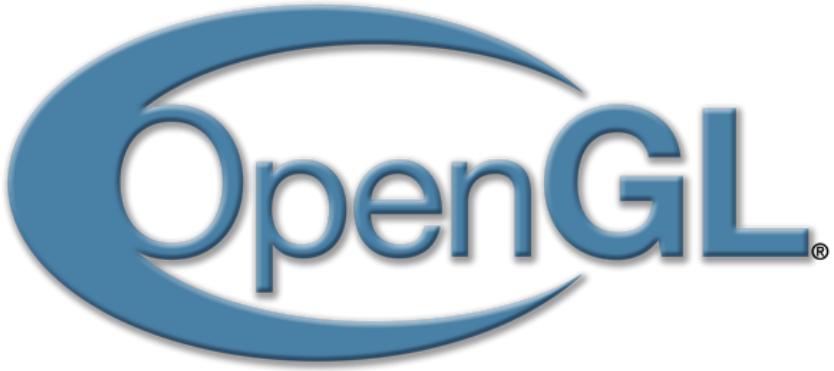


PV112 – Programování grafických aplikací

4. přednáška – Framebuffer, Nastavení viewportu, Clip planes, Mlha, Stereoview, Blending

Framebuffer

- Skládá se z několika dílčích bufferů – každý je určen pro jednu nebo více specifických operací
- Čtení/zápis **fragmentů** – dále nedělitelné plošky, které kromě své barvy a průhlednosti obsahují i informaci o hloubce (vzdálenosti od kamery)
- Každý buffer je paměť obsahující rastrová data, která mají podle určení bufferu svůj specifický význam

Typy bufferů

- **Color buffer** – buffer(y) pro uložení barvy
- **Depth buffer** – paměť hloubky (Z-buffer)
- **Stencil buffer** – paměť šablony

Framebuffer

- Při inicializaci grafického kontextu OpenGL musíme zadat, které z těchto bufferů budeme používat a dále specifikovat jejich vlastnosti a bitovou hloubku
- Při vlastním vykreslování pak můžeme měnit vlastnosti jednotlivých bufferů, nelze je však již vytvářet nebo rušit

Color buffer

- Obsahuje barevnou informaci o vykreslované scéně
- V nejjednodušším případě je pouze jeden, může jich být nainicializováno i několik
- Jeden z bufferů je vždy zobrazen na obrazovce
- Pixely uloženy ve formátu RGBA

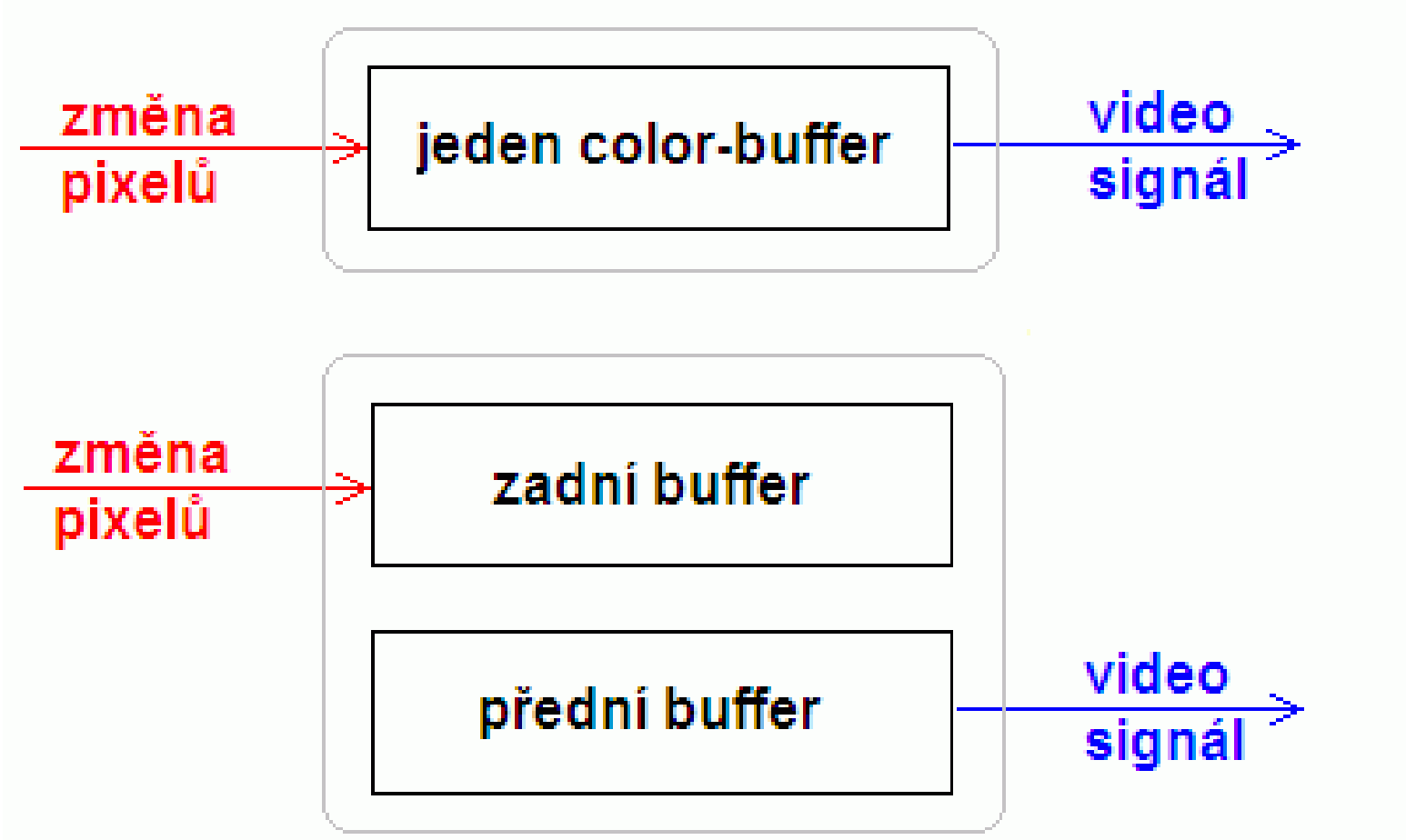
Color buffer

- 4 základní buffery
  - Přední levý
  - Přední pravý
  - Zadní levý
  - Zadní pravý
- Obsah předních bufferů vykreslen na obrazovku
- Obsah zadních bufferů neviditelný

Double buffering

- Zabránění problikávání scény při interaktivní práci s prostorovou scénou
- Vykreslování do neviditelného bufferu za současného zobrazení druhého bufferu, po vykreslení se úloha bufferů prohodí
- Při double-bufferingu se používají **přední** i **zadní** buffery

Double buffering - princip



Color buffer - stereo

- Stereoskopické zobrazení – využití dvou barevných bufferů pro každé oko (pravý a levý buffer)
- Při použití zároveň double-bufferingu:
  - FRONT_RIGHT,
  - FRONT_LEFT
  - BACK_RIGHT
  - BACK_LEFT

Symbolic Constant	Front Left	Front Right	Back Left	Back Right
NONE				
FRONT_LEFT	•			
FRONT_RIGHT		•		
BACK_LEFT			•	
BACK_RIGHT				•

Podpora stereo a double bufferingu

- Informace o podpoře stereo pohledu a double bufferingu:

```
glGetBooleanv(STEREO, &result);
```

```
glGetBooleanv(DOUBLEBUFFER, &result);
```

Depth buffer

- Paměť hloubky, zajišťuje vykreslení pouze viditelných částí objektů díky jejich uspořádání vzhledem k pozici pozorovatele

- Povolení:

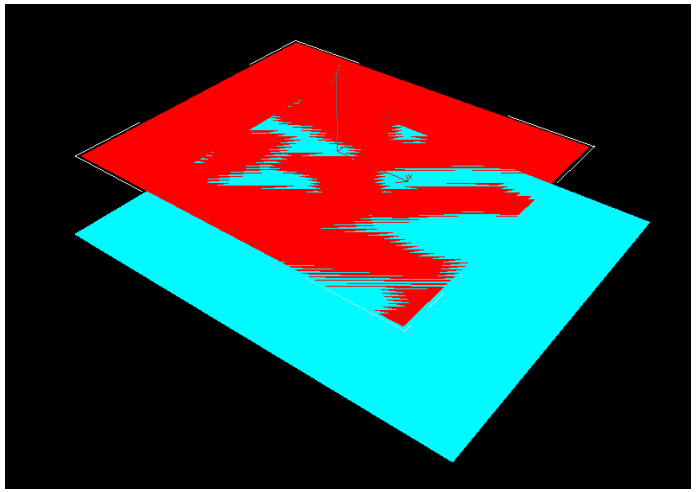
```
glEnable(GL_DEPTH_TEST);
```

Princip funkce hloubkového bufferu

- Při rasterizaci se testuje Z-ová hloubka vytvořeného fragmentu s hloubkou uloženou ve framebufferu
- Z-ová hloubka fragmentu je **menší** než ve framebufferu → fragment se uloží do framebufferu a aktualizuje se Z-ová hodnota
- Z-ová hloubka fragmentu je **větší** než ve framebufferu → fragment je neviditelný, je tedy zahozen

Bitová hloubka bufferu

- U depth bufferu je zásadní
- Např. osmibitový depth buffer → lze rozlišit pouze 256 vzdáleností, což vede k vizuálním artefaktům (Z-fighting)
- Někdy se artefakty přesto objevují – např. u částečně se překrývajících trojúhelníků či objektů v těsné blízkosti



http://upload.wikimedia.org/wikipedia/commons/5/58/Z-fighting.png

Operace nad depth bufferem

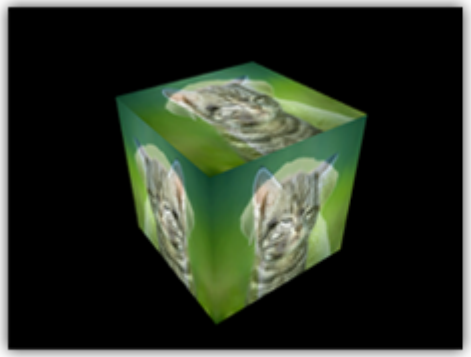
`glDepthFunc(enum func)`

- Funkce použitá pro porovnání hloubky každého příchozího pixelu s hloubkou uloženou v hloubkovém bufferu
- *func* = symbolická konstanta GL_NEVER, **GL_LESS**, GL_LEQUAL_GL_GREATER, GL_GEQUAL_GL_EQUAL, GL_NOTEQUAL, GL_ALWAYS

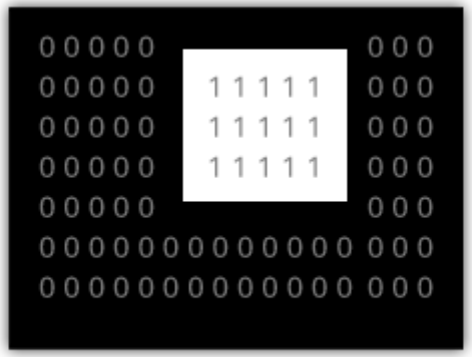
Stencil buffer

- Paměť šablony – určujeme, do kterých míst na obrazovce je povoleno vykreslování
- Příklad:
  - Vykreslování 2D GUI současně s 3D scénou
  - Vykreslování zrcadlících ploch
  - Tvorba stereo obrázků na monitoru s použitím stereo brýlí (dva pohledy na scénu zobrazené prokládaně na jedné obrazovce)

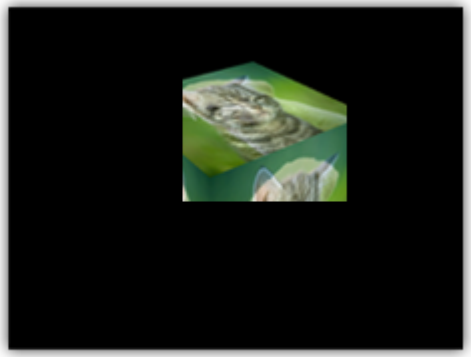
Stencil buffer



Color buffer without stencil test



Stencil buffer



Color buffer with stencil test

Stencil buffer

- Je možné zadat relační operaci, která se provádí mezi hodnotou fragmentu a hodnotou uloženou ve stencil bufferu
- Zde není bitová hloubka tak kritická, jako u depth bufferu. Pro jednodušší úlohy stačí dokonce jednobitová hloubka (např. ořezání scény do obrazce, vytváření stereo obrázku výše zmíněným způsobem)

Stencil buffer

- Má odlišnou funkci od ořezávání – při ořezávání se odstraňují vrcholy grafických primitiv, ale při použití paměti šablony se pracuje přímo s jednotlivými fragmenty
- Povolení:

```
glEnable(GL_STENCIL_TEST)
```

Operace nad stencil bufferem

- Stencilový test

```
glStencilFunc(enum func, int ref, uint mask)
```

- *func* = symbolická konstanta GL_NEVER, GL_LESS, GL_LEQUAL_GL_GREATER, GL_GEQUAL_GL_EQUAL, GL_NOTEQUAL, GL_ALWAYS
- *ref* = referenční hodnota pro porovnání
- *mask* = bitová AND operace

Operace nad stencil bufferem

```
glStencilFuncSeparate(enum face, enum func,
int ref, uint mask)
```

- Možné aplikovat na přední a zadní stěny zvlášť
- *face* = GL_FRONT, GL_BACK, GL_FRONT_AND_BACK

Operace nad stencil bufferem

```
glStencilOp(enum sfail, enum dpfail,
enum dppass)
```

- *sfail* = akce, když stencilový test selže. Hodnoty **GL_KEEP**, GL_ZERO, GL_REPLACE, GL_INCR, GL_INCR_WRAP, GL_DECR, GL_DECR_WRAP a GL_INVERT.
- *dpfail* = když stencilový test projde, ale hloubkový test selže
- *dppass* = když projdou oba testy

Operace nad stencil bufferem

```
glStencilOpSeparate(enum face, enum sfail,
enum dpfail, enum dppass)
```

- Možné aplikovat na přední a zadní stěny zvlášť
- *face* = GL_FRONT, GL_BACK, GL_FRONT_AND_BACK

Operace nad stencil bufferem

```
glStencilMask(uint mask)
```

- *mask* = určuje bitovou masku, podle které je povolen nebo zakázán zápis jednotlivých bitů ve stencilové rovině. Iniciální nastavení – všude hodnoty 1 (povoleno k zápisu).

Operace nad stencil bufferem

```
glStencilMaskSeparate(enum face, uint mask)
```

- Možné aplikovat na přední a zadní stěny zvlášť
- *face* = GL_FRONT, GL_BACK, GL_FRONT_AND_BACK

Stencil buffer

Stencil buffer

Mazání bufferů

- Mazání bufferů je velmi časově náročná operace – je nutné projít všechny pixely v každém bufferu a nastavit je na nějakou hodnotu (často 0)
- Proto je vhodné všechny buffery mazat současně:

```
void glClear(bitfield mask);
```

  - *mask* specifikuje jeden nebo více bufferů, které se mají smazat

glClear() – parametr *mask*

- Parametr *mask*:
  - COLOR_BUFFER_BIT
  - DEPTH_BUFFER_BIT
  - STENCIL_BUFFER_BIT

- Typicky se tento příkaz používá před začátkem vykreslování scény, např.:

```
glClear(GL_COLOR_BUFFER_BIT |
GL_DEPTH_BUFFER_BIT);
```

Nastavení mazací hodnoty

- Při mazání mohou být rovnou do bufferů nastaveny určité hodnoty

```
void glClearColor(float r, float g,
float b, float a);
```

```
void glClearDepth(float depth);
```

```
void glClearStencil(int s);
```

Výběr bufferu, do kterého se bude vykreslovat

- Lze vykreslovat do některého z barevných bufferů

```
void DrawBuffer(enum mode);
```

  - *mode* může nabývat hodnot:
    - FRONT, BACK, RIGHT, LEFT, FRONT_RIGHT, FRONT_LEFT, BACK_RIGHT, BACK_LEFT, NONE
- Defaultně – FRONT, při double bufferingu i BACK

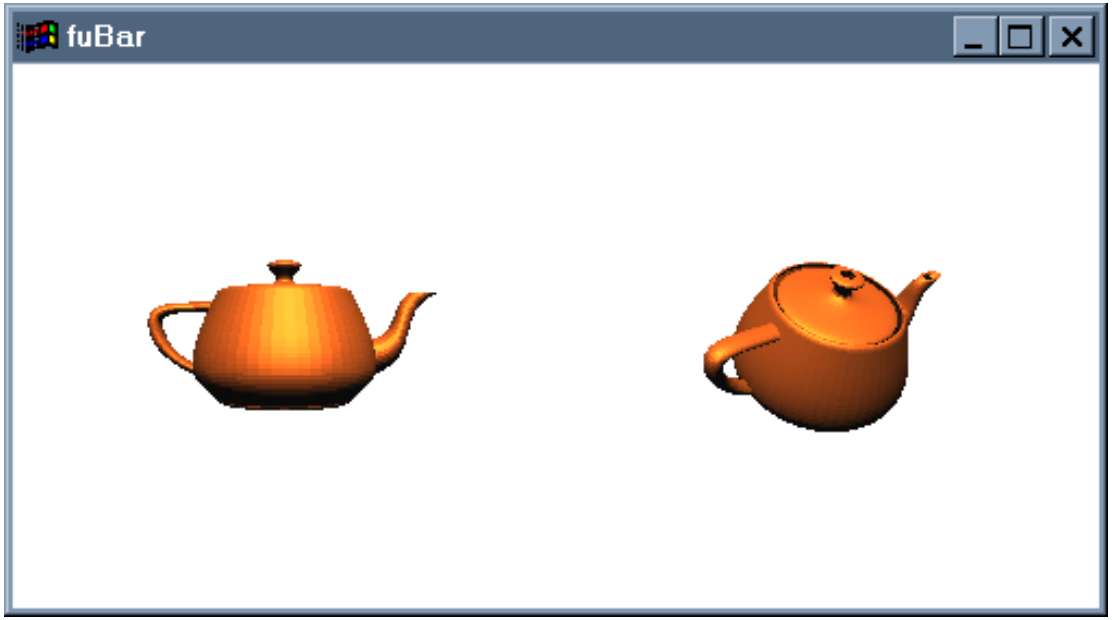
Nastavení viewportu

```
void glViewport(GLint x,GLint y,
GLsizei w, GLsizei h)
```

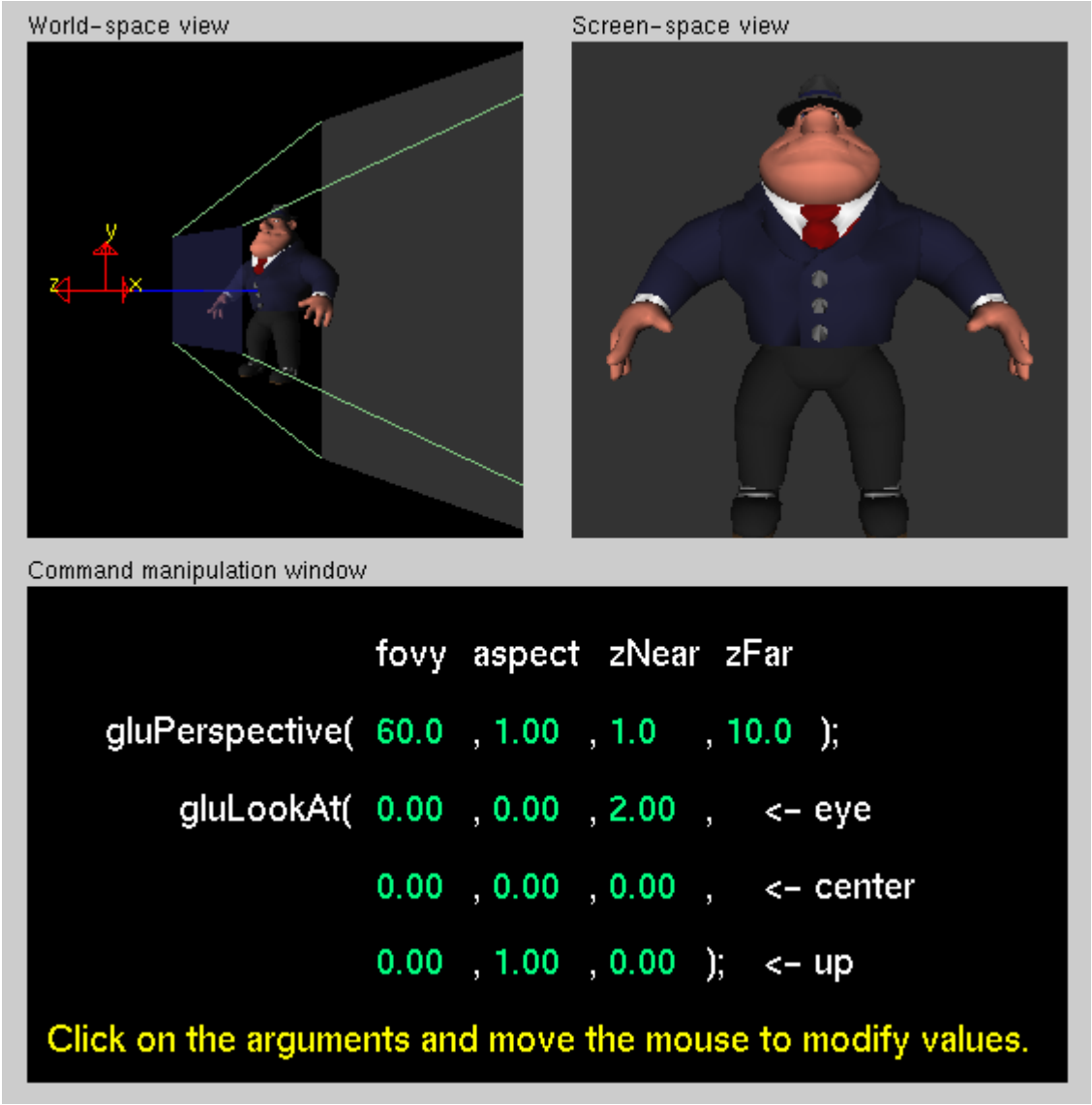
- x a y určují dolní levý roh
- w a h určují velikost obdélníku viewportu

Příklad – rendering do dvou viewportů

```
glViewport(0,0,x/2,y);
glViewport(x/2,0,x/2,y);
```



Projekce – interaktivní ukázka



Transformace souřadnice Z

```
void glDepthRange(GLclampd near,GLclampd far)
```

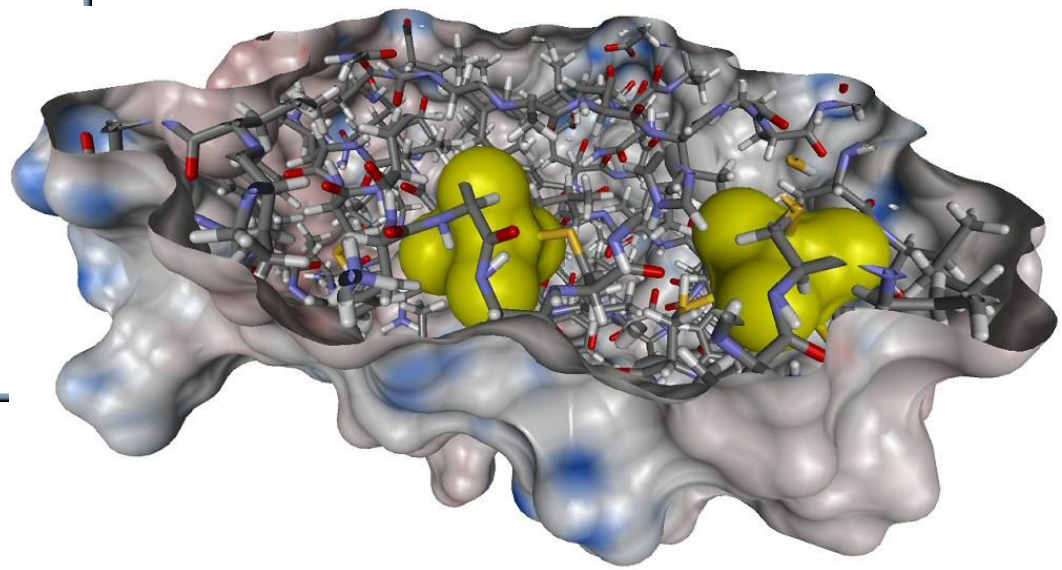
- *near* – mapování bližší ořezávací roviny na souřadnice okna. Implicitní hodnota 0.0.
- *far* – mapování vzdálenější ořezávací roviny na souřadnice okna. Implicitní hodnota 1.0.

Ořezávací roviny

- Rovnice roviny se pošle do vertex shaderu jako uniform proměnná
- Vertex shader vypočte vzdálenost od zadané roviny a výsledek uloží do proměnné *gl_ClipDistance[i]*, kde *i* je číslo ořezávací roviny (0 – minimálně 7)
- Pixely se zápornou vzdáleností jsou ořezány
- Rovina musí být povolena:

```
glEnable(GL_CLIP_DISTANCEi)
```

Ořezávací roviny



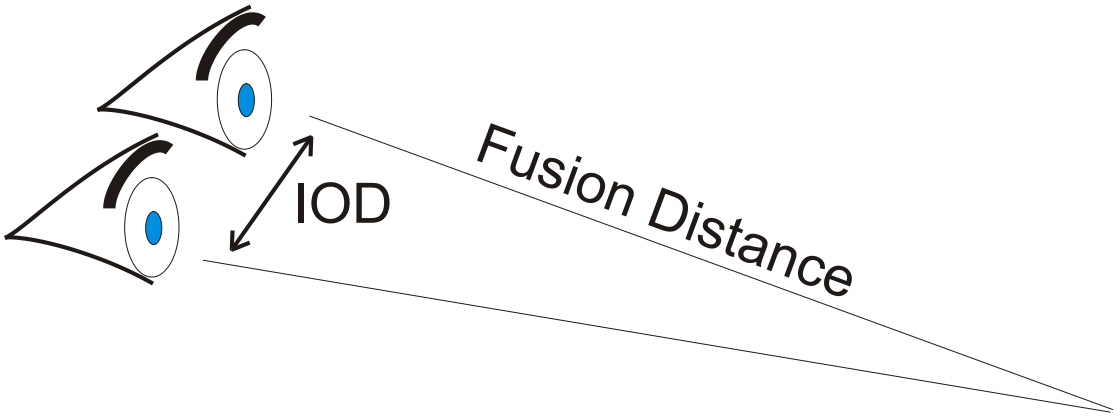
Stereo viewing

- Vytvoření dvou pohledů na scénu pro pravé a levé oko zvlášť
- OpenGL podporuje několik rendering bufferů
- Pro vykreslení frame ve stereomódu:
  - Displej musí tuto funkci podporovat
  - Levé/pravé oko musí být vykresleno v levém/pravém zadním bufferu
  - Zadní buffery musí být náležitě zobrazeny

```
void DrawBuffer(enum mode)
```

Stereo viewing

- Pro určení reálného stereo pohledu musíme definovat dvě hodnoty:
  - Interocular distance (IOD)
  - Fusion distace



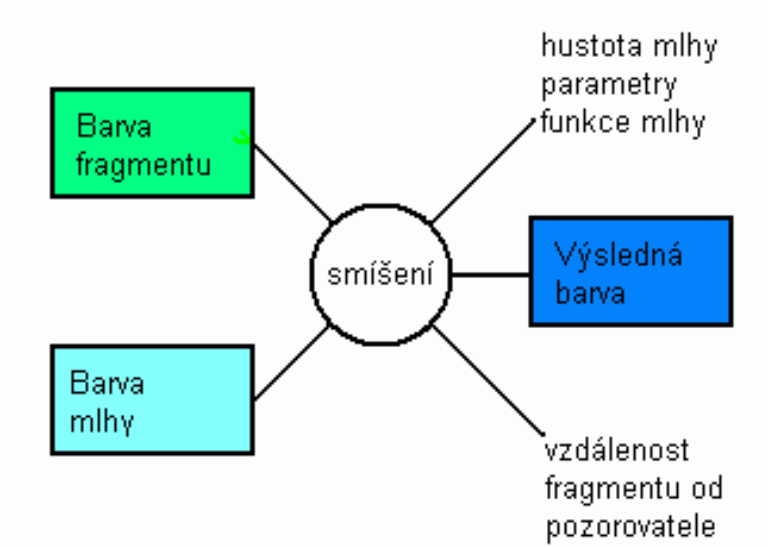
- Dále musíme správným způsobem umístit kameru
- Nastavení pomocí funkce LookAt

Mlha (fog)

- Efekt, kdy se při vzrůstající vzdáleností od pozorovatele postupně mění barva vykreslovaných těles

Mlha

- Zadání základních vlastností mlhy
- Mísení barvy každého fragmentu se zadanou barvou mlhy – poměr původní barvy fragmentu a barvy mlhy ve výsledné barvě je dán parametry funkce mlhy a vzdáleností fragmentu od pozorovatele



Mlha

- Intenzita mlhy závisí na vzdálenosti a rovnici mlhy:

$$\text{Color} = f\,\text{Color}_{\text{incoming}} + (1-f)\,\text{Color}_{\text{fog}}$$

- Můžeme vykreslit pouze viditelné objekty

Mlha

- Možné způsoby výpočtu:
  - **lineární funkce**

$$f = \frac{end - z}{end - start}$$

  - **exponenciální funkce** se záporným exponentem

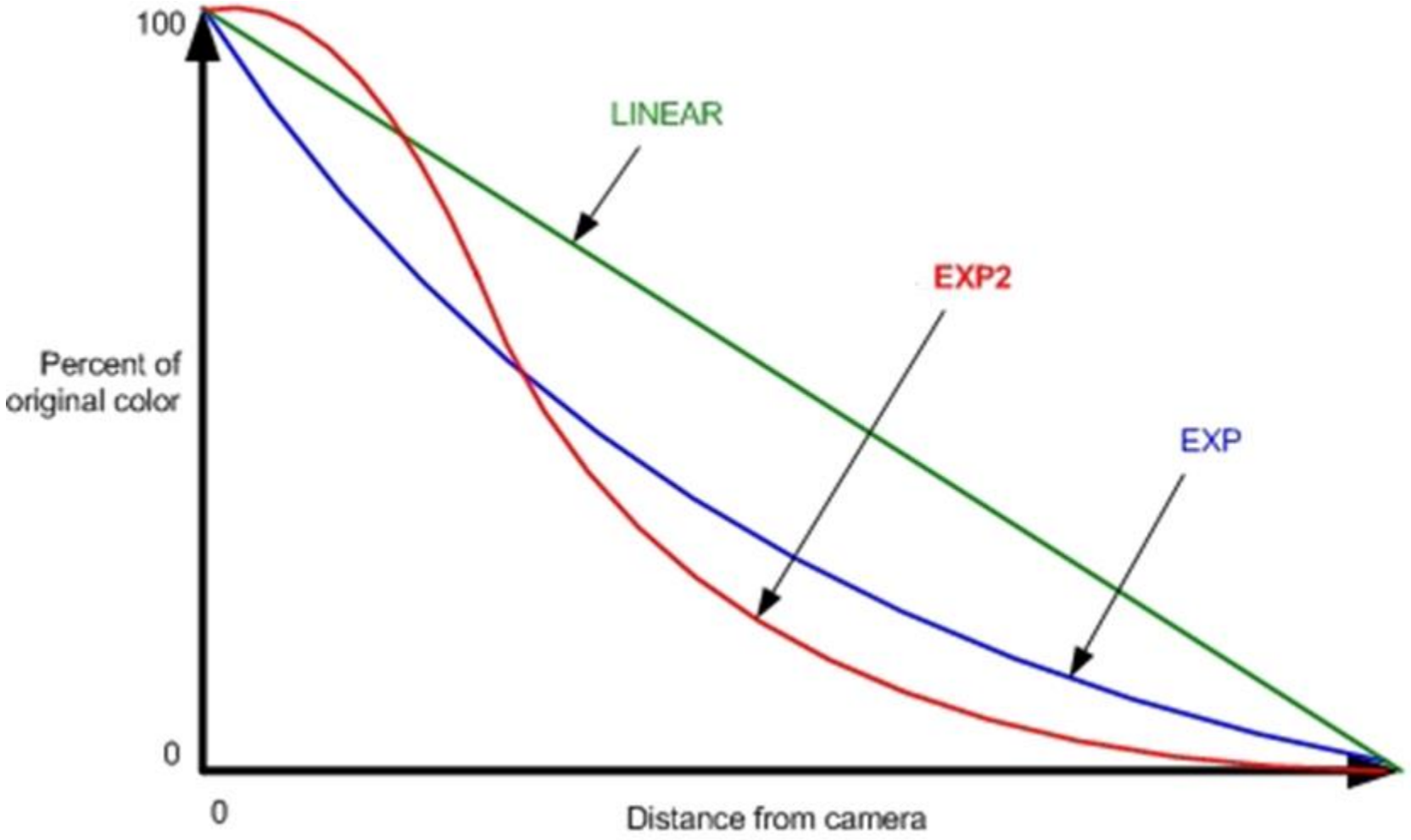
$$f = e^{-density*z}$$

kde *density* je hustota mlhy

  - **Exponenciální funkce** s dvojitým záporným exponentem

$$f = e^{-(density*z)^2}$$

Mlha


100
LINEAR
EXP2
Percent of original color
EXP
0
0
Distance from camera

Uniform proměnná pro definici mlhy

```
struct MyFogParameters
{
    vec4 color;
    float density;
    float start;
    float end;
    float scale;
};
uniform MyFogParameters MyFog;
```

Vertex shader

```
const float LOG2 = 1.442695;
FogFragCoord = length(vVertex);
fogFactor = exp2(-MyFog.density *
    MyFog.density * FogFragCoord *
    FogFragCoord * LOG2);
fogFactor = clamp(fogFactor, 0.0,
    1.0);
```

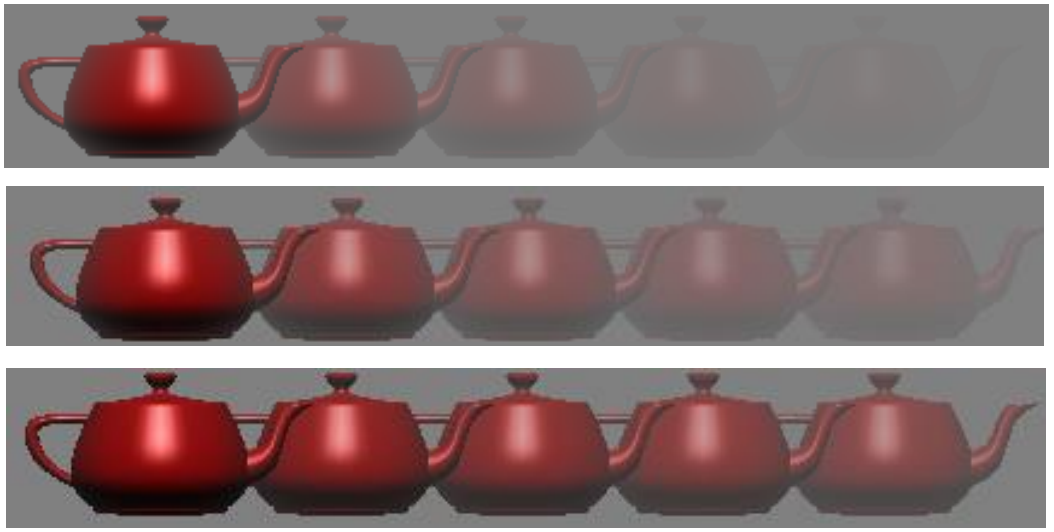
- length (vVertex) je vzdálenost mezi kamerou a aktuálně zpracovávaným vrcholem

Fragment shader

```
out_color = mix(MyFog.color,
finalColor, fogFactor);
```

- Umožní vytvořit interpolaci barvy mezi barvou mlhy (*MyFog.color*) a barvou fragmentu (*finalColor*) a to s ohledem na hodnotu *fogFactor* mlhy

Mlha - příklad



DENSITY=0.8

DENSITY=0.5

DENSITY=0.2

Blending (míchání barev)

- Míchání barev na základě hodnoty alfa

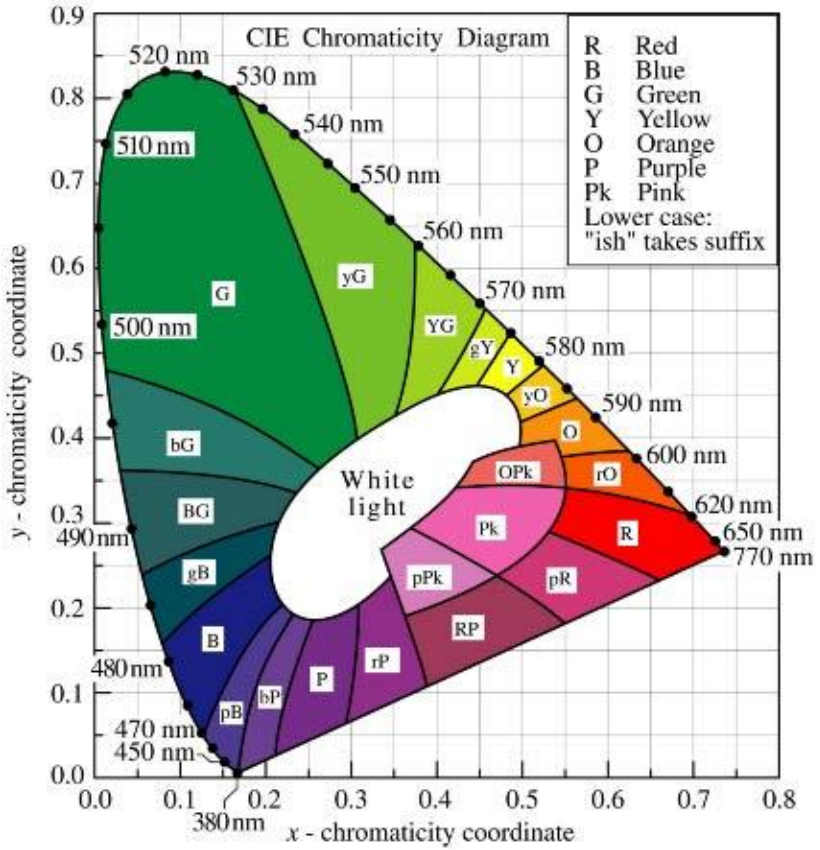
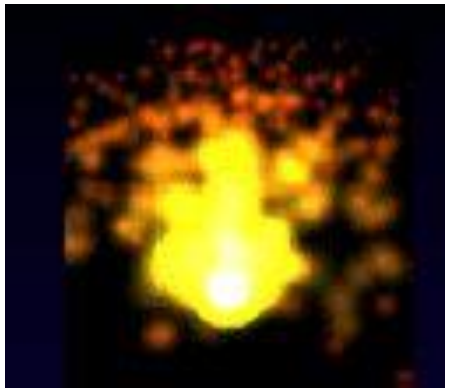


Fig. 10.3. CIE chromaticity diagram. Monochromatic colors are located on the perimeter and white light is located in the center of the diagram (adopted from Gage *et al.*, 1977).

Blending

- Blending přidává do scény průhlednost – za použití alfa hodnoty
- Blending musí být povolen:

```
glEnable(GL_BLEND)
```



Rovnice míchání barev

zdrojová barva (R_s, G_s, B_s, A_s)

cílová barva (R_d, G_d, B_d, A_d)

zdrojový faktor míchání (S_r, S_g, S_b, S_a)

cílový faktor míchání (D_r, D_g, D_b, D_a)

Je vyhodnocena nová RGBA čtveřice

$$\mathbf{(R_s.S_r+R_d.D_r,\ G_s.S_g+G_d.D_g,\ B_s.S_b+B_d.D_b,\ A_s.S_a+A_d.D_a)}$$

Rovnice míchání barev

- Zdroj: barva nově příchozího fragmentu
- Cíl: barva odpovídajícího fragmentu ve framebufferu, který již byl vykreslen dříve

```
void glBlendEquation(enum mode)
```

- Koeficienty míchání lze nastavit pro barvu a hodnotu průhlednosti zvlášť

```
void
glBlendEquationSeparate(enum
modeRGB,enum modeAlpha);
```

Rovnice míchání barev

Mode	RGB Components	Alpha Component
`FUNC_ADD`	$R = R_s * S_r + R_d * D_r$ $G = G_s * S_g + G_d * D_g$ $B = B_s * S_b + B_d * D_b$	$A = A_s * S_a + A_d * D_a$
`FUNC_SUBTRACT`	$R = R_s * S_r - R_d * D_r$ $G = G_s * S_g - G_d * D_g$ $B = B_s * S_b - B_d * D_b$	$A = A_s * S_a - A_d * D_a$
`FUNC_REVERSE_SUBTRACT`	$R = R_d * D_r - R_s * S_r$ $G = G_d * D_g - G_s * S_g$ $B = B_d * D_b - B_s * S_b$	$A = A_d * D_a - A_s * S_a$
`MIN`	$R = \min(R_s, R_d)$ $G = \min(G_s, G_d)$ $B = \min(B_s, B_d)$	$A = \min(A_s, A_d)$
`MAX`	$R = \max(R_s, R_d)$ $G = \max(G_s, G_d)$ $B = \max(B_s, B_d)$	$A = \max(A_s, A_d)$

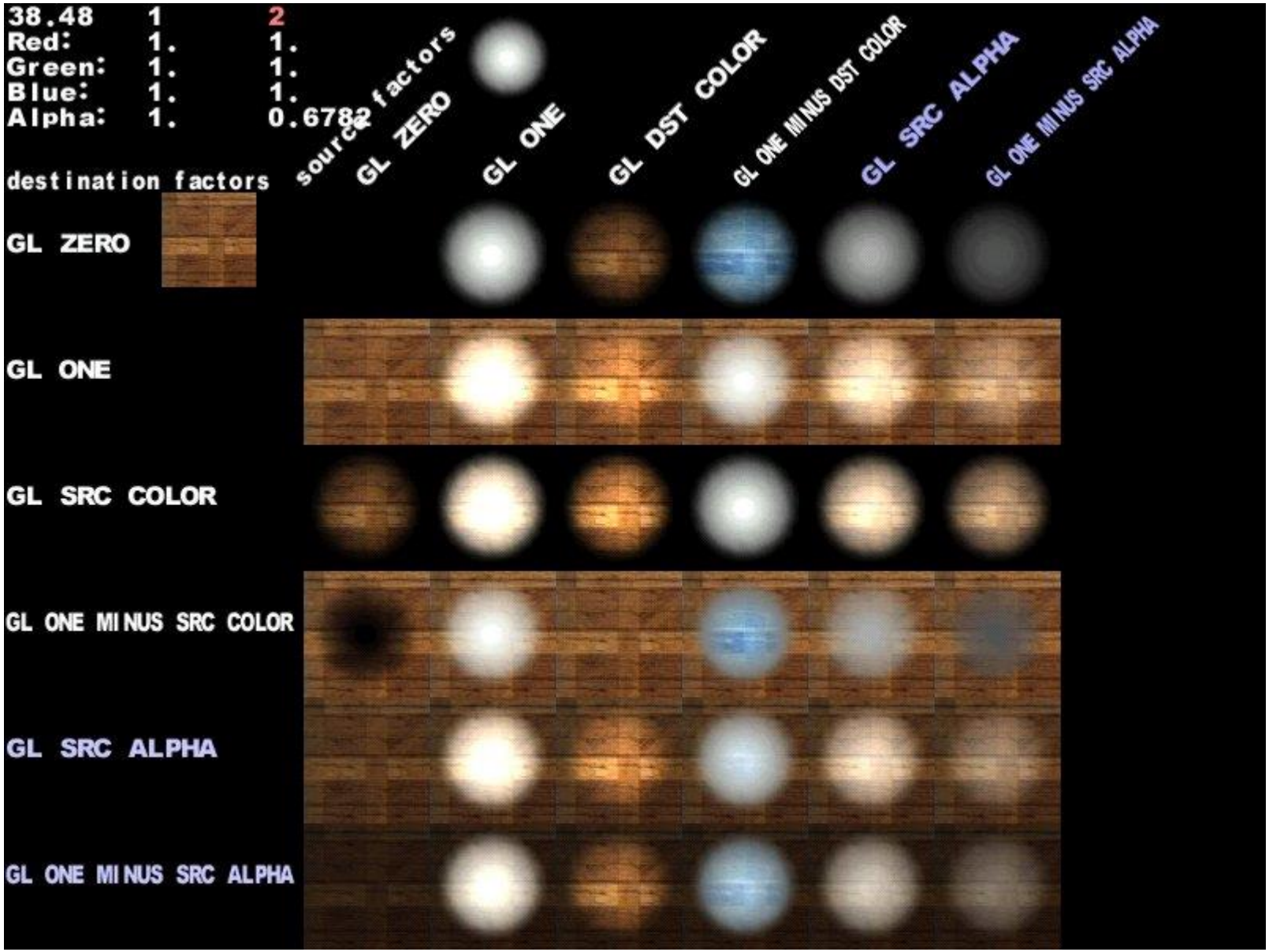
Rovnice míchání barev

```
void glBlendFunc(enum src, enum
dst)
```

src, dst = specifikují faktor míchání pro zdroj a cíl

Faktor míchání může být různý pro barvu a průhlednost:

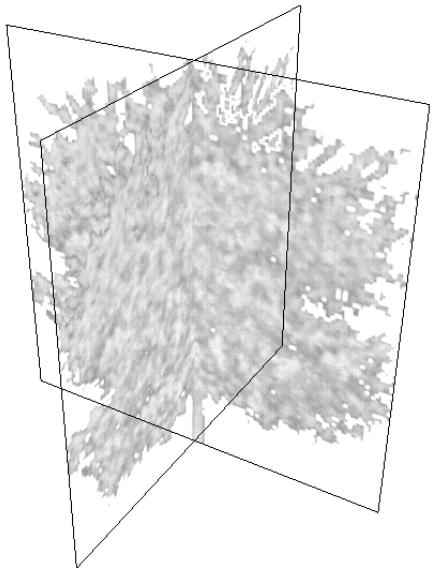
```
void glBlendFuncSeparate(enum
srcRGB, enum dstRGB, enum srcAlpha,
enum dstAlpha)
```



http://nerdposts.blogspot.cz/2009/11/opengl-blending.html

Příklady využití faktorů

- Smíchání dvou a více obrázků v daném poměru
- Úprava každé barevné komponenty zvlášť – ekvivalentní aplikování jednoduchého filtru
- Billboarding



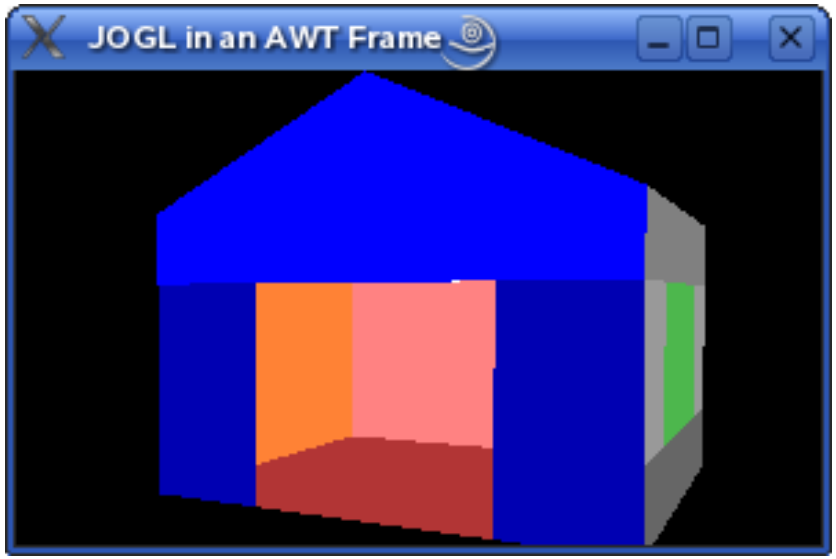
Příklad

- Rovnoměrný blending poloviny jednoho a poloviny druhého obrázku

```
glEnable(GL_BLEND);
glBlendFunc(GL_ONE, GL_ZERO);
RenderFirst();
glBlendFunc(GL_SRC_ALPHA,
GL_ONE_MINUS_SRC_ALPHA);
SetAlphaToTheImage(0.5);
RenderSecond();
```

3D blending s využitím hloubkového bufferu

- Pořadí, ve kterém jsou polygony vykreslovány, je zásadní
- Využití hloubkového bufferu
- Vykreslování průhledných i neprůhledných objektů zároveň



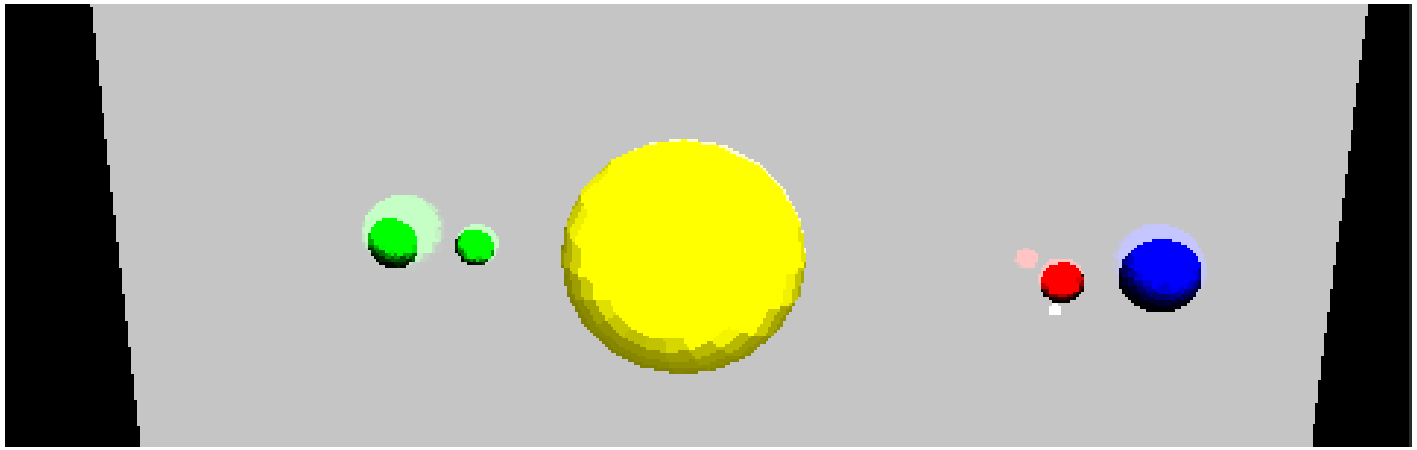
http://www.felixgers.de/teaching/jogl/blendingZBuffer.html

3D blending s využitím hloubkového bufferu

- Při vykreslování průhledných objektů zapneme hloubkový buffer pouze pro čtení

  `glDepthMask()`

  s parametrem FALSE (jen pro čtení)
  nebo TRUE (povolen i zápis)



http://www.felixgers.de/teaching/jogl/blendingZBuffer.html

http://www.root.cz/clanky/opengl-28-blending/

Funkce glFlush a glFinish

- Pokud dochází k bufferování příkazů do tzv. *command queue*

```
void glFlush(void)
```

- Všechny dříve zavolané GL příkazy budou dokončeny v konečném čase

```
void glFinish(void)
```

- Přinutí všechny dříve zavolané GL příkazy, aby skončily