PV112 Programování grafických aplikací

Jaro 2017

Výukový materiál

---

4. přednáška: Framebuffer, nastavení viewportu, clip planes, mlha, stereoview, blending

Framebuffer

Již dříve jsme si popsali způsob vykreslování scén pomocí grafické knihovny OpenGL. Vykreslování probíhá tím způsobem, že se každé vykreslované primitivum podléhá několika transformacím (reprezentovány maticemi) a poté se v obrazové rovině rastruje na dále nedělitelné plošky nazývané fragmenty. Tyto fragmenty jsou poté podrobeny několika testům a dalším operacím. Po provedení těchto operací mohou být zapsány do framebufferu.

Framebuffer se skládá z několika dílčích bufferů, z nichž každý je určen pro jednu nebo více specifických operací, které lze při vykreslování scény provádět. Do framebufferu se zapisují nebo se z něho čtou fragmenty, což jsou pixely, které kromě své barvy a průhlednosti obsahují i informace o hloubce (vzdálenosti od kamery), popř. i další informace. Na každý buffer se můžeme dívat jako na paměť, v níž jsou uložena rastrová data (pixely), která mají podle určení bufferu svůj specifický význam. Buffer lze vymazat nebo přečíst do hlavní paměti počítače, popř. do něj data přímo zapsat, což se používá například při programování některých grafických efektů.

Pro naše účely si představme, že každý pixel framebufferu je sada určitého počtu bitů. Jejich počet závisí na konkrétní GL implementaci.

Při práci s OpenGL můžeme přímo či nepřímo používat tři typy bufferů:

Color buffer(s) – barevný (barvový) nebo více barevných bufferů
Depth buffer – paměť hloubky, nazývaný také Z-buffer
Stencil buffer – paměť šablony

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.

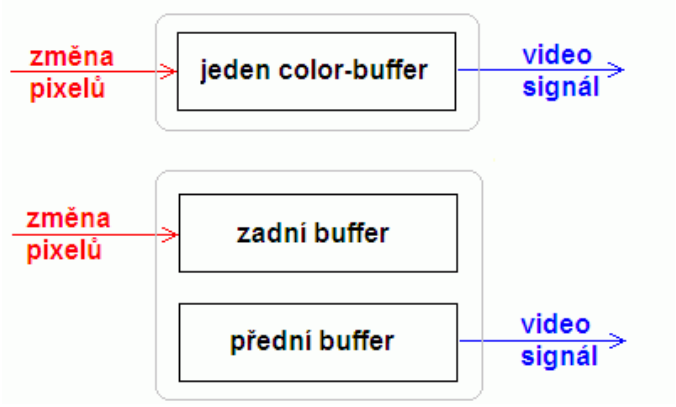
Color buffer

Color buffer se ve skutečnosti skládá z několika bufferů s různými funkcemi. Je v něm uložena barevná informace o vykreslované scéně. Jeden z barevných bufferů je vždy zobrazen na obrazovce. Barvový buffer je v nejjednodušším případě pouze jeden, avšak může jich být nainicializováno několik. V barevném bufferu jsou jednotlivé pixely uloženy ve formátu RGBA.

Ve framebufferu jsou defaultně dostupné čtyři základní buffery – přední levý, přední pravý, zadní levý a zadní pravý. Obsah předních bufferů je zobrazen na obrazovce, zatímco obsah zadních bufferů je neviditelný.

Při interaktivní práci s prostorovou scénou je nutné měnit pohled na scénu, tj. bod umístění kamery a další parametry kamery. Po každé změně pohledu se musí celá prostorová scéna znovu překreslit, protože se většinou změní souřadnice všech vrcholů grafických primitiv na obrazovce. Toto překreslení celé scény (které ve většině případů začíná vymazáním plochy obrazovky barvou pozadí) však může nějakou dobu trvat a v této době by uživatel viděl problikávání způsobené postupnou změnou barev pixelů v barvovém bufferu (color bufferu). Tomuto problikávání lze zabránit tak, že se místo jednoho barvového bufferu použijí buffery dva.

Double-buffering je způsob vykreslování, kdy do předního bufferu zobrazujeme vyrenderovanou scénu a souběžně se do neviditelného zadního bufferu renderuje další obrázek (např. následující obrázek animace). Po dokončení vykreslení snímku v zadním bufferu se úloha barevných bufferů prohodí. Podrobněji se budeme double-bufferingem zabývat dále. V případě double-bufferingu jsou tyto dva barevné buffery nazývány přední buffer (*front buffer*) a zadní buffer (*back buffer*).



Některé implementace OpenGL, které umožňují generování stereo obrázků pro různá 3D výstupní zařízení (brýle, helmy, stereo monitory apod.), mohou zobrazovat současně dva barevné buffery pro každé oko. Tyto buffery se také podle toho jmenují levý buffer a pravý buffer. V příkazech OpenGL se používají symbolické konstanty LEFT a RIGHT. Pokud se kromě

generování stereo obrázků používá i double buffering, existují současně čtyři barevné buffery označované jako FRONT_RIGHT, FRONT_LEFT, BACK_RIGHT a BACK_LEFT.

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

Získání informací o podpoře stereoskopického pohledu lze získat pomocí funkce *glGetBooleanv(STEREO, &result)*. Informace o podpoře double-bufferingu se získají po zavolání funkce *glGetBooleanv(DOUBLEBUFFER, &result)*.

Depth buffer

Depth buffer, tedy paměť hloubky, je někdy také nazýván *Z-buffer*. Jedná se o paměť, v níž jsou většinou uloženy informace zajišťující vykreslení pouze viditelných částí objektů, tj. vzdálenější plošky jsou překryty ploškami bližšími.

V tomto bufferu jsou uloženy informace o hloubce fragmentu, tj. o vzdálenosti fragmentu od projekční roviny. Zjednodušeně řečeno, tyto vzdálenosti jsou většinou ukládány takovým způsobem, že nejbližší objekty mají hloubku 0 a nejvzdálenější mají hloubku 1.

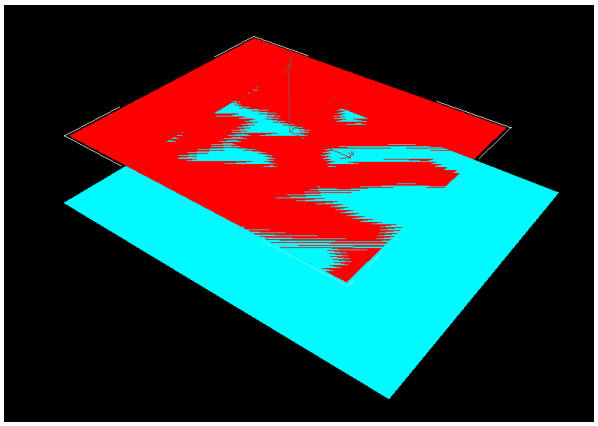
Ve většině implementací OpenGL se používá pouze jedna paměť hloubky, jejich větší počet postrádá pro značnou část vykreslovacích postupů smysl.

Používání hloubkového bufferu je třeba nejdříve povolit pomocí funkce *glEnable(GL_DEPTH_TEST)*.

V naprosté většině případů je funkce paměti hloubky nastavena tak, že se při rasterizaci testuje Z-ová hloubka vytvořeného fragmentu s hloubkou uloženou ve framebufferu na pozici vkládaného fragmentu. Pokud je Z-ová hloubka fragmentu **menší** než ve framebufferu, fragment se do framebufferu uloží a samozřejmě přepíše i Z-ovou pozici. Pokud je Z-ová hloubka fragmentu **větší**, je jasné, že je fragment ve výsledné scéně neviditelný, a proto je zahozen.

U depth bufferu je velmi důležitá jeho bitová hloubka. Pokud je například depth buffer pouze osmibitový (tak ho měly implementovány první grafické akcelerátory na PC), je možné rozlišit pouze 256 vzdáleností, což může vést k vytvoření vizuálních artefaktů ve scéně (nazývaných Z-fighting – viz obrázek). V dnešní době se díky větším bitovým hloubkám množství artefaktů

snižuje (avšak nastat mohou, typicky pokud vytvoříme dva trojúhelníky, které se částečně překrývají nebo jsou objekty příliš blízko sebe).



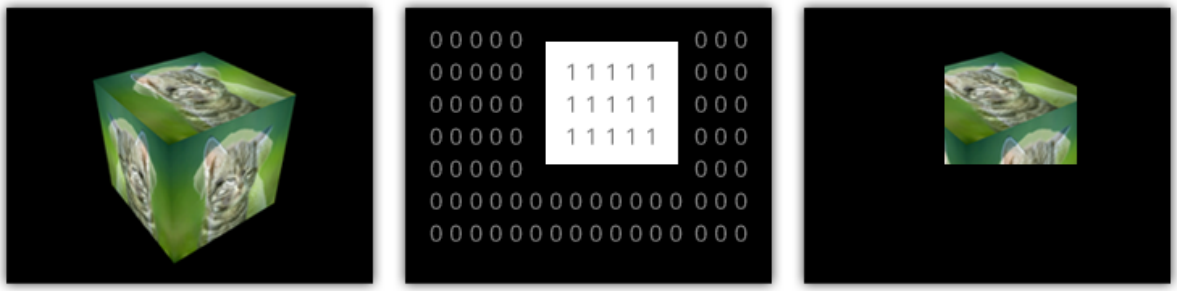
Mezi operace prováděné nad depth bufferem patří *glDepthFunc* s parametrem *enum func*. Jedná se o funkci použitou pro porovnání hloubky každého příchozího pixelu s hloubkou uloženou v hloubkovém bufferu. Parametr může nabývat následujících hodnot: GL_NEVER, GL_LESS, GL_LEQUAL_GL_GREATER, GL_GEQUAL_GL_EQUAL, GL_NOTEQUAL, GL_ALWAYS. Defaultní hodnota je GL_LESS.

Stencil buffer

Stencil buffer je volitelné rozšíření hloubkového bufferu, které nám dává větší kontrolu nad tím, které fragmenty mají být vykresleny a které ne. Podobně jako u hloubkového bufferu je pro každý pixel uložena jeho hodnota, ale v tomto případě můžeme určit, kdy a jak se tato hodnota změní a kdy bude fragment vykreslen (v závislosti na této hodnotě).

Stencil buffer neboli paměť šablony je tedy používán pro určení, do kterých míst na obrazovce je povoleno vykreslování. Jedno z možných použití stencil bufferu je při vykreslování 2D grafického uživatelského rozhraní současně s trojrozměrnou scénou. Dalším, pokročilejším použitím stencil bufferu je algoritmus pro vykreslování zrcadlících ploch nebo pro tvorbu stereo obrázků na monitoru s použitím stereo brýlí (dva pohledy na scénu snímané ze dvou bodů zobrazené prokládaně na jedné obrazovce).

Použití stencil bufferu si ukážeme na jednoduchém příkladě. Nejdříve je stencil buffer promazán (všude jsou nastaveny nuly) a poté je do něj nastaven obdélník obsahující jedničky. Vykreslovací operace pak použije tyto hodnoty k určení, které fragmenty z color bufferu budou vykresleny.



Color buffer without stencil test | Stencil buffer | Color buffer with stencil test

U stencil bufferu lze zadat relační operaci, která se provádí mezi hodnotou fragmentu a hodnotou uloženou ve stencil bufferu. Také je možné zadat, zda a popř. při jaké podmínce se budou měnit data ve stencil bufferu. U stencil bufferu většinou není bitová hloubka tak kritická jako v případě depth bufferu. Dokonce nám pro některé jednodušší úlohy postačuje jednobitová hloubka (typicky ořezání scény do nějakého obrazce a výše zmíněná tvorba stereo obrázků).

Paměť šablony má odlišnou funkci od ořezávání. Při ořezávání se odstraňují vrcholy grafických primitiv, ale při použití paměti šablony se pracuje přímo s jednotlivými fragmenty (tj. obrazovými elementy).

Nad stencil bufferem se provádí následující operace, které určí, které pixely mají být vykresleny. Jedná se o tzv. **stencilový test**. Ten podmíněně eliminuje pixely na základě porovnání referenční hodnoty a hodnoty uložené ve stencil bufferu. Test je nejdříve třeba povolit pomocí volání glEnable(GL_STENCIL_TEST). Následuje funkce *glStencilFunc* s následujícími parametry:

- *func* je symbolická konstanta určující typ porovnávací funkce. Může nabývat některé z osmi hodnot GL_NEVER, GL_LESS, GL_LEQUAL_GL_GREATER, GL_GEQUAL_GL_EQUAL, GL_NOTEQUAL, GL_ALWAYS.
- *ref* je referenční hodnota typu integer, která se používá pro porovnávání.
- *mask* je bitová AND operace.

Existuje i varianta této funkce, kterou je možné aplikovat na přední a zadní stěny objektů zvlášť:

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

kde parametr *face* může nabývat hodnot GL_FRONT, GL_BACK, GL_FRONT_AND_BACK.

Pro nastavení akce prováděné po stencilovém testu slouží funkce *glStencilOp*. Ta má následující parametry.

- *sfail* specifikuje akci, která se má provést v případě, že stencilový test selže. Může nabývat
  hodnot GL_KEEP, GL_ZERO, GL_REPLACE, GL_INCR, GL_INCR_WRAP, GL_DECR, GL_DECR_WRAP a GL_INVERT. Defaultní hodnota je GL_KEEP.
- *dpfail* specifikuje akci, která se má provést v případě, že stencilový test projde, ale hloubkový test selže. Nabývá stejných hodnot jak sfail.
- *dppass* specifikuje akci, která se má provést v případě, že stencilový i hloubkový test projdou nebo v případě, že hloubkový test není dostupný nebo nemáme k dispozici hloubkový buffer. Hodnoty opět stejné.

Obdobně jako pro glStencilFunc existuje i varianta této funkce, kterou je možné aplikovat na přední a zadní stěny objektů zvlášť (parametr *face* nabývá stejných hodnot jako u glStencilFunc):

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

Funkce *glStencilMask* řídí zápis individuálních bitů do stencilové masky. Má parametr *mask*, který určuje bitovou masku, podle které je povolen nebo zakázán zápis jednotlivých bitů ve stencilové rovině. Iniciální nastavení má všude hodnoty 1 (povoleno k zápisu). Hodnota 0 znamená, že daný bit je chráněn proti zápisu.

Opět existuje i varianta této funkce, kterou je možné aplikovat na přední a zadní stěny objektů zvlášť:

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

Mazání bufferů

Mazání bufferů je operace časově náročná, protože je nutné projít všechny pixely v každém bufferu a nastavit je na nějakou hodnotu (většinou nulu). Proto je vhodné všechny buffery mazat současně, což umožňuje příkaz OpenGL void *glClear(bitfield mask)*, kde pomocí parametru *mask* lze specifikovat jeden nebo více bufferů, které se mají smazat.

Parametr *mask* může být vytvořen logickým součtem hodnot COLOR_BUFFER_BIT (maže se zadní barvový buffer), DEPTH_BUFFER_BIT (maže se paměť hloubky) a STENCIL_BUFFER_BIT (maže se paměť šablony).

Typicky se příkaz glClear() používá před začátkem vykreslování scény.

Přímo při mazání mohou být rovnou do bufferů nastaveny určité hodnoty. Toho dosáhneme použitím následujících funkcí.

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

Fukce *glClearColor* se čtyřmi parametry určujícími RGBA hodnoty vyčistí barevný buffer a nastaví do něj hodnoty specifikované těmito parametry (definovanými v rozsahu 0 až 1).

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

Funkce *glClearDepth* s jedním parametrem specifikuje nastavení hodnoty hloubky po vyčištění bufferu.

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

Podobným způsobem funguje i funkce *glClearStencil*.

Výběr bufferu, do kterého se bude vykreslovat

Vykreslování lze provádět do některého z barvových bufferů: předního, zadního, levého předního, levého zadního, pravého předního, pravého zadního, nebo do libovolného dalšího barevného bufferu (viz později v dalších přednáškách). Pomocí příkazu *void DrawBuffer(enum mode)* lze zadat, do kterého bufferu se budou provádět další vykreslovací operace. Parametr mode může nabývat těchto hodnot:

FRONT – přední buffer
BACK – zadní buffer
RIGHT – pravý buffer (není použit double buffering)
LEFT – levý buffer (není použit double buffering)
FRONT_RIGHT – přední pravý buffer
FRONT_LEFT – přední levý buffer
BACK_RIGHT – zadní pravý buffer
BACK_LEFT – zadní levý buffer
NONE – nevykresluje se do žádného bufferu

Pro většinu klasických způsobů vykreslování je možné ponechat základní nastavení, tj. pokud není použit double-buffering, ponechat hodnotu FRONT, a pokud je použit, vykreslovat do zadního bufferu (BACK).

Nastavení viewportu

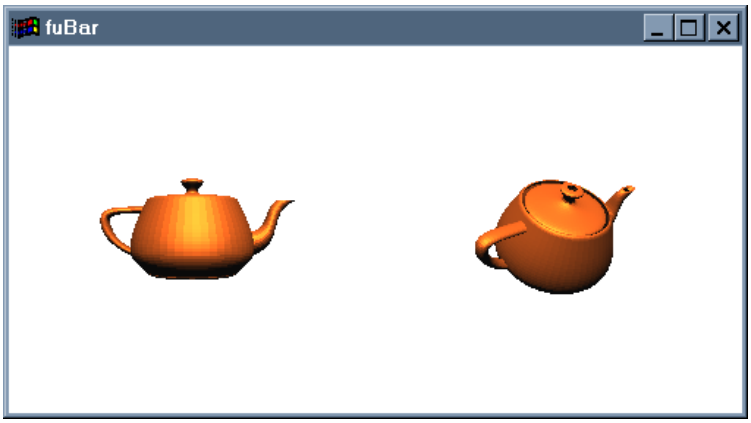
Viewport specifikuje oblast na obrazovce, do které se bude vykreslovat.

Viewport je možné nastavit pomocí příkazu glViewport s parametry x, y, které definují dolní levý roh viewportu a w, h, které určují velikost obdélníku viewportu.

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

Jako příklad nastavení viewportu si uvedeme rendering do dvou viewportů.

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



Transformace souřadnice Z

Funkce *glDepthRange* určuje mapování hodnot Z z normalizovaných souřadnic zařízení do souřadnic okna. Implicitní hodnoty parametrů:

- *near* = 0.0, kde near definuje mapování bližší ořezávací roviny na souřadnice okna.
- *far* = 1.0, kde far definuje mapování vzdálenější ořezávací roviny na souřadnice okna.

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

Vše v OpenGL je transformováno na interval [0.0, 1.0].

Ořezávací roviny

Ořezávací roviny se vytvářejí následujícím způsobem. Rovnice roviny se pošle do vertex shaderu jako běžná uniform proměnná. Vertex shader pak udělá výpočet vzdálenosti od zadané roviny sám a výsledek uloží do proměnné *gl_ClipDistance[i].* Tato hodnota je pak interpolována jako každá jiná a pixely, které by měly vzdálenost zápornou, jsou ořezány.

Je ale pravda, že se to stále musí povolit pomocí glEnable, nyní ovšem s parametrem GL_CLIP_DISTANCEi, kde i je 0 až minimálně 7.

Abychom mohli danou ořezávací rovinu využít, musí být povolena pomocí zadání příkazu *glEnable(GL_CLIP_DISTANCEi)*.

Stereo viewing

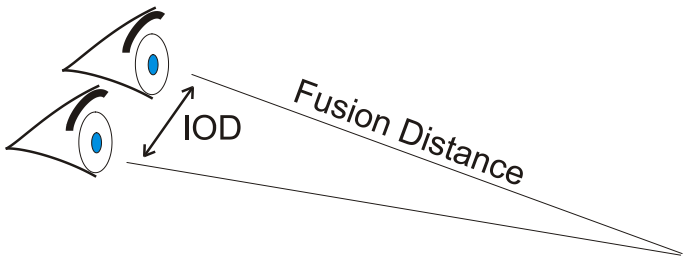
Stereo viewing je běžnou technikou, která je využívána pro zvýšení realističnosti 3D scén nebo zlepšení uživatelské interakce. Principem je vytvoření dvou pohledů na scénu, zvlášť pro levé a pravé oko. Ačkoliv knihovna GLUT poskytuje prostředky pro vytvoření stereo pohledu, je potřeba nad stereo zobrazením mít občas větší kontrolu. Využijeme skutečnosti, že OpenGL podporuje několik „rendering bufferů". Abychom mohli vykreslit daný frame ve stereomódu, je potřeba splnit následující předpoklady:

- Displej musí tuto funkci podporovat
- Levé/pravé oko musí být vykresleno v levém/pravém zadním bufferu
- Zadní buffery musí být náležitě zobrazeny – dle potřeb použitého hardware

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

Abychom mohli stereo pohled nastavit co nejreálněji, musíme vzít v úvahu dva parametry:

- IOD (interocular distance) – vzdálenost mezi očima
- Fusion distance – vzdálenost očí od pozorovaného objektu



Nastavení probíhá pomocí funkce LookAt.

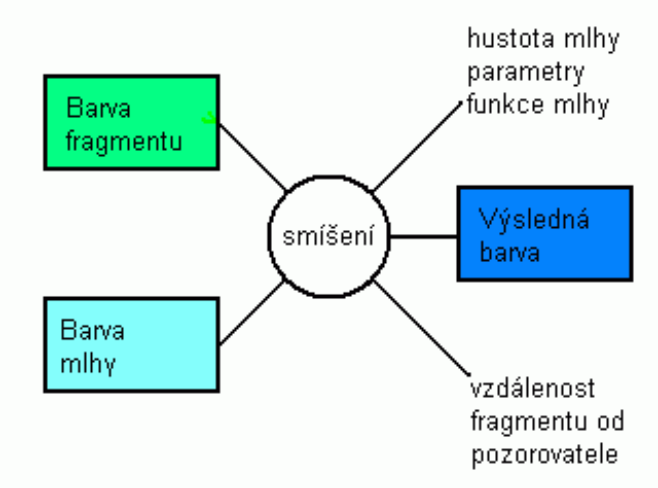
Mlha (fog)

Pomocí shaderů lze dosáhnout různých efektů a jedním z nich je *mlha* (*fog*). Jedná se o grafický efekt, při kterém se se vzrůstající vzdáleností od pozorovatele postupně mění barva vykreslovaných těles. Lze tak simulovat atmosférické jevy, jako například opar, kouř atd. V blízkosti pozorovatele (tj. malé vzdálenosti) má těleso svoji původní barvu, ve velké

vzdálenosti pak barvu mlhy. Pokud je barva mlhy totožná s barvou pozadí, dosáhne se vcelku realistického efektu, kdy se tělesa ztrácí v mlze nebo ve tmě. Barva mlhy a pozadí však může být odlišná, čehož lze využít pro tvorbu různých nerealistických efektů.

Ukázku použití této metody znázorňuje daný obrázek. Je patrné, že pouze travnatý povrch je zobrazen svou původní barvou (resp. barvou textury), vzdálenější povrch i budova se však ztrácí v mlze, která je, podobně jako pozadí, nastavena na šedou barvu.

Na první pohled vypadá tento efekt velmi složitě. Ve skutečnosti je jeho princip velice jednoduchý. Pomocí příkazů z grafické knihovny OpenGL se zadají základní vlastnosti mlhy, což ve skutečnosti představuje výběr jedné ze tří funkcí a nastavení parametrů těchto funkcí. Potom se při výpočtu barvy každého fragmentu získá jeho barva bez působení mlhy (tj. použije se například texturování nebo osvětlení) a následně se tato barva mísí s předem zadanou barvou mlhy. Poměr původní barvy fragmentu a barvy mlhy ve výsledné barvě je dán parametry funkce mlhy a vzdáleností fragmentu od pozorovatele (tato vzdálenost se stejně musí počítat, minimálně pro potřeby hloubkového bufferu). Princip tohoto efektu je naznačen na obrázku.



Intenzita mlhy závisí na vzdálenosti a rovnici mlhy:

$$Color = f\ Color_{incoming} + (1-f)\ Color_{fog}$$

Důležité je zmínit, že můžeme vykreslit pouze viditelné objekty.

Jak jsme již zmínili, efekt mlhy funguje tak, že se kombinuje původní barva fragmentu s barvou mlhy. To, jakým způsobem (resp. v jakém poměru) budou tyto barvy smíšeny, závisí na třech faktorech:

- vzdálenosti fragmentu od pozorovatele

- nastavené funkci hustoty mlhy

-parametrech funkce hustoty mlhy

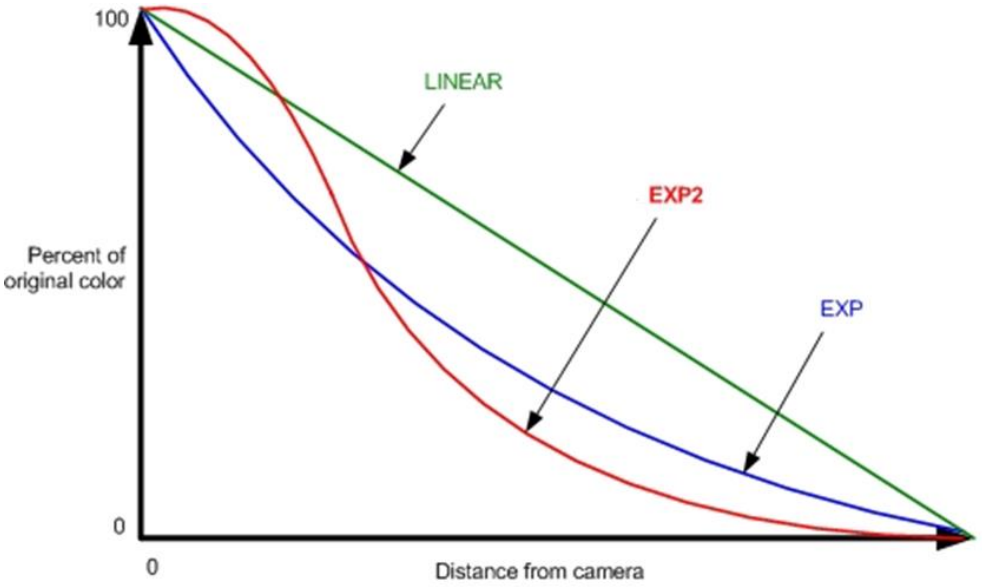
Pro výpočet hustoty mlhy v místě fragmentu je možné definovat například následující tři typy funkcí. Tyto funkce vlastně určují, jakým způsobem se bude měnit barva těles se vzdáleností od pozorovatele:

1. Nejjednodušší funkcí je lineární funkce. Tuto funkci lze vyjádřit vztahem *f=(end-z)/(end-start)*, kde *start* a *end* určují počáteční a koncovou vzdálenost, ve kterých hodnoty lineárně klesají. Mimo rozsah těchto parametrů se funkční hodnoty nepočítají. Výhodou je při použití této funkce vysoká rychlost zobrazování, ale efekt mlhy neodpovídá realitě.

2. Poněkud složitější funkcí je exponenciální funkce se záporným exponentem, tj. funkce je klesající: $f=e^{-density*z}$, kde *density* je hustota mlhy a *z* je vzdálenost fragmentu od pozorovatele. Tato funkce je na výpočet poněkud složitější než funkce předchozí, ale vizuální výsledky lépe odpovídají realitě.

3. Poslední, nejsložitější funkcí, je $f=e^{-(density*z)2}$. Člen *density* představuje opět hustotu mlhy, *z* je vzdálenost fragmentu. Výpočet této funkce je nejpomalejší, výsledky však nejlépe odpovídají realitě, kde se také intenzita světla snižuje se čtvercem vzdálenosti. Použití této funkce na počítači má však za následek velmi rychlé „mizení“ objektů.

Následující obrázek ukazuje vliv typu funkce a parametru hustoty mlhy na průběh směšovací hodnoty.



Při definování mlhy si vytvoříme vlastní uniform proměnnou, například pomocí následující struktury:

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

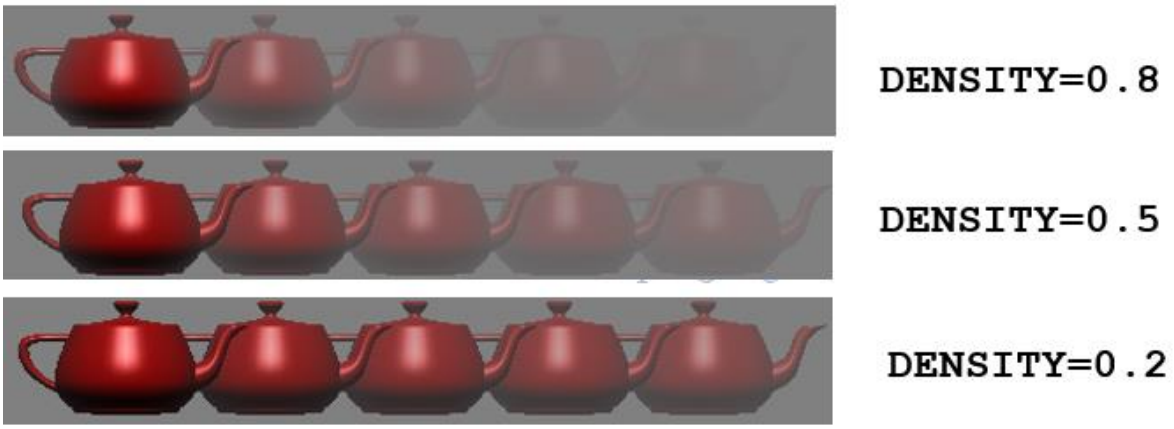
Poté bude příslušný vertex shader vypadat například takto:

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

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

Ve fragment shaderu pak vypadá použití například takto:
`out_color = mix(MyFog.color, finalColor, fogFactor);`
Tento příkaz umožní vytvořit interpolaci barvy mezi barvou mlhy (*MyFog.color*) a barvou fragmentu (*finalColor*) a to s ohledem na hodnotu *fogFactor* mlhy.

Příklad různého nastavení hustoty mlhy:



Blending (míchání barev)

Již jsme se seznámili s definicí barvy v OpenGL zadané pomocí tří barevných složek RGB. Pokud je barva zadána pomocí čtyř hodnot, je čtvrtá složka označována jako **alfa** hodnota, která označuje průhlednost. Pokud je míchání barev povoleno, je alfa hodnota použita pro kombinování barvy aktuálně zpracovávaného fragmentu s barvou odpovídajícího pixelu, která je již uložena ve framebufferu.
Alfa hodnoty mohou být rovněž využity v tzv. alfa testu, kdy je daný fragment na základě této hodnoty postoupen dalšímu zpracování nebo „zamítnut“.

Pomocí alfa složky (někdy také nazývané alfa-kanál) lze specifikovat míru průhlednosti resp. neprůhlednosti vybraných objektů nebo jejich plošek. Alfa složku lze také nastavit samostatně pro každý texel ve vykreslované textuře, čehož se využívá při programování mnoha grafických efektů, například výbuchů či „magických jevů“ (viz obrázky).
Při spuštění programu, který používá pro vykreslování grafickou knihovnu OpenGL, je vliv alfa složky na vykreslovanou plošku zakázán. Proto pokud chceme programovat některé grafické efekty, musíme před vykreslením vhodně nastavit režim míchání již nakreslené části scény s nově vykreslovanými tělesy.
Při použití *blendingu* (tj. míchání barev) musíme nejdříve specifikovat, jakým způsobem se budou kombinovat právě vykreslované *fragmenty* s hodnotami uloženými ve *framebufferu*. Připomeňme, že *fragment* je datová struktura složená z barvy pixelu, jeho průhlednosti, vzdálenosti od pozorovatele a případných dalších informací. *Framebuffer* je ve své podstatě pravidelná matice fragmentů. Barvy fragmentů tvoří ve framebufferu samostatný *color-*

buffer, který se zobrazuje na obrazovce. *Rasterizace* je proces, kterým se matematický model plošky (polygonu) převádí na jednotlivé fragmenty.
Blending musí být povolen pomocí funkce *glEnable(GL_BLEND).*

Rovnice míchání barev

Bez použití míchání každý nový fragment přepíše aktuálně uložené hodnoty barev ve framebufferu, což odpovídá neprůhledným povrchům. Pomocí míchání můžeme specifikovat a ovlivňovat, jakým způsobem bude aktuální barva zkombinována s novou hodnotou barvy fragmentu. Způsob míchání barvy uložené ve framebufferu a barvy vykreslovaného fragmentu se řídí uživatelem definovanou rovnicí míchání (*blending function*).

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

V této rovnici vystupují následující členy:
Zdroj (*source*) je fragment vzniklý rasterizací v právě běžícím rasterizačním procesu.
Cíl (*destination*) je hodnota zapsaná ve framebufferu, tj. barva fragmentu, který již byl vykreslen dříve. Tato hodnota bude v závislosti na nastavené blending funkci přepsána nebo jinak ovlivněna.
V knihovně OpenGL lze stanovit koeficienty míchání pro barvu a průhlednost zvlášť.

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

Následující tabulka popisuje možné parametry funkce glBlendEquation a glBlendEquationSeparate.

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	$R = \max(R_s, R_d)$ $G = \max(G_s, G_d)$ $B = \max(B_s, B_d)$	$A = \max(A_s, A_d)$

Míchací rovnici (původně ve vektorovém tvaru), která se při blendingu používá pro výpočet nové barvy fragmentu, lze rozepsat do čtyř rovnic odpovídajících barvovému modelu RGBA:

$R_n=R_sS_r+R_dD_r$

$G_n=G_sS_g+G_dD_g$

$B_n=B_sS_b+B_dD_b$

$A_n=A_sS_a+A_dD_a$

Kde R, G, B označují barevné složky, A je hodnota alfa kanálu, S je označení zdrojového fragmentu, D je označení cílového fragmentu a n je index nově spočtených hodnot R, G, B, A daného fragmentu.

Koeficienty S a D se nezadávají přímo číselnou hodnotou, protože se mohou měnit v závislosti na barvách zdrojových a cílových fragmentů. Místo toho se používají symboly, jejichž konkrétní hodnota se vypočte automaticky při rasterizaci.

Pro zadání míchacích koeficientů se používá funkce

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

kde první parametr *src* určuje způsob výpočtu míchacích faktorů S_r, S_g, S_b a S_a a druhý parametr *dst* způsob výpočtu faktorů D_r, D_g, D_b a D_a.

Faktor míchání může být různý pro barvu a průhlednost:

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

Příklad využití faktorů:

Ne všechny kombinace možných zdrojů a cílů dávají smysl. Většina aplikací využívá pouze malou podmnožinu možných kombinací. Nyní si uvedeme několik příkladů nejčastěji používaných kombinací.

- Smíchání dvou obrázků v daném poměru (viz dále)
- Míchání tří obrázků ve stejném poměru (viz dále)
- Aplikace jednoduchého filtru – pokud máme zdroj nastaven na hodnotu GL_DST_COLOR nebo GL_ONE_MINUS_DST_COLOR a cílový faktor nastaven na hodnotu GL_SRC_COLOR nebo GL_ONE_MINUS_SRC_COLOR, můžeme efektivně upravovat každou barevnou komponentu zvlášť. Tato operace je ekvivalentní aplikování jednoduchého filtru (například vynásobení červené komponenty 80ti procenty, zelené komponenty 40ti procenty a modré komponenty 72 procenty simuluje pohled na scénu skrz fotografický filtr, který blokuje 20% červeného světla, 60% zeleného a 28% modrého světla).
- Vytvoření efektu „neobdélníkového“ rastrového obrázku přiřazením různých alfa hodnot jednotlivým fragmentům obrázku. Každému „neviditelnému“ fragmentu je

přiřazena hodnota alfy 0 a každý neprůhledný fragment má naopak hodnotu alfy 1. Příkladem může být vykreslení polygonu ve tvaru stromu (viz obrázek), na který je aplikována texturová mapa listí. Uživatel vidí skrz části obdélníkové textury, které nejsou součástí stromu. Tato technika se nazývá **billboarding** a je mnohem rychlejší, než vytváření stromu pomocí 3D polygonů. Proto je také využíván v herním průmyslu. Hlavním principem je to, při rotaci je vždy objekt natočen čelní stranou k pozorovateli.

Pomocí dalších kombinací lze dosáhnout různých zajímavých efektů, jako například matting, skládání atd.

Jako příklad na použití blendingu si uvedeme jednoduchý problém. Máme zobrazit dvě plošky přes sebe, ale při překrytí má být bližší ploška průhledná z 50% Postup řešení tohoto problému je následující:

- Blending globálně povolíme příkazem *glEnable(GL_BLEND)*.
- Nastavíme zdrojový faktor na hodnotu *GL_ONE*.
- Nastavíme cílový faktor na hodnotu *GL_ZERO*.
- Vykreslíme první (spodní) plošku. Tato ploška je vykreslena svou originální barvou, protože cílový fragment (tj. pozadí) je vynulován a barva plošky je vynásobena jedničkou.
- Nastavíme zdrojový faktor na hodnotu *GL_SRC_ALPHA*.
- Nastavíme cílový faktor na hodnotu *GL_ONE_MINUS_SRC_ALPHA*.
- Vykreslíme druhou (vrchní) plošku. Alfa hodnota barvy této plošky musí být nastavena na 0.5. To znamená, že se zdrojový i cílový faktor vynásobí stejnou hodnotou a posléze se sečtou. Výsledkem našeho snažení je, že tato druhá ploška je vykreslena s padesátiprocentní průhledností.

Pokud bychom chtěli obrázek složit ze 75ti procent z prvního obrázku a z 25ti z druhého, stačí vykreslit první obrázek stejně jako výše a druhému obrázku nastavit hodnotu alfa na 0.25.

Pořadí, ve kterém jsou polygony vykreslovány, významně ovlivňuje výsledek míchání. Pokud vykreslujeme 3D průhledné objekty, můžeme získat zcela odlišné výsledky v závislosti na tom, zda vykreslujeme jejich polygony zezadu dopředu nebo zepředu dozadu.
Pro určení správného pořadí polygonů je nezbytné využívat hodnoty obsažené v hloubkovém (depth, Z) bufferu. Hloubkový buffer se obvykle využívá pro eliminaci skrytých polygonů nebo jejich částí.
Typicky chceme vykreslovat jak neprůhledné, tak průhledné objekty zároveň do jedné scény, přičemž využíváme hloubkový buffer pro odstranění objektů ležících za neprůhlednými objekty. Pokud neprůhledný objekt zastiňuje jiný průhledný nebo neprůhledný objekt, chceme po hloubkovém bufferu, aby tento vzdálenější objekt odstranil. Pokud je ale průhledný objekt blíže pozorovateli, pak jej chceme smíchat se zadním neprůhledným objektem. U statických scén není toto určení žádným problémem. Ten nastává, pokud se pozorovatel nebo scéna pohybuje.

Řešením je zapnout depth-buffering, ale pokud vykreslujeme průhledné objekty, pak musíme nastavit hloubkový buffer pouze pro čtení. Nejdříve vykreslíme všechny neprůhledné objekty pomocí klasického nastavení hloubkového bufferu. Poté hloubkový buffer nastavíme pouze pro čtení, abychom toto nastavené pořadí zachovali. Poté když vykreslujeme průhledné objekty, porovnáváme jejich hloubku s hloubkou neprůhledných objektů uložených v hloubkovém bufferu, tedy nejsou vykresleny v případě, že jsou zakryty neprůhlednými objekty. Pokud jsou blíže k pozorovateli, neeliminují neprůhledné objekty, protože hloubkový buffer nemůže být modifikován. Místo toho jsou s nimi smíchány.
Pro určení, zda je pro hloubkový buffer povolen zápis, lze použít funkci DepthMask(). Pokud nastavíme parametr této funkce na FALSE, buffer bude pouze pro čtení. Použití parametru TRUE znovu obnoví normální stav bufferu, tedy bude přístupný i pro zápis.

Funkce glFlush a glFinish

Příkazy můžou být v určitých implementacích GL bufferovány v tzv. *command queue* a jsou pak poslány k provedení najednou. Tuto frontu příkazů můžeme ovládat pomocí dvou příkazů.

Funkce glFlush způsobí to, že všechny dříve zavolané GL příkazy budou dokončeny v konečném čase.

```
void glFlush(void)
```

Funkce glFinish přinutí všechny dříve zavolané GL příkazy, aby skončily. Funkce neskončí, dokud neskončí tyto příkazy.

```
void glFinish(void)
```