Buffery, obrázky, textury

Framebuffer

- Soubor bufferů nutných k vytvoření snímku
  - vertex buffer (VBO)
  - element buffer (EBO)
  - color buffer
  - depth buffer (Z-buffer)
  - stencil buffer
    - Šablona
- Také
  - Frame buffer object (FBO)
    - Off-screen rendering
  - Pixel buffer object (PBO)
    - Rychlejší transfer textur

Buffer

- Buffer = lineární paměť v prostoru GPU
  - identifikovaná jménem (Gluint)
  - Alokovat ID
    glCreateBuffers(), glGenBuffers()
  - Zvolit jako aktivní
    glBindBuffer
  - Získat data
    - Naplnit daty
      glBufferData()
    - Namapovat existující data v prostoru CPU
      glMapBuffer()

Výběr colorbufferu pro kreslení

- glDrawBuffer(GLenum buffers)
  - až 4 najednou
  - použije aktuální framebuffer
- glNamedFramebufferDrawBuffer(GLuint framebuffer,GLenum buf)
  - libovolný buffer z libovolného framebufferu
- Některé buffery
  - double-buffering
    - GL_FRONT, GL_BACK
  - stereoskopické vykreslování
    - GL_FRONT_RIGHT, GL_BACK_LEFT, …
  - oba najednou (pozadí, neměnné části...)
    - GL_FRONT_AND_BACK
- Zpětné čtení (např. screenshot)
  - glReadPixels(x_start, y_start, width, height, format, type, *pixels)

Zjišťování vlastností bufferu

- glGetIntegerv(GLenum pname, GLint *vysledek)
  - zjistí jeden parametr
    - GL_RED_BITS, GL_GREEN_BITS, GL_BLUE_BITS, GL_ALPHA_BITS, GL_DEPTH_BITS, GL_ACCUM_RED_BITS, …
- int glfwGetWindowAttrib (GLFWwindow *window, int attrib)
  - součást knihovny GLFW, jiná množina parametrů

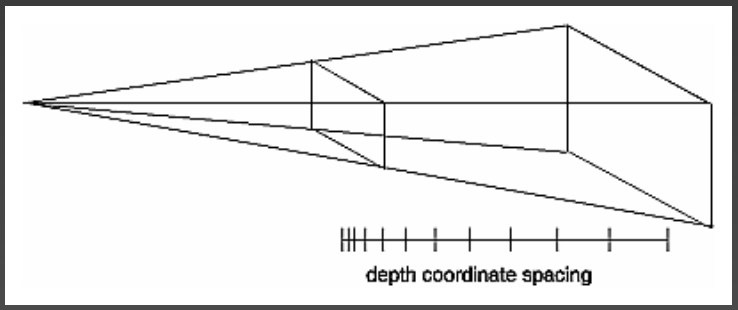
    GLFW_FOCUSED, GLFW_MAXIMIZED, ...

Color buffer

- Nejméně jeden
- Barevná informace fragmentů
  - použité pro vykreslování
  - většinou RGBA (paleta je zastaralá)
- Pro správné vykreslování dva buffery (ev. více)
  - double-buffering: GLFW_DOUBLEBUFFER
- Pro stereoskopické vidění dva (čtyři) buffery
  - levý a pravý: GLFW_STEREO, GL_LEFT, GL_RIGHT
- Možné i další – omezeno jen kapacitou paměti

Depth buffer

- Paměť hloubky, většinou jen jedna
- Rozsah near…far mapován na 0.0f … 1.0f
  - standardně mazáno hodnotou 1.0
    - přenastavit lze: glClearDepth(GLdouble depth)
- Pro vykreslování viditelných částí těles
  - zakrytá část má větší hloubku a nevykreslí se
- Lze různě nastavovat a používat pro efekty
  - stíny (nejdou přímo) apod.
- Důležité pohlídat si bitovou hloubku
  - může být i jen 8 bitů = 256 úrovní
    - artefakty
  - dnes spíše 16b, 24b, 32b, …

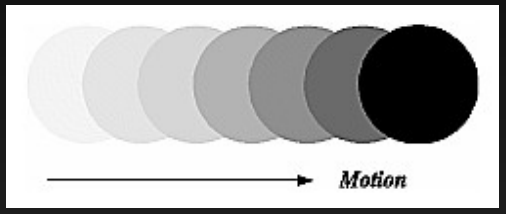


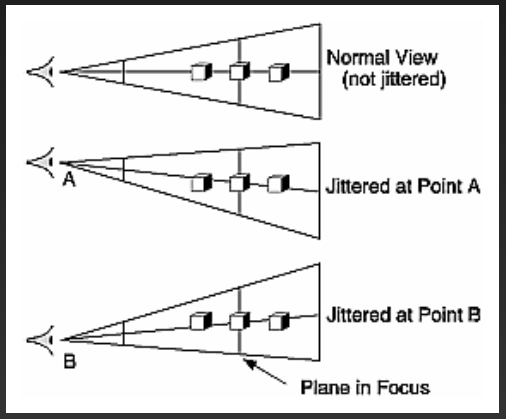
Stencil buffer

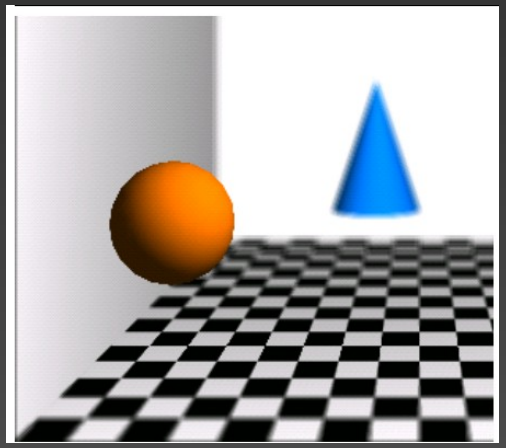
- Šablona pro maskování fragmentů
  - vylučuje zápis zamaskovaných fragmentů
- Rozdíl proti ořezu
  - ořezávání = vrcholy
  - šablona = fragmenty
- Funkce pro maskování je měnitelná
  - CSG, stíny, GUI...
- Většinou stačí 1b hloubka (maska), nebo 8b (rychlost)

Možné využití bufferů

- FBO+zpracování → colorbuffer
- Sloučení více obrazů do jednoho
  - Pro efekty – motion blur, depth-of-field (DOF) apod.
  - Antialiasing – do FBO vykreslení s vyšším rozlišením, pak průměrování
- Scéna vytvořena v samostatném FBO
  - přesun + další zpracování…
  - … do colorbufferu…
  - … swap buffers a zobrazení





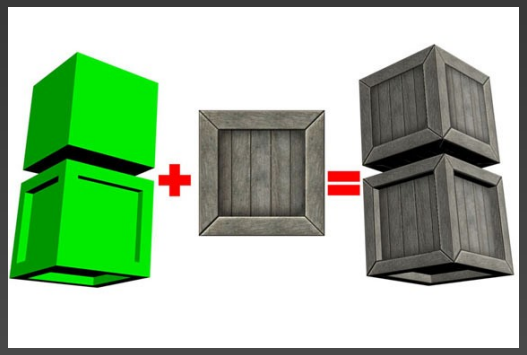


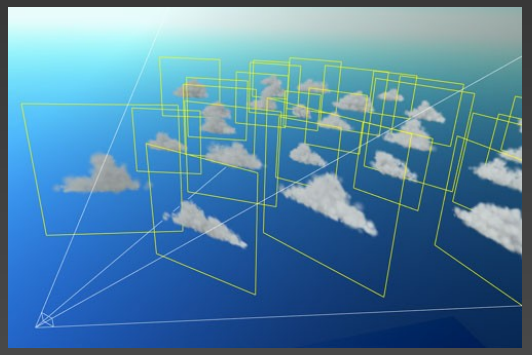
Textury

Textury

- Na danou geometrii nanášíme obrázek
  - v podstatě použije texturu jako zdroj per-fragment difuzního materiálu

- Pro vykreslení složitých těles bez nutnosti modelování přes plošky
  - tráva, stromy, mraky, kameny, zdi,...
  - nejsou to plošky – ztrácíme prostorovou informaci
  - billboarding – správně orientovaná průhledná textura





Speciální rastrové formáty – textury

Potřebujeme náhodný, malý kousek velkého obrázku (textury). Nemáme dost paměti ani času na kompletní dekompresi → PNG, JPEG atd. nelze použít!

- Potřebujeme
  - náhodný přístup, rychlost, dobrý kompresní poměr, kvalitu
- Obvykle dlaždicový ztrátový formát. Každá malá dlaždice nezávisle komprimovaná s podporou HW → takřka náhodný přístup, rychlé, efektivní.
- S3TC, DDS, DXT, PWR, ETC

Vlastnosti textur

- Typy textur
  - 1D, 2D (nejčastější), 3D
  - rastrové – běžný obrázek
  - procedurální – vzniklé matematickým výpočtem
    - výpočet předem
    - výpočet za běhu (shadery, volumetrické efekty, nižší rozlišení)
- Výhody
  - dostatečně kvalitní pro oklamání oka
    - zeď, kámen, písek, dřevo...
  - jednoduchá hardwarová implementace
- Nevýhody
  - rozlišení: nutné zvolit správný poměr kvalita / obsazená paměť
  - Rastr → filtrace + antialiasing → zpomaluje (přídavné paměťové přenosy)

Postup při texturování

1. Načtení nebo výpočet textury
2. Vytvoření texturovacího objektu
   - Přiřazení (bind), nastavení formátu
   - Nastavení aktivní texturovací jednotky
3. Odeslání uniform s číslem aktivní tex. jednotky
4. (kód shaderu) Výběr způsobu nanášení
5. Vykresli vertexy s novým atributem
   - texturovacími souřadnicemi

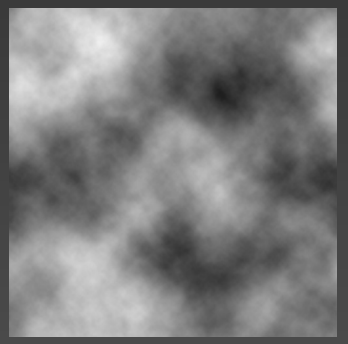
Načtení (výpočet) textury

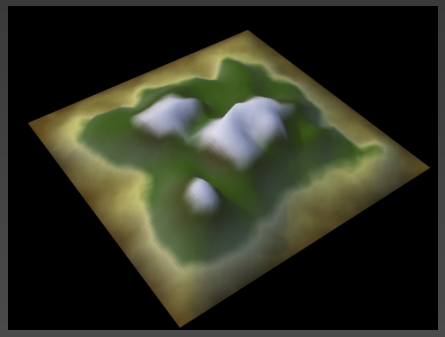
- Není přímá podpora pro textury → knihovny
- Čtverec, rozměr 2^n

```
NPOT: glewIsSupported(„GL_ARB_texture_non_power_of_two“);
   glewIsSupported(„GL_ARB_texture_rectangle“);
   (texcoords ne <0..1>x<0..1> ale <0..w>x<0..h> a další omezení)
```

- Procedurální textury
  - předepsány rovnicemi, výpočet obvykle náročný
  - vysoká kvalita (libovolné rozlišení)
  - volumetrické efekty: kouř, oheň
  - fraktály (Perlinův šum apod.): krajina, mraky...

```
#include <glm/gtc/noise.hpp>
glm::simplex(…)
glm::perlin(…)
```





Použití textury

- Vytvořit objekt textury, připojit (bind) texturu k texturovací jednotce, nastavit vlastnosti

```
GLuint texName[CNT];
glGenTextures(CNT, texName);
```

  - glBindTexture(GL_TEXTURE_2D, texName[0]);
    - standardně GL_TEXTURE0, změna glActiveTexture(unit)
  - glTexParameteri(…)
    - parametry aplikace textury
  - glTexImage2D(target, level, components, width, height, border, format, type, *pixels)
    - GL_TEXTURE_1D, GL_TEXTURE_2D
    - mipmap level, components (R, G, B, A), size, borders, format (GL_RGB, GL_BGRA, …), data type (GL_UNSIGNED_BYTE, …)
    - pointer na data v paměti

Způsoby nanesení textury (kód do shaderu)

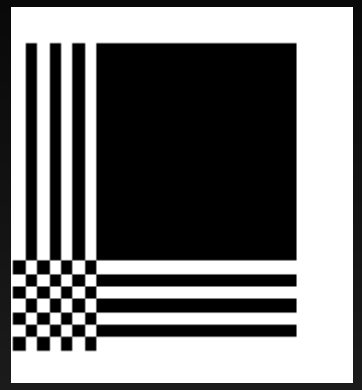
- Modulace
  - nejobvyklejší, barva podkladu z osvětlení násobená texturou
    $RGBA = RGBA_{texture} * RGBA_{light}$

    ```
    vec4 color_out = light_color * texture(texunit,coords);
    ```

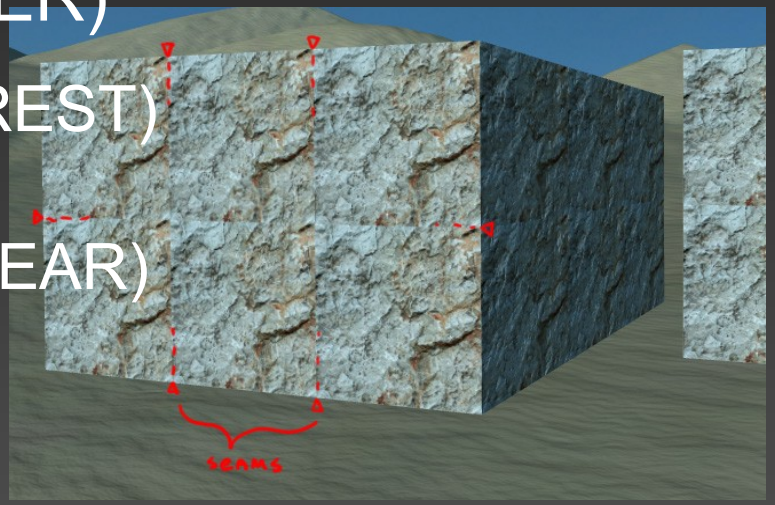
- Přímá aplikace
  - data textury přímo nanesena, včetně průhlednosti
    $RGB = (1-A_{texture})RGB_{light} + A_{texture}*RGB_{texture}$
    $A = A_{texture}$

    ```
    vec4 color_out = texture(texunit,coords);
    ```

Parametry nanášení



- glTexParameter{if}{v}(target, param, val)
- Opakování (GL_TEXTURE_WRAP_S, _T,_R)
  - jako dlaždice (GL_REPEAT,GL_MIRRORED_REPEAT)
  - opakovat krajní hodnotu (GL_CLAMP_TO_EDGE)
  - vyplnit zbylou oblast barvou (GL_CLAMP_TO_BORDER)
    - additional texture border color GL_TEXTURE_BORDER_COLOR
- Filtration (GL_TEXTURE_MIN_FILTER)
  - žádná = nejbližší soused (GL_NEAREST)
  - bilinear (GL_LINEAR)
  - trilinear (GL_LINEAR_MIPMAP_LINEAR)



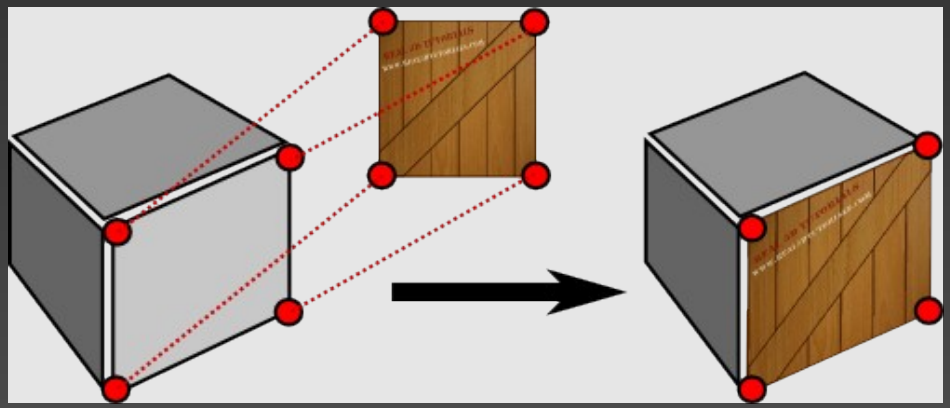
Texturovací jednotky a texturovací souřadnice

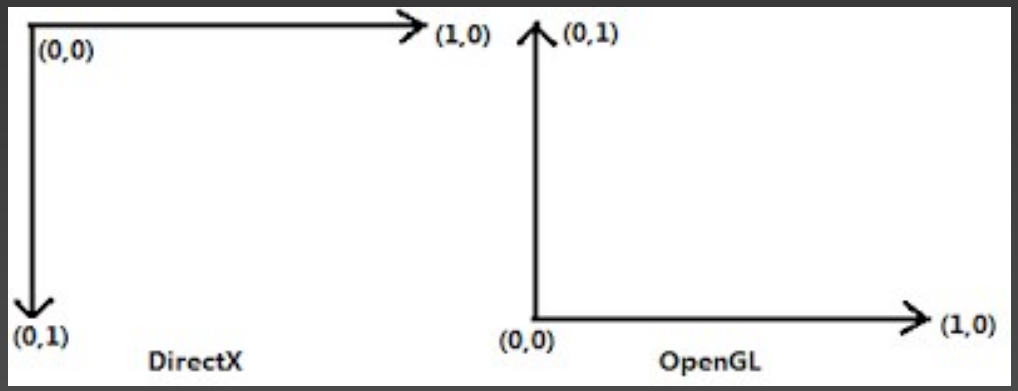
- Souřadnice
  - textura je definována jako čtverec o straně 1.0
    - rozpory v terminologii: ST vs. UV souřadnice (UV coordinates)

- Multitexturing
  - aplikace více textur najednou
  - Počet texturovacích jednotek je omezený
    glGetIntegerv(GL_MAX_TEXTURE_UNITS, &num)
  - Active unit: glActiveTexture(GL_TEXTUREx) + poslat uniform

    → glTexImage*(), glTexParameter*(), glTexEnv*(), glTexGen*(), glBindTexture()

```
vec4 out_color = texture(texunit1,texcoord1) * texture(texunit2, texcoord2)
```

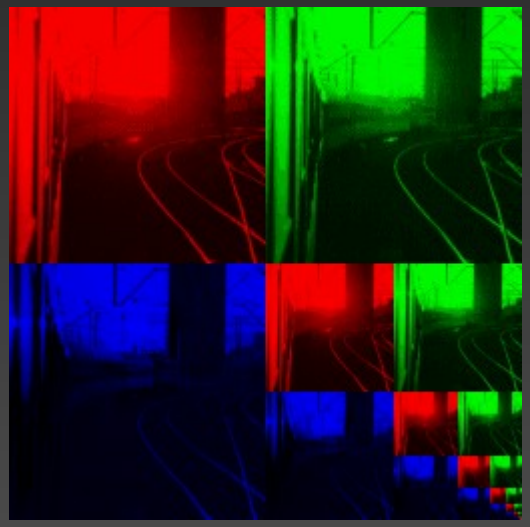


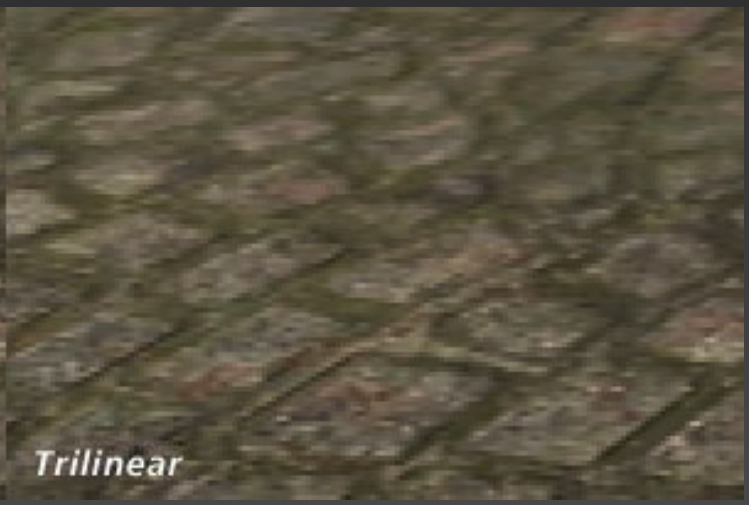


Mipmapping

- Pro snížení (odstranění) aliasu
- Pouze pro zmenšování
- Lineární interpolace ve třech osách
  - v textuře – s, t
  - mezi MIPMAP texturami podle Z vzdálenosti

glGenerateMipmap(GL_TEXTURE_2D)



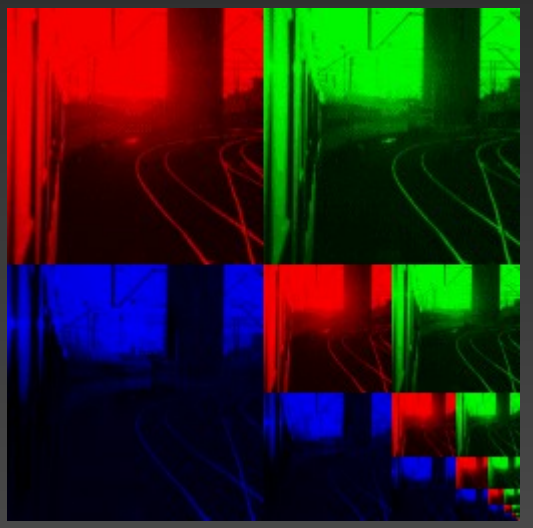


MIP map

- „multum in parvo“ - mnoho v malém
- Hierarcická reprezentace obrázku
  - spočítaná a filtrované předem, ne v reálném čase

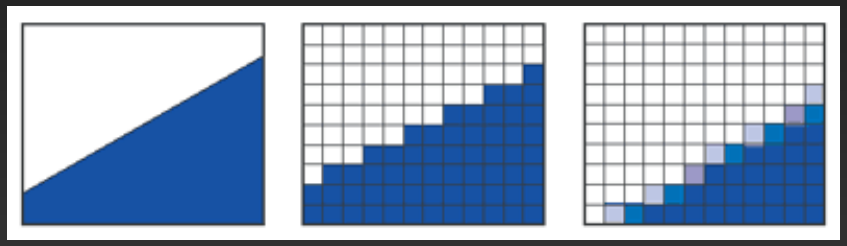
  + Při zmenšení zvyšuje kvalitu
  - šetří čas → zrychluje vykreslení (připravena offline, menší)
  - vyšší kvalita (lepší filtrace)
  - mapování textur na vzdálené objekty

  - Spotřeba paměti
  - 4/3 původní spotřeby
  - uložena v jediném obrázku
    - menší fragmentace paměti
  - nutné správně vytvořit, volit souřadnice



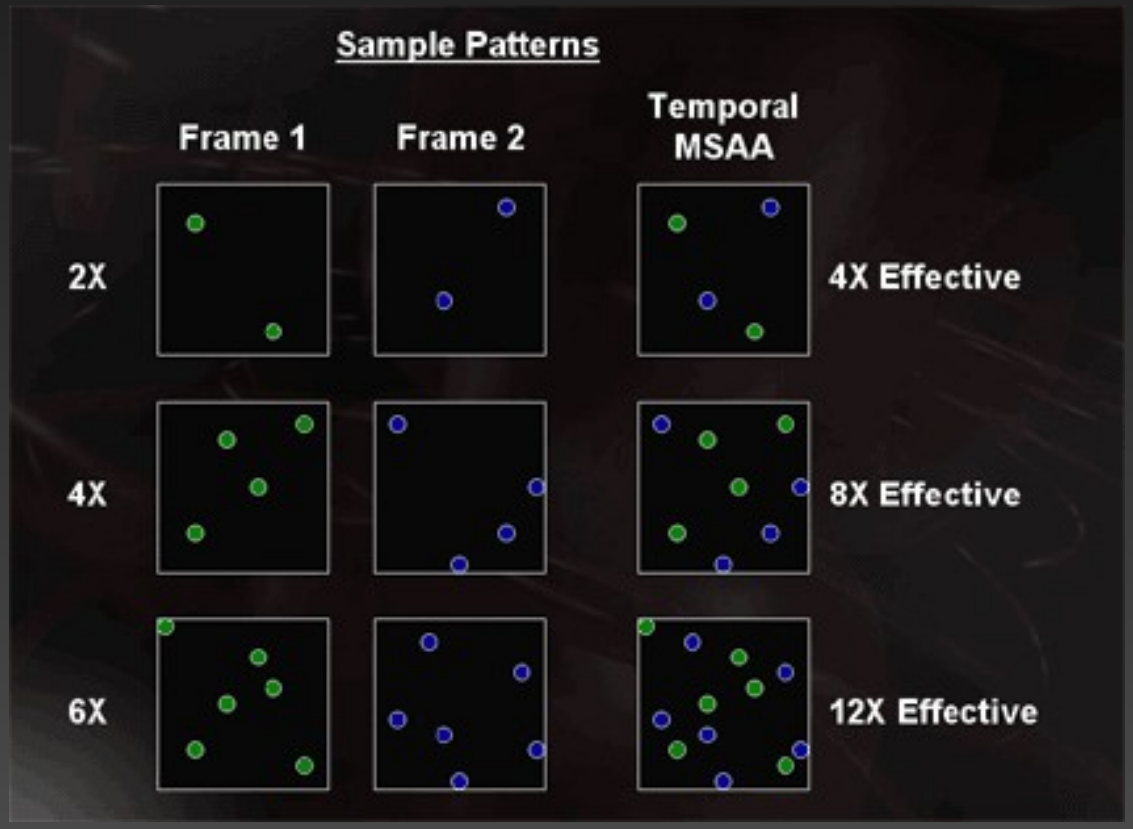
Antialiasing

- Malé rozlišení výstupního zařízení způsobuje alias – zubaté hrany, přetrhané tenké linie
- Vyhlazování hran pomocí průměrování více vzorků – samplů
- Temporální
  - využívá vzorky předchozích snímků (Nvidia TXAA apod.)
- Spatiální
  - více vzorků pro jeden pixel
  - vyšší kvalita



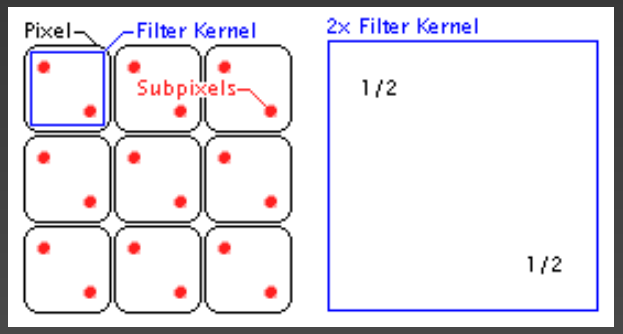
Temporální antialiasing

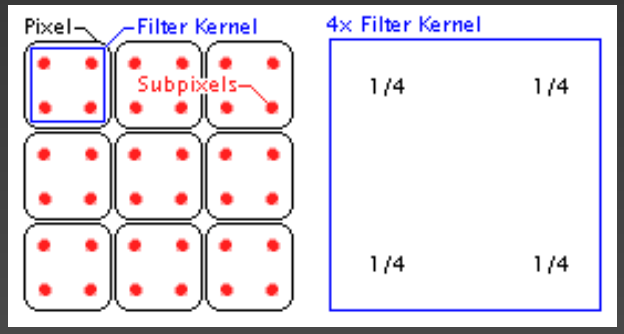
- Vzorky z předchozího snímku (snímků)
  - + malé výpočetní nároky
  - - rozmazává, chvějící se obraz při malém FPS

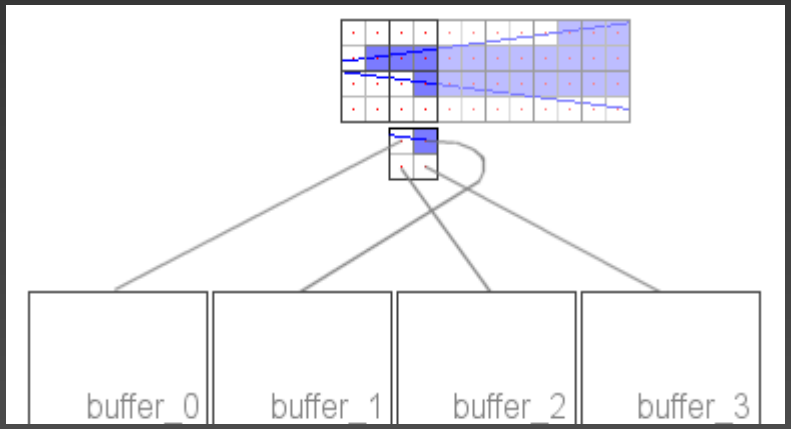


Supersampling
Full Scene Anti-Aliasing = FSAA

- Standardně je fragment obarven podle vzorku uprostřed pixelu
- 4FSAA – interně 4x více subpixelů na různých místech, pak průměr
  - + vyhlazuje nejen hrany, ale i textury, průhledné textury apod.
  - - značné nároky na paměť (multisample buffers) a výkon







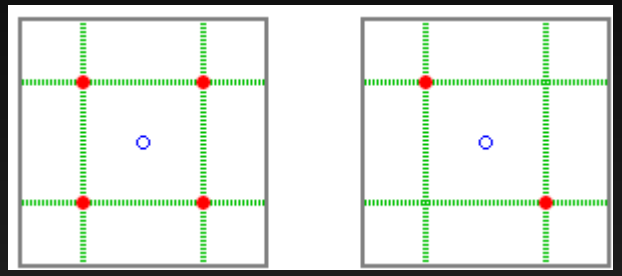
Multisampling

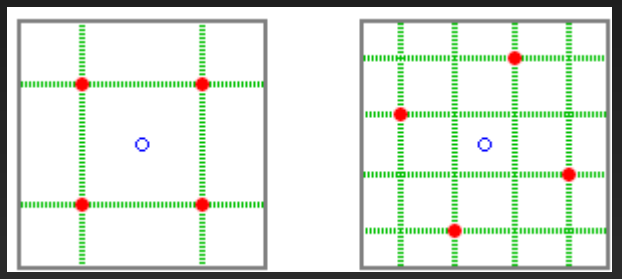
- Optimalizace
  - zjistí se jen příslušnost subpixelu k trojúhelníku
  - následně interpolace vstupů a fragment shader se spouští jen jednou
  - rozlišení zvětšené jen pro Z-buffer a stencil buffer
- Podpora? GL_ARB_multisample
- Různé metody volby samplů
  - snaha o minimum samplů tak, aby interní rozlišení maximálně narostlo

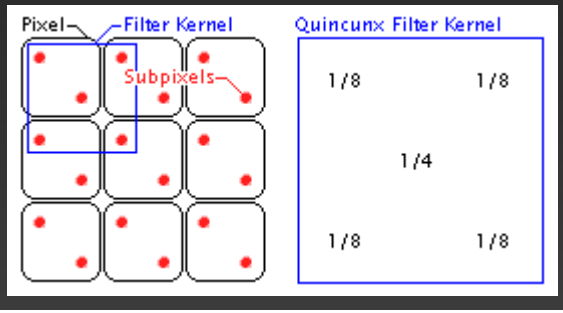
  glfwWindowHint(GLFW_SAMPLES, 4)

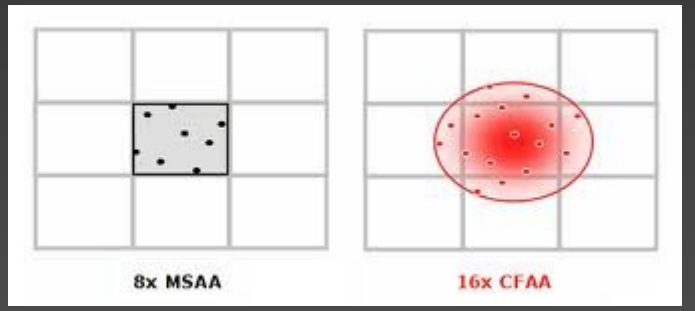
Příklady multisamplingu

- Obdobná kvalita
  (2x nárůst interního rozlišení)
- Lepší volba pozice subsample
- Využití okolních fragmentů
  - mírně rozmazává obraz
- Kombinace postupů





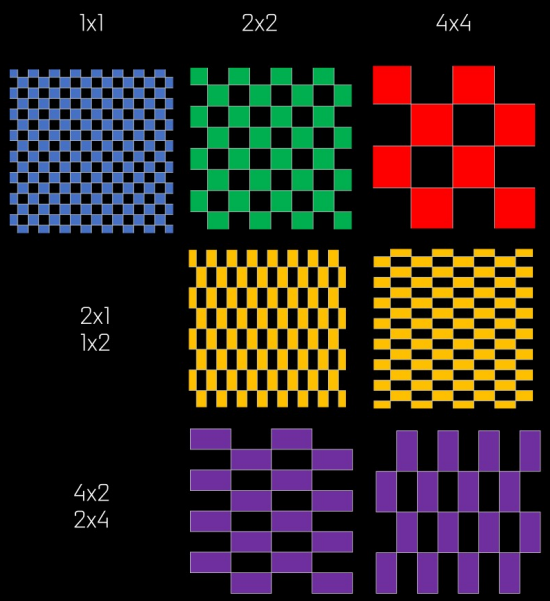


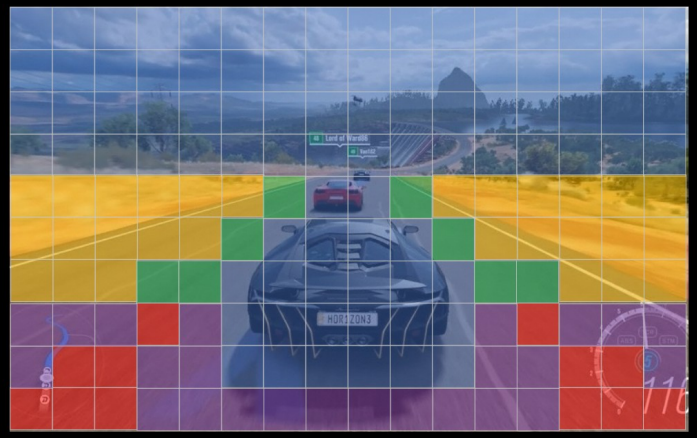


Speed Optimization

VARIABLE RATE SHADING

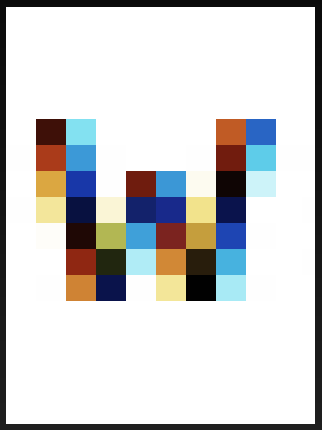
Shade More Where Needed

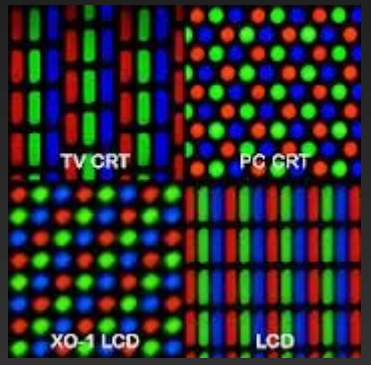


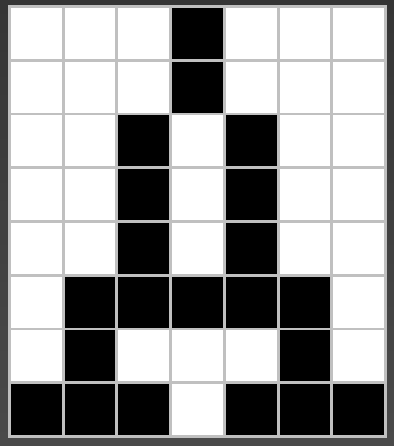


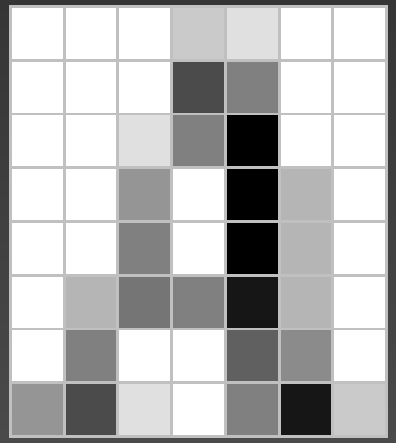
Sub-pixel antialiasing

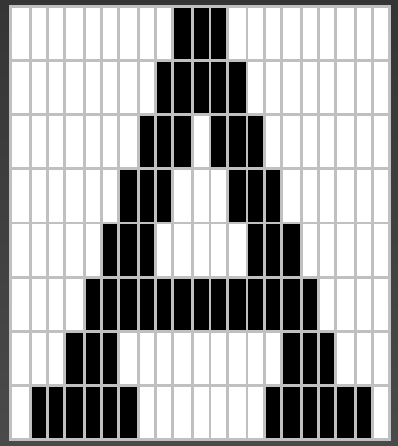
- Náročné na výpočet
- Použití především u písma
  - CoolType, ClearType, FreeType apod.
- Princip – využití R,G,B subpixelů
  - W = R+G+B = B+R+G = G+B+R
  - citlivé na pořadí subpixelů

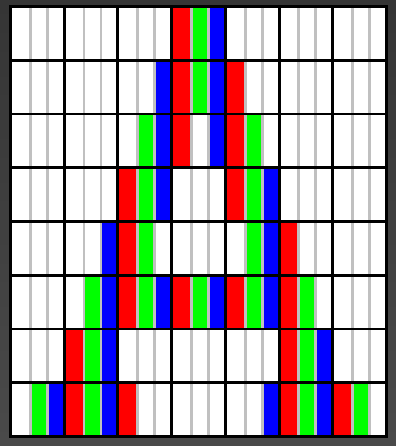


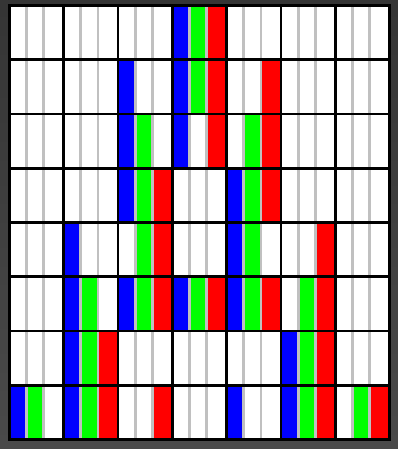












Shrnutí možností obarvení

- Vzestupně vizuální kvalita i výpočetní náročnost
  - Konstantní barva ve vertexu + interpolace v ploše

    vertex atributy: barva
  - Per fragment osvětlovací model a materiály

    vertex atributy: normála

    uniform: světla, materiály
  - Per fragment osvětlovací model, materiály a textury

    vertex atributy: normála, texturovací souřadnice

    uniform: světla, materiály, texturovací jednotky

1) Načtení a transformace souřadnic zadaných vertexů

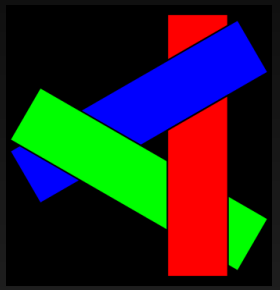
2) Rasterizace

3) Výpočet barvy fragmentu

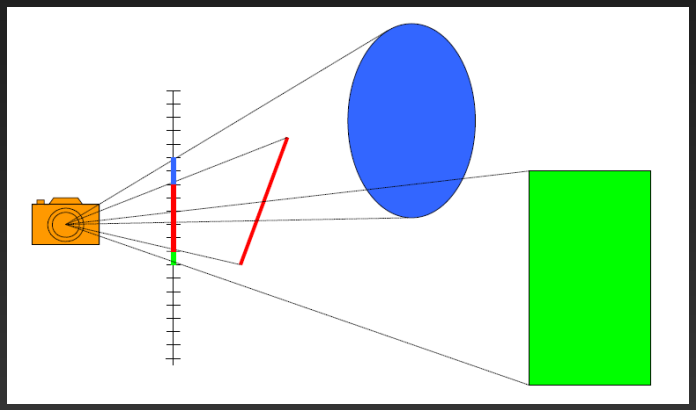
4) Průhlednost, mlha a zakrývání podle Z

Paměť hloubky

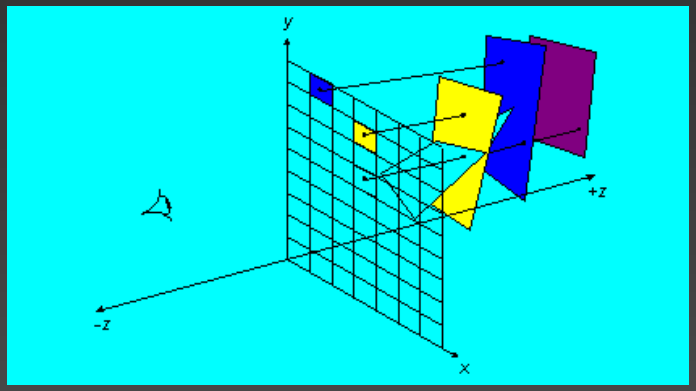
- Neboli Z-buffer (depth buffer)



- Nutná pro korektní zobrazení v situacích neřešitelných malířovým algoritmem



- Blokuje zápis fragmentů do výsledného obrázku
  - vzdálenější fragment je blokován



- Obvykle 1 Z-buffer na snímek

Použití paměti hloubky

- Without Z-Buffer (depth buffer) all polygons are displayed in draw order, no matter the distance and visibility
  - manually – painters algorithm – too hard for complex scenes
- For proper per fragment decision we need:
  1) enable depth test

     glEnable(GL_DEPTH_TEST)
  2) define „zero“ depth (usually not necessary)

     glClearDepth(depth)
     - from 0.0 to 1.0 (near plane to far plane)
  3) choose type of depth test

     glDepthFunc(func)

     GL_NEVER, GL_LESS, GL_EQUAL, GL_LEQUAL, GL_GREATER, GL_NOTEQUAL, GL_GEQUAL, GL_ALWAYS
     - most common GL_LESS or GL_LEQUAL (close hides distant)
  4) clear Z-buffer when starting new frame

     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)

Průhlednost

- RGBA, RGB+alfa kanál - „neprůhlednost“, krytí

  A=0.0 plně průhledné
- Určuje způsob kombinace vykreslovaných fragmentů s color-bufferem
  - fragment = barva + průhlednost + Z-souřadnice...
- Bez míchání barva fragmentu přepíše existující hodnotu
- Výpočet nové barvy podle zvolené funkce
  - smícháním stávající a nově příchozí

Míchání (blending)

- Zdroj = příchozí fragment
- Cíl = obsah již uloženého pixelu v color-bufferu
- Mísicí rovnice

$$R_n = R_s \cdot S_r + R_d \cdot D_r$$
$$G_n = G_s \cdot S_g + G_d \cdot D_g$$
$$B_n = B_s \cdot S_b + B_d \cdot D_b$$
$$A_n = A_s \cdot S_a + A_d \cdot D_a$$

- RGBA, S(D) – míchací faktor zdroje (cíle)
- Indexy: rgb, n – nový pixel, s – source, d – dest.

- glBlendEquation(GLenum mode)

  GL_FUNC_ADD, GL_FUNC_SUBTRACT, GL_FUNC_REVERSE_SUBTRACT, GL_MIN, GL_MAX

Nastavení mísicí funkce

- Shodné faktory pro RGB i A

  glBlendFunc(srcfactor, dstfactor)
- Různé faktory pro RGB, A

  glBlendFuncSeparate(srcRGB, dstRGB, srcA, dstA)
- Faktory vybírány z tabulky (viz OpenGL dokumentace)
  - nejběžnější příklad:
    - zdrojový faktor: GL_SRC_ALPHA

      nastaví pro RGBA faktor (As,As,As,As)
    - cílový faktor: GL_ONE_MINUS_SRC_ALPHA

      nastaví pro RGBA faktor (1-As,1-As,1-As,1-As)
    - výsledek (pro červenou složku) Rd=As*Rs+(1-As)*Rd

Použití

- Vykreslit všechna neprůhledná tělesa
- Povolit mísení

  glEnable(GL_BLEND)
- Nastavit faktory

  glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
- Zablokovat paměť hloubky pouze pro čtení
  - těleso je průhledné, neblokovat vykreslování když je něco za ním, chceme to vidět

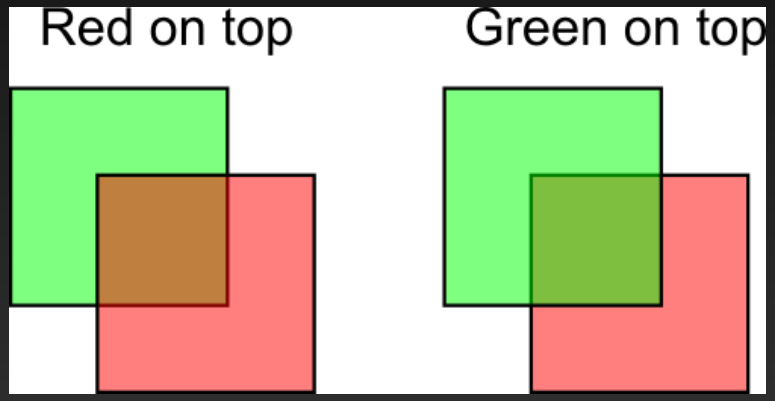
  glDepthMask(GL_FALSE)
- Vykreslit průhledná tělesa (odzadu dopředu, nebo pomocí OIT)
- Zablokovat mísení a povolit test hloubky

  glDisable(GL_BLEND)

  glDepthMask(GL_TRUE)

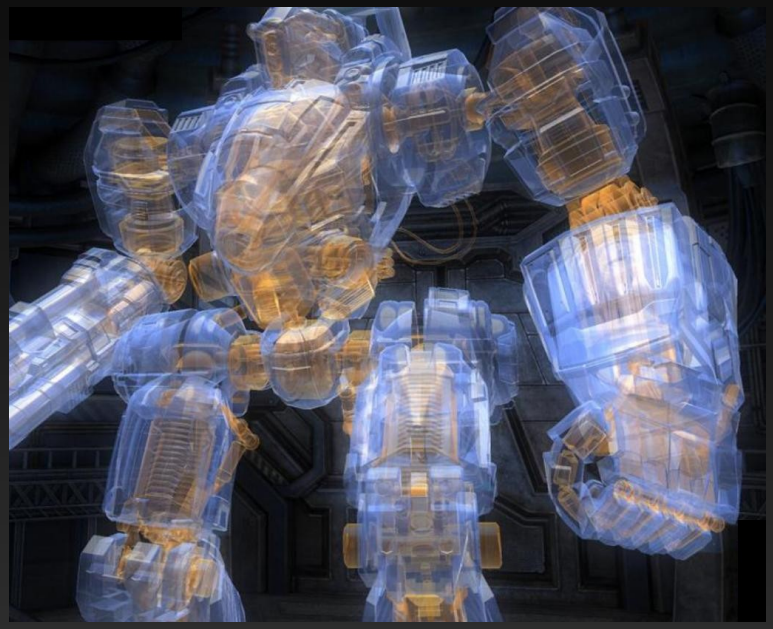
Závislost na pořadí operací

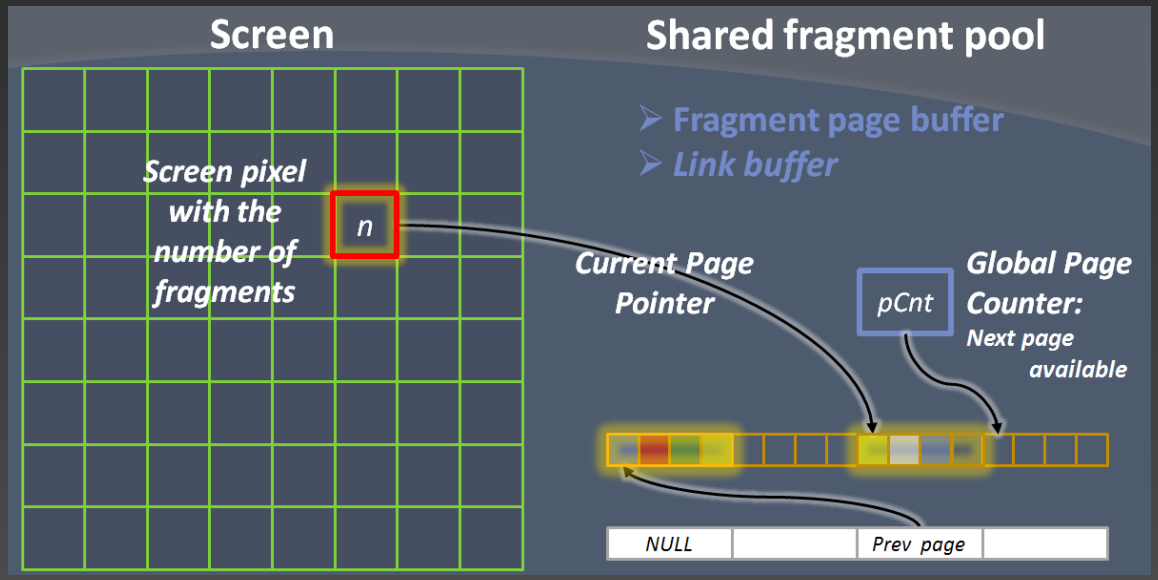
$$RGB_d = A_s \times RGB_s + \left(1.0 - A_s\right) \times RGB_d$$
$$RGB_d = RGB_d + A_s \times \left(RGB_s - RGB_d\right)$$

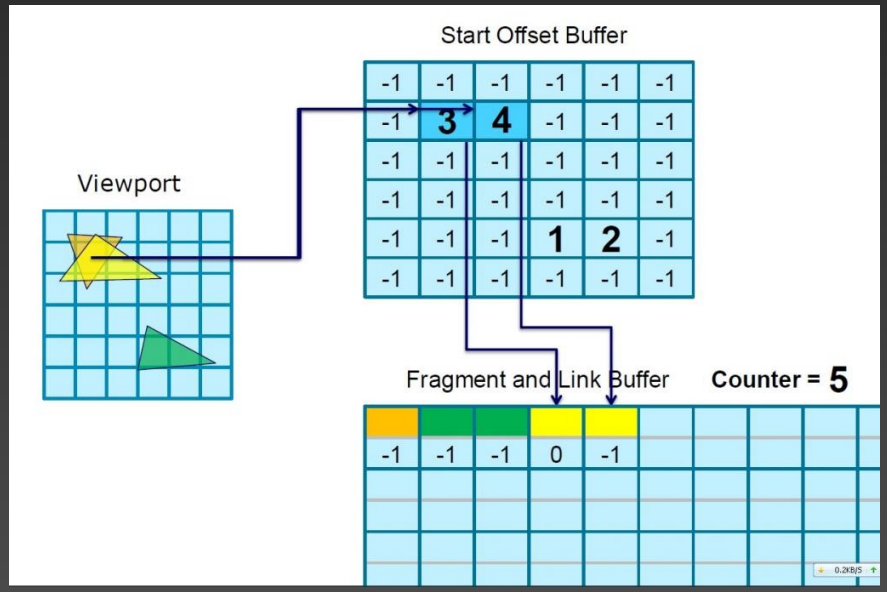


OIT
Order Independent Transparency

- A-Buffer, Depth peeling
- atomické čítače, HW 2011
  - OpenGL - 2012
  - D3D – 2015 (v.12)

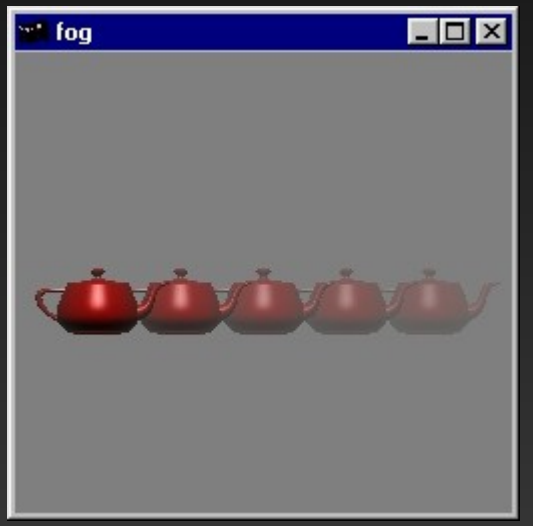






Mlha

- Hlavní důvody
  - zvýšení reálnosti
  - zvýšení rychlosti zobrazení
    - těsně za hustou mlhou nastavíme ořez
- Hustota se zvyšuje se vzdáleností
  - parametricky nastavitelné
  - různé matematické funkce mlhy
- Libovolná barva
  - obvykle šedá nebo černá (objekty mizí v dálce v oparu nebo ve tmě)



Možnosti mlhy

- Lineární

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

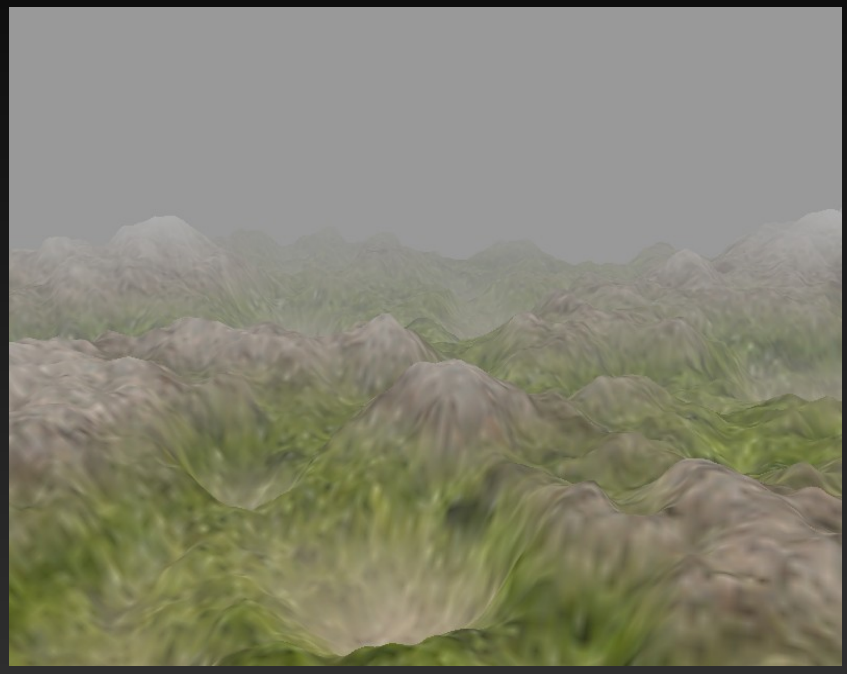
- Exponenciální

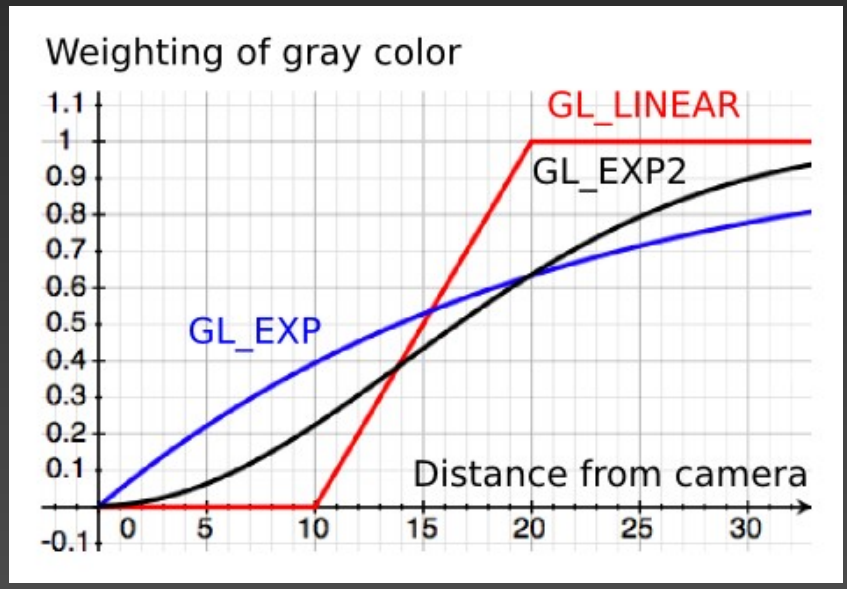
$$f = e^{-density \cdot z}$$

- Exponenciální kvadratická

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

  - nejlépe odpovídá realitě
  - kvadratický úbytek světla se vzdáleností





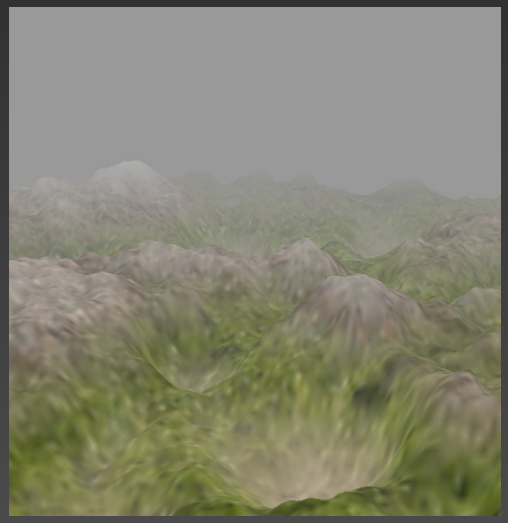
Fog in fragment shader

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

```
uniform float fog_density;
uniform vec4 fog_color;

void main(void)
{
    vec4 color = ... lights ... textures ...

    float fog = exp(-fog_density * fog_density * gl_FragCoord.z * gl_FragCoord.z);
    fog = clamp(fog, 0.0, 1.0);

    // outputs final color
    FragColor = mix(vec3(fog_color), color, fog); //linear interpolation
}

// ====== or (for example) ======

uniform vec4 fog_color;

float log_depth(float depth, float steepness = 0.5f, float offset = 5.0f)
{
    float linear_depth = (2.0 * near * far) / (far + near - (depth * 2.0 - 1.0) * (far - near));
    return (1 / (1 + exp(-steepness * (linear_depth - offset))));
}

void main()
{
    vec4 color = ... lights ... textures ...

    // outputs final color
    float depth = log_depth(gl_FragCoord.z);
    FragColor = color * (1.0f - depth) + depth * fog_color; //linear interpolation, manual
}
```

Další vlastnosti mlhy

- Per-fragment operace v závěru pipeline
- Závisí jen na vzdálenosti od kamery
  - Z souřadnice, nezávisí na výšce nad terénem apod.
  - nelze vyrobit přízemní mlhu, cáry mlhy
    - jen přes analýzu scény (a FS)



Shrnutí real-time 3D grafiky :-)