

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

24. ledna 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 počítačových 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 24. ledna 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

Úvod	1
1 Herní engine	3
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
2 Vykreslování scény	13
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
3 Modelování rostlin s využitím L-systémů	21
3.1 L-systém	22
3.2 Interpretace řetězců pomocí želvy	22
3.3 Větvení v L-systémech	22
3.4 Stochastické L-systémy	24
3.5 Implementace	26
3.5.1 Formát L-systému	26
3.5.2 Želva	26
3.5.3 Rozšířená abeceda	27
3.5.4 Ovládání želvy	28
3.6 Modelování rostlin	30

3.6.1	Simulace růstu	32
3.7	Výsledky použití L-systémů	32
Závěr		35
Literatura		37
A	Seznam použitých zkratek	39
B	Obsah přiloženého CD	41

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	Face culling	16
2.4	Face culling	17
2.5	Phongův osvětlovací model	19
3.1	Identické stromy	21
3.2	Kvadratické Kochovy ostrovy	23
3.3	Struktury připomínajících rostliny	24
3.4	Využití stochastického L-systému	25
3.5	Nepravidelná koruna akáciového stromu	29
3.6	Akáciový strom rostoucí v přírodě	30
3.7	Model akáciového stromu	31
3.8	Tráva na planinách	31
3.9	Fáze růstu keře	32
3.10	Vegetace vygenerovaná na základě L-systémů	33

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, co se stal od posledního volání `processInput`. `update` posune herní simulaci o jeden krok. `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 4×4 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);
}
```

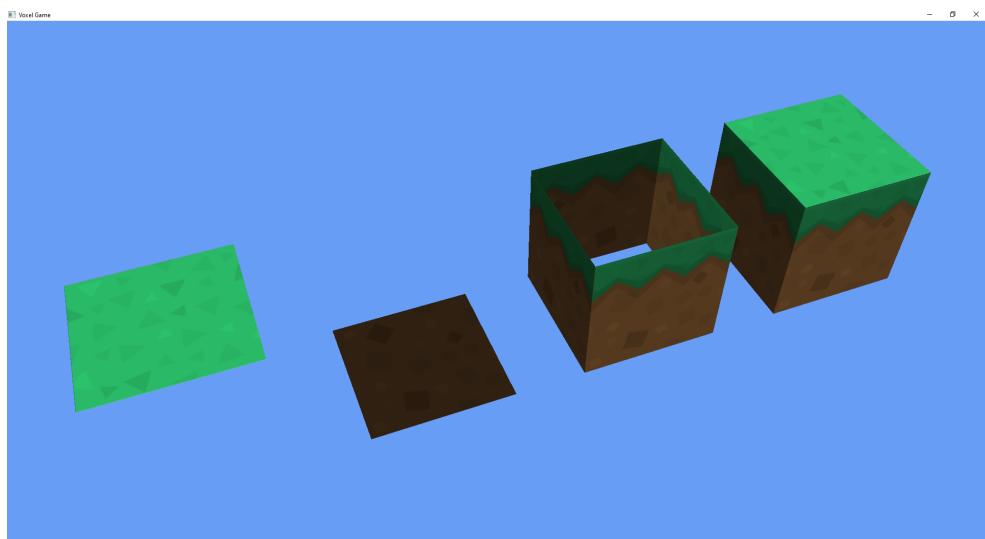
1.2.1.1 Instancing

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.

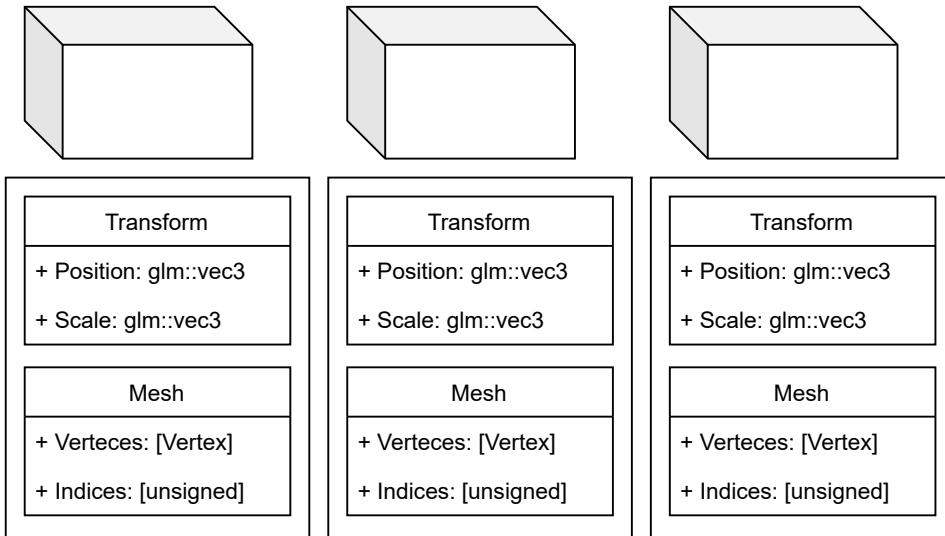
1.2.2 Společná data objektů

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 $8 * 4 = 32B$ a **Mesh** jich obsahuje $6 * 4$. Každá stěna krychle má čtyři různé vrcholy¹. Pole indexů má velikost $12 * 3 * 4$

¹Krychle 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:

$$8 * 4 * 6 * 4 + 12 * 3 * 4 = 912B$$

Pokud má být schopný engine 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.

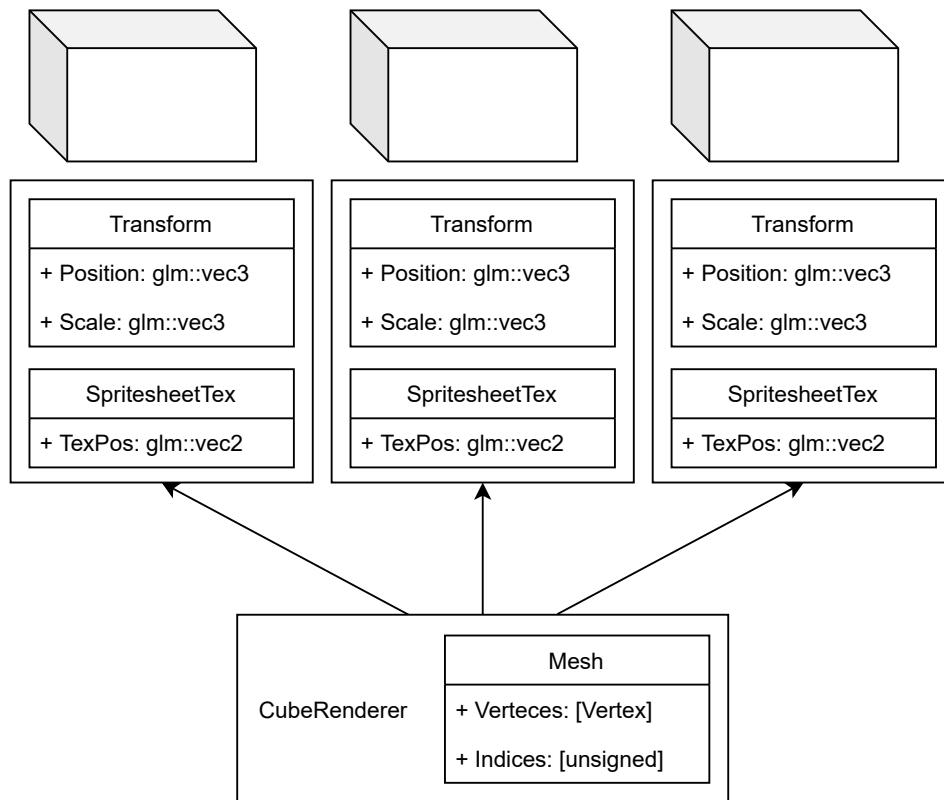
Komponenta `SpritesheetTex` obsahuje dvě čísla ve formátu float (`float` je zvolen pro jednodušší komunikaci s grafickou kartou — všechna předaná data mají formát `float`). Komponenta v paměti zabírá pouhých 8 B. Velikost herního byla redukována o 904 B.

`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. Zpětné volání zajišťují 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));
}

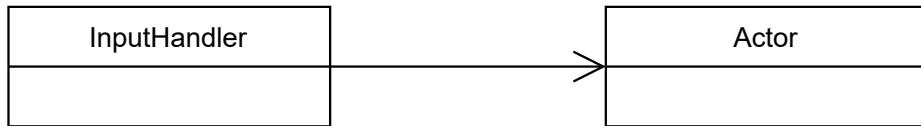
void ScrollCallback(GLFWwindow*, double, double yOffset) {
    currentHandler->ProcessMouseScroll(
        static_cast<float>(yOffset));
}

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;
    WindowManagerGl::SetCursorPosCallback(Detail::CursorPosCallback);
```

1. HERNÍ ENGINE



Obrázek 1.4: Přímé propojení InputHandler s Actor

```
 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:

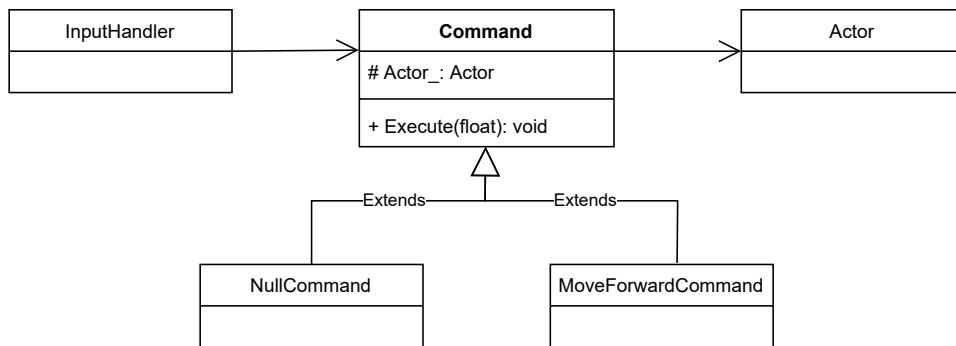
```
void InputHandler::handleInput() {  
    if (isPressed(BUTTON_W)) object.MoveForward();  
    else if (isPressed(BUTTON_S)) object.MoveBackward();  
    else if (isPressed(BUTTON_A)) object.MoveLeft();  
    else if (isPressed(BUTTON_D)) object.MoveRight();  
}
```

Tento kód funguje, pokud jsme ochotni natvrdo zadávat 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 je mimo obrazovku 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, s z vykreslovacího procesu a snížit počet volání fragment shaderu. Tato technika se nazývá face culling [9].

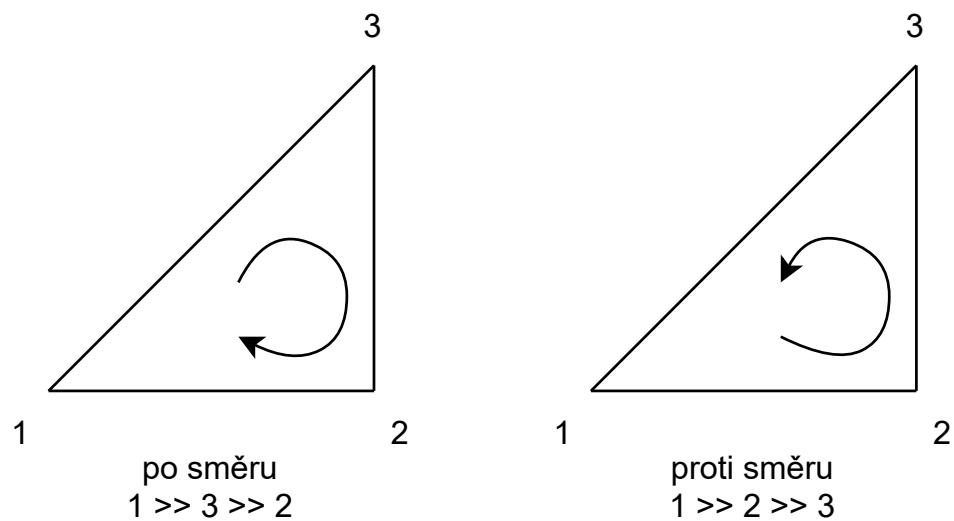
K rozpoznání stěn (trojúhelníků), které směřují pryč od hráče OpenGL používá pořadí jeho vrcholů. Vrcholy mohou být definované ve směru nebo protisměru hodinových ručiček – obrázek 2.1. Při pohledu na grafické primitivum z druhé strany, se změní pořadí jeho vrcholů. V základním nastavení OpenGL, jsou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

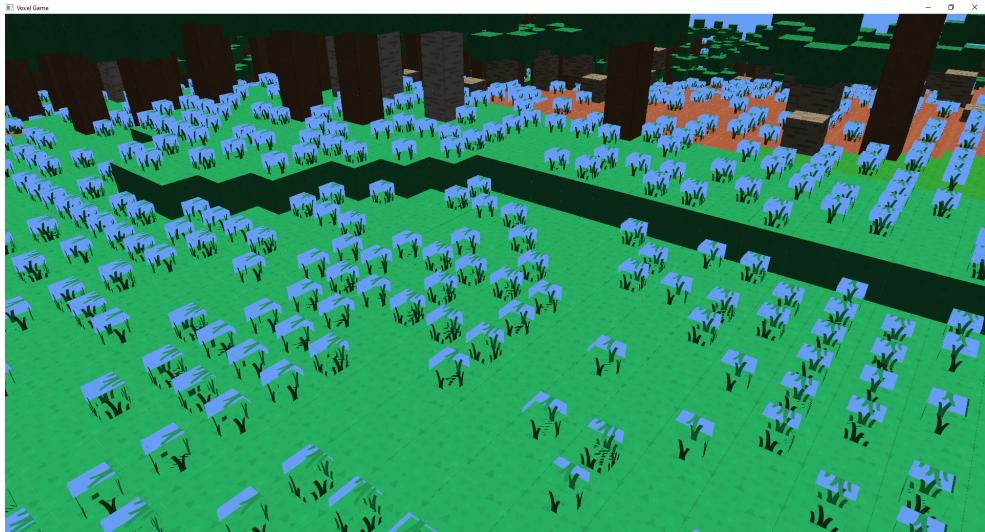


Obrázek 2.1: Pořadí vrcholů



Obrázek 2.2: Face culling

2. VYKRESLOVÁNÍ SCÉNY



Obrázek 2.3: Face culling

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 řádcích 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 zahozením 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: Face culling

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 zjednodušené 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. 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 fragmentu². 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.

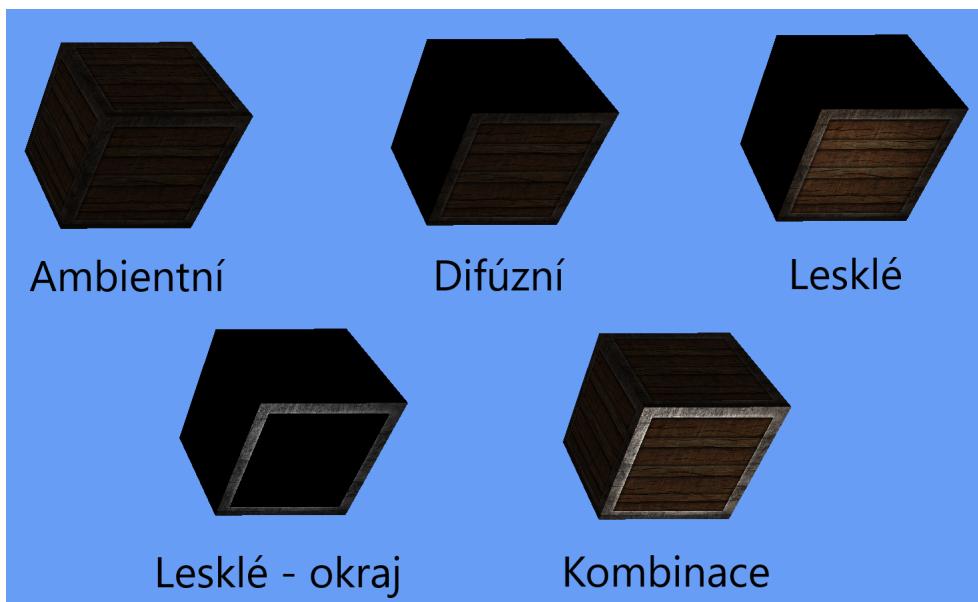
Pokud vektory svírají úhel větší než 90° je hodnota skalárního součinu záporná. Tuto situaci si lze představit jako dopad světla z opačné strany fragmentu. Ten by tedy neměl být osvětlený, `diff` je nastaven na 0. Hodnota difuzní složky je vynásobena s barvou světla a fragmentu.

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

²Normálový 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));
```

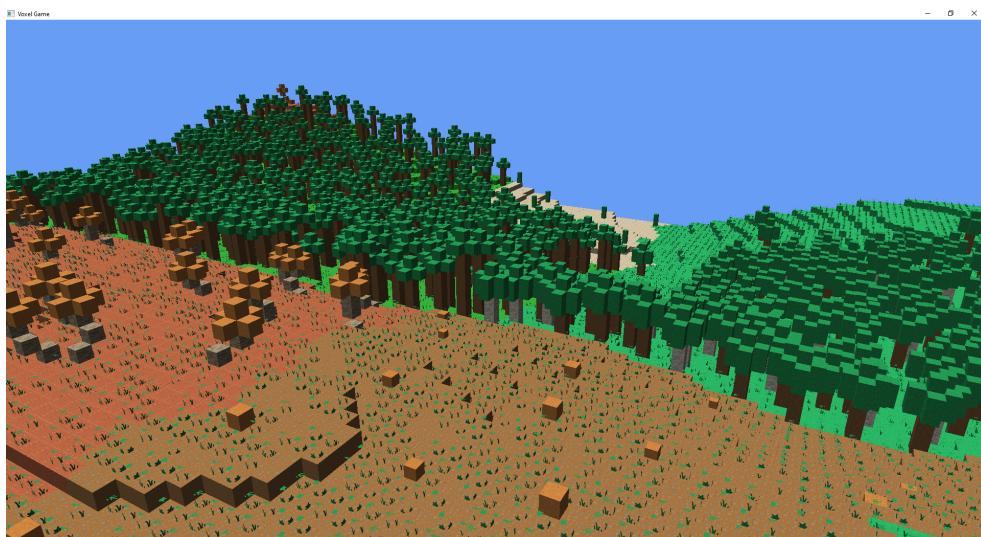
Na obrázku 2.5 je vykreslena bedna osvícena všemi druhy světel a jejich kombinací.

KAPITOLA 3

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 3.1 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 3.1: Identické stromy

3.1 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 [12].

3.2 Interpretace řetězců pomocí želvy

Řetězce lze graficky reprezentovat pomocí želvy, konsumující 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, α) , kde (x, y) představuje kartézské souřadnice reprezentující polohu v prostoru a α úhel kam želva směruje. Zadáním délky kroku d a změny úhlu δ lze želvu ovládat pomocí následujících symbolů.

- F – Posun dopředu o délku d . Stav želvy se změní na (x', y', α) , 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 δ . Nový stav želvy $(x, y, \alpha + \delta)$.
- - – Rotace doprava o úhel δ . Nový stav želvy $(x, y, \alpha - \delta)$.

Nechť je definován následující L-systém. Bud' ω počáteční axiom, p přepisovací pravidlo, $\delta = 90^\circ$ a d zmenšené čtyřnásobně pro každý obrázek [13].

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

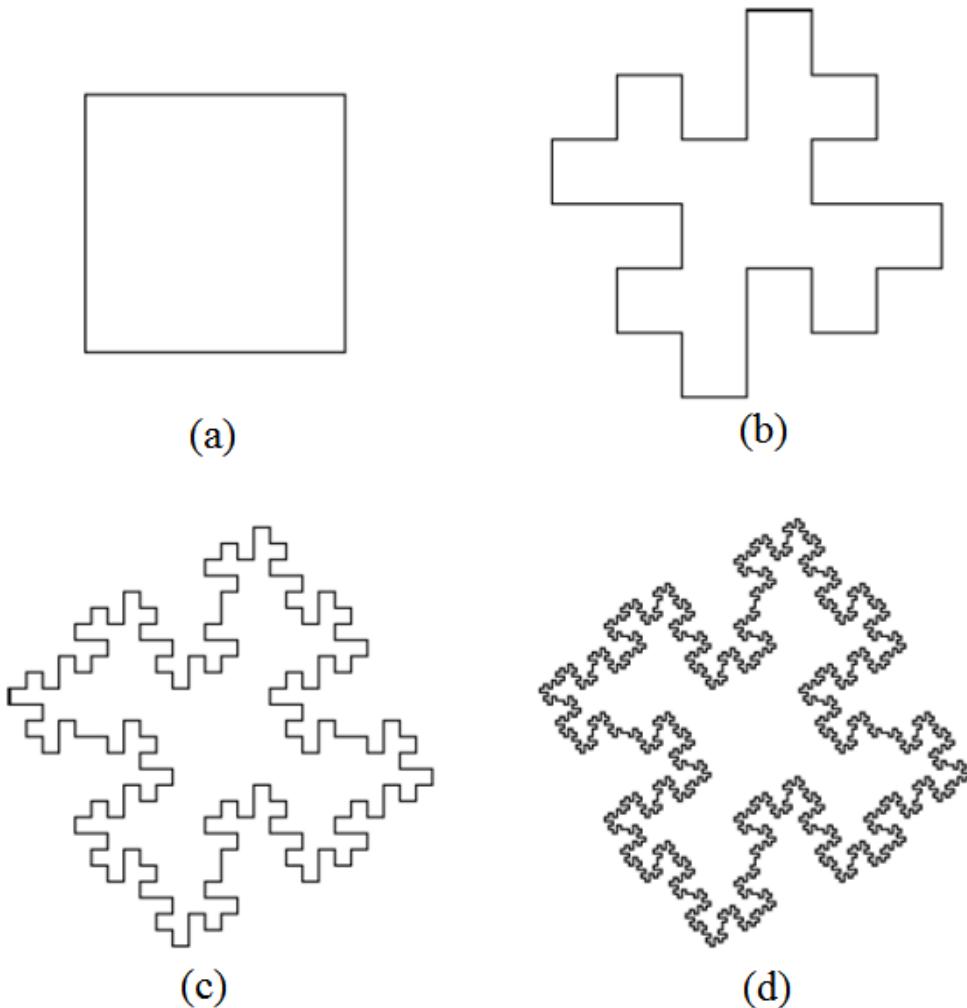
Želva interpretující daný L-systém generuje kvadratické Kochovy ostrovy 3.2. Obrázky jsou vygenerovány derivacemi o délce 0 až 3.

3.3 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 [14].

Symboly jsou interpretovány želvou následovně:



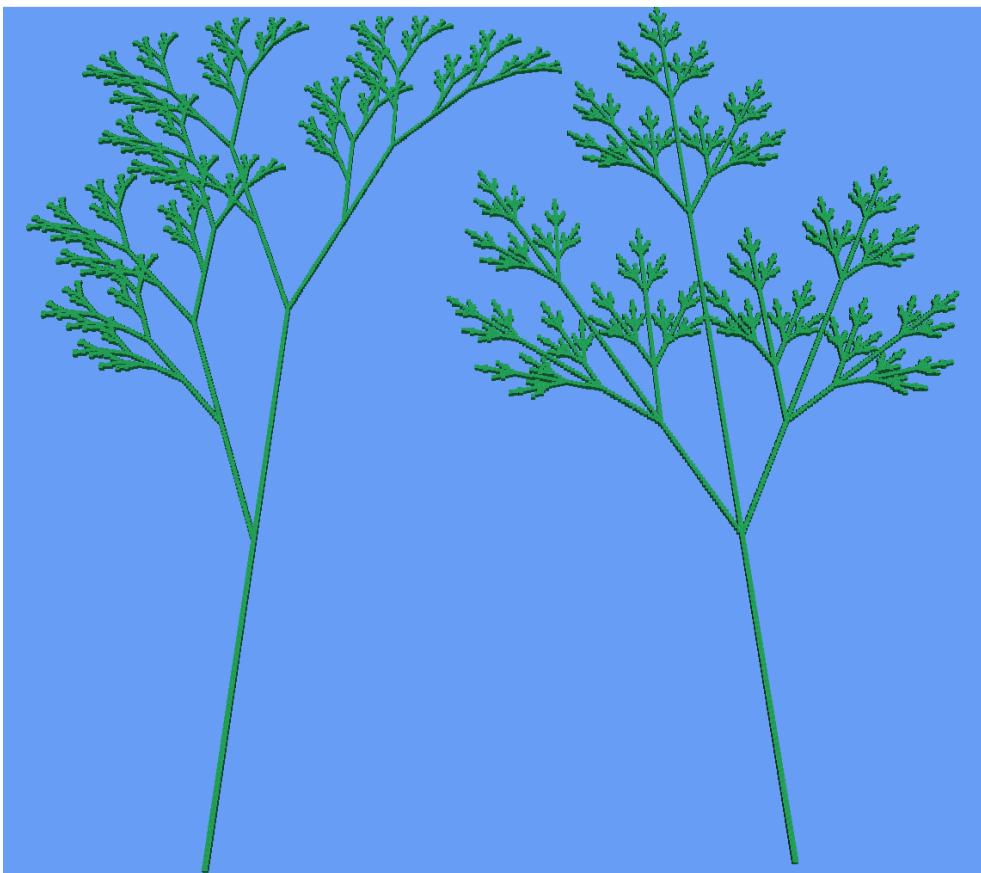
Obrázek 3.2: Kvadratické Kochovy ostrovy

- [– 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 3.3 jsou generované následujícími L-systémy:

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

3. MODELOVÁNÍ ROSTLIN S VYUŽITÍM L-SYSTÉMŮ



Obrázek 3.3: Struktury připomínající rostliny generované pomocí závorkovaného systému

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

$$\omega : E$$

$$p1 : F \rightarrow FF$$

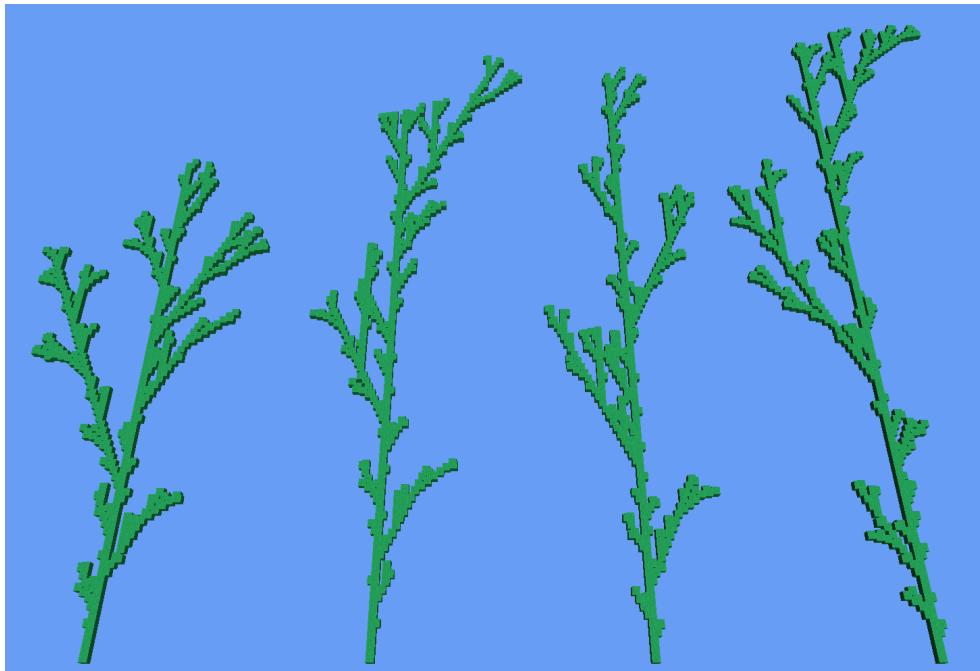
$$p2 : E \rightarrow F[+E][-E]FE$$

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

3.4 Stochastické L-systémy

Rostliny generované deterministickým L-systémem jsou všechny stejné. Jejich použití v 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 3.4: Využití stochastického L-systému

segmentů rostliny zachovávají topologii struktury, ze které je generovaná. Stochastické L-systémy mohou měnit topologii struktury [15].

-systém, který byl do ted' používán nemohl mít více přepisovacích pravidel pro stejný symbol abecedy. Pokud má stochastický L-systém vice přepisovacích pravidel, je z nich vybrána jedna s pravděpodobností $1/n$, kde n je počet přepisovacích pravidel pro daný symbol abecedy³.

Scéna 3.4 byla vygenerována za pomocí stochastického L-systému, kde:

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

$$\omega : F$$

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

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

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

³Tato definice se liší, od definice uvedené ve [15]. Tento způsob náhodného výběru je použit v implementaci, kde pravděpodobností distribuci zastupuje několikanásobné zopakování přepisovacího pravidla.

3.5 Implementace

3.5.1 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 [16]:

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

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

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

3.5.2 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 směrující 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 směrujícího 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° .

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

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

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;
}
```

3.5.3 Rozšířená abeceda

Následující symboly abecedy mají speciální význam pro jejich interpretaci.

- U a u — Posuň želvu nahoru.

3. MODELOVÁNÍ ROSTLIN S VYUŽITÍM L-SYSTÉMŮ

- F a f — Posuň želvu dopředu.
- x — Zmenši želvu.
- X — Zvětši želvu.
- S — Nastav původní velikost želvy.
- + — 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.

3.5.4 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 3.5 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

3.5. Implementace



Obrázek 3.5: Nepravidelná koruna akáciového stromu

želva otočí o 45° nahoru (^), pokládá větve (u, U), skloní se o 45° (&) 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.

3. MODELOVÁNÍ ROSTLIN S VYUŽITÍM L-SYSTÉMŮ



Obrázek 3.6: Akáciaový strom rostoucí v přírodě [17]

3.6 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 3.6 a následné pokusy o její napodobení obrázek 3.7.

Velice se osvědčilo pravidlo:

$$U > uU$$

Díky němuž 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 3.8. 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.

3.6. Modelování rostlin



Obrázek 3.7: Model akáciového stromu



Obrázek 3.8: Tráva na planinách

3. MODELOVÁNÍ ROSTLIN S VYUŽITÍM L-SYSTÉMŮ



Obrázek 3.9: Fáze růstu keře

3.6.1 Simulace růstu

Křovinatý biotop je porostlý dvěma druhy keřů rozděleným 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 3.9⁴ lze pozorovat stejné zakončení větví — rozdělení 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.

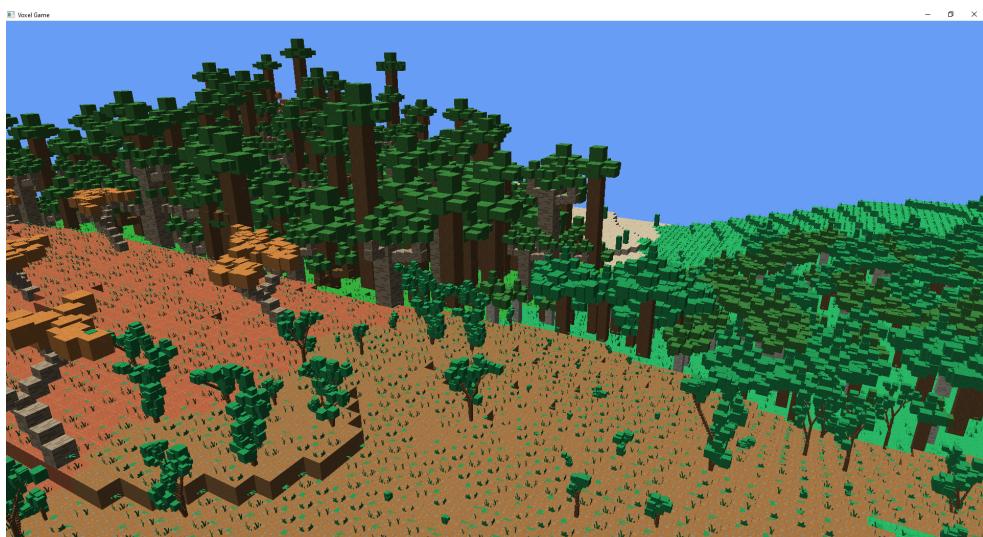
Díky těmto krokům biotop obsahuje rostliny navzájem podobného vzhledu, lišících se v drobných detailech.

3.7 Výsledky použití L-systémů

Definice L-systémů jsou krátké (≈ 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.

⁴Pro větší názornost byly odstraněny části modelu představující listy.

3.7. Výsledky použití L-systémů



Obrázek 3.10: 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 3.1 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 3.10 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
- [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
- [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] 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].
- [13] Przemysław Prusinkiewicz, A. L.: Graphical modeling using L-systems. In *The Algorithmic Beauty of Plants*, Springer, 1996, str. 7, [cit. 2022-01-10].
- [14] Przemysław Prusinkiewicz, A. L.: Graphical modeling using L-systems. In *The Algorithmic Beauty of Plants*, Springer, 1996, str. 24, [cit. 2022-01-10].
- [15] Przemysław Prusinkiewicz, A. L.: Graphical modeling using L-systems. In *The Algorithmic Beauty of Plants*, Springer, 1996, str. 28, [cit. 2022-01-10].
- [16] Przemysław Prusinkiewicz, A. L.: Modeling of trees. In *The Algorithmic Beauty of Plants*, Springer, 1996, str. 57, [cit. 2022-01-10].
- [17] 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

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 ..... text práce
    ├── thesis.pdf ..... text práce ve formátu PDF
    └── thesis.ps ..... text práce ve formátu PS
```