

STŘEDOŠKOLSKÁ ODBORNÁ ČINNOST

Obor: 18. Informatika

# Grafický engine v C++ a OpenGL

Vítězslav Lužný

Praha 2020

# STŘEDOŠKOLSKÁ ODBORNÁ ČINNOST

## GRAFICKÝ ENGINE V C++ A OPENGL

GRAPHICS ENGINE IN C++ AND OPENGL

AUTOR      Vítězslav Lužný

ŠKOLA      Gymnázium Christiana Dopplera

KRAJ      Praha

ŠKOLITEL      Pavel Obdržálek

OBOR      18. Informatika

Praha 2020

## **Prohlášení**

Prohlašuji, že svou práci na téma *Grafický engine v C++ a OpenGL* jsem vypracoval samostatně a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Dále prohlašuji, že tištěná i elektronická verze práce SOČ jsou shodné a nemám závažný důvod proti zpřístupňování této práce v souladu se zákonem č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a změně některých zákonů (autorský zákon) v platném změní.

V Praze dne: \_\_\_\_\_

---

Vítězslav Lužný

## **Poděkování**

Rád bych poděkoval autorům všech open source knihoven, které jsem použil. Bez nich by tento projekt nebyl možný nebo by byl velice omezený.

## Anotace

Cílem této práce je vytvořit vývojové prostředí a knihovny pro multiplatformní vývoj 2D a 3D grafických aplikací pro Linux, Windows a Android včetně aplikací mobilní virtuální reality pro Android.

## Klíčová slova

C++, OpenGL, 2D Grafika, 3D Grafika, Virtuální realita, Multiplatformní vývoj, Normálové mapovaní, Materiálové mapování, Parallaxové mapování, Kosterní animace, Úroveň detailu, Deferred shading, Blinn-Phongův model, Screen door průhlednost, Alpha průhlednost

## Annotation

The goal of this work is to create integrated development environment and libraries for multiplatform development of 2D and 3D graphics apps for Linux, Windows and Android including virtual reality apps for Android.

## Keywords

C++, OpenGL, 2D Graphics, 3D Graphics, Virtual Reality, Multiplatform Development, Material Mapping, Normal Mapping, Parallax Mapping, Skeletal Animations, Level of Detail, Deferred Shading, Blinn-Phong Model, Screen Door Transparency, Alpha Blending

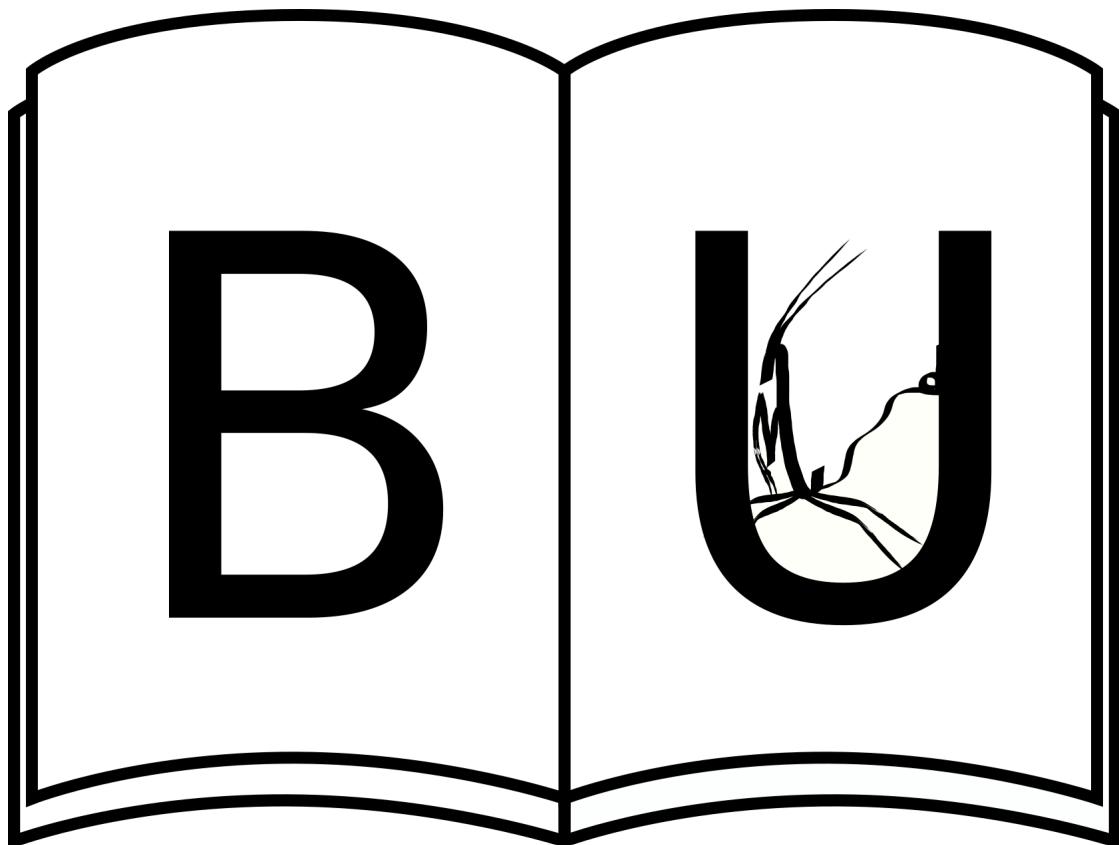
# Obsah

<b>1</b>	<b>Úvod</b>	<b>8</b>
<b>2</b>	<b>Použití</b>	<b>9</b>
2.1	Instalace . . . . .	9
2.1.1	Ubuntu . . . . .	9
2.1.2	Windows . . . . .	9
2.2	Vytvoření projektu ve vývojovém prostředí IDEal . . . . .	9
<b>3</b>	<b>Podrobný popis a teorie</b>	<b>13</b>
3.1	IDEal . . . . .	13
3.1.1	Kompilace pro Linux (Ubuntu) . . . . .	13
3.1.2	Kompilace pro Windows . . . . .	14
3.1.3	Kompilace pro Android . . . . .	14
3.1.3.1	Základní postup komplikace pro Android . . . . .	15
3.2	Bibliotekum_Ultimatum . . . . .	17
3.2.1	C++ . . . . .	17
3.2.2	Základní nástroje . . . . .	17
3.2.2.1	Dynamické pole - ulm::Array . . . . .	17
3.2.2.2	String - ulm::String . . . . .	17
3.2.2.3	Bytové dynamické pole - ulm::ByteArray . . . . .	17
3.2.3	Využití open source knihoven . . . . .	17
3.2.3.1	Vytvoření okna - SDL . . . . .	17
3.2.3.2	Vytvoření okna (Activity) pro virtuální realitu na Android - Google VR SDK . . . . .	18
3.2.3.2.1	Princip virtuální reality . . . . .	18

3.2.3.2.2	Implementace . . . . .	18
3.2.3.3	GUI - Nuklear . . . . .	19
3.2.3.4	Manipulace s obrázky - STB . . . . .	20
3.2.3.5	Matematika - GLM . . . . .	20
3.2.4	Grafika - OpenGL . . . . .	21
3.2.4.1	OpenGL rendering pipeline . . . . .	21
3.2.4.1.1	Vertex Specification . . . . .	21
3.2.4.1.2	Vertex Shader . . . . .	22
3.2.4.1.3	Geometry Shader . . . . .	23
3.2.4.1.4	Vertex post-processing . . . . .	23
3.2.4.1.5	Primitive Assembly . . . . .	24
3.2.4.1.6	Rasterization . . . . .	24
3.2.4.1.7	Fragment Shader . . . . .	25
3.2.4.2	Zatím nezmíněné OpenGL objekty . . . . .	27
3.2.4.2.1	Uniform Buffer Object . . . . .	27
3.2.4.2.2	Textura . . . . .	27
3.2.4.3	2D grafika . . . . .	29
3.2.4.3.1	Polygon - <i>ulm::Sprite</i> . . . . .	29
3.2.4.3.2	ShaderToy - <i>ulm::ShaderToy</i> . . . . .	32
3.2.4.4	3D grafika . . . . .	36
3.2.4.4.1	Mesh . . . . .	36
3.2.4.4.2	Lineární algebra . . . . .	38
3.2.4.4.3	Transformace vektoru pomocí matice . . .	38
3.2.4.4.4	Blinn-Phongův model světla . . . . .	55
3.2.4.4.5	Deffered Shading . . . . .	57
3.2.4.4.6	Typický postup 3D vykreslování v <i>BU</i> . .	65
3.2.4.4.7	Vlajky vykreslovače <i>ulm::Renderer3D</i> .	68
3.2.4.4.8	Face culling . . . . .	79
3.2.4.4.9	Úroveň detailu (LOD) . . . . .	80
4 Závěr		82

# 1. Úvod

Účelem této práce je vytvořit vývojové prostředí (*IDEal*) a knihovny (*Bibliotekum\_Ultimatum*) pro multiplatformní vývoj 2D a 3D grafických aplikací.



Obrázek 1.1: Logo projektu

# 2. Použití

## 2.1 Instalace

### 2.1.1 Ubuntu

Vývojové prostředí *IDeal* včetně knihovny *Bibliotekum\_Ultimatum* si můžete stáhnout z následujícího odkazu na *github*: [https://github.com/luzny274/IDEal\\_linux](https://github.com/luzny274/IDEal_linux).

Nejnovější verzi knihovny *Bibliotekum\_Ultimatum* si můžete stáhnout vždy z: [https://github.com/luzny274/Bibliotekum\\_Ultimatum](https://github.com/luzny274/Bibliotekum_Ultimatum).

### 2.1.2 Windows

Vývojové prostředí *IDeal* včetně knihovny *Bibliotekum\_Ultimatum* si můžete stáhnout z následujícího odkazu na *github*: [https://github.com/luzny274/IDEal\\_win](https://github.com/luzny274/IDEal_win).

*IDeal* pro Windows je omezenější než ten pro Linux. Chybí zde například skripty pro komplikaci na Android a také si nemůžete vybrat komplikátor. Je tedy potřeba si také **nainstalovat Clang a MSVC**, které jsou jím používány.

## 2.2 Vytvoření projektu ve vývojovém prostředí *IDeal*

Projekt se vytváří pomocí skriptů *create\_project.sh* na Linux a *create\_project.bat* na Windows, které se nachází ve složce *Workspace* spolu se složkami projektů.

Při spuštění tohoto skriptu se vygeneruje projekt se specifikovaným názvem.

Ve složce *main* projektu se nachází dva hlavní soubory, *main.cpp* a *header.cpp*. Také se zde nachází složka se jménem *jmenoProjektu\_resources*, zde by se měli uchovávat všechny soubory, které mají být k dispozici při běhu aplikace. Cestu relativní vůči této složce můžeme získat statickou metodou *getResourcePath* třídy *ulm::Properties*.

*header.cpp* - Zde se specifikují použité zdrojové soubory knihovny *Bibliotekum\_Ultimatum*:

```
#ifndef BU_PROGRAM_HEADER
#define BU_PROGRAM_HEADER

#ifndef BU_APP_NAME
    #define BU_APP_NAME "Something went wrong"
#endif

#include "Bibliotekum_Ultimatum/Base/base.hpp"
#include "Bibliotekum_Ultimatum/Tools/Tools.hpp"

#include "Bibliotekum_Ultimatum/Graphics/Core/Core.hpp"
#include "Bibliotekum_Ultimatum/Graphics/OpenGL/Renderers/Renderer2D.hpp"
#include "Bibliotekum_Ultimatum/Graphics/OpenGL/Renderers/Renderer3D.hpp"
#include "Bibliotekum_Ultimatum/Graphics/OpenGL/Renderers/BillboardRenderer.hpp"
#include "Bibliotekum_Ultimatum/Graphics/OpenGL/Renderers/SkyBoxRenderer.hpp"
#include "Bibliotekum_Ultimatum/Graphics/OpenGL/Renderers/LightManager.hpp"

#endif
```

*main.cpp* - Zde se nachází samotný *program*:

```
#include "header.hpp"

class Main : public ulm::Program{
public:
    Main() {
        /* Initialization */
    }
}
```

```

void update(float deltaTime){
    /* Run each frame */
}

void render(ulm::Eye eye){
    /* Render frame */
}

void resizeCallback(int width, int height){
    /* When resizing */
}

~Main() {
    /* Destruction */
}
};

ulm::Program * ulm::Properties::onStart(){
    ulm::Window::initialize(BU_APP_NAME, 256, 256);
    ulm::Window::maximize(true);
    return new Main();
}

void ulm::Properties::handleError(ulm::String error){
    printf("\n%s\n", error.getPtr());
    fflush(stdout);
}

```

Zde je potřeba implementovat funkce *ulm::Properties::handleError*, pro určení, jak nakládat s errory a *ulm::Properties::onStart()*, která říká, co se má stát při začátku aplikace a který *program* se má spustit jako první. Programem se myslí třída, která dědí z třídy ***ulm::Program***, tedy například v ukázce je to třída *main*.

Pro používání *IDEalu* doporučuji textový editor *Visual Studio Code*, v němž při práci na daném projektu si otevřeme složku *main*. Také si můžeme na nějaké klávesy namapovat, v souboru *keybindings.json*, terminálové příkazy na kompliaci projektu, například takto:

```
...
{
  "key": "f7",
  "command": "workbench.action.terminal.sendSequence",
  "args": { "text": "cd Compilation_scripts\n      ./make-androidVR-test.sh\n      cd ..\n" }
},
{
  "key": "f8",
  "command": "workbench.action.terminal.sendSequence",
  "args": { "text": "cd Compilation_scripts\n      ./make-android-test.sh\n      cd ..\n" }
},
{
  "key": "f11",
  "command": "workbench.action.terminal.sendSequence",
  "args": { "text": "cd Compilation_scripts\n      ./make-test.sh\n      cd ..\n" }
},
...

```

# 3. Podrobný popis a teorie

## 3.1 IDEal

Účelem vývojového prostředí (*Integrated Development Environment*) **IDEalu** je automatizovat zakládání projektů využívajících knihovnu **Bibliotekum\_Ultimatum** a jejich komplikaci.

K tomuto je pro jednoduchost a spolehlivost využíváno *bash skriptů* na *Linuxu* a *batch skriptů* na *Windows*. A samozřejmě také *C++* kompilátorů typu *GCC*, jako například *clang*, *gcc* nebo *NDK Toolchain*. Pro vytvoření android aplikace je ještě využito *Android Software Development Kit* (dále jen „*SDK*“) a *Java Development Kit* (dále jen „*JDK*“).

### 3.1.1 Kompilace pro Linux (Ubuntu)

Pro využití *OpenGL* je potřeba kompilovat s následujícími vlajkami:

```
-lGL -ldl -lXinerama -lXrandr -lXi -lXcursor -lX11 -lXxf86vm -lpthread
```

Také potřebujeme staticky nalinkovat knihovnu *SDL*:

```
Bibliotekum_Ultimatum/Binaries/SDL/libSDL2.a
```

Dále by bylo užitečné nadefinovat *makra* určující vlastnosti výsledného programu:

```
-DBU	SDL -DBU_APP_NAME=jmenoAplikace -DBU_LINUX -DBU_DESKTOP
```

Tato komplikace je automatizována *bash skriptem make.sh*

*Bash skript make-final.sh* navíc přidá vlajku *-O2* pro optimalizaci.

### 3.1.2 Kompilace pro Windows

Pro využití *OpenGL* je potřeba kompilovat s následujícími vlajkami:

```
-lopengl32 -lshell32
```

Také potřebujeme nalinkovat knihovnu *SDL*:

```
-lSDL2main -lSDL2
```

Dále by bylo užitečné nadefinovat *makra* určující vlastnosti výsledného programu:

```
-DBU	SDL -DBU_APP_NAME=\"jmenoAplikace\" -DBU_WINDOWS -DBU_DESKTOP
```

Tato komplikace je automatizována *batch* skriptem *make.bat*

*Batch* skript *make-final.bat* navíc přidá vlajku *-O3* a *-Xlinker /subsystem:windows*.

### 3.1.3 Kompilace pro Android

Zde to již není tak přímočaré. Aplikace pro Android se vyvíjejí v tzv. *JVM* jazycích, které se komplikují do *Java bytekódu* a jsou spouštěny virtuálním strojem *Java Virtual Machine*.

Jazyky *C/C++* do této skupiny programovacích jazyků nepatří. Ty se komplikují přímo do strojového kódu.

Pro „základ“ Android aplikace jsem si vybral jazyk *Java*, ta může používat funkce definované v *C/C++* a implementované ve sdílené knihovně pomocí rozhraní *Java Native Interface*. Pokud bychom tedy volali ve správném pořadí určité funkce v této sdílené knihovně, tak bychom mohli pomocí *C++* funkcí vykreslovat a předávat jim vstupy .

*C/C++* kód můžeme zkompilovat do sdílené knihovny pro android (.so) pomocí *Native Development Kit* (dále jen „*NDK*“), zde se nachází mimo jiné nástroj *NDK Toolchain*, který je ve stylu *GCC* komplikátoru, a tak ho budeme používat.

*SDL* již tuto *JVM* část aplikace naštěstí definuje, a tak již z *C++* máme přístup ke spoustě funkcí.

Ale jakmile vložíme do *.apk* souboru naši složku *resources* se soubory („*assety*“), které projekt využívá, tak se vlastně zazipují. *JVM* část je tedy nutné ještě rozšířit o zkopírování všech *assetů* ve složce *resources* projektu buď do externího úložiště, pokud je to možné, nebo do interního. Díky čemuž můžeme se soubory v této složce následně pracovat pomocí klasických *C* funkcí jako je *fscanf* nebo *sprintf*.

*SDL* není příliš kompatibilní s knihovnou *Google VR SDK*, která je využívána pro mobilní virtuální realitu. Navíc ve virtuální realitě příliš s Androidem stejně interagovat nepotřebujeme, stačí nám jen znát rotaci telefonu v prostoru, případně nějaké kliknutí nebo *Cardboard trigger* či vstup z gamepadu a na základě toho něco vykreslovat. Není tedy problém si tuto část vytvořit pro VR.

### 3.1.3.1 Základní postup komplilace pro Android

Nejprve zkompilujeme náš *C/C++* kód pro API level 21 (Android 5.0 a vyšší, podporuje *OpenGL ES 3.1*)<sup>1</sup>, ale pro různé *Application Binary Interface* (dále jen „(*ABI*)<sup>1</sup>). *ABI* jsou specifické různým architekturám procesorů, na kterých může Android běžet.

Rozlišujeme *ABI*: *armeabi-v7a*, *arm64-v8a*, *x86* a *x86\_64*.

Využijeme tedy tyto *NDK Toolchains*: *aarch64-linux-android21-clang++*, *armv7a-linux-android21-clang++*, *i686-linux-android21-clang++* a *x86\_64-linux-android21-clang++*.

Získané sdílené knihovny vložíme do příslušné složky ve složce *lib* s názvem specifikujícím *ABI*.

Dále pomocí *Android Asset Packaging Tool* (dále jen „*AAPT*“) na základě složky *res* a *XML* souboru *Android Manifestu* vytvoříme *R.java* soubor, který umožňuje ostatním *java* souborům v balíčku specifikovaném v *Android Manifestu* přístup k souborům uloženým ve složce *res*. Nástroj *AAPT* najdeme v *Android SDK*.<sup>2</sup>

Pomocí nástroje *javac* nyní přeložíme naše *.java* soubory, včetně *R.java*, do *java bytekódu*. Nástroj *javac* najdeme v *JDK*.<sup>3</sup>

Všechny soubory v *java bytekódu* nyní zkompilujeme do ***Dalvikova bytekódu***, který je specifický Androidu. Tuto komplikaci provedeme nástrojem *dx*, který se

nachází v *Android SDK*.<sup>4</sup>

Následně soubory s *Dalvikovým bytekódem*, složku *res*, složku se sdílenými knihovnami a složku *assets*, ve které máme uložené *resources* našeho projektu, poskládáme do *.apk* souboru pomocí nástroje *AAPT*.

Tento *.apk* soubor je následně nutné *podepsat klíčem* obsahující informace o společnosti vyvíjející tuto aplikaci. Bez této fáze nebude *.apk* soubor spustitelný, a tak se v ***IDEalu*** ve složce *Link/Keystore* nachází klíč s nesmyslnými údaji, kterým jsou aplikace *podepisovány*. Tento soubor je samozřejmě možné nahradit. Tuto fázi provádí nástroj *apkigner* nacházející se v *Android SDK*.<sup>5</sup>

A nakonec soubor *.apk* *zarovnáme* pomocí nástroje *zipalign*, který se nachází v *Android SDK*.<sup>6</sup>

Tímto postupem nám vzniknou dva soubory *.apk*, jeden z nich je instalovatelný, určený pro ladění, a druhý je určený pro distribuci přes *Google Play*.

Celý tento postup je automatizován *Bash skriptem build.sh*, který se vždy nachází ve složce Android projektu vygenerovaného ***IDEalem***. Tento skript je poté volán skripty *make-android.sh* a *make-androidVR.sh*.

## 3.2 Bibliotekum\_Ultimatum

Účelem knihovny **Bibliotekum\_Ultimatum** (dále jen „**BU**“) je zjednodušení vývoje multiplatformních aplikací v *C/C++*, především těch *grafických*.

### 3.2.1 C++

Jazyk *C++* jsem si vybral kvůli jeho elegantnosti, rozšířenosti, multiplatformnosti a výrazných kompilačních optimalizací.

Všechny funkce a třídy v **BU** používají namespace **ulm**.

### 3.2.2 Základní nástroje

#### 3.2.2.1 Dynamické pole - **ulm::Array**

Třída **ulm::Array** implementuje dynamické pole, jehož časová složitost přidávání  $n$  prvků je lineární.

#### 3.2.2.2 String - **ulm::String**

Třída obalující **ulm::Array** pro využití jako *String*.

#### 3.2.2.3 Bytové dynamické pole - **ulm::ByteArray**

Tato třída dědí ze třídy **ulm::Array** a je možné do ní přidávat jakýkoliv datový typ, který se do něj zapíše jako série *bytů*, využívá se především pro inicializaci *Vertex Buffer Objectu* a *Uniform Buffer Objectu*.

### 3.2.3 Využití open source knihoven

#### 3.2.3.1 Vytvoření okna - **SDL**

Důležité funkce pro manipulaci s oknem v knihovně *SDL* jsou obaleny ve třídě **ulm::Window**. Ta je definována takovým způsobem, aby ji bylo případně možné podložit jakoukolivjinou knihovnou, jako je tomu například při sestavování projektu pro mobilní virtuální realitu.<sup>7</sup>

### 3.2.3.2 Vytvoření okna (Activity) pro virtuální realitu na Android - Google VR SDK

#### 3.2.3.2.1 Princip virtuální reality

Pro každé oko vykreslíme svět z trochu posunuté pozice, čímž dodáme obrazu 3D efekt. Tento obraz každého oka se ještě musí deformovat inverzně k deformaci, kterou provádí čočka, kterou se na obraz díváme. Tato deformace je unikátní pro každý headset a obvykle se zapisuje do QR kódu, který je poté načten Androidem a uloží si parametry této deformace. Obraz za nás deformuje knihovna *Google VR SDK*.<sup>8</sup>

#### 3.2.3.2.2 Implementace

Okno je definováno třídou (v *Javě*), která dědí ze třídy *GVRActivity* a implementuje třídu *GVRView*. Následně při získání vstupu, či při vykreslování volá příslušné *nativní* funkce přes *Java Native Interface*

Z hlediska *C++* kódu je potřeba upravit projekční matici kamery a její vektory pohledu podle výstupu ze třídy ***ulm::CardboardVR***:

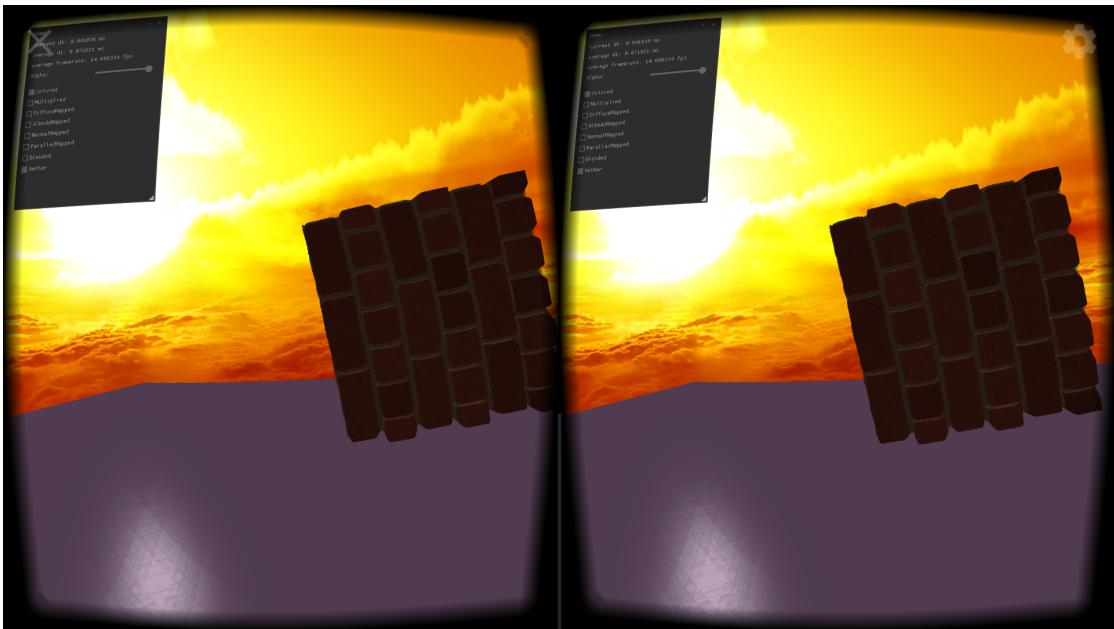
```
...

```

```

#ifndef BU_VR
/* Handle VR */
cam.position -= (float)(eye) *
    ulm::CardboardVR::getRight() *
    eyeDistance;
//pozici kamery vratime zpaky
#endif
}

```



Obrázek 3.1: Ukázka aplikace v *BU* na *Android VR*

### 3.2.3.3 GUI - Nuklear

Ke správě grafického uživatelského rozhraní (dále jen „*GUI*“) využívám knihovny Nuklear. Tato knihovna je zcela nezávislá na vykreslovači, a tak je pro tento projekt velmi vhodná.<sup>9</sup>

Vykresluje se zatím pomocí upraveného ukázkového *SDL* a *OpenGL* vykreslo-

vače.

### 3.2.3.4 Manipulace s obrázky - STB

Pro načítání obrázků různých formátů je využito knihovny STB.

Třída `ulm::Image` obaluje některé funkce této knihovny pro jednodušší používání.<sup>10</sup>

### 3.2.3.5 Matematika - GLM

Pro matematické operace jako je vektorové násobení, maticové násobení, vytváření perspektivní matice a podobně je zde využito knihovny **GLM**. **GLM** znamená *OpenGL Mathematics*, je to tedy knihovna přímo určená pro využití s *OpenGL*.<sup>11</sup>

### 3.2.4 Grafika - OpenGL

OpenGL (Open Graphics Library) je průmyslový standard specifikující multiplatformní rozhraní (API) pro tvorbu aplikací počítačové grafiky.<sup>12</sup>

Důvodem, proč jsem si vybral *OpenGL* pro tento projekt, je především ten, že se jedná o široce podporované *multiplatformní* grafické *API* a oproti ostatním grafickým *API* je vysokoúrovňové, je tedy vhodný pro začátečníky.

Pro počítače je v *BUT* využito verze *OpenGL 3.3*, zatímco pro mobilní zařízení verze *OpenGL ES 3.1*.

#### 3.2.4.1 OpenGL rendering pipeline

Jakmile pomocí OpenGL spustíme *draw call*, tedy necháme něco vykreslit, tak samotné vykreslování probíhá v následujících fázích, které si můžeme stručně popsat takto<sup>13</sup>:

##### 3.2.4.1.1 Vertex Specification

V této fázi jsou jednotlivým během programu *Vertex Shaderu* přiřazovány vstupy.

Použijeme-li k vykreslení *draw call* *glDrawArrays*, tak tyto vstupy definuje pouze *Vertex Buffer Object* (dále jen „*VBO*“).

Na tomto *VBO* také musíme definovat *úseky atributů* a v rámci těchto úseků jednotlivé *atributy*. K tomu využijeme především příkazů *glVertexAttribPointer* a *glEnableVertexAttribArray*.

Použijeme-li k vykreslení *draw call* *glDrawArrays*, tak tyto vstupy definuje pouze *Vertex Buffer Object* (dále jen „*VBO*“).

Dále je třeba specifikovat, zda se jedná o klasické *VBO* nebo o *instancované VBO*. V klasickém *VBO* má každý běh *vertex shaderu* ve všech *instancích* vlastní atributový úsek, jenž přijímá jako vstup. V instancovaném *VBO* mají všechny běhy *vertex shaderu* jedné *instance* přiřazen stejný *atributový úsek*, který přijímají jako vstup. K tomu využijeme příkazu *glVertexAttribDivisor*<sup>14</sup>. *Atributové úseky* ve své podstatě představují jednotlivé vrcholy (*vertexy*) objektu, který vykreslujeme.

Plochu, kterou vykreslujeme, nazýváme *primitiv*. Může se jednat buď o bod, úsečku či trojúhelník. V *BUT* se ovšem setkáme pouze s vykreslováním trojúhelníků.

Nyní je primitiv definován *atributovými úseky* (*vertexy*), které se nacházejí ve *VBO* hned za sebou. Někdy bychom ale chtěli, pro snížení paměťové složitosti, specifikovat pořadí těchto *vertexů*. Například pokud vykreslujeme obdélník dvěma trojúhelníky, tak je zbytečné pro každý tento trojúhelník definovat tři *atributové úseky*, protože dva a dva budou vždy shodné. Toto pořadí specifikujeme pomocí *Index Buffer Objectu* (dále jen „*IBO*“). Zde jsou za sebou specifikovány *indexy vertexů*, tvořících *primitivy*. V tomto případě je nutné použít *draw call* ***glDrawElements***.<sup>15 16</sup>

### 3.2.4.1.2 Vertex Shader

*Vertex Shader* je program, jehož každý běh představuje určení pozice jednoho vrcholu (*vertexu*) v tzv. *clipovacím prostoru*.

Vstupem tohoto programu je *úsek atributů* a jeho výstupem je pozice vrcholu (*vertexu*) v tzv. *clipovacím prostoru*.

Všechny *Vertex Shadery* v ***BU*** jsou psané v jazyce *GLSL* verze 3.30.6 pro počítače a v jazyce *GLSL ES* ve verzi 3.00.6 pro mobilní platformy (Android). Tyto jazyky jsou až na vzácné vyjimky shodné.<sup>17</sup>

*Vertex Shaderům* můžeme také jako vstupy posílat kromě *atributů* také například *uniformy*, které se nastavují před *draw calleem* jednotně pro všechny běhy všech *shaderů* v tomto *draw callu*.

*Vertex Shader* také může posílat hodnoty *Fragment Shaderu*<sup>18</sup>.

Následující *Vertex Shader* dostane jako vstup atribut, definující pozici *vertexu* v prostoru *meshe*, kterou transformuje pomocí matici *MVP*, kterou získá jako *uniform*, do *clipovacího prostoru* (*clip space*). Další atribut určuje barvu *vertexu*.

```
#ifdef GL_ES
    #version 300 es
#else
    #version 330
#endif

layout(location = 0) in vec3 pozice; //atribut;
//pozice vertexu v prostoru meshe
```

```

layout(location = 1) in vec3 barva; //attribut; barva
uniform mat4 MVP; //uniform, MVP matice

out vec3 barva_pixelu; //výstup do Fragment Shaderu

void main() {
    barva_pixelu = barva;
    gl_Position = MVP * vec4(pozice, 1.0); //výstup,
                                              //pozice v clip spacu
}

```

### 3.2.4.1.3 Geometry Shader

Tato fáze je pro *OpenGL* volitelná, ale v *OpenGL ES* se nenachází. Z toho důvodu není v *BÚ* využita<sup>19</sup>.

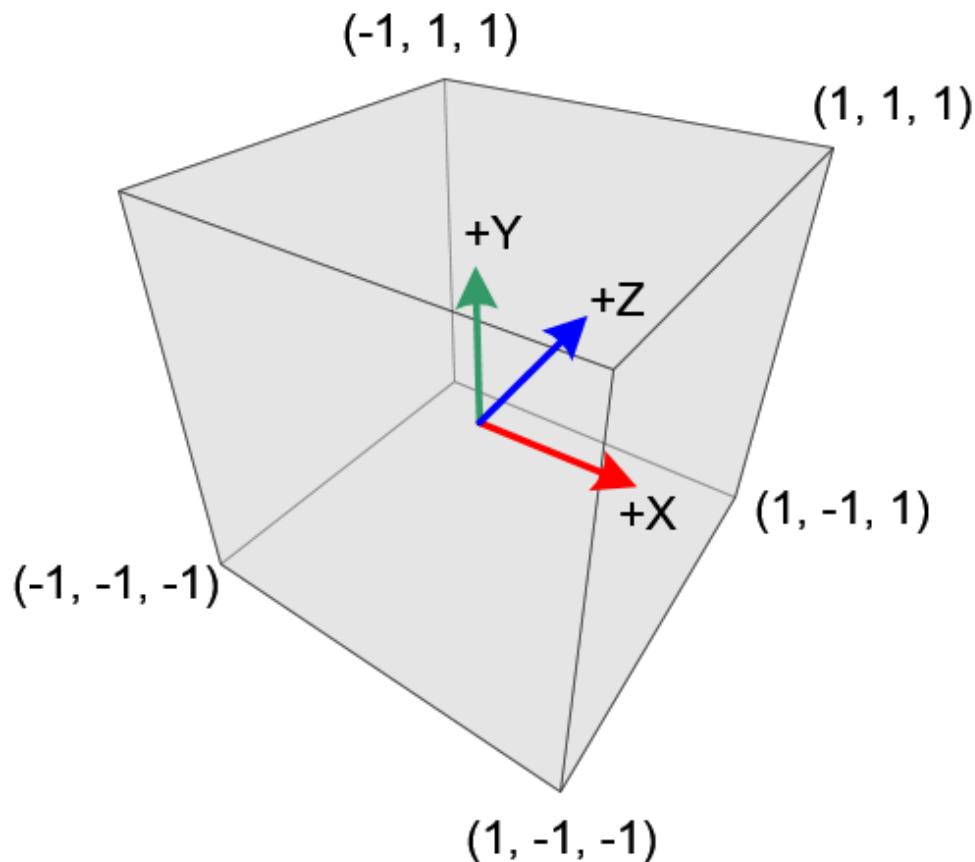
### 3.2.4.1.4 Vertex post-processing

Zde je výstup z *Vertex Shaderu*, tedy pozice *vertexu* v *clipovacím prostoru*, transformována do *normalizovaných souřadnic zařízení* pomocí *perspektivního dělení* a následně do *viewport souřadnic*, tedy souřadnic definovaných pomocí pixelů na obrazovce, přesněji řečeno *Framebuffera*.<sup>20</sup>

Tuto fázi provádí *OpenGL* automaticky.

#### 3.2.4.1.4.1 Normalizované souřadnice zařízení

Perspektivním dělením výstupní pozice z *Vertex Shaderu*, získáváme pozici *vertexu* v tzv. *normalizovaných souřadnicích zařízení*. Vykresleny budou pouze ty *primitivy*, nebo části *primitiv*, které se nacházejí uvnitř krychle se středem v počátku a se stranou dvě, viz. obrázek 3.2. Více informací v sekci Matice perspektivní projekce a *clipovací prostor*.



Obrázek 3.2: Krychle v normalizovaných souřadnicích zařízení, jejíž objem je zobrazen do *Framebuffera*<sup>46</sup>

### 3.2.4.1.5 Primitive Assembly

*Vertexty* jsou převedeny na *primitivy*.<sup>21</sup>

Tuto fázi provádí *OpenGL* automaticky.

### 3.2.4.1.6 Rasterization

V rasterizaci jsou *primitivy* převedeny na tzv. *fragmenty*, což jsou pixely ve *Framebuffelu*, které jsou daným *primitivem* překrývány.<sup>22</sup>

Tuto fázi provádí *OpenGL* automaticky.

### 3.2.4.1.7 Fragment Shader

*Fragment Shader* je program, jehož každý běh určuje barvu pixelu. Stane se pro každý pixel, jež je zakrýván nějakým *primitivem*. Může přijímat vstup z *Vertex Shaderu*, přičemž není-li specifikováno jinak, je tento vstup váženým průměrem příslušných výstupů z běhu *Vertex Shaderu vertexů*, které tvoří daný primitiv. Čím blíže je daný *fragment* nějakému z *vertexů*, potom je jeho vstup více ovlivněn výstupem jeho *Vertex Shaderu*.

Všechny fragmenty, které se nacházejí mimo *normalizované souřadnice zařízení* budou zahozeny.<sup>23</sup>

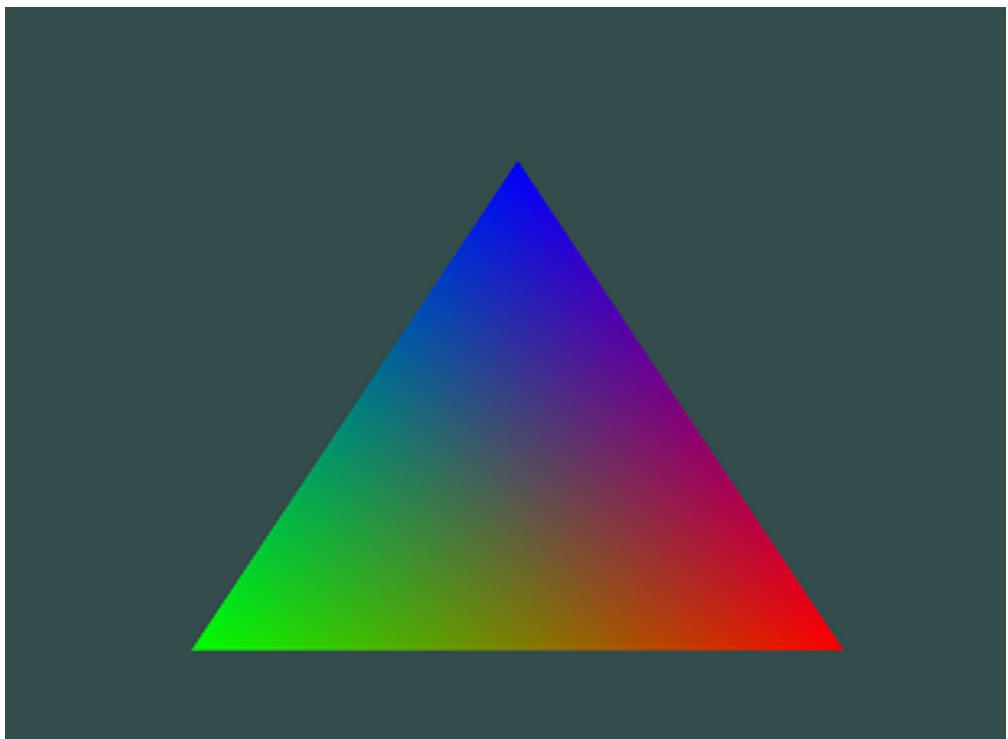
Následující *Fragment Shader* přijme jako vstup z *Vertex Shaderu* barvu a tu nastaví jako svůj výstup:

```
#ifdef GL_ES
    #version 300 es
#else
    #version 330
#endif

in vec3 barva_pixelu; //vstupní barva
out vec4 vystupni_barva; //výstup Fragment Shaderu ,
                           //specifikuje barvu pixelu

void main(){
    vystupni_barva = barva_pixelu;
}
```

Pokud bychom vykreslovali trojúhelník a výstupem z levého dolního *vertexu* byla zelená, pravého dolního červená a horního modrá, tak by tento trojúhelník vypadal takto:



Obrázek 3.3: Demonstrace váženého průměru výstupů z Vertex Shaderu<sup>24</sup>

Tohoto váženého průměru můžeme využít například při vyhlazování normálových vektorů nebo při mapování texturových souřadnic.

#### 3.2.4.1.7.1 Per-Sample Processing

V této fázi se rozhoduje, zda fragment bude vykreslen či ne. Způsoby rozhodování vždy specifikujeme před *draw call*em.<sup>25</sup>

Může se zde provádět například *depth-testing* a *stencil-testing*:

#### 3.2.4.1.7.2 Depth-Testing

*Primitivy* se překreslují podle jejich pořadí v *IBO* nebo případně ve *VBO*. Takto by tedy překreslení *fragmentů* nebylo nijak závislé na hloubce z pohledu obrazovky. Mohlo by se nám tedy stát, že *fragmenty*, které jsou blíže ke kameře než jiné, tak budou jimi překresleny.

Tento problém řeší právě *Depth-Testing* (*hloubkové testování*). Při rozhodování

zda *fragment* vykreslit je porovnána *z*-ová souřadnice zapsaná v *Depth Bufferu* s novou *z*-ovou souřadnicí. Pokud *fragment* tímto porovnáním projde, tak nejen že se může vykreslit, ale také může přepsat příslušnou hodnotu v *Depth-Bufferu* na svou *z*-ovou souřadnici.<sup>26</sup>

#### 3.2.4.1.7.3 Stencil-Testing

*Stencil-Testing* (šablonové testování) se provádí pomocí *Stencil Bufferu*, zde jsou pro každý pixel uložené hodnoty, podle nichž se rozhoduje zda vykreslit či nevykreslit *fragmenty* o dané pozici. *Fragment*, který testem projde, může přepsat hodnotu ve *Stencil Bufferu*.<sup>27</sup>

Pro fragment, který neprojde *Stencil-Testingem*, tak nebude poslán ani do fáze *Fragmentového shaderu*. Je tedy možné ho díky tomuto použít v některých případech pro optimalizaci.

### 3.2.4.2 Zatím nezmíněné OpenGL objekty

#### 3.2.4.2.1 Uniform Buffer Object

*Uniform Buffer Object* (dále jen „*UBO*“) funguje podobně jako *uniformy*, mohou k němu tedy přistupovat všechny *shaderové programy*.

Obvykle může být větší než například pole *uniformů*, ale stále je na velikosti výrazně omezen. V nejhorším případě bude jeho maximální velikost 64KB<sup>28</sup>. Potenciálně také může být aktualizován rychleji než *uniformy* o stejně velikosti.<sup>29</sup>

#### 3.2.4.2.2 Textura

*Uniformy* a *VBO* mají obvykle dosti omezenou velikost a nehodí se například k mapování barvy, či obecně k držení paměti o velké velikosti.

Naopak *textura* může obvykle obsahovat velké množství dat a dá se využít například na mapování barvy, světelných vlastností, výšky, normálových vektorů a na cokoliv, kde se pracuje s velkým množstvím dat.<sup>30</sup>

Podle použití texturového objektu můžeme použít různé *cíly*. V *BÚ* se především používají tyto dva:

### **3.2.4.2.2.1 GL\_TEXTURE\_2D**

Tento *cíl* obsahuje dvourozměrné pole vektorů o maximálním počtu členů čtyři. Využívá se například na mapování barvy, světelných vlastností, výšky, normálových vektorů apod.

V *Shaderech* se k hodnotě v textuře tohoto *cíle* přistupuje pomocí funkce *texture*, která jako argument bere index slotu, kde je textura navázána a vektor o počtu členů dva, specifikující pozici v textuře.

### **3.2.4.2.2.2 GL\_TEXTURE\_CUBE\_MAP**

Tento *cíl* obsahuje šest dvourozměrných polí vektorů o maximálním počtu členů čtyři. Tato pole reprezentují stěny krychle se středem v počátku. Využívá se například na mapování oblohy.

V *Shaderech* se k hodnotě v textuře tohoto *cíle* přistupuje pomocí funkce *texture*, která jako argument bere index slotu, kde je textura navázána a jednotkový vektor o počtu členů tři, specifikující směr ukazující na pozici v textuře.<sup>31</sup>

### 3.2.4.3 2D grafika

Pro dvourozměrné vykreslování je v ***BUT*** určena třída ***ulm::Renderer2D***.

#### 3.2.4.3.1 Polygon - ***ulm::Sprite***

Dvourozměrný objekt si můžeme nadefinovat pomocí pole jeho vrcholů, jejich barvy a texturových souřadnic.

K tomuto nadefinování slouží třída ***ulm::Sprite***, jejímiž hlavními atributy jsou:

```
ulm::VertexArray vertexArray;  
  
ulm::Array<glm::vec2> vertices;  
ulm::Array<glm::vec2> texture_coordinates;  
ulm::Array<glm::vec4> colors;  
ulm::Array<unsigned int> indices;  
  
glm::mat4 modelMat;  
  
glm::vec4 colorMultiplier;  
ulm::Texture texture;
```

Prvky se stejnými indexy v polích *vertices*, *texture\_coordinates* a *colors* postupně tvoří *úsek atributů*, ty se při inicializaci zapíší do *VBO*. Pole *indices* reprezentuje *IBO* a při inicializaci je uloženo právě do *IBO*.

Pole *vertices* obsahuje pozici *vertexů*.

Pole *texture\_coordinates* obsahuje texturové souřadnice.

Pole *colors* reprezentuje barvu *vertexů*.

*VBO* a *IBO* jsou spravovány atributem *vertexArray* typu ***ulm::VertexArray***.

#### 3.2.4.3.1.1 Vykreslení spritu

Vykreslení pomocí třídy ***ulm::Renderer2D*** proměnných *sprite* a *sprite2* typu ***ulm::Sprite*** bude vypadat přibližně takto:

```

ulm::Renderer2D::begin(kamera);
ulm::Renderer2D::draw(sprite, vlajky);
ulm::Renderer2D::draw(sprite2, vlajky);
ulm::Renderer2D::end();

```

Při začátku vykreslování (*ulm::Renderer2D::begin*) specifikujeme kameru (typu ***ulm::Camera***), která bude transformovat pozice spritů. Můžeme také místo ní poslat transformační matici typu ***glm::mat4***.

Poté vykreslujeme jednotlivé *sprity* pomocí vlajek. Tyto vlajky jsou reprezentovány pomocí bitů v datovém typu *16-bitového integeru*.

Vlajky mohou nabývat hodnot:

*ulm::Colored* - barvy definované na každý vertex budou použity.

*ulm::Multiplied* - výsledná barva bude vždy vynásobena vektorem *colorMultiplier*, který je atributem třídy ***ulm::Sprite***.

*ulm::Textured* - bude použita barva v textuře *texture*, která je atributem třídy ***ulm::Sprite***, podle příslušných texturových souřadnic.

*ulm::DepthTextured* - bude použita barva v textuře *texture*, která je atributem třídy ***ulm::Sprite***, podle příslušných texturových souřadnic. Navíc ovšem očekává, že texturou je *depth buffer* a tak bude použit pouze jeden kanál této textury a hodnota hloubky bude linearizována. Neočekává se, že by uživatel ***BUT*** tuto vlajku používal.

Při vykreslování se také využívá *alpha blendingu* (*alpha transparentnosti*). Přičemž k němu dojde podle pořadí vykreslení *spritů*.

Vykreslení fialového obdélníku pomocí ***BUT*** uprostřed obrazovky bude tedy vypadat následovně:

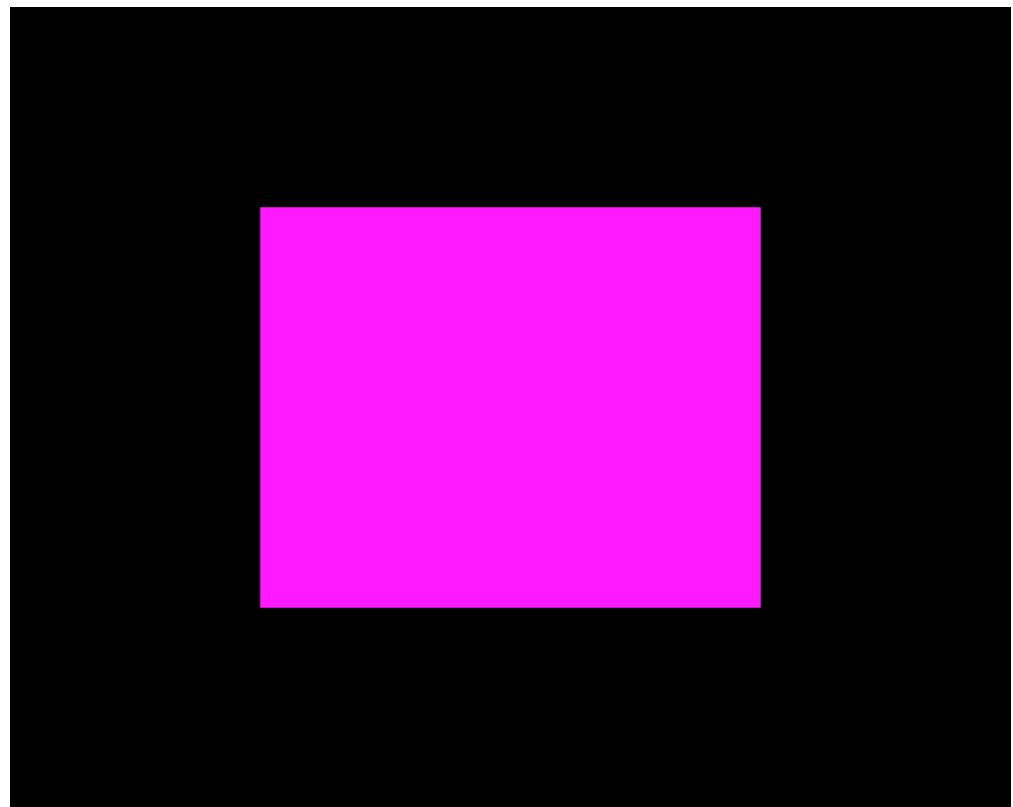
```

class Main : public ulm::Program{
public:
    ...
    ulm::Sprite obdelnik;

Main() {
    /* Initialization */
}

```

```
obdelnik = ulm::SpriteFactory::createRectangle(  
    -0.5f, -0.5f, //pozice  
    1.f, 1.f, //velikost  
    glm::vec4(1.0f, 0.1f, 1.0f, 1.f);  
    //barva  
    ...  
}  
...  
  
void render(ulm::Eye eye){  
    ...  
  
    ulm::Renderer2D::begin(glm::mat4(1.f));  
    ulm::Renderer2D::draw(obdelnik, ulm::Colored);  
    ulm::Renderer2D::end();  
  
    ...  
}  
...  
}
```



Obrázek 3.4: Vykreslený obdélník

#### 3.2.4.3.2 ShaderToy - *ulm::ShaderToy*

Ne všechny dvourozměrné objekty se dají vyjádřit pomocí polygonů. Z tohoto důvodu tady existuje třída *ulm::ShaderToy* a zjednodušuje celý vykreslovací proces na pouze *fragment shader*, ten se zde píše v jazyce *GLSL* verze 100.

Vykreslení kolečka:

```
class Main : public ulm::Program{
public:
    ...
    ulm::ShaderToy st;

Main() {
    /* Initialization */
```

```

st . code = R"====(
    varying vec2 position;

    uniform vec2 pozice;
    uniform float polomer;
    uniform float tloustka;
    uniform vec4 barvicka;

    void main()
    {
        if (length(position - pozice) < (polomer + tloustka) &&
            length(position - pozice) > (polomer - tloustka)){
            gl_FragColor = barvicka;
        } else{
            gl_FragColor = vec4(0.0 , 0.0 , 0.0 , 0.0);
        }
    })=====";

    st . compile ();
    ...
}

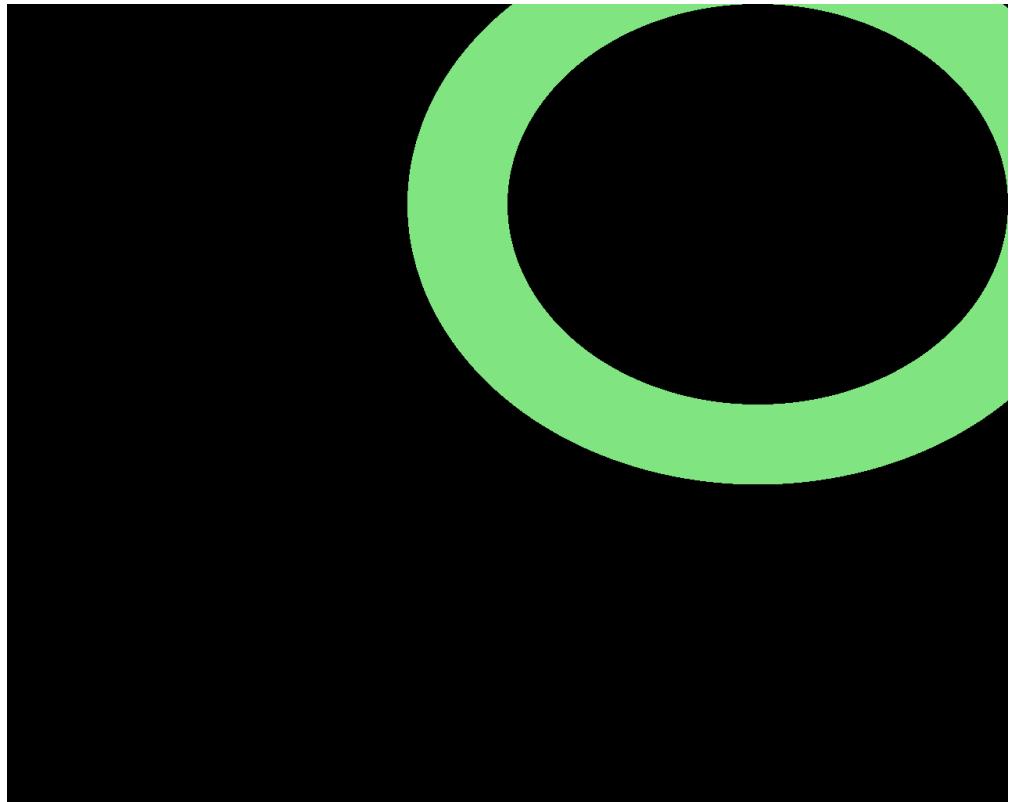
...
}

void render (ulm :: Eye eye){
    ...

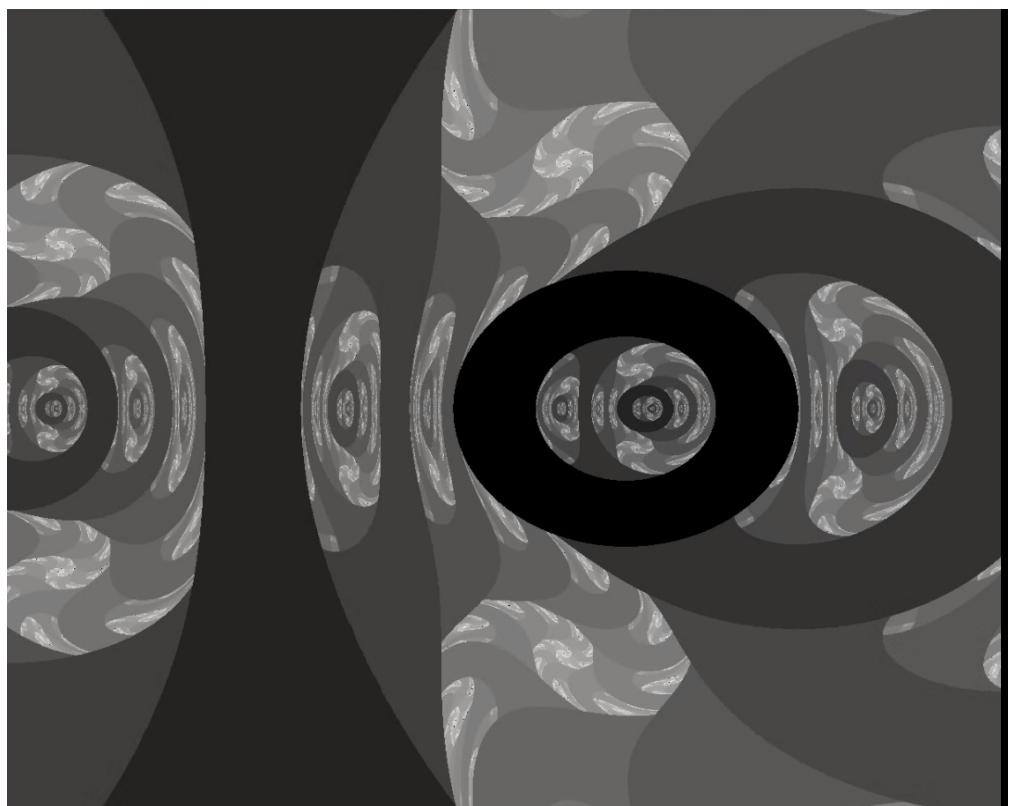
    ulm :: Renderer2D :: beginST ( st );
    st . setVec2 ( "pozice" , glm :: vec2 ( 0.5f , 0.5f ) );
    st . setFloat ( "polomer" , 0.6f );
    st . setFloat ( "tloustka" , 0.1f );
    st . setVec4 ( "barvicka" , glm :: vec4 ( 0.5f , 0.9f , 0.5f , 1.f ) );
    ulm :: Renderer2D :: drawST ( -1.f , -1.f , //pozice
                                2.f , 2.f , //velikost
                                glm :: mat4 ( 1.f ) //transformace
                                );
}

```

```
    ulm :: Renderer2D :: endST ();  
    ...  
}  
...  
}
```



Obrázek 3.5: Vykreslené kolečko



Obrázek 3.6: Fraktál vykreslený pomocí *ulm::ShaderToy*

### 3.2.4.4 3D grafika

#### 3.2.4.4.1 Mesh

V rasterizační trojrozměrné grafice se objekty obvykle popisují pomocí *mesh*, tedy souhrnů vrcholů, hran a ploch daného objektu.<sup>32</sup>

K tomu abychom v *OpenGL* vykreslili *mesh* potřebujeme znát pozice, vyjádřené pomocí trojrozměrného vektoru, všech jeho vrcholů (dále jen „*vertexů*“) a také potřebujeme znát, které jeho *vertexy* tvoří *primitivy*, v našem případě trojúhelníky.<sup>15</sup>

Pozice *vertexů* se obvykle ukládají do *VBO* a indexy *vertexů*, tvořících *primitivy* (trojúhelníky), se za sebou ukládají do *IBO*. Obvykle ale potřebujeme o objektu znát více informací než jen pozice *vertexů* jako například normálové vektory k povrchu, barvu, texturové souřadnice, tangentové vektory a podobně. Tyto informace se dále ukládají buď do *VBO* k pozicím nebo případně do textury nebo *UBO* nebo jako *uniformy*.

##### 3.2.4.4.1.1 Implementace v BU

*Mesh* je implementován ve třídě ***ulm::Mesh***, jejíž hlavními atributy jsou:

```
ulm :: Array<glm :: vec3> vertices;
ulm :: Array<glm :: vec3> normals;
ulm :: Array<glm :: vec2> texture_coordinates;
ulm :: Array<Material> materials;

ulm :: Array<glm :: vec3> tangents;

ulm :: Array<glm :: ivec4> joint_indices;
ulm :: Array<glm :: vec4> joint_weights;

ulm :: Array<unsigned int> indices;

ulm :: VertexArray va;
ulm :: MeshProperties properties;
```

Prvky o stejných indexech v polích *vertices*, *normals*, *texture\_coordinates*, *materials*, *tangents*, *joint\_indices*, *joint\_weights* postupně tvoří *atributové úseky*. Pole

*indices* představuje *IBO*. Při inicializaci *meshe* se všechny tyto informace vloží do *VBO* a *IBO*, které spravuje proměnná *va* typu ***ulm::VertexArray***.

Pole *vertices* obsahuje pozice *vertexů* v prostoru *meshe*.

Pole *normals* obsahuje normálové vektory v prostoru *meshe*.

Pole *texture\_coordinates* obsahuje texturové souřadnice.

Pole *materials* obsahuje barvy a světelné vlastnosti.

Pole *tangents* obsahuje tangentové vektory.

Pole *joint\_indices* obsahuje indexy kostí ovlivňující dané *vertexy*.

Pole *joint\_weights* obsahuje váhy kostí.

#### 3.2.4.4.2 Lineární algebra

Už tedy víme, jak definovat objekt pomocí meshe. Pokud bychom ale takovýto mesh nechali vykreslit do výchozích OpenGL souřadnic, tak nebude vypadat příliš zajímavě. Zaprvé se bude jednat o ortografickou projekci, tedy bez jakéhokoliv trojrozměrného efektu, ale také bude jen stát na místě.

Tyto meshe bychom tedy chtěli nějakým způsobem transformovat. Mohli bychom například chtít je otáčet či posouvat. Takže bychom například mohli do našeho *vertex shaderu* poslat vektor posunu a poté ho přičítat k vertexům meshe, dále bychom poslali vektor reprezentující eulerovy úhly, pomocí nichž bychom vertexy rotovali kolem počátku. Také bychom mohli chtít mesh scalovat a poslat tam další vektor, který by to popisoval. A tímto způsobem bychom mohli pokračovat ještě například s perspektivními a ortografickými projekcemi.

Toto všechno by pak bylo velice nelegantní a pravděpodobně i výpočetně náročné, naštěstí ovšem existuje lepší možnost, jak pracovat s transformacemi.

#### 3.2.4.4.3 Transformace vektoru pomocí matic

Daleko elegantnějším způsobem, jak provádět transformace vektoru, je pomocí matic a maticového násobení. Příčemž při maticovém násobení vektoru a matice pracujeme s vektorem jako by se jednalo o matici o šířce 1. Pro transformace se obvykle používá čtvercová matice o hodnosti 4 a trojrozměrný vektor se rozšíří o čtvrtý člen o velikosti 1, což umožní například translaci tohoto vektoru, ale později také *perspektivní dělení*.

Je třeba si dávat pozor na nekomutativnost maticového násobení a vektor musíme násobit maticí vždy zleva.<sup>33</sup> <sup>34</sup>

Maticové násobení matice a vektoru:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} a_{11}x + a_{12}y + a_{13}z + a_{14}w \\ a_{21}x + a_{22}y + a_{23}z + a_{24}w \\ a_{31}x + a_{32}y + a_{33}z + a_{34}w \\ a_{41}x + a_{42}y + a_{43}z + a_{44}w \end{pmatrix}$$

Další užitečnou vlastností transformačních matic je, že jejich transformace můžeme skládat pomocí maticového násobení.

Vynásobení translační matice  $T$  scalovací maticí  $S$  **zleva** a následná transformace vektoru výslednou transformační maticí:

$$\begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} S_x & 0 & 0 & S_x T_x \\ 0 & S_y & 0 & S_y T_y \\ 0 & 0 & S_z & S_z T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (3.1)$$

$$\begin{pmatrix} S_x & 0 & 0 & S_x T_x \\ 0 & S_y & 0 & S_y T_y \\ 0 & 0 & S_z & S_z T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} S_x x + S_x T_x \\ S_y y + S_y T_y \\ S_z z + S_z T_z \\ 1 \end{pmatrix} \quad (3.2)$$

Vynásobení translační matice  $T$  scalovací maticí  $S$  **zprava** a následná transformace vektoru výslednou transformační maticí:

$$\begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} S_x & 0 & 0 & T_x \\ 0 & S_y & 0 & T_y \\ 0 & 0 & S_z & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (3.3)$$

$$\begin{pmatrix} S_x & 0 & 0 & T_x \\ 0 & S_y & 0 & T_y \\ 0 & 0 & S_z & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} S_x x + T_x \\ S_y y + T_y \\ S_z z + T_z \\ 1 \end{pmatrix} \quad (3.4)$$

Na uvedených příkladech si můžeme všimnout projevu **nekomutativnosti** maticového násobení. V případě, kdy jsme scalovací maticí  $S$  násobili translační matici  $T$  zleva (3.1), tak byl scale aplikován i na translaci, zatímco při násobení zprava translace zůstává stále stejná.

Také si můžeme všimnout **asociativnosti** maticového násobení, kdy v případě (3.2) by byl výsledek stejný kdybychom nejprve provedli translaci maticí  $T$  a na výsledný vektor scale maticí  $S$ .

Výsledek maticového násobení  $S \cdot T$  (3.1) nazveme maticí  $TS$  podle pořadí, v jakém se aplikují transformace.

Protože rotace a scale se vždy provádějí vzhledem k počátku, tak je obvyklým způsobem pořadí transformací objektu  $RST$ , kde  $R$  je rotační matice,  $S$  scalovací a  $T$  translační. Tuto matici získáme součinem  $T \cdot S \cdot R$ , protože transformace jsou na vektor aplikovány násobením zleva.<sup>35</sup>

Matici  $RST$  nazveme **maticí modelu** ( $M$ ), ta transformuje *vertexy meshe* ze *souřadnicového systému meshe* (dále jen „prostor meshe“) do *souřadnicového systému světa* (dále jen „prostor světa“).

$$M = RST = T \cdot S \cdot R$$

#### 3.2.4.4.3.1 Translace

Translační matice  $T$  definována vektorem posunutí  $\vec{T}(T_x, T_y, T_z)$  vypadá takto.<sup>36</sup>

$$T = \begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (3.5)$$

Transformace vektoru touto maticí:

$$\begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + T_x \\ y + T_y \\ z + T_z \\ 1 \end{pmatrix}$$

#### 3.2.4.4.3.2 Transformační matice definována pomocí vektorů

Transformaci si můžeme na definovat pomocí vektorů

$\vec{P}(P_x, P_y, P_z)$  reprezentující translaci,

$\vec{F}(F_x, F_y, F_z)$  „vektor dopředu“ reprezentující transformaci *z-ové* souřadné osy, tedy její nový směr a také scale, který je reprezentován velikostí tohoto vektoru  $|\vec{F}|$ ,

$\vec{R}(R_x, R_y, R_z)$  „vektor doprava“ reprezentující transformaci *x-ové* souřadné osy,

$\vec{U}(U_x, U_y, U_z)$  „vektor nahoru“ reprezentující transformaci *y-ové* souřadné osy.

Takto nadefinovaná transformační matice bude vypadat takto:<sup>37</sup>

$$A = \begin{pmatrix} R_x & U_x & F_x & P_x \\ R_y & U_y & F_y & P_y \\ R_z & U_z & F_z & P_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (3.6)$$

Z takto nadefinované transformační matice si snadno můžeme odvodit i matici translační (3.5), tím že za vektory  $\vec{R}$ ,  $\vec{U}$ ,  $\vec{F}$  dosadíme jednotkové vektory rovnoběžné s osami.

Nyní z této matice  $A$  vytkneme zleva matici translační  $T$  definovanou vektorem  $\vec{P}$ :

$$A = T \cdot T^{-1} \cdot A$$

$$A = T \cdot B$$

$$B = T^{-1} \cdot A$$

$$B = T^{-1} \cdot \begin{pmatrix} R_x & U_x & F_x & P_x \\ R_y & U_y & F_y & P_y \\ R_z & U_z & F_z & P_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$B = \begin{pmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} R_x & U_x & F_x & P_x \\ R_y & U_y & F_y & P_y \\ R_z & U_z & F_z & P_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$B = \begin{pmatrix} R_x & U_x & F_x & 0 \\ R_y & U_y & F_y & 0 \\ R_z & U_z & F_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Můžeme si všimnout, že k translaci definované vektorem  $\vec{P}$ , dojde až po transformaci maticí  $B$ . Vektor  $\vec{P}$  tedy nereprezentuje translaci v původních souřadnicích

nýbrž v souřadnicích transformovaných pomocí matice  $B$ . Na toto je potřeba si dávat pozor.

Transformace touto maticí  $A$  bude tedy vypadat následovně:

$$\begin{pmatrix} R_x & U_x & F_x & P_x \\ R_y & U_y & F_y & P_y \\ R_z & U_z & F_z & P_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} R_x \cdot x + U_x \cdot y + F_x \cdot z + P_x \\ R_y \cdot x + U_y \cdot y + F_y \cdot z + P_y \\ R_z \cdot x + U_z \cdot y + F_z \cdot z + P_z \\ 1 \end{pmatrix}$$

#### 3.2.4.4.3.3 Scale

Matici scalu si můžeme velmi snadno odvodit z transformační matice (3.6) a to tak, že si za vektory  $\vec{R}$ ,  $\vec{U}$  a  $\vec{F}$  dosadíme vektory rovnoběžné se souřadnými osami, kde jejich velikosti budou reprezentovat scale na dané ose:

$$S_x = | \vec{R} |$$

$$S_y = | \vec{U} |$$

$$S_z = | \vec{F} |$$

K translaci nyní nedochází a tak nám pro scale stačí matice o hodnosti tří. Scale definovaný vektorem  $\vec{S}(S_x, S_y, S_z)$  bude tedy vypadat následovně:<sup>38</sup>

$$\begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & S_z \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} S_x x \\ S_y y \\ S_z z \end{pmatrix}$$

Transformace použitím matice o hodnosti čtyři:

$$\begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} S_x x \\ S_y y \\ S_z z \\ 1 \end{pmatrix}$$

### 3.2.4.4.3.4 Rotace

Vytvoření rotační matice je už trošku složitější. Například máme-li rotaci nadefinovanou pomocí vektoru osy rotace  $\vec{O}(o_x, o_y, o_z)$  a úhlem  $\alpha$ , tak bude rotační matice vypadat následovně<sup>39</sup>:

$$R_3 = \begin{pmatrix} \cos(\alpha) + o_x^2 \cdot (1 - \cos(\alpha)) & o_y \cdot o_x \cdot (1 - \cos(\alpha)) - o_z \cdot \sin(\alpha) & o_x \cdot o_z \cdot (1 - \cos(\alpha)) + o_y \cdot \sin(\alpha) \\ o_y \cdot o_x \cdot (1 - \cos(\alpha)) + o_z \cdot \sin(\alpha) & \cos(\alpha) + o_y^2 \cdot (1 - \cos(\alpha)) & o_y \cdot o_z \cdot (1 - \cos(\alpha)) - o_x \cdot \sin(\alpha) \\ o_z \cdot o_x \cdot (1 - \cos(\alpha)) - o_y \cdot \sin(\alpha) & o_z \cdot o_y \cdot (1 - \cos(\alpha)) + o_x \cdot \sin(\alpha) & \cos(\alpha) + o_z^2 \cdot (1 - \cos(\alpha)) \end{pmatrix}$$

K provedení rotace nám sice stačí matice o hodnosti tří, ale můžeme ji také rozšířit na hodnost čtyři s ekvivalentním transformačním efektem<sup>39</sup>:

$$R_4 = \begin{pmatrix} \cos(\alpha) + o_x^2 \cdot (1 - \cos(\alpha)) & o_y \cdot o_x \cdot (1 - \cos(\alpha)) - o_z \cdot \sin(\alpha) & o_x \cdot o_z \cdot (1 - \cos(\alpha)) + o_y \cdot \sin(\alpha) & 0 \\ o_y \cdot o_x \cdot (1 - \cos(\alpha)) + o_z \cdot \sin(\alpha) & \cos(\alpha) + o_y^2 \cdot (1 - \cos(\alpha)) & o_y \cdot o_z \cdot (1 - \cos(\alpha)) - o_x \cdot \sin(\alpha) & 0 \\ o_z \cdot o_x \cdot (1 - \cos(\alpha)) - o_y \cdot \sin(\alpha) & o_z \cdot o_y \cdot (1 - \cos(\alpha)) + o_x \cdot \sin(\alpha) & \cos(\alpha) + o_z^2 \cdot (1 - \cos(\alpha)) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotační matice je vždy maticí **ortogonální**<sup>39</sup>. Tedy její sloupce tvoří **ortonormální bázi**<sup>40</sup>. Tato matice má tu vlastnost, že její inverzní maticí je její matice transponovaná<sup>42</sup>:

$$R^{-1} = R^T \quad (3.7)$$

### 3.2.4.4.3.5 Matice pohledu

Dále by bylo vhodné nadefinovat si kamery, abychom se mohli ve světě otáčet a pohybovat.

Kameru si nadefinujeme pomocí jednotkových vektorů, které směřují nahoru od kamery  $\vec{U}$ , ve směru pohledu kamery  $\vec{F}$  a doprava vzhledem ke směru kamery  $\vec{R} = \vec{F} \times \vec{U}$  a pomocí vektoru  $\vec{T}(T_x, T_y, T_z)$ , jež reprezentuje translaci v rámci *prostoru světa*.

Transformační matici této kamery  $K$  si tedy můžeme zapsat jako maticový součin její rotační matice  $R_K$  a její translační matice  $T_K$ :

$$K = R_K \cdot T_k$$

Translační matici si vyjádříme pomocí vzorce (3.5):

$$T_K = \begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotační matici si můžeme vyjádřit pomocí vzorce (3.6):

$$R_K = \begin{pmatrix} R_x & U_x & F_x & 0 \\ R_y & U_y & F_y & 0 \\ R_z & U_z & F_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\Rightarrow K = \begin{pmatrix} R_x & U_x & F_x & 0 \\ R_y & U_y & F_y & 0 \\ R_z & U_z & F_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Musíme si ale uvědomit, že matice  $K$  transformuje vektor pozice ze *souřadnicového prostoru pohledu kamery* (dále jen „prostoru pohledu“) do *prostoru světa*. Zatímco my ale potřebujeme *mesh* transformovat z *prostoru světa*, který získáme násobením maticí *modelu M*, do *prostoru pohledu*. Jinými slovy my **nepotřebujeme transformovat kameru** ve světě, nýbrž **potřebujeme transformovat svět** do pohledu kamery. Po této transformaci, podíváme-li se z počátku po ose  $z$  v kladném směru, tak budeme vidět svět pohledem kamery.

Matici transformující vektor z *prostoru světa* do *prostoru pohledu* nazveme *maticí pohledu V*. Ta bude inverzní k matici transformace kamery  $K$ :

$$V = K^{-1}$$

$$V = R_K^{-1} \cdot T_K^{-1}$$

Potřebujeme tedy zjistit inverzní matici k matici translace kamery  $T_K^{-1}$  a inverzní matici rotace kamery  $R_K^{-1}$

Inverzní translační matici si můžeme vyjádřit jednodušše, stačí jen použít opačný translační vektor k vektoru  $\vec{T}$ :

$$T_K^{-1} = \begin{pmatrix} 1 & 0 & 0 & -T_x \\ 0 & 1 & 0 & -T_y \\ 0 & 0 & 1 & -T_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Co se získání inverzní rotační matice týče, zde to bude o něco komplikovanější. Nejprve snížíme hodnost rotační matice  $R_K$  na tři, výsledná matice bude obsahovat stejnou rotaci jako matice  $R_K$  o hodnosti čtyři:

$$R_{K3} = \begin{pmatrix} R_x & U_x & F_x \\ R_y & U_y & F_y \\ R_z & U_z & F_z \end{pmatrix}$$

Víme, že sloupce této matice  $R_{K3}$  tvoří vektory, které jsou jednotkové a také jsou navzájem kolmé. Tyto vektory tedy tvoří *ortonormální bázi*.<sup>40</sup>

Matici, jejíž sloupce tvoří ortonormální bázi, se nazývá *ortogonální matice*<sup>41</sup>.

Ortogonalní matice má spoustu zajímavých vlastností. Nyní je pro nás důležitá její vlastnost, že její **inverzní matice je rovna její matici transponované**.<sup>42</sup>

Díky této vědomosti si můžeme nyní snadno invertovat matici  $R_{K3}$ :

$$R_{K3}^{-1} = \begin{pmatrix} R_x & U_x & F_x \\ R_y & U_y & F_y \\ R_z & U_z & F_z \end{pmatrix}^{-1} = \begin{pmatrix} R_x & R_y & R_z \\ U_x & U_y & U_z \\ F_x & F_y & F_z \end{pmatrix}$$

Tato matice obsahuje inverzní rotaci k rotaci  $R_{K3}$ , tím pádem rozšířením hodnosti této matice na čtyři, získáme inverzní matici k rotační matici kamery  $R_K^{-1}$

$$R_K^{-1} = \begin{pmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ F_x & F_y & F_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Nyní dosadíme za  $R_K^{-1}$  a  $T_K^{-1}$ :

$$V = \begin{pmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ F_x & F_y & F_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & -T_x \\ 0 & 1 & 0 & -T_y \\ 0 & 0 & 1 & -T_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

A můžeme si vyjádřit vzorec pro *matici pohledu*  $V$ :

$$V = \begin{pmatrix} R_x & R_y & R_z & -T_xR_x - T_yR_y - T_zR_z \\ U_x & U_y & U_z & -T_xU_x - T_yU_y - T_zU_z \\ F_x & F_y & F_z & -T_xF_x - T_yF_y - T_zF_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (3.8)$$

### 3.2.4.4.3.6 Matice trojrozměrné projekce

Nyní provádíme transformace stále jen v rámci trojrozměrného prostoru. Obrazovka počítače je ovšem pouze dvourozměrná, a tak bychom potřebovali tento trojrozměrný prostor namapovat na dvourozměrnou plochu. K tomuto účelu slouží projekční matice.<sup>43</sup>

V 3D rasterizační grafice využíváme především *ortografických projekcí*, které jsou definovány pomocí krychle, jejíž objem zobrazujeme na plochu a *perspektivních projekcí*, která je definována komolým jehlanem, jehož objem zobrazujeme.

### 3.2.4.4.3.7 Matice ortografické projekce

Ortografická projekce ve své podstatě jen posouvá hranice os souřadnicového systému.

Definuje se obvykle pomocí šesti čísel:

*vlevo, vpravo* - představující hranice osy x,

*dole, nahore* - představující hranice osy y,

*blizko, daleko* - představující hranice osy z.

Tato čísla definují velikosti a pozici krychle, která ohraničuje oblast, jež je zobrazována touto ortografickou projekcí. Všechny *vertexy* nacházející se v této krychle se budou po transformaci nacházet v *normalizovaných souřadnicích zařízení OpenGL*, tedy všechny členy vektoru pozice se budou nacházet v intervalu

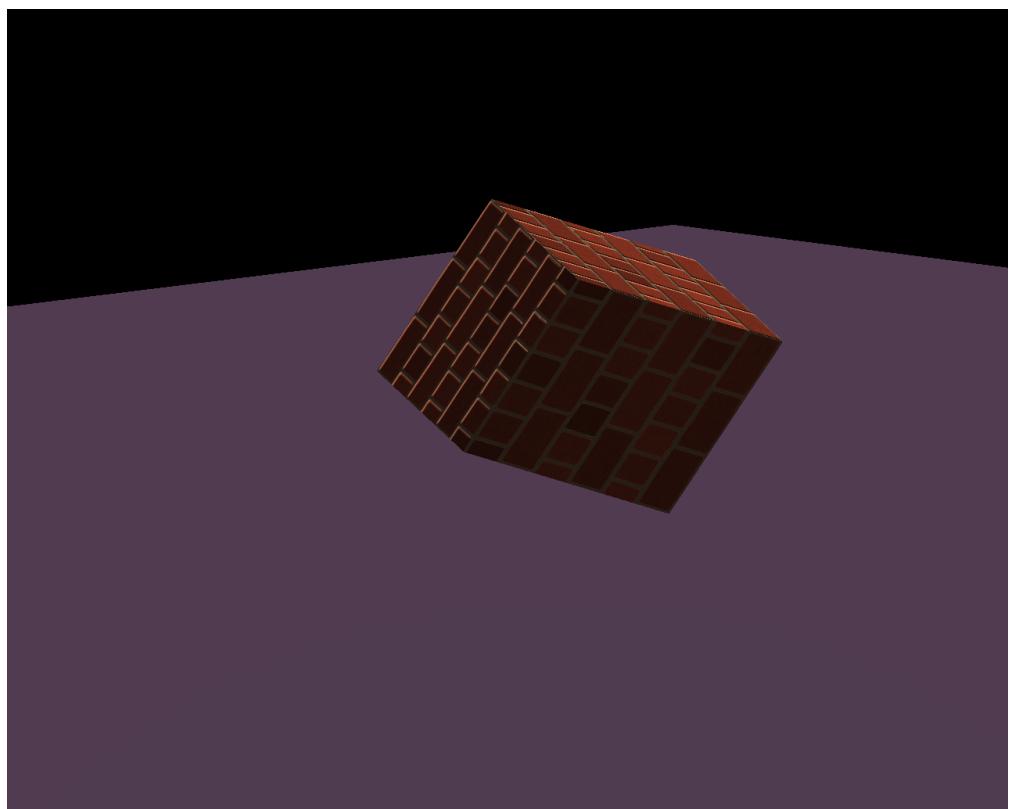
$\langle -1, 1 \rangle$  a budou vykresleny. Naopak všechny *vertexy* mimo tuto krychli se po transformaci budou nacházet mimo *normalizované souřadnice zařízení* a vykresleny nebudou.<sup>44</sup>

Pomocí těchto čísel můžeme získat matici ortografické projekce:<sup>45</sup>

$$P_O = \begin{pmatrix} \frac{2}{vpravo-vlevo} & 0 & 0 & \frac{vpravo+vlevo}{vlevo-vpravo} \\ 0 & \frac{2}{nahore-dole} & 0 & \frac{nahore+dole}{dole-nahore} \\ 0 & 0 & \frac{2}{blizko-daleko} & \frac{daleko+blizko}{blizko-daleko} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Transformace vektoru touto maticí bude potom:

$$\begin{aligned} & \begin{pmatrix} \frac{2}{vpravo-vlevo} & 0 & 0 & \frac{vpravo+vlevo}{vlevo-vpravo} \\ 0 & \frac{2}{nahore-dole} & 0 & \frac{nahore+dole}{dole-nahore} \\ 0 & 0 & \frac{2}{blizko-daleko} & \frac{daleko+blizko}{blizko-daleko} \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \\ &= \begin{pmatrix} \frac{2}{vpravo-vlevo}x + \frac{vpravo+vlevo}{vlevo-vpravo} \\ \frac{2}{nahore-dole}y + \frac{nahore+dole}{dole-nahore} \\ \frac{2}{blizko-daleko}z + \frac{daleko+blizko}{blizko-daleko} \\ 1 \end{pmatrix} \end{aligned}$$

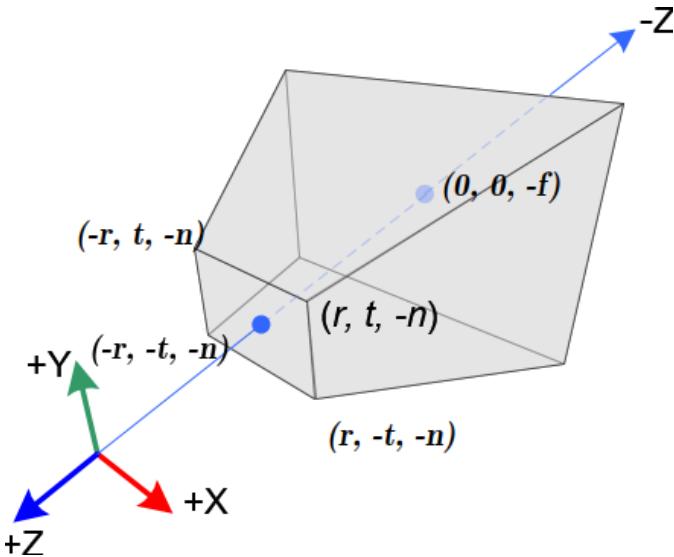


Obrázek 3.7: Ukázka ortografické projekce

#### 3.2.4.4.3.8 Matice perspektivní projekce a *clipovací prostor*

*Perspektivní projekce* definujeme komolým jehlanem, jehož objem zobrazujeme.

Mějme komolý jehlan nadefinovaný čísly  $r, t, n, f$ , takto:



Obrázek 3.8: Komolý jehlan perspektivní projekce<sup>46</sup>

Perspektivní matici definovanou tímto jehlanem můžeme zapsat takto<sup>46</sup>:

$$\begin{pmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & \frac{f+n}{n-f} & \frac{2fn}{n-f} \\ 0 & 0 & -1 & 0 \end{pmatrix} \quad (3.9)$$

Transformace touto maticí bude tedy vypadat následovně:

$$\begin{pmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & \frac{f+n}{n-f} & \frac{2fn}{n-f} \\ 0 & 0 & -1 & 0 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{n}{r} \cdot x \\ \frac{n}{t} \cdot y \\ \frac{f+n}{n-f} \cdot z + \frac{2fn}{n-f} \\ -z \end{pmatrix}$$

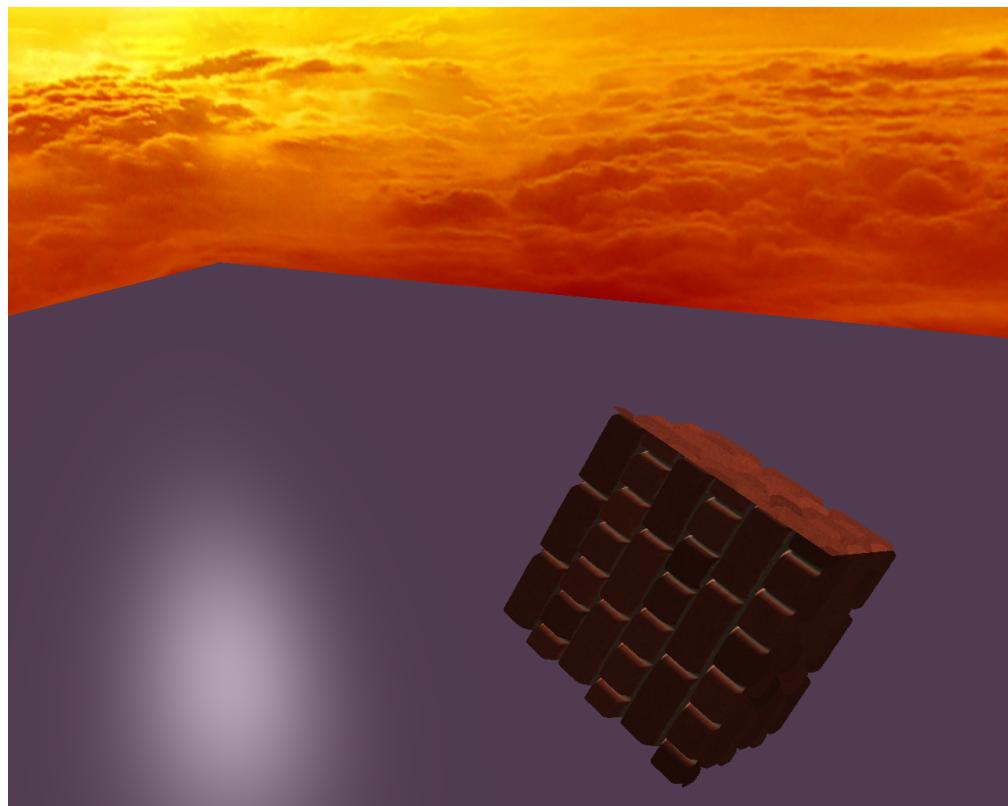
Můžeme si všimnout, že po této transformaci má náš vektor čtvrtý člen, který není roven jedné, jak jsme zvyklý. Je to kvůli tomu, že vektor nebyl transformován rovnou do *normalizovaných souřadnic zařízení*, nýbrž do tzv. *clipovacího prostoru* (*clip space*). Abychom transformovali tento vektor do *normalizovaných souřadnic zařízení* musíme ho vydělit tzv. *perspektivním členem*, což je právě tento čtvrtý člen vektoru<sup>47</sup>

$$\begin{pmatrix} x_{NDC} \\ y_{NDC} \\ z_{NDC} \end{pmatrix} = \begin{pmatrix} \frac{x_c}{w_c} \\ \frac{y_c}{w_c} \\ \frac{z_c}{w_c} \end{pmatrix}$$

Tedy po dosazení získáme:

$$\begin{pmatrix} x_{NDC} \\ y_{NDC} \\ z_{NDC} \end{pmatrix} = \begin{pmatrix} \frac{\frac{n}{r} \cdot x}{-z} \\ \frac{\frac{n}{t} \cdot y}{-z} \\ \frac{\frac{f+n}{n-f} \cdot z + \frac{2fn}{n-f}}{-z} \end{pmatrix}$$

Transformaci z *clipovacího prostoru* do *normalizovaných souřadnic zařízení* za nás *OpenGL* provádí automaticky. Výstup z *vertex shaderu* tedy *OpenGL* očekává pozici vertexu v *clipovacím prostoru*.<sup>47</sup>



Obrázek 3.9: Ukázka perspektivní projekce

### 3.2.4.4.3.9 Transformace normálového vektoru

Normálový vektor nazveme takový vektor, který je kolmý na povrch *meshe* v nějakém bodě.

Normálové vektory se obvykle využívají při výpočtu světelných efektů, a tak je obvykle chceme transformovat z prostoru *meshe* do prostoru *světa*. K této transformaci se u pozičních vektorů používá matici modelu *M*.

Normálový vektor ovšem, narozdíl od pozičního, nereprezentuje pozici, nýbrž směr. Tento vektor je také jednotkovým. Jediný způsob, jak takovýto vektor transformovat je tedy pomocí rotace.

Translace *vertexů* nijak neovlivňuje rotaci normálových vektorů. Pokud je tedy matici modelu *M*, složená jen z rotace a translace, tak si transformační matici normálového vektoru *Q*, můžeme vyjádřit snížením hodnosti matice *M* na tři, čímž odstraníme translaci:

$$\begin{aligned} n' &= Q \cdot n \\ Q &= \text{mat3}(M) \\ v' &= M \cdot v \\ n' &= \text{mat3}(M) \cdot n \end{aligned} \tag{3.10}$$

Kde *n* je sloupcová matice normálového vektoru (dále jen „normála“), *n'* je transformovaná normála. *v* je sloupcová matice pozičního vektoru (dále jen „pozice“) a *v'* je transformovaná pozice.

Jak tomu ovšem bude, pokud k tomu navíc aplikujeme scale? Jedná-li se o jednotný scale, tedy scale je na všech souřadných osách stejný, tak stačí výsledný normálový vektor normalizovat a tím odstraníme tento scale:

$$n' = \frac{\text{mat3}(M) \cdot n}{|\text{mat3}(M) \cdot n|} \tag{3.11}$$

Co když se ale jedná o nejednotný scale? Zatím jsme transformační matici normály *Q* jen odhadovali, nyní si už budeme muset odvodit její obecný vzorec.

Víme, že normálový vektor  $\vec{n}$  je kolmý na povrch trojúhelníku tvořeného pozicičními vektory (*vertexy*)  $\vec{v}_1, \vec{v}_2, \vec{v}_3$ , tím pádem musí být také kolmý na všechny strany tohoto trojúhelníku. Skalární součin  $\vec{n}$  s jakýmkoliv vektorem strany trojúhelníku tedy musí být roven nule:

$$\vec{n} \cdot (\vec{v}_1 - \vec{v}_2) = 0$$

Pokud si vektory vyjádříme jako sloupcové matice, tak si tento skalární součin můžeme přepsat jako<sup>48</sup>

$$n^T \cdot (v_1 - v_2) = 0$$

Tato rovnice musí samozřejmě platit i po transformaci do *prostoru světa*:

$$(Q \cdot n)^T \cdot (M \cdot v_1 - M \cdot v_2) = 0$$

$$n^T \cdot Q^T \cdot M \cdot (v_1 - v_2) = 0$$

Dáme do rovnice:

$$n^T \cdot Q^T \cdot M \cdot (v_1 - v_2) = n^T \cdot (v_1 - v_2)$$

Z této rovnice je očividné, že pokud  $Q^T$  bude rovno inverzní *matici modelu*  $M^{-1}$ , tak se po vynásobení s *maticí modelu* zkrátí v jednotkovou matici a obě strany rovnice se budou rovnat. Transformační matici normály  $Q$  si tedy můžeme pomocí *matici modelu* vyjádřit takto:

$$Q = (M^{-1})^T$$

Výpočet inverzní matice bývá dosti výpočetně náročný, a tak tento výpočet raději provedeme na *CPU* a do *vertex shaderu* tuto matici pošleme v podobě *uniformu*, nebo tuto matici approximujeme podle vzorce (3.10) na *matici modelu* se sníženou hodností na tři a po provedení transformace normálu znormizujeme (3.11).

### 3.2.4.4.3.10 Shrnutí a MVP matice

V této kapitole jsme si nadefinovali *prostor meshe*, v tomto prostoru si *mesh* obvykle modelujeme pomocí nějakého modelovacího softwaru jako je například *Blender*. Počátek souřadnic *prostoru meshe* se obvykle nachází přibližně ve středu *meshe* a vzhledem k tomuto bodu provádíme rotaci, scale a translaci, tyto transformace jsou uloženy v *modelové matici M*.

Po transformaci *modelovou maticí M* se *mesh* dostává do *prostoru světa*. Tento prostor následně transformujeme do *prostoru pohledu* vynásobením *maticí pohledu V*.

Nakonec provedeme transformaci soustavy souřadnic, abychom tím převedli *prostor pohledu* do *clipovacího prostoru*. Tato transformace je provedena *maticí projekce P*.

Všechny tyto transformační fáze můžeme díky asociativnosti maticového násobení vyjádřit jedinou maticí, kterou označíme **MVP**:

$$MVP = P \cdot V \cdot M$$

### 3.2.4.4.3.11 Implementace v BU

*Matici modelu M* můžeme najít jako atribut třídy **ulm::MeshProperties**, která je atributem třídy **ulm::Mesh**.

*Matice pohledu V* a *matice projekce P* můžeme získat pomocí třídy **ulm::Camera**.

Tato třída má atributy:

```
glm::vec3 front;  
glm::vec3 up;  
glm::vec3 position;  
glm::mat4 projection;
```

Vektor *front* směřuje dopředu v pohledu kamery, vektor *up* směřuje nahoru v pohledu kamery, vektor *position* specifikuje pozici kamery v *prostoru světa*.

Matice *projection* reprezentuje matici projekce.

Vytvoření *matice projekce P*, je v této třídě automatizováno metodami *createPerspective* a *createOrthographic*.

*Matici pohledu*  $V$  získáme metodou `getView()`, která *matici pohledu* spočítá na základě vektorů *front*, *up* a *position*.

### 3.2.4.4.4 Blinn-Phongův model světla

Efekt, kterou má světlo na barvu povrchu si rozdělíme na tři části:

#### 3.2.4.4.4.1 Spekulární odraz

Spekulární odraz je vlastně odraz světla od povrchu. Tedy pokud se vektor pohledu na *fragment* bude blížit vektoru světla, odraženého od povrchu, potom bude tato část nejintenzivnější.<sup>49</sup> Blinn-Phongův model tuto část approximuje na<sup>50</sup>

$$b_s = \vec{c}_l \cdot \max\left(\vec{n} \cdot \frac{\vec{l} + \vec{v}}{|\vec{l} + \vec{v}|}, 0\right)^g \cdot s$$

Kde  $\vec{n}$  je normálový vektor k povrchu,  $\vec{l}$  je vektor dopadu světla,  $\vec{v}$  je vektor pohledu,  $\vec{c}_l$  je barva světla a čísla  $g$  a  $s$  jsou parametry materiálu, v  $\text{BU}$  jsou označovány jako *gloss* a *specular*, přičemž *gloss* se ještě násobí číslem 1024.

Vektory dopadu světla a pohledu si ještě můžeme snadno vyjádřit pomocí pozice pozorovatele  $\vec{p}_p$ , pozice fragmentu  $\vec{p}_f$  a pozice světelného zdroje  $\vec{p}_s$  v prostoru světa:

$$\begin{aligned}\vec{l} &= \frac{\vec{p}_s - \vec{p}_f}{|\vec{p}_s - \vec{p}_f|} \\ \vec{v} &= \frac{\vec{p}_p - \vec{p}_f}{|\vec{p}_p - \vec{p}_f|}\end{aligned}$$

#### 3.2.4.4.4.2 Difúzní odraz

Difúzní odraz bychom mohli nazvat jako rozptyl. Víme, že čím kolměji světlo dopadá na povrch, tím více světla se odrazí, protože hustota dopadajícího světla je v takovém případě největší.<sup>51</sup> Blinn-Phongův model tuto část approximuje na<sup>50</sup>:

$$b_d = \vec{c}_l \cdot \vec{c}_m \cdot \max(\vec{n} \cdot \vec{l}, 0)$$

Kde  $\vec{n}$  je normálový vektor k povrchu,  $\vec{l}$  je vektor dopadu světla,  $\vec{c}_l$  je barva světla a  $\vec{c}_m$  je barva materiálu.

#### 3.2.4.4.4.3 Ambientní světlo

Světlo se samozřejmě nechová tak jednodušše, aby se jeho efekt dal popsat výše zmíněnými rovnicemi. Například samotné odražené světlo poté osvětlí další objekty, a tak i povrch zcela odvrácený od světelného zdroje jím může být osvětlen.

Trasovat dráhy paprsků světla by ovšem bylo časově velmi náročné, v reálném čase zcela nemožné, a tak tento efekt velmi hrubě approximujeme konstantním koeficientem  $a$ :

$$b_a = \vec{c}_l \cdot \vec{c}_m \cdot a$$

#### 3.2.4.4.4.4 Výsledný efekt a jeho oslabení

Výsledná barva  $b$  je tedy rovna součtu její výše zmíněných částí, ale musíme ji ještě zmenšit v závislosti na vzdálenosti, což provedeme koeficientem  $k$ :

$$b = (b_a + b_d + b_s) \cdot k$$

Koeficient  $k$  si nadefinujeme pomocí lineárního koeficientu  $L$  a kvadratického  $Q$ , tyto koeficienty jsou vlastností světelného zdroje:

$$k = \frac{1}{(1 + L \cdot d + Q \cdot d^2)}$$

Kde  $d$  je vzdálenost *fragmentu* od zdroje světla.

#### 3.2.4.4.4.5 Implementace v *BU*

Pro reprezentaci světelných zdrojů byly vytvořeny třídy *ulm::DirectionalLight* popisující rovinné světlo, *ulm::PointLight* popisující bodový zdroj světla, *ulm::SpotLight* popisující bodový zdroj koncentrovaného světla.

Materiálové vlastnosti *meshe* ve třídě *ulm::Mesh* jsou nadefinovány buď v atrributech *materials*, které je specifikují pro každý vrchol. Nebo je také možné je nadefinovat pomocí textur, reprezentovanými atributy *albedoMap* a *diffuseMap*, která je specifikuje pro každý fragment.

Materiál se v *BU* skládá z vektoru o čtyřech členech reprezentujícího *difúzní* barvu, kde čtvrtým členem je *alpha*, a vektorem o tří členech, které popořadě reprezentují *gloss*, *spekulární koeficient* a *emisní koeficient*. Emisní koeficient určuje, jako moc má fragment sám od sebe zářit.

### 3.2.4.4.5 Deffered Shading

Můžeme si všimnout, že výpočty světelného efektu jsou velmi časově náročné.

Představme si následující situaci. Máme několik ploch za sebou, takovým způsobem, že je vidět pouze jedna. Světelné výpočty by se potom prováděli pro každý fragment každé plochy, přestože by stačilo, aby se provedli pouze pro tu plochu, která je v popředí.

Tento problém řeší metoda *Deferred shading* tím, že vykreslování rozdělíme do dvou fází:

#### 3.2.4.4.5.1 Geometrická fáze

Zatím nebudeme provádět žádné výpočty světelných efektů a ani nebudeme nic vykreslovat do *výchozího framebufferu*, nýbrž si vytvoříme vlastní. Do něj, místo toho abychom vykreslovali výsledný obraz, budeme „vykreslovat“ informace potřebné pro výpočet světla.

Tento *framebuffer* nazveme *Geometrický buffer* (dále jen „*GBuffer*“), který se bude skládat ze čtyř textur:

První textura o třech kanálech bude reprezentovat difúzní barvu materiálu fragmentu.

Druhá textura o třech kanálech bude reprezentovat světelné vlastnosti materiálu, tedy po pořadě *spekulární koeficient* (červený kanál), *gloss* (zelený kanál) a *emisní koeficient* (modrý kanál).

Třetí textura o třech kanálech bude reprezentovat normálové vektory.

Čtvrtá textura o dvou kanálech bude obsahovat *Depth buffer* a *Stencil buffer*.

Pro vykreslení do *GBufferu* využijeme funkce *OpenGL Multiple Render Targets*<sup>52</sup>.

Fragmentový shader bude vypadat zjednodušeně takto:

```
#ifdef GL_ES
    #version 300 es
#else
    #version 330
#endif
```

```

...
...

layout(location = 0) out vec4 out_diffuse;
layout(location = 1) out vec4 out_albedo;
layout(location = 2) out vec4 out_normal;

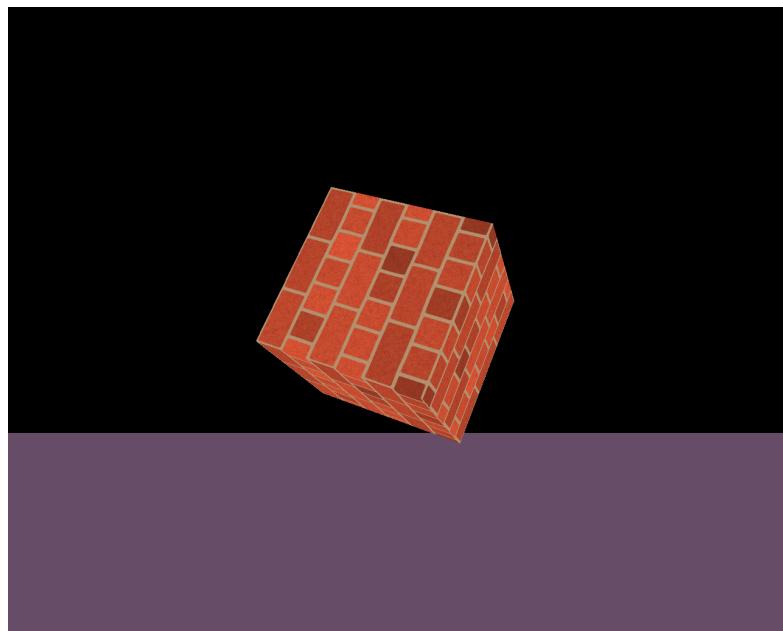
void main(){
    ...
    ...

    out_diffuse = vec4(diffuseColor, 1.0);
    out_albedo = vec4(specular, gloss, emission, 1.0);
    out_normal = vec4(normalVec, 1.0);
}

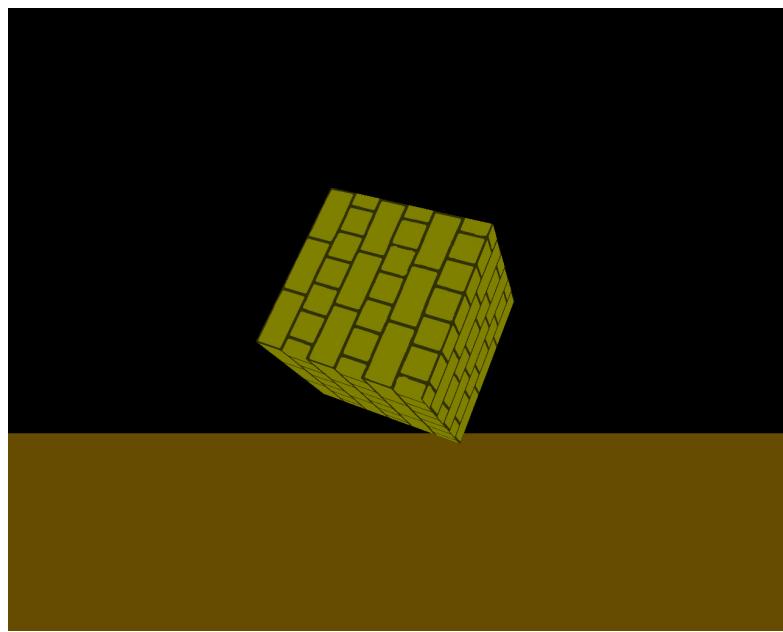
```

Zápis do *Depth bufferu* bude proveden automaticky. Bude proveden také *Depth test*, a tak nám v tomto *Framebufforu* zbydou informace pouze o těch fragmentech, které jsou vidět.

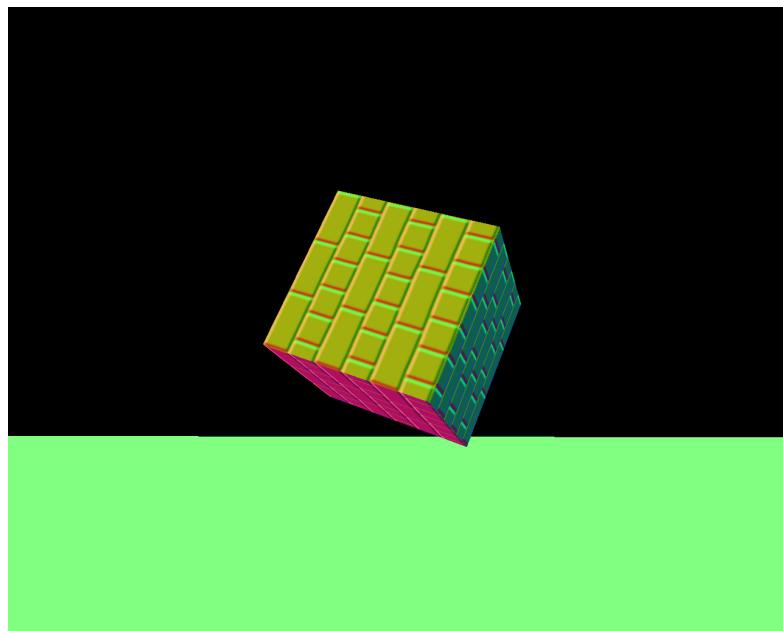
Fragmenty necháme také zapsat do *Stencil bufferu*, což ještě více optimalizuje světelnou fázi tím, že se světelné výpočty stanou jen pro ty fragmenty, které existují. Také to později usnadní vykreslení oblohy.



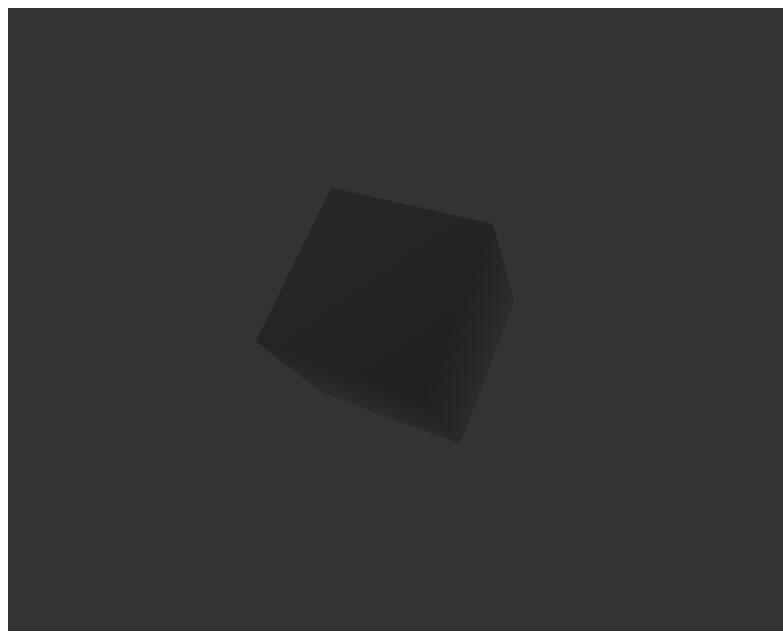
Obrázek 3.10: Difúzní barva v *GBufferu*



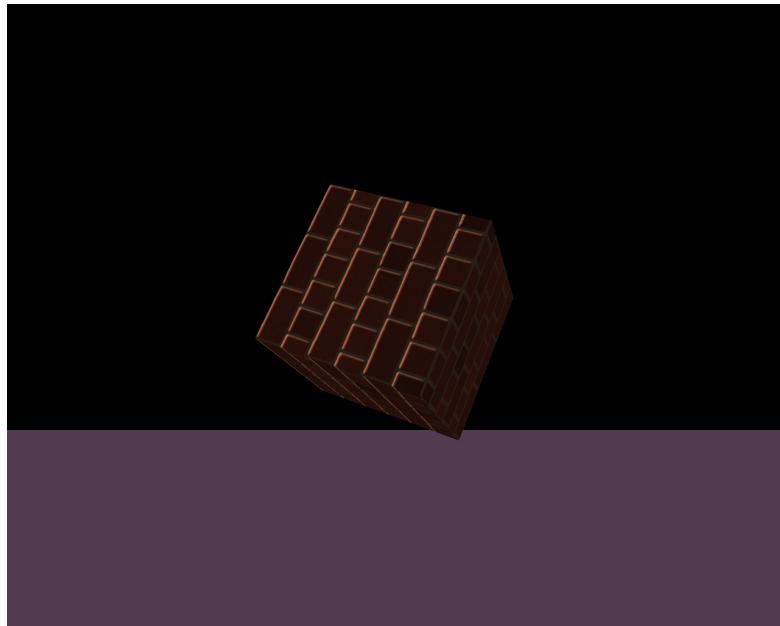
Obrázek 3.11: Koeficienty světelných vlastností v *GBufferu*



Obrázek 3.12: Normálové vektory v *GBufferu*



Obrázek 3.13: Linearizovaný hloubkový buffer v *GBufferu*



Obrázek 3.14: Výsledek světelné fáze provedené na *GBufferu*

#### 3.2.4.4.5.2 Světelná fáze

Zde již budeme vykreslovat výsledný obraz po aplikování světelných efektů. Vytvoříme si další *framebuffer*, nazveme jej *výstupním framebufferem*. Tento *framebuffer* bude s *GBufferem* sdílet *Depth buffer* a *Stencil buffer*, také bude mít navíc texturu, která bude obsahovat výsledný obraz.

Pozici fragmentu v *prostoru světa* můžeme z hodnoty v *Depth bufferu* získat následujícím způsobem:

$$\vec{p} = V^{-1} \cdot \text{persp}(P^{-1} \cdot \begin{pmatrix} t_x \cdot 2 - 1 \\ t_y \cdot 2 - 1 \\ d \cdot 2 - 1 \\ 1 \end{pmatrix})$$

Kde  $\vec{p}$  je pozice světa,  $V$  je matice pohledu,  $P$  matice projekce,  $\vec{t}$  jsou texturové souřadnice a  $d$  je hodnota v *Depth bufferu* a *persp* je *perspektivní dělení*.

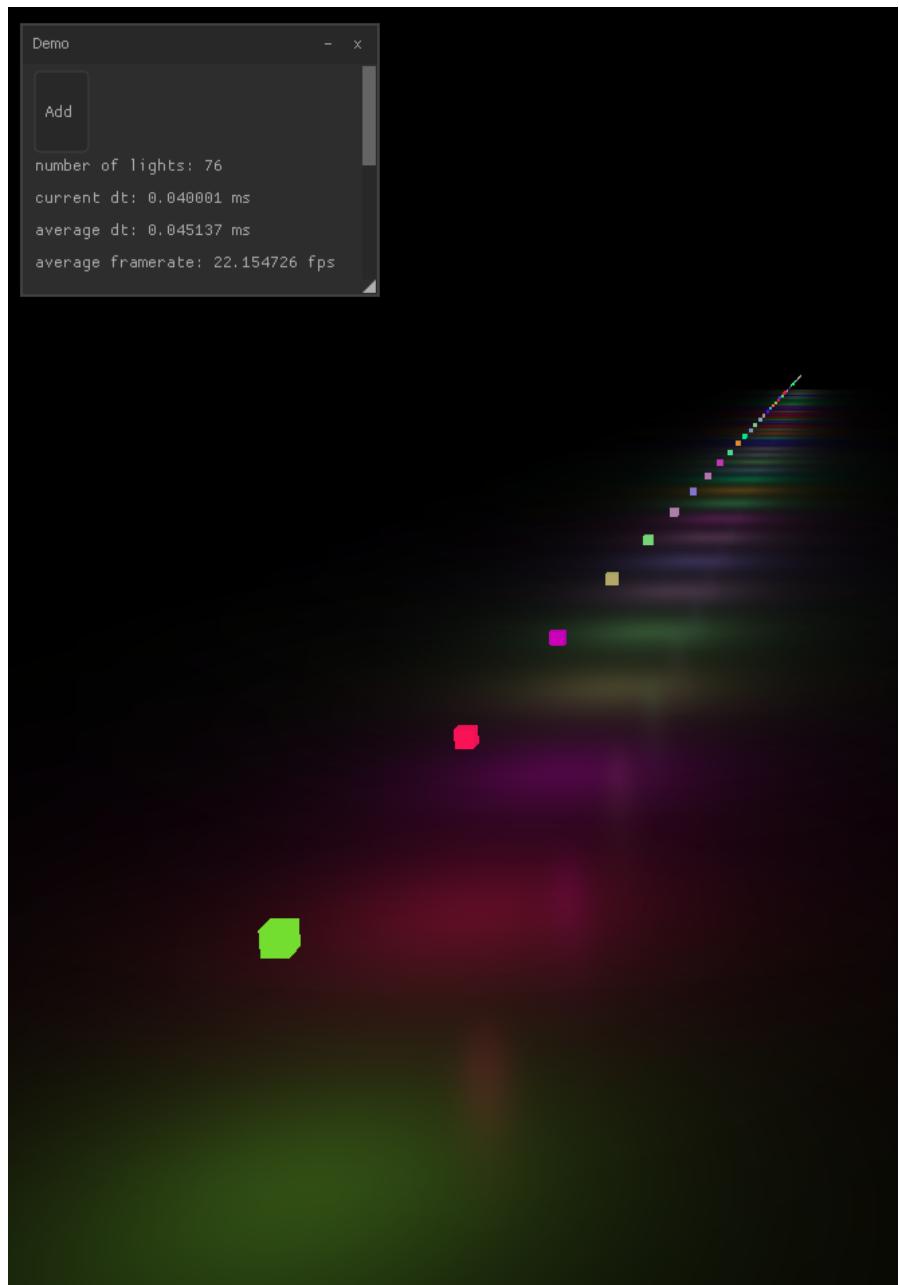
Nyní známe všechny informace potřebné k vykreslení světelných efektů.

Budeme je vykreslovat v rámci čtverců ohraničující oblast, kam dosáhne světlo,

které budou instancované pro každý druh světla. Tyto čtverce budou navíc osekané *Stencil testem* a také tyto světelné efekty budeme skládat aditivní průhledností:



Obrázek 3.15: Ukázka použití aditivní průhlednosti a *Stencil testu* u čtverců vykreslujících světelné efekty.



Obrázek 3.16: Efektivní vykreslení 76 bodových světel pomocí *Deferred shadingu* v *BU*.

#### 3.2.4.4.5.3 Implementace v *BU*

Třída *ulm::GBuffer* se navzdory svému jménu nestará jen o samotný *GBuf-*

*fer*, ale také o *výstupní buffer*. S touto třídou poté pracují vykreslovače jako *ulm::Renderer3D*, *ulm::LightManager*, *ulm::SkyBoxRenderer* a *ulm::BillboardRenderer*.

### 3.2.4.4.6 Typický postup 3D vykreslování v *BU*

Nejprve inicializujeme *GBuffer*, kamenu, sprite pokrývající celou obrazovku, dále nebe, všechny *meshe*, *rovinná světla*, *bodová světla* a *konceントrovaná světla*.

V metodě *render* nejprve *GBuffer* vyčistíme, poté do něj vykreslíme všechny *meshe*, čímž provedeme *geometrickou fázzi*, dále provedeme *světelnou fázzi* do *výstupního framebufferu*. Dále vykreslíme do *výstupního framebufferu* nebe, na všechny pixely do kterých se ještě nevykreslovalo, které se dozvíme pomocí *Stencil testu*. A úplně nakonec vykreslíme do *výstupního framebufferu* všechny „*Aetherické*“ *meshe*, protože ty používají *Depth test*, ale sami do *Depth bufferu* nezapisují.

*Výsledný framebuffer* poté vykreslíme 2D vykreslovačem na obrazovku jako texturu spritu.

```
class Main : public ulm::Program{
public:
    ...
    ulm::GBuffer gbuffer;
    ulm::Camera cam;
    ulm::Sprite fullscreenSprite;
    ulm::SkyBox sky;

    ulm::Array<ulm::Mesh> meshes;
    ulm::Array<ulm::DirectionalLights> dLights;
    ulm::Array<ulm::SpotLights> sLights;
    ulm::Array<ulm::PointLights> pLights;

    float eyeDistance = 0.05f;

    Main() {
        /* Initialization */
        sky.loadFromResources(...);

        cam.createPerspective(...);
```

```

gbuffer . initialize ( ulm :: Window :: screenSize );

fullscreenSprite = ulm :: SpriteFactory :: createRectangle (
    -1.f , -1.f ,
    2.f , 2.f );

...
/* Initialize all lights and meshes */
...
...
}

...
...

void render ( ulm :: Eye eye ) {
    #ifdef BU_VR
        /* Handle VR */
        cam . projection = ulm :: CardboardVR :: getPerspective ();
        cam . up = ulm :: CardboardVR :: getUp ();
        cam . front = ulm :: CardboardVR :: getFront ();
        cam . position += ( float )( eye ) *
            ulm :: CardboardVR :: getRight () *
            eyeDistance ;
    #endif
    gbuffer . clear ();

...
/* Update geometry buffer */
ulm :: Renderer3D :: begin ( cam , & gbuffer );
ulm :: Renderer3D :: submit ( meshes , vlajky );
ulm :: Renderer3D :: draw ( ulm :: CullBackFace );

/* Handle lights , draw to output framebuffer */
ulm :: LightManager :: begin ( cam3D , & gbuffer );
ulm :: LightManager :: submit ( dLights );
ulm :: LightManager :: submit ( pLights );
ulm :: LightManager :: submit ( sLights );
ulm :: LightManager :: draw ();
}

```

```

/* Draw sky */
ulm::SkyBoxRenderer :: drawSkyBox(sky, &gbuffer, cam3D);

/* Draw Aetheric objects, that use additive blending */
ulm::Renderer3D :: drawAether(ulm :: CullNone);

/* Draw the result framebuffer to screen */
fullscreenSprite.texture = gbuffer.getResult();

ulm::Renderer2D :: begin(glm :: mat4(1. f));
ulm::Renderer2D :: draw(fullscreenSprite, ulm :: Textured);
ulm::Renderer2D :: end();
...

#ifndef BU_VR
/* Handle VR */
cam.position -= (float)eye *
    ulm :: CardboardVR :: getRight() *
    eyeDistance;
#endif
}

...
}

```

### **3.2.4.4.7 Vlajky vykreslovače *ulm::Renderer3D***

Vlajky ukládáme do datového typu *16-bitového integeru* (std::uint\_fast16\_t), přičemž každá vlajka je reprezentována *bitem*.

Každá kombinace vlajek má vlastní *shader*, který se od shaderů ostatních kombinací vlajek liší jiným *preprocessorovým definováním makra*, která následně rozhodují, které části kódu *shaderu* budou zkompilovány a které ne. Tento *shader* je zkompilován při prvním použití dané kombinace vlajek.

#### **3.2.4.4.7.1 Barva vertexů - *ulm::Colored***

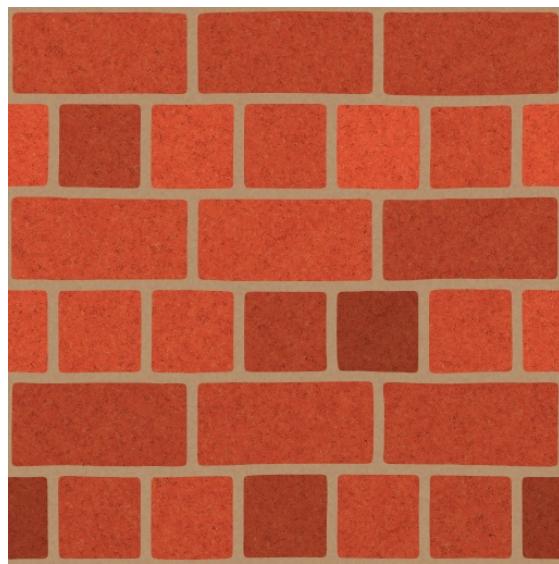
Tato vlajka určuje, zda bude materiál definovaný pomocí *atributů* použit.

#### **3.2.4.4.7.2 Násobení materiálu - *ulm::Multiplied***

Tato vlajka určuje, zda budou vektory *diffuseMultiplier* a *albedoMultiplier* ve třídě *ulm::Mesh* použity k vynásobení materiálů všech fragmentů.

#### **3.2.4.4.7.3 Mapování barvy difúzního odrazu - *ulm::DiffuseMapped***

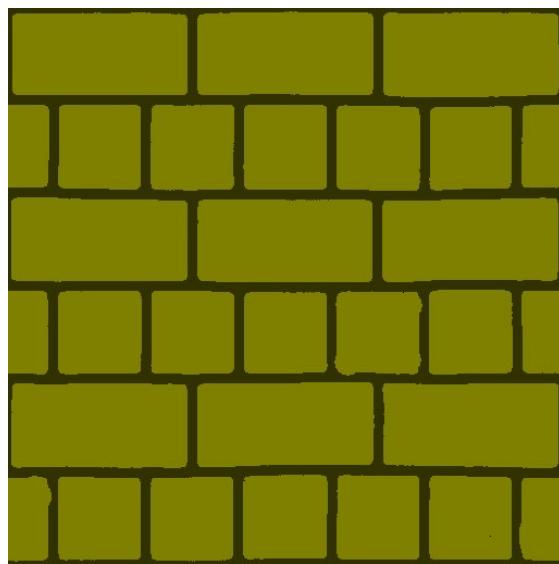
Tato vlajka určuje, zda bude textura *diffuseMap* ve třídě *ulm::Mesh* použita jako barva difúzního odrazu fragmentu.



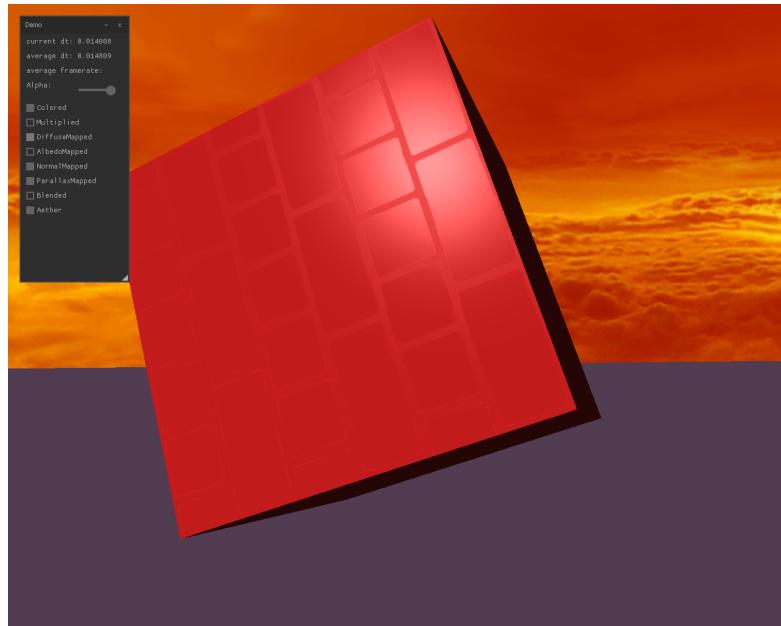
Obrázek 3.17: Ukázka mapy difúzního odrazu<sup>53</sup>

#### 3.2.4.4.7.4 Mapování ostatních světelných vlastností - `ulm::AlbedoMapped`

Tato vlajka určuje, zda bude textura *albedoMap* ve třídě ***ulm::Mesh*** použita pro *spekulární koeficient*, *gloss* a *emission koeficient*.



Obrázek 3.18: Ukázka albedo mapy



Obrázek 3.19: Ukázka vykreslení albedo mapované krychle o homogenní červené barvě pomocí ***ulm::Renderer3D***

#### 3.2.4.4.7.5 Normálové mapování - ***ulm::NormalMapped***

Tato vlajka určuje, zda bude textura *normalMap* ve třídě ***ulm::Mesh*** použita k získání normálového vektoru.

Normálový vektor v této textuře je definovaný v tzv. *tangentovém prostoru*, tento si můžeme definovat pomocí *tangentového vektoru*  $\vec{T}$ , *normálového vektoru*  $\vec{N}$  a *Bitangentového vektoru*  $\vec{B} = \vec{T} \times \vec{N}$ . Všechny tyto vektory se nacházejí v prostoru *meshe*. Transformační matici z *tangentového prostoru* do prostoru *meshe* *TBN* můžeme zapsat podle vzorce (3.6) jako:

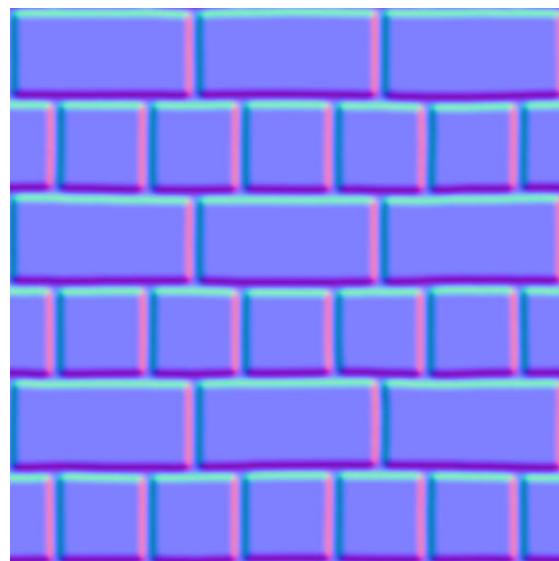
$$TBN = \begin{pmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{pmatrix}$$

Normálový vektor získaný z textury, pak jen vynásobíme touto maticí, čímž ho transformujeme do prostoru *meshe* a dále s ním pokračujeme jako obvykle.

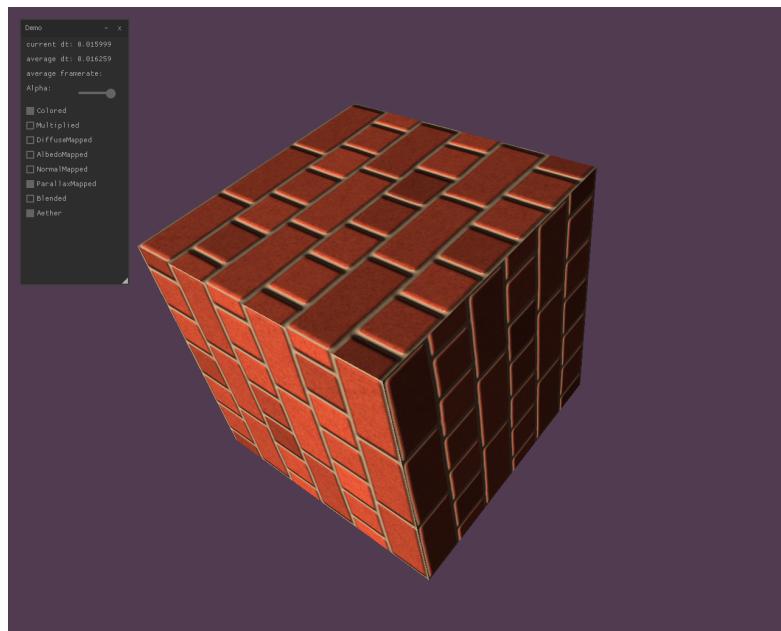
Tangentových vektorů může být nekonečně mnoho, ale konvence nám říká,

že tento vektor má směřovat po *ose x* na textuře normálové mapy. Pomocí této informace se tangentové vektory automaticky vypočítávají při inicializaci *meshe*.  
54

Může se ovšem stát, že vektorové normály byli vyhlazeny, tím pádem vektory  $\vec{T}$ ,  $\vec{B}$  a  $\vec{N}$  by potom již nemuseli tvořit *ortonormální bázi* a nedefinovali by tedy prostor. Pro jistotu bychom ještě tedy měli tyto vektory reortogonalizovat pomocí Gramova–Schmidtovy ortogonalizace.<sup>55</sup>



Obrázek 3.20: Ukázka normálové mapy<sup>56</sup>

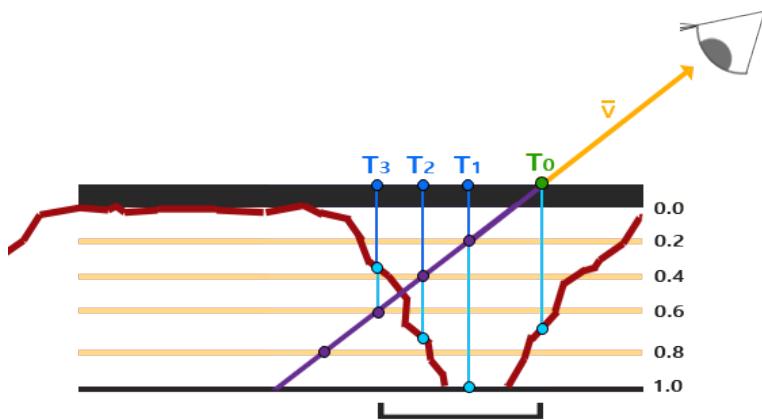


Obrázek 3.21: Ukázka vykreslení normálově mapované krychle pomocí *ulm::Renderer3D*

#### 3.2.4.4.7.6 Parallaxové mapování - *ulm::ParallaxMapped*

Parallaxové mapování upravuje texturové souřadnice na základě hloubkové textury.<sup>56</sup>

V *BU* je parallaxové mapování implementováno pomocí algoritmu *Steep Parallax Mapping*. Která spočívá v krokovacím posouvání (po vrstvách) po vektoru pohledu na fragment a následném odhadování průsečíku tohoto vektoru a výškové mapy.



Obrázek 3.22: Steep parallax mapping<sup>56</sup>

Provedení tohoto algoritmu je ovlivněno několika atributy třídy ***ulm::Mesh***:

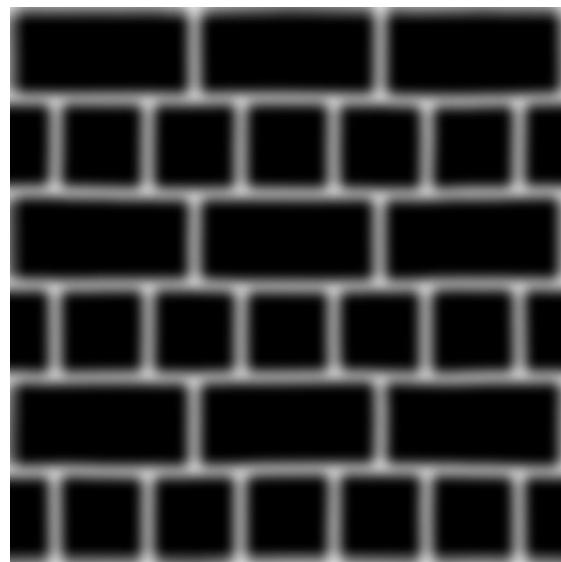
```
unsigned int parallax_minLayers = 4;
unsigned int parallax_maxLayers = 16;
float parallax_heightScale = 0.05;
bool parallax_repeatTexture = false;
```

Atribut *parallax\_minLayers* určuje minimální počet vrstev (pokud se díváme kolmo na povrch), kterými krokujeme,

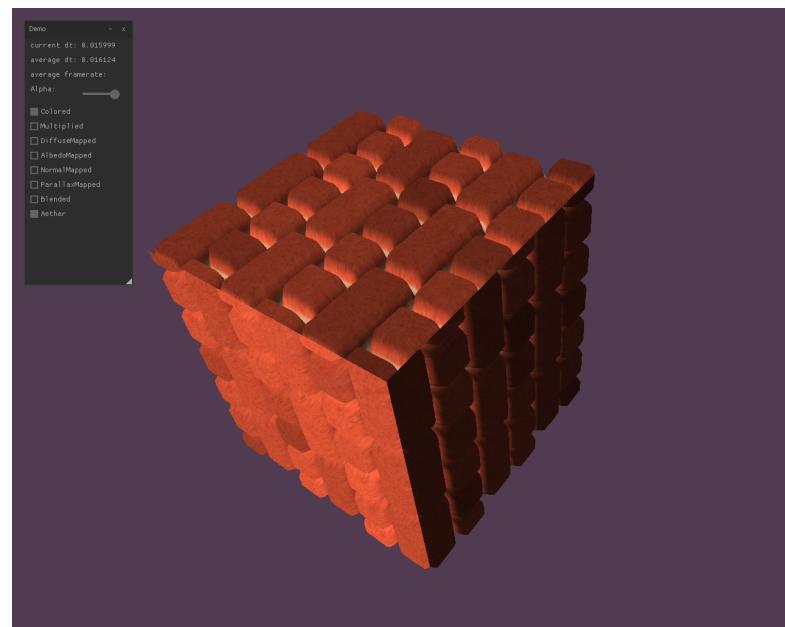
*parallax\_maxLayers* určuje maximální počet vrstev (pokud se díváme rovnoběžně na povrch), kterými krokujeme,

*parallax\_heightScale* určuje maximální hloubku, tedy hloubku, která odpovídá černé barvě ve výškové mapě,

*parallax\_repeatTexture* určuje zda, pokud jsme se dostali s texturovými souřadnicemi mimo texturu, bude textura opakována nebo bude *fragment* odstraněn.



Obrázek 3.23: Ukázka hloubkové mapy<sup>56</sup>



Obrázek 3.24: Ukázka vykreslení parallaxově mapované krychle pomocí *ulm::Renderer3D*

### 3.2.4.4.7.7 Kosterní animace - *ulm::Animation*

Kosterní animace jsou definovány pomocí heirarchie kostí, jejich translací a rotací. Přičemž tyto translace a rotace jsou vždy vůči svému předkovi.

Jednotlivé *vertexy* mají specifikováno, které kosti a jak moc je ovlivňují. To je definováno v atributech *joint\_indices* a *joint\_weights* ve třídě *ulm::Mesh*, tyto hodnoty se při inicializaci přenesou do *VBO*.

Než budeme transformovat *vertexy* pomocí kostí, tak musíme nejdříve vypočítat transformační matici dané kosti. Víme, že *rotace* a *translace* kosti je definována vzhledem k jejímu předkovi. K tomu, abychom získali výslednou transformaci, tak musíme nejprve transformovat svět, tak aby počátek (souřadnic a tedy i rotace) byl v kosti předka. Následně provedeme transformaci definovanou translací a rotací této kosti. Nyní se ale nacházíme v prostoru, který byl postupně posouván podle pozice kostí. Musíme tedy tyto translace znova odstranit.

Všechny kosti se nacházejí v nějakém výchozím stavu, který obvykle opisuje *mesh*, který ovlivňuje. Což je užitečné pro snadnější výpočet vah kostí k *vertexům* a také pro snadnější představivost. V této výchozí pozici chceme, aby nedošlo k žádné transformaci *vertexů*, a tak si například při načtení *kostry*, uložíme inverzní matice všech kostí. Touto inverzní výchozí maticí musíme transformovat novou matici kosti.<sup>57</sup>

Výslednou matici kosti můžeme tedy zapsat jako:

$$K_i = \left[ \prod_{p \in P_i} (T_p \cdot R_p) \cdot T_i \cdot R_i \right] \cdot \left[ T \left( - \sum_{p \in P_i} (\vec{T}_p) - \vec{T}_i \right) \right] \cdot D_i^{-1}$$

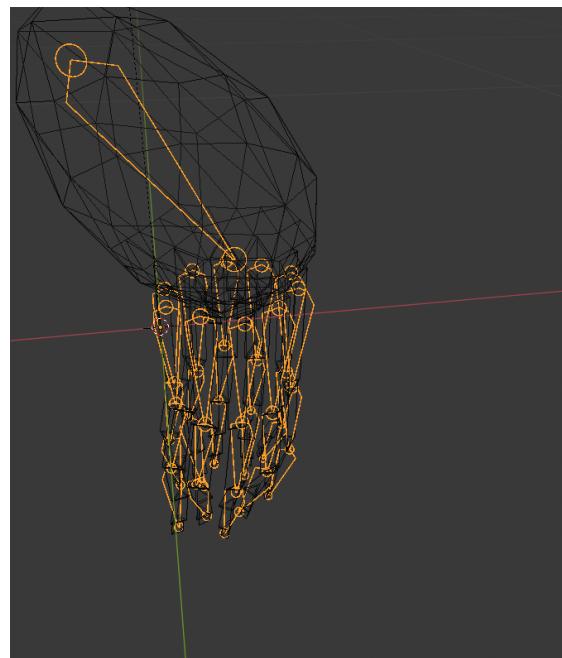
Kde  $K_i$  je transformační matice kosti s indexem  $i$ ,

$P_i$  je množinou všech předků kosti  $i$ , seřazenou od kořene k přímému předkovi,

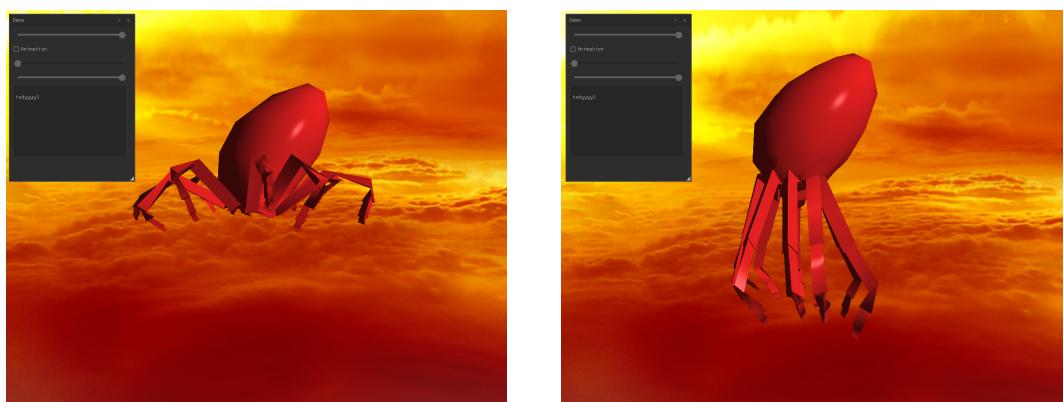
$T_x$ ,  $R_x$  jsou translační a rotační matice kosti  $x$ ,

$\vec{T}_x$ , je translační vektor kosti  $x$  a

$D_i$  je výchozí transformační matice kosti  $i$ .



Obrázek 3.25: Ukázka kostry, ve výchozí pozici, meshe chobotnice. (*Blender*)



Obrázek 3.26: Chobotnice vykreslena podle dvou různých pozic její kostry pomocí **ulm::Renderer3D**

#### 3.2.4.4.7.8 Vynucení přesnosti transformace normálových vektorů - **ulm::ForcePrecision**

Při instancování nebo při kosterních animacích se pro snížení paměťové a časové

složitosti aproxiimuje inverzní transponovaná matice modelu za matici modelu se sníženou hodnotí na tři. To může vést ke špatné transformaci normálových vektorů při nejednotném scalu.

Tato vlajka vynutí použití inverzní transponované matice.

#### 3.2.4.4.7.9 Instancování - *ulm::Instanced*

Pro omezení množství *draw callů* používáme tzv. *instancování*.

Při *draw callu* specifikujeme počet instancí *meshe*, které se mají vykreslit. Také vytvoříme instancovaný *VBO*, který bude obsahovat *matice modelu* jednotlivých instancí, o to se stará třída *ulm::ModelMatrixBuffer*.<sup>58 59</sup>

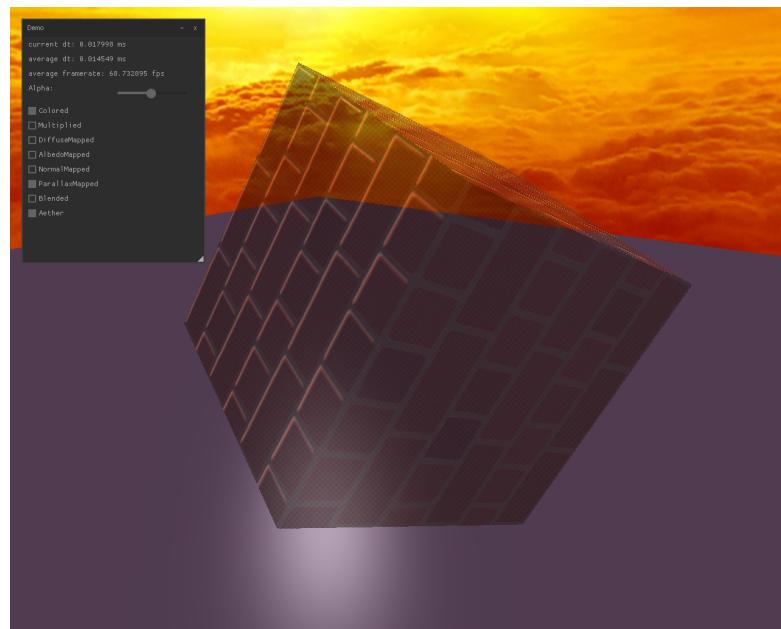
Je-li zapnuto také kostení animování, tak jsou ještě do *shaderů* poslány matice všech kostí všech instancí jako *UBO*, o to se stará třída *ulm::SkeletonBuffer*.

#### 3.2.4.4.7.10 Screen door průhlednost - *ulm::Blended*

Tato vlajka zapíná *screen door průhlednost*, která spočívá v tom, že na základě hodnoty *alpha* některé fragmenty odstraníme.<sup>60</sup>

Ve fragment shaderu tato operace funguje následovně:

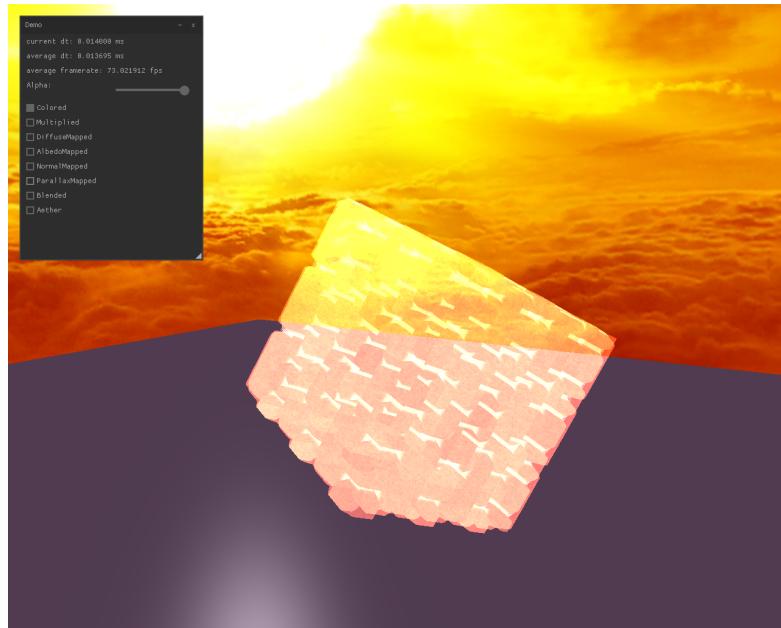
```
mat4 threshold = mat4(
    1.0 / 17.0, 9.0 / 17.0, 3.0 / 17.0, 11.0 / 17.0,
    13.0 / 17.0, 5.0 / 17.0, 15.0 / 17.0, 7.0 / 17.0,
    4.0 / 17.0, 12.0 / 17.0, 2.0 / 17.0, 10.0 / 17.0,
    16.0 / 17.0, 8.0 / 17.0, 14.0 / 17.0, 6.0 / 17.0
);
...
int x = int(gl_FragCoord.x);
int y = int(gl_FragCoord.y);
if (color.w < threshold[x % 4][y % 4])
    discard;
```



Obrázek 3.27: Ukázka vykreslení Screen door průhledné krychle pomocí **ulm::Renderer3D**

#### 3.2.4.4.7.11 Aditivní průhlednost - ulm::Aether

*Aditivní průhlednost* narozený od *alpha průhlednosti* je asociativní. Také dodává objektů zářící efekt. Spočívá v tom, že pokud se nějaké dva fragmenty překrývají, tak se jejich barvy sečtou, místo toho aby se navzájem nahrazovaly.<sup>61</sup>



Obrázek 3.28: Ukázka vykreslení *Aetherické* průhledné krychle pomocí ***ulm::Renderer3D***

#### 3.2.4.4.8 Face culling

Je zcela zbytečné vykreslovat odvrácené trojúhelníky, protože ty u všech uzavřených *meshů* budou překryté jinými trojúhelníky. Pokud tedy uspořádáme *vertexy takovým způsobem*, aby při pohledu na ně normálovým vektorem byli v protisměru hodinových ručiček, tak potom vždy když se na ně podíváme z druhé strany, tak budou *vertexy* uspořádány ve směru hodinových ručiček. Takovýmto způsobem může *OpenGL* poznat, zda se je trojúhelník odvrácený či přivrácený.<sup>62 63</sup>

K uspořádání *vertexů* tímto způsobem je možné využít funkci *prepareFaceCulling()* nacházející se ve třídě ***ulm::Mesh***. Obvykle ale modelovací softwary jako *Blender* automaticky takto *vertexy* seřazují při exportu.

Touto metodou můžeme odstranit více než polovinu během *fragment shaderu* v *geometrické fázi*, které jsou zcela zbytečné.

Typ *Face Cullingu* je možné specifikovat při vykreslení pomocí ***ulm::Renderer3D***.

### 3.2.4.4.9 Úroveň detailu (LOD)

Nachází-li se *mesh* daleko, tak ho nepotřebujeme vykreslovat v plné kvalitě, čímž bychom mohli snížit časovou složitost jeho vykreslení. Zároveň pro zachování paměťové složitosti definujeme tyto úrovně pomocí úseků v *IBO*, které tento *mesh* budou definovat pomocí menšího počtu jeho *vertexů*.

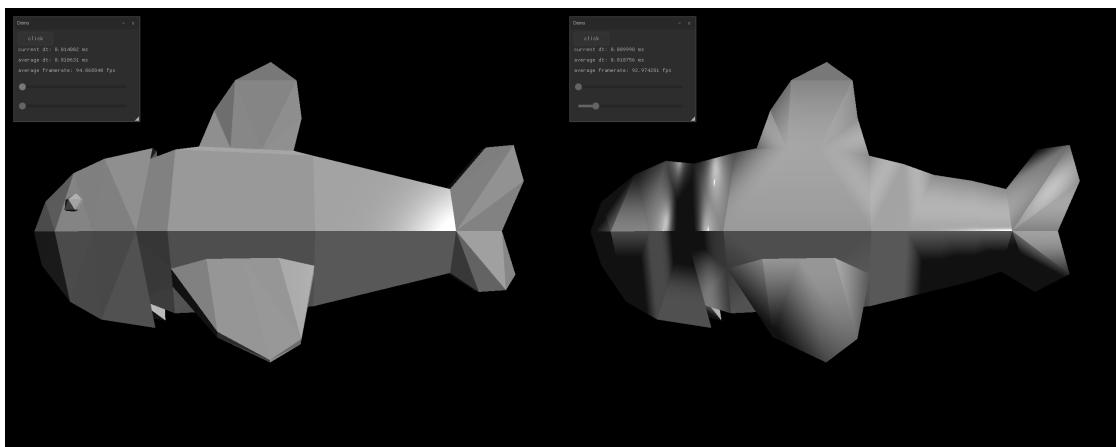
Pozice a velikosti těchto úseků je možné nalézt jako atribut třídy ***ulm::Mesh***:

```
ulm :: Array<glm :: ivec2> LOD_locations ;
```

#### 3.2.4.4.9.1 Half-edge collapse algoritmus

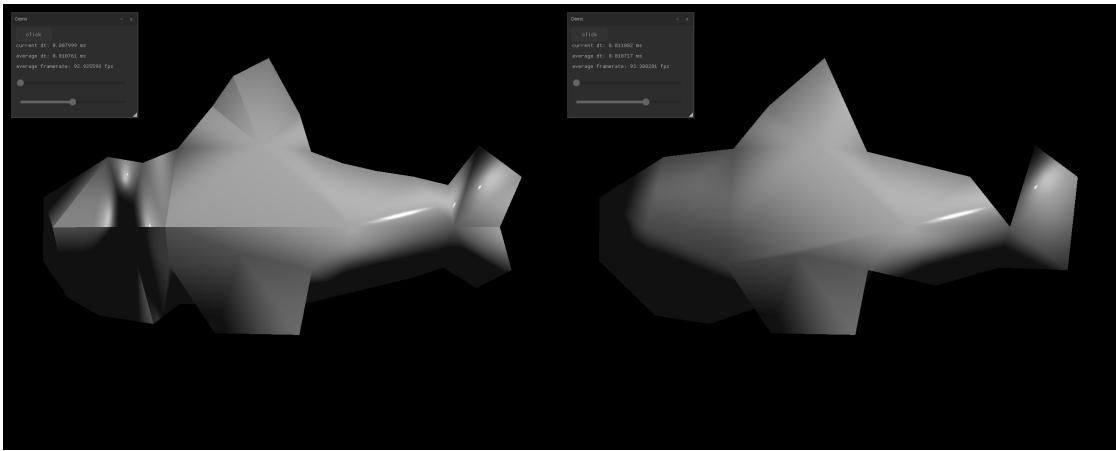
Jedním algoritmem, jak tyto úrovně detailu získat je například *Half-edge collapse*, který spočívá v tom, že se hledají krátké hrany *meshe* a poté je jeden vrchol této hrany nahrazen vrcholem druhým.<sup>64</sup>

Tento algoritmus je implementován v metodě *halfEdgeCollapse(int numberOfLevels)* ve třídě ***ulm::Mesh***.



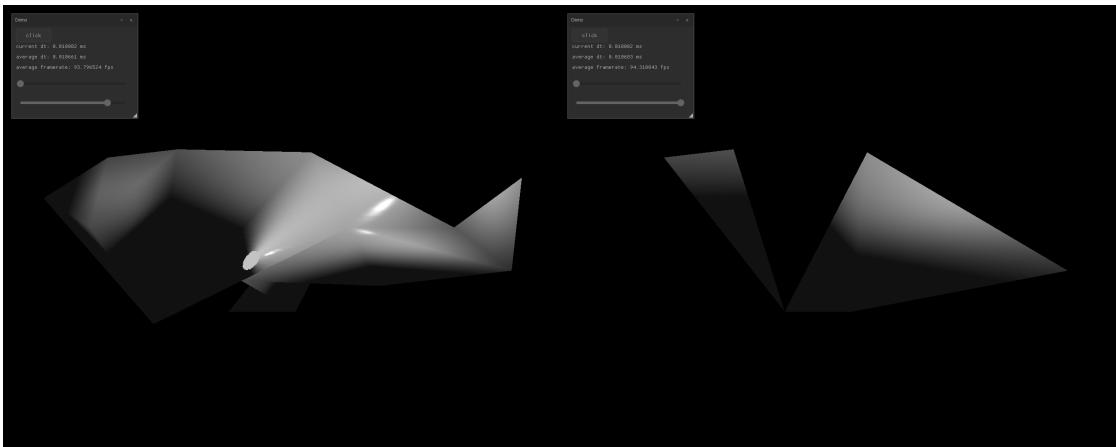
LOD 0/6 meshe ryby

LOD 1/6 meshe ryby



LOD 3/6 meshe ryby

LOD 4/6 meshe ryby



LOD 5/6 meshe ryby

LOD 6/6 meshe ryby

Obrázek 3.31: Různé úrovně detailu meshe ryby vykreslené pomocí *ulm::Renderer3D*

## 4. Závěr

V rámci tohoto projektu bylo vytvořeno vývojové prostředí (*Integrated Development Environment*) *IDEal* a knihovna *Bibliotekum\_Ultimatum*, pro usnadnění vývoje multiplatformních grafických aplikací v *C++*.

# Seznam literatury

<sup>1</sup>OpenGL ES | Android Developers. Android Developers [online]. Dostupné z: <https://developer.android.com/guide/topics/graphics/opengl>

<sup>2</sup>AAPT2 | Android Developers. Android Developers [online]. Dostupné z: <https://developer.android.com/studio/command-line/aapt2>

<sup>3</sup>javac. [online]. Copyright © [cit. 13.02.2020]. Dostupné z: <https://docs.oracle.com/javase/10/tools/javac.htm#JSW0R627>

<sup>4</sup>Dalvik bytecode | Android Open Source Project. Android Open Source Project [online]. Dostupné z: <https://source.android.com/devices/tech/dalvik/dalvik-bytecode>

<sup>5</sup>Android keystore system | Android Developers. Android Developers [online]. Dostupné z: <https://developer.android.com/training/articles/keystore>

<sup>6</sup>zipalign | Android Developers. Android Developers [online]. Dostupné z: <https://developer.android.com/studio/command-line/zipalign>

<sup>7</sup>Simple DirectMedia Layer - Homepage. Simple DirectMedia Layer - Homepage [online]. Dostupné z: <https://www.libsdl.org/>

<sup>8</sup>Quickstart for Google VR SDK for Android | Google Developers. Google Developers [online]. Dostupné z: <https://developers.google.com/vr/develop/android/get-started>

<sup>9</sup>GitHub - vurtun/nuklear: A single-header ANSI C gui library. The world's leading software development platform · GitHub [online]. Copyright © 2020 GitHub, Inc. [cit. 13.02.2020]. Dostupné z: <https://github.com/vurtun/nuklear>

<sup>10</sup>GitHub - nothings/stb: stb single-file public domain libraries for C/C++. The world's leading software development platform · GitHub [online]. Copyright © 2020 GitHub, Inc. [cit. 13.02.2020]. Dostupné z: <https://github.com/nothings/stb>

<sup>11</sup>OpenGL Mathematics. GLM [online]. Dostupné z: <https://glm.g-truc.net/0.9.9/index.html>

<sup>12</sup>OpenGL – Wikipedie. [online]. Dostupné z: <https://cs.wikipedia.org/wiki/OpenGL>

<sup>13</sup>Rendering Pipeline Overview - OpenGL Wiki. The Khronos Group Inc [online]. Dostupné z: [https://www.khronos.org/opengl/wiki/Rendering\\_Pipeline\\_Overview](https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview)

<sup>14</sup>glVertexAttribDivisor - OpenGL 4 Reference Pages. The Khronos Group Inc [online]. Copyright © 2010 [cit. 11.02.2020]. Dostupné z: <https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glVertexAttribDivisor.xhtml>

<sup>15</sup>Vertex Specification - OpenGL Wiki. The Khronos Group Inc [online]. Dostupné z: [https://www.khronos.org/opengl/wiki/Vertex\\_Specification](https://www.khronos.org/opengl/wiki/Vertex_Specification)

<sup>16</sup>Vertex Specification Best Practices - OpenGL Wiki. The Khronos Group Inc [online]. Dostupné z: [https://www.khronos.org/opengl/wiki/Vertex\\_Specification\\_Best\\_Practices](https://www.khronos.org/opengl/wiki/Vertex_Specification_Best_Practices)

<sup>17</sup>OpenGL Shading Language - Wikipedia. [online]. Dostupné z: [https://en.wikipedia.org/wiki/OpenGL\\_Shading\\_Language](https://en.wikipedia.org/wiki/OpenGL_Shading_Language)

<sup>18</sup>Vertex Shader - OpenGL Wiki. The Khronos Group Inc [online]. Dostupné z: [https://www.khronos.org/opengl/wiki/Vertex\\_Shader](https://www.khronos.org/opengl/wiki/Vertex_Shader)

<sup>19</sup>Geometry Shader - OpenGL Wiki. The Khronos Group Inc [online]. Dostupné z: [https://www.khronos.org/opengl/wiki/Geometry\\_Shader](https://www.khronos.org/opengl/wiki/Geometry_Shader)

<sup>20</sup>Vertex Post-Processing - OpenGL Wiki. The Khronos Group Inc [online]. Dostupné z: [https://www.khronos.org/opengl/wiki/Vertex\\_Post-Processing](https://www.khronos.org/opengl/wiki/Vertex_Post-Processing)

<sup>21</sup>Primitive Assembly - OpenGL Wiki. The Khronos Group Inc [online]. Dostupné z: [https://www.khronos.org/opengl/wiki/Primitive\\_Assembly](https://www.khronos.org/opengl/wiki/Primitive_Assembly)

<sup>22</sup>Rasterization - OpenGL Wiki. The Khronos Group Inc [online]. Dostupné z: <https://www.khronos.org/opengl/wiki/Rasterization>

<sup>23</sup>Fragment Shader - OpenGL Wiki. The Khronos Group Inc [online]. Dostupné z: [https://www.khronos.org/opengl/wiki/Fragment\\_Shader](https://www.khronos.org/opengl/wiki/Fragment_Shader)

<sup>24</sup>LearnOpenGL - Shaders. Learn OpenGL, extensive tutorial resource for learning Modern OpenGL [online]. Dostupné z: <https://learnopengl.com/Getting-started/Shaders>

<sup>25</sup>Per-Sample Processing - OpenGL Wiki. The Khronos Group Inc [online]. Dostupné z: [https://www.khronos.org/opengl/wiki/Per-Sample\\_Processing](https://www.khronos.org/opengl/wiki/Per-Sample_Processing)

<sup>26</sup>Depth Test - OpenGL Wiki. The Khronos Group Inc [online]. Dostupné z: [https://www.khronos.org/opengl/wiki/Depth\\_Test](https://www.khronos.org/opengl/wiki/Depth_Test)

<sup>27</sup>LearnOpenGL - Stencil testing. Learn OpenGL, extensive tutorial resource for learning Modern OpenGL [online]. Dostupné z: <https://learnopengl.com/Advanced-OpenGL/Stencil-testing>

<sup>28</sup>Uniform Buffers VS Texture Buffers - Rastergrid. Daniel Rákos [online]. Dostupné z: <http://rastergrid.com/blog/2010/01/uniform-buffers-vs-texture-buffers/>

<sup>29</sup>Uniform Buffer Object - OpenGL Wiki. The Khronos Group Inc [online]. Dostupné z: [https://www.khronos.org/opengl/wiki/Uniform\\_Buffer\\_Object](https://www.khronos.org/opengl/wiki/Uniform_Buffer_Object)

<sup>30</sup>Texture - OpenGL Wiki. The Khronos Group Inc [online]. Dostupné z: <https://www.khronos.org/opengl/wiki/Texture>

<sup>31</sup>Cubemap Texture - OpenGL Wiki. The Khronos Group Inc [online]. Dostupné z: [https://www.khronos.org/opengl/wiki/Cubemap\\_Texture](https://www.khronos.org/opengl/wiki/Cubemap_Texture)

<sup>32</sup>Polygonová síť – Wikipédie. [online]. Dostupné z: [https://cs.wikipedia.org/wiki/Polygonov%C3%A1\\_s%C3%AD](https://cs.wikipedia.org/wiki/Polygonov%C3%A1_s%C3%AD)

<sup>33</sup>Transformation matrix - Wikipedia. [online]. Dostupné z: [https://en.wikipedia.org/wiki/Transformation\\_matrix](https://en.wikipedia.org/wiki/Transformation_matrix)

<sup>34</sup>LearnOpenGL - Transformations. Learn OpenGL, extensive tutorial resource for learning Modern OpenGL [online]. Dostupné z: <https://learnopengl.com/Getting-started/Transformations>

<sup>35</sup>Why Transformation Order Is Significant - Windows Forms | Microsoft Docs. [online]. Dostupné z: <https://docs.microsoft.com/en-us/dotnet/framework/winforms/advanced/why-transformation-order-is-significant>

<sup>36</sup>Translation (geometry) - Wikipedia. [online]. Dostupné z: [https://en.wikipedia.org/wiki/Translation\\_\(geometry\)](https://en.wikipedia.org/wiki/Translation_(geometry))

<sup>37</sup>Understanding the View Matrix - 3D Game Engine Programming. - 3D Game Engine Programming [online]. Dostupné z: <https://www.3dgep.com/understanding-the-view-matrix/>

<sup>38</sup>Transformation matrix stretching - Wikipedia. [online]. Dostupné z: [https://en.wikipedia.org/wiki/Transformation\\_matrix#Stretching](https://en.wikipedia.org/wiki/Transformation_matrix#Stretching)

<sup>39</sup>Rotation matrix - Wikipedia. [online]. Dostupné z: [https://en.wikipedia.org/wiki/Rotation\\_matrix](https://en.wikipedia.org/wiki/Rotation_matrix)

<sup>40</sup>Ortonormální báze – Wikipedie. [online]. Dostupné z: [https://cs.wikipedia.org/wiki/Ortonorm%C3%A1ln%C3%AD\\_b%C3%A1ze](https://cs.wikipedia.org/wiki/Ortonorm%C3%A1ln%C3%AD_b%C3%A1ze)

<sup>41</sup>Orthogonal Vectors and Matrices [online]. Copyright © [cit. 10.02.2020]. Dostupné z: <http://www.math.kent.edu/~reichel/courses/intr.num.comp.1/fall11/lecture5/lecture3.pdf>

<sup>42</sup>Orthogonal matrix - Wikipedia. [online]. Dostupné z: [https://en.wikipedia.org/wiki/Orthogonal\\_matrix](https://en.wikipedia.org/wiki/Orthogonal_matrix)

<sup>43</sup>3D projection - Wikipedia. [online]. Dostupné z: [https://en.wikipedia.org/wiki/3D\\_projection](https://en.wikipedia.org/wiki/3D_projection)

<sup>44</sup>LearnOpenGL - Coordinate Systems. Learn OpenGL, extensive tutorial resource for learning Modern OpenGL [online]. Dostupné z: <https://learnopengl.com/Getting-started/Coordinate-Systems>

<sup>45</sup>Orthographic projection - Wikipedia. [online]. Dostupné z: [https://en.wikipedia.org/wiki/Orthographic\\_projection](https://en.wikipedia.org/wiki/Orthographic_projection)

<sup>46</sup>OpenGL Projection Matrix. Song Ho Ahn [online]. Copyright © [cit. 10.02.2020]. Dostupné z: [http://www.songho.ca/opengl/gl\\_projectionmatrix.html](http://www.songho.ca/opengl/gl_projectionmatrix.html)

<sup>47</sup>Vertex Post-Processing - OpenGL Wiki. The Khronos Group Inc [online]. Dostupné z: [https://www.khronos.org/opengl/wiki/Vertex\\_Post-Processing](https://www.khronos.org/opengl/wiki/Vertex_Post-Processing)

<sup>48</sup>Matrix multiplication - Wikipedia. [online]. Dostupné z: [https://en.wikipedia.org/wiki/Matrix\\_multiplication](https://en.wikipedia.org/wiki/Matrix_multiplication)

<sup>49</sup>Specular reflection - Wikipedia. [online]. Dostupné z: [https://en.wikipedia.org/wiki/Specular\\_reflection](https://en.wikipedia.org/wiki/Specular_reflection)

<sup>50</sup>Blinn–Phong reflection model - Wikipedia. [online]. Dostupné z: [https://en.wikipedia.org/wiki/Blinn%E2%80%93Phong\\_reflection\\_model](https://en.wikipedia.org/wiki/Blinn%E2%80%93Phong_reflection_model)

<sup>51</sup>Diffuse reflection - Wikipedia. [online]. Dostupné z: [https://en.wikipedia.org/wiki/Diffuse\\_reflection](https://en.wikipedia.org/wiki/Diffuse_reflection)

<sup>52</sup>Multiple Render Targets - Wikipedia. [online]. Dostupné z: [https://en.wikipedia.org/wiki/Multiple\\_Render\\_Targets](https://en.wikipedia.org/wiki/Multiple_Render_Targets)

<sup>53</sup>3DEngineCpp/res/textures at master · BennyQBD/3DEngineCpp · GitHub. The world's leading software development platform · GitHub [online]. Copyright © 2020 GitHub, Inc. [cit. 12.02.2020]. Dostupné z: <https://github.com/BennyQBD/3DEngineCpp/tree/master/res/textures>

<sup>54</sup>LearnOpenGL - Normal Mapping. Learn OpenGL, extensive tutorial resource for learning Modern OpenGL [online]. Dostupné z: <https://learnopengl.com/Advanced-Lighting/Normal-Mapping>

<sup>55</sup>Gramova–Schmidtova ortogonalizace – Wikipedie. [online]. Dostupné z: [https://cs.wikipedia.org/wiki/Gramova-%C4%80%C5%99%C5%A1midtova\\_ortogonalizace](https://cs.wikipedia.org/wiki/Gramova-%C4%80%C5%99%C5%A1midtova_ortogonalizace)

<sup>56</sup>LearnOpenGL - Parallax Mapping. Learn OpenGL, extensive tutorial resource for learning Modern OpenGL [online]. Dostupné z: <https://learnopengl.com/Advanced-Lighting/Parallax-Mapping>

<sup>57</sup> Tutorial 38 - Skeletal Animation With Assimp . OpenGL Step by Step - OpenGL Development [online]. Dostupné z: <http://ogldev.atspace.co.uk/www/tutorial38/tutorial38.html>

<sup>58</sup>Instancing. The Khronos Group Inc [online]. Dostupné z: [https://www.khronos.org/opengl/wiki/Vertex\\_Rendering#Instancing](https://www.khronos.org/opengl/wiki/Vertex_Rendering#Instancing)

<sup>59</sup>LearnOpenGL - Instancing. Learn OpenGL, extensive tutorial resource for learning Modern OpenGL [online]. Dostupné z: <https://learnopengl.com/Advanced-OpenGL/Instancing>

<sup>60</sup>Unity Stipple Transparency Shader - Alex Ocius Blog. Alexander Ocius [online]. Dostupné z: <https://ocius.com/blog/unity-stipple-transparency-shader/>

<sup>61</sup>LearnOpenGL - Blending. Learn OpenGL, extensive tutorial resource for learning Modern OpenGL [online]. Dostupné z: <https://learnopengl.com/Advanced-OpenGL/Blending>

<sup>62</sup>Face Culling - OpenGL Wiki. The Khronos Group Inc [online]. Dostupné z: [https://www.khronos.org/opengl/wiki/Face\\_Culling](https://www.khronos.org/opengl/wiki/Face_Culling)

<sup>63</sup>LearnOpenGL - Face culling. Learn OpenGL, extensive tutorial resource for learning Modern OpenGL [online]. Dostupné z: <https://learnopengl.com/Advanced-OpenGL/Face-culling>

<sup>64</sup>Mesh Simplification [online]. Copyright © [cit. 12.02.2020]. Dostupné z: [http://graphics.stanford.edu/courses/cs468-10-fall/LectureSlides/08\\_Simplification.pdf](http://graphics.stanford.edu/courses/cs468-10-fall/LectureSlides/08_Simplification.pdf)