Opakování látky pro předmět PV227

Milí studenti a studentky,

Vítáme vás v předmětu PV227, který je zaměřen zejména na programování shaderů v OpenGL. Jsme rádi, že jste si předmět zapsali a věříme, že vás bude bavit.

V tomto předmětu předpokládáme znalosti z jiných předmětů, a to zejména z předmětu PV112, ve kterém se vyučují základy OpenGL, a dalších předmětů, ve kterých se vyučují základy počítačové grafiky a základy programování. Opakování látky je třeba udělat na začátku každého předmětu. Vzhledem k tomu, že znalosti každého z vás jsou jiné a každý z vás vyžaduje jiné tempo opakování, rozhodli jsme se neudělat opakování na prvních hodinách, ale zvolili jsme tento dokument. Tady najdete ty nejdůležitější věci, které budete potřebovat znát, abyste pochopili látku předmětu, ale které byste měli znát z jiných předmětů.

Nevyžadujeme, abyste opakovanou látku znali do nejmenších podrobností. Není například nutné, abyste u každé funkce OpenGL věděli, jaké má parametry, jakého jsou typu apod. Stejně tak například nevyžadujeme, abyste znali vzorce pro vytváření rotačních či projekčních matic. Na druhou stranu ovšem chceme, abyste věděli, k čemu uvedené věci slouží, jak se s nimi pracuje, co je a co není možné udělat apod.

Věříme, že studenti, kteří znají OpenGL velmi dobře, nebudou mít s opakováním žádné potíže, v rychlosti proletí tento dokument a po několika minutách budou mít zopakováno. Pokud ale narazíte při opakování na věci, které vám nejsou jasné, použijte studijní materiály k předmětu PV112 (jsou přístupné v ISu ve studijních materiálech tohoto předmětu), diskuzní fórum předmětu, nebo napište e-mail nám vyučujícím.

Ještě dvě poznámky. Zaprvé, předměty PV112 a PV227 se v posledních letech předělávaly, což zapříčinilo, že máme v předmětu studenty různých úrovní znalostí. Zejména pak studenti, kteří se učili OpenGL od semestru jaro 2016 znají nové OpenGL a shadery do větších podrobností, než studenti, kteří se učili OpenGL do semestru jaro 2015 a znají starší OpenGL a shadery jen velmi zlehka.

Na cvičeních budeme používat nové OpenGL, ale rozhodli jsme se, že nebudeme na přednáškách věnovat čas tomu, v čem se staré a nové OpenGL liší. Tyto rozdíly uvádíme zde v tomto opakování. Text, který to popisuje, je zvýrazněn pruhem vlevo, jako tento odstavec. Doufáme, že přechod ze starého OpenGL

na nové nebude činit žádné problémy, kdybyste něčemu nerozuměli, opět se neváhejte zeptat.

A druhá poznámka na závěr. Toto dokument prosím *neslouží*, opakuji ***neslouží***, jako způsob, jakých se chceme my učitelé zbavit otravné práce a vynechat nudné opakování. Opravdu ne. Chceme tím hlavně ušetřit čas na přednáškám a věnovat ho něčemu zajímavějšímu. Také nechceme, aby ti, co se věnují OpenGL více a znají ho velmi dobře, byli potrestáni nudným opakováním na prvních cvičení (a byli tak odsouzeni k úmornému pročítání příspěvků na Facebooku a nuceni do otravného a vyčerpávajícího chatování).

Tímto dokumentem vám také nebereme možnost se na cokoliv zeptat. Opět, pokud zjistíte, že něco nechápete či nemůžete najít, neváhejte a napište nám učitelům e-mail nebo napište na diskuzní fórum předmětu, kde vám odpovíme my nebo vaši spolužáci.

Vaši učitelé

1 Opakování OpenGL

V této části si zopakujeme knihovnu OpenGL.

1.1 Pipeline

Měli byste znát pipeline OpenGL, tedy hlavně to, které operace je nutné provést pro zobrazení trojúhelníku a v jakém pořadí se provedou. Ty nejpodstatnější kroky jsou:

- Získání dat vrcholů z paměti
- Operace na vrcholech (např. transformace), to dělá vertex shader
- Rasterizace
- Operace na fragmentech (např. texturování), to dělá fragment shader
- Depth test, stencil test, blending apod.
- Uložení výsledku do framebufferu

1.2 Shadery

V novém OpenGL není možné nepoužívat shadery, je tedy nutné je znát velmi dobře. Ze shaderů byste měli znát:

- Co je to vertex a fragment shader a k čemu slouží.
- Programovací jazyk GLSL:
  - Syntax a sémantika podobná C++
  - Skalární datové typy (bool, int, float), vektorové typy (vec2, vec3, vec4), maticové typy (mat2, mat3, mat4)
  - Přetížené operátory pro práci s vektory a maticemi.
- Dostupné funkce (vyznačené jsou ty, které se opravdu hodí znát)
  - Základní funkce podobné těm v C++, např.: sin, cos, sqrt, apod. Ty se hodí znát, ale tak často je většinou potřebovat nebudeme.
  - Speciální funkce, které se hodí pro psaní shaderů, jako jsou **clamp**, **mix**, **min**, **max**, step, smoothstep. Ty lze často nahradit jednoduchým ifem, ale použití funkcí je čitelnější.
  - Funkce pro práci s vektory, a to **dot**, **cross**, length, **normalize**, distance, reflect.
  - Funkce pro porovnávání, jako jsou lessThan, greaterThanEqual apod.
  - Funkce **main**
- Speciální proměnné, a to zejména gl_Position

- Textury, uniformní proměnné, vstupní a výstupní proměnné. Těmto věcem věnujeme samostatné kapitoly tohoto opakování.

Dále pro ty, co přechází ze staršího OpenGL. V novém OpenGL je nutné používat vertex a fragment shader, není možné používat fixní pipeline. Z tohoto důvodu byl také odstraněn výpočet transformací a osvětlení, které se nyní provádí v shaderech, a také příslušné proměnné, které s tím souvisí.

V novém OpenGL tedy již nenajdete proměnné jako jsou matice gl_ModelViewProjectionMatrix, gl_FrontMaterial s parametry materiálu, či gl_LightSource s parametry světel. Pokud tyto proměnné potřebujeme, musíme si je definovat jako vlastní uniformní proměnné a nastavit jejich hodnoty. Tedy například místo:

```
void main()
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

je třeba napsat

```
in vec4 my_Vertex;
uniform mat4 my_ModelViewProjectionMatrix;
void main()
{
    gl_Position = my_ModelViewProjectionMatrix * my_Vertex;
}
```

To se týká zejména matic (projekční matice, pohledová matice, model matice, matice textur), parametrů materiálu, parametrů světel, vstupních proměnných vertex shaderu a výstupních proměnných fragment shaderu. Vše je nutné definovat vlastními proměnnými. Vstupních proměnných vertex shaderu a výstupních proměnných fragment shaderu se ještě věnují další odstavce.

1.3 Vykreslovací příkazy a primitiva

1.3.1 Primitiva

Nové OpenGL podporuje následující primitiva (důležitá jsou zvýrazněná, ty si pamatujte): **GL_POINTS**, **GL_LINES**, **GL_LINE_STRIP**, GL_LINE_LOOP, **GL_TRIANGLES**, **GL_TRIANGLE_STRIP**, GL_TRIANGLE_FAN.

OpenGL ještě podporuje další primitiva pro geometry shadery a teselační shadery, ty ale nebyly uvedeny v předmětu PV112 a budou uvedeny až zde v tomto předmětu.

Staré OpenGL podporovalo ještě čtyřúhelníky (GL_QUADS, GL_QUAD_STRIP)

a mnohoúhelníky (GL_POLYGON). Tyto primitiva již ovšem nejsou v novém OpenGL podporována, je třeba použít trojúhelníky.

1.3.2 Vykreslovací příkazy

Měli byste znát funkce glDrawArrays a glDrawElements a také jak se vykresluje geometrie s indexy (ta, která se vykresluje pomocí glDrawElements), tedy co jsou to ty indexy, na co odkazují apod.

Pro ty, co přechází ze starého OpenGL, už se nepoužívá glBegin a glEnd a s tím spojená specifikace dat vrcholů pomocí glVertex, glNormal a dalších. Pro všechno je nutné definovat pole s daty a použít funkce glDrawArrays a glDrawElements.

1.4 Data vrcholů

Měli byste znát práci s buffery v OpenGL, tedy funkce pro práci s buffer objektem (glGenBuffers, glBindBuffer, glDeleteBuffers), funkce pro alokaci paměti bufferu (glBufferData) a funkce pro aktualizaci dat bufferu (glBufferSubData, glMapBuffer, glUnmapBuffer). Měli byste znát, co je a k čemu slouží GL_ARRAY_BUFFER a GL_ELEMENT_ARRAY_BUFFER.

Měli byste znát, co jsou vstupní proměnné vertex shaderu, k čemu slouží a jak je specifikovat.

Měli byste znát, jak předat data z bufferu do vstupních proměnných shaderu. K tomu je třeba získat index vstupní proměnné shaderu (funkce glGetAttribLocation a glBindAttribLocation), povolit získávání dat pro tuto proměnnou (funkce glEnableVertexAttribArray) a nastavit, ve kterém bufferu se nachází data pro tuto proměnnou a jak jsou v něm uložená (funkce glVertexAttribPointer).

Měli byste znát, co je vertex array objekt (VAO), k čemu slouží a jak se používá.

Měli byste znát, jak specifikovat výstupní data vertex shaderu a vstupní data fragment shaderu. Dále byste měli vědět, jak probíhá interpolace dat mezi shadery, neboli jaké získáme data na vstupu fragment shaderu a jak jsme tyto data získali z dat na výstupu vertex shaderu.

Nakonec, měli byste co jsou to interface bloky a jak seskupit data mezi vertex shaderem a fragment shaderem do interface bloku.

Pro ty, co přechází ze starého OpenGL. Nové OpenGL udělalo pár úprav ve specifikaci dat vrcholů a ve způsobu, jakým se posílají data mezi shadery.

V shaderech se již nepoužívá označení attribute pro vstupní proměnnou vertex shaderu, ani varying pro výstupní proměnné vertex shaderu a vstupní proměnné fragment shaderu. Místo toho se používá jednodušší in a out pro označení vstupních a výstupních proměnných shaderů, ať už vertex shaderu nebo fragment shaderu. Také neexistují žádné předdefinované vstupní proměnné jako gl_Vertex, gl_Normal, gl_MultiTexCoord apod., všechny proměnné si musíme

definovat. Kód shaderů se tedy změní následovně, místo:

```
// Vertex shader
attribute float in_temperature;
varying float VS_temperature;
void main()
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    gl_TexCoord[0] = gl_MultiTexCoord0;
    VS_temperature = in_temperature;
}

// Fragment shader
varying float VS_temperature;
void main()
{
    // něco s VS_temperature
    // něco s gl_TexCoord[0]
}
```

je třeba napsat

```
// Vertex shader
in vec4 in_Vertex;
in vec2 in_TexCoord;
in float in_temperature;

out vec2 VS_TexCoord;
out float VS_temperature;

uniform mat4 my_ModelViewProjectionMatrix;

void main()
{
    gl_Position = my_ModelViewProjectionMatrix * in_Vertex;
    VS_TexCoord = in_TexCoord;
    VS_temperature = in_temperature;
}
// Fragment shader
in vec2 VS_TexCoord;
in float VS_temperature;
void main()
{
    // něco s VS_temperature
    // něco s VS_TexCoord
}
```

Všimněte si, že ne všechny proměnné byly odstraněny, gl_Position například zůstala. Těch proměnných, které zůstaly, je ovšem jen velmi málo, z těch, co jsme používaly, to je jen gl_Position, všechny ostatní je nutné (a možné) nahradit vlastními proměnnými.

Na příkladu je vidět, že ve fragment shaderu máme vstup nazván jako VS_TexCoord. To je nutné, proměnná se musí jmenovat stejně jako ve vertex shaderu, ale není to hezké vstup označovat jako VS, když VS značí vertex shader :-). Proto vznikly tzv. interface bloky. Ty slouží k seskupování proměnných, díky nim můžeme kód napsat takto:

```
// Vertex shader
in vec4 in_Vertex;
in vec2 in_TexCoord;
in float in_temperature;

// Seskupíme výstupní data vrchlu
out VertexData
{
    vec2 TexCoord;
    float temperature;
} outData;

uniform mat4 my_ModelViewProjectionMatrix;

void main()
{
    gl_Position = my_ModelViewProjectionMatrix * VS_Vertex;
    outData.TexCoord = in_TexCoord;
    outData.temperature = in_temperature;
}

// Fragment shader

// Seskupíme vstupní data vrchlu
in VertexData
{
    vec2 TexCoord;
    float temperature;
} inData;

void main()
{
    // něco s inData.temperature
    // něco s inData.TexCoord
}
```

Přesná pravidla pro práci s interface bloky lze najít ve studijních materiálech pro předmět PV112, jaro 2016, 8. přednáška. Jsou přístupná ve studijních materiálech předmětu PV227.

Další věcí, která se změnila, je to, že pro posílání dat je nutné použít buffery. Nelze si tedy vytvořit pole na CPU a poslat ukazatel na toto pole do funkcí glDrawElements či glVertexAttribPointer, jako to bylo možné ve starším OpenGL. Místo toho si musíme vytvořit buffer OpenGL (takové to GL_ARRAY_BUFFER a GL_ELEMENT_ARRAY_BUFFER), do něj nahrát data a ten použít při kreslení.

A poslední věc, do nového OpenGL byly přidány tzv. vertex array objekty, zkráceně VAO, které obsahují informace o tom, ze kterého bufferu se získávají data pro které vstupy a jak jsou tam ta data uložená. Jsou to objekty podobně jako objekty bufferů či textur, vytváří se pomocí funkce glGenVertexArrays (jako glGenBuffers) a ruší se pomocí glDeleteVertexArrays (jako glDeleteBuffers). Navazují se pomocí funkce glBindVertexArray.

Zjednodušeně řečeno, VAO si uchovávají informaci o volání funkcí glBindBuffer(GL_ARRAY_BUFFER /, GL_ELEMENT_ARRAY_BUFFER), glEnableVertexAttribArray a glVertexAttribPointer. Jejich použití lze ilustrovat na tomto příkladu. V prvním kódu, který nepoužívá VAO, je nutné vše nastavovat až při kreslení:

```
// Inicializace ve funkci init
// - načtení dat, vytvoření bufferů, shaderů apod.
// - nic víc

// Vykreslování ve funkci draw
glBindBuffer(GL_ARRAY_BUFFER, vbo_s_pozicemi_a_normalami);
glEnableVertexAttribArray(VS_Vertex_index);
glVertexAttribPointer(VS_Vertex_index, 3, GL_FLOAT, GL_FALSE,
    sizeof(float) * 6, 0);
glEnableVertexAttribArray(VS_Normal_index);
glVertexAttribPointer(VS_Normal_index, 3, GL_FLOAT, GL_FALSE,
    sizeof(float) * 6, (const char *)(sizeof(float)*3));
glBindBuffer(GL_ARRAY_BUFFER, vbo_s_texturami);
glEnableVertexAttribArray(VS_TexCoord_index);
glVertexAttribPointer(VS_TexCoord_index, 2, GL_FLOAT, GL_FALSE, 0, 0);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vbo_s_indexy);

glDrawArrays(...)

glDisableVertexAttribArray(VS_Vertex_index);
glDisableVertexAttribArray(VS_Normal_index);
glDisableVertexAttribArray(VS_TexCoord_index);
```

Při použití VAO na v následujícím kódě ale můžeme vše nastavit jen jednou

při načtení dat a poté jen napojit před kreslením:

```
// Inicializace ve funkci init
// - načtení dat, vytvoření bufferů, shaderů apod.

// Vytvoření VAO
glGenVertexArrays(1, &my_vao);
glBindVertexArray(my_vao);

// Navázání bufferů jako předtím před kreslením
glBindBuffer(GL_ARRAY_BUFFER, vbo_s_pozicemi_a_normalami);
glEnableVertexAttribArray(VS_Vertex_index);
glVertexAttribPointer(VS_Vertex_index, 3, GL_FLOAT, GL_FALSE,
    sizeof(float) * 6, 0);
glEnableVertexAttribArray(VS_Normal_index);
glVertexAttribPointer(VS_Normal_index, 3, GL_FLOAT, GL_FALSE,
    sizeof(float) * 6, (const char *)(sizeof(float)*3));
glBindBuffer(GL_ARRAY_BUFFER, vbo_s_texturami);
glEnableVertexAttribArray(VS_TexCoord_index);
glVertexAttribPointer(VS_TexCoord_index, 2, GL_FLOAT, GL_FALSE, 0, 0);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vbo_s_indexy);

// Vykreslování ve funkci draw - stačí navázat a nic víc.
glBindVertexArray(my_vao);
glDrawArrays(...)
```

1.5 Textury

Měli byste znát:

- Dimenze textur, a to hlavně GL_TEXTURE_1D, GL_TEXTURE_2D, a GL_TEXTURE_CUBE_MAP.
- Funkce pro práci s objektem textury, a to glGenTextures, glBindTexture, glDeleteTextures.
- Funkce pro alokaci dat textury a nahrání dat, hlavně glTexImage2D. Můžete si zopakovat i podobné funkce jako glTexSubImage2D, glCopyTexImage2D apod.
- Proces od načtení textury z disku po nahrání do paměti OpenGL.
- Parametr GL_TEXTURE_WRAP_S/T/R, a hodnoty alespoň GL_CLAMP_TO_EDGE a GL_REPEAT.
- Filtrování při zvětšování a zmenšování textury, tedy GL_TEXTURE_MIN/MAG_FILTER, GL_NEAREST, GL_LINEAR, mipmapping.

- Co je to texturovací souřadnice, texturovací jednotky, přístup ze shaderů, proměnné typu sampler1D/sampler2D/samplerCube, funkce texture a textureLod v GLSL.

Pro ty, co přechází ze starého OpenGL, změnilo se jen pár drobností. Textura již nemůže mít border, parametr border ve funkci glTexImage2D musí být nula (pro zvědavé, border může být, ale musí mít jen jednu barvu, kterou lze zadat funkcí glTexParameter).

Dále v GLSL místo funkcí texture1D, texture2D, textureCube apod. existuje jediná přetížená funkce texture.

A nakonec, vzhledem k tomu, že veškeré texturování a práce se získanými daty probíhá v shaderu, můžeme směle zapomenout na funkce glTexEnv či glTexGen.

1.6 Uniformní proměnné

Měli byste znát:

- Co jsou to uniformní proměnné, jak se liší od vstupní per-vertex proměnných vertex shaderu (těch označených jako in).
- Jak nahrát do uniformních proměnných data (získání indexu pomocí glGetUniformLocation, funkce glUniform*).

Dále byste měli znát uniform buffer objekty (UBO) a jak se s nimi pracuje. Nemusíte znát do podrobna pravidla layout (std140), je ale dobré vědět, k čemu slouží.

Pro ty, co přechází ze starého OpenGL, je nutné vysvětlit ty uniform buffer objekty. Jedná se o způsob, jakým uložit data uniformních proměnných do bufferu a poté použít tento buffer jako zdroj dat shaderu.

Prvním krokem je seskupení uniformních proměnných do interface bloku, tedy například:

```
layout (std140) uniform CameraData
{
    mat4 view_matrix;
    mat4 projection_matrix;
    vec3 eye_position;
};
```

Důležitý je název bloku (zde CameraData) a rozložení dat (zde layout (std140)). Rozložení dat udává, jak jsou data umístěna v paměti a jak jsou zarovnána. My budeme používat výhradně rozložení layout (std140), jehož popis lze najít v 8. přednášce PV112 (jaro 2016, dostupné ve studijních materiálech tohoto předmětu). Přesná pravidla tohoto layoutu není třeba vědět, je ale dobré vědět alespoň obrazně, jaká jsou.

Data pro unifomní proměnné umístěné do takového bloku jsou získávána

z OpenGL bufferu. Práce s tímto bufferem (vytvoření, rušení, alokace/nahrávání/aktualizace dat) je stejná jako například u vertex buffer objektů, jen se místo GL_ARRAY_BUFFER používá GL_UNIFORM_BUFFER.

Navázání dat mezi bufferem a blokem je komplikovanější a připomíná navázání textury na sampler. Připomeňme si, že texturu nenavazujeme přímo na sampler, ale na texturovací jednotku, na kterou navážeme i sampler. Podobně i zde buffer nenavazujeme přímo na blok, ale na tzv. binding point, na který navážeme i blok. V praxi to probíhá následovně:

```
// Inicializace ve funkci init

// Získáme index bloku CameraData, podobně jako získáváme index
//    proměnné sampleru
int CameraData_loc = glGetUniformBlockIndex(program, "CameraData");
// Řekneme, že blok CameraData má brát data z bufferu, který se
//    navazuje na binding point 1, podobně jako sampleru říkáme,
//    že má brát data z textury na texturovací jednotce 1.
glUniformBlockBinding(program, CameraData_loc, 1);

// Kreslení ve funkci draw

// Navázání bufferu CameraData_buffer na binding point 1, podobné
//    navázání textury na texturovací jednotku 1.
glBindBufferBase(GL_UNIFORM_BUFFER, 1, CameraData_buffer);
```

Důležité jsou tyto funkce:

- funkce glGetUniformBlockIndex, která vrací index bloku, je to jakoby ekvivalent funkce glGetUniformLocation
- funkce glUniformBlockBinding, která sváže daný blok s binding pointem, jakoby ekvivalent funkce glUniform1i, kterou navazujeme sampler z GLSL na texturovací jednotku
- funkce glBindBufferBase, která naváže buffer na příslušný binding point, jakoby ekvivalent funkcí glActiveTexture a glBindTexture, kterými navazujeme textury na texturovací jednotky.

Podrobnější popis lze najít v 8. přednášce PV112.

1.7 Výstup z fragment shaderu a framebuffer objekty

Měli byste znát:

- Co je to framebuffer objekt, funkce pro vytvoření/zrušení/navázání framebufferu (glGenFrameBuffers, glDeleteFramebuffers, glBindFramebuffer).
- Co jsou to color/depth/stencil attachmenty, funkce glFramebufferTexture2D pro připojení textur.

- Navázání výstupu ze shaderu na jednotlivé attachmenty (funkce glGetFragDataLocation, glBindFragDataLocation, glDrawBuffers).

Pro ty, co přechází ze starého OpenGL, je toto nová látka. Framebuffer objekty umožňují kreslení do textury místo do hlavního okna aplikace. Kreslením do textury můžeme například vykreslit scénu v mnohem vyšším rozlišení než je rozlišení okna a výsledek uložit jako screenshot. Můžeme také získanou texturu použít pro další kreslení a udělat tak třeba ve hře obrazovku z bezpečnostních kamer. Možností, jak toho využít, je obrovské množství.

Framebuffery samotné neosahují žádná data, jsou to jen kontejnery, na které napojíme textury, které připravujeme běžným způsobem. Framebuffery jsou objekty jako buffery nebo textury, a pro jejich vytvoření a zrušení používáme obdobné funkce, a to funkce glGenFramebuffers a glDeleteFramebuffers.

Framebuffery navazujeme funkcí glBindFramebuffer. Kromě indexu framebuffer objektu, který chceme navázat, musíme uvést i to, jestli framebuffer navazujeme s tím, že do něj budeme zapisovat (GL_DRAW_FRAMEBUFFER), nebo číst (GL_READ_FRAMEBUFFER), nebo obojí (GL_FRAMEBUFFER). Prakticky když chceme kreslit do textur, tak do framebuffer zapisujeme, když chceme data kopírovat do jiné textury či do paměti, tak z něj čteme, a pokud chceme kopírovat data z jednoho framebufferu do druhého (pomocí funkce glBlitFramebuffer), tak musíme mít navázané dva různé framebuffery, jeden jako READ a druhý jako DRAW.

Jak bylo řečeno, framebuffer neobsahuje žádné textury, textury se vytváří samostatně a na framebuffer se napojují. K napojení slouží funkce glFramebufferTexture2D. Můžu napojit až 8 textur a ukládat tak až 8 různých výstupů z fragment shaderu. Tomuto navázání se říká *attachment*, máme tedy až 8 barevných attachmentů označených jako GL_COLOR_ATTACHMENT0 až GL_COLOR_ATTACHMENT7. Navíc je možné napojit textury pro data hloubky a stencilu, to je nutné zejména, když chceme při kreslení do textur využívat test hloubky nebo stencil.

V praxi může vypadat kód vytvoření framebufferu včetně jeho textur následovně:

```
// Vytvoření tří textur, dvě s výstupními daty, a to barvou a
//    intenzitou světla, a třetí s daty pro testy hloubky a stencilu
GLuint my_tex[3];
glGenTextures(3, my_tex);

// Textura 2048x2048 pro výstup barvy, formát RGBA, 8 bitů na kanál
glBindTexture(GL_TEXTURE_2D, my_tex[0]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, 2048, 2048, 0, GL_RGBA,
    GL_UNSIGNED_BYTE, nullptr);

// Textura 2048x2048 pro výstup intenzity, jeden 32 bitový float
glBindTexture(GL_TEXTURE_2D, my_tex[1]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_R32F, 2048, 2048, 0, GL_RED,
```

```
    GL_FLOAT, nullptr);

// Textura 2048x2048 pro hloubku a stencil
glBindTexture(GL_TEXTURE_2D, my_tex[2]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH24_STENCIL8, 2048, 2048, 0,
    GL_DEPTH_STENCIL, GL_UNSIGNED_INT_24_8, nullptr);

// Vytvoření framebuffer objektu
GLuint my_fbo;
glGenFramebuffers(1, &my_fbo);

// Navázání textur na framebuffer
glBindFramebuffer(GL_FRAMEBUFFER, my_fbo);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
    GL_TEXTURE_2D, my_tex[0], 0);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1,
    GL_TEXTURE_2D, my_tex[1], 0);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT,
    GL_TEXTURE_2D, my_tex[2], 0);
```

Fragment shader může dávat na výstup více dat, jako například barvu světla a intenzitu v minulém příkladu. K propojení jednotlivých výstupů je třeba udělat následující věci.

Je nutné získat index výstupních proměnných fragment shaderu. Tento postup připomíná získávání indexů vstupních proměnných vertex shaderu a používají se k tomu funkce glGetFragDataLocation nebo glBindFragDataLocation. Jakmile máme tyto indexy, pomocí funkce glDrawBuffers řekneme, které color attachmenty máme přiřadit kterým indexům. Příkladem může být:

```
// Fragment shader
out vec4 final_color;
out float final_brightness;

// Inicializace ve funkci init: řekneme, že final_color dává data
//    na výstup 0 a final_brightness dává data na výstup 1
glBindFragDataLocation(my_program, 0, "final_color");
glBindFragDataLocation(my_program, 1, "final_brightness");

// Vykreslování ve funkci draw: řekneme, že první dva výstupy (tedy
//    výstupy 0 a 1) jdou do GL_COLOR_ATTACHMENT0 a GL_COLOR_ATTACHMENT1.
glBindFramebuffer(GL_FRAMEBUFFER, my_fbo);
GLenum bufs[] = {GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1};
glDrawBuffers(2, bufs);
// ... vykreslení scény
```

Když máme jen jeden výstup z fragment shaderu (všechny naše programy, co jsme doposud psali), nemusíme díky defaultním hodnotám tento výstup nikam navazovat. Je navázán na index 0, který je navázán na color attachment 0. To je u hlavního okna to, co se zobrazuje uživateli.

Na závěr, pokud nechceme kreslit do framebufferu, ale opět do hlavního okna, použijeme glBindFramebuffer(GL_FRAMEBUFFER, 0).

1.8 Instancování

Měli byste znát:

- Co je instancování a k čemu slouží.
- Jak vykreslit instancovaně (funkce glDrawArraysInstanced a glDrawElementsInstanced).
- Jak pracovat s instancováním v shaderech, proměnná gl_InstanceID, glVertexAttribDivisor.

Pro ty, co přechází ze starého OpenGL, je toto nová látka. Instancování je jakoby optimalizace, která nám umožňuje vykreslit naráz desítky či stovky objektů. Vzhledem k tomu, že se tyto objekty kreslí jedním příkazem, instancování má nižší overhead při zpracovávání kreslení.

Pokud chceme vykreslit jeden objekt několikrát instancovaně, použijeme místo funkcí glDrawArrays a glDrawElements funkce glDrawArraysInstanced a glDrawElementsInstanced, které mají navíc poslední parametr udávající počet instancí. Volání funkce

```
glDrawArraysInstanced(..., 100);
```

je tedy (téměř) totožné jako:

```
for (int i = 0; i < 100; i++)
    glDrawArrays(...);
```

Pokud používáme instancování, každý trojúhelník je zpracován několikrát (jednou pro každou instanci), a tak i každý vrchol je vertex shaderem zpracován několikrát (jednou pro každou instanci).

Pro instancování musíme mít připravené speciální shadery. Vzhledem k tomu, že kreslíme všechny objekty zaráz, nemůžeme měnit data jednotlivých objektů mezi kreslením (nemůžeme měnit transformační matice, barvy a další). Data všech objektů musíme mít dostupná v shaderech a musíme v shaderech vybrat ta data, která odpovídají objektu, který je právě shaderem zpracováván.

Možnosti, jak toho docílit, jsou dvě. První z nich je použít speciální proměnnou gl_InstanceID, která je dostupná ve vertex shaderu a která má hodnotu od 0 po počet instancí -1. Data všech instancí v těchto situacích bývají uložené jako pole uniformních proměnných, často v uniform buffer objektu. Příkladem takového vertex shaderu může být:

```
// Data všech objetků
uniform ObjectData
{
    mat4 model_matrices[400];
    vec4 model_colors[400];
};

// Výstupní barva tohoto objektu
out vec4 color;

void main()
{
    ...
    // Získávám data jednoho objektu: pozici a barvu
    mat4 model_matrix = model_matrices[gl_InstanceID];
    color = model_colors[gl_InstanceID];
    ...
}
```

Druhou možností je definovat data objektů jako data jednotlivých vrcholů (tedy použít kvalifikátor in) a použít navíc funkci glVertexAttribDivisor. Tato funkce říká, jestli je daná vstupní proměnná per-vertex (použiju hodnotu 0), nebo per-instance (použiju hodnotu 1). Obdobný kód vertex shaderu vypadá následovně:

```
// Model matice jednoho zpracovávaného objektu
in mat4 model_matrix;
// Barva jednoho zpracovávaného objektu
in vec4 model_color;

// Výstupní barva tohoto objektu
out vec4 color;

void main()
{
    ...
    // Použiju model_matrix k transformaci
    // Pošlu barvu do fragment shaderu
    color = model_color;
    ...
}
```

Použití funkce glVertexAttribDivisor k nastavení dat model matic vypadá následovně:

```
// Použiju buffer s daty matice
```

```
glBindBuffer(GL_ARRAY_BUFFER, model_matrices_vbo);

// mat4 je chápána jako 4 po sobě jdoucí vec4 proměnné
glEnableVertexAttribArray(model_matrix_loc);
glEnableVertexAttribArray(model_matrix_loc+1);
glEnableVertexAttribArray(model_matrix_loc+2);
glEnableVertexAttribArray(model_matrix_loc+3);

// Nastavím data matice jako každé jiné in proměnné
glVertexAttribPointer(model_matrix_loc, 4, GL_FLOAT, GL_FALSE,
    16*sizeof(float), nullptr);
glVertexAttribPointer(model_matrix_loc+1, 4, GL_FLOAT, GL_FALSE,
    16*sizeof(float), (void *)(4*sizeof(float)));
glVertexAttribPointer(model_matrix_loc+2, 4, GL_FLOAT, GL_FALSE,
    16*sizeof(float), (void *)(8*sizeof(float)));
glVertexAttribPointer(model_matrix_loc+3, 4, GL_FLOAT, GL_FALSE,
    16*sizeof(float), (void *)(12*sizeof(float)));

// Označím tato data jako data pro instancování
glVertexAttribDivisor(model_matrix_loc, 1);
glVertexAttribDivisor(model_matrix_loc+1, 1);
glVertexAttribDivisor(model_matrix_loc+2, 1);
glVertexAttribDivisor(model_matrix_loc+3, 1);
```

1.9 layout(location), layout(binding)

Pro zjednodušení práce se shadery budeme používat layout (location = x) pro vstupní proměnné vertex shaderu a výstupní proměnné fragment shaderu. Také budeme používat layout (binding = x) pro indexy UBO a stejný layout (binding = x) pro samplery.

Pro ty, co přechází ze starého OpenGL, je toto nová látka. Kvalifikátor layout neslouží pouze pro nastavení rozložení proměnných v paměti (layout (std140)), ale našel uplatnění i jinde. Pomocí layout (location = x) lze v shaderu nastavit index vstupních proměnných VS a výstupních proměnných FS, na který se mají tyto proměnné navázat. To je totéž, čeho jsme schopni docílit použitím funkcí glBindAttribLocation a glBindFragDataLocation.

Pomocí layout (binding = x) lze svázat uniform blok s binding pointem, tedy to, co dělá funkce glUniformBlockBinding. Pokud layout (binding = x) použijeme na sampler, nastavíme tak texturovací jednotku, ze které sampler bude brát data.

V našich cvičeních to budeme používat, protože to trochu zpřehledňuje kód (doufáme :-)). Použití je následující:

```
// Vertex shader
```

```
layout (location = 0) in vec4 position;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec2 tex_coord;

// Fragment shader
layout (location = 0) out vec4 final_color;
layout (location = 1) out float final_intensity;

// Uniform block
layout (std140, binding = 1) uniform CameraData
{
    mat4 view_matrix;
    mat4 projection_matrix;
    vec3 eye_position;
};

// Textura
layout (binding = 0) uniform sampler2D wood_tex;
```

Příprava shaderů se pak zjednodušší na prosté načtení a zkompilování kódu, protože díky layoutu máme zajištěno, že OpenGL bude hledat data tam, kde chceme.

1.10 Další

Zopakujte si následující:

Míchání barev

Měli byste vědět:

- Co je míchání barev.
- Co se spolu míchá (data ve framebufferu – destination, výstup z fragment shaderu – source).
- Že je nutné to povolit a možné zakázat pomocí glEnable/glDisable a GL_BLEND.
- Funkce glBlendFunc a parametry alespoň GL_ONE, GL_ZERO, GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA.
- Hodí se vědět i to, že existuje něco jako glBlendEquation, a jaké ruzné jsou ty rovnice.

Test hloubky

Měli byste vědět:

- Základní princip, jak to funguje.
- Co je to hloubkový buffer a k čemu je potřeba.
- Že je nutné to povolit a možné zakázat pomocí glEnable/glDisable a GL_DEPTH_TEST.
- Že je nutné ho vymazat, funkce glClearDepth.
- Hodí se vědět i to, že porovnávací funkci lze nastavit pomocí glDepthFunc.

Stencil test

Stencil test pravděpodobně nepoužijeme, ale i tak si můžete zopakovat základní princip, jak to funguje, co je to stencil buffer a k čemu je potřeba.

Ostatní

- Zopakujte si, k čemu slouží glViewport.
- Zopakujte si mlhu.

Závěrem, ve studijních materiálech se nachází projekt Repetition, ve kterém můžete najít, jak se prakticky s uvedenými věcmi pracuje. Více o tomto projektu dále v tomto dokumentu.

2 Opakování teorie

Vzhledem k tomu, že OpenGL je dosti low-level záležitost, je nutné dobře znát i teorii, která se za tím vším skrývá. Tomuto opakování by měli věnovat více času zejména ti, co přechází ze starého OpenGL, protože staré OpenGL řešilo mnohé "za nás", a proto teorie nebyla tolik potřeba.

2.1 Transformace

Měli byste znát:

- Co jsou to homogenní souřadnice, jak získám z 4D homogenní souřadnice běžnou 3D souřadnici (vydělím w), jaká je hodnota w pro body (w=1) a jaká je hodnota pro normály, směry a body v nekonečnu (w=0).
- Matice. Algebra: násobení matice vektorem, násobení dvou matic, základní vztahy:

$$
\begin{aligned}
&A \cdot B \neq B \cdot A \\
&A \cdot B \cdot C = A \cdot (B \cdot C) = (A \cdot B) \cdot C \\
&(A \cdot B)^T = B^T \cdot A^T \\
&(A \cdot B)^{-1} = B^{-1} \cdot A^{-1}
\end{aligned}
$$

- Konkrétní matice
  - translace (to byste mohli vědět, jak vypadá),
  - scale (to taky zvládnete),
  - rotace (alespoň si všimněte, že to je jen horní levá 3x3 část, jinak se ji neučte),
  - lookAt (OK, ta se neprobírala, ale věžte, že existuje, zájemci si můžou najít na internetu, neučte se ji),
  - matice pro ortogonální a perspektivní projekci (ty se taky neučte, ale kouknout se na ně můžete).
- Skládání transformací, pořadí transformací, lokální a globální přístup skládání transformací.

2.2 Osvětlení

Měli byste znát:

- Blinn-Phongův osvětlovací model (Prosíím, fakt se ho naučte, je to děs, když ho studenti neumí :-)), a to zejména:
  - Co je to normála. Proč je dobré/nutné ji normalizovat. Proč ji normalizovat, ve kterém shaderu, kdy a proč.

  - Co je to ambietní, difuzní a spekulární složka, kde se která složka projevuje. Co musím udělat, když je chci dát dohromady (odpověď: sečíst).
  - Jaký je faktor násobení u ambientní složky (odpověď: žádný se nepoužívá).
  - Jaký je faktor násobení u difuzní složky (odpověď: max(dot(N, L), 0)).
  - Jaký je faktor násobení u spekulární složky, co je to half-vektor a co je to shininess (odpověď: H = (L+Eye).normalize(), pow(max(dot(N, H), 0), shininess)).
  - Co je to materiál, proč se používá separátně barva materiálu a světla, jak dát dohromady barvu materiálu a světla (odpověď: snásobit).
  - Jak dát dohromady příspěvky několika světel (odpověď: sečíst).

- Typy světel, tedy alespoň jednoduché bodové, směrové a kuželové světlo. Zopakujte si i matematiku u těch světel, sice ji stejně budeme kopírovat z jednoho shaderu do druhého, ale ať víte, o čem to je.
- Jednoduchý model útlumu světla, ten co používá konstantní, lineární a kvadratický faktor (ten, který se zmínil v PV112).
- Transformace normály, jakou matici použít (odpověď: inverzi transpozice horní 3x3 části), kdo chce, zopakujte si i proč se má použít zrovna tato matice.
- Transformace světel, popřemýšlejte, co všechno bude třeba transformovat, pokud chcete pohnout bodovým, směrovým či kuželovým světlem.

2.3 Souřadnicové systémy

Tato látka se se starým OpenGL neprobírala prakticky vůbec a s tím novým se jí také moc času nevěnovalo. Nicméně je to velmi důležité. Nemělo by to být nic nového, ale zopakovat se musí.

Měli byste znát velmi dobře, co jsou to souřadnicové systémy, co je to souřadnicový systém objektu (object space), světa (world space) a kamery (camera space), měli byste vědět, jak převádět souřadnice z jednoho systému do druhého. Pokud něco z toho nevíte, pokračujte ve čtení následující zelené části.

Object space, world space, camera space

Zjednodušeně, souřadnicovým systémem zde rozumíme (trojrozměrný) prostor, ve kterém máme určený počátek (souřadnici [0,0,0]) a směry tří hlavních os. Souřadnicových systémů můžeme mít prakticky nekonečné množství, my se ale budeme zabývat třemi nejdůležitějšími.

Prvním z nich je souřadnicový systém objektu (často označován jako object

space nebo jako model space). Je to systém, ve kterém jsou definovány pozice jednotlivých vrcholů, normály v těchto bodech, je to systém, ve kterém je tento objekt uložen. Pokud bych měl takto uloženou jednotkovou kouli, její vrcholy budou mít souřadnice od [-1,-1,-1] do [1,1,1].

Druhým důležitým souřadnicovým systémem je souřadnicový systém světa (world space). Je to systém, ve kterém definujeme scénu, ve kterém máme umístěné všechny objekty apod. Pokud bychom například vytvářeli scénu s pokojem, můžeme počátek [0,0,0] umístit do jednoho rohu místnosti, osy x a y směřovat do dvou směrů této místnosti a osu z směřovat nahoru.

Pokud umístíme naši jednotkovou kouli doprostřed této místnosti, souřadnice jejich vrcholů už nebudou mít hodnoty od [-1,-1,-1] do [1,1,1], protože tyto souřadnice se nachází někde okolo rohu místnosti, kam jsme umístili bod [0,0,0]. Nicméně stále můžeme použít kouli, kterou jsme načetli ze souboru a která má vrcholy od [-1,-1,-1] do [1,1,1]. Stačí ji jen transformovat modelovou maticí. Toto je přesně to, k čemu slouží modelová matice.

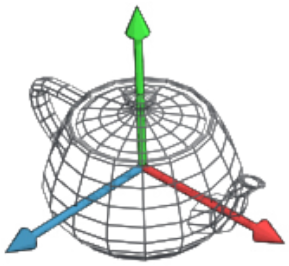
Z matematiky byste měli vědět, že je možné provést i opačnou transformaci, tedy převést objekt ze souřadnicového systému světa do souřadnicového systému objektu. Je to velmi jednoduché, používá se k tomu inverzní matice. Pokud bychom tedy měli nějaký jiný objekt v místnosti a chtěli pozici jeho vrcholů vyjádřit v souřadnicovém systému té koule, použijeme inverzi modelové matice té koule k transformaci objektu, který má souřadnice vyjádřen v souřadnicovém systému světa.

Nyní co je souřadnicový systém kamery (camera space nebo taky view space). Pokud umístíme do našeho světa kameru tak, abychom se dívali na kouli, tak vůči té kameře budou mít vrcholy té koule x-ovou a y-ovou souřadnici v intervalu od [-1,-1] do [1,1], protože ta kamera je zaměřená na tu kouli. Souřadnice z ale bude záporná, protože je ta koule před námi, a bude odpovídat zhruba vzdálenosti koule od kamery.

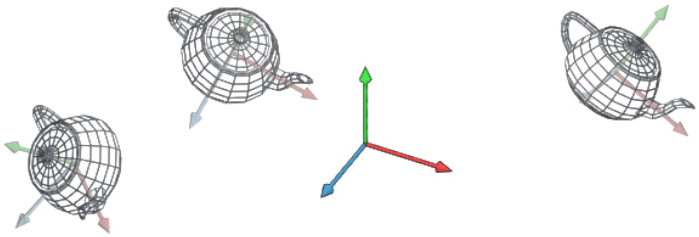
V souřadnicovém systému kamery je počáteční bod [0,0,0] umístěn do místa, ze kterého se díváme, osa x směřuje doprava (doprava od toho, na co se díváme, ne doprava ve světě, kamera může být i vzhůru nohama), osa y směřuje nahoru a osa z směřuje k nám, díváme se tedy ve směru $-z$, používáme right-handed souřadnicový systém. Pro transformaci ze souřadnicového systému světa do souřadnicového systému kamery se používá view matice.

Existují i další souřadnicové systémy, s některými z nich se v předmětu setkáme. Tyto tři popsané jsou ale patrně nejdůležitější.

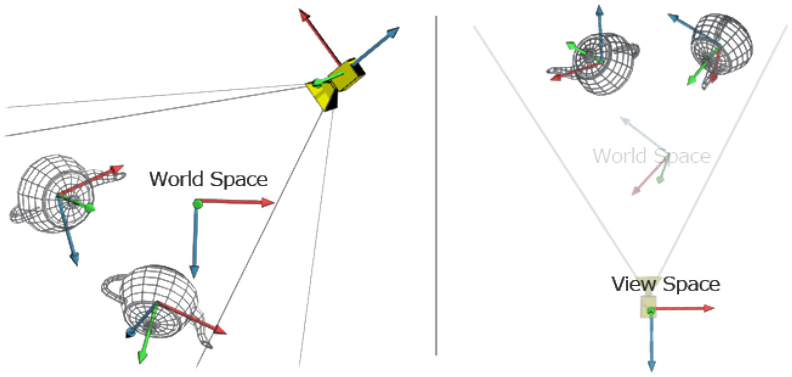
Na stránce `http://www.codinglabs.net/article_world_view_projection_matrix.aspx` lze najít hezký popis těchto prostorů včetně několika ilustrací. Uvádíme zde jen ty nejdůležitější.



Obrázek 1: Souřadnicový prostor objektu (object space). Na obrázku vidíme konvičku a její souřadnicový systém. V tomto systému jsou definovány pozice jednotlivých vrcholů, normály atd.



Obrázek 2: Souřadnicový prostor světa (world space). Na obrázku vidíme tři konvičky, které jsou reprezentovány stejným modelem, souřadnice jejich bodů v souřadnicovém systému objektu (slabé trojice šipek) jsou stejné. Vidíme, že každá konvička má jinou modelovací matici, každá tato matice posune objekt na jiné místo. V souřadicovém systému světa tak mají stejné vrcholy různých konviček jiné souřadnice (nic překvapivého :-)).



Obrázek 3: Souřadnicový prostor světa (world space) a kamery (zde jako view space). Na levé části obrázku vidíme scénu se dvěma konvičkama a kamerou v souřadnicovém systému světa. Vidíme zde, že souřadice Z (ta modrá), je u té spodní konvičky kladná a u té konvičky vlevo kladná i záporná. Na obrázku vpravo je totéž, ale v souřadnicovém systému kamery (zde view space). Souřadnice Z všech vrcholů všech konviček je v tomto systému záporná.

K čemu je to dobré

Souřadnicové systémy musíme mít na paměti, když se snažíme něco spočítat. Typickým příkladem je osvětlení.

Při výpočtu osvětlení je nutné spočítat skalární součin normály a směru ke světlu, a získat tak úhel mezi těmito sdvěma vektory. Tento úhel ale musíme počítat ve správném souřadnicovém systému, nemůžeme vzít normálu v souřadnicovém systému objektu a směr světla v souřadnicovém systému světla. Oba směry je nutné transformovat správnou maticí do správného souřadnicového systému.

Pokud si zvolíme souřadnicový systém světa, musíme normálu transformovat modelovou maticí objektu (přesněji inverzí transpozice modelové matice). Musíme transformovat i směr světla, protože i světlo může být umístěno někde ve světě (například jako lampička na stole). V PV227 budeme nejčastěji používat pro výpočet osvětlení právě souřadnicový systém světa.

Můžeme si zvolit i jiný souřadnicový systém. Staré OpenGL například počítá osvětlení v souřadnicovém systému kamery, všechny normály a směry tedy násobí ještě view maticí. Ani volba tohoto souřadnicového systému nemusí být špatná, kupříkladu pozice pozorovatele v tomto systému je rovna [0,0,0].

Podobné příklady nalezneme i jinde. Prostě a jednoduše, vždy mějte na paměti, ve kterém souřadnicovém systému máte data a jestli není třeba nejdřív něco transformovat.

3 Framework, ve kterém budeme pracovat

V této části si popíšeme framework, se kterým budeme na cvičení pracovat. Nejprve si ale popíšeme použité knihovny. Budeme používat stejné knihovny, jako na cvičení C++ předmětu PV112 v semestru jaro 2016. Ti, co to znají, můžou přeskočit následující žlutý text.

Knihovna FreeGLUT

Knihovna GLUT, či přesněji její novější udržovaná verze FreeGLUT, je knihovna, která se stará o cross-platformní práci s okny, příjem zpráv z klávesnice, myši apod. Je to obdoba Javovského JOGL.

Knihovna GLEW

Knihovna GLEW slouží k přípravě funkcí OpenGL verze 1.2 a výš. V C++ jsou tyto funkce reprezentovány pomocí ukazatelů na funkce, a tyto ukazatele je nutné správně inicializovat, což dělá právě knihovna GLEW. Díky ní tak máme přístup ke všem dostupných verzím OpenGL a všem rozšířením, které naše grafická karta podporuje.

Knihovna GLM

V novém OpenGL si musíme všechny matice počítat a skládat sami. K tomu budeme používat knihovnu GLM.

Knihovna GLM má výhodu v tom, že se snaží vypadat stejně jako GLSL. Její třídy a funkce jsou umístěny v namespace glm, ale jinak jsou prakticky stejné jako ty, které jsou v GLSL. Najdeme zde třídy jako vec2, vec3, vec4, mat3, mat4 apod., funkce inverse, transpose, normalize, dot, cross, přetížené operátory apod. Knihovna navíc obsahuje funkce pro vytváření základních matic, jako jsou funkce translate, rotate, scale, perspective a lookAt.

Ohledně pořadí transformací, na následujících řádcích je několik transformací složených stejným způsobem, ale každá vyjádřená v jiném prostředí. Doufám, že si každý naleznete způsob, který je vám známý, a podle něj si odvodíte, co používat v GLM, jak a v jakém pořadí. Na naších cvičeních budeme používat poslední dva způsoby (podle potřeby).

```
// Staré OpenGL
glLoadIdentity();
glTranslate(T1);
glRotate(R1);
glTranslate(T2);
glRotate(R2);
.. draw
```

```
// Nové OpenGL v Javě, PV112, jaro 2016
Mat4 matrix = Mat4.MAT4_IDENTITY;
matrix = matrix.translate(T1);
matrix = matrix.rotate(R1);
matrix = matrix.translate(T2);
matrix = matrix.rotate(R2);
.. draw

// GLM, matice se přinásobují
glm::mat4 matrix = glm::mat4(1.0f);
matrix = glm::translate(matrix, T1);
matrix = glm::rotate(matrix, R1);
matrix = glm::translate(matrix, T2);
matrix = glm::rotate(matrix, R2);
.. draw

// GLM, matice generují samostatně
glm::mat4 t1 = glm::translate(glm::mat4(1.0f), T1);
glm::mat4 r1 = glm::rotate(glm::mat4(1.0f), R1);
glm::mat4 t2 = glm::translate(glm::mat4(1.0f), T2);
glm::mat4 r2 = glm::rotate(glm::mat4(1.0f), R2);
glm::mat4 matrix = t1 * r1 * t2 * r2;
.. draw
```

Vrchol se transformuje stejně jako v GLSL, můžeme tedy i v C++ použít

```
glm::vec4 transformed = matrix * untransformed;
```

To můžeme použít například pro transformaci pozice světla do správného souřadnicového systému.

Ohledně pořadí pro snásobení projekční, pohledové a modelové matice, pořadí je opět stejné jaké bychom použili v GLSL, tedy

```
gl_ModelViewProjection = PVM = projection * view * model;
```

Na závěr, pro předání dat do OpenGL potřebujeme ukazatel na interní data GLM objektu. K tomu slouží GLM funkce value_ptr, tedy například:

```
glm::mat4 view_matrix = ...;
glUniformMatrix4fv(view_matrix_loc, 1, GL_FALSE, glm::value_ptr(view_matrix));
```

Knihovna DevIL

Knihovna DevIL je jednoduchá knihovna, která slouží pro načítání obrázků. Tuto knihovnu ale prakticky znát nemusíte, kód, který načte obrázek a načte ho do OpenGL bude připraven.

Visual Studio

Budeme používat Visual Studio 2013 (nebo 2015), připravené projekty by měly běžet i na Visual Studiu 2012. Pro ty, co VS neznají nebo s tím nemají zkušenosti:

- Chci všechno skompilovat, ale nechci to spustit: Build → Build Solution, nebo F7
- Chci všechno skompilovat a spustit: Debug → Start Debugging, nebo F5, nebo ten zelený trojúhelník ve tvaru Play s nápisem Local Windows Debugger.
- Zmizel mi seznam souborů se kterými pracuji: View → Solution Explorer
- Napsal jsem prvních pár písmen a chci doplnit zbytek: Ctrl + mezera
- Napsal jsem tečku nebo šipku a chci seznam metod a atributů: Ctrl + mezera
- Napsal jsem jméno funkce a otevírací závorku, chci seznam parametrů té funkce: Shift + Ctrl + mezera
- Chci umístit breakpoint: kliknout myší nalevo před začátek řádku, nebo F9
- Chci se kouknout na hodnotu proměnné: najedu myší nad proměnnou
- Chci krokovat aplikaci: Step Over (F10), Step Into (F11), Step Out (Shift + F11)
- Chci zase aplikaci rozjet a nekrokovat: Debug → Continue (F5)
- Chci zobrazit čísla řádků: Tools → Options → Text Editor → C/C++ → General → Display → Line Numbers
- Chci zvýrazňovat syntax v GLSL: Tools → Options → Text Editor → File Extension, tady zadat Extension glsl, Editor Microsoft Visual C++ a Add.

  Toto je jen základní zvýrazňování, Visual Studio bude chápat soubory .glsl jako C++, takže bude odsazovat apod., ale klíčová slova jako třeba uniform stejně znát nebude a bude je podtrhávat. I přesto to budeme používat, ale kdo chce, může zkusit nějaký externí editor (a pokud bude fakt hezký, může nám dát vědět a můžeme ho používat i my).

Knihovna AntTweakBar

Knihovna AntTweakBar je jednoduchá knihovna pro vytvoření GUI v OpenGL. Její použití je velmi jednoduché, doufáme, že během prvních cvičení ji pochopíte.

Framework PV227

Po zkušenostech s jinými frameworky jsme se rozhodli vytvořit vlastní framework přesně pro potřeby předmětu PV227. Nechceme ale, aby tento framework byl pro vás něčím záhadným či magickým. Proto by neměl obsahovat jiné věci než ty, které jsou vám známy z PV112.

Nechceme věnovat příliš času procházením kódu, na druhou stranu bychom ovšem chtěli, abyste tomuto frameworku rozuměli a uměli s ním pracovat. Proto bude část první přednášky věnována představení tohoto framework s tím, že procházení podrobností necháme vám na samostudium.

Zde vám dáme osnovu k tomuto frameworku:

- PV227_Basics.h, PV227_Basics.cpp obsahuje základní věci:
  - Konstanty, ty si projděte
  - Funkce pro práci s aplikací, těm moc času nevěnujte
  - Funkce pro práci se shadery, ty si projděte dobře, ty budeme používat. Neobsahují nic zvláštního, vše byste měli znát.
  - Funkce pro práci s geometrií, ty si projděte dobře, ty budeme používat. Třídu Geometry byste měli pochopit, funkce Create* taky (neděste se, jsou prakticky totožné). Načítání geometrií z externího OBJ souboru přeskočte, to je nudné a s největší pravděpodobností se předělá.
  - Funkce pro práci s kamerou, ty si projděte. Nemusíte tomu věnovat příliš času, je to jednoduchá implementace kamery.
  - Funkce pro načítání textur, ty si projděte jen rychle. Většina z toho je práce s knihovnou DevIL, to můžete prakticky ignorovat, ale zbytek, tedy práci s OpenGL, si můžete projít a zopakovat.
- PV227_UBO.h, PV227_UBO.cpp obsahuje základní uniform bloky, které budeme používat. Prakticky všechny třídy dělají totéž, pracují s uniform buffer objekty a kopírují do něj data. U všech těchto bloků naleznete i kód GLSL, který rozložením proměnných odpovídá tomu, jak jsou rozložené proměnné v C++ kódu. Máme zde připravené tyto uniform bloky:
  - CameraData_UBO: obsahuje projekční matici, pohledovou matici a pozici pozorovatele. Navíc obsahuje i inverze těchto matic a transpozici inverze horní 3x3 části view matice, sice se tyto matice příliš nepoužívají, ale pokud bychom je potřebovali, máme je připravené.
  - ModelData_UBO: obsahuje model matici. Navíc opět obsahuje i její inverzi transpozici inverze její 3x3 části, které se používaji k transformaci normál.
  - PhongLight: Toto není UBO, ale struktura, umožňuje nám definovat data základních světel (bodového, směrového a kuželového).
  - PhongLightsData_UBO: obsahuje data osvětlení, tedy globální ambientní světlo (vždy přítomné ambientní světlo), počet aktivních světel

ve scéně a jejich data. Můžeme zde nalést i připravený GLSL kód funkce EvaluatePhongLight pro vyhodnocení osvětlení jednoho světla.

- MaterialData a MaterialData_UBO: Struktura se základními daty materiálu a UBO pro tato data.

Jak se tento framework používá se nejlépe ukáže na příkladu. Pro demonstraci jsme vytvořili projekt *Repetition*, na kterém kromě tohoto frameworku ukážeme i všechny důležité věci z OpenGL, které byste měli znát. Kód tohoto příkladu se nachází v souborech repetition_main.h, repetition_main.cpp, a samozřejmě v shaderech umístěných ve složce Shaders.

- Soubory repetition_main.h a repetition_main.cpp jsou rozděleny na následující části. Budeme se snažit takto dělit všechna cvičení, bude-li to možné.
  - Scéna: obsahuje vše, s čím budeme pracovat a co budeme měnit. Obsahuje zejména všechny shadery, všechny geometrie, všechny textury, všechna data objektů, všechny framebuffer objekty atd. Tento kód si důkladně projděte, tento kód je velmi důležitý, protože ukazuje práci s naším frameworkem.
  - GUI: obsahuje GUI programu. Zde budeme zasahovat jen do části, která definuje GUI, v situacích, kdy budeme chtít toto GUI změnit.
  - Aplikace: Tato část obsahuje reakci na GLUT a inicializaci programu. Tento kód si můžete projít, ale moc času tomu věnovat nemusíte, je to jen nudný nutný kód.
- A samozřejmě budeme pracovat se shadery, které budou umístěné ve složce Shaders. Tyto shadery si projděte, se shadery budeme pracovat v tomto předmětu asi nejvíc.