FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Návrh a implementácia voxelového enginu
Student:	Daniel Breiner
Vedoucí:	Ing. Petr Pauš, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Počítačová grafika
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

1. Analyzujte nástroje pro tvorbu voxelového enginu.
2. Rozeberte princip ECS (Entity-Component-System).
3. Analyzujte možnosti kolizí, greedy meshingu a procedurálního generování objektů.
4. Na základě analýzy definujte požadavky na prototyp a navrhněte ho.
5. Ve zvoleném nástroji implementujte navržený prototyp.
6. Proveďte základní testování prototypu.

Elektronicky schválil/a Ing. Radek Richtr, Ph.D. dne 1. listopadu 2021 v Praze.

Bakalárska práca

NÁVRH A IMPLEMENTÁCIA VOXELOVÉHO ENGINU

Daniel Breiner

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedúci: Ing. Petr Pauš, Ph.D.
10. mája 2022

Obsah

Zoznam obrázkov

Zoznam tabuliek

Zoznam výpisov kódu

Rád by som sa poďakoval predovšetkým Ing. Petrovi Paušovi, Ph.D. za odborné vedenie práce a jeho cenné poznatky a rady. Ďakujem mojej priateľke Márii za jej oporu a trpezlivosť. Ďakujem mojej rodine za ich nepretržitú podporu počas môjho štúdia. A ďakujem mojim priateľom, že ma vždy motivovali ísť dopredu.

Vyhlásenie

Vyhlasujem, že som predloženú prácu vypracoval samostatne a že som uviedol všetky použité informačné zdroje v súlade s Metodickým pokynom o dodržovaní etických princípov pri príprave vysokoškolských záverečných prác.

Beriem na vedomie, že sa na moju prácu vzťahujú práva a povinnosti vyplývajúce zo zákona č. 121/2000 Zb., autorského zákona, v znení neskorších predpisov. V súlade s ust. § 2373 odst. 2 zákona č. 89/2012 Zb., občiansky zákonník, v znení neskorších predpisov, týmto udeľujem nevýhradné oprávnenie (licenciu) k použitiu tejto mojej práce, a to vrátane všetkých počítačových programov, ktoré sú jej súčasťou či prílohou a všetku ich dokumentáciu (ďalej súhrnne iba „Dielo“), a to všetkým osobám, ktoré si prajú Dielo použiť. Tieto osoby sú oprávnené Dielo použiť akýmkoľvek spôsobom, ktorý neznižuje hodnotu Diela, avšak iba k neziskovým účelom. Toto oprávnenie je časovo, teritoriálne i početne neobmedzené.

V Prahe dňa 10. mája 2022 .....

Abstrakt

Táto bakalárska práca sa zaoberá tvorbou enginu pre vykresľovanie voxelových scén. Analyzuje existujúce riešenia, metódy renderovania, kolízií a procedurálnej generácie terénu. Popisuje programovací vzor ECS a optimalizačný algoritmus greedy meshing. Jej cieľom je nájsť vhodné technológie pre vytvorenie prototypu a na základe analýzy ho navrhnúť. Nakoniec prototyp voxelového enginu implementovať a otestovať jeho funkčnosť a výkon. Výsledkom praktickej časti je multiplatformný prototyp voxelového enginu s požadovanými náležitosťami.

Kľúčové slová počítačová grafika, herný engine, voxelové scény, renderovacie algoritmy, greedy meshing, procedurálna generácia, Unity, A-Frame

Abstract

This bachelor thesis addresses the creation of an engine to render voxel scenes. It analyzes existing solutions, rendering methods, collisions and procedural generation of terrain. It describes the programming pattern ECS and the greedy meshing optimization algorithm. It aims to find appropriate technologies for a prototype and to design it based on the analysis. Lastly, to implement the voxel engine prototype and to test its operability and performance. The result of the practical part is a multiplatform prototype of a voxel engine with the requested features.

Keywords computer graphics, game engine, voxel scenes, rendering algorithms, greedy meshing, procedural generation, Unity, A-Frame

Zoznam skratiek

AABB	Axis-aligned bounding box (osovo zarovnaný ohraničujúci kváder)
API	Application programming interface
AR	Augmented reality (rozšírená realita)
CPU	Central processing unit (procesor)
CT	Computed tomography (počítačová tomografia)
ECS	Entity-Component-System
FPS	Frames per second (snímky za sekundu)
-GL	Graphics library (grafická knižnica)
GLSL	OpenGL Shading Language
GPU	Graphics processing unit (grafický procesor)
HTML	HyperText Markup Language
IE	Internet Explorer
JS	JavaScript
NPM	Node Package Manager
OOP	Object-oriented programming (objektovo orientované programovanie)
RAM	Random-access memory (pamäť s priamym prístupom)
RPG	Role-playing game (hra na hrdinov)
TS	TypeScript
VR	Virtual reality (virtuálna realita)

Úvod

Väčšina 3D herných enginov sa snaží o aproximáciu reálneho sveta. Vykresľujú modely – štruktúry mnohouholníkov, materiálov a textúr, ktoré sú založené na realite. Za roky progresu grafického hardvéru a softvéru nastal posun od renderovania postáv tvorených niekoľkými desiatkami trojuholníkov, po vizualizácie hyper-realistických prostredí a bytostí. Avšak nie každá hra sa chce priblížiť k realite cez uveriteľnosť jej vizuálov, niektoré sa o to snažia vytvorením imerzie – pocitu, že ich svet je podobný reálnemu v jeho dynamickosti. Toho perfektným príkladom je hra Minecraft, ktorej štylizované scény nie sú tvorené precíznymi sieťami trojuholníkov. Namiesto toho je jej svet tvorený z tzv. voxelov – z ang. volumetric element (objemový prvok) – laicky 3D pixelov alebo jednoducho kociek, kostičiek. Voxelové svety dovoľujú to, čo je v mnohouholníkových technicky náročné – sú premenné, živé, až volatilné, hráč ich smie meniť, ničiť alebo vytvárať. Tento pocit voľnosti a neobmedzenej kreativity je to, na čom stojí úspech ako Minecraftu – najpredávanejšej hry na Zemi – tak aj iných sandboxových voxelových hier – a je aj pre túto prácu hlavnou inšpiráciou.

Cieľom tejto záverečnej práce je rozobrať kroky potrebné na vytvorenie voxelového enginu. V kapitole 1 Analýza budú popísané existujúce riešenia, algoritmy pre vykresľovanie voxelov – okrem iných aj optimalizačný algoritmus greedy meshing, metódy kontroly kolízií a spôsoby procedurálnej generácie terénu. Okrem toho bude zanalyzovaný vzor softvérovej architektúry ECS a porovnané technológie primerané pre vytvorenie prototypu enginu, najadekvátnejší vybraný. Kapitola 2 Návrh sa bude zaoberať konkrétnymi požiadavkami pre prototyp. Nájde vhodné štruktúrovanie scén – delenie sveta na logické celky s náležitými vzťahmi a komunikáciou. Taktiež sa v nej navrhne patričný spôsob generovania terénu za pomoci vrstvenia šumu. Kapitola 3 Implementácia sa bude venovať rozboru kódu vybraných algoritmov a ďalšími záležitosťami vyplývajúcimi z implementácie softvérových požiadaviek. A v poslednej kapitole 4 Testovanie budú vyložené výsledky testovania funkčnosti a výkonu prototypu.

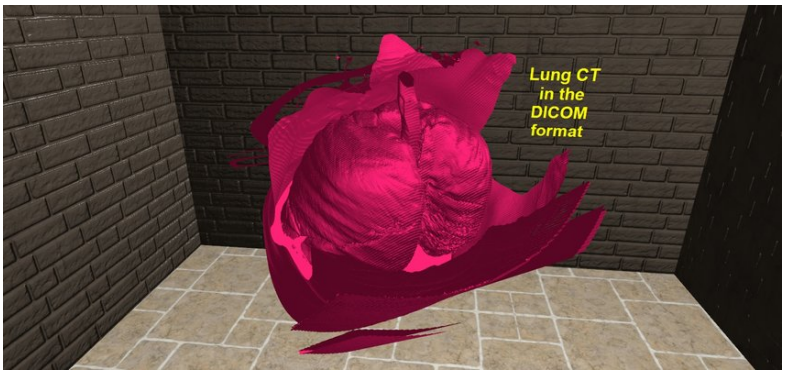
Kapitola 1
Analýza

1.1 Voxelové enginy

Voxely a enginy pre ich vykresľovanie majú väčšie využitie, ako sa môže na prvý pohľad zdať. Často sa používajú pre vizualizáciu a analýzu vedeckých a medicínskych dát (viď obr. 1.1). Táto práca sa ale bude venovať voxelovým enginom v kontexte počítačových hier, kde sa stali obzvlášť populárnymi. Súčasťou vývoja voxelových hier je avšak aj množstvo dodatočných technických prekážok, akými sú napr. kolízie, procedurálna generácia (nemusí byť súčasťou, ale je veľmi obvyklá), ale hlavne výkon, renderovanie v reálnom čase. Hry musia vykresľovať obraz 60krát za sekundu, oproti vedeckým vizualizáciám, ktorým častokrát stačí na vykreslenie jedného pohľadu alebo vizualizácie niekoľko sekúnd. Tieto aspekty budú predmetom nasledujúcich podkapitol (1.2 Renderovanie, 1.3 Kolízie, 1.4 Procedurálna generácia).

1.1.1 Existujúce riešenia

V tejto sekcii budú ukázané a popísané vybrané existujúce voxelové hry a enginy. Cieľom ich nie je porovnať, ale získať všeobecný prehľad o ich vlastnostiach a funkcionalite, ktorý bude neskôr využitý na formuláciu funkčných požiadaviek vlastného prototypu.



Obr. 1.1 Vizualizácia CT skenu vo voxelovom engine [1]

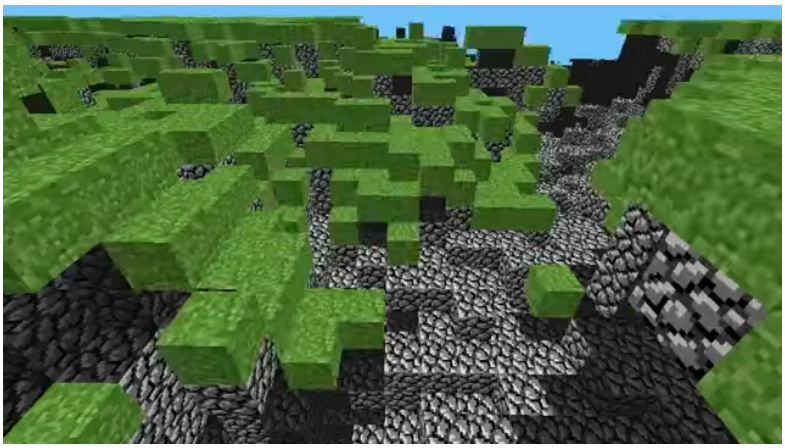
1.1.1.1 Hry

Minecraft Čo začalo ako „Cave game tech test“ (obr. 1.2) – jednoduchý voxel engine s kolíziami, v ktorom ani nebolo možné modifikovať terén [2] – sa za približne 10 rokov premenilo na najpredávanejšiu hru na svete [3]. Markus Persson, tiež známy ako „Notch“, bol v 2009 inšpirovaný malou indie hrou Infiniminer [4] – voxelovým sandboxom, z ktorého mala vzniknúť tímová kompetetívna hra o ťažení vzácnych materiálov [5]. Notch avšak videl v Infinimineri isté nedostatky a svoju hru chcel smerovať viac RPG štýlom [6]. Ďalšie dva roky pokračoval vo vývoji Minecraftu, ktorý veľmi rýchlo naberal na popularite [7]. Zo ziskov preto Notch založil herné štúdio Mojang [8]. V roku 2011 skončila Beta verzia [7] a smerovanie hry prevzal Jens „Jeb“ Bergensten [9]. Za ďalšie tri roky vydal Mojang 8 veľkých updateov pre Minecraft [10], ktorého hráčska základňa stále stabilne rástla [11]. V roku 2014 Microsoft odkúpil Notchov podiel v Mojangu a práva k Minecraftu, čím sa Notch stal miliardárom a prestal sa podieľať na ďalšom vývoji [12]. V roku 2019 sa Minecraft stal napredávanejšiou hrou na svete [13] a dodnes má desiatky miliónov aktívnych hráčov na stovkách tisícoch multiplayer serveroch [11], veľkú módovaciu komunitu, popularitu na sociálnych sieťach ako YouTube alebo Twitch a stále prebieha jeho vývoj od Mojangu, ktorý preň každoročne vydáva aktualizácie [10].

Minecraft je považovaný za jednu z najvplyvnejších hier vôbec, vyhral viacero ocenení a je veľmi pozitívne hodnotený ako kritikmi, tak aj hráčmi [14]. Stal sa inšpiráciou pre množstvo projektov (vrátane tohto) a hier – niektoré z nich ešte budú v tejto sekcii spomenuté. Vďaka jeho popularite je však tiež terčom kopírovania a vytvárania klonov, väčšinou so snahou speňažiť jeho úspech, no niektoré s dobrými úmyslami, ako napr. cvičné demá alebo vzdelávacie ukážky pre programátorov. Práve vývoj Minecraftom inšpirovaných voxelových enginov pre vzdelávacie a ukážkové účely sa stal vlastným žánrom na webovom portáli YouTube a viac bude popísaný v sekcii 1.1.1.2 Enginy.

Hytale Hra, ktorá je nielenže veľmi silno inšpirovaná, ale dokonca priamo vznikla z Minecraftu je Hytale. Jej vývojári – Hypixel Studios – sú skupina, ktorá sa oddelila od spoločnosti za serverom Hypixel – najväčším aktívnym Minecraft serverom [16]. Vývoj začal v roku 2015 [17], v 2018 vyšiel trailer ktorý mal do mesiaca viac ako 30 miliónov videní [18] a Hypixel Studios predpokladajú, že do roku 2023 by hru mohli vydať na viacerých platformách. Hytale bude procedurálne generovaná sandbox fantasy hra, s výraznými RPG prvkami, postavená na módovateľnom voxelovom engine.

Cube World Neúspešnou ukážkou voxelovej hry je akčné RPG Cube World, ktorého vývoj začal v roku 2011 [19]. Jeho vývojár – Picroma – zverejnil alpha verziu hry v júli 2013 [20]. Tá mala



Obr. 1.2 Cave game tech test [2]

Obr. 1.3 Minecraft [15]

Obr. 1.4 Hytale [27]

veľmi pozitívny ohlas, no počas ďalších rokov bez online prítomnosti vývojára považovalo Cube World mnoho ľudí za vaporware[1] [22]. Keď Picroma konečne vydal hru v roku 2019, množstvo kritikov a hráčov zostalo sklamaných a nespokojných – hru v hodnoteniach popisujú ako „vizuálne pôvabnú“, ale „nudnú“ a „plytkú“ [23][24].

Teardown V akčnej puzzle hre Teardown má hráč za úlohu plniť misie v kompletne zničiteľných voxelových leveloch [25]. Štruktúry môžu byť deštruované pomocou zbraní, výbušnin, vozidiel, ale aj ľubovoľne kreatívne, napríklad zrútením jednej budovy na druhú, narazením auta do steny atď. Titul vyšiel na platforme Steam ako hra s predbežným prístupom v roku 2020 a okamžite získal množstvo pozitívnych hodnotení (96%) [26]. Jej vývojári Teardown ďalej dvakrát aktualizovali, jej posledná aktualizácia – v apríli 2022 – bola zároveň oficiálnym vydaním [25].

Space Engineers Space Engineers je sandbox voxelová hra od nezávislého českého štúdia Keen Software House, v ktorej hráči stavajú vesmírne lode a stanice, cestujú po vesmíre a zbierajú suroviny [28]. Taktiež začala v roku 2013 ako hra s predbežným prístupom, do jej oficiálneho vydania v 2019 predala cez 3 milióny kópií [29]. Je postavená na realistickom

[1] Produkt, väčšinou softvér, ktorého vývoj je oznámený verejnosti, ale jeho vydanie má meškanie alebo nikdy nenastane a nie je oficiálne zrušený [21].

Obr. 1.5 Space Engineers [28]

engine „VRAGE 2“, kde sa objemové objekty správajú akoby mali reálnu hmotnosť, rýchlosť, smer a zotrvačnosť [30].

1.1.1.2 Enginy

Projekty vytvorené pre edukáciu a zábavu na platforme YouTube Tzv. devlog[2] je článok alebo video (vlog) popisujúce a vysvetľujúce vývoj softvéru. Devlogy sa stali obzvlášť obľúbenými na sociálnej platforme pre zdieľanie videí YouTube, najmä tie, ktorých predmetom sú počítačové hry. Teda nie je prekvapivé, že so stúpajúcou popularitou Minecraftu narástol aj počet videí a kanálov, ktoré sa zaoberajú vytváraním voxelových enginov.

Najpopulárnejší kanál „Hopson“ [31] vytvoril niekoľko Minecraft klonov v C++ za pomoci OpenGL, jeden za iba 7 dní (ktorého video má cez 5 miliónov zhliadnutí) [32] a najnovší s podporou multiplayeru [33].

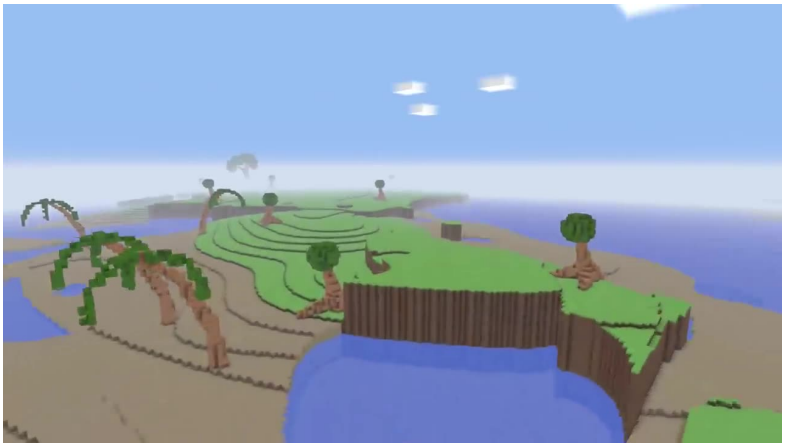
Jeden z najsledovanejších devlogov o tvorbe Minecraft klonu patrí kanálu „jdh“ [34], ktorý stihol vývoj hry v jazyku C za iba 2 dni [35]. Jeho druhý klon, napísaný v C++, nemal taký úspech napriek tomu, že bol funkcionalitou nadradený [36].

Komplexnými hernými prvkami bol nabitý Minecraft klon od YouTubera „SimonDev“ [37], ktorý dal extra dôraz na procedurálnu generáciu, ako terénu – s rôznymi typmi šumu, tak aj vegetácie – za pomoci funkcie orientovanej vzdialenosti a L-systémov (obr. 1.6) [38]. Jeho výsledok je pôsobivý, pretože jeho tech stack[3] bol Three.js a slabo typovaný JavaScript – jeho nevýhody budú popísané v sekcii 1.6 Technológie.

Za zmienku stojí séria videí od „GamesWithGabe“ [40], ktorý do svojej hry vyvíjanej v C++ pridal množstvo pokročilých algoritmov a funkcionalít, napr. multiplayer, komplexné osvietenie alebo optimalizačné renderovacie techniky, ako greedy meshing [41].

Všetky zatiaľ uvedené projekty boli kódené na relatívne nízkej vrstve – ich vývojári písali kód pre spracovanie a komunikáciu s grafickou kartou. Preto by bolo vhodné spomenúť aj klony, na ktoré boli použité herné enginy – riešenia vyššej úrovne. Najzaujímavejšími sú video od „Sam Hogan“, ktorý vytvoril jednoduchú voxel hru v Unity za 24 hodín (pravdepodobne najúspešnejší klon na YouTube, s viac ako 10 miliónmi videní) [42], alebo týždňová výzva vytvoriť klon Minecraftu v Unity od kanálu „LegendOfLinq“ [43].

[2] z ang. *development vlog* (vlog o vývoji) alebo *development log* (záznam o vývoji)
[3] zbierka všetkých technológií použitých na vytvorenie a spustenie jednej aplikácie [39]



Obr. 1.6 Minecraft klon od SimonDev [37]

Minetest Veľké výhody open-source voxelového enginu Minetest spočívajú v jeho zameraní sa na módovateľnosť, rozšíriteľnosť a dostupnosť [44]. Vývojári majú k dispozícii celý zdrojový kód a skriptovacie rozhranie v programovacom jazyku Lua, pomocou ktorých ich obmedzuje iba vlastná kreativita vo vytváraní nadstavieb, tzv. *hier*, pre tento engine. Hráči si tieto *hry* nájdu v na to určenej online databáze [45] a jednoducho si ich stiahnu, a spustia. Aktívna komunita sa stará o nepretržitý vývoj enginu a stále zväčšovanie databázy *hier*, v ktorej je v súčasnosti cez 1000 položiek [44].

voxel.js Voxel.js je voxelový engine pre prehliadače [46], ktorý je architekturálne postavený na moduloch – menších častiach logiky, ktoré je možné spojiť do hotového celku podľa potrieb konkrétneho projektu. Využíva výhody ekosystému Node.js, ktoré sú tiež popísané v sekcii 1.6.1.2 A-Frame: Výhody a nevýhody. Vývojár môže použiť hotové moduly od Voxel.js, ktorých existujú desiatky [47], alebo si vytvoriť vlastné, podľa svojich potrieb.

1.2 Renderovanie

V tejto podkapitole budú popísané optimalizačné algoritmy pre zjednodušenie meshe a porovnanie spôsobov vykresľovania textúr na voxeloch.

Cieľom týchto algoritmov je minimalizovať komplexitu meshe – vytvoriť trojuholníkovú sieť s čo najmenším počtom trojuholníkov (štvoruholníkov). Dôležité pre tieto algoritmy je rozlišovať dva typy voxelov: prázdny alebo priehľadný voxel (vzduch, príp. sklo, voda, ...)[4] a plný alebo nepriehľadný voxel (napr. kameň, hlina, tráva, ...), ktorého viditeľné plochy musia byť súčasťou meshe.

1.2.1 Vyraďovanie plôch

Jeden z najpriamočiarejších algoritmov pre zjednodušenie meshe je tzv. vyraďovanie plôch[5]. Spočíva v odstránení plôch z meshe v miestach, na ktorých ich nie je možné vidieť – teda medzi dvomi nepriehľadnými voxelmi.

[4] Prázdne voxely sú podmnožinou priehľadných voxelov, sú to iba voxely s nulovou hodnotou (často označované ako *vzduch*).

[5] ang. *face culling*

1.2.1.1 Popis algoritmu

Pri vytváraní meshe prechádza algoritmus všetkými voxelmi. Pre každý nepriehľadný voxel skontroluje jeho 6 susedov. Do meshe sú zahrnuté iba tie plochy, ktoré sú medzi nepriehľadnými voxelmi a priehľadnými voxelmi. Zvyšné plochy sú vyradené – nepoužité.

1.2.2 Greedy meshing

Ďalším algoritmom pre optimalizáciu meshí je tzv. greedy meshing. Mikola Lysenko a jeho článok o tejto problematike z roku 2012 *Meshing in a Minecraft Game*[6] [48] je základom pre obrovskú väčšinu ostatných online zdrojov, preto z neho bude do istej miery čerpať aj táto sekcia.

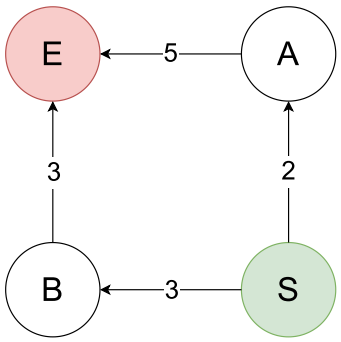
1.2.2.1 Greedy algoritmy

Greedy algoritmus (niekde sa používa aj poslovenčený výraz „pažravý algoritmus“) sa nazýva taký algoritmus, ktorý sa vždy rozhodne pre najlepšiu okamžitú (lokálnu) možnosť [49]. Nech je ako príklad uvedený algoritmus na vyhľadávanie najkratšej cesty. Ak by si pri každom vrchole, kde si môže vybrať cestu vybral práve tú najkratšiu, bez ohľadu na ďalšie vrcholy, hrany alebo ich ceny, tak by bol nazvaný greedy. Takýto algoritmus by pre vstup z obr. 1.7 nenašiel optimálne riešenie[7]. Častokrát ale môžu greedy algoritmy vytvoriť za primerané množstvo času lokálne optimálne riešenia, ktoré sa približujú ku globálne optimálnym riešeniam. To je aj prípad greedy meshingu, ktorého kvalita bude popísaná v sekcii 1.2.2.3 Rozbor algoritmu pri pohľade na komplexnosť meshe.

1.2.2.2 Popis algoritmu

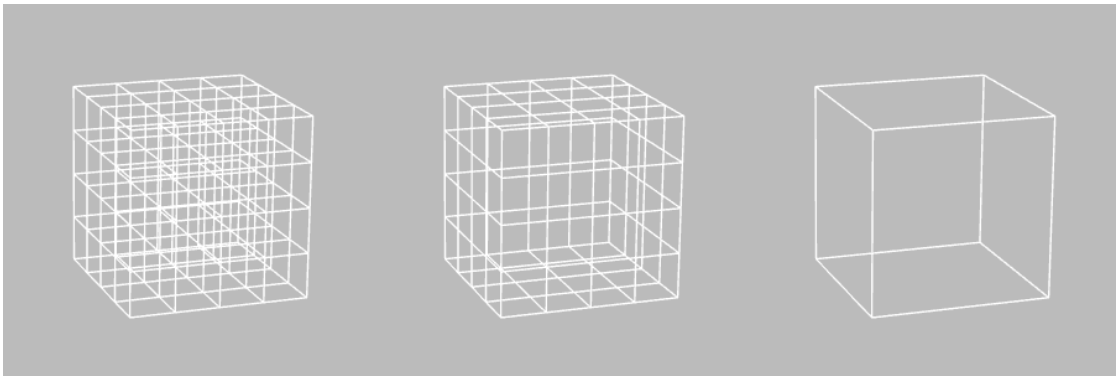
Algoritmus bude najprv popísaný v jednom rozmere, potom bude rozšírený do dvoch a troch rozmerov. Detailné vysvetlenie algoritmu a jeho kódu sa nachádza v sekcii 3.2.3 Implementácia: Greedy meshing.

Na vstupe je pole voxelov. Pre účely algoritmu budú voxely rozdelené na horespomenuté plné a prázdne (iba nulová hodnota, vzduch). Pre každý voxel na vstupe si bude potrebné držať informáciu o tom, či už bol použitý – pridaný do meshe. Algoritmus začne prechádzať vstupným poľom. Prázdne voxely môžu byť rovno označené ako použité – nie je ich potrebné vykresľovať. Pri prvom plnom voxeli si algoritmus vytvorí pole, do ktorého ho vloží. Ďalšie (susediace) plné voxely tiež pridá do tohto poľa, až kým nenarazí na prázdny voxel. V tom momente vytvorí pre všetky prvky poľa spoločnú plochu (namiesto vlastnej plochy pre každý voxel), tú pridá



Obr. 1.7 Vstup pre algoritmus, ktorý hľadá najkratšiu cestu z vrcholu S do vrcholu E

[6] sk. *Meshovanie v Minecraftovej hre*
[7] Z vrcholu S by pokračoval do vrcholu A, pretože je to najlepšia lokálna možnosť.



Obr. 1.8 Porovnanie algoritmov na vytváranie meshe: naivný, s vyraďovaním, greedy [50]

do meshe. Tieto voxely zaznačí ako použité a pole zahodí. Cyklus sa opakuje, kým algoritmus neprejde celým vstupom.

Pri priechode v jednom rozmere je pre algoritmus informácia o použitých voxeloch zbytočná, no v dvoch rozmeroch sa bez nej už nezaobíde. Algoritmus sa jednoducho rozšíri v mieste, kde narazí na prázdny voxel pri spájaní plných voxelov. V dvoch rozmeroch sa ešte pred vytvorením plochy pokúsi rozšíriť pole susedov o voxely v ďalšom riadku (posunie sa v druhom rozmere). Voxely zahrnie buď všetky – v prípade že je každý plný a nepoužitý, alebo žiaden.

V troch rozmeroch v našom prípade nebude algoritmus rozšírený o ďalšie zväčšovanie poľa susedných voxelov – to by namiesto spájania plôch spájal celé voxely do kvádrov. Namiesto toho iba použije dvojrozmerný algoritmus na každom zo šiestich smerov.

Spomenutia hodnou poznámkou je, že pre priehľadné voxely (nie vzduch) je potrebný druhý beh algoritmu. Ich textúry obsahujú transparentnú zložku, na ktorú sa používa rozdielny shader, teda je potrebná aj samostatná mesh.

1.2.2.3 Rozbor algoritmu pri pohľade na komplexnosť meshe

Môže byť konštatované, že greedy mesh (výsledok tohto algoritmu), má nanajvýš 8-krát toľko štvoruholníkov, ako optimálna mesh – čo je konštantný koeficient od optimálnosti. Počet štvoruholníkov v rôznych meshiach má nasledovnú progresiu: optimálna $\leq$ greedy $\leq$ s vyraďovaním $\leq$ naivná [48].

1.2.3 Textúry

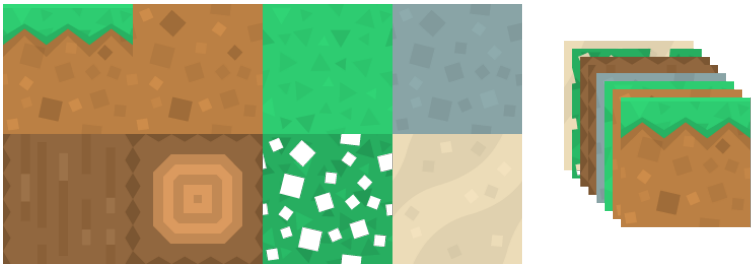
V nasledujúcej sekcii budú popísané najčastejšie prístupy pre vykresľovanie textúr vo voxelových hrách.

Atlas textúr Atlas textúr, voľne preložené z anglického *texture atlas*, často tiež pod názvom *spritesheet*, je obrázok, ktorý obsahuje viacero menších obrázkov, zabalených dokopy. Individuálne obrázky sú potom rozbalené za pomoci textúrových súradníc.

Výhody:

- Menší počet vstupných a výstupných operácií disku a grafickej karty oproti použitiu individuálnych textúr.
- Potenciálne zmenšenie celkových rozmerov textúr (rovnaké časti sa môžu prekrývať).

Nevýhody:



Obr. 1.9 Atlas textúr vs. pole textúr

- Textúrové súradnice sú často ukladané ako čísla s pohyblivou rádovou čiarkou v rozmedzí <0;0> až <1;1>, teda ich presnosť je obmedzená a s ňou aj počet a kvalita obrázkov, ktoré je možné zabaliť do jedného atlasu.
- Pri kompresii alebo inom filtrovaní ako bodové vzorkovanie[8] môže nastať pretekanie textúr na ich okrajoch.
- Obmedzená schopnosť textúrového dlaždicovania[9].

Pole textúr Pole textúr je pole obrázkov rovnakých rozmerov, ktoré GPU vníma ako jeden objekt. Ku každému individuálnemu obrázku je možné pristupovať pomocou jeho indexu. Tiež sa často označuje ako *Texture2DArray*, podľa na to používaného dátového typu v OpenGL.

Výhody:

- Menší počet vstupných a výstupných operácií disku a grafickej karty oproti použitiu individuálnych textúr.
- Nie je potrebné vypočítavať textúrové súradnice, stačí si udržiavať index textúry, najjednoduchšie ho pripojiť ku každému vrcholu meshe.
- Triviálne textúrové dlaždicovanie.
- Veľmi jednoduché vytvoriť dynamicky pri inicializácii aplikácie.

Nevýhody:

- Všetky textúry v poli musia mať rovnaký rozmer.

Vzhľadom na to, že vo voxelovom engine je logické, aby mali všetky textúry rôznych typov voxelov rovnaké rozmery, v prototype bude použité pole textúr. Taktiež má mnoho výhod, z ktorých najdôležitejšími sú jednoduché textúrové dlaždicovanie a fakt, že textúry na okrajoch nepretekajú, obe veľmi podstatné pri textúrovaní meshe, ktorá vznikne z greedy meshingu.

1.3 Kolízie

V tejto sekcii budú popísané možnosti kolízií v kandidátskych technológiách pre prototyp a vlastný, elegantnejší spôsob akým môžu byť kolízie riešené v užšom prípade voxelových scén.

1.3.1 Vstavané riešenia

V Unity je základom pre simuláciu fyziky komponent *Rigidbody*[10], s ktorým objekty reagujú na gravitáciu, a ktorého rozhranie dovoľuje v logike hry na teleso aplikovať fyzikálne sily. Okrem

[8] ang. *nearest-neighbor interpolation*
[9] ang. *texture tiling*
[10] sk. tuhé teleso

iného obsahuje aj optimalizáciu tzv. „spania“ – komponent sa deaktivuje, ak sa jeho objekt nehýbe a tak zbytočne nevyužíva výkon. Na simuláciu vzájomného vplývania viacerých telies je potrebné na nich pridať komponent *Collider*. Ten definuje tvar telesa, podľa ktorého koliduje s ostatnými objektami v scéne. *Collider* má viacero konfigurácií, tu sú najdôležitejšie dve – statický *Collider* a *Rigidbody Collider*. Statický je v prípade, že existuje na objekte bez *Rigidbody* – teda na objekte, ktorý sa nebude hýbať. Statický *Collider* počíta iba s kolíziami s *Rigidbody*, najčastejšie sa používa na objekty prostredia, s ktorými majú dynamické objekty kolidovať. Naopak *Rigidbody Collider* sa väčšinou používa na spomenuté dynamické objekty, vrátane napr. hráča. Dôležité je spomenúť, že jednoduchšie komponenty *BoxCollider* (kváder), *SphereCollider* (guľa) alebo *CapsuleCollider* (kapsula) sú výpočetne oveľa nenáročnejšie a teda aj rýchlejšie ako zložitejší *MeshCollider*.

Vstavané kolízie v A-Frame neexistujú, no pre účely prototypu by bolo možné použiť knižnice *aframe-physics-system* a *aframe-physics-extras*, ktoré fungujú obdobne ako Unity.

1.3.2 Vlastné riešenia

Alternatívou k vstavaným riešeniam, ktoré majú väčšinou rozsiahle prípady využitia (pre prototyp až zbytočne rozsiahle), môže byť vo voxelových scénach inteligentnejšie testovanie kolízií priamo v trojrozmernom poli voxelov scény. Okolo hráča (alebo ďalších dynamických objektov) sa vytvorí tzv. AABB – osovo zarovnaný ohraničujúci kváder, ktorého rohy patria do určitého voxelu – v prípade, že je to plný voxel, nastane kolízia.

1.4 Procedurálna generácia

Procedurálna generácia má v počítačovej grafike široké využitie, hlavne v kinematografii a vývoji hier – jedno z najčastejších (a základných) využití je generácia textúr, ktorá bude aj úlohou prototypu. Prístupy procedurálnej generácie sa líšia podľa konkrétnych požiadaviek, od fraktálov, cez Voroného teseláciu, po šum, práve ktorý sa často používa na generáciu prírodne vyzerajúceho terénu.

1.4.1 Šum

V tejto sekcii budú popísané základné typy šumu, ktoré budú vrstvené pomocou čiastkového Brownovho pohybu, aby bola vygenerovaná prírodne vyzerajúca výšková mapa terénu.

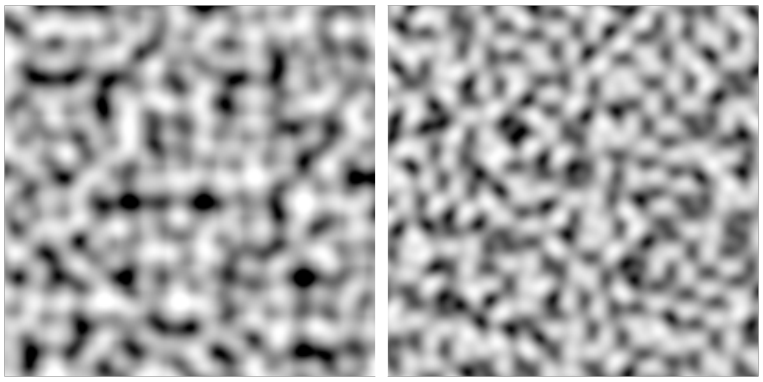
1.4.1.1 Základné typy

Perlinov šum Perlinov šum je pomenovaný podľa svojho tvorcu Kena Perlina, ktorý ho vytvoril v roku 1983 pre film Tron. Tento algoritmus vytvára n-dimenzionálnu textúru pseudonáhodného vzhľadu so spojitými vizuálnymi detailmi rovnakej veľkosti [51].

Simplex šum V roku 2001 vytvoril Ken Perlin vylepšenie klasického Perlinovho šumu nazvaný Simplex šum. Jeho hlavné zlepšenia sú jednoduchší výpočet, a teda väčšia rýchlosť, a odstránenie artefaktov [52].

1.4.1.2 Čiastkový Brownov pohyb

Súčet po sebe idúcich oktáv šumu, každá s vyššou frekvenciou a nižšou amplitúdou sa nazýva čiastkový Brownov pohyb. Na type použitého šumu nevyhnutne nezáleží. Okrem toho využíva dva parametre – lakunaritu a perzistenciu. Každý z týchto pojmov bude vysvetlený [54].



Obr. 1.10 Perlinov šum vs. Simplex šum [53]

Počet vrstiev šumu sa označuje ako oktávy, čím ich je viac, tým bude výsledok detailnejší, no aj algoritmus pomalší.

Frekvencia označuje počet detailov, ktoré sa zmestia do vytvoreného priestoru. Väčšia frekvencia vytvorí viac detailov.

Amplitúda označuje veľkosť, resp. výšku detailov. Čím je toto číslo väčšie, tým výraznejšie detaily vo výsledku budú.

Lakunarita násobí frekvenciu naprieč oktávami. Štandardná hodnota je 2, jej zmena upravuje počet malých detailov.

Perzistencia násobí amplitúdu naprieč oktávami. Štandardná hodnota je $\frac{1}{lakunarita} = \frac{1}{2}$, jej zmena upravuje vplyv malých detailov.

V prototype bude použitý čiastkový Brownov pohyb a Simplex šum, konkrétne parametre a spracovanie vygenerovaných hodnôt bude popísané v sekcii 2.3 Návrh: Procedurálna generácia.

1.5 Entity-Component-System

Táto podkapitola sa bude zaoberať architekturálnymi a programovacími vzormi a princípmi v softvérovom návrhu, konkrétnejšie v hernom vývoji. Porovná tradičnú OOP paradigmu komponent voči dátovo založenému ECS a rozoberie ich výhody a nevýhody.

1.5.1 Objektovo orientované programovanie

OOP (z anglického *Object-oriented programming*) je programovacia paradigma založená na používaní *objektov* – inštancií *tried*, predpisov a abstrakcií logiky a dát.

OOP je zrejme najznámejším a najrozšírenejším vzorom v softvérovom inžinierstve – jeho princípy aplikuje väčšina najpoužívanejších programovacích jazykov. Vplyvným dielom popisujúce objektovo orientovaný návrh a programovacie vzory preň je *Design Patterns: Elements of Reusable Object-Oriented Software*[11] od tzv. *Gang of Four*[12] [55]. Tieto vzory sú samozrejme aplikovateľné aj pre počítačové hry, ale konkrétnejšie sa na herný vývoj zameral Robert Nystrom v *Game Programming Patterns*[13] [56]. Práve v tomto diele sú popísané najčastejšie a najužitočnejšie architekturálne paradigmy pre vývoj hier, vrátane *komponentu* – vzoru používaného najväčšími hernými enginmy ako Unity alebo Unreal. Tieto komponenty sú súčasťou objektovo orientovaného návrhu – majú v sebe ako logiku, tak aj dáta, čo je hlavný rozdiel oproti komponentom v ECS.

[11] sk. *Návrhové vzory: Prvky znovupoužiteľného objektovo orientovaného softvéru*
[12] sk. *Gang Štyroch* – Gamma, Helm, Johnson a Vlissides
[13] sk. *Vzory pre programovanie hier*

Avšak je možné vidieť paralelu v entitách (v Unity *GameObject*, v Unreal *Actor*), ktoré rovnako ako aj v ECS slúžia iba ako kontajner pre komponenty.

1.5.2 Kompozícia pred dedičnosťou

Princíp kompozície pred dedičnosťou[14] je základným pilierom dátovo zameraného ECS. Pre dôkladné pochopenie ECS je potrebné si každý z týchto pojmov podrobne vysvetliť.

Dedičnosť je v objektovo orientovanom programovaní mechanizmus, ktorý ustanovuje medzi triedami vzťah „je“. Dedičnosť je veľmi mocný nástroj, no spôsobuje silnú previazanosť kódu – zmeny v rodičovskej triede takmer vždy spôsobujú potrebu meniť aj ich potomkov. Kvalitne navrhnutý kód je slabo previazaný, pretože silná previazanosť zhoršuje čitateľnosť a udržateľnosť. To avšak neznamená, že dobrý kód nesmie obsahovať dedičnosť, práve naopak, v mnohých prípadoch je vhodná a veľmi užitočná. Avšak dnes je už OOP a teda aj dedičnosť tak rozšírenou paradigmou, že sa často používa aj na miestach, kde by stačilo kód previazať výrazne menej – napríklad kompozíciou.

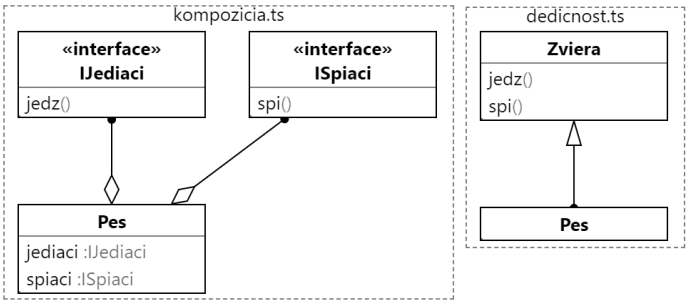
Kompozícia alebo objektová kompozícia je spôsob, akým sa medzi triedami určuje vzťah „používa“ alebo „obsahuje“. Spolu s dedičnosťou sú to základné techniky, akým je do jednoduchých štruktúr a typov pridávaná komplexita a robustnosť, bez čoho sa nie je možné zaobísť pri vytváraní netriviálneho kódu. Kým pomocou dedičnosti by bola trieda `Pes` definovaná ako dieťa triedy `Zviera`, ktorá obsahuje metódy `jedz()` a `spi()`, pomocou kompozície by mohla byť určená ako kombinácia rozhraní (zvyčajne členovia triedy) `IJediaci` a `ISpiaci`. Zmena v rodičovskej triede (v prípade kompozície rozhrania) by tak neovplyvnila triedy, ktorých sa nemá týkať.

Dátovo zamerané programovanie sa okrem preferovania kompozície tiež snaží oddeliť logiku od dát – výborne zvládnuté v ECS – alebo písať kód deklaratívne (čo má byť výsledkom) namiesto imperatívne (aké sú kroky, ktorými sa program dostane k výsledku).

1.5.3 Popis vzoru

Základom ECS sú tri prvky, podľa ktorých je vzor pomenovaný:

- *Entita* je zbierka, kontajner, súbor komponentov. Častokrát nie je definovaná ničím, okrem jej unikátneho čísla.
- *Komponent* označuje v entite konkrétnu vlastnosť a drží v sebe pre tú vlastnosť relevantné dáta.



Obr. 1.11 Kompozícia vs. dedičnosť

[14] ang. *composition over inheritance*

- *Systém* je jediný zdroj hernej logiky v ECS. Dotazuje sa na všetky entity s určitými komponentmi a pracuje nad ich dátami.

Okrem toho sa väčšinou v ECS enginoch tiež vyskytuje spôsob zoskupenia entít, často sa označuje ako *svet*, príp. *scéna*. Práve tu systémy vyhľadávajú entity na spracovanie.

Dátové zameranie ECS spočíva najmä v jeho vlastnosti, že entity nie sú definované typom, ale iba komponentmi, ktoré sú na nich nasadené.

1.5.4 Výhody a nevýhody

Výhody:

- Výkon – charakter systémov ich dovoľuje veľmi jednoducho paralelizovať.
- Jednoduchosť návrhu, prirodzené rozdelenie kódu na dáta a logiku.
- Znovupoužiteľnosť komponentov a systémov naprieč projektmi.

Nevýhody:

- Nová paradigma môže byť pre vývojárov náročná, preorientovanie a naučenie sa nových praktík zaberie istý čas.
- V systémoch často veľmi záleží na ich poradí, vytváranie nových systémov vo väčšom projekte môže byť zložité.

Celkovo je ECS kvalitný vzor so širokým uplatnením a bude zaujímavé sledovať jeho presadenie sa s prebiehajúcim posunom v architekturálnych paradigmách u veľkých herných enginov, hlavne Unity.

1.6 Technológie

1.6.1 A-Frame

A-Frame je open-source webový ECS framework zameraný primárne na 3D, VR a AR [57]. V roku 2015 ho predstavil Mozilla WebVR tím a vydal prvú verejnú verziu 0.1.0 [58]. Postavený na Three.js, jeho cieľom je preň poskytnúť deklaratívnu a komponovateľnú štruktúru.

1.6.1.1 Three.js a WebGL

Three.js je open-source JavaScriptová knižnica obsahujúca API pre vytváranie a zobrazovanie 3D grafiky v prehliadači pomocou WebGL [59]. Obsahuje základné abstrakcie pre jednoduchý a rýchly vývoj aplikácií.

WebGL je nízkoúrovňové Javascriptové API pre renderovanie 2D[15] grafiky v prehliadači [60]. Vývojár pomocou neho píše GLSL shader kód, ktorý sa vykonáva na grafickej karte.

1.6.1.2 Výhody a nevýhody

A-Frame je moderný a kvalitne navrhnutý framework. Avšak webové technológie, ktoré používa majú množstvo kompromisov.

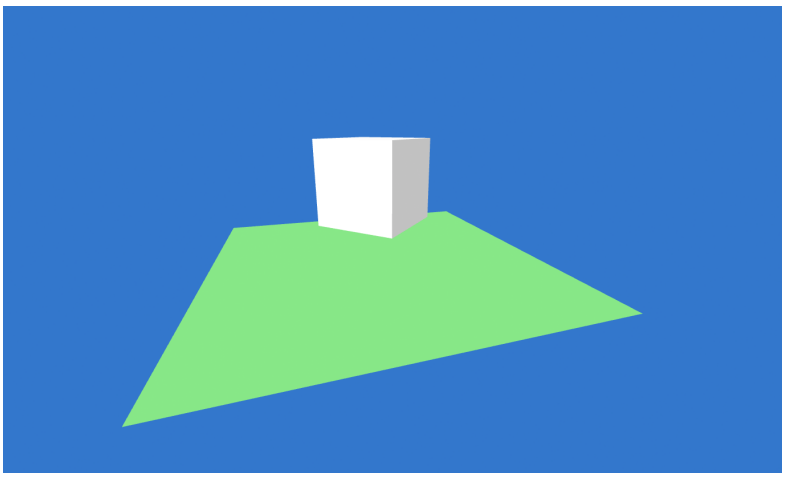
Výhody:

- Jednoduchosť, vysoká rýchlosť prototypovania.

[15] Transformáciu 3D priestoru na dvojrozmernú obrazovku má na starosti vývojár.

```
<html>
  <head>
    <script src="https://aframe.io/releases/1.3.0/aframe.min.js"></script>
  </head>
  <body>
    <a-scene>
      <a-box position="0 1 0" rotation="0 45 0" color="#dddddd"></a-box>
      <a-plane position="0 0 0" rotation="-90 0 0" width="4" height="4"
      ↪  color="#77cc77"></a-plane>
      <a-sky color="#3377cc"></a-sky>
    </a-scene>
  </body>
</html>
```

Výpis kódu 1.1 Ukážkový kód v A-Frame



Obr. 1.12 Výsledok ukážkového kódu 1.1

- Open-source softvér, prebiehajúci vývoj.
- Aktívna komunita. Znovu-použiteľnosť komponentov je posilnená JavaScriptovým/Node.js ekosystémom, ktorý je založený na zdieľaní modulov a knižníc pomocou správcu balíkov NPM [61].
- Prístupnosť a zrozumiteľnosť, používa najznámejšie a najpoužívanejšie technológie – HTML (graf scény) a JavaScript (logika).
- Dá sa spustiť takmer hocikde, jediný predpoklad je nezastaralý prehliadač (s podporou WebGL, ktorú dostala väčšina prehliadačov medzi 2011-2013, vrátane IE11 [62]).

Nevýhody:

- Ťažko rozšíriteľný, slabo typovaný, minimálna podpora TypeScriptu[16].
- Výkon a rýchlosť obmedzená možnosťami prehliadača a WebGL.

Ako bude neskôr popísané v kapitole 3 Implementácia, okrem hlavného prototypu bolo tiež vytvorené demo v A-Frame. Jeho hlavnými plusmi boli veľká databáza existujúcich komponentov a vlastné rozsiahle skúsenosti s programovaním v JavaScripte (4 roky profesionálneho vývoja webov). Avšak neskôr sa jeho mínusy ukázali ako príliš veľké – vývoj v ňom značne spomaľoval jeho slabo typovaný charakter – chyby a problémy bolo náročné nájsť, ladiť a opravovať. Taktiež bol výkonnostne nepostačujúci (viď sekcia 4.2 Testovanie: Výkon).

1.6.2 Unity

Unity je multiplatformný engine pre tvorbu 2D a 3D hier od Unity Technologies [63]. Poskytuje obrovský počet nástrojov vo vizuálnom Unity Editore a programovacích rozhraniach v C#. Známe tituly vytvorené v Unity sú okrem mnohých ďalších aj *Hearthstone* [64], *Rust* [65], *Rimworld* [66], *Among Us* [67], *Hollow Knight* [65] alebo *Cuphead* [65].

1.6.2.1 DOTS

Unity Technologies od roku 2018 pracujú na prestavbe jadra Unity pomocou DOTS – *Data-Oriented Technology Stack*[17] [68]. Vďaka zmene architektúry na ECS má DOTS dovoliť vývojárom písať vysoko výkonný kód s jednoduchým multithreadingom. V súčasnosti je DOTS dostupný ako náhľadový balík vo verzii 0.5 a jeho plné vydanie 1.0 je plánované na 2022 [69].

1.6.2.2 Výhody a nevýhody

Výhody:

- Dobrý výkon vďaka kompilácii na strojový kód.
- Podpora veľmi širokej škály platforiem a zariadení.
- Obrovské množstvo online vzdelávacích zdrojov, tutoriálov, videí atď.
- Veľká komunita a aktívny vývoj[18].

Nevýhody:

[16] Plynie zo záveru sekcie 3.1 Implementácia: Demo v A-Frame.
[17] sk. dátovo zameraná zbierka technológií
[18] Vzhľadom na veľkosť, šírku využitia a closed-source povahu Unity nie je vývoj tak rýchly a flexibilný ako u menších, obzvlášť open-source enginov.

- Ťažkopádny editor spomaľuje rýchlosť vývoja.
- Prudká krivka učenia.

Hlavný prototyp voxelového enginu bude vytvorený v Unity, množstvo online zdrojov je neporovnateľné s alternatívami a veľkou výhodou sú aj vlastné, pomerne rozsiahle skúsenosti s týmto enginom, hlavne s vývojom projektov podobnej veľkosti, pre ktoré boli jeho nástroje veľmi užitočné a zlepšovali a zrýchľovali vývoj.

1.6.3 Vlastné riešenie v OpenGL

Alternatívou k používaniu existujúcich herných enginov je vytvorenie si vlastného. Samozrejme záleží na ako nízkej úrovni má vývojár ambíciu pracovať, ale najbežnejším prístupom je použiť grafické API akými sú OpenGL alebo Vulkan. Hlavná nevýhoda je pochopiteľne potreba napísať obrovské množstvo kódu pre každú časť aplikácie – od renderovania, cez pomocné matematické triedy, po samotnú hernú logiku a plno ďalších tried, vrstiev a aspektov enginu.

Výhody:

- Najlepší výkon – aplikáciu je možné podľa potreby perfektne zoptimalizovať na každej vrstve.
- Riešenie je malé, ľahké, nemusí obsahovať ani riadok kódu navyše, ktorý nevyužije.

Nevýhody:

- Veľmi pomalý vývoj, potreba všetku potrebnú logiku napísať sám, vrátane tej na nízkych vrstvách.
- Obmedzená podpora, netriviálne skompilovať pre viacero platforiem.

Vývoj vlastného herného enginu je možné odporučiť iba v týchto prípadoch: aplikácia má veľmi špecifické požiadavky, ktoré existujúce enginy nedokážu splniť alebo existuje legitímny dôvod dávať veľký dôraz na výkon a zároveň je na vývoj vyhradené dostatočne veľké množstvo času a zdrojov.

Kapitola 2
Návrh

2.1 Analýza požiadaviek

2.1.1 Funkčné požiadavky

F1 – Generácia terénu
Aplikácia bude generovať procedurálny terén. Predvolený terén by mal pôsobiť prírodne, približovať sa povrchu reálnej pahorkatiny. Parametre pre procedurálnu generáciu budú používateľovi sprístupnené, aby bolo jednoduché porovnať ich vplyv na výsledok.

F2 – Nekonečný svet
Generovaný terén bude vytváraný kontinuálne, nekonečne po všetkých osiach, obmedzený iba parametrom maximálnej vzdialenosti vykresľovania. Prioritne by mali byť vytvárané regióny v okolí hráča.

F3 – Modifikácia terénu
Aplikácia bude dovoľovať používateľovi modifikovať terén – odstraňovať a pridávať voxely do prostredia. Okrem vizuálneho ukazovateľa bude disponovať aj možnosťou výberu typu voxelu na vytvorenie.

F4 – Meshovanie
Aplikácia bude z vygenerovaného terénu schopná vytvoriť trojuholníkovú sieť. Používateľ bude mať možnosť meniť spôsoby meshovania medzi jednoduchším meshovaním s vyraďovaním plôch a pokročilejším greedy meshovaním.

F5 – Textúrovanie
Aplikácia bude renderovať vytvorenú mesh s textúrami, každý typ voxelu bude mať vlastnú textúru alebo textúry (na rôznych plochách). Vykresľovanie vo fragment shaderi bude používať pole textúr.

F6 – Ovládač hráčovej postavy
Hlavná kamera na hráčovej postave bude ovládateľná dvomi spôsobmi. Prvým bude voľný let, používateľ bude schopný lietať a prechádzať cez terén. Druhým bude štandardné ovládanie v prostredí s kolíziami, pohybom do strán a skákaním.

2.1.2 Nefunkčné požiadavky

N1 – Výkon, rýchlosť
Aplikácia má byť schopná generovať a vykresľovať prostredie dostatočne efektívne a zároveň

bez väčších kompromisov na množstve zobrazovaného obsahu.

N2 – Rozšíriteľnosť kódu
Aplikácia a jej kód má byť vytvorený spôsobom, aby mohol byť v budúcnosti jednoducho a rýchlo rozšíriteľný. Okrem kvalitného návrhu musí byť tiež zdokumentovaný.

N3 – Módovateľnosť
Aplikácia má byť navrhnutá tak, aby do budúcnosti dovoľovala jej jednoduché rozširovanie externým spôsobom, spôsobom, ktorý neovplyvní jej kódové jadro. Napr. dovolí pridávanie nových voxelov pomocou konfiguračných súborov.

2.2 Štruktúra scény

2.2.1 Chunky

Kedže nie je možné renderovať nekonečnú mesh, scénu je nutné rozdeliť na menšie časti. Vo voxelových enginoch je zaužívaný pojem *chunk* (slovensky *časť*, *kus*, *sekcia*). Každý chunk bude mať v Unity vlastný GameObject s komponentmi pre renderovanie meshe (*MeshRenderer* a *MeshFilter*) a kolízie (*MeshCollider*) a riadiacim komponentom `TerrainChunk`. Ten bude obsahovať informáciu o voxeloch v danom chunku a bude zodpovedný za vytváranie meshe.

Všetky chunky budú rovnako veľké, ich rozmer definujeme ako počet voxelov, ktoré obsahujú. Pre zjednodušenie generovania terénu budú chunky iba v jednej vertikálnej vrstve. Nech je predvolený rozmer chunkov $(8, 64, 8)$, iné veľkosti budú predmetom testovania. Dôležité je uvedomiť si, že veľkosť chunkov je kompromisom medzi rýchlosťou vykresľovania meshe a efektivitou jej vytvorenia – chunky sú dynamické, každá ich úprava spôsobuje pregenerovanie ich meshe, ktoré musí byť rozumne rýchle.

2.2.1.1 Abstrakcia voxelov

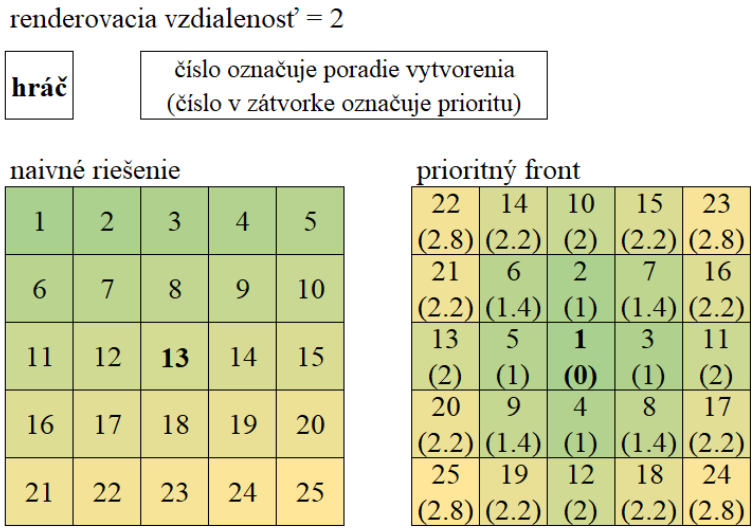
Každá inštancia `TerrainChunk` bude obsahovať trojrozmerné pole voxelov. Tie budú abstrahované do triedy `Block`, ktorá bude držať všetky potrebné informácie o danom voxeli. Táto trieda bude v prototype obsahovať iba textúry pre každú plochu, no do budúcnosti by tu mohli byť uložené aj informácie ako napr. transparentnosť, priechodnosť alebo vlastná logika. Aby sa predišlo duplicitným inštanciám a s tým spojenými problémami (veľká pamäťová záťaž, rozpor v záznamoch o rovnakom voxeli), odkazovať sa na `Block` bude iba cez ich unikátne identifikátory, ktoré sa vytvoria pri štarte aplikácie (pri štarte namiesto pri kompilácii, aby dovolili načítanie záznamov zo súboru – módovateľnosť).

2.2.2 Vytváranie a odstraňovanie chunkov

Scéna rozdelená na časti stále nemôže byť v jednej chvíli načítaná celá. Preto budú vytvorené iba tie chunky, ktoré sa nachádzajú bližšie k hráčovi ako určená maximálna vzdialenosť. Tento parameter bude nazvaný renderovacia vzdialenosť (nastavuje vzdialenosť, do ktorej sa chunky majú vykresľovať). Existuje niekoľko spôsobov, v akom poradí je možné chunky vytvárať a odstraňovať.

2.2.2.1 Naivné vytváranie po mriežke

Najpriamočiarejšia metóda vytvárania chunkov je priechod mriežky o veľkosti 1 + 2 * renderovacia vzdialenosť, kde pozícia hráča je v strede tejto mriežky. Tento prístup má avšak niekoľko nevýhod. Za prvé je výsledný tvar vytvorených chunkov okolo hráča štvorec namiesto optimálnejšieho kruhu – chunky na diagonálach, v rohoch sú najďalej, ďalej ako renderovacia vzdialenosť, a teda za rovnaký využitý výkon sú menej užitočné. Druhou nevýhodou je samotné poradie vytvárania



Obr. 2.1 Porovnanie metód vytvárania chunkov[2]

chunkov. Okrem toho, že sa ako prvé vytvoria chunky, ktoré sú najďalej od hráča a teda by ich priorita mala byť najnižšia, ich pozícia tiež spôsobuje, že majú najväčšiu šancu byť čoskoro odstránené ak by sa hráč hýbal do opačnej strany.

2.2.2.2 Vytváranie s prioritným frontom

Lepším riešením je použitie prioritného frontu. Front je dátová štruktúra, v ktorej sú prvky usporiadané v nemennom poradí. Prvky na začiatku frontu sa spracúvajú najskôr, nové sa vkladajú na koniec frontu. Prioritná fronta je zoradený zoznam dvojíc – prvkov a jeho priorít. Vkladanie nájde dvojici vhodné miesto podľa jej priority, vyberanie vráti prvok s najväčšou prioritou[1].

Pretože je vhodné najskôr vytvárať chunky najbližšie pri hráčovi, prioritný front sa ukazuje ako veľmi vhodná štruktúra – za prioritu bude považovaná vzdialenosť chunku od hráča. Vždy, keď sa hráč presunie cez hranicu chunku, prioritný front sa jednoducho prepočíta. Dokonca je pre zlepšenie výkonu možné pri výpočte vzdialenosti vynechať odmocňovanie – výsledok bude korektný, ak druhú stranu rovnice (renderovaciu vzdialenosť) umocníme.

2.2.2.3 Odstraňovanie

Aby neboli vykresľované a načítané v pamäti chunky, ktoré sú od hráča príliš ďaleko a mohla by tak byť ušetrená výpočtová sila a pamäť, pri každom prekročení hranice chunku je potrebné skontrolovať vzdialenosť existujúcich chunkov. Pri naivnom riešení je to krok navyše, no pomocou metódy s prioritným frontom budú už tieto vzdialenosti známe. Chunky, ktoré sú vzdialenejšie ako renderovacia vzdialenosť budú odstránené (resp. zrecyklované, viď nasledujúca sekcia).

[1] Najväčšia priorita môže byť najnižšia hodnota – v tomto prípade aj je, keďže ide o vzdialenosť.

[2] Úlohou tejto vizualizácie je ukázať rozdiely v poradí vytvárania chunkov. Pri prioritnom fronte by sa totiž chunky 14 a ďalej nevytvorili – nespĺňajú podmienku renderovacej vzdialenosti.

2.2.3 Object pooling

Odstraňovanie a hlavne vytváranie GameObjectov sú pomerne drahé operácie. Preto bude ako optimalizácia implementovaný tzv. object pool[3] – zásobník neaktívnych objektov, z ktorého a do ktorého budú inštancie vyberané a vkladané (recyklované) podľa aktuálnej potreby programu. Veľkosť poolu môže byť upravovaná podľa nastavenia renderovacej vzdialenosti.

2.3 Generácia terénu

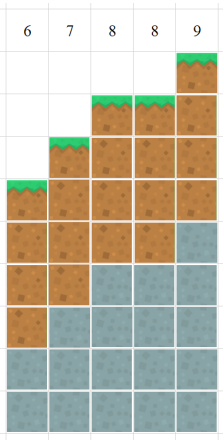
Procedurálna generácia terénu bude prebiehať v dvoch krokoch: generácia výškovej mapy a jej spracovanie. Implementácia sa bude nachádzať v triede `TerrainGenerator`. Jej úlohou je naplniť trojrozmerné pole voxelov pre všetky vytvárané inštancie `TerrainChunk`. Výškové mapy budú pre každý chunk generované individuálne – je teda dôležité, aby boli naprieč chunkami spojité.

2.3.1 Spracovanie výškovej mapy

Výšková mapa je dvojrozmerné pole celočíselných hodnôt určujúce vertikálnu pozíciu najvyššieho voxelu na konkrétnej súradnici. Spracovanie výškovej mapy znamená z tohto poľa naplniť trojrozmerné pole voxelov daného chunku. Pre tento krok použijeme na to definované pravidlá, ktoré budú v prototype implementované jednoduchým poľom s prvkami `(pocetBlockov, typBlocku)`. Pre každú súradnicu bude algoritmus postupne prechádzať týmto poľom a počnúc najvyšším voxelom nastavovať určený počet daného typu voxelu. Nech sú v prototype predvolené pravidlá `[(1, trava), (3, hlina), (999, kamen)]`.

2.3.2 Generácia výškovej mapy

Generácia výškovej mapy bude implementovaná inkrementálne, aby boli zrejmé a porovnateľné rozdiely a progres u komplikovanejších algoritmov. Výstupy z algoritmov nám pre každú súradnicu chunku vrátia hodnotu v intervale <0;1>, ktorá bude ešte následne upravená – vynásobená škálovacím parametrom (nech sa nazýva `heightMapScaler`) a zaokrúhlená na celé číslo.



Obr. 2.2 Ukážka spracovania výškovej mapy

[3] sk. *bazén* al. *fond*

2.3.2.1 Biely šum

Prvý algoritmus na generovanie výškovej mapy vyberie pre každú súradnicu náhodnú hodnotu. Použije vstavaný getter `UnityEngine.Random.value` (pseudo-náhodná hodnota v intervale <0;1>).

2.3.2.2 Simplex šum

Na generáciu Simplex šumu bude použitá knižnica FastNoiseLite, ktorá v jazyku C# implementuje rôzne typy šumu, vrátane šumu OpenSimplex2 (open-source verzia patentovaného Simplex šumu) [70].

2.3.2.3 Čiastkový Brownov pohyb

Čiastkový Brownov pohyb za použitia už naimplementovaného Simplex šumu bude fungovať presne ako bol popísaný v analýze. Všetky jeho parametre (počet oktáv, počiatočná frekvencia, počiatočná amplitúda, lakunarita a perzistencia) budú zverejnené používateľovi, ktorý ich bude mať možnosť za behu aplikácie meniť a pozorovať rozdiel v generácii terénu. Aby výsledná suma oktáv neprekročila hodnotu 1, bude po ich sčítaní vydelená $\sqrt{pocetOktav}$.

2.4 Modifikácia terénu

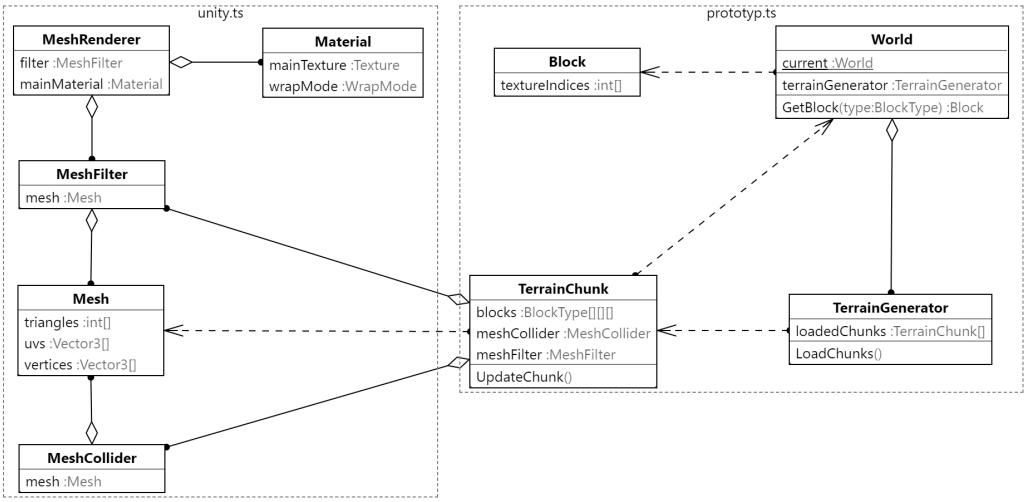
Pre používateľovu interakciu so svetom bude vytvorený komponent `PlayerTerrainInteraction`. Ten bude čakať na stlačenie tlačidla myši – ľavého pre odstránenie voxelu a pravého pre vytvorenie voxelu. Po stlačení myši sa od hráča vyšle paprsok na určenú vzdialenosť (parameter `playerRange` – dosah hráča). Toto vysielanie paprsku a kontrolu jeho pretnutia zabezpečuje Unity v metóde `Physics.Raycast()`. Po pretnutí s konkrétnym voxelom (presnejšie meshou jeho chunku) získame informácie o lúči, bode pretnutia a pretnutom objekte. Nám na zistenie pozície voxelu stačí od bodu pretnutia odčítať (alebo pričítať, ak chceme vytvoriť nový voxel) zlomok z normály meshe v tomto bode. Keďže ide o voxely, normála bude vždy kolmá na plochu na ktorú sme klikli.

2.5 Meshovanie

Za meshovanie je zodpovedný komponent `TerrainChunk`. Po naplnení jeho trojrozmerného poľa voxelov zavolá trieda `TerrainGenerator` metódu `UpdateChunk()`, ktorá spustí vybraný meshovací algoritmus. Implementované budú dva analyzované meshovacie algoritmy – s vyraďovaním tvárí a greedy meshing. Používateľ bude mať možnosť v prototype medzi týmito algoritmami prepínať. Úlohou algoritmov bude vytvoriť štruktúru `UnityEngine.Mesh` pomocou polí vrcholov a trojuholníkov. Táto štruktúra bude následne priradená do komponentu `MeshFilter` (a komponentu `MeshCollider`), ktorý ho ďalej poskytuje komponentu `MeshRenderer` zabezpečujúcej vykresľovanie. Algoritmus meshovania s vyraďovaním je pomerne jednoduchý a bude stručne popísaný v sekcii 3.1.2, greedy meshing bude detailnejšie rozobraný v sekcii 3.2.3 Implementácia: Greedy meshing.

2.5.1 Textúrovanie

Súčasťou štruktúry `UnityEngine.Mesh` je aj pole textúrových súradníc `uv` (aj ďalšie polia `uv2`, `uv3` atď. pre možnosť používať viac textúr v jednej meshi). Konkrétne textúry resp. v tomto prípade 2D pole textúr sa v Unity nastavuje pomocou materiálov, ktoré sa následne priradzujú komponentu `MeshRenderer`. Pre chunky bude vytvorený nový materiál `VoxelMaterial` s 2D



Obr. 2.3 Zjednodušený UML diagram prototypu

poľom textúr obsahujúce konkrétne obrázky (použijú sa textúry od Kenney.nl licencované CC0 – verejná doména [71]). Konkrétne meshovacie algoritmy majú odlišné spôsoby nastavovania uv súradníc. Meshovanie s vyraďovaním triviálne nastaví každému vrcholu index textúry podľa jeho typu voxelu. Greedy meshing využije jednoduchosť poľa textúr a jeho parameter *wrap mode* s hodnotou *repeat* (opakovať) pre priamočiare namapovanie uv súradníc.

Kapitola 3
Implementácia

3.1 Demo v A-Frame

V tejto sekcii bude popísaná motivácia a závery z implementácie časti navrhnutého prototypu vo webovom frameworku A-Frame. Podsekcie sa budú zaoberať konkrétnymi implementačnými krokmi.

Pre vytvorenie dema v A-Frame existovalo mnoho dôvodov, vrátane ale nie obmedzené na:

- Záujem naučiť sa novú technológiu a možnosť použiť moderný vzor ECS.
- Využitie osobných skúseností s webovými technológiami (hlavne JS a TS).
- Dostupnosť a podpora na väčšine zariadení.
- Rozsiahla databáza komponentov, ekosystém Node.js, správca balíkov NPM.
- Zdanlivá jednoduchosť a rozšíriteľnosť.

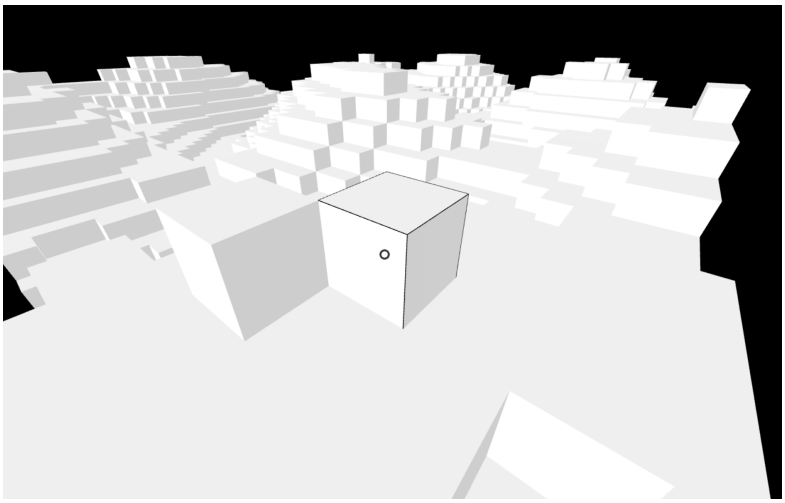
Najmä posledný bod sa ukázal byť mylný. Po implementácii ďalej popísanej funkcionality – teda pri postupnej zmene rozsahu z malej na stredne veľkú aplikáciu – sa začal vývoj drasticky spomaľovať. Kód stále väčšieho a zložitejšieho dema začal byť neudržateľný, chyby sa hľadali a ladili veľmi ťažko a pomaly. Hlavným dôvodom bola slabá typovanosť JavaScriptu a tendencia so zvyšujúcou sa komplexitou aplikácie používať stále viac frameworkovej a knižničnej logiky, ktorá avšak obsahovala výhradne funkcie a štruktúry bez typov. Riešením by bola vstavaná podpora TypeScriptu od A-Frame, ktorá bohužiaľ neexistuje. Horším, ale stále solídnym riešením by mohla byť knižnica tretej strany, ktorá A-Frame objekty zaobaľuje do typovaných abstrakcií. Takáto knižnica existuje, nazýva sa `aframe-typescript-toolkit` a funguje pomerne dobre, avšak ohľadom návrhu kódu nie je názorovo nevyhranená. Veľmi pozitívnym vývinom by bolo uchopenie sa tohto problému priamo vývojármi A-Frame.

3.1.1 Štruktúra scény

Podľa návrhu bola v grafe scény vytvorená riadiaca entita s komponentom `world`. Tá pri inicializácii vytvára nové entity, na ktoré pripája komponent `terrain-chunk`. Počet chunkov je pre nepostačujúci výkon veľmi obmedzený. Okrem toho je v scéne entita hráča s komponentmi pre pohyb, ovládanie myšou a modifikáciu terénu.

```
<a-scene>
  <!-- Entita hráča -->
  <a-entity camera fps-look-controls wasd-controls custom-controls>
    <a-crosshair />
  </a-entity>
  <!-- Vstup do logiky terénu -->
  <a-entity world />
  <!-- ... -->
</a-scene>
```

Výpis kódu 3.1 Demo v A-Frame: graf scény



Obr. 3.1 Demo v A-Frame

3.1.2 Generácia terénu

Na generáciu šumu je tiež použitá knižnica FastNoiseLite, ktorá je okrem v C# implementovaná aj v JavaScripte. Každý komponent `terrain-chunk` sa pri inicializácii dotazuje na hodnotu výškovej mapy a podľa nej vyplní daný počet voxelov. Za vytváranie trojuholníkovej siete je zodpovedný ECS systém s rovnakým názvom `terrain-chunk`[1], ktorý implementuje meshovanie s vyraďovaním plôch (viď výpis kódu 3.2).

3.1.3 Modifikácia terénu

Hráč môže editovať terén pomocou komponentu `chunk-edit`, ktorý využíva vstavaný komponent `raycaster` a jeho event `raycaster-intersection`. Z paprsku sa spôsobom popísaným v návrhu vypočíta zvolený voxel a informácia o ňom sa uloží. Pri kliknutí myšou je daný voxel odstránený a mesh jeho chunku pregenerovaná.

[1] V A-Frame sú komponenty a systémy s rovnakým menom automaticky prepojené.

```
AFRAME.registerSystem("terrain-chunk", {
  GenerateGeometry(chunk) {
    const vertices = [];
    const triangles = [];

    // Pomocná funkcia pre vytvorenie plochy voxelu
    const CreateFace = (position, direction) => {
      const indices = [
        vertices.length, // 1. trojuholník
        vertices.length + 1,
        vertices.length + 2,
        vertices.length, // 2. trojuholník
        vertices.length + 2,
        vertices.length + 3,
      ];
      triangles.push(...indices);
      const voxelVertices = MeshHelper.VoxelFaceVertices(direction);
      vertices.push(...voxelVertices.map((vertex) => vertex.add(position)));
    };

    // Pomocná funkcia pre vytvorenie voxelu
    const CreateVoxel = (position) => {
      for (let direction = 0; direction < 6; direction++) {
        // V prípade, že v tomto smere nemá voxel suseda, vytvor plochu.
        if (!chunk.HasNeighbour(position.clone(), direction)) {
          CreateFace(position, direction);
        }
      }
    };

    for (let x = 0; x < ChunkSize; x++) {
      for (let y = 0; y < ChunkSize; y++) {
        for (let z = 0; z < ChunkSize; z++) {
          const position = new THREE.Vector3(x, y, z);
          // Ak na danej súradnici nie je vzduch, vytvor voxel
          if (chunk.GetBlock(position) !== 0) {
            CreateVoxel(position);
          }
        }
      }
    }
    const geometry = new THREE.BufferGeometry();
    geometry.setIndex(triangles);
    geometry.setAttribute("position", MeshHelper.VerticesToBuffer(vertices));
    geometry.computeVertexNormals();
    return geometry;
  },
});
```

Výpis kódu 3.2 Demo v A-Frame: meshovanie s vyraďovaním plôch

3.2 Prototyp v Unity

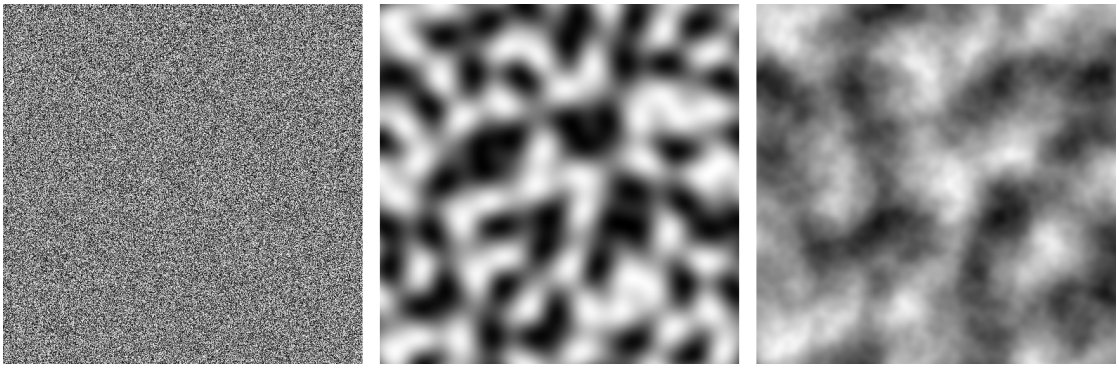
Táto sekcia sa zaoberá detailným rozborom implementácie algoritmu greedy meshing. Okrem toho vysvetlí aj kód vytvárania a odstraňovania chunkov a prejde postupnou implementáciou generácie terénu.

3.2.1 Vytváranie a odstraňovanie chunkov

Implementácia `TerrainGenerator` spĺňa náležitosti návrhu, najzložitejšia časť – generácia prioritného frontu – je popísaná vo výpise kódu 3.3.

```
1  public class TerrainGenerator
2  {
3    // Mapa pozícií na referencie TerrainChunk
4    private Dictionary<Vector2Int, TerrainChunk> loadedChunks;
5    // Prioritný front chunkov na vytvorenie
6    private PriorityQueue<int, Vector2Int> chunkLoadQueue;
7
8    // Vytvorí prioritný front podľa aktuálnych vzdialeností chunkov k hráčovi
9    private void LoadChunks() {
10     // FilterLoadedChunks() zrecykluje už nepotrebné chunky a vráti pole
       ↪  chunkov, ktoré majú zostať načítané (spĺňajú render distance)
11     boolean[,] keep = UnloadDistantChunks();
12
13     // Najprv prebehne prepočítanie chunkov, ktoré čakajú na vytvorenie
14     // V prioritnom fronte zostanú iba tie, ktoré spĺňajú RD
15     FilterLoadQueue();
16
17     // Potom sa prejdú všetky chunky, ktoré by mohlo byť potrebné vytvoriť
18     for (int x = 0; x < renderDiameter; x++) {
19       for (int z = 0; z < renderDiameter; z++) {
20         Vector2Int relativePosition = new Vector2Int(x, z) - renderDistance;
21         int sqrDistance = relativePosition.sqrMagnitude;
22         // Zatiaľ nevytvorené chunky spĺňajúce RD sú pridané do frontu
23         if (!keep[x, z] && sqrDistance <= renderDistanceSqr) {
24           Vector2Int chunkPosition = relativePosition + playerChunkPosition;
25           chunkLoadQueue.Enqueue({sqrDistance, chunkPosition});
26         }
27       }
28     }
29   }
30   // ...
31 }
```

Výpis kódu 3.3 Prototyp: prepočítavanie prioritného frontu chunkov na vytvorenie



Obr. 3.2 Porovnanie algoritmov pre generáciu výškových máp; sprava: biely šum, simplex šum, čiastkový Brownov pohyb[2]

3.2.2 Generácia terénu

Obr. 3.2, 3.3, 3.4 a 3.5 ukazujú postup implementácie stále zložitejších algoritmov pre generáciu výškových máp a ich následný vzhľad v prototype. Používateľ má v záverečnom prototype možnosť medzi týmito algoritmami prepínať. Čiastkový Brownov pohyb je implementovaný manuálne, napriek možnosti ho generovať pomocou knižnice FastNoiseLite. Parametre terénu vygenerovaného v obr. 3.5 boli zvolené za predvolené a sú nasledovné: počet oktáv = 4; počiatočná frekvencia = 0,3; počiatočná amplitúda = 3; lakunarita = 2,5; perzistencia = 0,3.

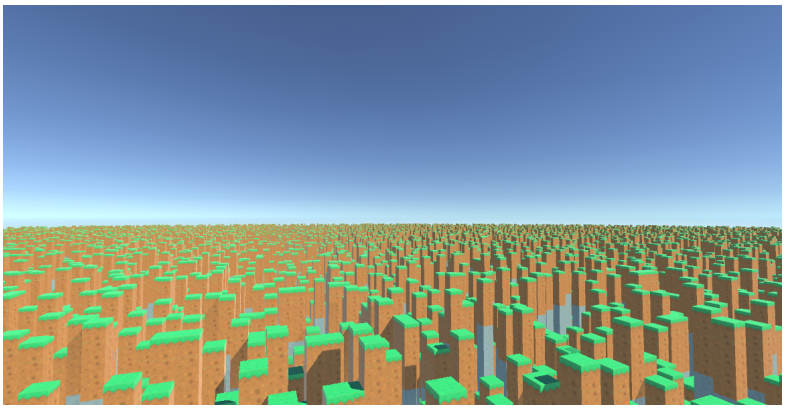
3.2.3 Greedy meshing

V tejto sekcii bude popísaná celá funkcia `TerrainChunk.UpdateChunkGreedy()` zodpovedná za eshovanie chunku pomocou algoritmu greedy meshing.

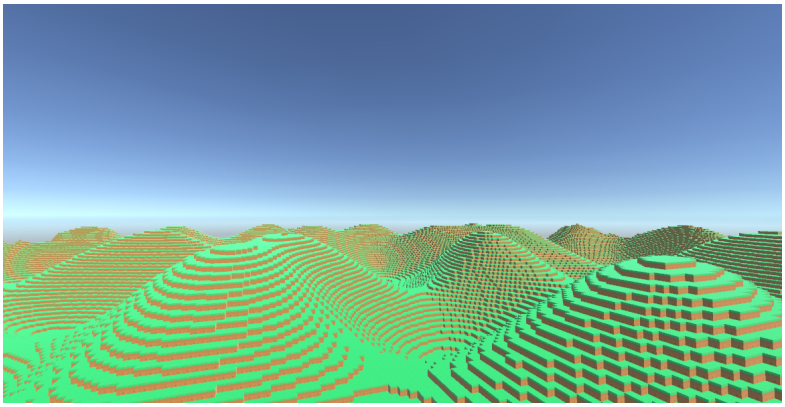
```
1  private void UpdateChunkGreedy() {
2    // Cieľom je naplnit' nasledovné štruktúry -- členy triedy Mesh
3    List<Vector3> vertices = new();
4    List<int> triangles = new();
5    List<Vector3> uvs = new();
6
7    // Pre každý smer aplikuj 2D greedy meshing
8    for (Direction direction = 0; (int)direction < 6; direction++) {
9      int axis = direction.ToAxis(); // Index osi -- x=0, y=1, z=2
10     int perpAxis1 = (axis + 1) % 3;
11     int perpAxis2 = (axis + 2) % 3;
12
13     // Premenná start označuje prvý voxel výsledného štvoruholníka
14     for (Vector3Int start = Vector3Int.zero; start[axis] < Size[axis];
       ↪  start[axis]++) {
```

Výpis kódu 3.4 Greedy meshing – začiatok funkcie

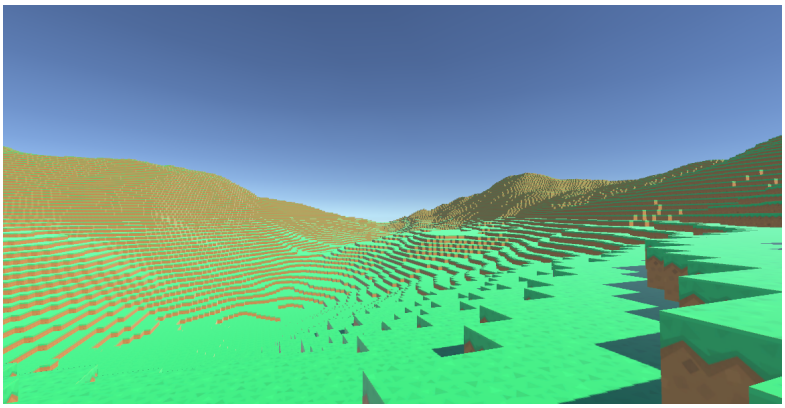
[2] Vygenerované výškové mapy majú rozmer (256, 256).



Obr. 3.3 Terén procedurálne vygenerovaný pomocou bieleho šumu



Obr. 3.4 Terén procedurálne vygenerovaný pomocou Simplex šumu



Obr. 3.5 Terén procedurálne vygenerovaný pomocou čiastkového Brownovho pohybu

Ako bolo vysvetlené v analýze, pre každý voxel si držíme stav, či bol použitý. Prvý krok je kontrola, či vôbec môže byť práve prechádzaný voxel začiatkom štvoruholníka – teda či nie je použitý, je to plný voxel a susedí v súčasnom smere s prázdnym voxelom (je viditeľný).

```
bool[,] used = new bool[Size[perpAxis1], Size[perpAxis2]];
for (start[perpAxis1] = 0; start[perpAxis1] < Size[perpAxis1];
↪  start[perpAxis1]++) {
  for (start[perpAxis2] = 0; start[perpAxis2] < Size[perpAxis2];
  ↪  start[perpAxis2]++) {
    int startBlockType = GetBlock(start);

    if (used[start[perpAxis1], start[perpAxis2]]
      || startBlockType == 0
      || GetBlock(start + direction.ToVector3Int()) != 0) {
      continue;
    }
```

Výpis kódu 3.5 Greedy meshing – kontrola počiatočného voxelu

Štvoruholník sa algoritmus pokúsi rozšíriť v 1. kolmej osi. Bude ho rozširovať, kým nenarazí na prvý voxel nespĺňajúci podmienky – je rovnaký typ ako počiatočný voxel, je viditeľný a nie je použitý.

```
// Premenná faceSize označuje veľkosť výsledného štvoruholníka od start
Vector3Int faceSize = Vector3Int.one;

// Rozširovanie štvoruholníku v 1. kolmej osi
Vector3Int end = start;
for (end[perpAxis1]++; end[perpAxis1] < Size[perpAxis1]; end[perpAxis1]++) {
  if (startBlockType != GetBlock(end)
    || GetBlock(end + direction.ToVector3Int()) != 0
    || used[end[perpAxis1], end[perpAxis2]]) {
    break;
  }
  faceSize[perpAxis1]++;
}
```

Výpis kódu 3.6 Greedy meshing – rozšírenie v 1. kolmej osi

Rovnaký postup rozširovania je aj v 2. kolmej osi, no je k nemu ešte potrebné prechádzať a kontrolovať namiesto individuálnych celé riadky voxelov.

```
// Rozširovanie štvoruholníku v 2. kolmej osi
end = start;
for (end[perpAxis2]++; end[perpAxis2] < Size[perpAxis2]; end[perpAxis2]++) {
  for (end[perpAxis1] = start[perpAxis1]; end[perpAxis1] <= Size[perpAxis1];
  ↪  end[perpAxis1]++) {
    if (startBlockType != GetBlock(end)
      || GetBlock(end + direction.ToVector3Int()) != 0
      || used[end[perpAxis1], end[perpAxis2]]) {
      break;
    }
  }

  if (end[perpAxis1] - start[perpAxis1] != faceSize[perpAxis1]) {
    break;
  }
  faceSize[perpAxis2]++;
}
```

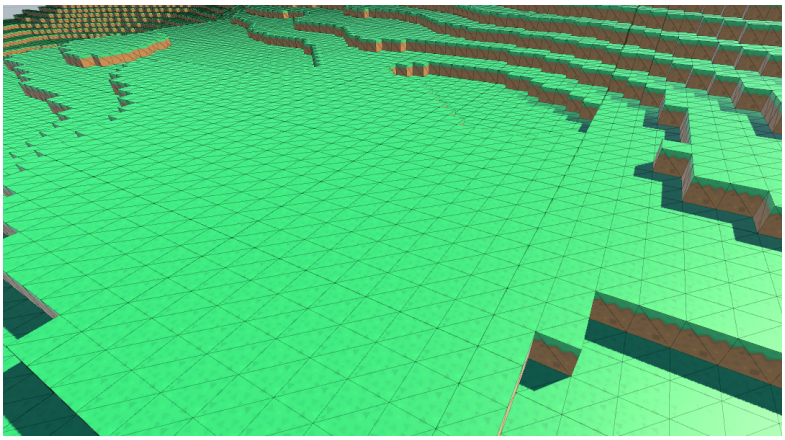
Výpis kódu 3.7 Greedy meshing – rozšírenie v 2. kolmej osi

Nakoniec sú pre daný štvoruholník vytvorené vrcholy, trojuholníky a uv súradnice. Všetky voxely, ktoré boli jeho súčasťou sú označené ako použité.

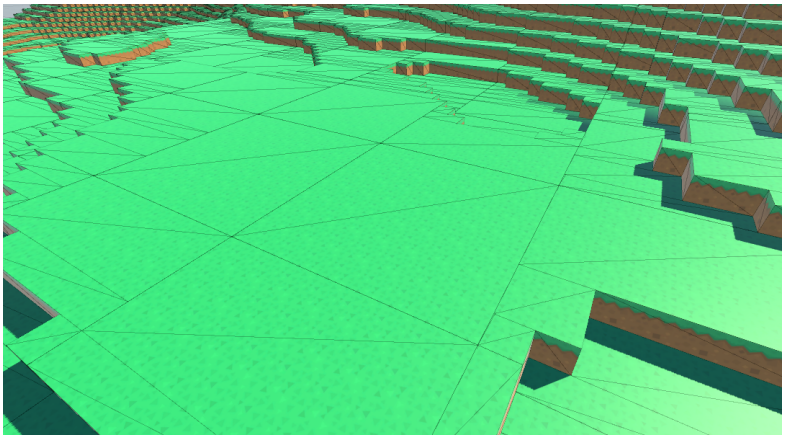
```
CreateFace(direction, start, faceSize, startBlockType);

for (int x = 0; x < faceSize[perpAxis1]; x++) {
  for (int y = 0; y < faceSize[perpAxis2]; y++) {
    used[start[perpAxis1] + x, start[perpAxis2] + y] = true;
  }
}
```

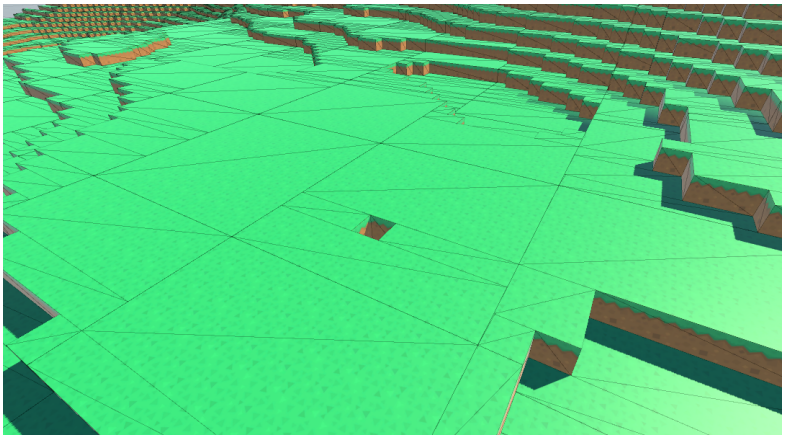
Výpis kódu 3.8 Greedy meshing – vytvorenie štvoruholníka



Obr. 3.6 Meshovanie s vyraďovaním plôch



Obr. 3.7 Greedy meshing



Obr. 3.8 Greedy meshing, modifikácia terénu

Kapitola 4
Testovanie

Táto kapitola sa bude zameriavať na testovanie funkčnosti a porovnanie výkonu dema a prototypu pri použití rôznych algoritmov a parametrov. Použité testovacie zariadenia a ich technické parametre sa nachádzajú v tabuľkách 4.1 a 4.2.

4.1 Funkčnosť

Prototyp bol testovaný na funkčnosť na zariadení A na platformách Windows (Windows 10 Home 21H2), a Linux (Manjaro KDE Linux 5.15.32-1) a na zariadení B na platforme macOS (macOS Big Sur 11.6.4), na žiadnom neboli nájdené väčšie problémy. Všetky menšie chyby (problémy so vstupom, používateľským rozhraním, rozlíšením) boli promptne opravené.

4.2 Výkon

V tejto sekcii budú popísané a porovnané výsledky testovania výkonu dema a prototypu. Testovanie výkonu prebehlo výhradne na zariadení A. Testovanou metrikou bol priemerný počet snímkov za sekundu (FPS) po dobu 3 sekúnd pri určených parametroch al. nastaveniach.

Do prototypu bola implementovaná možnosť vytvárať konzistentné testovacie prostredie, ktorého súčasťou boli jednoduché testy výkonu – odstránenie všetkých chunkov scény a ich znovuvytvorenie a testovanie FPS pri konkrétnom počte upravených chunkov za frame, ktorý spôsobil

Názov	*Dell Inspiron 15 G5 (5587)*
CPU	*Intel Core i7-8750H*
GPU	*NVIDIA GeForce GTX 1050 Ti*
RAM	*16GB DDR4*

Tabuľka 4.1 Testovacie zariadenie A

Názov	*Apple MacBook Pro (13-inch, 2017)*
CPU	*Intel Core i5*
GPU	*Intel Iris Plus Graphics 640*
RAM	*16GB LPDDR3*

Tabuľka 4.2 Testovacie zariadenie B

pregenerovanie ich meshí (vo výsledkoch testov označené ako „chunk updates“ – aktualizácie chunkov).

4.2.1 Demo v A-Frame

Výkon dema v A-Frame bol značne nepostačujúci, čo bolo citeľné už pri jeho vývoji. Aj veľmi jednoduchá scéna – 30x30 chunkov o veľkosti (8, 8, 8), teda spolu scéna s rozmerom (240, 8, 240) voxelov – hraničila s nepoužiteľnosťou. Priemerné FPS bolo 31 a často sa dostávalo pod kritickú hodnotu 30.

Veľkosť scény	Priemerné FPS
(80, 80)	*56*
(160, 160)	*40*
(240, 240)	*31*
(320, 320)	*23*

Tabuľka 4.3 Výsledky testovania výkonu dema v A-Frame[1]

4.2.2 Meshovacie algoritmy

Pre tento test bolo testovacie prostredie spustené na scéne o veľkosti (320, 64, 320) voxelov s rozmerom chunkov (8, 64, 8), porovnaný bol meshovací algoritmus s vyraďovaním plôch a greedy meshing. Výsledky sú spriemerované hodnoty FPS naprieč tromi iteráciami testovania.

Algoritmus	Chunk updates				
	0	1	2	3	5
Greedy	*258*	*212*	*167*	*129*	*82*
S vyraďovaním	*222*	*164*	*106*	*85*	*52*

Tabuľka 4.4 Výsledky testovania výkonu meshovacích algoritmov (v FPS)

Výsledok tohto testu je pomerne nečakaný, keďže samotný algoritmus greedy meshing by mal byť zložitejší na výpočet a tým pádom pomalší. Do budúcna by mohlo byť zaujímavé sprofilovať špecifické kroky programu, volania funkcií, napríklad dobu trvania prepísania starej meshe za novú na GPU, alebo priamo dĺžky volaní konkrétnych meshovacích funkcií.

V každom prípade je zrejmé, že sa implementácia greedy meshingu výrazne oplatila. Okrem rýchlejšieho vykresľovania bolo zefektívnené aj vytváranie chunkov.

[1] Použitý prehliadač bol Mozilla Firefox 100.0.

4.2.3 Rozmer chunkov

Pre tento test bolo testovacie prostredie spustené na scéne o veľkosti (288, 64, 288) voxelov, maximálny počet vytvorených chunkov za frame bol nastavený na 5, porovnávané boli rozmery chunkov. Priamo porovnávať možno iba výsledky v stĺpcoch Chunk updates 0 a Znovuvytvorenie scény, keďže testovacie prostredie pregenerovanie meshí vynucovalo na rozdielnych chunkoch, nie na stále rovnakých konkrétnych voxeloch. Napríklad zmeny v oblasti 10x10 voxelov by mohli spustiť znovuvytvorenie meshe deviatich chunkov o veľkosti (4, 64, 4) ale iba jednej o veľkosti (16, 64, 16).

Rozmer chunkov	Chunk updates					Znovuvytvorenie scény
	0	1	2	3	5	
(4, 64, 4)	*230*	*211*	*196*	*182*	*159*	*170*
(6, 64, 6)	*256*	*229*	*197*	*168*	*115*	*113*
(8, 64, 8)	*283*	*221*	*165*	*122*	*77*	*74*
(12, 64, 12)	*293*	*172*	*94*	*66*	*40*	*37*
(16, 64, 16)	*305*	*105*	*57*	*39*	*23*	*22*
(24, 64, 24)	*306*	*55*	*28*	*18*	*11*	*10*

Tabuľka 4.5 Výsledky testovania výkonu veľkostí chunkov (v FPS)

Ako bolo možné očakávať, väčšie chunky sú efektívnejšie na vykresľovanie, ale pomalšie pri znovuvytváraní ich meshí. Výber predvoleného rozmeru chunku je teda kompromis medzi týmito dvomi faktormi.

Záver

Cieľom tejto práce bolo popísať návrh a implementáciu voxelového enginu a zanalyzovať pre to potrebné teoretické poznatky.

Práca ukázala existujúce voxelové hry a enginy, ich vlastnosti slúžili ako motivácia pre konkrétne požiadavky prototypu. Teoreticky vyložila algoritmy pre tvorbu trojuholníkovej siete rôznej komplexity a kvality výsledku, až po konštantný koeficient od optimálnosti vďaka greedy meshingu. Analyzovala aj najčastejšie programovacie paradigmy používané vo vývoji hier, vrátane stále modernejšieho Entity-Component-System. Avšak záver z rozboru technológií vhodných pre tvorbu prototypu bol, že najvhodnejším softvérom bude Unity, ktorá má v súčasnosti možnosti programovania pomocou ECS stále veľmi obmedzené.

Podľa analýzy boli sformulované funkčné a nefunkčné požiadavky, ktoré ďalej pomohli vytvoriť návrh prototypu. Vytvorili sa abstrakcie a postupy pre štruktúru scény, generáciu a modifikáciu terénu a meshovanie a textúrovanie chunkov.

Implementácia sa pevne držala návrhu, jej výsledkom je multiplatformný prototyp, ktorý sa nachádza v priloženom médiu. Okrem zapracovania všetkých požiadaviek sú najzložitejšie časti, algoritmy popísané a vysvetlené. Časť návrhu bola tiež implementovaná ako demo vo webovom frameworku A-Frame, a motivácia a záver z tohto počínania boli vysvetlené.

Nakoniec boli prototyp a demo testované, jednak na funkčnosť na najpoužívanejších platformách a taktiež na výkon pri rôznych parametroch a nastaveniach, výsledky z testovania boli objasnené.

Do budúcna by mohol byť prototyp rozšírený o ďalšiu funkcionalitu, menovite procedurálnu generáciu budov a vegetácie, príšery a zvieratá s umelou inteligenciou vrátane schopnosti vyhľadávania ciest alebo voxelovú simuláciu tekutín (vody).

Bibliografia

1. RODIN, Daniil; ELBER, Gershon. Multi-modal Non-linear Continuous 3D Presentations. *Journal of Physics: Conference Series*. 2020, roč. 1518, s. 012028. Dostupné z DOI: 10.1088/1742-6596/1518/1/012028.
2. PERSSON, Markus. Cave game tech test. [video]. *YouTube* [online]. 2009 [cit. 2022-04-21]. Dostupné na: https://www.youtube.com/watch?v=F9t3FREAZ-k.
3. WIKIPEDIA CONTRIBUTORS. *List of best-selling video games* [online]. [B.r.] [cit. 2022-04-21]. Dostupné na: https://en.wikipedia.org/wiki/List_of_best-selling_video_games.
4. PERSSON, Markus. *The origins of Minecraft* [online]. 2009 [cit. 2022-04-21]. Dostupné na: https://web.archive.org/web/20091102174928/http://notch.tumblr.com/post/227922045/the-origins-of-minecraft.
5. BARTH, Zach. About Infiniminer [online]. [B.r.] [cit. 2022-04-21]. Dostupné na: https://www.zachtronics.com/infiniminer/.
6. DAVIES, Marsh. Blockbuster - The Making of Minecraft [online]. 2012 [cit. 2022-04-21]. Dostupné na: https://www.pcgamer.com/the-making-of-minecraft/.
7. KNAPP, Alex. Minecraft Has Sold Over Four Million Copies - And Officially Launches Next Week [online]. 2011 [cit. 2022-04-21]. Dostupné na: https://www.forbes.com/sites/alexknapp/2011/11/08/minecraft-has-sold-over-four-million-copies-and-officially-launches-next-week/.
8. PERSSON, Markus. *Hiring some people, getting an office, and all that!* [Online]. 2010 [cit. 2022-04-21]. Dostupné na: https://web.archive.org/web/20100908235613/http://notch.tumblr.com/post/1075326804/hiring-some-people-getting-an-office-and-all-that.
9. PERSSON, Markus. *Och med dom orden så passar jag micken.* [Online]. 2011 [cit. 2022-04-21]. Dostupné na: https://web.archive.org/web/20111227053605/http://notch.tumblr.com/post/13633493969/och-med-dom-orden-sa-passar-jag-micken.
10. COLINI, Henrique. *Minecraft Timeline* [online]. [B.r.] [cit. 2022-04-21]. Dostupné na: https://minecraft-timeline.github.io/.
11. CURRY, David. Minecraft Revenue and Usage Statistics (2022) [online]. 2022 [cit. 2022-04-21]. Dostupné na: https://www.businessofapps.com/data/minecraft-statistics/.
12. STUART, Keith; HERN, Alex. Minecraft sold: Microsoft buys Mojang for $2.5bn [online]. 2014 [cit. 2022-04-21]. Dostupné na: https://www.theguardian.com/technology/2014/sep/15/microsoft-buys-minecraft-creator-mojang-for-25bn.

13. PCGAMESN. How many Minecraft players are there? [Online]. 2020 [cit. 2022-04-21]. Dostupné na: https://www.pcgamesn.com/minecraft/minecraft-player-count.

14. METACRITIC.COM; METACRITIC REVIEWERS. *Minecraft for PC Reviews* [online]. [B.r.] [cit. 2022-04-21]. Dostupné na: https://www.metacritic.com/game/pc/minecraft.

15. OJALA, Andreas. *25 BEST 1.18 Minecraft Seeds: RANKED (2022)* [online]. [B.r.] [cit. 2022-04-22]. Dostupné na: https://whatifgaming.com/best-1-18-minecraft-seeds/.

16. JARVEY, Natalie. Riot Games Leads Investment in Hypixel Game Studio [online]. 2018 [cit. 2022-04-21]. Dostupné na: https://www.hollywoodreporter.com/business/digital/riot-games-leads-investment-hypixel-game-studio-1168889/.

17. PHILLIPS, Tom. Hytale is a brand new game from giants of the Minecraft community, backed by Riot [online]. 2018 [cit. 2022-04-21]. Dostupné na: https://www.eurogamer.net/hytale-is-a-brand-new-game-from-giants-of-the-minecraft-community.

18. FISHER, Tyler. First Trailer of 'Minecraft' Inspired Game 'Hytale' Has 31 Million Views In Less Than 30 Days [online]. 2019 [cit. 2022-04-21]. Dostupné na: https://comicbook.com/gaming/news/hytale-trailer-minecraft-pc/.

19. ROSSIGNOL, Jim. Cube World Looks Chunky [online]. 2011 [cit. 2022-04-21]. Dostupné na: https://www.rockpapershotgun.com/cube-world-looks-chunky.

20. HERNANDEZ, Patricia. The Current Status Of Cube World, And Why Fans Are Worried About It [online]. 2013 [cit. 2022-04-21]. Dostupné na: https://kotaku.com/the-current-status-of-cube-world-and-why-fans-are-worr-1449026160.

21. *"Vaporware.", Merriam-Webster.com Dictionary* [online]. [B.r.] [cit. 2022-04-22]. Dostupné na: https://www.merriam-webster.com/dictionary/vaporware.

22. HERNANDEZ, Patricia. Cube World Hasn't Been Updated In Years, But Some Fans Still Play Every Day [online]. 2017 [cit. 2022-04-21]. Dostupné na: https://kotaku.com/cube-world-hasnt-been-updated-in-years-but-some-fans-s-1795021131.

23. CUNNINGHAM, James. Review: Cube World [online]. 2019 [cit. 2022-04-21]. Dostupné na: https://hardcoregamer.com/reviews/review-cube-world/358830/.

24. METACRITIC.COM; METACRITIC REVIEWERS. *Cube World (2019) – User Reviews* [online]. [B.r.] [cit. 2022-04-21]. Dostupné na: https://www.metacritic.com/game/pc/cube-world-2019/user-reviews.

25. TUXEDO LABS. *Teardown on Steam* [online]. [B.r.] [cit. 2022-04-21]. Dostupné na: https://store.steampowered.com/app/1167630/Teardown/.

26. PINHEIRO, Daniel. Destructible puzzler Teardown to see a full release this month [online]. 2022 [cit. 2022-04-21]. Dostupné na: https://www.pcinvasion.com/destructible-puzzler-teardown-release-month/.

27. HYTALE TEAM. Summer 2021 Development Update [online]. 2021 [cit. 2022-04-22]. Dostupné na: https://hytale.com/news/2021/7/summer-2021-development-update.

28. KEEN SOFTWARE HOUSE. *Space Engineers on Steam* [online]. [B.r.] [cit. 2022-04-21]. Dostupné na: https://store.steampowered.com/app/244850/Space_Engineers/.

29. ROSA, Marek. Space Engineers: Release Date for the Major Overhaul of Survival & Going out of Early Access [online]. 2019 [cit. 2022-04-21]. Dostupné na: https://blog.marekrosa.org/2019/02/space-engineers-release-date.html.

30. KEEN SOFTWARE HOUSE. *VRAGE* [online]. [B.r.] [cit. 2022-04-21]. Dostupné na: https://web.archive.org/web/20130911233446/http://www.keenswh.com/vrage.html.

31. Hopson - YouTube [online]. [B.r.] [cit. 2022-04-21]. Dostupné na: https://www.youtube.com/channel/UCeQhZOvNKSBRUOMdg7V44wA.

32. HOPSON, Matthew. Coding Minecraft in One Week - C++/OpenGL Programming Challenge. [video]. *YouTube* [online]. 2017 [cit. 2022-04-21]. Dostupné na: https://www.youtube.com/watch?v=Xq3isov6mZ8.
33. HOPSON, Matthew. Let's Code A Multiplayer Voxel Game in C++ - The Engine. [video]. *YouTube* [online]. 2019 [cit. 2022-04-21]. Dostupné na: https://www.youtube.com/watch?v=4Rg1RriQZ9Q.
34. jdh - YouTube [online]. [B.r.] [cit. 2022-04-21]. Dostupné na: https://www.youtube.com/c/jdhvideo.
35. JDH. Making Minecraft from scratch in 48 hours (NO GAME ENGINE). [video]. *YouTube* [online]. 2020 [cit. 2022-04-21]. Dostupné na: https://www.youtube.com/watch?v=4O0_-1NaWnY.
36. JDH. I programmed Minecraft from scratch... again.. [video]. *YouTube* [online]. 2022 [cit. 2022-04-21]. Dostupné na: https://www.youtube.com/watch?v=9f7Hu1ZNYT8.
37. SimonDev - YouTube [online]. [B.r.] [cit. 2022-04-21]. Dostupné na: https://www.youtube.com/channel/UCEwhtpXrg5MmwlHO4ANpL8A.
38. SIMONDEV. I made an EVEN BETTER Minecraft. [video]. *YouTube* [online]. 2021 [cit. 2022-04-21]. Dostupné na: https://www.youtube.com/watch?v=MQt0exToUEY.
39. RODRIGUEZ, Ivelisse. Tech Stack: Definition + 9 Examples from the World's Top Brands [online]. 2021 [cit. 2022-04-21]. Dostupné na: https://blog.hubspot.com/customers/auditing-your-companys-tech-stack-or-platform-apps.
40. GamesWithGabe - YouTube [online]. [B.r.] [cit. 2022-04-21]. Dostupné na: https://www.youtube.com/c/GamesWithGabe.
41. AMBROSIO, Gabe. Adding Multiplayer, Colored Lights, and More | Coding Minecraft Devlog #2. [video]. *YouTube* [online]. 2022 [cit. 2022-04-21]. Dostupné na: https://www.youtube.com/watch?v=UAUdIQZKV88.
42. HOGAN, Sam. I Made Minecraft in 24 Hours. [video]. *YouTube* [online]. 2020 [cit. 2022-04-21]. Dostupné na: https://www.youtube.com/watch?v=Nj8gt_92c-M.
43. LEGENDOFLINQ. Minecraft in Unity 3D - One-Week Programming Challenge. [video]. *YouTube* [online]. 2013 [cit. 2022-04-21]. Dostupné na: https://www.youtube.com/watch?v=4Rg1RriQZ9Q.
44. THE MINETEST TEAM. *Minetest – Open source voxel engine* [online]. [B.r.] [cit. 2022-04-26]. Dostupné na: https://www.minetest.net/.
45. WARDY, Ruben. *ContentDB* [online]. [B.r.] [cit. 2022-04-26]. Dostupné na: https://content.minetest.net/.
46. VOXEL.JS PROJECT. *voxel.js * blocks in yo browser* [online]. [B.r.] [cit. 2022-04-26]. Dostupné na: http://www.voxeljs.com/.
47. VOXEL.JS PROJECT. *Voxel.js Project: Repositories* [online]. [B.r.] [cit. 2022-04-26]. Dostupné na: https://github.com/orgs/voxel/repositories?q=voxel-.
48. LYSENKO, Mikola. Meshing in a Minecraft Game [online]. 2012 [cit. 2022-04-21]. Dostupné na: https://0fps.net/2012/06/30/meshing-in-a-minecraft-game/.
49. BLACK, Paul. *"greedy algorithm", in Dictionary of Algorithms and Data Structures* [online]. 2005 [cit. 2022-04-21]. Dostupné na: https://xlinux.nist.gov/dads/HTML/greedyalgo.html.
50. RODRIGUEZ, Ivelisse. *Tech Stack: Definition + 9 Examples from the World's Top Brands* [online]. 2021 [cit. 2022-04-21]. Dostupné na: https://blog.hubspot.com/customers/auditing-your-companys-tech-stack-or-platform-apps.

51. PERLIN, Ken. Making Noise [online]. 1999 [cit. 2022-04-29]. Dostupné na: https://web.archive.org/web/20001018094127/http://www.noisemachine.com/talk1/index.html.
52. GUSTAVSON, Stefan. Simplex noise demystified [online]. 2005 [cit. 2022-04-29]. Dostupné na: https://weber.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf.
53. GROENEBOOM, N. E.; DAHLE, H. Introducing Gamer: A fast and accurate method for ray-tracing galaxies using procedural noise. *The Astrophysical Journal*. 2014, roč. 783, č. 2, s. 138. Dostupné z DOI: 10.1088/0004-637x/783/2/138.
54. ANON. Fractional Brownian Motion [online]. 2010 [cit. 2022-04-29]. Dostupné na: https://code.google.com/archive/p/fractalterraingeneration/wikis/Fractional_Brownian_Motion.wiki.
55. GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. *Design patterns: elements of reusable object-oriented software*. Boston: Pearson Education, 1994. ISBN 9780321700698.
56. NYSTROM, Robert. *Game Programming Patterns*. Genever — Benning, 2014. ISBN 9780990582915.
57. A-FRAME. *Introduction – A-Frame* [online]. [B.r.] [cit. 2022-04-21]. Dostupné na: https://aframe.io/docs/1.3.0/introduction/.
58. MOZILLA MIXED REALITY. *Introducing A-Frame: Building Blocks for WebVR* [online]. 2015 [cit. 2022-04-21]. Dostupné na: https://blog.mozvr.com/introducing-aframe/.
59. THREE.JS AUTHORS. *three.js* [online]. [B.r.] [cit. 2022-04-27]. Dostupné na: https://github.com/mrdoob/three.js/.
60. TAVARES, Gregg. WebGL Fundamentals [online]. 2012 [cit. 2022-04-27]. Dostupné na: https://www.html5rocks.com/en/tutorials/webgl/webgl_fundamentals/.
61. A-FRAME. *A-Frame Registry* [online]. [B.r.] [cit. 2022-04-21]. Dostupné na: https://aframe.io/aframe-registry/.
62. DEVERIA, Alexis. *Can I use WebGL?* [Online]. [B.r.] [cit. 2022-04-21]. Dostupné na: https://caniuse.com/?search=WebGL.
63. UNITY TECHNOLOGIES. *Unity Real-Time Development Platform | 3D, 2D VR & AR Engine* [online]. [B.r.] [cit. 2022-04-28]. Dostupné na: https://unity.com/.
64. WILSON, Jason. Even Hearthstone runs on Unity — and that's why it's already on iPad [online]. 2014 [cit. 2022-04-28]. Dostupné na: https://venturebeat.com/2014/04/24/even-hearthstone-runs-on-unity-and-thats-why-its-already-on-ipad/.
65. G2A NEWS TEAM. 10 Games made with Unity Engine [online]. 2021 [cit. 2022-04-28]. Dostupné na: https://www.g2a.com/news/features/games-made-with-unity-engine/.
66. SYLVESTER, Tynan. *Re: Engine Used?* [Online]. 2013 [cit. 2022-04-28]. Dostupné na: https://ludeon.com/forums/index.php?topic=258.0.
67. INNERSLOTH. *Among Us by Innersloth* [online]. [B.r.] [cit. 2022-04-28]. Dostupné na: https://innersloth.itch.io/among-us.
68. UNITY TECHNOLOGIES. *DOTS - Unity's new multithreaded Data-Oriented Technology Stack* [online]. [B.r.] [cit. 2022-04-28]. Dostupné na: https://unity.com/dots.
69. GIBERT, Laurent. DOTS Development Status And Next Milestones - December 2021 [online]. 2021 [cit. 2022-04-28]. Dostupné na: https://forum.unity.com/threads/dots-development-status-and-next-milestones-december-2021.1209727/.
70. PECK, Jordan. *Auburn/FastNoiseLite: Fast Portable Noise Library - C# C++ C Java(Script) HLSL* [online]. [B.r.] [cit. 2022-05-03]. Dostupné na: https://github.com/Auburn/FastNoiseLite.

71. KENNEY. *Kenney • Voxel pack* [online]. [B.r.] [cit. 2022-05-03]. Dostupné na: `https://kenney.nl/assets/voxel-pack`.

Obsah priloženého média

- `readme.txt` ..... stručný popis obsahu média
- `bin` ..... adresár so spustiteľnou formou implementácie
  - `windows`
  - `macos`
  - `linux`
- `src`
  - `impl` ..... zdrojové kódy implementácie
  - `thesis` ..... zdrojová forma práce vo formáte LaTeX
- `thesis.pdf` ..... text práce vo formáte PDF