  - Implicitní hodnoty sfactor = **GL_ONE**, dfactor = **GL_ZERO**
  - Průhledné objekty a antialiazing bodů a čar
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
  - Antialiasing polygonů
    glBlendFunc(GL_SRC_ALPHA_SATURATE, GL_ONE);

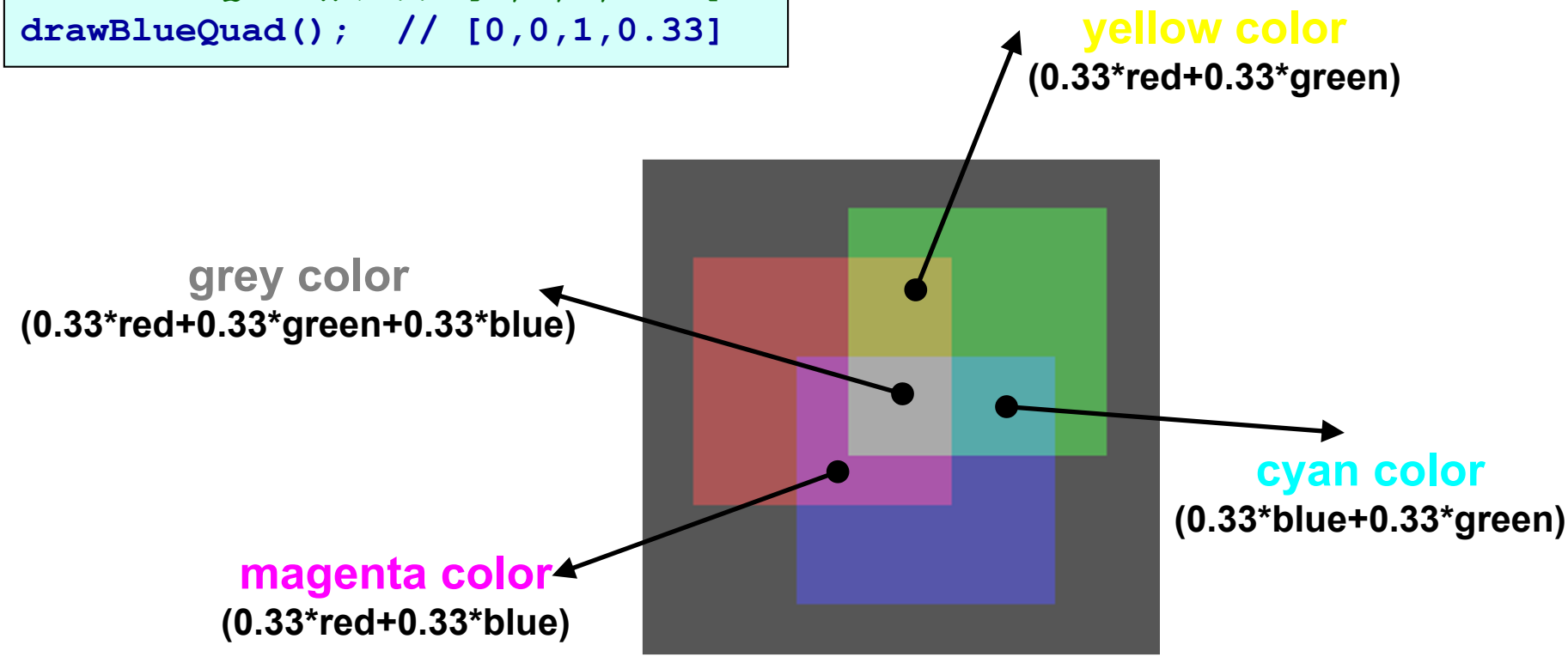
Míchání barev (Blending) – příklad I

```
/* initialize alpha blending
function */
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE);
drawRedQuad();   // [1,0,0,0.33]
drawGreenQuad(); // [0,1,0,0.33]
drawBlueQuad();  // [0,0,1,0.33]
```

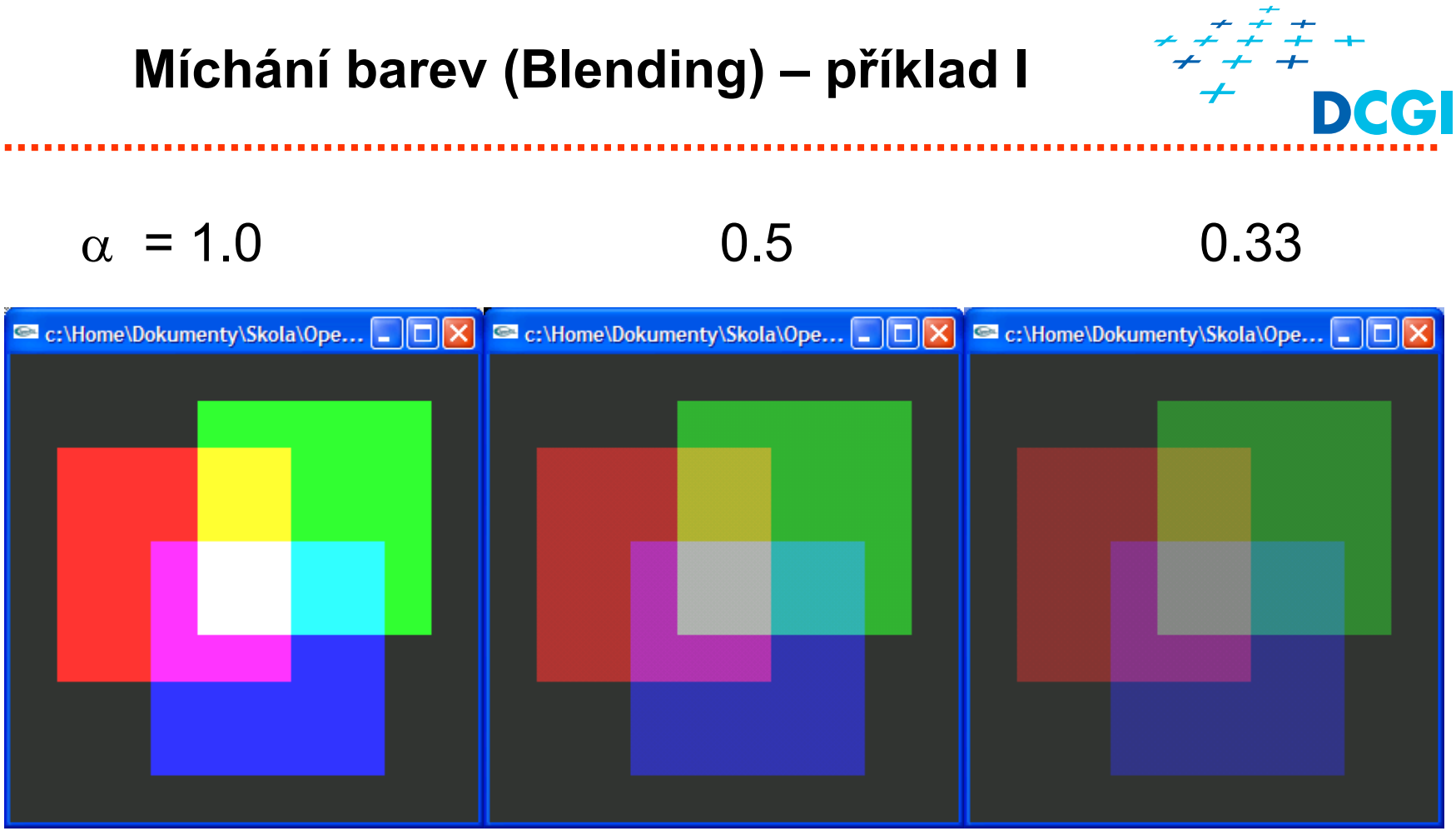
Míchání tří obrázků ve stejném poměru

(Cílové alpha se nepoužívá)

(zdrojové alfa všech fragmentů je 0.33)



Míchání barev (Blending) – příklad I

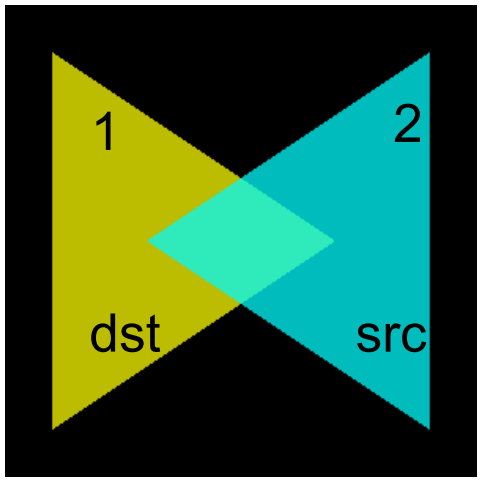


1. glEnable(GL_BLEND);
2. glBlendFunc(GL_SRC_ALPHA, GL_ONE);

Pořadí vykreslování je důležité – příklad II

Míchání v poměru popředí ku pozadí 0.8 : 0.2

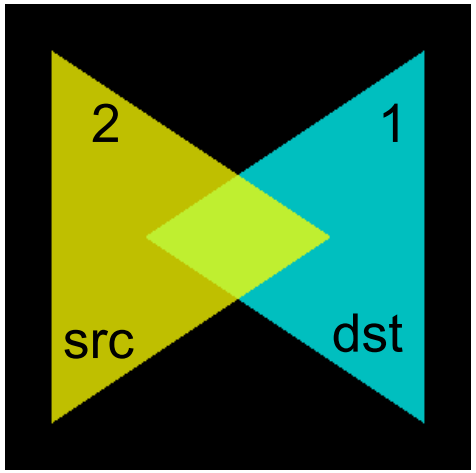
```
glEnable(GL_BLEND);                    // cílové alpha (na obrazovce) se nepoužívá
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```



žlutý překryt tyrkysovým

```
/* žlutý trojuhelnik */
// color=(1.0, 1.0, 0.0, 0.8);
drawLeftTriangle();   ↓
 // sám nakreslí (0.8, 0.8, 0.0, 0,64)
```

```
/* tyrkysový trojuhelnik */
// color=(0.0, 1.0, 1.0, 0.8);
drawRightTriangle();
// sám nakreslí (0.0, 0.8, 0.8 , 0,64)
```



tyrkysový překryt žlutým

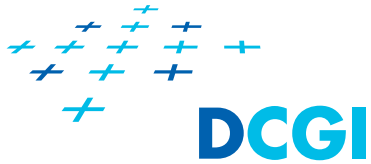
0.8 * (0.0, 1.0, 1.0, **0.8**) + …src tyrkys2.
+(1-0.8)* (0.8, 0.8, 0.0, 0,64) = …dst žlutý1
= (0.16, 0.96, 0.8, 0.768)

≠

src žlutý … 0.8 * (1.0, 1.0, 0.0, **0.8**) +
dst tyrkys…+ (1-0.8)* (0.0, 0.8, 0.8, 0,64) =
= (0.8, 0.96, 0.16, 0.768)

Pořadí vykreslování objektů je při míchání důležité!!!

Pořadí vykreslování je důležité – příklad III

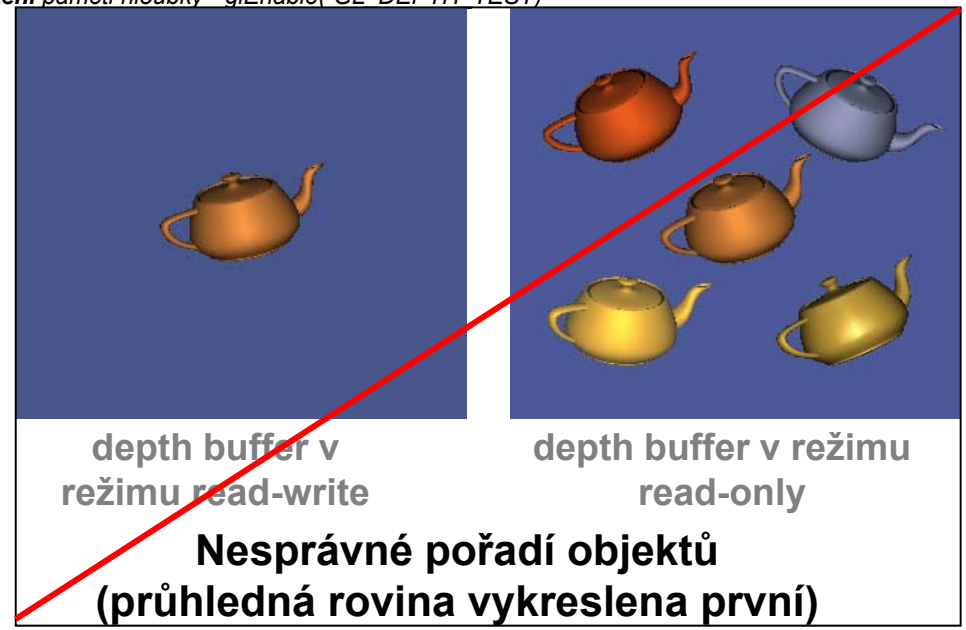


Při zobrazování neprůhledných těles s průhlednými v jedné scéně je pořadí vykreslování objektů velmi důležité!!! Správně je toto pořadí vykreslování:

1. **neprůhledné** objekty (non-transparent) zobrazit jako první
   -> naplní se Z-buffer (testování hloubky i zápis do Z povoleny)
2. **průhledné** objekty zobrazit jako poslední, seřadit dle vzdálenosti k pozorovateli a zobrazit **odzadu dopředu** (testování hloubky povoleno)
   *(Optim: zakázat **zápis** do paměti hloubky – pomocí příkazu glDepthMask(GL_FALSE))*

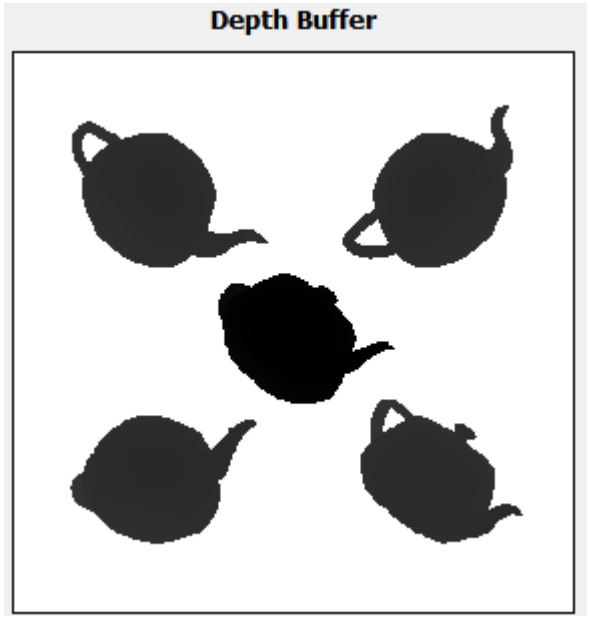
*a **zachovat čtení** paměti hloubky - glEnable(GL_DEPTH_TEST)*

Správné pořadí vykreslování

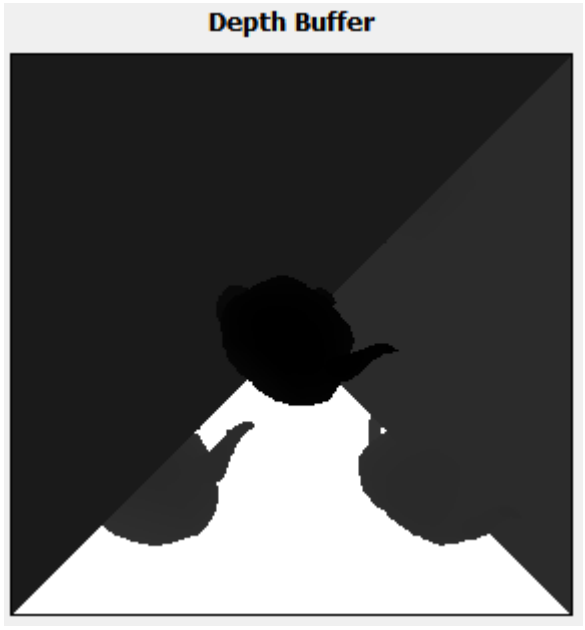


Nesprávné pořadí objektů (průhledná rovina vykreslena první)

Paměť hloubky – povolení zápisu



Jen pro neprůhledné čajníky
(správně)

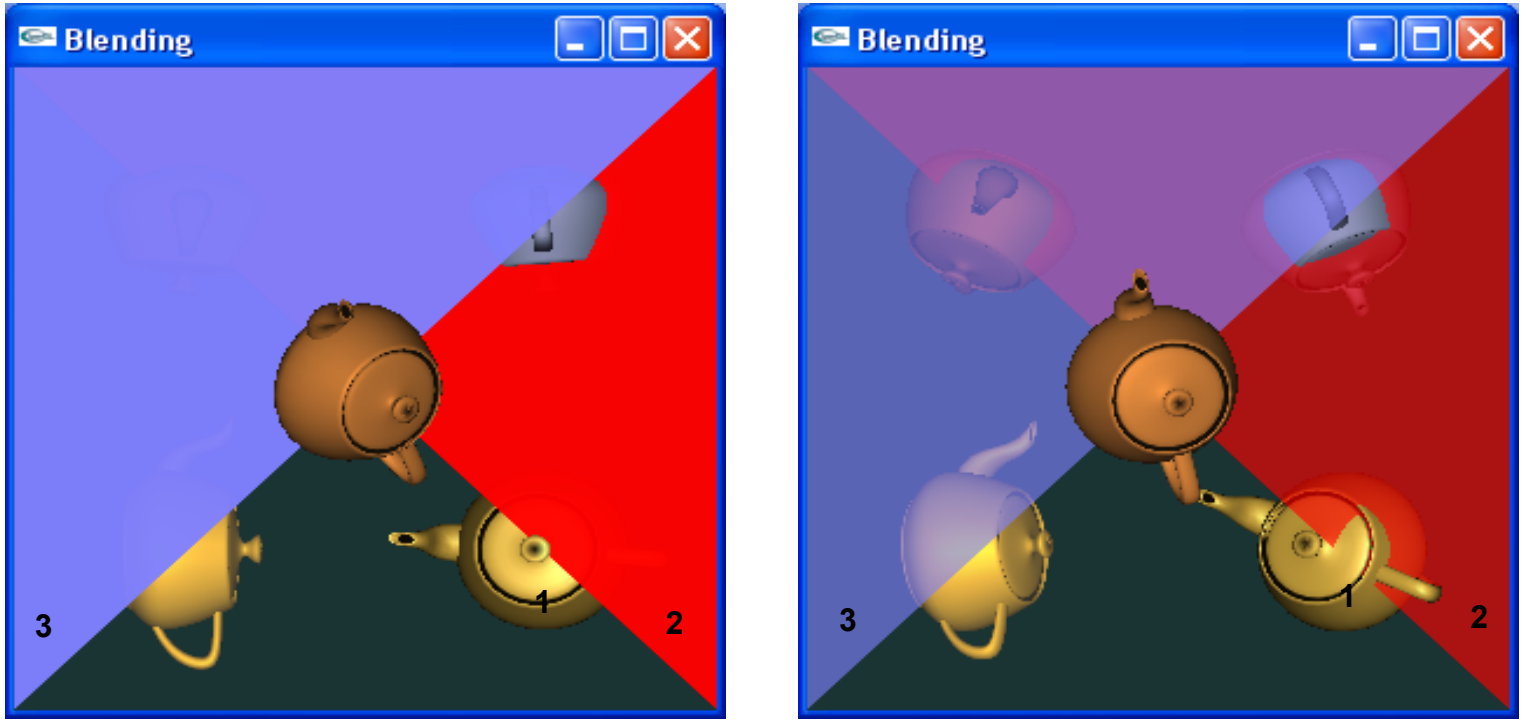


I pro průhledné trojúhelníky
(špatně – zbytečný)

Průhledné vykreslovány správně B-F

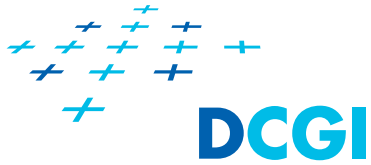
Pokud je **správně pořadí** (neprůhledné, pak průhledné odzadu dopředu),
je obrázek správně (používané vypnutí zápisu do Z-bufferu při kreslení neprůhledných neovlivní správný výsledný obraz, šetří čas)

$\alpha \sim 1.0$ $\alpha \sim 0.5$



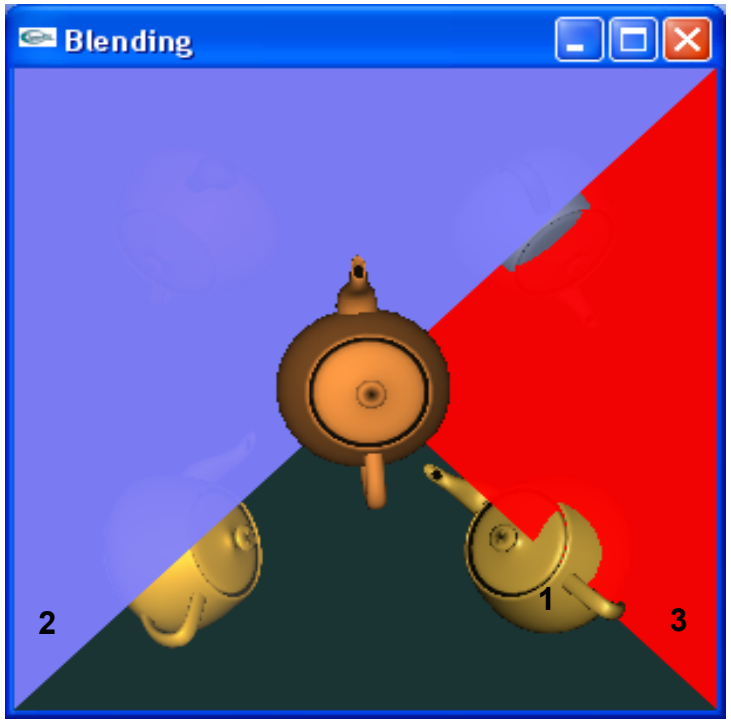
(Zde navíc červená průhledná rovina za modrou)

Průhledné špatně F-B a zapnutý zápis Z

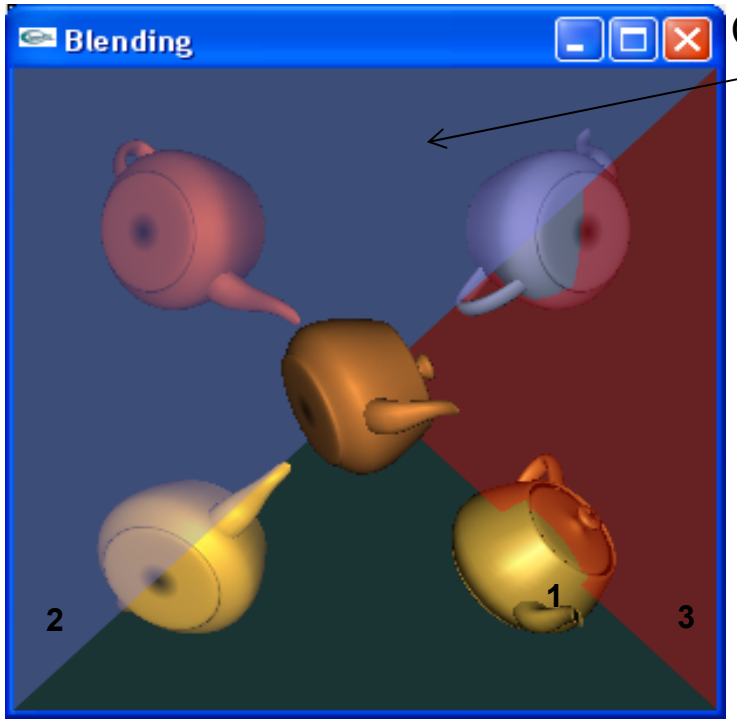


Pokud se správně napřed nakreslí neprůhledné objekty, ale je **špatně pořadí průhledných** (průhledné zepředu dozadu), je obrázek pro $\alpha \neq 1$ **špatně**. **Zapnutý zápis do Z-bufferu** řeší aspoň viditelnost bližší roviny.

$\alpha \sim 1.0$ - OK



$\alpha \sim 0.5$



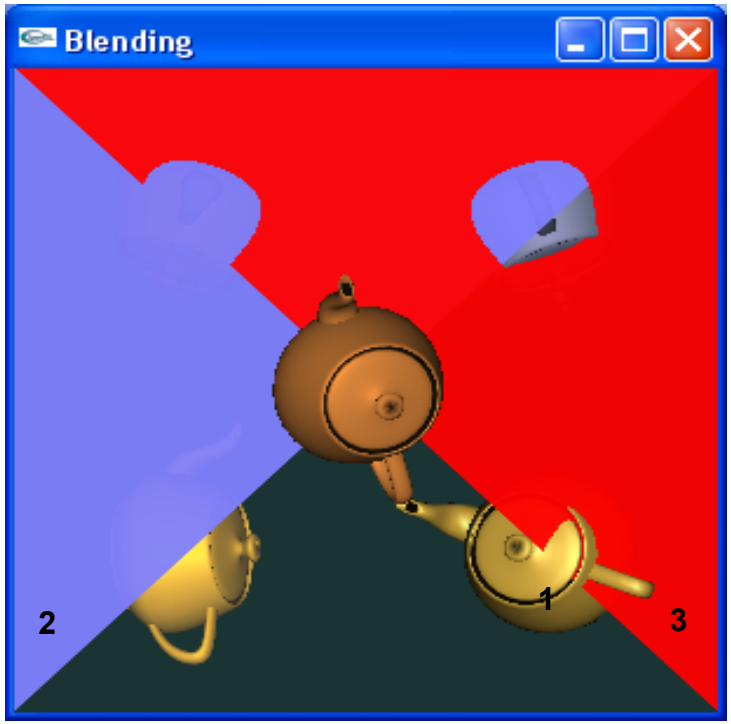
Červená se nekreslí, protože je dál než modrá

Špatně za modrou není vidět červená

Průhledné špatně F-B a vypnutý zápis Z

Pokud se správně napřed nakreslí neprůhledné objekty, ale je **špatně pořadí průhledných** (průhledné zepředu dozadu), je obrázek pro $\alpha \neq 1$ **špatně**. **Vypnutý zápis do Z-bufferu** pokazí i viditelnost bližší roviny

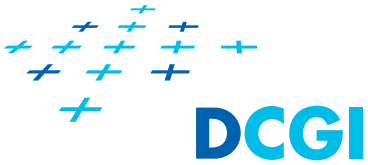
$\alpha \sim 1.0$ - OK



$\alpha \sim 0.5$

Špatně: Modrá je překryta vzdálenější červenou

Míchání v OpenGL 4.x



Více výstupních bufferů – každý se nastaví zvlášť

void glEnable(GL_BLEND); // povolí pro všechny buffery

void glEnablei(GL_BLEND, GLuint **index);** // povolí pro jeden

index = GL_DRAW_BUFFER*i*
= *číslo bufferu v glDrawBuffers*

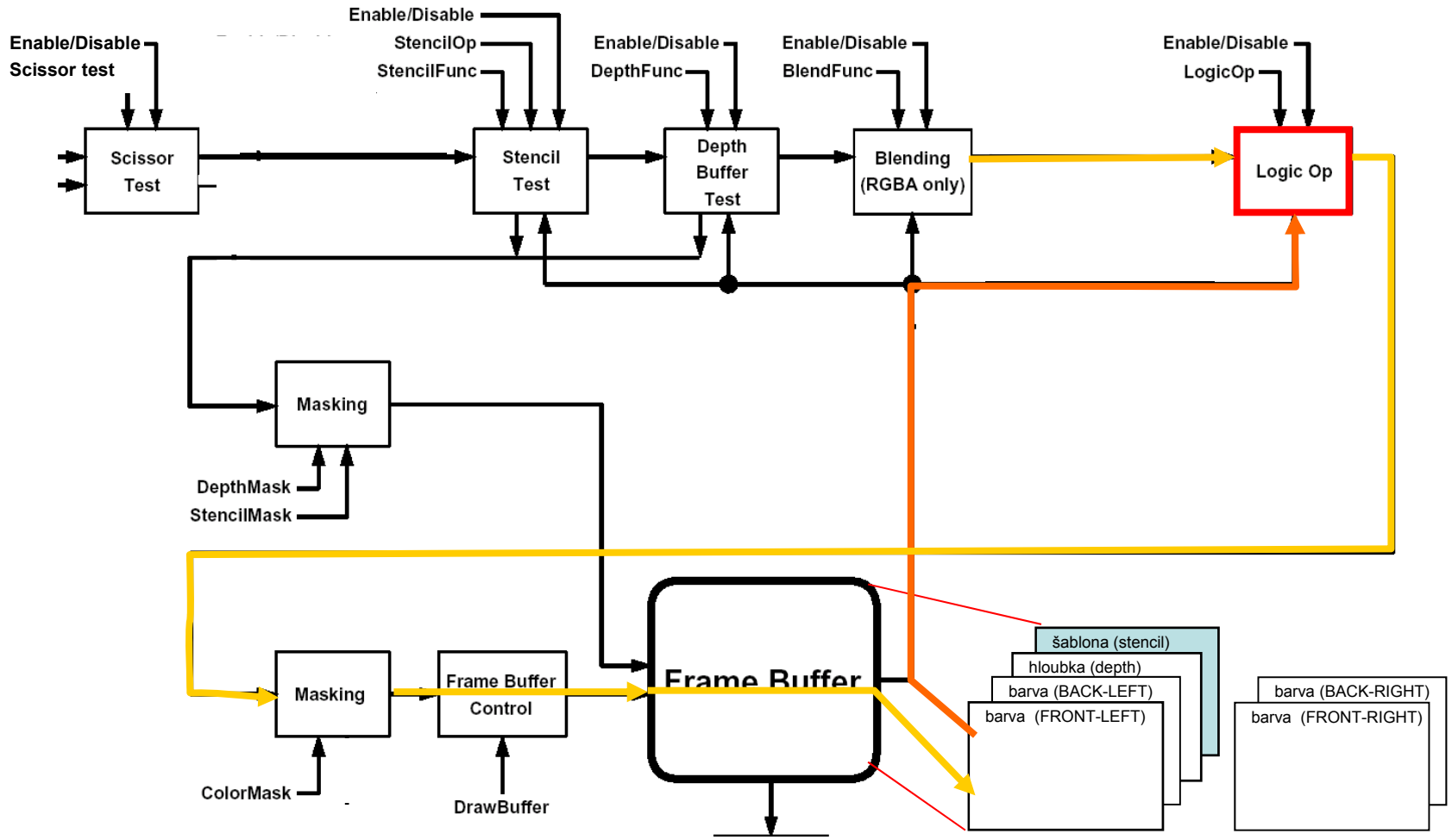
glBlendEquation(GLenum **mode);** // pro všechny buffery

glBlendEquationi(GLuint **buf,** GLenum **mode);** // pro jeden

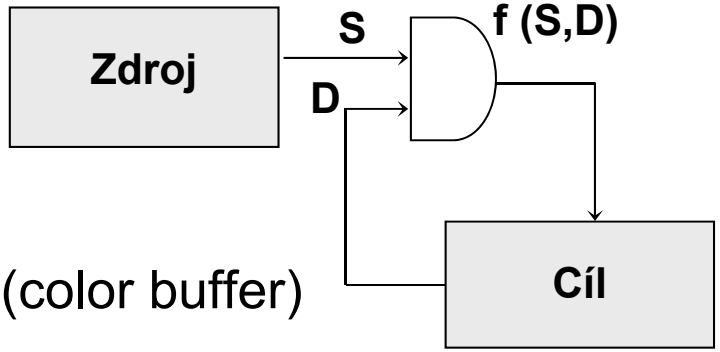
glBlendFunc(GLenum **sfactor,** GLenum **dfactor);**

glBlendFunci(GLuint **buf,**
GLenum **sfactor,** GLenum **dfactor);**

5. Logické operace (1)



Logické operace



- Zdroj *S* - bit fragmentu
- Cíl *D* - bit pixelu barevné paměti (color buffer)

Hodnoty cíle a zdroje 0 a 1 ⇒ f(z,d) má 16 možných kombinací
⇒ 16 módů zápisu

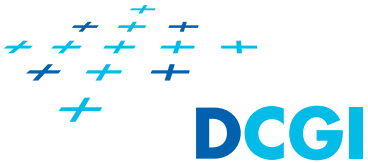
- Výběr logické operace (módu zápisu)

```
void glLogicOp(GLenum mód)
```

- Povolení / zákaz aplikace logické operace

```
glEnable(GL_LOGIC_OP)
glDisable(GL_LOGIC_OP);
```

5. Logické operace (3)



- Logické operace mezi barvou RGBA fragmentu a barvou uloženou v barevné paměti (colorBuferu)
- 16 operací (módů)

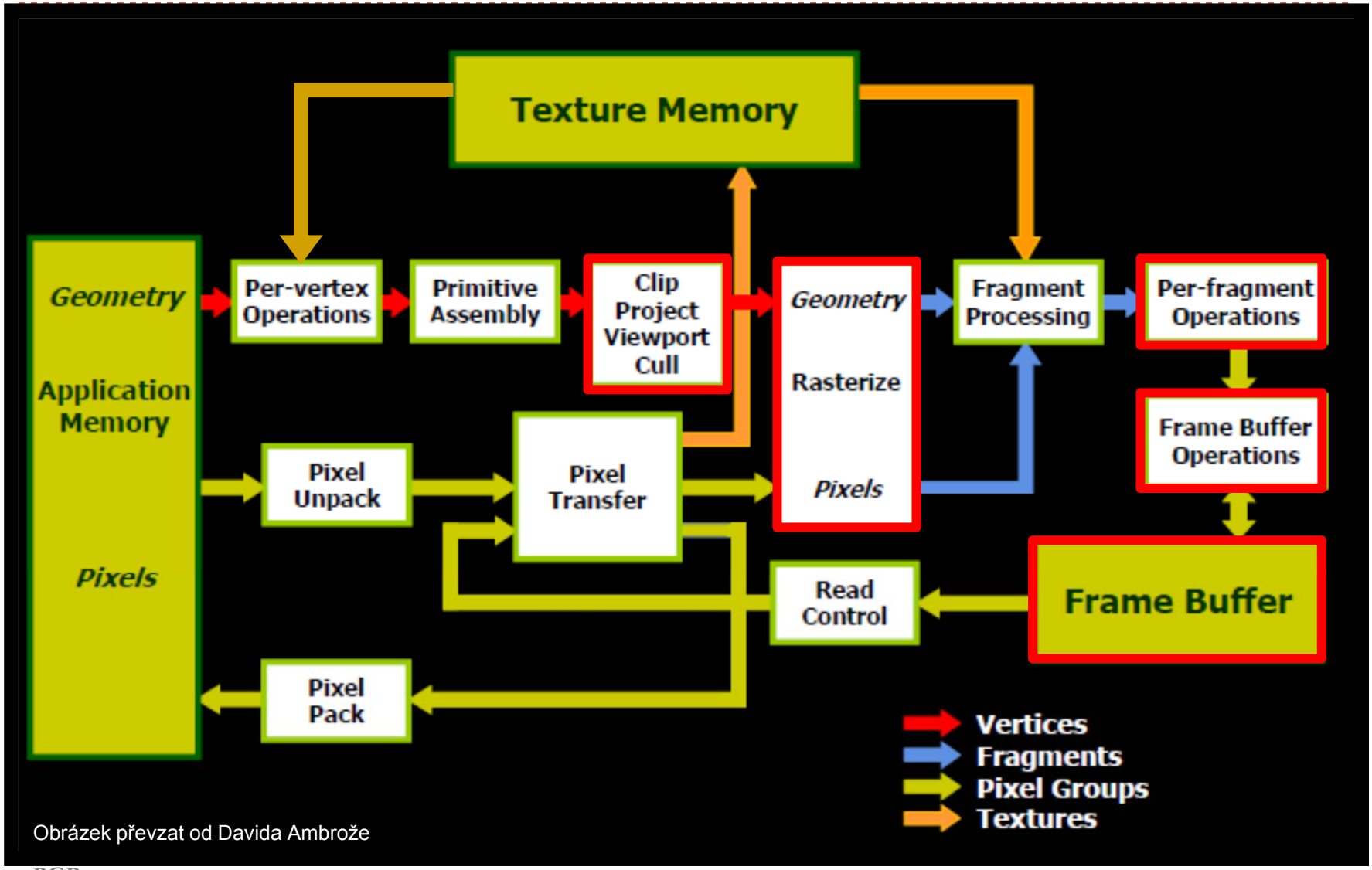
  S = source, D = destination

	operationCode	meaning		operationCode	meaning
0	**GL_CLEAR**	**0**	**8**	**Gl_AND**	**S ∧ D**
1	**GL_COPY**	**S**	**9**	**GL_OR**	**S ∨ D**
2	**GL_NOOP**	**D**	**10**	**GL_NAND**	**¬ (S ∧ D)**
3	**GL_SET**	**1**	**11**	**GL_NOR**	**¬ (S ∨ D)**
4	**GL_COPY_INVERT**	**¬ S**	**12**	**GL_XOR**	**S xor D**
5	**GL_INVERT**	**¬ D**	**13**	**GL_EQUIV**	**¬ (S xor D)**
6	**GL_AND_REVERSE**	**S ∧ ¬ D**	**14**	**GL_AND_INNVERTED**	**¬ S ∧ D**
7	**GL_OR_REVERSE**	**S ∨ ¬ D**	**15**	**GL_OR_INVERTED**	**¬ S ∨ D**

Shrnutí

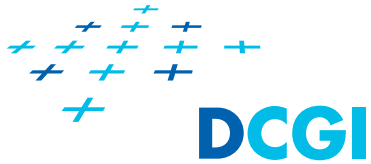
- **Zobrazovací řetězec**
  - Část od sestavení primitiv po rasterizaci
- **Součásti obrazové paměti**
  - na co se používají
  - jak se mažou a jak se do nich zapisuje
- **Testy a operace s fragmenty**
  - provádí se po výpočtu pozice a barvy fragmentu
  - určí, jestli se fragment dostane na obrazovku
  - šablona, test hloubky, …

Zobrazovací řetězec – dnes probrané části



Obrázek převzat od Davida Ambrože

Odkazy



- Steve Baker, *Learning to Love your Z-buffer*.
  https://www.sjbaker.org/steve/omniv/love_your_z_buffer.html
  Precision calculator/.
- *OpenGL FAQ 12: The Depth Buffer,*
  https://www.opengl.org/archives/resources/faq/technical/depthbuffer.htm
- M. Kilgard: Improving Shadows and Reflections via the Stencil Buffer
  http://artis.imag.fr/Recherche/RealTimeShadows/pdf/stencil.pdf