

Sem vložte zadání Vaší práce.





**FAKULTA  
INFORMAČNÍCH  
TECHNOLOGIÍ  
ČVUT V PRAZE**

Diplomová práce

## **Engine pro renderování a procedurální generování voxelových světů**

*Bc. Lukáš Hepner*

Katedra softwarového inženýrství  
Vedoucí práce: Ing. Adam Veseczký

28. února 2022



---

## **Poděkování**

Doplňte, máte-li komu a za co děkovat. V opačném případě úplně odstraňte tento příkaz.



---

## **Prohlášení**

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítacových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 28. února 2022

.....

České vysoké učení technické v Praze

Fakulta informačních technologií

© 2022 Lukáš Hepner. Všechna práva vyhrazena.

*Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.*

### Odkaz na tuto práci

Hepner, Lukáš. *Engine pro renderování a procedurální generování voxelových světů*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2022.

---

# **Abstrakt**

V několika větách shrňte obsah a přínos této práce v češtině. Po přečtení abstraktu by se čtenář měl mít čtenář dost informací pro rozhodnutí, zda chce Vaši práci číst.

**Klíčová slova** Nahraďte seznamem klíčových slov v češtině oddělených čárkou.

---

# **Abstract**

Sem doplňte ekvivalent abstraktu Vaší práce v angličtině.

**Keywords** Nahraďte seznamem klíčových slov v angličtině oddělených čárkou.



---

# Obsah

<b>Úvod</b>	<b>1</b>
<b>1 Herní engine</b>	<b>3</b>
1.1 Herní smyčka . . . . .	3
1.1.1 Variabilní časový krok . . . . .	4
1.2 Vykreslení objektů . . . . .	4
1.2.1 Předání dat grafické kartě . . . . .	4
1.2.1.1 Instancing . . . . .	5
1.2.2 Společná data objektů . . . . .	6
1.3 Zpracování vstupu . . . . .	7
1.3.1 Zpětné volání z OpenGL . . . . .	9
1.3.2 Návrhový vzor příkaz . . . . .	10
<b>2 Vykreslování scény</b>	<b>13</b>
2.1 Vyřazení neviditelných částí objektu . . . . .	14
2.2 Průhledné a polopruhledné textury . . . . .	16
2.3 Osvětlení scény . . . . .	17
2.3.1 Phongův osvětlovací model . . . . .	17
2.3.2 Materiál objektu . . . . .	19
2.3.3 Zdroje světla . . . . .	20
2.3.3.1 Globální světlo . . . . .	20
2.3.3.2 Bodové světlo . . . . .	21
<b>3 Mapování stínů</b>	<b>23</b>
3.1 Vykreslení mimo obrazovku . . . . .	24
3.2 Vykreslení stínu . . . . .	25
3.3 Stínové akné . . . . .	25
3.4 Ostré stíny . . . . .	27
3.5 Pozice mapy stínů na scéně . . . . .	28
3.5.1 Souřadnice frusta kamery . . . . .	29

3.5.2	Matice pohledu a projekce světla . . . . .	30
3.6	Kaskádové mapování stínů . . . . .	31
3.6.1	Mapy stínů . . . . .	32
3.6.2	Změna počtu kaskád . . . . .	35
3.7	Optimalizace vykreslování . . . . .	35
3.7.1	Průhledné textury . . . . .	36
3.7.2	Ořezání scény . . . . .	37
3.8	Výsledky . . . . .	38
<b>4</b>	<b>Modelování rostlin s využitím L-systémů</b>	<b>41</b>
4.1	L-systém . . . . .	42
4.2	Interpretace řetězců pomocí želvy . . . . .	42
4.3	Větvení v L-systémech . . . . .	42
4.4	Stochastické L-systémy . . . . .	44
4.5	Implementace . . . . .	46
4.5.1	Formát L-systému . . . . .	46
4.5.2	Želva . . . . .	46
4.5.3	Rozšířená abeceda . . . . .	47
4.5.4	Ovládání želvy . . . . .	48
4.6	Modelování rostlin . . . . .	50
4.6.1	Simulace růstu . . . . .	52
4.7	Výsledky použití L-systémů . . . . .	52
<b>Závěr</b>		<b>55</b>
<b>Literatura</b>		<b>57</b>
<b>A Seznam použitých zkratek</b>		<b>61</b>
<b>B Obsah přiloženého CD</b>		<b>63</b>

---

# Seznam obrázků

1.1	Složení bloku ze tří částí . . . . .	6
1.2	Komponenty herního objektu . . . . .	7
1.3	Použití vzoru Flyweight . . . . .	8
1.4	Přímé propojení InputHandler s Actor . . . . .	10
1.5	Využití třídy Command . . . . .	11
2.1	Pořadí vrcholů . . . . .	15
2.2	Face culling . . . . .	15
2.3	Prolínání průhledných částí textury s pozadím . . . . .	16
2.4	Zastínění průhlednou částí textury . . . . .	17
2.5	Phongův osvětlovací model . . . . .	19
2.6	Graf intenzity světla v závislosti na vzdálenosti . . . . .	22
2.7	Intenzita světla v závislosti na vzdálenosti . . . . .	22
3.1	Porovnání scnény bez stínu a se stíny . . . . .	23
3.2	Stínové akné . . . . .	26
3.3	Důvod vzniku stínového akné . . . . .	26
3.4	Zamezení vzniku stínového akné . . . . .	26
3.5	Ostrá hrana stínu . . . . .	27
3.6	Měkká hrana stínu . . . . .	28
3.7	Frustum světla okolo frusta kamery . . . . .	29
3.8	Rozdělení frusta kamery na tři části . . . . .	33
3.9	Velikost stínového akné v závislosti na kaskádě . . . . .	34
3.10	Ořezání neviditelných chunků . . . . .	38
3.11	Ořezání chunků při pohledu kolmo dolů . . . . .	39
3.12	Výsledek použití kaskádového mapování stínů . . . . .	40
4.1	Identické stromy . . . . .	41
4.2	Kvadratické Kochovy ostrovy . . . . .	43
4.3	Struktury připomínající rostliny . . . . .	44
4.4	Využití stochastického L-systému . . . . .	45

4.5	Nepravidelná koruna akáciového stromu . . . . .	49
4.6	Akáciový strom rostoucí v přírodě . . . . .	50
4.7	Model akáciového stromu . . . . .	51
4.8	Tráva na planinách . . . . .	51
4.9	Fáze růstu keře . . . . .	52
4.10	Vegetace vygenerovaná na základě L-systémů . . . . .	53

---

# Úvod



# Herní engine

Herní engine zajišťuje vykreslování scény a komunikaci s grafickou kartou, zpracování vstupu od uživatele, správu zdrojů. Poskytuje třídy s obecnou funkcionalitou, využívané generátorem terénu. Mezi ně patří:

Historie herních enginů?

- Práce s náhodnými jevy.
- Načítání a generování L-systémů.
- Továrna pro vytváření herních objektů.
- Správa textur.

## 1.1 Herní smyčka

Moderní grafické programy nezpracovávají data dávkově, ale obvykle čekají na vstup od uživatele, který následně zpracují. Na rozdíl od většiny softwaru, hry běží i když uživatel neposkytuje žádný vstup. Hra nezamrzne, animace se vykreslují, monstra se pohybují po scéně. Klíčovou částí je neblokující zpracování vstupu [1].

```
while (true) {  
    processInput();  
    update();  
    render();  
}
```

Základními kameny jsou zpracování vstupu, který se stal od posledního volání `processInput`. Funkce `update` posune herní simulaci o jeden krok. Volání `render` na závěr vše vykreslí na obrazovku. Zde vyvstává otázka, jak rychle herní smyčka běží.

### 1.1.1 Variabilní časový krok

Mějme dva hráče. První má výkonný počítač a hra běží rychlostí 60 FPS (snímků za vteřinu). Druhý hráč má méně výkonný počítač a hra běží rychlostí 6 FPS. Avatar druhého hráče by se pohyboval pouze desetinovou rychlostí. Rychlosť hry je přímo závislá na rychlosti hardwaru.

Pokud nemáme kontrolu nad hardwarem, na kterém hra běží, je toto řešení nepřípustné. Implementace musí brát v potaz, jak dlouho trvá jedna iterace herní smyčky, a podle ní zvětšit nebo zmenšit krok o který bude simulace posunuta vpřed.

```
while (!game->Finished()) {  
    const auto currentFrame = static_cast<float>(glfwGetTime());  
    deltaTime = currentFrame - lastFrame;  
    lastFrame = currentFrame;  
  
    game->ProcessInput(deltaTime);  
    game->Update(deltaTime);  
    game->Render();  
}
```

Metodám měnícím herní stav je předána hodnota `deltaTime`. Vykreslení zachycuje stav scény v okamžiku zavolání, proto nezávisí na `deltaTime`.

## 1.2 Vykreslení objektů

Vykreslovací engine má za úkol převod scény definované v 3D prostoru, na 2D plochu zobrazovacího zařízení hráče. Tento proces začíná na CPU, kde jsou herní objekty převedeny na data (matice), která jsou poslána grafické kartě. Zde jsou za pomocí série kroků transformována na zobrazitelné pixely. Tyto kroky jsou vysoce specializované a výstup každého z nich je použit jako vstup pro další.

Tyto kroky je možné masivně paralelizovat a využít tisíců jader, které může grafická karta obsahovat. Každé jádro spouští malý program, pro každý krok vykreslovacího řetězce. Tyto programy jsou nazývané shadery. Některé ze shaderů může definovat vývojář a nahradit nimi existující výchozí shadery. OpenGL pro jejich programování využívá OpenGL shading language (GLSL) [2].

### 1.2.1 Předání dat grafické kartě

Scéna je složena z kusů terénu (třída `Chunk`), obsahující herní objekty. Chunky jsou rozmístěné na ploše vymezené osami `x` a `z`. Třída `Scene` má za úkol správu chunků a serializaci jejich dat do 1D pole, předané grafické kartě.

Každý herní objekt lze reprezentovat jako matici 4x4 obsahující informace o posunu vůči počátku světa, škálování a textuře.

Převod herního objektu na matici:

dopsat informace o  
model matrix

```
for (const auto& o : obs) {
    assert(o.GetComponent<Components::Transform>());
    auto model =
        o.GetComponent<Components::Transform>().ModelMat();

    assert(o.GetComponent<Components::SpritesheetTex>());
    const auto& texPos =
        o.GetComponent<Components::SpritesheetTex>().GetTexPos();
    Helpers::Math::PackVecToMatrix(model, texPos);

    buffer[cube]->push_back(model);
}

[[nodiscard]] glm::mat4 ModelMat() const {
    auto model = glm::mat4(1.0f); // identity matrix
    model = glm::translate(model, Position);
    return glm::scale(model, Scale);
}
```

### 1.2.1.1 Instancing

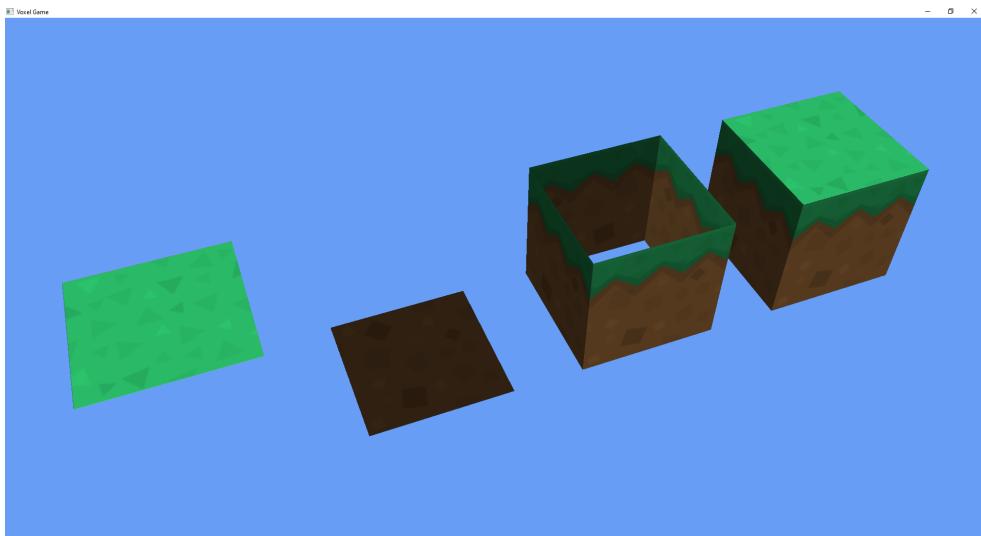
Herní scéna obsahuje velké množství objektů se stejnou geometrií (např. krychle), s různou transformací vůči počátku světa. Každá krychle je složena z 12 trojúhelníků. Její vykreslení je téměř okamžité. Pokud by pro každou z nich bylo voláno samostatné vykreslení (draw call), velice rychle doje k drastickému snížení výkonu (komunikace s kartou přes sběrnici, uložení dat do patřičných bufferů...). Lepší řešení je poslat všechna data kartě najednou a ty následně vykreslit pomocí stejné geometrie. Toto řešení se nazývá instancing [3].

Scéna umožňuje definovat vlastní geometrie – krychle, krychle bez podstav, nebo jakákoli jiná kombinace stěn krychle. Pro každou z nich alokuje místo na grafické kartě, uloží do něj data objektů a následně samostatně vykreslí každou z nich.

```
// bind data
glBindBuffer(GL_ARRAY_BUFFER, InstanceDataBufferIds_[cube]);
unsigned offset = 0;
for (const auto& chunk : instancesData) {
    glBindBufferSubData(
        GL_ARRAY_BUFFER,
        offset * sizeof(glm::mat4),
```

## 1. HERNÍ ENGINE

---



Obrázek 1.1: Složení bloku ze tří částí

```
chunk->size() * sizeof(glm::mat4),  
chunk->data());  
offset += chunk->size();  
}  
  
CubeRenderers_[cube].GetDefaultMesh().BindBatchAttribPtrs();
```

Rozdílné geometrie jsou užitečné, pokud chceme definovat krychli s rozdílnými podstavami a stěnami. Na obrázku 1.1, lze vidět blok trávy složený ze tří částí.

### 1.2.2 Společná data objektů

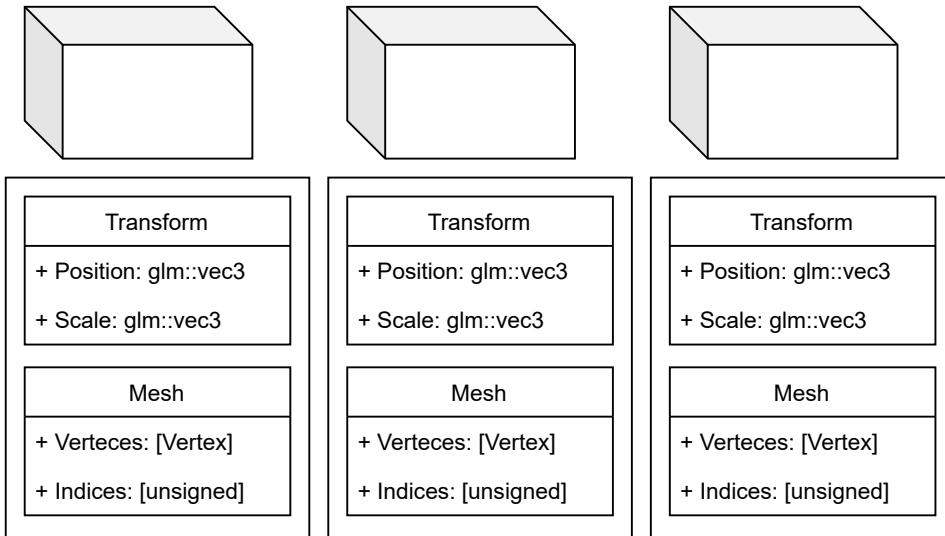
Instancing umožňuje grafické kartě používat společná data pro všechny vykreslované objekty. Jeho protějškem v objektovém světe je návrhový vzor Flyweight.

Pokud má být objekt vykreslitelný musí obsahovat komponentu **Transform** (určující jeho pozici a velikost) a komponentu **Mesh** (představující geometrii objektu) – obrázek 1.2.

**Mesh** obsahuje informace o vrcholech (**Vertex**) vykreslovaných trojúhelníků – pozici na scéně, normálový vektor k úsečce mezi dvěma vrcholy trojúhelníku, pozici na textuře. A pole indexů, které říká, z jakých vrcholů jsou trojúhelníky složeny. Každý vrchol v má velikost  $8 * 4 = 32$  B a **Mesh** jich obsahuje  $6 * 4$ . Každá stěna krychle má čtyři různé vrcholy<sup>1</sup>. Pole indexů má velikost  $12 * 3 * 4$  B

---

<sup>1</sup>Krychle má 8 vrcholů. Vrcholy jednotlivých stěn se ale liší v normálovém vektoru a pozicí na textuře.



Obrázek 1.2: Komponenty herního objektu

(počet trojúhelníků, počet vrcholů trojúhelníku, velikost `unsigned`). Celková velikost `Mesh`u představující krychli je:

$$8 * 4 * 6 * 4 + 12 * 3 * 4 = 912 \text{ B}$$

Pokud má být engine schopný vykreslovat desetitisíce objektů je tato paměťová náročnost neúnosná. Komponentu `Mesh` není možné odebrat z herního objektu, protože obsahuje pozici na textuře, lišící se mezi objekty. Pozice na textuře představuje čtverec, který bude vybrán z textury a aplikován na povrch krychle. Všechny objekty mají texturu čtverce. Díky tomu je možné definovat čtverec na počátku textury a jeho posun uložit do samostatné komponenty – `SpritesheetTex`. Toto nové rozložení je znázorněno na obrázku 1.3.

Komponenta `SpritesheetTex` obsahuje dvě čísla ve formátu float (`float` je zvolen pro jednodušší komunikaci s grafickou kartou — všechna předaná data mají formát float). Komponenta v paměti zabírá pouhých 8 B. Velikost herního byla redukována o 904 B.

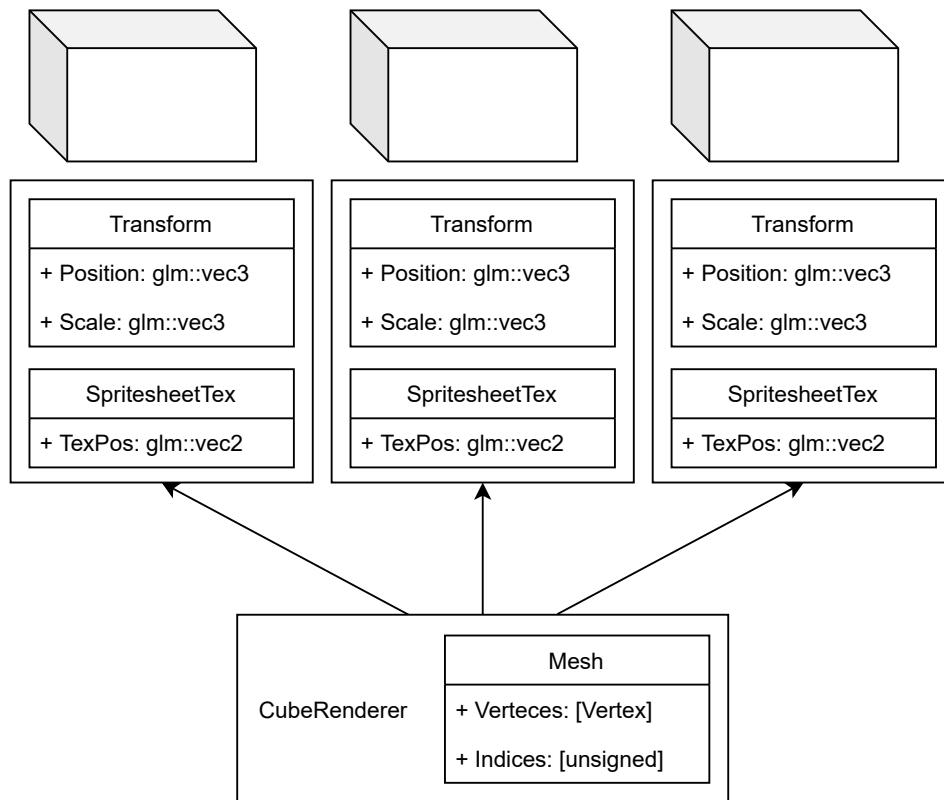
`Mesh` není dále používána jako komponenta herního objektu. Její implementace ze jmenného prostoru `Renderer`, kterou komponenta `Mesh` obsahuje jako svůj atribut, je použita pro vykreslení všech objektů se stejnou geometrií najednou. Vykreslení objektů zajišťuje třída `CubeRenderer`, viz obrázek 1.3.

### 1.3 Zpracování vstupu

O vstup z klávesnice a myši se stará abstraktní třída `InputHandler`, obsahující pole příkazů (třída `Command`) ovládajících kamery. Konkrétní implementaci má na starosti třída `InputHandlerGL`, pevně svázaná s voláními OpenGL.

## 1. HERNÍ ENGINE

---



Obrázek 1.3: Použití vzoru Flyweight

Vstup je zpracován dvojím způsobem:

1. `ProcessInput`, voláno v rámci herní smyčky.
2. Zpětné volání z OpenGL.

Pohyb hráče zajišťuje metoda `ProcessInput`. Délka pohybu je škálována parametrem `delta`. Vstupy jsou zpracovány při volání metody, kontrolou kláves, které jsou právě stlačené. Tento způsob je ideální pro kontinuální změnu stavu (např. pozice) objektu. Hráč může klávesu držet pro plynulý pohyb, nebo ji opakovaně mačkat pro drobné korekce.

Tento způsob je naprosto nevhodný pro změnu stavu nabývajícího několika diskrétních hodnot. Například vypnutí/zapnutí světla. Hráč musí klávesu stisknout a pustit po dobu trvání jednoho snímku — při 60 FPS na 16,67 ms. Při běžném stisknutí může dojít k několikanásobné změně stavu.

### 1.3.1 Zpětné volání z OpenGL

K zamezení opakovaného čtení stisknuté klávesy lze využít nastavení funkce, kterou OpenGL zavolá při stisknutí klávesy. Signatura této funkce musí být [4]:

```
void function_name(
    GLFWwindow* window,
    int key,
    int scancode,
    int action,
    int mods
)
```

Výhody třídní hierarchie (dědění z obecné třídy pro zpracování vstupu a její konkrétní implementace pro OpenGL, možnost mít více objektů kontrolujících vstup a jejich výměna pro změnu ovládaní) znemožňují nastavení přímého volání metody z OpenGL – metoda je závislá na stavu objektu, pro použití jako zpětné volání by musela být statická. Zpětné volání zajišťuje funkce umístěné ve jmenném prostoru `Input::Detail`.

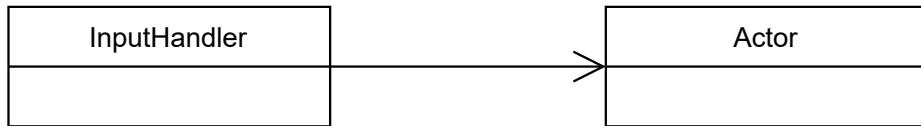
```
namespace Input::Detail {
void CursorPosCallback(GLFWwindow*, double xPos, double yPos) {
    currentHandler->ProcessMouse(
        static_cast<float>(xPos),
        static_cast<float>(yPos));
}

void ScrollCallback(GLFWwindow*, double, double yOffset) {
    currentHandler->ProcessMouseScroll(
        static_cast<float>(yOffset));
}

void KeyCallback(GLFWwindow*, int key,
                 int scanCode, int action, int mods) {
    currentHandler->ProcessKey(key, scanCode, action, mods);
}
} // namespace Input::Detail
```

Tyto funkce nejsou umístěny v hlavičkovém souboru a uživatel by je neměl napřímo využívat. Zpětné volání je automaticky nastaveno při inicializaci objektu.

```
void Input::InputHandlerGl::SetCallbacks()
{
    Detail::currentHandler = this;
```



Obrázek 1.4: Přímé propojení InputHandler s Actor

```
 WindowManagerGl::SetCursorPosCallback(Detail::CursorPosCallback);  
 WindowManagerGl::SetScrollCallback(Detail::ScrollCallback);  
 WindowManagerGl::SetKeyCallback(Detail::KeyCallback);  
 }
```

Jmenný prostor `Input::Detail` obsahuje pouze jednu proměnnou, označující současný objekt zpracovávající vstup.

### 1.3.2 Návrhový vzor příkaz

Jednoduchá implementace kódu zpracovávajícího vstup by mohla vypadat:

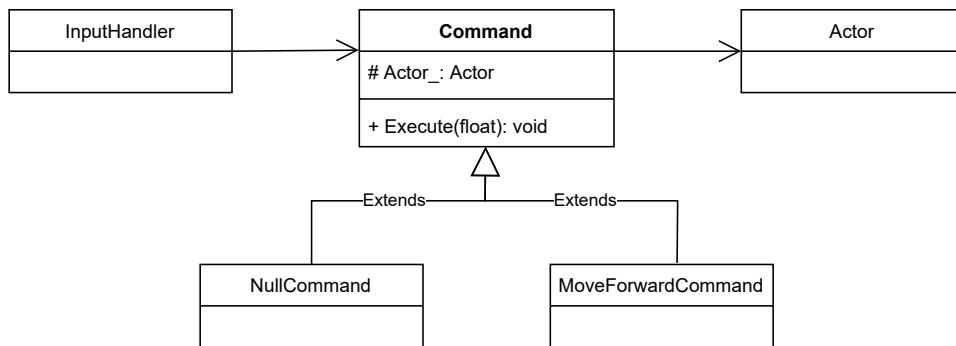
```
void InputHandler::handleInput() {  
    if (isPressed(BUTTON_W)) object.MoveForward();  
    else if (isPressed(BUTTON_S)) object.MoveBackward();  
    else if (isPressed(BUTTON_A)) object.MoveLeft();  
    else if (isPressed(BUTTON_D)) object.MoveRight();  
}
```

Tento kód funguje, pokud jsme ochotni natvrdo zadřátovat vstup hráče k herním akcím viz obrázek 1.4, ale hodně her hráče nechá nakonfigurovat si vlastní mapování tlačítek. K tomuto potřebujeme vyměnit přímé volání metody A za metodu B. K tomu potřebujeme objekt reprezentující volání metody [5].

Definujme abstraktní třídu `Command` představující spustitelný příkaz.

```
class Command {  
protected:  
    Renderer::Camera* Actor;  
  
public:  
    explicit Command(Renderer::Camera* actor) : Actor(actor) {}  
    virtual ~Command() = default;  
  
    virtual void Execute(float delta) = 0;  
};
```

### 1.3. Zpracování vstupu



Obrázek 1.5: Využití třídy Command

V konkrétní implementaci příkazu je z něj poděděno a metoda `Execute` přepsána pro specifické chování.

```

class MoveForwardCommand : public Command {
public:
    explicit MoveForwardCommand(Renderer::Camera* actor)
        : Command(actor) {}
    void Execute(float delta) override {
        Actor->Move(delta, 0.0, 1.0);
    }
};
  
```

`InputHandler` je odstíněn od volání prováděných na `actor` (obrázek 1.5), sníží se tak spojení mezi třídami a zvýší se soudržnost (high cohesion). `InputHandler` má na starosti zpracování vstupu, ne ovládání herní postavy.

Příkazy jsou uloženy v poli:

```

std::array<std::unique_ptr<Commands::Command>, Keys::Count>
Commands
  
```

Toto pole je naplněno při inicializaci `InputHandleru`, neobsahuje tedy žádné `nullptr` ukazatele. Při odstranění příkazu reagujícího na stlačení tlačítka, by při jeho stisku došlo k vyhození `nullptr` výjimky a pádu programu. Toto chování může být ošetřeno důslednou kontrolou obsahu `unique_ptr`, byl by tím ovšem porušen objektový návrh. Řešení použité v enginu je `NullCommand`.

```

class NullCommand : public Command {
public:
    NullCommand() : NullCommand(nullptr) {}
    explicit NullCommand(Renderer::Camera*) : Command(nullptr){}
    void Execute(float) override {}
};
  
```

## 1. HERNÍ ENGINE

---

Toto řešení zachovává principy objektového návrhu. Přijímá parametr `actor`, svému předku však předá `nullptr`. Tělo metody `Execute` je prázdné a s `actorem` není interagováno. Volání metody `Execute` mohou být bezpečně provedena na všech prvcích pole.

## Vykreslování scény

OpenGL představuje stavový automat. Jeho výchozí hodnoty jsou nastaveny při spuštění programu (např.: způsob mísení barev, povolení testu hloubky...). Některé se musí měnit podle dat, které se mají vykreslit (např.: face culling). Grafická karta má před každým voláním pro vykreslení scény uložena data objektů ve svých bufferech. Přiřazené textury do texturovacích jednotek. Nastavené proměnné shaderů a shadery, které se mají pro vykreslení použít. Tyto parametry představují stav OpenGL, podle něhož se vykreslí scéna. Vykreslení může probíhat ve více krocích – vykreslovací příkaz OpenGL je zavolán několikanásobně. Při každém volání proběhnou následující kroky [6]:

1. Specifikace vrcholů — Načtení formátu vrcholů a předání dat, která budou zpracována dále v řetězci.
2. Vertex shader — Provede zpracování vrcholu na základě uživatelem specifikovaného programu a předá ho k dalšímu zpracování. Zpracován může být pouze jeden vrchol a ten musí být předán do dalšího kroku. Poskytnutí vertex shaderu je povinné.
3. Teselace — Nepovinný krok, který může rozdělit primitivum (trojúhelník, úsečku...) na několik menších primitiv [7].
4. Geometry shader — Uživatelem definovaný program, zpracovávající primitiva. Výstupem je nula nebo více primitiv. Vstupní a výstupní primitiva musí být přesně definovaná. Geometry shader může změnit jeho geometrii, provést transformaci souřadnic... Tento krok není povinný.
5. Post-processing vrcholů — Složená primitiva (např.: pruh trojúhelníků) jsou převedena na jednoduchá primitiva (úsečky, body, trojúhelníky). Primitiva jejichž část se nachází mimo prostor obrazovky jsou rozdělena, aby vznikla nová, jež jsou uvnitř prostoru obrazovky. Trojúhelníková primitiva mohou být vyřazena, pokud nemíří k pozorovateli. Všechna nová primitiva jsou uložena do výstupních bufferů pro další zpracování.

## 2. VYKRESLOVÁNÍ SCÉNY

---

6. Rasterizace — Primitiva, která se dostanou do této fáze jsou převedena na fragmenty. Fragment je soubor hodnot obsahující výstupy předchozích shaderů. Jejich hodnota je nastavena interpolací hodnot vrcholů, z kterých se primitivum skládá. Každý fragment obsahuje pozici v prostoru obrázovky [8].
7. Fragment shader — Výstupem fragment shaderu je list barev, které budou zapsány do výstupních bufferů, hodnota hloubky a hodnota šablony (stencil value). Fragment shader může být definován uživatelem. Pokud není, hodnota barev není definovaná, je pouze zapsána hodnota hloubky a šablony.
8. Operace provedené na vzorcích (Per-Sample Operations) — Výstupní data fragmentu jsou podrobena sérii testů (mohou být specifikované uživatelem, např.: hloubkový test), které je mohou vyřadit z podílení se na výsledné barvě pixelu. Pokud projdou, je provedeno míchání barev s barvami, které již obsahuje framebuffer.

world coords, etc.

### 2.1 Vyřazení neviditelných částí objektu

Část scény tvoří krychle mající všech šest stěn vyplňených neprůhlednou texturou. Při pohledu na krychli může hráč vidět maximálně tři její stěny najednou (z určitých úhlů pouze jednu nebo dvě). Minimálně 50 % každé krychle je vykreslováno zbytečně. OpenGL je schopno vyřadit stěny, které směřují pryč od hráče, z vykreslovacího procesu a snížit počet volání fragment shaderu. Tato technika se nazývá face culling [9].

K rozpoznání stěn (trojúhelníků), které směřují pryč od hráče OpenGL používá pořadí jeho vrcholů. Vrcholy mohou být definované ve směru nebo protisměru hodinových ručiček – obrázek 2.1. Při pohledu na grafické primitivum z druhé strany, se změní pořadí jeho vrcholů. V základním nastavení OpenGL, jsou primitiva s vrcholy definovanými po směru hodinových ručiček, považována za směřující k hráči.

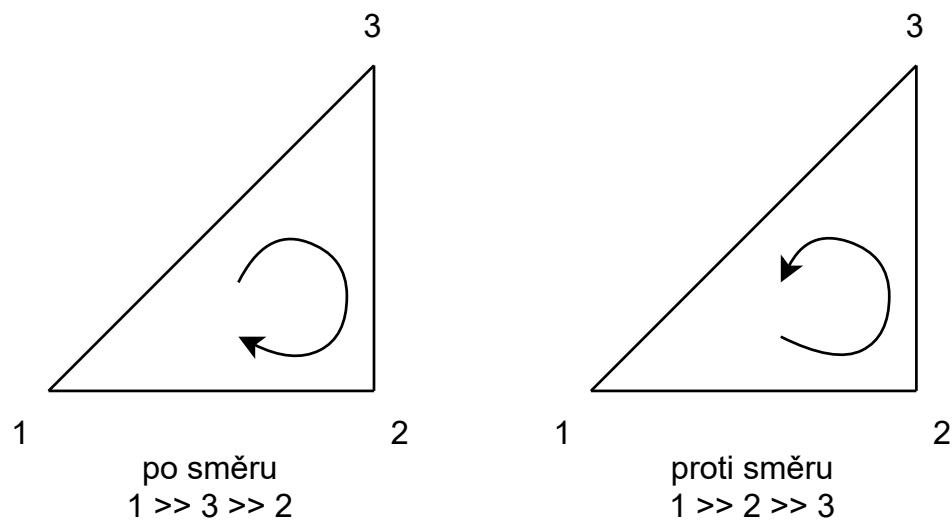
Tato technika není v základním nastavení OpenGL zapnutá. Musí se povolit zavoláním funkce:

```
glEnable(GL_CULL_FACE);
```

Pro objekty, které nemají všech šest stěn nebo mají průhlednou texturu, je nutné vypnout face culling. Hráči se zobrazí i vnitřek objektu, který by jinak nebyl vidět. Příkladem je vykreslování trávy – obrázek 2.2. Na pravém bloku je tráva vykreslena se zapnutým face cullingem. Na levém bloku je face culling vypnutý.

## 2.1. Vyřazení neviditelných částí objektu

---



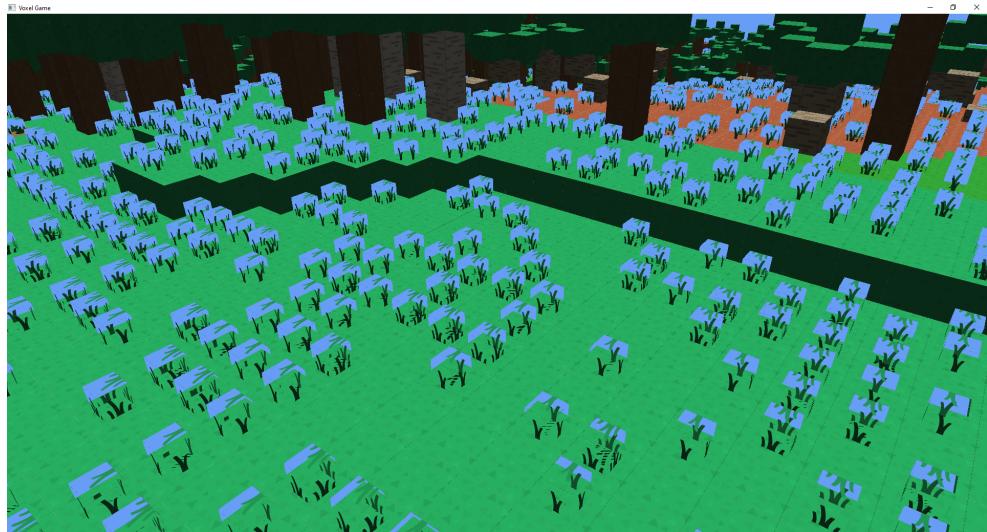
Obrázek 2.1: Pořadí vrcholů



Obrázek 2.2: Face culling

## 2. VYKRESLOVÁNÍ SCÉNY

---



Obrázek 2.3: Prolínání průhledných částí textury s pozadím

### 2.2 Průhledné a poloprůhledné textury

Při vykreslování textur majících průhlednost ( $\alpha < 1$ ) může nastat několik problémů. Prvním z nich je pořadí vykreslování objektů. Pokud má textura průhlednost, musí se její barva skombinovat s barvami textur, které překrývá. Na obrázku 2.3 jsou textury vykreslovány postupně z pravého dolního rohu, po rádcích směrem k levému hornímu rohu. Lze si všimnout, že průhledná část trávy je korektně smíchána s barvou bloku pod ní (je vykreslen první). Bloky v pozadí obrázku jsou překryté modrou barvou v místech, kde je textura trávy průhledná.

Tento jev nastává kvůli použití depth bufferu, ten šetří výpočetní výkon zahodením fragmentů, které by nebyly vidět. Jsou do něj však uloženy všechny fragmenty nezávisle na průhlednosti. V ideálním případě by nejdříve měly být vykreslené všechny fragmenty, které se budou podílet na barvě pixelu, od nejvzdálenějšího k nejbližšímu [10].

Toto řešení vyžaduje řazení objektů na scéně podle jejich vzdálenosti od kamery. Poloha kamery se s pohybem hráče neustále mění a řazení objektů by snižovalo výkonost.

Avšak objekty bez průhledných částí překryjí všechny objekty nehledě na průhlednost. Scénu tak lze rozdělit na dvě části. A objekty bez průhlednosti vykreslit jako první. Na obrázku 2.4 je tato technika implementována. Barva průhledné části trávy je korektně smíchána s barvou bloku v pozadí.

Problém s pořadím textur obsahujících průhlednost stále přetrvává. Na obrázku 2.4 lze jasně vidět průhlednou část textury trávy, překrývající trávu v pozadí. Pro textury mající části úplně průhledné nebo úplně neprůhledné lze



Obrázek 2.4: Zastínění průhlednou částí textury

problém vyřešit ve fragment shaderu. Všechny fragmenty, mající průhlednost menší, než stanovená mez, budou vyřazeny z vykreslovacího řetězce.

```
float alpha = vec4(texture(texture_diffuse1, TexCoord)).a;
if (alpha < 0.05)
    discard;
```

Problém s poloprůhlednými texturami však přetravává a pro jeho vyřešení je nutné poloprůhledné textury seřadit.

## 2.3 Osvětlení scény

Osvětlení v reálném světě je extrémně složitý problém. Jeho přesná simulace v aplikacích reálného času je výpočetně náročná. Pro zjednodušení výpočtu existují modely, které produkují výsledky podobné reálnému světu.

### 2.3.1 Phongův osvětlovací model

Jednou z approximací reálného světa je Phongův osvětlovací model, skládající se ze tří hlavních částí [11].

- Okolní (ambient) světlo — I za tmy jsou objekty nasvíceny světlem odraženým od ostatních objektů (měsíc) nebo vzdáleným zdrojem světla (hvězdy). Objekty většinou nejsou zcela tmavé. A nastavením konstanty pro ambientní osvětlení můžeme simulovat tento jev.

## 2. VYKRESLOVÁNÍ SCÉNY

---

- Difúzní světlo — Simuluje dopad světla na objekt. Pokud je objekt natočen ke zdroji světla, bude ním více ovlivněn.
- Lesklé světlo — Simuluje odraz světla od lesklého předmětu.

Výpočet osvětlení může být prováděn na CPU, nastavením světlosti každé stěny bloku. Pokud má být docíleno věrnějšího výsledku, je nutné počítat světlost pro každý fragment objektu (úhel dopadu světla se může výrazně lišit pro fragmenty na opačných stranách objektu). Výpočet je proto prováděn na grafické kartě, přesněji ve fragment shaderu.

Ambientní složka světla je předána grafické kartě, kde je vynásobena s barvou objektu – textury.

```
vec3 ambient =  
    light.ambient * vec3(texture(texture_diffuse1, TexCoord));
```

Pro výpočet difuzní složky, je shaderu předána pozice světla (`light.position`) a jeho barva (`light.diffuse`). Světlo dopadající kolmo na fragment má největší efekt na jeho výslednou barvu. Pro výpočet úhlu dopadu je využit normálový vektor fragmentu<sup>2</sup>. Vynásobením (skalární součin) normálového vektoru fragmentu a vektoru směřujícího od fragmentu ke světlu, je získána hodnota 1 pro vektory svírající nulový úhel a 0 pro kolmé vektory. Rozsah hodnot je zaručen normalizací vektorů.

Pokud vektory svírají úhel větší než  $90^\circ$  je hodnota skalárního součinu záporná. Tuto situaci si lze představit jako dopad světla z opačné strany fragmentu. Ten by tedy neměl být osvětlený, `diff` je nastaven na 0. Hodnota difuzní složky je vynásobena s barvou světla a fragmentu.

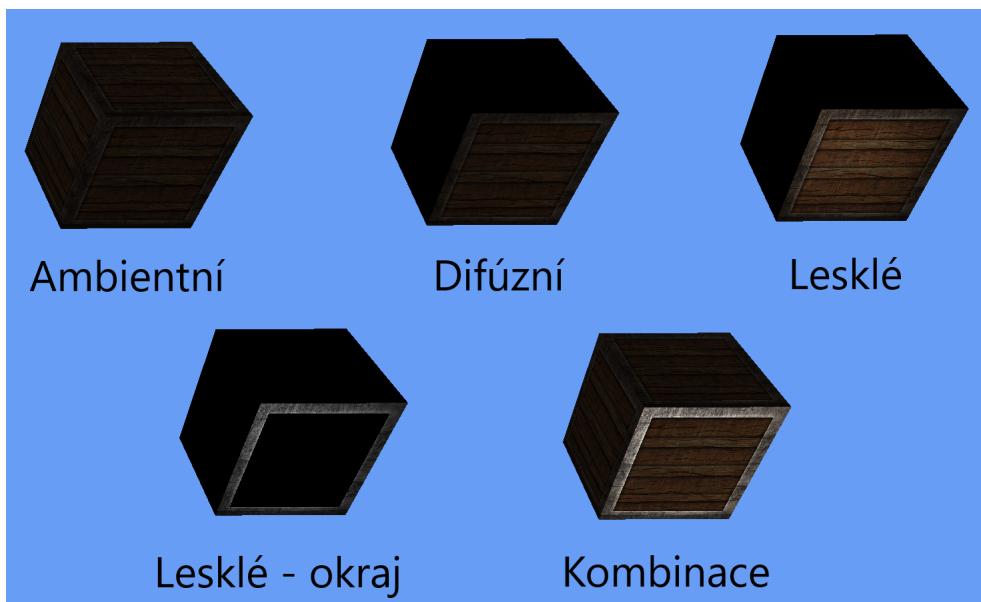
```
vec3 lightDir = normalize(light.position - FragPos);  
float diff = max(dot(norm, lightDir), 0.0);  
vec3 diffuse = diff * light.diffuse *  
    vec3(texture(texture_diffuse1, TexCoord));
```

Intenzita odraženého světla (lesklá složka) je závislá na pozici pozorovatele (předána shaderu jako `view_pos`). Pokud odražené světlo směruje přímo do oka pozorovatele, je jeho intenzita nejvyšší. Vektor odraženého světla lze spočítat pomocí funkce `reflect`. Její první parametr je vektor směřující od světla k fragmentu, tedy vektor opačný k `lightDir`. Druhý parametr je normálový vektor.

Intenzita je opět spočítána pomocí skalárního součinu, a umocněna lesklostí materiálu. Čím vyšší lesklost, tím méně je světlo rozptýleno do všech směrů a velikost efektu se zmenší [11].

---

<sup>2</sup>Normálový vektor je předán vertex shaderu jako jeden z atributů vrcholu trojúhelníku. Ten ho následně předá fragment shaderu.



Obrázek 2.5: Phongův osvětlovací model

```

vec3 viewDir = normalize(view_pos - FragPos);
vec3 reflectDir = reflect(-lightDir, norm);
float spec =
    pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
vec3 specular = spec * light.specular *
    vec3(texture(texture_diffuse1, TexCoord));

```

Na obrázku 2.5 je vykreslená bedna osvícena všemi druhy světel a jejich kombinací.

### 2.3.2 Materiál objektu

Obrázek 2.5 zobrazuje dvě možnosti použití lesklého světla. Bedna jejíž celý povrch odráží světlo působí nepřirozeným dojmem. V reálném světě dřevo světlo neodráží<sup>3</sup>. Lesknout by se měl pouze její kovový okraj, a to jen v místech kde není poškozen.

Tento problém je vyřešen předáním nové textury pro lesklé světlo. Řádek vyhodnocující barvu lesklého světla nebude používat texturu `texture_diffuse1`, na místo ní využije `texture_specular1`.

```

vec3 specular = spec * light.specular *
    vec3(texture(texture_specular1, TexCoord));

```

<sup>3</sup>Dřevo s naleštěným povrchem světlo odrážet může, tato povrchová úprava však pravděpodobně nebude aplikována na přepravní bednu.

Lesklá textura nemusí definovat barvu odrazu – odražené světlo má barvu zdroje světla, ale pouze její intenzitu. Dřevěná část textury má černou barvu (žádný odraz), kovová část je převedena do černo-bílého spektra [12].

Pokud objekt nemá používat lesklé odrazy, nemusí být textura definována. Při změně nastavení textur, jsou všechny navázané textury odpojeny (unbound from texture samplers). Následně jsou navázané všechny definované textury.

```
unsigned int diffuseNr = 1;
unsigned int specularNr = 1;
for (unsigned int i = 0; i < Textures.size(); i++) {
    // retrieve texture number (the N in diffuse_textureN)
    std::string number;
    std::string name = Textures[i].Type_;
    if (name == "texture_diffuse")
        number = std::to_string(diffuseNr++);
    else if (name == "texture_specular")
        number = std::to_string(specularNr++);
    // now set the sampler to the correct texture unit
    shader.SetInteger((name + number).c_str(), i);
    Textures[i].Bind(i);
}
```

Pokud dojde ke čtení z textury, která není navázána, bude vrácena černá barva, představující nulový lesk [13].

### 2.3.3 Zdroje světla

Aktuální implementace představuje bodový zdroj světla, jehož intenzita neklesá s uraženou vzdáleností. Tento model nereprezentuje reálné chování světla. Zavedené jsou dva nové druhy světla:

- Globální osvětlení — Představuje zdroj světla nekonečně vzdálený od scény, jehož paprsky jsou rovnoběžné a jeho intenzita se nemění. Jeho pomocí lze modelovat objekty jako je Slunce.
- Bodové osvětlení — Paprsky se šíří všemi směry. Intenzita klesá s uraženou vzdáleností.

#### 2.3.3.1 Globální světlo

Pro výpočet globálního světla není potřeba jeho polohu, pouze směrový vektor jeho paprsků, definovaný směrem od zdroje světla. Při výpočtu intenzity světla byl použit směr od fragmentu ke světlu. Jedinou změnou oproti dosavadnímu výpočtu je převrácení jeho směru.

```

float globalInt =
    max(dot(norm, normalize(-light.globalDir)), 0.0);
vec3 global = globalInt * light.global *
    vec3(texture(texture_diffuse1, TexCoord));

```

### 2.3.3.2 Bodové světlo

Intenzitu světla je možné snižovat lineárně s vzdáleností, kterou urazí. Toto řešení zajistí nižší nasvícení vzdálených objektů, a však vypadá poněkud uměle. Světla v reálném světě jsou z pravidla velmi jasné, pokud se nacházíme v jejich blízkosti. Jejich jas velice rychle klesá s nárůstají vzdáleností. V určitém bodě se klesání zpomalí a přiblížuje se k nule.

K výpočtu útlumu světla lze použít následující rovnici [14]:

$$F_{att} = \frac{1.0}{K_c + K_l * d + K_q * d^2}$$

Proměnná  $d$  reprezentuje vzdálenost fragmentu od zdroje světla. Dále jsou nastaveny tři konstanty:

- $K_c$  — Konstantní složka. Obvykle je ponechána na hodnotě 1. Zajišťuje, aby hodnota jmenovatele neklesla pod 1 a nedošlo k navýšení intenzity světla.
- $K_l$  — Lineární složka.
- $K_q$  — Kvadratická složka.

Graf 2.6 zobrazuje porovnání hodnot intenzity světla danou rovnicí útlumu (oranžová křivka) a lineární závislostí na vzdálenosti (šedá přímka).

Hodnoty parametrů jsou určeny požadovaným dosvitem světla. Parametr  $K_c = 1$ ,  $K_l = 0,045$  a  $K_q = 0,0075$  [15]. Lineární rovnice má tvar:

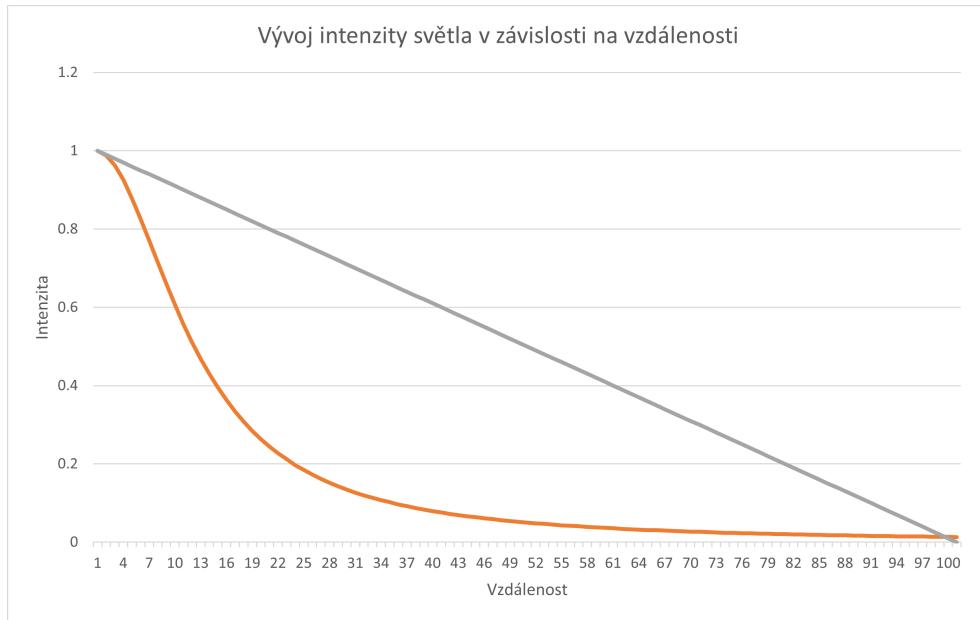
$$F_{att} = \frac{100 - d}{100}$$

Obrázek 2.7 ve své levé části zobrazuje scénu vykreslenou pomocí rovnice útlumu. Pravá část snižuje intenzitu světla lineárně.

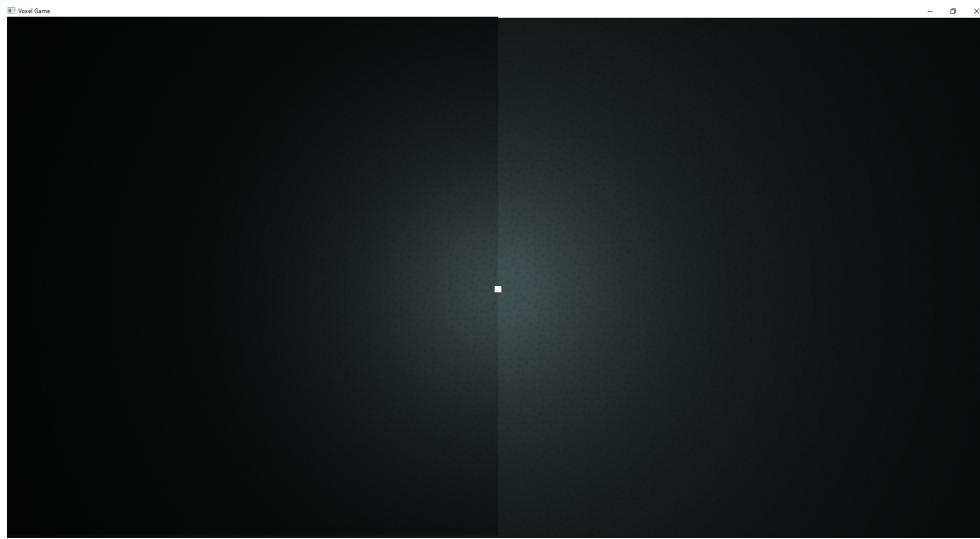
Je tento obrázek potřeba?

## 2. VYKRESLOVÁNÍ SCÉNY

---



Obrázek 2.6: Graf intenzity světla v závislosti na vzdálenosti



Obrázek 2.7: Intenzita světla v závislosti na vzdálenosti

# KAPITOLA 3

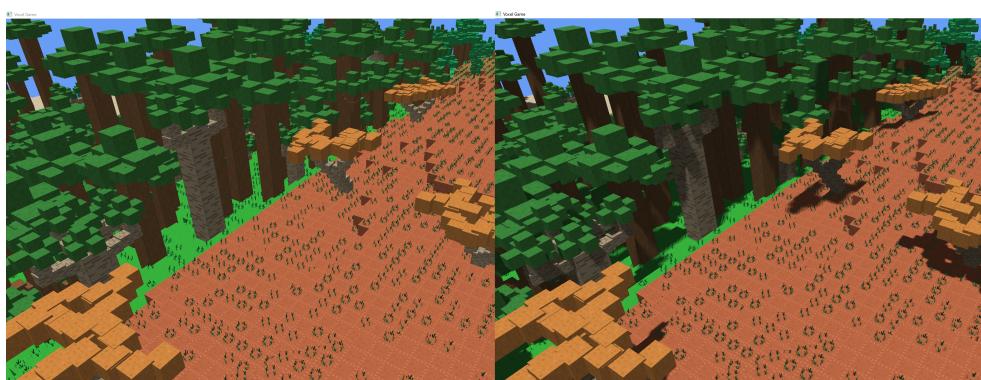
## Mapování stínů

V levé části obrázku 3.1 lze vidět scénu nasvícenou globálním světlem. Každý blok scény je nasvícen individuálně a ignoruje ostatní bloky, které by ho mohli zastiňovat. Tento efekt působí velice nereálně zvláště v podrostu džungle a pod širokými korunami akácií, které jsou nasvíceny plným světlem. Přidáním stínů lze docílit věrohodnější reprezentace reálného světa a hráč získá lepší představu o pozici bloků vůči sobě.

Trasování tisíců paprsků světla v reálném čase je výpočetně neúnosné. Proto je použita technika mapování stínů, využívající hloubkového bufferu grafické karty. Scéna je nejprve vykreslena z pohledu světla a hodnoty hloubky fragmentů uloženy do textury nazývané mapa stínů.

Moc siroky obra-  
zek

Pro druhé vykreslovací kolo, je mapa stínů předána fragment shaderu. Ten provede transformaci pozice fragmentu z prostoru světa (world space) na pozici v prostoru světla a porovná jeho hloubku s hloubkou uloženou v mapě stínů [16].



Obrázek 3.1: Porovnání scény bez stínu a se stínem

### 3. MAPOVÁNÍ STÍNU

---

#### 3.1 Vykreslení mimo obrazovku

OpenGL umožnuje změnit objekt, do kterého se uloží výsledek vykreslovacího řetězce. V základním nastavení to je obrazovka hráče. Toto nastavení lze změnit vytvořením nového framebufferu a textury, do které se bude vykreslovat.

Textura může mít jiné rozlišení, než má okno, na které je vykreslována scéna. Zvětšením rozlišení se zlepší kvalita stínů, na úkor rychlosti vykreslování. Mapa stínů nemusí pokrývat veškeré objekty, které hráč vidí. Proto je textuře nastaven okraj, s maximální hloubkou. Při čtení pixelů mimo texturu se díky parametru `GL_CLAMP_TO_BORDER` vrátí maximální hloubka a vykreslovaný objekt nebude mít stín.

```
Texture_.Generate(Width, Height, nullptr);
glBindTexture(GL_TEXTURE_2D, Texture_.Id);
// set no shadow outside of the shadow map
constexpr float borderColor[] = { 1.0f, 1.0f, 1.0f, 1.0f };
glTexParameteri(
    GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
glBindTexture(GL_TEXTURE_2D, 0);
```

Pro vykreslení je vygenerován nový framebuffer, jemuž je textura předána jako hloubková složka. Při výpočtu stínu není potřeba buffer pro barvu. Framebuffer objekt by bez něj nebyl kompletní, a proto se musí explicitně nastavit `glDrawBuffer` a `glReadBuffer` na `GL_NONE`.

```
// attach to framebuffer
 glGenFramebuffers(1, &FBO_);
 glBindFramebuffer(GL_FRAMEBUFFER, FBO_);
 glFramebufferTexture2D(
    GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
    GL_TEXTURE_2D, Texture_.Id, 0);
 glDrawBuffer(GL_NONE);
 glReadBuffer(GL_NONE);
```

Před vykreslením se musí nastavit velikost zobrazovacího zařízení, na které se má scéna vykreslit. V tomto případě velikost textury mapy stínů.

```
glViewport(0, 0, Width, Height);
```

Pro vykreslení do mapy stínů je použit shader, který transformuje vertex do souřadnic světla. A předá pozici textury k dalšímu zpracování.

```
TexCoord = vec2(aTexCoord.x + shift.x, aTexCoord.y + shift.y);
gl_Position = lightSpaceMatrix * model * vec4(aPos, 1.0);
```

Fragment shader vyřadí všechny průhledné fragmenty a zapiše jejich hloubku.

```
float alpha = vec4(texture(texture_diffuse1, TexCoord)).a;
if (alpha < 0.05)
    discard;

gl_FragDepth = gl_FragCoord.z;
```

## 3.2 Vykreslení stínu

K vykreslení stínu je použit shader z kapitoly o světle. Globální složka světla je vynásobena hodnotou  $(1 - shadow)$ , kde  $shadow = 1$  znamená maximální stín a  $shadow = 0$  žádný stín.

```
float shadow = ShadowCalculation(fs_in.FragPos, dotLightNormal);
vec3 global =
    (1 - shadow) * globalInt * light.global *
    vec3(texture(texture_diffuse1, TexCoord));
```

Funkce `ShadowCalculation` vrátí hodnotu 1, pokud je hloubka fragmentu větší než hodnota v mapě stínů, jinak vrátí 0.

```
// perform perspective divide
vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
// transform to [0,1] range
projCoords = projCoords * 0.5 + 0.5;
float closestDepth = texture(shadowMap, projCoords.xy).r;
float currentDepth = projCoords.z;
return currentDepth > closestDepth ? 1.0 : 0.0;
```

Scéna 3.2 je vykreslena s použitím toho shaderu. Stíny stromů jsou správně vykresleny, oko diváka však přitáhnou artefakty vzniklé při výpočtu stínu, nazývající se stínové akně.

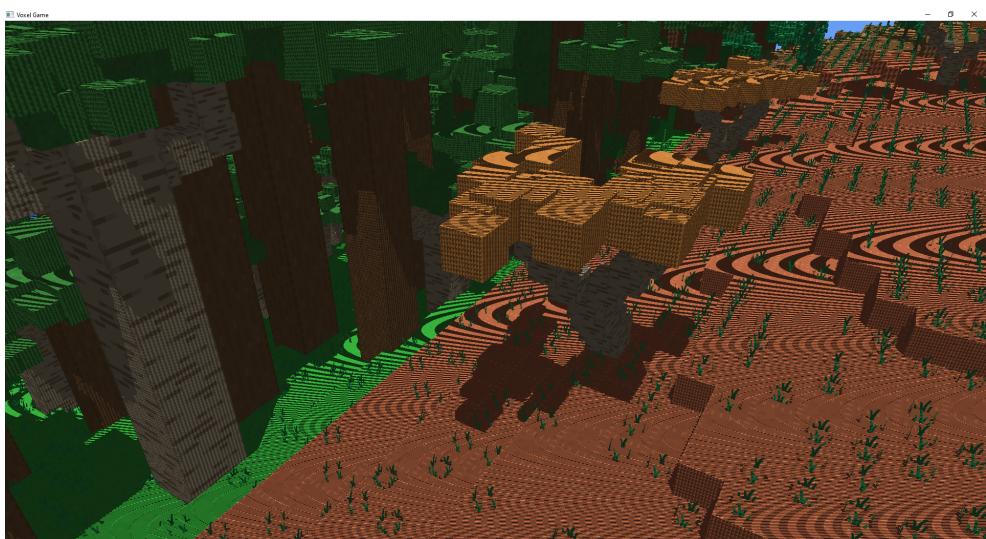
## 3.3 Stínové akně

Viditelné artefakty vznikají na površích, které by neměly mít stín. Problém ilustruje diagram 3.3, kde je mapa stínů promítnuta na vodorovný povrch. Každá šipka představuje jeden paprsek světla – texel mapy stínů – dopadající na povrch. Omezené rozlišení mapy stínů má za důsledek, že několik fragmentů může číst hodnotu hloubky ze stejného texelu. Fragmenty nacházející se nad žlutou křivkou nemají stín, fragmenty nacházející se pod ní stín mají.

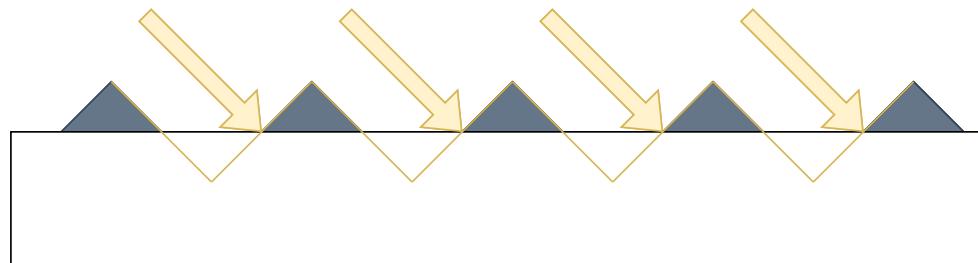
K tomuto problému dochází v případě, kdy světlo dopadá na povrch pod úhlem. V jednoduché scéně, kdy je slunce nad hlavou hráče, by k tomuto

### 3. MAPOVÁNÍ STÍNU

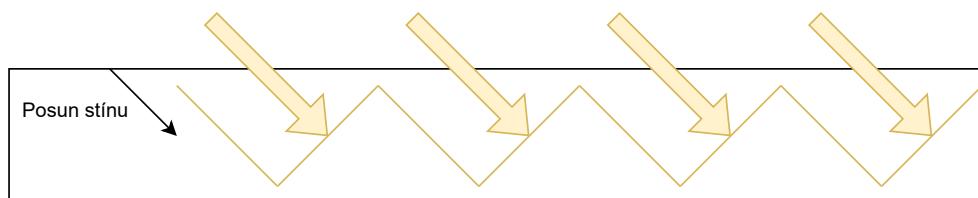
---



Obrázek 3.2: Stínové akně



Obrázek 3.3: Důvod vzniku stínového akné



Obrázek 3.4: Zamezení vzniku stínového akné

problému nedocházelo. Takové scény nejsou vizuálně zajímavé a značně by omezovali možnosti enginu.

Problém lze vyřešit posunutím mapy stínů (nebo povrchu objektu) tak, aby se texely mapy stínů nenacházely na povrchem objektu, viz diagram 3.4.

Pro posun mapy lze zvolit konstantu.

```
float bias = 0.0085;
```



Obrázek 3.5: Ostrá hrana stínu

```
float shadow = currentDepth - bias > closestDepth ? 1.0 : 0.0;
```

Toto řešení funguje pro světlo dopadající na povrch pod stejným úhlem. Pokud se změní zdroj světla nebo orientace objektu, nemusí být konstanta do statečně velká a problém se bude opakovat. Robustnějším řešením je vypočítat odchylku – **bias** – podle úhlu dopadu světla, který bude největší pro světlo dopadající na povrch pod ostrým úhlem a nejmenší pro světlo dopadající kolmo.

```
float bias = max(0.0085 * (1.0 - dotLightNormal), 0.00085);
```

## 3.4 Ostré stíny

Při pohledu na scénu 3.5 lze odhalit nedostatečné rozlišení mapy stínů. Několik vykreslovaných fragmentů se namapuje na jeden texel z mapy stínů. Výsledkem jsou zubaté hrany stínu s velmi ostrým přechodem. Řešení, které nevyžaduje zvětšení rozlišení je procentuálně bližší filtrování (percentage-closer filtering – PCF).

Technika PCF produkuje jemnější stíny. Ty se jeví méně hranaté a zubatý efekt není tak výrazný. PCF zahrnuje více způsobu filtrování, jehož základem je vícenásobné čtení vzorků z textury stínu a jejich zprůměrování. Jednoduchá technika je čtení hodnot ve čtverci kolem zpracovávaného texelu.

```
float shadow = 0.0;
vec2 texelSize = 1.0 / textureSize(texture_shadow, 0);
for(int x = -1; x <= 1; ++x) {
    for(int y = -1; y <= 1; ++y) {
```

### 3. MAPOVÁNÍ STÍNU

---



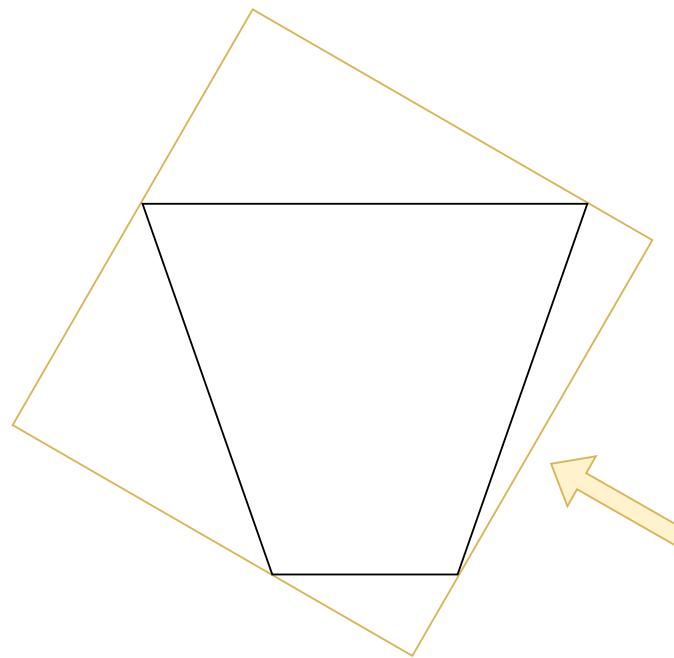
Obrázek 3.6: Měkká hrana stínu

```
float pcfDepth = texture(
    texture_shadow,
    projCoords.xy + vec2(x, y) * texelSize).r;
shadow += currentDepth - bias > pcfDepth ? 1.0 : 0.0;
}
shadow /= 9.0;
```

Scéna 3.6 je vykreslena při použití 9 vzorků. Při pohledu z dálky lze pozorovat lepší výsledky, zuby stínů nejsou tolik patrné, jako při použití jednoho vzorku. Při bližším pohledu na stébla trávy, lze pozorovat rozšíření stínu o jeden texel. Štíhlé textury již nevrhají přesné stíny, výměnou za zlepšení pohledu z dálky, kdy byl vrhaný stín nepřesný (velikost texelu neodpovídala velikosti textury) a byla vykreslena pouze jeho část s velice ostrým přechodem.

## 3.5 Pozice mapy stínů na scéně

Mapa stínů musí být posouvána podle pozice kamery v herním světě. Objekty jsou generovány kolem postavy hráče a pokud se hráč posune od počátku světa, objekty budou smazány a nemohou vrhat stíny. Mapa stínů má však omezené rozlišení a stínováním objektů, které jsou vygenerované, ale hráč je nemá šanci vidět (jsou například za ním), se snižuje kvalita stínů. Pro maximální využití dostupného rozlišení stačí pokrýt pouze objekty, které může hráč v daném snímku vidět. Viditelná část světa je určena maticí pohledu (view) a maticí projekce (projection). Tyto matice tvoří frustum, ve kterém



Obrázek 3.7: Frustum světla okolo frusta kamery

budou objekty viditelné. Toto frustum je potřeba obsáhnout frustum světla s co nejmenšími přesahy. Tato skutečnost ve 2D světě, je ilustrována na obrázku 3.7.

### 3.5.1 Souřadnice frusta kamery

Matice pohledu a projekce transformují souřadnice světa na normalizované souřadnice zařízení (NDC – normalized device coordinates). Každá souřadnice na jedné ze tří os NDC nabývá hodnot  $[-1, 1]$ . Do souřadnic světa je lze převést vynásobením inverzí matic pohledu a projekce [17].

$$\begin{aligned} v_{NDC} &= M_{proj} * M_{view} * v_{world} \\ v_{world} &= (M_{proj} * M_{view})^{-1} * v_{NDC} \end{aligned}$$

Pro vytvoření frusta světla, je nutné zjistit souřadnice rohů frusta kamery. Díky nim je možné přesně zjistit jakou část světa má frustum světla pokrývat. Souřadnice jsou vypočítané následujícím způsobem:

```
std::vector<glm::vec4> Helpers::Math::FrustumCornersWorldSpace(
    const glm::mat4& proj, const glm::mat4& view) {
    const auto inv = glm::inverse(proj * view);
```

### 3. MAPOVÁNÍ STÍNU

---

```
    std::vector<glm::vec4> frustumCorners;
    for (unsigned int x = 0; x < 2; ++x) {
        for (unsigned int y = 0; y < 2; ++y) {
            for (unsigned int z = 0; z < 2; ++z) {
                const glm::vec4 pt =
                    inv * glm::vec4(
                        2.0f * x - 1.0f, 2.0f * y - 1.0f,
                        2.0f * z - 1.0f, 1.0f);
                frustumCorners.emplace_back(pt / pt.w);
            }
        }
    }

    return frustumCorners;
}
```

#### 3.5.2 Matice pohledu a projekce světla

Pro výpočet matice pohledu světla je potřeba zjistit bod, který je ve středu stínované oblasti. Tento bod se nachází ve středu frusta kamery. A je ho možné získat zprůměrováním pozic rohů frusta.

```
glm::vec3 Helpers::Math::FrustumCenter(
    const std::vector<glm::vec4>& corners) {
    auto center = glm::vec3(0.0f);
    for (const auto& v : corners) {
        center += glm::vec3(v);
    }
    center /= corners.size();

    return center;
}
```

Matice pohledu je získána pomocí funkce `glm::lookAt` a směru světla.

```
const auto lightView = glm::lookAt(
    center,
    center + lightDir,
    glm::vec3(0.0f, 1.0f, 0.0f));
```

Pro stínování scény je použito globální světlo. K výpočtu matice projekce bude tedy použita funkce `glm::ortho` zaručující ortografickou projekci. Pro výpočet je nutné zjistit parametry `left`, `right`, `bottom`, `top`, `zNear` a `zFar`.

Při pohledu od zdroje světla bude jeho frustum osově zarovnaný kvádr, těsně objímající frustum kamery. Rohy kamery lze pomocí matice pohledu světla transformovat do souřadnicového systému pohledu světla. Z těchto

transformovaných rohů je vybráno maximum a minimum v každé ose, definující frustum světla.

```
for (const auto& v : corners) {
    const auto trf = lightView * v;
    minX = std::min(minX, trf.x);
    maxX = std::max(maxX, trf.x);
    minY = std::min(minY, trf.y);
    maxY = std::max(maxY, trf.y);
    minZ = std::min(minZ, trf.z);
    maxZ = std::max(maxZ, trf.z);
}
```

Před vytvořením matice projekce je potřeba upravit proměnnou `minZ` a `maxZ`, představující blízkou a vzdálenou plochu frusta. S aktuální hodnotou by stíny vrhaly pouze objekty viditelné hráčem. Například stín koruny stromu by se objevil pouze, pokud by ji hráč viděl.

Posunuté hranice frustra by měly pokrývat všechny objekty na scéně. Příliš velkorysým posunutím však dojde ke ztrátě přesnosti mapy stínů. Hranice by tedy měly být posunuté podle aktuální scény, například aby byl vykreslen stín pod nejvyšším stromem, u jehož paty hráč stojí.

[možna zahodit](#)

```
if (minZ < 0)
    minZ *= zMult;
else
    minZ /= zMult;
if (maxZ < 0)
    maxZ /= zMult;
else
    maxZ *= zMult;

const glm::mat4 lightProjection =
    glm::ortho(minX, maxX, minY, maxY, minZ, maxZ);

return lightProjection * lightView;
```

## 3.6 Kaskádové mapování stínů

Jednoduché mapování stínů má podstatnou nevýhodu. Pokud chceme rozšířit oblast pokrytu stíny, musíme zvýšit rozlišení mapy stínů. V opačném případě by stíny blízko hráče byly rozkostičkované. Tímto zvyšováním kvality stínů je masivně zatěžováno GPU. Stíny, které jsou vzdálenější od hráče budou mít stejnou kvalitu, jako ty mu blízké. Toto je zbytečné, vzdálenější stíny mohou mít horší kvalitu – hráč bude rozlišovat pouze jejich tvar a existenci, drobné

### 3. MAPOVÁNÍ STÍNU

---

detaily jsou zbytečné. Tento problém řeší kaskádové mapování stínů, skládající se z následujících kroků [17].

1. Rozděl frustum kamery na  $n$  subfrust, kde vzdálená plocha frusta  $i$  je blízká plocha frusta  $i + 1$ .
2. Pro každé frustum spočítej matici prostoru světla.
3. Vykresli mapu stínů pro každé frustum.
4. Předej všechny mapy stínu fragment shaderu.
5. Vykresli scénu, kde podle vzdálenosti fragmentu vyberes patřičnou mapu stínů.

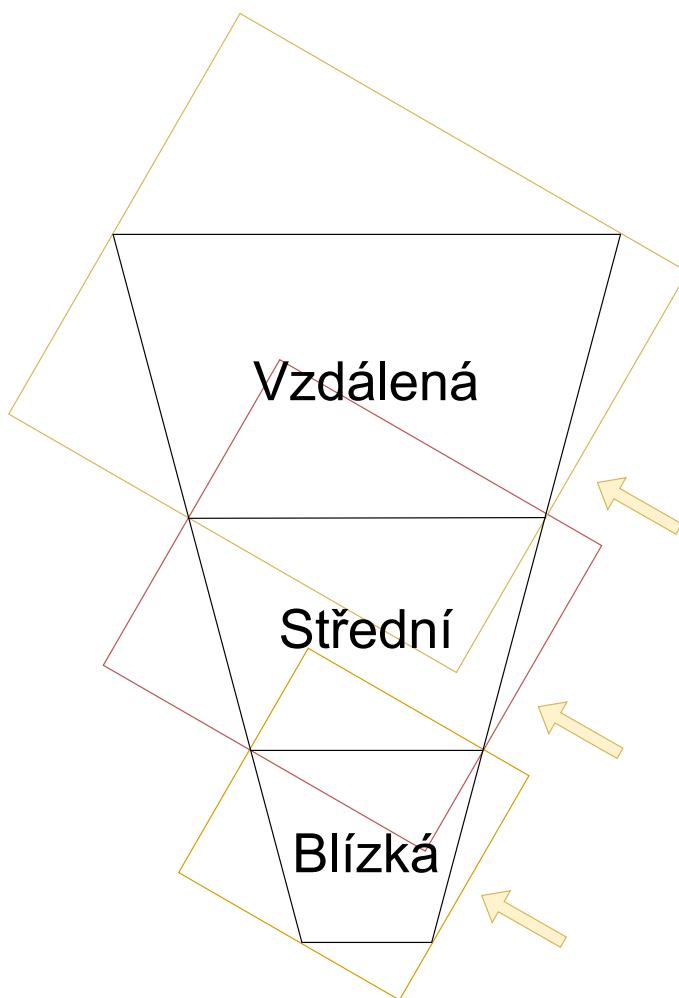
Na obrázku 3.8 je zobrazeno rozdelení frusta kamery na tři části. Velikost částí se zvětšuje s narůstající vzdáleností od blízké plochy kamery. Rozlišení však zůstává stejné, a proto jeden texel mapy stínů bude pokrývat větší plochu světa. Kvalita stínů se proto bude zhoršovat s rostoucí vzdáleností od hráče. Frustum lze rozdělit na libovolný počet částí. S větším počtem dělení, se snižuje viditelnost změny kaskády.

#### 3.6.1 Mapy stínů

Pro práci s více texturami, mající stejný rozměr, OpenGL poskytuje pole 2D textur [18]. Práce s polem textur usnadní práci (navázání pouze jedné textury) a umožní dynamické měnění počtu textur. Vytvoření 3D textury [19] probíhá obdobně jako vytvoření 2D textury. Jedním z rozdílů je zadání počtu textur, pro alokaci paměti grafické karty.

K vykreslení do pole textur se používá technika vrstveného vykreslování. V ní je za pomoci geometry shaderu vytvořena nová geometrie pro každou vrstvu map stínů. Pro každou matici světla z kaskády je vytvořen nový trojúhelník, který je pomocíní transformován, a nastaveno číslo vrstvy.

```
void main() {
    for (int i = 0; i < 3; ++i) {
        gl_Position = lightSpaceMatrices[gl_InvocationID] *
                      gl_in[i].gl_Position;
        gl_Layer = gl_InvocationID;
        TexCoord = gs_in[i].TexCoord;
        EmitVertex();
    }
    EndPrimitive();
}
```



Obrázek 3.8: Rozdělení frusta kamery na tři části

Pro vykreslení stínů je použit shader vykreslující mapu stínů s úpravami umožňujícími práci s mapami stínů. Pro zvolení správné matice světla je vypočítána hodnota hloubky fragmentu (v prostoru pohledu kamery), která je následně porovnána se vzdáleností nejbližší změny kaskády.

```

vec4 fragPosViewSpace = view * vec4(fragPosWorldSpace, 1.0);
float depthValue = abs(fragPosViewSpace.z);

int layer = CASCADE_COUNT - 1;
for (int i = 0; i < CASCADE_COUNT - 1; ++i) {
    if (depthValue < cascadePlaneDistances[i]) {
        layer = i;
        break;
    }
}

```

### 3. MAPOVÁNÍ STÍNU

---



Obrázek 3.9: Velikost stínového akné v závislosti na kaskádě

```
    }
}
vec4 fragPosLightSpace =
    lightSpaceMatrices[layer] * vec4(fragPosWorldSpace, 1.0);
```

Na obrázku 3.9 lze vidět tři kaskády. V první nedochází k stínovému akné, v druhé je již při bližším pozorování patrné a ve třetí je velice výrazné. K tomuto efektu dochází, protože texel v každé vrstvě pokrývá různě velkou plochu vykreslované geometrie. Proto je potřeba volit odchylku na základě vrstvy. Škálování odchylky inverzní hodnoty vzdálené plochy frusta, produkuje výsledky bez stínového akné.

```
float bias = max(0.0085 * (1.0 - dotLightNormal), 0.00085);
if (layer == cascadeCount) {
    bias /= farPlane * 0.008;
}
else {
    bias /= cascadePlaneDistances[layer] * 0.02;
}
```

Tato technika zavádí větvení, které v shaderech není chtěné. Zároveň neumožnuje dokonalé vylazení odchylky pro jednotlivé kaskády. Odchylka je pevně svázaná s počtem a vzdáleností kaskád. Její výpočet proto může být přesunut na CPU, k definici kaskád. Odchylky jsou předány shaderu pomocí pole a načteny následujícím způsobem.

```
float bias = max(0.0085 * (1.0 - dotLightNormal), 0.00085);
bias *= cascadeBiases[layer];
```

### 3.6.2 Změna počtu kaskád

Engine umožňuje měnit počet kaskád při běhu programu. Shadery proto musí mít alokované dostatečně velké místo, pro uložení všech matic. Jedinou výjimkou je geometry shader, ve kterém je specifikován počet vyvolání.

```
#define CASCADE_COUNT 3
layout(triangles, invocations = CASCADE_COUNT) in;
layout(triangle_strip, max_vertices = 3) out;
```

Pokud je `CASCADE_COUNT` menší, než počet kaskád  $n$ .  $n - 1$  nejvzdálenějších kaskád nebude vykresleno. Pokud je větší, bude plýtván výpočetní čas GPU. Počet kaskád shaderu nemůže být předán parametrem – vyžaduje běh shaderu. Engine proto umožňuje znova zkompilovat shadery za běhu a upravit hodnotu `maker`. Zdrojový kód je po načtení z disku projet a všechna makra nahrazena specifikovanou hodnotou, kód je následně přeložen. Opakovaný překlad je proveden při změně počtu kaskád.

```
ShaderDepth_ = ResourceManager::SetShaderMacros(
    "shadow_csm", {{ "CASCADE_COUNT", std::to_string(levels) }});
```

Makra jsou použita i ve zbylých shaderech, umožňující alokaci přesného počtu prvků v poli matic světla, kaskád, atd.

## 3.7 Optimalizace vykreslování

Po zapnutí kaskádového stínování došlo k viditelnému poklesu FPS. K měření výkonu a následné optimalizaci byla vytvořena třída `GpuTimer`, umožňující měřit délku operací probíhajících na GPU. Požadavky pro vykreslení na grafické kartě jsou asynchronní. Synchronizace CPU a GPU ale, probíhá jen v určitých bodech programu, např. výměna bufferů. Pro měření jednotlivých vykreslovacích volání, proto musí být do CPU kódu umístěna synchronizační bariéra – aktivní čekání. Od verze OpenGL 3.3 je možné využít dotazy na dobu běhu [20]. Vygenerování dotazů probíhá zavoláním funkce:

```
glGenQueries(2, QueryId_);
```

Aktuální hodnota časovače je zjištěna dotazem:

```
glQueryCounter(QueryId_[0], GL_TIMESTAMP);
```

### 3. MAPOVÁNÍ STÍNU

---

Čas je zaznamenán po doběhnutí všech předchozích volání [21]. Před vyčtením hodnoty časovače musí CPU aktivně čekat, dokud není výsledek dotazu k dispozici. A následně výsledky dotazu načíst.

```
glQueryCounter(QueryId_[1], GL_TIMESTAMP);

int available = 0;
while (!available) {
    glGetQueryObjectiv(
        QueryId_[1], GL_QUERY_RESULT_AVAILABLE, &available);
}
long long start, stop;

glGetQueryObjecti64v(QueryId_[0], GL_QUERY_RESULT, &start);
glGetQueryObjecti64v(QueryId_[1], GL_QUERY_RESULT, &stop);

return stop - start;
```

Všechna následující měření probíhala po načtení totožné scény, s kamerou směřující stejným směrem, zprůměrováním prvních 1000 hodnot časovače.

#### 3.7.1 Průhledné textury

Pro správné vykreslení stínů průhledných textur musí fragment shader pracovat s texturou objektu. Načíst zní hodnotu průhlednosti. A zapsat do hloubkového bufferu pouze, pokud je fragment neprůhledný.

```
float alpha = texture(texture_diffuse1, TexCoord).a;
if (alpha < 0.05)
    discard;

gl_FragDepth = gl_FragCoord.z;
```

Čtením z textury u fragmentu, který nemá průhlednost je ztrácen výkon. Další optimalizace, které nemůže být použita, je brzký test hloubky (early depth test). Grafická karta může provést test hloubky před spuštěním fragment shaderu a vyřadit fragmenty, které nebudou viditelné. Použitím klíčového slova `discard` a zapsáním do hloubkového bufferu je tento test vypnut [22].

Pro změření výkonu byl vytvořen prázdný fragment shader s explicitně zapnutým brzkým testem hloubky.

```
layout(early_fragment_tests) in;

void main() {}
```

Vykreslení mapy stínů s kontrolou průhlednosti trvalo v průměru 31,52 ms. Při použití prázdného shaderu na všechny objekty, trvalo vykreslení 23,58 ms. Optimalizace přináší očekávané výsledky. Při použití obou shaderů, pro průhledné a neprůhledné objekty, došlo k zhoršení času na 24,65 ms. Tento přístup však produkuje korektní stíny a díky nízkému procentu průhledných objektů na scéně (celkový počet objektů byl 159 021 z toho 22 613 průhledných), přináší významné zrychlení.

### 3.7.2 Ořezání scény

Doposud byly grafické kartě předávány všechny objekty na scéně. Ztráta výkonu byla omezena díky ořezávacím testům, probíhajícím před spuštěním výpočetního fragment shaderu. Testy z vykreslovacího řetězce vyřadí všechny fragmenty, které se nacházejí mimo prostor obrazovky.

Při vykreslování map stínů je použit geometry shader, generující nová primitiva pro každou kaskádu. Tyto primitiva nejsou automaticky ořezána grafickou kartou, jako v případě fragment shaderu. Dochází tedy ke generování primitiv, které hráč nemůže vidět a ani nebudou vrhat stín na viditelnou plochu.

Engine proto musí provést ořezání objektů, před jejich předáním grafické kartě. Testování je prováděno na úrovni chunků. Pokud je část chunku viditelná hráčem, pak je uložen ukazatel na jeho obsah, který je následně předán grafické kartě. Situaci ilustruje obrázek 3.10, kde jsou viditelné chunky zvýrazněny.

Při testu viditelnosti je každý roh chunku transformován do prostoru kamery. Pokud je bod viditelný, jeho souřadnice na ose x musí být z intervalu  $[-1, 1]$ . Jeho souřadnice na ose z musí být menší než 1 (nacházet se před blízkou plochou kamery). Souřadnice na ose y nejsou kontrolovány, protože roh chunku se může nacházet pod spodní rovinou vymezující frustum, ale jeho objekty budou přesto viditelné – například stromy. Kontrola zda je  $y < 1$  by způsobovala mizení chunků při sklonění kamery.

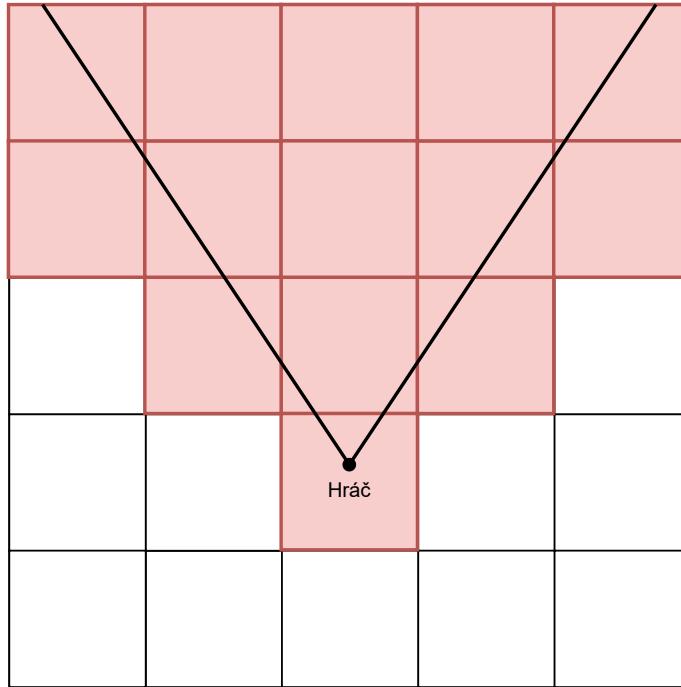
```
bool Scene::IsPointInView(
    const glm::vec3& position, const glm::mat4& projView) {
    auto pt = projView * glm::vec4(position, 1.0f);
    pt /= pt.w;

    return pt.x >= -1.0f && pt.x <= 1.0f && pt.z <= 1.0f;
}
```

Při pouhé kontrole rohů by mohlo dojít k ořezání chunků, jež se nachází před hráčem, ale žádný jejich roh není viditelný. K tomu může dojít, pokud má hráč úzké zorné pole, nebo se divá směrem k zemi pod dostatečným úhlem. Situace, kdy se hráč dívá kolmo k zemi je ilustrována na obrázku 3.11. Viditelná část herní plochy je vymezena červenými přímkami. Ke kontrole je

### 3. MAPOVÁNÍ STÍNU

---



Obrázek 3.10: Ořezání neviditelných chunků

předán bod, nacházející se na hranici chunků – vždy je vybrána hranice bližší k hráči – a pozici x rovné pozici hráče na ose x, respektive ose z, pokud je hranice rovnoběžná s osou z.

Pozice rohu na ose y je nejnižší bod v chunku. Pokud se hráč podívá vzhůru, všechny rohy v chunku budou oříznuty blízkou plochou kamery. Z tohoto důvodu je omezen maximální náklon kamery, který je použit při výpočtu matice pohledu. Náklon je oříznut na interval  $[-90^\circ, 0^\circ]$  vůči vodorovné ploše.

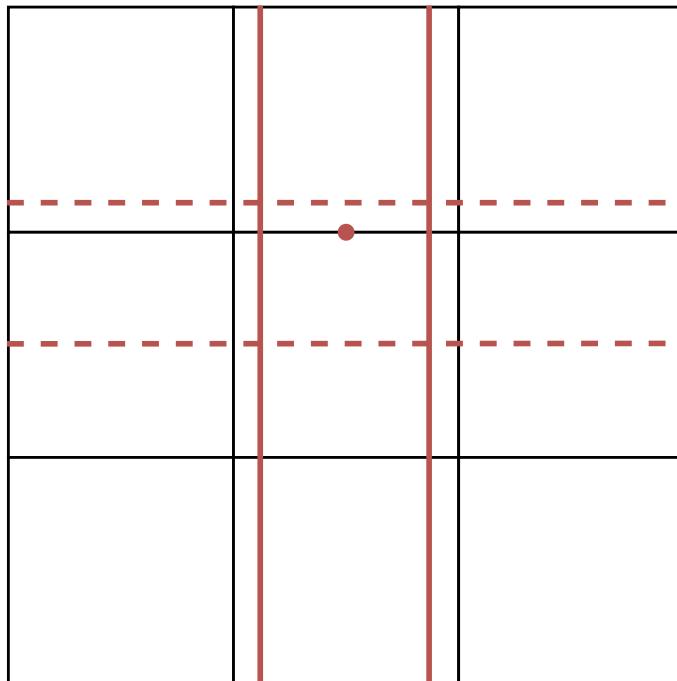
Ořezáním chunků scény byl omezen počet objektů, předávaných grafické kartě ze 159 021 na 43 605. Díky tomu se snížil čas potřebný k vykreslení map stínů z 24,65 ms na 7,22 ms.

## 3.8 Výsledky

Scéna na obrázku 3.12 je vykreslena pomocí kaskádového mapování stínů. Lze si všimnout dvou změn kaskád a zhoršení kvality stínů<sup>4</sup>. Rozlišení kaskád je nastaveno na 1024 x 1024 pixelů. S tímto rozlišením je možné pokrýt celou herní scénu v dostatečné kvalitě. Pro pokrytí celé scény v uspokojivé kvalitě, bylo potřeba zvýšit rozlišení jednoduché mapy stínů alespoň na 4096 x

---

<sup>4</sup>Kaskády byly pro viditelnější efekt posunuty k blízké ploše frusta kamery.



Obrázek 3.11: Ořezání chunků při pohledu kolmo dolů

4096 pixelů. Ztrátu kvality však bylo stále možné pozorovat při přiblížení se k objektu.

Měření ukázala, že použitím CSM, nedochází k vysoké ztrátě výkonu. Vykreslení jednoho snímku trvalo 37,92 ms, oproti 35,31 ms při použití jednoduché mapy. Kaskádové mapování lze zároveň lépe škálovat. Přidáním dvou kaskád se výrazně zvýšila kvalita stínů v blízkosti hráče. Vykreslení snímku trvalo 41,78 ms. Zvýšením rozlišení jednoduché mapy na 8192 x 8192 pixelů, nedošlo k dostatečnému zlepšení stínů v blízkosti hráče. A vykreslení snímku trvalo déle (68,47 ms) než při použité pěti kaskád. Použití CSM se tedy jeví jako lepší řešení.

### 3. MAPOVÁNÍ STÍNU



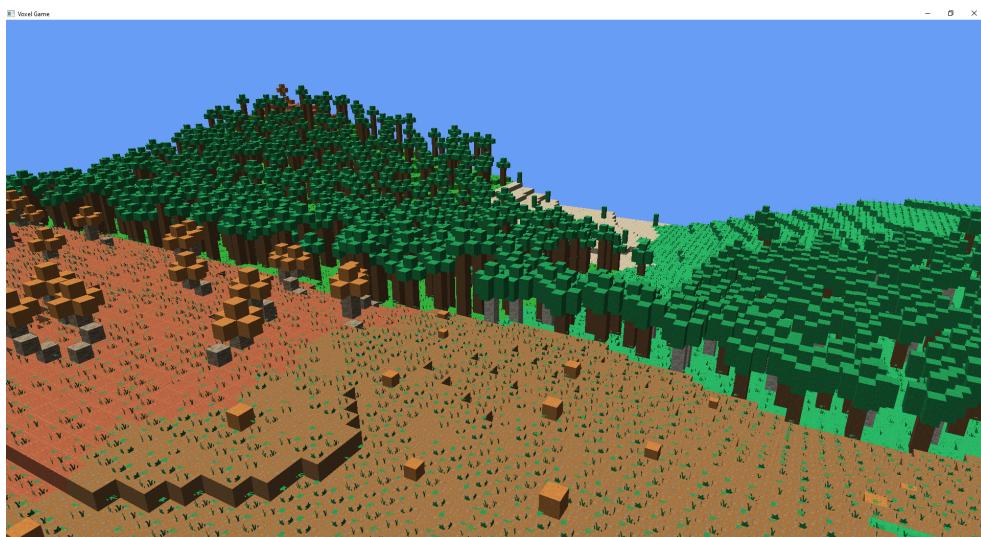
Obrázek 3.12: Výsledek použití kaskádového mapování stínů

# KAPITOLA 4

## Modelování rostlin s využitím L-systémů

Herní svět obsahuje velké množství vegetace a ač jsou si všechny stromy, keře typově podobné, hráč si velice rychle všimne, že jsou identické. Na obrázku 4.1 vidíme tři druhy stromů, které jsou zkopiované po scéně. Stromy v levé zadní části mají různou výšku, přesto působí umělým dojmem.

Stromy v reálném světě jsou si podobné – rozeznáváme jednotlivé druhy stromů, přesto neexistují dva stejné stromy. Pokud druh stromu zapíšeme formální gramatikou nazývanou L-systém, docílíme podobné struktury stromů, které se budou lišit v detailech.



Obrázek 4.1: Identické stromy

## 4.1 L-systém

L-systém nebo také Lindenmayerův systém je paralelní přepisovací systém vyvinutý maďarským teoretickým biologem a botanistou Aristidem Lindenmayerem v roce 1968. L-systém je typ formální gramatiky skládající se z abecedy, přepisovacích pravidel a počátečního axiomu. Pomocí postupného derivování počátečního axiomu je možné simulovat vývoj rostliny v čase [23].

## 4.2 Interpretace řetězců pomocí želvy

Řetězce lze graficky reprezentovat pomocí želvy, konzumující symboly abecedy. Každý symbol určuje akci, kterou má želva vykonat. Želva se může pohybovat ve 2D nebo 3D prostoru. Ve 2D si můžeme interpretaci představit jako želvu, držící tužku, pohybující se po papíře.

Želvu lze reprezentovat jako trojici  $(x, y, \alpha)$ , kde  $(x, y)$  představuje kartézské souřadnice reprezentující polohu v prostoru a  $\alpha$  úhel kam želva směruje. Zadáním délky kroku  $d$  a změny úhlu  $\delta$  lze želvu ovládat pomocí následujících symbolů.

- F — Posun dopředu o délku  $d$ . Stav želvy se změní na  $(x', y', \alpha)$ , kde  $x' = x + d \cos \alpha$  a  $y' = y + d \sin \alpha$ . Mezi body  $(x, y)$  a  $(x', y')$  je nakreslena čára.
- + — Rotace doleva o úhel  $\delta$ . Nový stav želvy  $(x, y, \alpha + \delta)$ .
- - — Rotace doprava o úhel  $\delta$ . Nový stav želvy  $(x, y, \alpha - \delta)$ .

Nechť je definován následující L-systém. Bud'  $\omega$  počáteční axiom,  $p$  přepisovací pravidlo,  $\delta = 90^\circ$  a  $d$  zmenšené čtyřnásobně pro každý obrázek [24].

$$\begin{aligned}\omega &: F - F - F - F \\ p &: F \rightarrow F - F + F + FF - F - F + F\end{aligned}$$

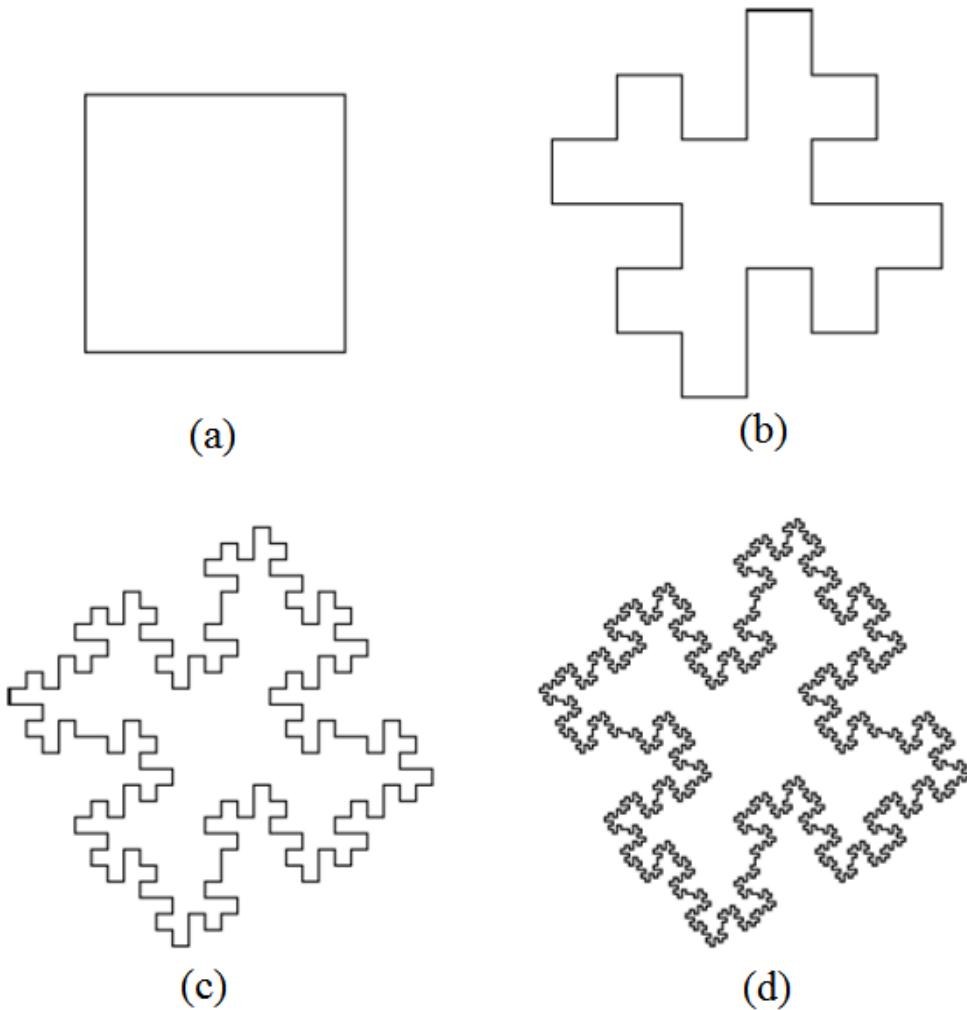
Želva interpretující daný L-systém generuje kvadratické Kochovy ostrovy 4.2. Obrázky jsou vygenerovány derivacemi o délce 0 až 3.

## 4.3 Větvení v L-systémech

S danými přepisovacími pravidly není možné generovat větvící se struktury. Želva vždy pokračuje od své poslední pozice. Říše rostlin je dominovaná větvícími se strukturami, potřebujeme proto matematické vyjádření této skutečnosti.

Větvení v řetězci můžeme reprezentovat pomocí dvou symbolů [ a ], kde [ značí začátek větve a ] konec větve [25].

Symboly jsou interpretovány želvou následovně:

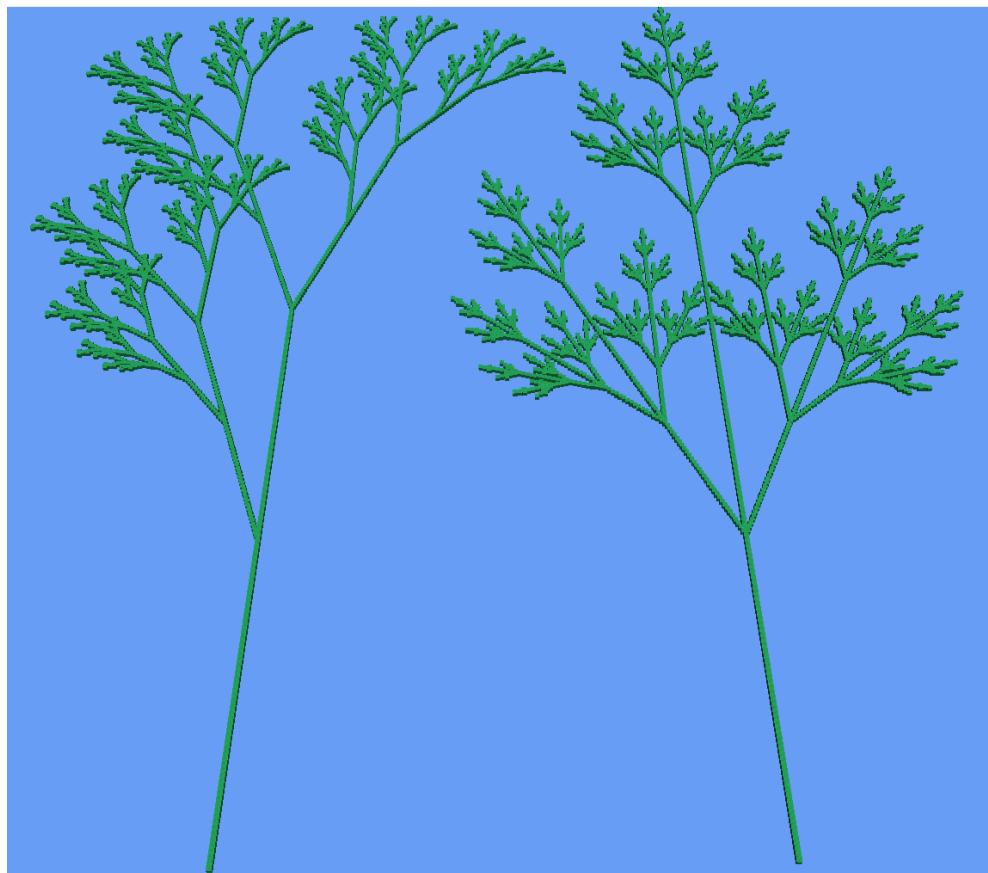


Obrázek 4.2: Kvadratické Kochovy ostrovy

- [ – Ulož atributy želvy do zásobníku.
- ] – Načti atributy želvy ze zásobníku a smaž je z vrcholu zásobníku (operace pop). Při této operaci není nakreslená žádná čára.

Díky nově přidaným symbolům lze generovat struktury připomínající rostliny. Struktury na obrázku 4.3 jsou generované následujícími L-systémy:

1.  $\delta = 20^\circ$
- $\omega : E$
- $p1 : F \rightarrow FF$
- $p2 : E \rightarrow F[+E]F[-E] + E$



Obrázek 4.3: Struktury připomínající rostliny generované pomocí závorkovaného systému

$$2. \delta = 25,7^\circ$$

$$\omega : E$$

$$p1 : F \rightarrow FF$$

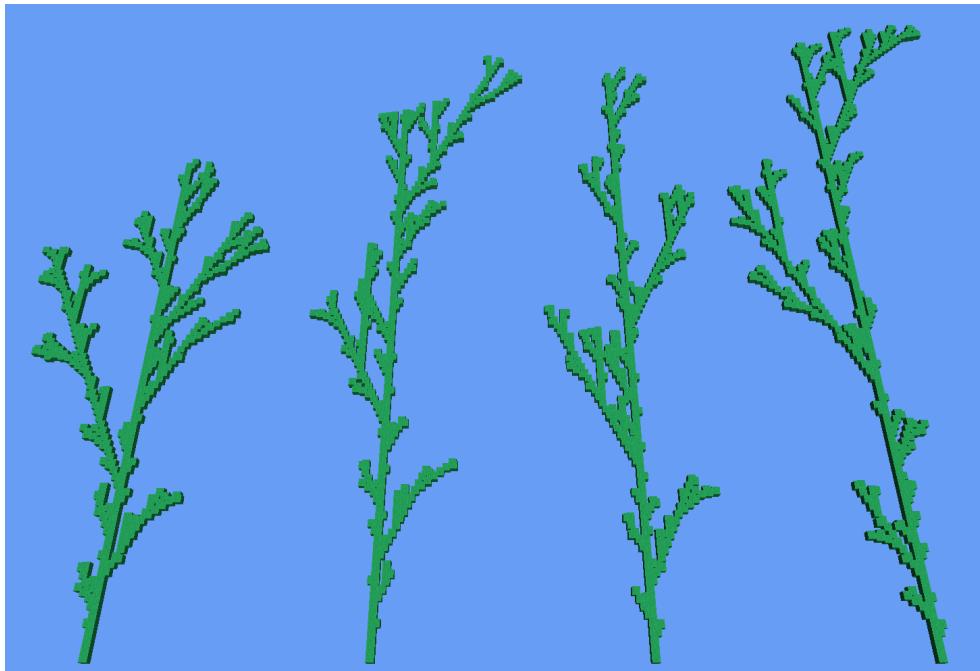
$$p2 : E \rightarrow F[+E][-E]FE$$

L-systém 1 generuje rostlinu vlevo, L-systém 2 generuje rostlinu vpravo.

#### 4.4 Stochastické L-systémy

Rostliny generované deterministickým L-systémem jsou všechny stejné. Jejich použití ve scéně by vytvářelo stejný efekt, který je popsán na začátku kapitoly.

K předejití tohoto efektu je nutné zavést variace v rámci druhu. Náhodná interpretace řetězce má limitované využití. Změna úhlu větvení, šířky a výšky



Obrázek 4.4: Využití stochastického L-systému

segmentů rostliny zachovávají topologii struktury, ze které je generovaná. Stochastické L-systémy mohou měnit topologii struktury [26].

L-systém, který byl do též používán, nemohl mít více přepisovacích pravidel pro stejný symbol abecedy. Pokud má stochastický L-systém vice přepisovacích pravidel, je z nich vybráno jedno s pravděpodobností  $1/n$ , kde  $n$  je počet přepisovacích pravidel pro daný symbol abecedy<sup>5</sup>.

Scéna 4.4 byla vygenerována za pomocí stochastického L-systému, kde:

$$\delta = 25, 7^\circ$$

$$\omega : F$$

$$p1 : F \rightarrow F[+F]F[-F]F$$

$$p2 : F \rightarrow F[+F]F$$

$$p3 : F \rightarrow F[-F]F$$

---

<sup>5</sup>Tato definice se liší, od definice uvedené ve [26]. Tento způsob náhodného výběru je použit v implementaci, kde pravděpodobnostní distribuci zastupuje několikanásobné zopakování přepisovacího pravidla.

## 4.5 Implementace

### 4.5.1 Formát L-systému

L-systém může být načten ze souboru pomocí třídy `LSystemParser`. L-systém musí mít následující formát:

```
yaw_angle pitch_angle shring_ratio
axiom
letter > production
.
.
.
letter > production
```

Soubor může obsahovat za sebou jdoucí L-systémy. `LSystemParser` je vrátí jako pole. Soubor může obsahovat komentáře na nových řádcích, začínající symbolem `#`.

Třída `LSystem` obsahuje gramatiku a tři atributy specifikující úhel náklonu podle osy y (yaw), podle osy x (pitch) a změnu velikosti bloku. Poslední atribut je využit při rozvětvení rostliny. Potomci mateřské větve by se měly řídit postulátem Leonarda da Vinci: „Všechny větve stromu, v každé úrovni jeho růstu, jsou v součtu jejich tloušťky rovné tloušťce kmene pod nimi.“ V případě dvojitého rozvětvení, tloušťky mateřské větve  $w_1$ , tloušťky potomků  $w_2$  dostaneme rovnici [27]:

$$w_1^2 = 2w_2^2$$

$$\frac{w_2}{w_1} = \frac{1}{\sqrt{2}} \approx 0,707$$

Hodnotu 0,7 je možné nalézt v L-systémech modelující keře.

### 4.5.2 Želva

Třída `Turtle` rozšiřuje pohyb želvy – popsané v kapitole Interpretace řetězců pomocí želvy – do 3D prostoru. Vnitřní stav želvy určují následující atributy:

- Pozice v prostoru.
- Velikost bloku vytvořeného želvou.
- Barva použitá pro kreslení (výstupní pole).
- Yaw — rotace podle osy y.
- Pitch — rotace podle osy x.

Želva si udržuje tři navzájem kolmé směrové vektory (nahoru, dopředu, doprava) jednotkové délky, které využívá pro pohyb po scéně. Vektory jsou aktualizované po každé rotaci. Pro výpočet je nutné znát vektor směřující kolmo vzhůru vůči scéně (WORLD\_UP). Generovaný svět je plochý, proto lze tento vektor nahradit konstantním vektorem  $(0, 1, 0)$ .

```
glm::vec3 front;
front.x = cos(glm::radians(Yaw_)) * cos(glm::radians(Pitch_));
front.y = sin(glm::radians(Pitch_));
front.z = sin(glm::radians(Yaw_)) * cos(glm::radians(Pitch_));

Front_ = glm::normalize(front);
Right_ = glm::normalize(glm::cross(Front_, WORLD_UP));
Up_ = glm::normalize(glm::cross(Right_, Front_));
```

Délku  $x$  v rovině určené osami  $x$  a  $z$  lze spočítat jako délku přilehlé odvěsnky.  $\cos(Yaw) = x/h$ , kde  $h$  je délka přepony. Víme, že vektor má jednotkovou délku, proto  $h = 1$ . Stejný postup aplikujeme pro rovinu určenou osami  $x$  a  $y$ .

Tímto způsobem dopočítáme délky  $y$  a  $z$  vektoru směřujícího dopředu a normalizujeme ho. Jelikož jsou na sebe vektory kolmé, využijeme vektorového součinu, jehož výsledkem je vektor kolmý k oběma původním vektorům. Všechny vektory je nutné normalizovat, aby se předešlo jejich zkracování s tím, jak se Pitch blíží  $\pm 90^\circ$ .

K zamezení převrácení os jsou z definičního oboru Pitch vyjmuty násobky  $90^\circ$ .

```
if (Helpers::Math::Equal(cos(glm::radians(Pitch_)), 0.0f))
    Pitch_ -= 0.01f;
```

Výsledná nepřesnost je menší než maximální rozdíl dvou čísle typu float  $\epsilon$ , která jsou považována za stejná. Funkce Equal porovnává desetinná čísla s přesností  $\epsilon$ .

Želva vystavuje metody pro pohyb ve všech třech osách využívající vektorů Up\_, Right\_, Forward\_. K pozici želvy je přičten patřičný vektor naškálovaný délkou pohybu. Např.:

```
void LSystems::Detail::Turtle::MoveForward(float dz) {
    Position_ += Front_ * dz;
}
```

### 4.5.3 Rozšířená abeceda

Následující symboly abecedy mají speciální význam pro jejich interpretaci.

- U a u — Posuň želvu nahoru.

#### 4. MODELOVÁNÍ ROSTLIN S VYUŽITÍM L-SYSTÉMŮ

---

- F a f — Posuň želvu dopředu.
- x — Zmenší blok produkovaný želvou.
- X — Zvětši blok produkovaný želvou.
- S — Nastav původní velikost bloku produkovaného želvou.
- + — Rotuj želvu doleva podle osy y.
- - — Rotuj želvu doprava podle osy y.
- ^ — Rotuj želvu nahoru podle osy x.
- & — Rotuj želvu dolů podle osy x.
- [ — Ulož kopii želvy na vrchol zásobníku.
- ] — Vyjmi želvu z vrcholu zásobníku.
- 0 – 9 — Přepni výstupní pole.

L-systém může obsahovat jakýkoliv jiný ASCII symbol – mimo bílých znaků a # – určený pro expanzi přepisovacích pravidel. Není želvou interpretován.

##### 4.5.4 Ovládání želvy

Implementace želvy se nachází ve jmenném prostoru `LSystems::Detail`, uživatel by ji neměl využívat přímo, ale je pro něj připravena třída `LSystemExecutor` zajišťující generování herních objektů z poskytnutého L-systému.

`LSystemExecutor` umožňuje generovat herní objekty na základě stochastického L-systému – topologie struktury výsledného modelu se může měnit mezi jednotlivými voláními generátoru na základě parametru `salt`. L-systém lze náhodně interpretovat na základě těchto parametrů:

- Rozsah počtu provedených derivací.
- Variace v rotaci želvy. K úhlu, o který se má želva otočit, se přičte  $x * \text{původní úhel}$ , kde  $x$  je z  $[-\text{angleVar}, \text{angleVar}]$ . Defaultní hodnota `angleVar` je 0,2.
- Výchozí velikost generovaných objektů. Lze určit rozsahem korespondujícím s počtem derivací.

Přidání náhodného úhlu má výrazný efekt na organický vzhled rostliny. Na obrázku 4.5 lze vidět akáciové stromy vyznačující se plochou korunou. V definici L-systému jsou všechny listy ve stejné výšce, výsledný rozdíl ve výškách je způsoben opakováním rotováním želvy. V zápisu lze vidět, že se

## 4.5. Implementace



Obrázek 4.5: Nepravidelná koruna akáciového stromu

želva otočí o  $45^\circ$  nahoru (^), pokládá větve (u, U), skloní se o  $45^\circ$  (&) a pokládá listy (F). Listy by tak měly být ve stejné rovině, ale nejsou.

L-systém generující akácie:

```
45.0 45.0 0.8
# make sure the plant has splits
# random lenght stem - then split
^uA1S&F+F+F+F
U > uU
A > +uuE
A > -uE
A > +uE
A > -E
A > uuE
A > uE
# top of the plant
E > x [++++UE1S&F+F+F+F] +UE
E > x [++++++UE1S&F+F+F+F] ++UE
E > x [++UE1S&F+F+F+F] -UE
```

Výstupem generátoru je 2D pole obsahující herní objekty rozdělené podle čísla výstupního bufferu, který měla želva při generování. Část enginu je tak odstíněna od textur, které jsou definované v části procedurálního generátoru. Díky tomuto rozdělení modelu je možné snadno měnit textury pro jednotlivá pole. Procedurální generátor tohoto využívá a používá stejný model – jiný běh generátoru – pro vytváření bříz a dubů, lišících se texturou kmene a listů.



Obrázek 4.6: Akáciaový strom rostoucí v přírodě [28]

## 4.6 Modelování rostlin

Při modelování vegetace bylo třeba velkého množství pokusů a ladění, kdy rostlina nevypadala přirozeně, ale nebylo jasné, v jaké časti gramatiky je problém. Nejvíce se mi osvědčila technika nalezení reálné rostliny obrázek 4.6 a následné pokusy o její napodobení obrázek 4.7.

Velice se osvědčilo pravidlo:

$$U > uU$$

Díky němu jsou větve blíže k zemi delší, než větve navazující na korunu stromu. Pokud má rostlina význačné části je vhodné je modelovat samostatně (kmen, větve, koruna) viz L-systém generující akácie.

Tato technika ne vždy přinášela ovoce. Modelování trávy rostoucí na planinách se projevilo jako problém. Tráva neměla dostatečnou hustotu a nezapadala do kresleného vzhledu – obrázek 4.8. S navyšujícím se počtem herních objektů dramaticky rostla spotřeba paměti. Jeden herní objekt s texturou trávy byl nahrazen desítkami herních objektů, z kterých se skládal model trávy. Tento problém by mohl být řešen přesunem vytváření modelu na grafickou kartu, přidáním vertexů v geometry shaderu.

Výsledné modely svou topologií připomínaly strukturu keřů. Byly proto upraveny – přidáním listí, změnou větvení – a využity v generátoru keřů.

#### 4.6. Modelování rostlin

---



Obrázek 4.7: Model akáciového stromu



Obrázek 4.8: Tráva na planinách

#### 4. MODELOVÁNÍ ROSTLIN S VYUŽITÍM L-SYSTÉMŮ



Obrázek 4.9: Fáze růstu keře

##### 4.6.1 Simulace růstu

Křovinatý biotop je porostlý dvěma druhy keřů, rozdělených do třech fází růstu. Ty jsou simulovány opakováním derivování počátečního axiomu. Keře nejmenšího vzrůstu jsou derivovány 2x, největší keře jsou derivovány 4x. Tloušťka kmene koreluje lineárně s počtem provedených derivací. Větší stromy mají širší kmen a dorůstají vyšší výšky.

Každý druh si zachovává své typické vlastnosti. Na obrázku 4.9<sup>6</sup> lze pozorovat stejné zakončení větví – rozdelení do dvou větví rostoucích na opačné strany – a podobný úhel v jakém se větve oddělují od kmene. Topologie rostliny se díky stochastickému L-systému mění mezi jednotlivými jedinci.

Díky těmto krokům biotop obsahuje rostliny navzájem podobného vzhledu, lišících se v drobných detailech.

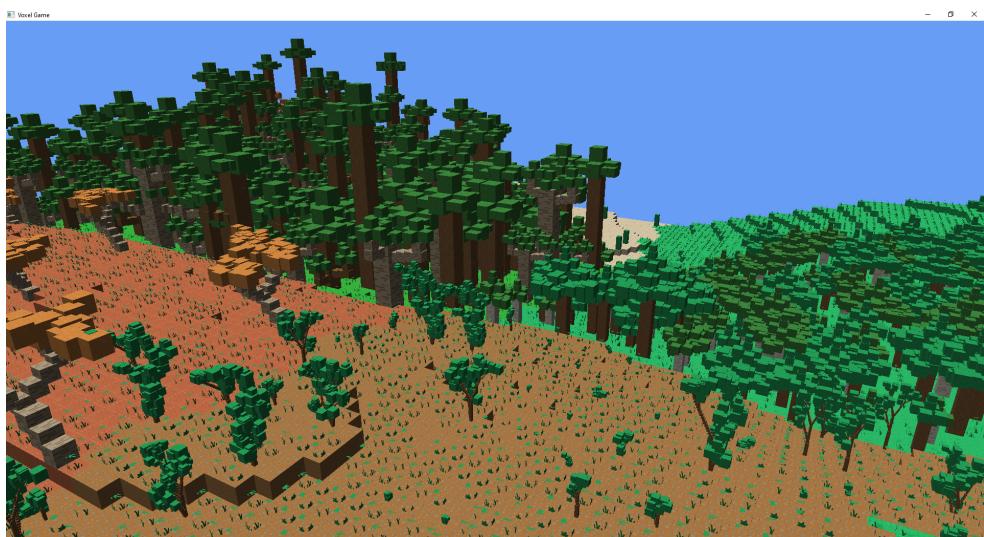
### 4.7 Výsledky použití L-systémů

Definice L-systémů jsou krátké ( $\approx 10$  řádků na jeden druh rostliny) a produkují velké množství rozdílných jedinců. Předchozí manuální definování rostlin v kódu bylo náročnější, a i při přidání více jedinců pro každý druh, by bylo snadné najít stejné. Odebráním definic rostlin z kódu se zvýšila jeho čitelnost – definice L-systémů jsou zdroje dat, který engine konzumuje. Nutnost komplikace při změně modelu byla odstraněna a zvýšila se rychlosť iterace, s kterou je možné upravovat model.

<sup>6</sup>Pro větší názornost byly odstraněny části modelu představující listy.

## 4.7. Výsledky použití L-systémů

---



Obrázek 4.10: Vegetace vygenerovaná na základě L-systémů

Vytvořením vlastního formátu pro zápis modelu rostliny se oddělila závislost na programovacím jazyce. Modely tak může vytvářet jiný člen týmu bez znalosti programování a překladu kódu.

Generováním rostlin za běhu programu se snížila rychlosť jeho běhu. Tento problém lze mitigovat cachováním rostlin obsahujících velké množství herních objektů. Toto bylo provedeno pro keře. Před spuštěním generace terénu je naplněn buffer obsahující keře vygenerované na základě seedu. Buffer musí být dostatečně velký na to, aby nedošlo ke snížení diverzity rostlin. Při vytváření keře na scéně je vybrán náhodný index do bufferu, závislý na pozici keře. Vybraný model je zkopirován a přesunut na dané místo.

Při porovnání scény 4.1 ze začátku kapitoly si lze všimnout přirozenějšího vzhledu krajiny. Koruny stromů se mohou překrývat, scéna díky tomu působí více organicky – stromy v přírodě nemají přesně stanové hranice, kde končí jeden a začíná druhý. Výsledná scenérie 4.10 působí méně jednolitě díky rozdílným vývojovým stádiím rostlin.



---

## Závěr



---

# Literatura

- [1] Návrhový vzor Game Loop [online]. [cit. 2022-01-12]. Dostupné z: <http://gameprogrammingpatterns.com/game-loop.html>
- [2] Learn modern OpenGL graphics programming in a step-by-step fashion [online]. [cit. 2022-01-16]. Dostupné z: <https://learnopengl.com/Getting-started>Hello-Triangle>
- [3] Learn modern OpenGL graphics programming in a step-by-step fashion [online]. [cit. 2022-01-16]. Dostupné z: <https://learnopengl.com/Advanced-OpenGL/Instancing>
- [4] Dokumentace GLFW [online]. [cit. 2022-01-12]. Dostupné z: [https://www.glfw.org/docs/3.3/group\\_\\_input.html](https://www.glfw.org/docs/3.3/group__input.html)
- [5] Návrhový vzor Command [online]. [cit. 2022-01-12]. Dostupné z: <http://gameprogrammingpatterns.com/command.html>
- [6] Dokumentace OpenGL [online]. [cit. 2022-01-24]. Dostupné z: [https://www.khronos.org/opengl/wiki/Rendering\\_Pipeline\\_Overview](https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview)
- [7] Dokumentace OpenGL [online]. [cit. 2022-01-24]. Dostupné z: <https://www.khronos.org/opengl/wiki/Tessellation>
- [8] Dokumentace OpenGL [online]. [cit. 2022-01-24]. Dostupné z: <https://www.khronos.org/opengl/wiki/Fragment>
- [9] Learn modern OpenGL graphics programming in a step-by-step fashion [online]. [cit. 2022-01-17]. Dostupné z: <https://learnopengl.com/Advanced-OpenGL/Face-culling>
- [10] Learn modern OpenGL graphics programming in a step-by-step fashion [online]. [cit. 2022-01-17]. Dostupné z: <https://learnopengl.com/Advanced-OpenGL/Blending>

## LITERATURA

---

- [11] Learn modern OpenGL graphics programming in a step-by-step fashion [online]. [cit. 2022-01-24]. Dostupné z: <https://learnopengl.com/Lighting/Basic-Lighting>
- [12] Learn modern OpenGL graphics programming in a step-by-step fashion [online]. [cit. 2022-01-26]. Dostupné z: <https://learnopengl.com/Lighting/Lighting-maps>
- [13] Čtení z nenastavené textury [online]. [cit. 2022-01-26]. Dostupné z: <https://stackoverflow.com/questions/28411686/opengl-reading-from-unbound-texture-unit>
- [14] Learn modern OpenGL graphics programming in a step-by-step fashion [online]. [cit. 2022-01-26]. Dostupné z: <https://learnopengl.com/Lighting/Light-casters>
- [15] Útlum světla v závislosti na vzdálenosti [online]. [cit. 2022-01-26]. Dostupné z: <https://wiki.ogre3d.org/tiki-index.php?page=-Point+Light+Attenuation>
- [16] Learn modern OpenGL graphics programming in a step-by-step fashion [online]. [cit. 2022-02-19]. Dostupné z: <https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping>
- [17] Learn modern OpenGL graphics programming in a step-by-step fashion [online]. [cit. 2022-02-19]. Dostupné z: <https://learnopengl.com/Guest-Articles/2021/CSM>
- [18] Dokumentace OpenGL [online]. [cit. 2022-01-24]. Dostupné z: [https://www.khronos.org/opengl/wiki/Array\\_Texture](https://www.khronos.org/opengl/wiki/Array_Texture)
- [19] Dokumentace OpenGL [online]. [cit. 2022-01-24]. Dostupné z: <https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glTexImage3D.xhtml>
- [20] Časové dotazy OpenGL [online]. [cit. 2022-02-27]. Dostupné z: <http://www.lighthouse3d.com/tutorials/opengl-timer-query/>
- [21] Dokumentace OpenGL [online]. [cit. 2022-02-27]. Dostupné z: <https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glQueryCounter.xhtml>
- [22] Dokumentace OpenGL [online]. [cit. 2022-02-27]. Dostupné z: [https://www.khronos.org/opengl/wiki/Early\\_Fragment\\_Test](https://www.khronos.org/opengl/wiki/Early_Fragment_Test)
- [23] Togelius, J.; Shaker, N.; Nelson, M. J.: Grammars and L-systems with applications to vegetation and levels. In *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, editace

- N. Shaker; J. Togelius; M. J. Nelson, Springer, 2016, str. 75, [cit. 2022-01-10].
- [24] Przemysław Prusinkiewicz, A. L.: Graphical modeling using L-systems. In *The Algorithmic Beauty of Plants*, Springer, 1996, str. 7, [cit. 2022-01-10].
- [25] Przemysław Prusinkiewicz, A. L.: Graphical modeling using L-systems. In *The Algorithmic Beauty of Plants*, Springer, 1996, str. 24, [cit. 2022-01-10].
- [26] Przemysław Prusinkiewicz, A. L.: Graphical modeling using L-systems. In *The Algorithmic Beauty of Plants*, Springer, 1996, str. 28, [cit. 2022-01-10].
- [27] Przemysław Prusinkiewicz, A. L.: Modeling of trees. In *The Algorithmic Beauty of Plants*, Springer, 1996, str. 57, [cit. 2022-01-10].
- [28] Fotografie akácie [online]. [cit. 2022-01-11]. Dostupné z: [https://commons.wikimedia.org/wiki/File:Umbrella\\_thorn\\_acacia\\_or\\_israeli\\_babool\\_tree\\_plant\\_acacia\\_tortillis.jpg](https://commons.wikimedia.org/wiki/File:Umbrella_thorn_acacia_or_israeli_babool_tree_plant_acacia_tortillis.jpg)



## **Seznam použitých zkrátek**

**GUI** Graphical user interface

**XML** Extensible markup language



## Obsah přiloženého CD

```
readme.txt ..... stručný popis obsahu CD
├── exe ..... adresář se spustitelnou formou implementace
├── src
│   ├── impl ..... zdrojové kódy implementace
│   └── thesis ..... zdrojová forma práce ve formátu LATEX
└── text
    ├── thesis.pdf ..... text práce ve formátu PDF
    └── thesis.ps ..... text práce ve formátu PS
```