FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Systém pro generování procedurálního voxelového 2.5D terénu
Student:	Martin Ševela
Vedoucí:	Ing. Petr Pauš, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

Cílem práce je navrhnout, implementovat a optimalizovat systém pro generování voxelového terénu s koncovým využitím jako součásti 2.5D hry. Vygenerovaný terén musí být zajímavý z pohledu hráče a generátor zároveň musí být dostatečně rychlý pro generování v reálném čase. Prototyp bude naprogramován ve vhodném enginu.

1. Analyzujte a vyberte technické možnosti implementace (knihovny, frameworky,atd.).
2. Analyzujte vhodné algoritmy pro generování terénu.
3. Navrhněte základní podobu algoritmu pro generování terénu.
4. Implementujte generátor.
5. Minimalizujte časovou a paměťovou náročnost generátoru a prototyp otestujte.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 21. listopadu 2022 v Praze.

Bakalářská práce

GENERÁTOR 2.5D VOXELOVÉHO TERÉNU

Martin Ševela

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Petr Pauš, Ph.D.
11. května 2023

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Rád bych vyjádřil své poděkování především Ing. Petrovi Paušovi, Ph.D. za jeho vstřícný přístup k vedení projektu a mnoho užitečných poznatků.

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, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 11. května 2023 .....

Abstrakt

Předmětem této bakalářská práce je návrh, implementace a optimalizace prototypu hloubkově omezeného procedurálního generátoru voxelového terénu v herním enginu Godot. Její součástí je i analýza technických prostředků využitelných pro realizaci zadání. Generátor využívá sémantického rozdělení terénu do jednodušších vrstev a jejich následného spojení na základě rozdělení biomů. Výsledný terén je možné využít pro tvorbu herního prostředí, které je možné dále rozšiřovat pomocí systému vkládání modelů. Systém zároveň poskytuje uživatelské rozhraní, které lze využít k ovládání, monitorování a vizualizaci generátoru.

Klíčová slova generátor terénu, voxelový terén, procedurální generování, Godot Engine, C#, optimalizace

Abstract

The subject of this bachelor's thesis is the design, implementation, and optimization of a prototype depth-limited procedural voxel terrain generator in the Godot game engine. It also includes an analysis of technical means that can be utilized for the realization of the project. The generator employs a semantic division of terrain into simpler layers and their subsequent combination based on biome distribution. The resulting terrain can be used for creating a game environment which can be further expanded using a model insertion system. The system also provides a user interface that can be used for control, monitoring, and visualization of the generator.

Keywords terrain generator, voxel terrain, procedural generation, Godot Engine, C#, optimalization

Seznam zkratek

VT	Voxelový terén
PG	Procedurální generování
GC	Garbage collector
GUI	Graphical user interface
Rx	Reactive Extensions
SIMD	Single instruction, multiple data

Úvod

Voxelová reprezentace 3D modelů není novým vynálezem – již dlouho se využívala například pro analýzu a vizualizaci vícedimenzionálních dat. Po úspěchu několika her na začátku 2. desetiletí 21. století, které voxelovovou formu terénu využívají jako centrální komponentu herního světa, se takové systémy značně rozšířily i do herního průmyslu. Umožnují totiž efektivně docílit plně destruktibilního terénu, který je díky své systematické struktuře možno automaticky generovat. V kombinaci s vhodným systémem generování je tak možné dosáhnout velikosti a komplexity herních světů, které manuální tvorbou nelze konkurovat.

Pro vývojáře, který by rád ve svém projektu voxelový terén využil, existuje obrovské množství zdrojů. Naprostá většina z nich se ovšem zaměřuje pouze na jeden aspekt voxelového terénu. Zkombinovat je dohromady v jeden funkční celek je netriviální a možství zdrojů pro tento proces je relativně limitované, obvzáště s ohledem na konkrétní implementaci.

Tato práce si dává za cíl pokusit se o návrh a impementaci generátoru voxelového terénu, který je dostatečně flexibilní pro generování terénu zajímavého pro potenciálního hráče a zároveň dostatečně efektivní pro generování terénu v reálném čase. Generátor ovšem bude implementován jako 2.5D, což v kontextu této práce znamená, že bude mít omezenou hloubku. Koncepty v něm použité budou však přenositelné i na 3D.

Práce je rozdělena do čtyř hlavních kapitol. V té první probíhá analýza konceptuálního a technického charakteru, která si klade za cíl zjistit, jaké jsou možnosti a způsoby implementace. Druhá kapitola se věnuje abstraktnímu návrhu systému generování. Jsou v ní definovány požadavky na generátor, na základě kterých byl proveden hrubý návrh. Popis samotné implementace se pak nachází v třetí kapitole, kde jsou detailněji rozebrány její nejdůležitější aspekty a postup při jejich optimalizaci. V čtvrté a poslední kapitole je provedeno testování a ověření generátoru s ohledem na jeho výkonnost.

Kapitola 1
Analýza

V této počáteční kapitole se nachází přehled užitečných datových struktur, algoritmů, přístupů a nástrojů použitelných k tvorbě voxelových 3D aplikací a následně využitých pro vývoj prototypu. Konceptuální část se zabývá algoritmy a obecnými přístupy k tvorbě voxelového terénu. Technická část se pak zaměřuje na konkrétní knihovny a nástroje.

1.1 Konceptuální

1.1.1 Voxelový terén

Voxelový terén představuje speciální typ terénu, který se používá v počítačových simulacích a hrách. Na rozdíl od tradičních polygonových 3D modelů je voxelový terén definován volumetricky. To znamená, že neobsahuje pouze informace o povrchu terénu, ale také o jeho objemu, což poskytuje řadu zajímavých možností.

Existuje mnoho variant voxelového terénu, které lze reprezentovat a uložit v paměti různými způsoby. V rámci této práce se zaměříme na „binární“ terén, kde každý voxel představuje jednu krychli, kterou nazýváme blok.

▶ **Definice 1.1** (Voxel)**.** *Voxel je základní stavební jednotka terénu. Jeho hodnota reprezentuje míru, do jaké je daný voxel zaplněn.*

▶ **Definice 1.2** (Blok)**.** *Blok je 3D model krychle, který odpovídá jednomu plně zaplněnému voxelu.*

Chceme-li využít voxely pro modelování herního terénu, musíme být schopni rozlišit mezi prázdnými a zaplněnými částmi terénu. Nejjednodušší způsob, jak toho dosáhnout, je reprezentovat voxel pomocí boolean hodnoty, kde:

- true: voxel je zaplněn,
- false: voxel je prázdný.

Tento způsob je velmi efektivní [1] a jednoduchý, ale pro generování terénu, jak se dozvíme níže, nedostačující. Flexibilnější alternativou je voxel definovat jako desetinné číslo, kde:

- ≤ 0: voxel je zaplněn,
- > 0: voxel je prázdný.

[1] V paměti zabere pouze 1 bit.

V tomto případě hodnota voxelu představuje vzdálenost od povrchu. I když je tato metoda méně efektivní z hlediska paměťové náročnosti a může být méně intuitivní, nabízí více možností pro různé způsoby manipulace s terénem a jeho kombinování.

1.1.1.1 Paměťová reprezentace

Jeden samostatný voxel není příliš užitečný. Pro vytvoření rozsáhlého terénu je třeba tisíce, v případě větších či hustějších terénů dokonce miliony voxelů. Proto je klíčové použít vhodnou datovou strukturu pro ukládání voxelů, aby byla zajištěna efektivní práce s pamětí a rychlé provádění operací. Níže jsou uvedeny některé z možných datových struktur a jejich vlastnosti.

- **3D pole**
  Nejjednodušší metoda. Jedná se o tříúrovňové pole polí. Výhodou je přímé adresování a nativní podpora ve většině programovacích jazyků.
- **Ploché 3D pole**
  Podobné předchozí možnosti, ale všechna pole jsou „zploštěna“ do jednoho delšího. K poli se poté nepřistupuje přímo, ale index se musí spočítat na základě vstupních souřadnic. Výhodou je oproti jednoduchému poli jednodušší správa paměti a díky „cache-locality“[2] potenciálně vyšší výkon.
- **Hashmap**
  3D souřadnici voxelu lze použít jako klíč do hešovací tabulky. Tato struktura je díky nutnosti hashování výrazně pomalejší než přímý přístup do pole, ale za to má neomezenou velikost a neplýtvá pamětí na nevyužitých souřadnicích. Vhodné pro situace, kde bude většina voxelů prázdná.
- **Octree**
  Rovnoměrná stromová struktura. Skládá se z uzlů a listů. Uzel reprezentuje krychli v prostoru, kterou lze rovnoměrně rozložit na 8 menších uzlů. List reprezentuje voxel. Vlastnostmi podobná na Hashmap, ale s lepšími výkonnostními charakteristikami. [1] [2]

U žádné z těchto struktur nezáleží na konkrétním typu voxelu. Lze je také využít i pro 2D terén.

1.1.2 Chunkování

V praxi se často setkáváme s omezeními výpočetního výkonu a dostupné paměti, které nám neumožňují načítat nebo generovat celý herní svět najednou. Takové omezení je ještě výraznější, pokud pracujeme s potenciálně nekonečnými světy, kde by bylo nemožné zpracovat celý terén najednou. To by znamenalo výrazné zdržení při načítání nebo generování a následně by velká část terénu nevyužívaně zbytečně zadržovala systémovou paměť.

Proto je pro voxelové terény důležitý koncept „chunkování“, který zahrnuje rozdělení světa na menší části, tzv. chunky. Chunky jsou jednotlivě generovány a načítány, což umožňuje zpracovávat pouze tu část terénu, která je v bezprostředním okolí hráče nebo na které se aktuálně zaměřuje. Chunkování přináší několik výhod:

- **Zlepšuje efektivitu**
  Tím, že se generuje a zpracovává pouze relevantní část terénu, je možné udržet výkon na přijatelné úrovni, což je obzvláště důležité v případě real-time aplikací jako hry.
- **Šetří paměť**
  Zpracování menších částí terénu znamená, že systémová paměť je lépe využita a zbytečně senezatěžuje načítáním nevyužívaných částí terénu.

[2] Prvky jsou v paměti adresově blíže k sobě.

- **Usnadňuje paralelizaci**
  Chunky lze generovat a načítat paralelně, což dále zvyšuje výkon.
- **Umožňuje nekonečné světy**
  Díky chunkování je možné generovat nekonečné světy, protože vždy může být generována další část terénu, aniž by bylo nutné načítat celý svět najednou.
- **Flexibilita**
  Chunky mohou být generovány a načítány podle potřeby, což zvyšuje flexibilitu v průběhu hry, např. pokud se hráč pohybuje do dosud nezmapovaných oblastí nebo pokud je třeba změnit část terénu.

Chunkování ovšem vyžaduje sofistikovanější systém konktroly generování a načítaní světa. Systém musí zohlednit plynulou návaznost chunků mezi sebou a potenciální závislosti, obzláště s ohledem na struktury, které mohou potenciálně překrývat více chunků.

1.1.3 Procedurální generování

Systémy procedurálního generování mají za úkol na základě předem definovaných pravidel generovat zpravidla pseudonáhodné data. Umožnují tak automatizovat proces tvorby obsahu a tím jej vytvořit takřka neomezené množství. Tato práce se zaměřuje na generování voxelového terénu, ale podobné systémy existují pro celou řadu jiných typů dat.

Existující systémy procedurálního generování voxelového terénu lze obecně rozdělit do dvou hlavních kategorií podle jejich metodiky:

- **Top-down**
  Svět se generuje celý najednou, což omezuje jeho maximální velikost. Za to ale dává vývojáři plnou kontrolu nad každým aspektem světa. Tento přístup je tedy nejvhodnější pro komplikované uzavřené struktury. Generovaná struktura je často reprezentována jako abstraktní graf a na samotné voxely se převádí jako samostatný krok. [3] [4]
- **Bottom-up**
  Svět se generuje chunk po chunku. Každý chunk je zpracováván samostatně, struktury se do terénu vkládají až po sléze. Bottom-up přístup se vyznačuje možností efektivní paralelizace a nekonečného terénu za cenu nižší kontroly a složitejší implementace. Často se využívá v kombinaci s top-down přístupem – do hrubého terénu vkládají struktury, které by byly vygenerovány zvlášť. [5]

1.1.4 Šumové funkce

Základem naprosté většiny procedurálních generátorů jsou tzv. generátory šumu. Jsou to funkce $f(x_1, ..., x_n)$, jejichž výstupem jsou pseudonáhodné data. Klíčovou vlastností těchto funkcí je jejich determinismus a žádný vnitřní stav – pro konkrétní vstup vrátí vždy stejnou hodnotu. To v praxi umožnuje systémy využívající tyto funkce jednoduše paralelizovat. [5]

Jejich náhodnost spočívá ve speciální hodnotě anglicky označované „seed“. Jak tato hodnota funguje je závislé na konkrétním algoritmu a implementaci, ale obecně si ji lze představit jako jistou počáteční hodnotu či offset – vlastnost determinismu platí pouze pokud je hodnota seed nezměněna.

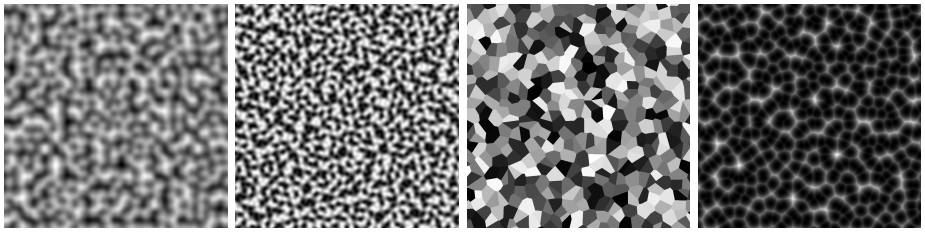
1.1.4.1 Gradientní šum

Jedná se se o speciální případy šumových funkcí, které na výstupu vracejí hodnoty s plynulými přechody, tzv. gradienty. Tato vlastnost je činí vhodnými právě pro generování terénu, od kterého očekáváme plynulost přechodů. [6]

1.1.4.2 Voronoi / Worley

Voronoi / Worleyho (dále VW) šum pro určitou vstupní souřadnici a množinu bodů vrací vzdálenost k nejbližšímu z nich, případně jeho pozici. Tento algoritmus ve své základní podobě není šumovou funkcí – musíme totiž mít předdefinovanou množinu bodů, která nikdy nemůže být nekonečná z důvodu pamětové náročnosti.

Vhodnou úpravou ho ovšem lze modifikovat tak, že se jako šumová funkce chová. Úprava spočívá v zadefinování množiny bodů jako všech okolních bodů vstupní souřadnice na pevně dané mřížce. Body mřížky se pak posunou o offset určený hashem jejich pozice. Takto upravený algoritmus ztrácí část své náhodnosti, ale pro potřeby generování terénu to nijak nevadí.



Obrázek 1.1 Vizualizace šumu [7] (popořadě: Perlin, OpenSimplex, Voronoi (hodnota buněk), Voronoi (vzdálenosti))

1.1.5 Reprezentace modelů

Samotné šumové funkce jsou velmi flexibilní nástroj, ale hodí se pouze pro generování hrubého terénu. Pro generování sofistikovanějších struktur je vhodné mít knihovnu předdefinovaných modelů, které se do terénu postupně vkládají.

1.1.5.1 Tradiční 3D model

Povrchový model tvořen polygony. V této podobě ho nelze do voxelového terénu vložit. Je tedy za potřebí ho nejprve převést do voxelové podoby.

▶ **Definice 1.3** (Voxelizace)**.** *Proces převedení polygonového 3D modelu na voxelový.*

Voxelizaci lze obecně provést dvěma způsoby:

- **Kolizně**
  Spočítáme bounding box modelu[3]. Vybereme jednu jeho stranu a rozdělíme ji mřižku odpovídající rozlišení našeho terénu. V každém bodu rastru „vystřelíme“ paprsek směrem do modelu a na pozicích procházejících povrchem modelu zapíšeme voxel jako plný. Seřazením bodů lze u plně uzavřených modelů také vyplnit vnitřní voxely. [8]

- **Povrchově**
  Pro každý trojúhelník modelu vyplníme všechny voxely, kterými prochází. Na rozdíl od paprskového přístupu se každý trojúhelník zpracuje právě jednou, takže je výrazně rychlejší. Nedokáže ovšem vyplnit vnitřní voxely a je méně přesný. [8]

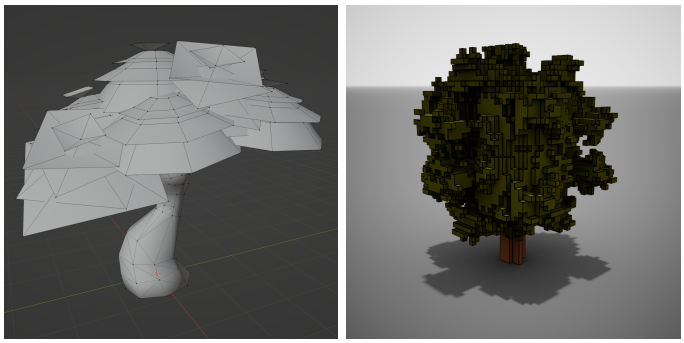
Velkou výhodou využití polygonových modelů ve voxelovém terénu i přes nutnost voxelizace je možnost model libovolně transformovat[4] před samotnou voxelizací, a to bez ztráty detailu nebo dír v modelu.

[3] Nejmenší možná krychle taková, že obsahuje všechny body modelu.
[4] Pozice, rotace a velikost na všech osách

1.1.5.2 Voxelový model

Je také možné model reprezentovat přímo na voxelové mřížce. Umělec tak má absolutní kontrolu nad pozicí bloků a může tak předejít problémům vznikým nepřesnostmi při voxelizaci. Není nutné ho voxelizovat, takže je jeho vkládání do světa rychlejší.

Takovýto model ovšem nelze volně transformovat bez ztráty detailu či vzniku artefaktů. Model totiž neposkytuje dostatek informací k zaplnění děr vznikých rotací nebo zvětšováním.



Obrázek 1.2 Ukázka reprezentace modelů (popořadě: polygonový strom, voxelový strom)

1.2 Technická

1.2.1 Herní engine

Prototyp bude vytvořen s pomocí herního enginu. Tento engine musí podporovat 3D vykreslování a být dostatečně rychlý. Editor není pro tvorbu prototypu potřeba, ale vývoj zjednoduší. Kritérem pro výběr bude také jeho složitost, otevřenost licence a dostupnost zdrojového kódu. V této sekci se nachází krátké shrnutí nejpopulárnějších herních enginů.

1.2.1.1 Unity 3D

V současné době je Unity 3D nejpopulárnějším herním enginem pro individuály, ale i malé až střední studia. Své pozice jakožto standard docílil podporou všech moderních platform a intuivním editorem. Díky své popularitě má velkou komunitu, která vytváří různé knihovny a nástroje. Je také velmi dobře dokumentován. [9]

Základní verze je zdarma, ale od určitého obratu je nutné platit licenci. Nemá veřejně dostupný zdrojový kód. Jako svůj hlavní jazyk používá C#, ale používá starý a pomalý Mono runtime. [5]

1.2.1.2 Unreal Engine

Unreal Engine je herní engine vyvinutý společností Epic Games. Je známý svým vysokým vizuálním standardem a obecně vynikajícími výsledky v oblasti renderingu. Má také mnoho funkcí pro tvorbu multiplayerových her a podporu pro VR. Jako jazyk používá C++ nebo vizuální jazyk Blueprint. Díky své efektivitě si našel své místo v převážně u velkých projektů.

Unreal Engine je open source a jeho zdrojový kód je dostupný na GitHubu, ale podobně jako Unity není zdarma – od určitého obratu je nutné platit licenci. Jeho editor je velmi složitý a náročný na systém. [10]

[5] V moderním ekosystému .NET byl nahrazen CoreCLR

1.2.1.3 Godot Engine

Nenáročný FOSS[6] projekt. Komunita je velmi aktivní a vývoj je velmi rychlý, ale velikostně zatím stále menší než u Unity nebo Unreal Engine. Podporuje C# a C++, na rozdíl od Unity však používá moderní .NET 7 runtime. Editor je jednoduchý a rychlý, ale není stejně flexibilní jako Unity nebo Unreal Engine. [11]

Tabulka 1.1 Srovnání herních enginů

Engine	Podporovaný jazyk	Open source	Zdarma
Unity	C#	Ne	Ne
Unreal	C++	Ano	Ne
Godot Engine	C# / C++	Ano	Ano

1.2.1.4 Výběr herního enginu

S ohledem na výše uvedené kritéria a faktory byl pro vývoj tohoto prototypu vybrán Godot Engine. Nabízí nejlepší kombinaci otevřenosti, dostupnosti zdrojového kódu a jednoduchosti vývojových nástrojů vhodných pro tento projekt.

Unity 3D je velmi oblíbený díky své intuitivní editaci a široké podpoře platforem, ale trpí omezeným přístupem ke zdrojovému kódu a starším Mono runtime. Unreal Engine nabízí skvělou grafickou kvalitu a funkce pro multiplayerové hry, avšak jeho editor je složitý a náročný na systém.

1.2.2 Knihovny

1.2.2.1 Generování šumu

- **FastNoise Lite [12]**
  Univerzální přenosná jednosouborová knihovna, které je implementována ve většině populárních jazyků. Podporuje Perlin, Simplex, Voronoi a value šum. Zdaleka nejpopulárnější volba díky své jednoduchosti. Za svou jednoduchost ale platí tím, že je vcelku pomalá.

- **FastNoise 2 [13]**
  C++ knihovna nahrazující původní FastNoise. Vysoce optimalizovaná pomocí SIMD[7], výrazně rychlejší něž FastNoise Lite. Pro využití z prostředí C# lze použít wrapper FastNoiseOO [14], který je objektově orientovanou nadstavbou nad oficiálními bindingy [15]. Ke knihovně je dostupná utilita NoiseTool [7], která umožňuje konstruovat graf šumového generátoru skrze GUI a následně jej vizualizovat.

1.2.2.2 Správa voxelových modelů

Na rozdíl od tradičních 3D modelů nejsou voxelové modely přímo podporovány ve většině herních enginů, je proto nutné je načítat manuálně z kódu. Pro voxelové modely neexistuje ani standardizovaný formát – každý editor má svůj vlastní proprietární a většinou closed-source[8] formát.

Pro jeden z nejpopulárnějších voxelových editorů MagicaVoxel[16] ovšem existuje alespoň veřejně dostupná dokumentace [17], podle které je implementována řada načítacích knihoven. Výběr mezi nimi je vcelku omezený, protože většina z nich je nekompletní a nebo obsahuje chyby. Konkrétně pro C# lze využít knihovna VoxReader [18]. Implementuje všechny zdokumentované operace. Nedokáže do modelu zapisovat, ale to pro potřeby tohoto projektu není nutné.

[6] Free Open Source Software
[7] Vektorové instrukce (SSE, AVX2, ...)
[8] Neexistuje veřejně dostupná knihovna nebo dokumentace pro daný formát

1.2.2.3 Vývojové rozhraní

Při vývoji prototypu bude zcela určitě potřeba interakce s generátorem a vizualizace jeho částí. K takovým debugovacím a vizualizačním účelům je určena populární knihova Dear ImGui [19], která implementuje immediate-mode[9] uživatelské rozhraní. Knihovna je psána v C++, ale pro volání z C# existuje několik wrapperů. Konkrétně pro Godot Engine je dostupná integrace imgui-godot [20], která vnitřně používá .NET knihovnu ImGui.NET [21].

1.2.2.4 Komunikace mezi vlákny a sledování změn

Pro efektivní paralelizaci generátoru je potřeba zajistit, aby vlákna mohla komunikovat mezi sebou a efektivně reagovat na změny stavu. Jednou z vhodných knihoven pro tento účel je Reactive Extensions (Rx) [22]. Tato knihovna poskytuje robustní způsob pro definování toků dat, které lze dále zpracovávat pomocí různých operátorů. Díky tomuto přístupu je možné snadno sledovat změny ve stavu generátoru terénu a reagovat na ně v závislosti na potřebě.

Pro pohodlnější práci s Reactive Extensions lze využít nadstavbu ReactiveUI [23]. ReactiveUI je framework primárně určený pro vývoj GUI[10] aplikací, ale jeho části lze napříč jménu použít i pro jiné účely.

Výhody použití Reactive Extensions a ReactiveUI v kontextu generátoru terénu zahrnují:

- Snadné navázání na změny stavu generátoru a rychlá reakce na ně.
- Jednoduchá manipulace s daty v rámci toku pomocí funkcionálních operátorů.
- Možnost propojení různých toků dat a reakcí na změny ve více částech generátoru současně.
- Efektivní zpracování asynchronních operací a možnost přizpůsobení zátěže jednotlivých vláken.
- Zvýšená modularita a snadnější údržba kódu díky oddělení jednotlivých částí generátoru.
- Snadnější tvorba testovatelného kódu díky vysokému oddělení mezi logikou a prezentační vrstvou.

1.2.3 Zdroje modelů

Jelikož cílem práce není manuálně vytvářet modely, ale pouze je v rámci generátoru využívat, je nutné pro ně najít vhodný zdroj. Zdrojů licenčně otevřených modelů zdarma existuje mnoho, ale pro potřeby prototypu byly vybrány následující:

- opengameart.org – Databáze volně dostupných 3D modelů, textur, animací a dalších prostředků pro vývojáře. Velký výběr modelů, ale mnoho z nich nedosahuje vysoké kvality.
- itch.io – Platforma pro nezávislé tvůrce her, která poskytuje mnoho bezplatných i placených zdrojů, včetně 3D modelů, textur, zvuků a skriptů.
- kenney.nl – Stránka tvůrce her a uměleckých zdrojů známého jako Kenney. Jeho zdroje jsou známé pro svou kvalitu a jednotný styl.

[9] Aktualizuje se každý snímek
[10] Graphical user interface

Kapitola 2
Návrh

V druhé kapitole jsou definovány funkční požadavky prototypu a návrh samotné implementace. Požadavky popisují očekávanou funkcionalitu a výstup, který bude generátor vytvářet. Návrh pak popisuje hlavní myšlenku procesu generování, potřebné a abstraktní architekturu generátoru.

2.1 Požadavky

2.1.1 Funkční požadavky

FP1 Generování 2.5D voxelového terénu
V kontextu této práce se pod 2.5D terénem rozumí 3D terén s omezenou hloubkou. Voxelový terén bude složen z bloků. (definice 1.2)

FP2 Neomezená velikost terénu
Terén se bude dynamicky generovat kolem pozice kamery bez nutnosti předgenerování celého terénu.

FP3 Dosazování předdefinovaných modelů do terénu
Systém by měl být schopen definovat modely polygonové i voxelové modely a pravidla pro jejich dosazování do terénu. Součást NFP3.

FP4 Sémantické rozdělení terénu na vrstvy
Konečný terén bude kombinací několika samostatných vrstev. Vrstvy budou zobrazitelné nezávisle na ostatních.

FP5 Uživatelské rozhraní pro interakci s a vizualizaci stavu generátoru
Bude složit k debugování, testování a vizualizaci stavu generátoru včetně vrstev.

FP6 Vlákování
Generování terénu bude probíhat průběžně na pozadí a to tak, aby byl plně využit potenciál moderních vícejádrových procesorů. Součást NFP1.

FP7 Podpora Windows Aplikace bude navržena a implementována tak, aby byla kompatibilní s operačním systémem Windows.

2.1.2 Nefunkční požadavky

NFP1 Přijatelná rychlost generování
Generátor by měl být schopen generovat terén dostatečně rychle pro zaplnění viditelné oblasti za předpokladu rozumné vzdálenosti kamery hráče.

NFP2 Přijatelná paměťová náročnost
Terén nesmí zabírat příliš velké množství paměti s ohledem na velikost prostoru, který je momentálně zobrazen.

NFP3 Zajímavost generovaného terénu
Z perspektivy potenciálního hráče by terén neměl být příliš jednoduchý a homogenní.

2.2 Definice voxelů

V předchozí části o voxelovém terénu 1.1 bylo nastíněno, že voxely mohou být reprezentovány různými způsoby. Pro účely tohoto projektu je nezbytné zvolit vhodný způsob reprezentace voxelů, který bude optimální pro konkrétní aplikaci.

Každý voxel v terénu reprezentován 16 bitovou strukturou, skládající se ze dvou 8 bitových čísel:

- **Typ bloku**
  První 8bitové číslo určuje typ bloku v daném voxelu. Nulová hodnota značí prázdný voxel, zatímco ostatní hodnoty představují unikátní identifikátory přiřazené jednotlivým typům. Tyto identifikátory budou použity pro vizualizaci voxelů a získání dalších informací o bloku z předem definované tabulky.
- **Jas**
  Druhé 8bitové číslo reprezentuje jas, který ovlivňuje výslednou barvu bloku. Slouží pro odlišení bloků stejného typu, které by jinak byly uniformní.

2.3 Proces generování

Pro splnění FP2 bude nutné využít chunkování. (kapitola 1.1.2) Při návrhu genrátoru s využitím chunků je třeba dbát na to, aby nebyly porušeny závislosti mezi okolními chunky. Při provánění některých operací je potřeba znát informace o voxelech v okolí, které ještě nebyly vygenerovány.

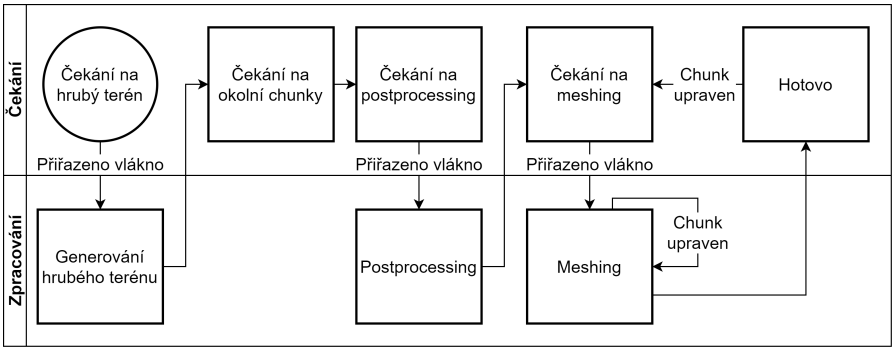
Dále je třeba zohlednit možnost paralelizace generování. Vzhledem k počtu chunků nedává smysl pro každý chunk vytvářet vlastní vlákno. Je tedy potřeba práci rozdělit tak, aby ji bylo možné provádět na omezeném počtu vláken. Na základě analýzy a požadavků by mohl vypadat proces generování terénu zjednodušeně následovně:

1. **Vyvoření chunku**
   Pokud je kamera dostatečně blízko k potenciální pozici chunku, který ještě nebyl vytvořen, je na této pozici vytvořen nový chunk.
2. **Generování hrubého terénu**
   V tomto kroku se generuje hrubý terén pro konkrétní chunk, který je základem pro další kroky. Krok musí být naprosto nezávislý na okolních chunkcích, protože ty ještě nemusí být vygenerovány. (kapitola 2.4)
3. **Čekání na okolí**
   Po dokončení kroku 2 generátor počká, než se vytvoří okolní chunky a dostanou se také do kroku 3. Tímto se splní závistost generovaného terénu na okolních chunkcích v následujících krocích.

4 **Postprocessing**
Postprocessing je zodpovědný za vytvoření finálního terénu. V tomto kroku se provádí všechny operace, které jsou závislé na okolních chunkcích. Hlavní takovou operací v tomto prototypu je dosazování modelů do terénu, ale např. odhad normálu bloku je také závislý na okolí.

4 **Meshing**
Po dokončení generování voxelů je potřeba je převést do polygonové podoby k zobrazení. Tento krok se bude opakovat také v případě, že je chunk modifikován po dokončení generování.

V každém kroku je chunku přiřazeno vlákno. Toto vlákno zavolá metodu na chunku, která se pokusí pokračovat v generování na základě aktuálního stavu. V případě, že už na chunku nelze provést další krok, (např. čekání na okolí) se metoda ukončí a vlákno bude přiřazeno jinému chunku. Celý tento proces je znázorněn na obrázku 2.1.



Obrázek 2.1 Proces generování chunku

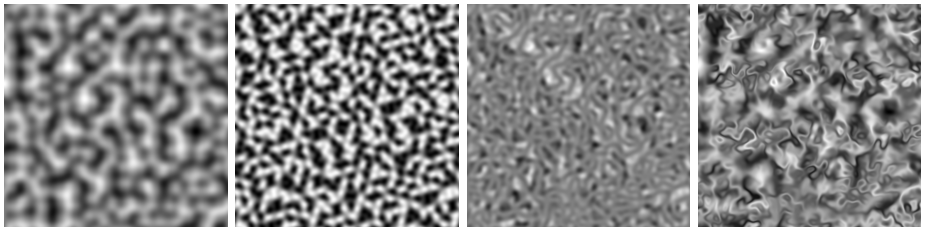
2.4 Hrubý terén

Generování hrubého terénu bude implementováno převážně pomocí šumových funkcí. (kapitola 1.1.4) Dále je také možné využít vlastní čisté funkce[1], které ale musí zachovat vlastnost nezávislosti na okolí.

Samotné funkce by negenerovaly moc zajímavý terén, a tím by nebyl splněn NFP3. Tyto dva typy funkcí lze ovšem kombinovat pomocí tradičních operací jako je sčítání a násobení, ale také použitím jedné funkce jako vstup pro druhou. Takovým spojením několika šumových funkcí dohromady lze dosáhnout mnohem zajímavějších výsledků. (obrázek 2.2)

Výsledek pak lze převést na terén reinterpretováním výsledných hodnot jako voxelů. (kapitola 1.1.1) Zde je také důležité voxelům přiřadit konkrétní typ bloku. Ten lze určit buď fixně pro danou vrstvu, pomocí poziční funkce, nebo na základě hodnoty datové vrstvy. (kapitola 2.6)

[1] Anglicky: „pure function“ – funkce bez vedlejších efektů a vnitřního stavu



Obrázek 2.2 Kombinování šumů [7] (popořadě: Perlin(x), Simplex(x), Perlin(x) * Simplex(x), Perlin(Simplex(x)))

2.5 Vkládání modelů

Možností vkládání modelů do terénu je mnoho. V tomto prototypu se bude používat ten nejjednodušší z nich – pro každou souřadnici na povrchu hrubého terénu chunku se zkontroluje, jestli jsou zde splněny podmínky pro vložení modelu, a pokud ano, tak ho generátor vloží. Pro kontrolu minimální vzdálenosti mezi modely se tato pozice také uloží do seznamu využitých pozic. Samotné 3D modely, ať už polygonové nebo voxelové, neobsahují pro tento účel dostatek informací. V rámci herního engine tedy bude potřeba definovat metadata, která budou určovat, jak přesně s nimi bude generátor zacházet.

- **Pivot**
  Pro umisťování a rotaci modelů je potřeba znát bod, kolem kterého bude transformace prováděna – tzv. „pivot“. Ten se ovšem nemusí vždy nacházet v geometrickém středu modelu. Bude tedy nutné jej definovat zvlášť.

- **Prostorové omezení**
  Vkládání modelů do terénu nemůže probíhat čistě náhodně, protože by se tak vytvořil nepřehledný a nesmyslný terén. Modely by kolidovaly mezi sebou, nebo by byly umístěny v nevhodných místech. Proto je nutné vytvořit jistá omezení, které těmto problémům zabrání. Konkrétně by šlo o definice „omezujících zón“, které by vyznačovaly části modelu a jeho okolí, které pro vložení musí být buď prázdné, nebo naopak zaplněné.

- **Typ bloku**
  Modely ve své základní podobě neobsahují informace o tom, jaký typ bloku mají být v terénu po provedení voxelizace. Některé modely se navíc mohou skládat z více typů bloků najednou. Pokud je model homogenní, tedy obsahuje pouze jeden typ bloků, stačí pouze definovat výchozí typ na úrovni modelu. Pokud je ovšem model složený z více typů bloků, lze využít podobného principu jako u definice omezení. Každý typ bloku různý od toho výchozího by měl mít definovanou svoji „přepisující zónu“, která by pro všechny bloky modelu v této zóně přepsala jejich typ na ten definovaný v zóně.

Tyto informace stále pro vložení modelu do terénu nestačí. Je potřeba ještě definovat, za jakých podmínek se má model vůbec vložit do terénu. To záleží mimo jiné na:

- typu bloku, na kterém se model může nacházet,
- povolené rotace modelu,
- vzdálenosti od ostatních modelů.

Tyto vlastnosti už ale není nutné definovat na úrovni modelu, ale na úrovni vrstvy. Mohli bychom se totiž setkat se situací, kdy bychom chtěli jeden model využít ve více vrstvách, ale s různými vlastnostmi.

2.5.1 Definice metadat v rámci Godot Engine

Metadata by bylo možné k modelu přidat přímo v kódu, ale to by bylo velmi nepřehledné a neefektivní. Proto je vhodnější definovat je v rámci editoru Godot Engine, kde budou viditelné přímo v 3D prostoru. Godot Engine využívá jako základní jednotku světa tzv. uzly, které vždy odpovídají právě jedné instanci objektu. Kombinací několika uzlů lze vytvořit hierarchii objektů, která se nazývá scénou. Tyto scény lze buď vytvářet přímo z kódu nebo využít editoru, který je umožňuje vytvářet pomocí grafického rozhraní a následně uložit do souboru. Tohoto systému můžeme využít i pro definici metadat modelů.

Každý model bude mít definovanou vlastní scénu, která bude obsahovat všechny potřebné informace. Zóny mohou být definovány jako 3D modely daného tvaru, které budou připojeny k modelu jako potomci. Tito potomci budou mít přiděleny speciální editorové skripty, které budou zajišťovat správné fungování zón.

Pivot bude implicitně definován jako počáteční bod (0, 0, 0) scény. Pro určení středu modelu pouze posuneme uzel modelu tak, aby jeho střed odpovídal počátečnímu bodu. Ostatní informace lze pak z editoru nastavit pomocí tzv. „Export“ attributů ve skriptu modelu či scény. Tyto atributy dokůžou zpřístupnit veřejné proměnné skriptu tak, že je bude možné nastavit přímo v editoru a následě spolu se scénou uložit do souboru.

2.6 Vrstvy

Vstvy budou samostatné části generétoru, které definují logiku generování dat[2] a hrají tedy klíčovou roli ve výsledné podobě generovaného terénu. Použitím vrstev je možné rozdělit složitý terén do menších a jednodušších částí, které bude možné generovat a ladit samostatně.

V závislosti na obsahu a využití vrstev se budou dělit na dva typy. Z pohledu generátoru se budou lišit pouze typem implementovaného rozhraní, samotná logika generování zůstane stejná pro všechny vrstvy.

- **Voxelová vrstva**
  Základní vrstva, která přímo generuje voxelový terén. Hlavní dělení těchto vrstev bude na základě tzv. biomů. Biomy představují typy terénu, které mají jasně definované vlastnosti.
  Příklady takových vrstev by mohly být: les, poušť, jeskyně, …
  Tento typ také bude definovat množinu modelů, které se v nich mohou nacházet. Pro správné vkládání modelů zde zároveň budou dodefinována metadata, které nejsou definovány přímo ve scéně modelu. (kapitola 2.5)

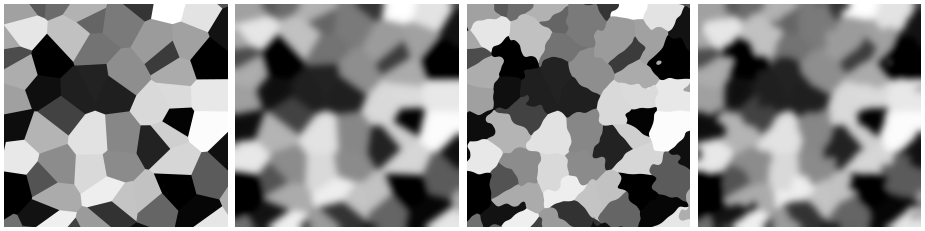
- **Datová vrstva**
  Speciální, abstraktnější typ vrstvy, která generuje pouze pomocná data, která se poté použijí pro generování voxelové vrstvy.
  Příklady takových vrstev by mohly být: výška povrchu, vlhkost, rozdělení biomů, …
  Tento typ vrstvy nebude implementovat rozhraní pro generování voxelů, ale pouze pro generování vizualizačních barev. Bloky této vrstvy tedy nebudou mít přiřazeny typ bloku, ale pouze barvu.

2.6.1 Kombinační vrstva

Rozdělení terénu do více vrstev je užitečná abstrakce pro vývoj a debugování. Pro koncové využití generátoru jako komponenty hry je ale potřeba, aby byl výstup generátoru jedna souvislá vrstva. Bude tedy za potřebí vrstvy navzájem kombinovat a to tak, aby byl splněn NFP3. (kapitola 2.1.2) K segmentaci terénu byde využit výše zmíněný Voronoi šum (kapitola 1.1.4), kde bude každé buňce přiřazen jeden biom.

[2] Ne nutně voxelů

Jak jde vidět na obrázku 2.3, takové rozdělení by ovšem vyústilo ve velmi ostré přechody mezi biomy a působilo by tak nepřirozeně. Přechody mezi biomy lze zjemnit pomocí jednoduché lineární interpolace, která bude využívat vzdálenost od hranice mezi biomy. Takto interpolovaný terén stále působí nepřirozeně, protože hrany mezi biomy jsou příliš přímé. Tento problém lze vyřešit pomocí přimíchání vrstvy 2D šumu ke každé souřadnici v terénu, která ji mírně posune. Postup se podobá kombinování šumových funkcí popsané v kapitole 2.4.



Obrázek 2.3 Vizualizace rozdělení biomů

2.7 Vývojové rozhraní

Pro efektivní vývoj a ladění terénu a modelů je nezbytné vytvořit vývojové uživatelské rozhraní, které umožní interakci s generátorem terénu a vizualizaci jeho jednotlivých částí. Toto rozhraní by mělo poskytovat následující funkce a možnosti:

- **Výpis základních informací o stavu generátoru**
  Zobrazování informací o aktuálním stavu generátoru terénu včetně výkonnostních metrik, které by mohly pomoci identifikovat potenciální problémy s rychlostí generátoru nebo regrese.
- **Výběr vrstvy k zobrazení**
  Možnost výběru konkrétní vrstvy terénu, která se má zobrazit. Díky této klíčové funkci se vývojář dokáže zaměřit na jednotlivé části terénu nebo naopak lépe porozumět, jak vrstvy vzájemně spolupracují.
- **Vizualizace definovaných modelů**
  Zobrazení modelů, které jsou definovány pro danou vrstvu terénu, umožní rychleji a lépe porozumět struktuře modelů a jejich chování v různých situacích. Rozhraní by mělo umožňovat mimo samotný výběr modelu nastavení 3D transformace a vizualizaci zón modelu (kapitola 2.5), a to včetně jejich voxelizované formy.
- **Manuální vkládání modelů do terénu**
  Rozhraní by také mělo nabídnout vývojáři možnost ručně vkládat modely do terénu. Tato funkce poskytne vývojářům schopnost otestovat chování modelů přímo při dosazování do voxelového světa na pozici určenou kurzorem. Před vložením modelu rozhraní zobrazí výpis všech omezujících zón (kapitola 2.5) a jestli současná pozice ve voxelovém světě splňuje podmínky pro vložení. Pro účely testování by ovšem vývojář měl mít možnost vložit model i přesto, že nejsou všechny splněny.

Funkcionalita bude rozdělena do dvou samostatných pohledů a implementována pomocí ImGui. (kapitola 1.2.2.3) Hrubý návrh vývojového rozhraní je zobrazen na obrázcích 2.4.

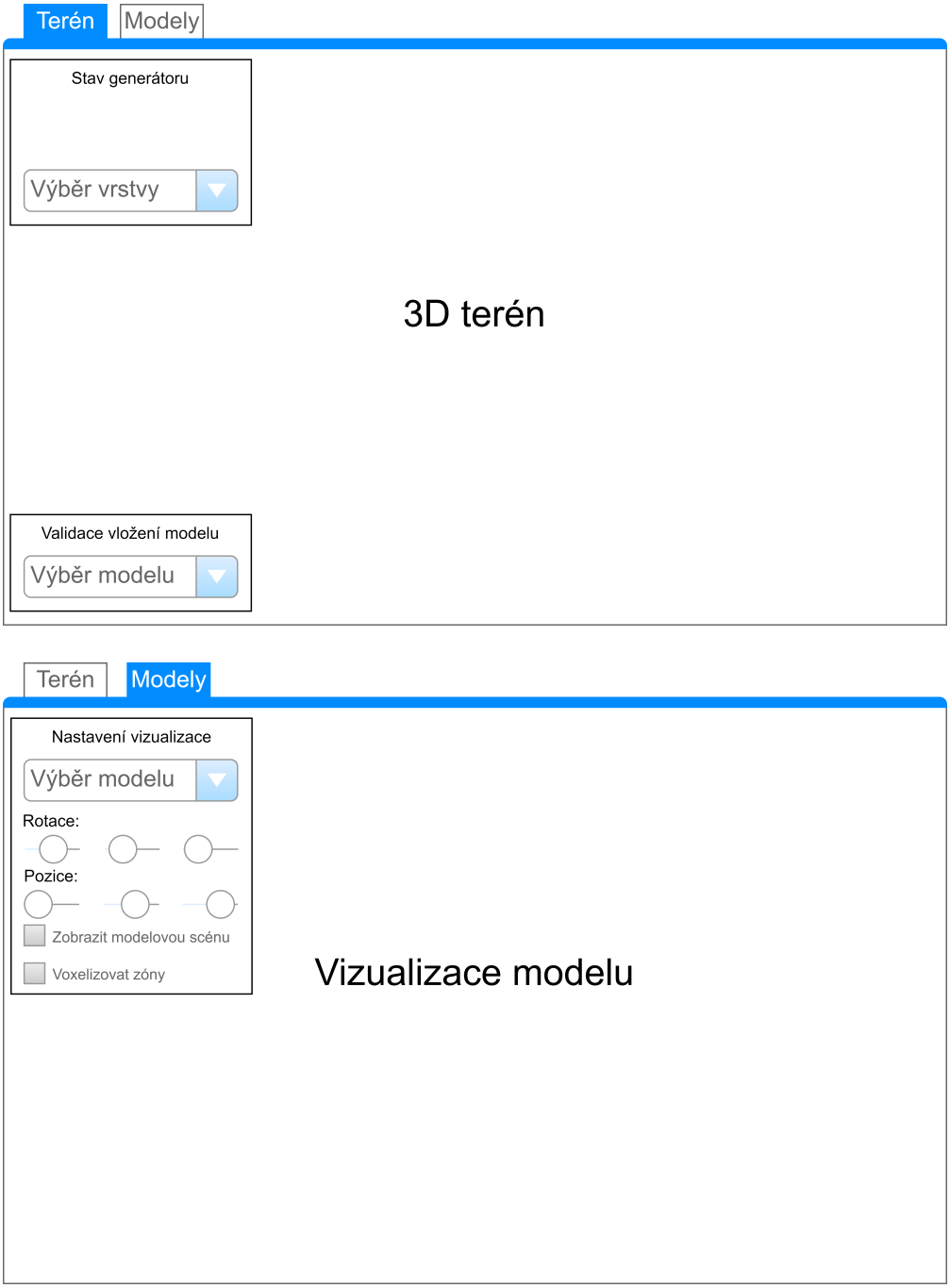
2.8 Recyklování prostředků

V průběhu generování terénu bude zcela určitě docházet k vytváření dočasných pomocných datových struktur. Pokud by se tyto struktury vytvářely a rušily příliš často, mohlo by dojít k vý-

konostním problémům z důvodu přetížení GC[3]. Proto je potřeba zajistit, aby se tyto struktury vytvářely a rušily co nejméně často.

Jedním ze způsobů, jak toho dosáhnout, je použití tzv. „pools“. Pools jsou ve své podstatě kolekce předalokovaných objektů. Pokud je potřeba vytvořit nový objekt, tak se pouze vybere jeden z těch v poolu a po použití se vrátí zpět. Tím se zamezí přetížení GC, ale je potřeba zajistit, že jsou objekty po použití opravdu vráceny do poolu. Zároveň je třeba brát ohled na to, že takto vypůjčený objekt již mohl být někdy předtím použit – nebude tedy ve výchozím stavu.

[3]Garbage collector



Obrázek 2.4 Hrubý návrh vývojového rozhraní

Kapitola 3

Implementace a optimalizace

3.1 Jednotky

3.1.1 Prostorové

Stjně jako v jiných enginech, v Godot Engine je základním prostorovým typem `Vector3` pro 3D a `Vector2` pro 2D. Jedná se o n-tice tří reálných čísel, které reprezentují souřadnice v prostoru. Voxelový terén ovšem nepracuje s reálnými souřadnicemi, ale s celočíselnými. Pro podobné účely je v Godot Engine k dispozici typ `Vector3i` a `Vector2i`, které jsou v podstatě obdobou `Vector3` a `Vector2`, ale s celočíselnými 32 bitovými souřadnicemi.

Zpočátku byly pro implementaci využívány právě tyto typy. V průběhu vývoje se ale ukázalo, že práce s nimi je v kontextu generátoru velmi náchylná na chyby. V procesu generování se totiž často mění prostor, ve kterém se souřadnice nacházejí, což není u těchto typů žádným způsobem rozlišeno. Konkrétním příkladem by byl přechod z globálních, světových souřadnic na souřadnice relativní vůči pozici chunku.

Z tohoto důvodu byly vytvořeny nové vlastní generické typy `Pos3<T>` a `Pos2<T>`. Implementací jsou identické těm integrovaným v Godot Engine, ale navíc v podobě generického parametru obsahují informaci o tom, v jakém prostoru se souřadnice nacházejí. Parametr není nikde v kódu využit ani kontrolován, je čistě informativního charakteru.

Parametr může nabývat následujících hodnot:

- **Global** Globální souřadnice ve světě.
- **Local** Souřadnice relativní vůči pozici nějakého chunku.
- **GlobalSnapped**
  Stejné jako **Global**, ale souřadnice jsou zaokrouhleny na nejbližší násobek velikosti chunku v blocích. Používá se převážně jako pozice chunku – je aplikovatený pouze na `Pos2<T>`.

3.1.2 Směrové

Analogicky prostorovým jednotkám, herní enginy popisují směry pomocí vektorů, což je pro voxelový terén často nevhodné. Pro potřeby generátoru byly vytvořeny vlastní typy `Direction2D`, `Direction3D` a `Direction3DExt`. Mimo popisu směrů zároveň poskytují i pomocné funkce pro rotaci a převod na vektor.

- **Direction2D** 4 směry ve 2D prostoru.

- **Direction3DExt** 8 směrů ve 2D prostoru.
- **Direction3D** 6 směrů ve 3D prostoru.

3.2 Svět

Nejdůležitější součástí generátoru tvoří singleton třída `VoxelWorld`. Má na starosti správu globálního stavu světa – ukládání a přístup k chunkům, vytváření nových chunků, správu a přiřazování worker[1] vláken a v neposlední řadě také poskytuje řadu pomocných funkcí využívaných ostatními částmi generátoru.

3.2.1 Ukládání a přístup k chunkům

Chunky se ukládají do hašovací tabulky, ve které je klíčem jejich pozice ve světě. Přístup k nim je tedy pomalejší, než kdyby byly uloženy v poli, ale tato operace není kritická a výhody hašovací tabulky převažují. Třída také poskytuje metody také zprostředkuje přístup k chunkům a voxelům v globálním kontextu.

3.2.2 Vytváření chunků

Hlavním úkolem generátoru je zaplnit co nejrychleji viditelnou část světa. Generátor tedy potřebuje vědět, které chunky jsou viditelné a které ne. K tomuto účelu slouží metoda `UpdateArea(…)`, která je volána z hlavního vlákna každý snímek aktivní kamerou nebo jinou komponentou, která má k této informaci momentálně přístup.

Nejjednodušší, naivní přístup tvorby chunků je vždy vplnit obsah vstupní oblasti a vytvořit chunky, které v ní ještě neexistují. Tento přístup je však velmi neefektivní – nemá smysl vytvářet chunky, pro které stejně nebude existovat volné worker vlákno. Další potenciální problém přestavuje samotné předávání nových chunků worker vláknům. Zpracování nových chunků může trvat až několik stovek milisekund a mezitím se kamera může posunout do jiné oblasti světa. Pokud by se chunky předávaly pomocí jednoduché fronty, při rychlém pohybu kamery by se fronta zaplnila a generátoru by trvalo dlouho, než by se dostal k aktuálně viditelným chunkům. Je tedy za potřebí v reálném čase dávat prioritu aktuálně viditelným chunkům a to nejlépe podle jejich vzdálenosti od středu kamery. (kapitola 3.2.3)

Standardním řešením v podobných situací je použití prioritní fronty. Ta však v tomto případě není vhodná, protože vzálenost chunku od středu kamery se může v čase měnit a tak by bylo potřeba, aby se chunky v prioritní frontě při každém přesunu kamery znovu seřadily.

Vhodnějším přístupem se ukázalo být použití jednoduché fronty, do které se ale chunky přidávájí až v okamžiku, kdy je alespoň jedno worker vlákno volné. Tím se zamezí přetížení fronty a zároveň i situaci, kdy by se generátor dostal k chunkům, které už nejsou viditelné. Toto řešení přidává do systému zpoždění, protože se chunky přidávají do fronty až v okamžiku, kdy sa zavolá výše zmíněná metoda `UpdateArea(Rect2 area)`. Ta se volá každý snímek – při 60 snímcích za sekundu tedy má tedy v nejhorším případě zpoždění zhruba 16 milisekund. Relativně vůči náročnosti generování je tato ztráta zanedbatelná a snížení latence je důležitější.

3.2.3 Kruhová iterace

Jednoduchá iterace přes viditenou oblast světa by vždy zpracovávala chunky od nejnižších souřadnic po nejvyšší nebo naopak. Pro generování chunků je však důležité, aby byly nejprve zpracovány chunky, které jsou nejblíže středu kamery a tím byla minimalizována šance, že se kamera posune do oblasti, která ještě není vygenerovaná.

[1] Vlákno, které na pozadí zpracovává chunky

K tomuto účelu se využívá kruhová iterace, která prochází chunky v kruzích od středu kamery směrem ven. Tato iterace je implementována pomocnou metodou `Loop.Radial2D(int radius, Func<Vector2i, bool> action)`, která přijímá jako parametr funkci, která se volá pro každý bod v daném kruhu do daného poloměru nebo dokud funkce nevrátí `true`.

3.2.4 Generalizace

Tento systém byl generalizován a používá se nejenom pro vytváření chunků, ale i pro všechny ostatní operace, které vyžadují přiřazení vlákna. Místo reference na chunk se do fronty přidává pouze pozice chunku a to v případě, že chunk buď neexistuje, nebo `Chunk.NeedsWorker == true`. (kapitola 3.3.1) Worker se poté pokusí získat referenci na chunk a pokud se mu to nepodaří, znamená to, že ještě nebyl vytvořen. V takovém případě ho sám vytvoří a přidá do tabulky.

Mimo konzistenci se zbytkem systému má přesunutí vytváření chunků do worker vláken jednu zásadní výhodu – pokud by se vytváření chunků provádělo v hlavním vlákně, mohlo by dojít k občasnému zpomalení, protože by hlavní vlákno muselo alokovat velké pole s voxelovými daty chunku.

3.2.5 Integrace s ostatními částmi prototypu

Samotná třída `VoxelWorld` neobsahuje žádnou logiku interakce s herním enginem – zaměřuje se pouze na správu chunků. Třída byla navržena tak, aby pomocí rozhraní využívající Reactive Extensions (kapitola 1.2.2.4) mohla být napojena na pomocnou třídu `ChunkRoot`, která se naopak stará pouze o interakci s herním enginem. Podobná architektura byla využita i pro samotné chunky.

Výpis kódu 3.1 Cyklus worker vlákna VoxelWorld

```
while (true)
{
  // poptej více práce
  Interlocked.Increment(ref _workRequests);
  WorkerStates[index] = null;

  // počkej na práci
  var position = _workQueue.Take();
  WorkerStates[index] = position;

  // najdi nebo vytvoř chunk
  var chunk = GetOrCreateChunk(position);

  // zpracuj chunk nad současnou vrstvou
  chunk.WorkerProcess(ActiveTerrainLayer);

  // zruš označení chunku jako zpracovávaného
  lock (_inUse)
      _inUse.Remove(position);
}
```

Výpis kódu 3.2 Kostra metody VoxelWorld.UpdateArea

```
public void UpdateArea(Rect2 area)
{
  if (_workRequests == 0)
      return;

  var center = // spočítej střed oblasti
  var maxRadius = // spočítej maximální poloměr oblasti

  // projdi chunky v kruzích od středu ven
  Loop.Radial2D(maxRadius, (x, y) =>
  {
    var position = // spočítej pozici chunku ve světě

    if (!area.HasPoint(position))
      return false; // pokud chunk není v oblasti, přeskoč

    // pokud je chunk označen jako zpracovávaný, přeskoč
    lock (_inUse)
      if (_inUse.Contains(position))
        return false;

    // pokud chunk neexistuje nebo je označen jako
    // potřebující zpracování
    if (!_chunks.TryGetValue(position, out var chunk) || chunk.NeedsWorker)
    {
      // předejdi přidání chunku do fronty vícekrát
      lock (_inUse)
        _inUse.Add(position);

      // přidej chunk do fronty
      _workQueue.Add(position);

      // pokud nejsou k dispozici žádná další vlákna,
      // ukonči zpracování oblasti
      return Interlocked.Decrement(ref _workRequests) == 0;
    }

    return false; // pokračuj v iteraci, chunk nepotřebuje zpracování
  });
}
```

3.3 Chunk

Obdobně jako `VoxelWorld` je implementace chunků rozdělena na části `VoxelChunk` a `ChunkView`, které spolu navzájem komunikují pomocí Rx. (kapitola 1.2.2.4) `VoxelChunk` se stará o generování voxelů, přístup k nim a operace s nimi, zatímco `ChunkView` napojuje data z `VoxelChunk` na herní engine.

3.3.1 Proces generování chunku

Dle návrhu 2.1 se může chunk nacházet v 8 různých stavech podle současné fáze generování. Tyto stavy jsou reprezentovány výčtovým typem `ChunkState`. Nejdůležitější část tohoto me-

chanismu je metoda `WorkerProcess`, která je volána výhradně z worker vláken a zpracovává chunk v závislosti na jeho současném stavu. (kód 3.1) `VoxelChunk` zpřístupňuje veřejnou vlastnost `NeedsWorker`, která indikuje, zda se chunk momentálně nachází ve stavu, který vyžaduje přiřazení do fronty ke zpracování. Tato vlastnost je využívána zejména ve výše zmíněné metodě `UpdateArea`. (kód 3.2)

Výpis kódu 3.3 Metoda VoxelChunk.WorkerProcess

```
  public void WorkerProcess(TerrainLayer activeLayer)
{
  switch (State)
  {
    // terén současnou vrstvou ještě nebyl vygenerován
      case ChunkState.Invalid:
      // vygeneruj hrubý terén
          State = ChunkState.RoughGen;
          GenerateDiscrete(activeLayer);
          State = ChunkState.AwaitingNeighbours;

          // hrubý terén je vygenerován, dej vědět sousedům
          foreach (var neighbor in _neighbors)
              neighbor?.IncrementNeighbourCounter();

          // pokud už jsou všichni sousedé vygenerováni,
          // rovnou nastav stav na AwaitingPostprocessing
          // nezačínej ale rovnou, pouze uvolni vlákno -
          // pořadí zpracování rozhodne VoxelWorld
          if (NeighboursGenerated)
              State = ChunkState.AwaitingPostprocessing;

          break;

      case ChunkState.AwaitingPostprocessing:
          State = ChunkState.Postprocessing;
          PostProcess(activeLayer);
          State = ChunkState.AwaitingRemesh;
          break;

      case ChunkState.AwaitingRemesh:
          // cyklus pro případ, že je chunk modifikován při meshingu
          // potřeba z důvodu minimalizace odezvy na změny
          while (State == ChunkState.AwaitingRemesh)
          {
              State = ChunkState.Meshing;
              Remesh(activeLayer);
          }
          State = ChunkState.Ready;
          break;
  }
}
```

3.3.2 Úložiště

Další klíčovou rolí třídy `VoxelChunk` je poskytování přístupu k voxelům. Protože generátor s voxely pracuje i mimo chunky, ale i v samotných vrstvách, bylo nutné implemetovat úložitě voxelů dostatečně genericky a to nejenom v závislosti na uloženém typu, ale i dimenzionalitě.

Základem je abstraktní třída `ArrayBase<TKey, TValue>`, která obsahuje společné vlastnosti a metody pro všechny implementace. Tato třída je parametrizována dvěma typy, kde `TKey` je typ klíče, kterým se přistupuje k datům, a `TValue` je typ uložených dat. Díky použití vektoru jako generického klíče společné metody fungují bez ohledu na to, jestli se jedná o 2D nebo 3D pole. Pro rychlou iteraci umožňuje tato třída přístup k datům i přímým indexem.

Konkrétní implementace úložišť pro různé dimenze, jako jsou `Array2D<T>` a `Array3D<T>`, dědí z této základní třídy a rozšiřují její funkčnost o metody pro přístup k datům bez použití klíče, či pomocné metody specifické jejich dimenzionalitě. V obou případech se jedná o ploché pole s nepřímou adresací.

3.3.3 Meshing

Meshing, tedy proces vytváření 3D modelu na základě voxelových dat, je implementován v metodě `NaiveMesher.BuildMesh`. Jedná se o jednoduchou naivní implementaci, která dává prioritu rychlosti před optimálním výstupem. Díky tomu, že bloky v tomto prototypu nemají textury, ale pouze barvu, mesher musí rovnou vytvořit povrch bloků s odpovídající barvou. Zároveň ovšem ne všechny chunky (kapitola 2.6) mají stejný typ voxelů – je potřeba ji každému bloku flexibilně přiřadit.

Tohoto je dosaženo zgeneričtěním metody, která přijímá jako generický parametr typ dat v poli voxelů. Pomocí dalšího parametru funkce `Func<T, Color>` je možné specifikovat, jakým způsobem se získá barva voxelu. Tato metoda je volána při každém voxelu a výsledná barva je použita pro vytvoření plochy bloku. Pokud má být blok prázdný, je funkcí vrácena průhledná barva.

Pokud se blok nachází na hraně chunku, kde není možné získat data o sousedních blocích, je o jeho okolí v sousedním chunku implicitně předpokládáno, že bude prázdné, a proto je mezi ně vložena plocha. Tímto jednoduchým způsobem je zamezeno chybějícím stranám bloků na hranici dvou chunků – i kdyby nakonec tato sousední pozice byla zaplněna, v pouze se v daném místě vytvoří dvojité stěny, které však nebudou viditelné.

3.4 Vrstvy

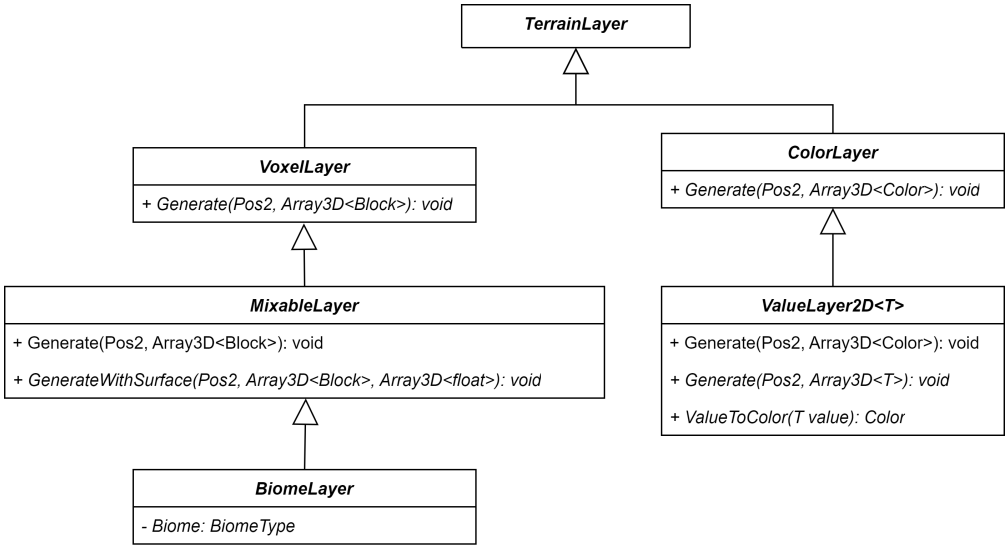
V návrhu byly identifikovány dva typy vrstev generátoru (kapitola 2.6), které se liší pouze typem výstupních dat. V průběhu implementace se ale ukázalo, že bude potřeba vrstvy rozšířit o další funkce podle využití. Výsledkem je hierarchie tříd, která je znázorněna na diagramu 3.1.

- **`TerrainLayer`**
  Základní třída pro všechny vrsvy. Neobsahuje žádnou funkcionalitu, slouží čistě k práci s vrstvami jako s kolekcí bez nutnosti znát konkrétní typ vrstvy – z tohoto důvodu také není generická.

- **`VoxelLayer`**
  Základní terénová vrstva. Má jednu abstraktní metoda `Generate(Pos2<GlobalSnapped>, Array3D<Block>)`, které má za úkol vygenerovat hrubá data pro jeden chunk v závislosti na jeho globální pozici. (kapitola 3.4.1)

- **`ColorLayer`**
  Identická s `VoxelLayer`, ale pracuje s polem barev `Array3D<Color>`.

- **`MixableLayer`**
  Poskytuje společnou implementaci methody `Generate` třídy `VoxelLayer` a rozšiřuje ji o metodu `GenerateWithSurface(Pos2<GlobalSnapped>, Array3D<Block>, Array3D<float>)`. Tato metoda je potřeba pro interpolaci sousedních biomů. Detailněji popsáno v sekci 3.4.2.



Obrázek 3.1 Diagram stromu dědičnosti vrstev

- `BiomeLayer`
  Vrstva popisující jeden konkrétní biom. Rozšiřuje `MixableLayer` o vlastnost `Biome` typu `BiomeType`. Využívané pro mapování vrstev na rozdělení biomů.

- `ValueLayer2D<T>`
  Pomocná třída pro 2D datové vrstvy. Obsahuje společnou univerzální implementaci metody `ColorLayer.Generate` a potomkům zprostředkovává příjemnější generické rozhraní.

Pokud vrstva vyžaduje vkládání modelů či jiné operace v rámci postprocessing kroku, může implementovat rozhraní `IPostprocessableLayer`, které obsahuje metodu `Postprocess(…)`. Pokud jej vrstva implementuje, tak je automaticky volána generátorem pro chunky ve stavu Postprocessing.

Vrstvy jsou definovány a vzájemně propojeny v singleton třídě `TerrainLayers`, která slouží jako registr vrstev.

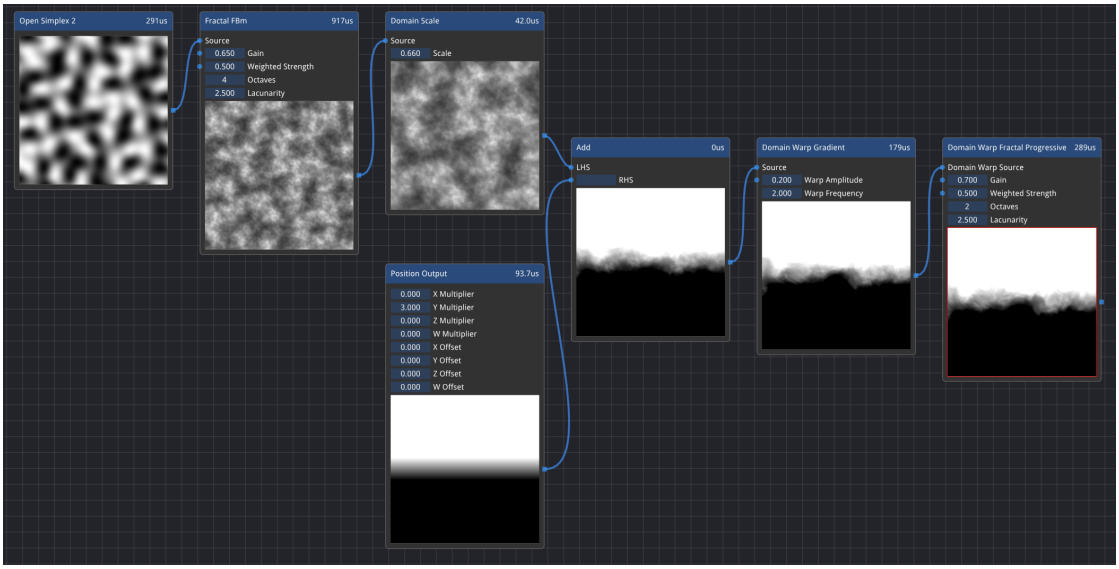
3.4.1 Hrubý terén

Hrubý terén je generován metodou `Generate` každé vrstvy. Z pohledu sytému generátoru na implementaci této vrstvy nezáleží, ale pro účely této práce je vždy použita knihovna FastNoise 2 v kombinaci s vlastními operacemi nad výslednými daty.

Knihovna využívá koncept generovacího grafu – každá operace je reprezentována jako uzel, který může být napojen na jiné uzly. Této způsobem lze tvořit i složité operace přímo prostřednictvím této knihovny. Vlastní operace, které nelze vyjádřit v grafové podobě, nebo by se grafem obtížně vyjadřovaly, jsou implementovány manuálně nad výslednými daty z uzlů.

Graf je vždy definován ve statickém konstruktoru třídy a reference na něj je uložena do statického readonly fieldu. Tím je zajištěno, že se graf vytvoří pouze jednou a všechny instance třídy ho budou sdílet. Operace generování jsou v knihovně FastNoise 2 thread safe, takže není třeba žádným způsobem synchronizovat přístup worker vláken.

Ukázku grafu pro generování šumu ve vizuální podobě lze najít na obrázku 3.2, implementační podobu v rámci vrstvy `CaveLayer` pak v kódu 3.4.



Obrázek 3.2 Ukázka grafu generátoru šumu FastNoise 2 [7]

Výpis kódu 3.4 Ukázka implementace generování hrubého terénu

```
  private static readonly Generator SurfaceGen;
  private static readonly Generator BlockGen;

  static CaveLayer()
  {
      // sestavení šumového grafu
      SurfaceGen = new Perlin()
          .WithScale(0.01f)
          .WithScale(Dim.Y, 3f)
          .WithBackWall();

      var value = new CellularValue();
      var distance = new CellularDistance();
      distance.SetReturnType(ReturnType.Index0Div1);

      var sub = new Subtract();
      sub.SetLHS(value);
      sub.SetRHS(distance);

      BlockGen = value
          .WithScale(0.05f);
  }

  public override void GenerateWithSurface(
    Pos2<GlobalSnapped> position,
    BlockArray blockArray,
    Array3D<float> surfaceArray)
{
  // generování povrchu
  SurfaceGen.FillArray(surfaceArray, position);

  // vypůjčení pomocného pole
```

```
    using var _ = Pool.Get(out Array3D<float> valueArray);

    // generování jasu bloků
    BlockGen.FillArray(valueArray, position);

    // vlastní operace nad výslednými daty
    int i = 0; // inkrementální index pro rychlý přístup do pole voxelů
    for (int z = 0; z < Bpa; z++)
    for (int y = 0; y < Bpa; y++)
    for (int x = 0; x < Bpa; x++)
    {
        // výpočet jasu bloku na základě dat z FastNoise 2
        var offset =
          (sbyte)(sbyte.MaxValue * Mathf.Clamp(valueArray[i], -1, 1));

        // vyplnění voxelu - jestli je na této pozici blok nebo ne
        // určuje pro VoxelLayer vrstvy vrstva CombinationLayer na
        // základě surfaceArray a BiomeMapLayer
        blockArray[i] = new Block(BlockType.Rock, offset);

        i++;
    }
}
```

3.4.1.1 Přechod na FastNoise 2

Z počátku nebyla implementace projektu postavena nad knihovnou FastNoise 2, ale nad její zjednodušenou a přenosnou verzí FastNoise Lite [12]. Tato knihovna je napsaná v C#, čímž nabízí lepší integraci se zbytkem projektu a nevyžaduje kompilaci a distribuci sdílené knihovny.

V průběhu vývoje se ale ukázalo, že generování šumu je zdaleka nejpomalejší částí generování terénu. Na základě výkonnostního testování (kapitola 4.1.1) těchto dvou knihoven bylo rozhodnuto o přechodu na FastNoise 2, která díky svému C++ jádru a SIMD optimalizacím nabízí výrazně lepší výkon.

3.4.1.2 Zpracování po chuncích

Další zásadní optimalizací procesu generování hrubého terénu byl přechod na zpracování po chuncích. Původní implementace generovala terén po jednotlivých blocích – metoda `Generate` byla volána pro každý blok zvlášť. Toto řešení bylo zvoleno z důvodu jednoduchosti implementace a ergonomie z pohledu vývojáře, nebylo totiž potřeba mít v každé vrstvě samostatný 3D cyklus.

Testování v sekci 4.1.2 ale ukázalo, že toto řešení nebylo výkonnostně optimální, protože volání metod v jazyce C# má jistou režii. Za normálních okolností je zanedbatelná, ale v případě generování terénu, kde je potřeba metodu volat řádově milionkrát, je tato režie značná.

3.4.2 Rozdělení a interpolace biomů

Na základě myšlenky z návrhové sekce 2.6.1 byla implementována datová vrstva `BiomeMapLayer : ValueLayer2D<PartialBiome[]>`, která generuje plynulé rozdělení biomů na 2D rovině. Každé buňce přiřadí náhodně vybraný typ biomu `BiomeType`. Pokud je ale vertikální souřadnice buňky vetší než 0, jako biom je přiřazen vždy `BiomeType.GrassSurface`, který odpovídá povrchu terénu. Na rozdíl od návrhu sama o sobě neobsahuje žádnou logiku pro interpolaci voxelů, díky čemuž ji lze samostatně vizualizovat.

Interpolaci terénu řeší vrstva `CombinationLayer`. Na základě plynulého rozdělení biomů z `BiomeMapLayer` lineárně interpoluje jednotlivé `BiomeLayer` vrstvy. Hodnoty pole dodatečného parametru metody `MixableLayer.GenerateWithSurface`, `Array3D<float> surface`, reprezentují vzádelnost dané pozice od povrchu. S touto informací je pak interpolace zjednodušena na triviální linární mapování.

`BiomeMapLayer` také vrací typy biomů použité v daném chunku. Protože celý systém generování funguje chunk po chunku, je důležité předejít zbytečnému generování nevyužitých vrstev.

3.4.2.1 Optimalizace biomových přechodů

Plynulé přechody biomů je možné implementovat standardním rozmazáním 2D hodnot voronoi šumu. Tento přístup však vyžaduje generování šumu i pro okolní oblasti, jinak by bylo rozmazání kolem hran chunku nedokonalé. Dále časová náročnost rozmazání závisí na poloměru rozmazání, a protože pro plynulé přechody je třeba použít velký poloměr, může být tento proces velmi pomalý.

Vzhledem k těmto nevýhodám byl nakonec zvolen alternativní přístup, který využívá Voronoiho diagramu a lineární interpolaci. Voronoiho diagram je generován pomocí implementace Fortuneova algoritmu, a to pouze pro oblast chunku. Místo průměrování hodnot okolních pozic stačí zjistit vzdálenost vůči úsečkám Voronoi diagramu, na základě kterých je pak aplikována lineární interpolace všech biomů v dosahu. Tento způsob je mnohem rychlejší a pro biomové přechody poskytuje přijatelné výsledky. Rychlost tohoto přístupu byla ověřena v testu uvedeném v sekci 4.1.4.

3.5 Recyklační systém

V kapitole 2.8 bylo rozhodnuto o implementaci recyklačního systému, který by měl zamezit nadměrné alokaci pomocných objektů za běhu programu. Tento systém je implementován pomocí singleton třídy `MultiObjectPool`, která je schopna vytvářet a recyklovat objekty všech typů s výchozím konstruktorem.

Pro usnadnění používání této třídy a zamezení chybám spojeným s návratem objektů do poolu, bylo vrácení objektů implemetováno pomocí `IDisposable` vzoru. Při vypůjčení objektu `var disposable = MultiObjectPool.Get<T>(out T obj)` je automaticky z poolu odebrán a po zavolání `disposable.Dispose()` je vrácen zpět do recyklačního systému. To umožnuje použít klíčové slovo `using` pro vytvoření bloku, ve kterém je objekt používán. Po opuštění bloku se automaticky zavolá metoda `Dispose()` a objekt je vrácen do poolu bez nutnosti explicitního volání metody.

Výpis kódu 3.5 Příklad použití MultiObjectPool

```
using (Pool.Get(out Foo foo))
{
  // použij foo
}

// nebo

// začátek nějakého bloku
{
  using var disposable = Pool.Get(out Foo foo, out Foo2 foo2);

  // použij foo1 a foo2
}
```

3.5.1 Další redukce alokací

Kromě recyklačního systému lze v některých případech předejít alokaci na haldě pomocí klíčového slova `stackalloc`, které umožňuje alokovat paměť přímo na zásobníku. Výhodou je rychlá a efektivní alokace paměti, která se automaticky uvolní po opuštění aktuálního bloku kódu bez zatížení GC. Tato technika je vhodná především pro práci s menšími pomocnými poli a datovými strukturami, které mají omezenou dobu života a nejsou předávány mezi různými částmi generátoru.

3.6 Modely

Základním prvkem systému vkládání modelů je třída `ModelScene`. Model a metadata jsou uloženy jako Godot scény a tato třída slouží jako jejich kořenový uzel. Zpřístupňuje veřejné rozhraní pro přístup k modelu a jeho zónám a také obsahuje metody pro transformaci a vložení modelu do terénu. Každá modelová scéna (kapitola 2.5) obsahuje:

- Právě jednu instanci `VoxelModel` nebo `MeshModel`.
- Libovolné množství omezovacích zón `ConstraintZone`.
- Libovolné množství přepisovacích zón `BlockOverrideZone`.

Transformace modelů jsou reprezentovány pomocí 4x4 transformačních matic, které jsou aplikovány na všechny vrcholy modelu před voxelizací. Pokud se jedná o voxelový model, rotace je omezena na násobky 90 stupňů.

Oba typy zón dědí z abstraktní třídy `ModelZone`, která obsahuje základní společnou funkcionalitu pro iteraci přes všechny pozice v zóně. Třída `ModelZone` sama dědí z Godot třídy `MeshInstance3D`, což umožňuje z editoru zóně přiřadit libovolný mesh, který zároveň slouží jako vizualizace dané zóny. `ModelZone` ovšem podporuje pouze 3 základní tvary poskytované přímo Godot Enginem: `BoxMesh`, `SphereMesh` a `CylinderMesh`. Kombinací těchto tří tvarů je možné pokrýt většinu potřebných zón.

3.6.1 Nedokonalá iterace omezujících zón

Na rozdíl od přepisovacích zón, které musí být vyhodnoceny precizně, je možné omezující zóny vyhodnotit pouze přibližně. Pokud některé pozice v zóně nejsou vyhodnoceny, může to vést k chybě v případě, že je omezení zóny porušeno právě v těchto pozicích. Cílem omezovacích zón je ovšem pouze zabránit těm nejhoším případům vložení modelu a proto je možné jisté procento chyb tolerovat. S ohledem na tuto skutečnost třída `ConstraintZone` implementuje dvě vlastnosti:

- `CheckStep`
  Velikost kroku iterace zóny. Hodnota 1 znamená, že se zóna vyhodnocuje na každé pozici. Hodnota 2 znamená, že se zóna vyhodnocuje pouze na každé druhé pozici atd. Čím vyšší je hodnota, tím rychlejší je vyhodnocení zóny, ale zároveň je zóna méně přesná.
- `PercentRequired`
  Minimální procento vyhodnocených pozic, které musí být splněno, aby byla omezovací zóna považována za splněnou. Umožňuje nastavit toleranci chyb.

Kapitola 4
Testování

Testování funkcionality probíhalo průběžně při vývoji. Naleznuté chyby byly co nejdříve debugovány a opraveny. Systematičtější testování je vzhledem k inherentní náhodnosti generátoru obtížné. Tato kapitola se proto zaměřuje především na testování výkonnosti.

4.1 Výkonností

Několik aspektů systému bylo výkonnostně testováno za účelem zvolení vhodného přístupu k implementaci. K dosažení konzistence mezi testy bylo testování prováděno na jednom počítači s pevně danými parametry:

- CPU: AMD Ryzen 5 5600H 45W
- RAM: 32GB DDR4 3200MHz
- GPU: RTX 3060 6GB 130W
- OS: Windows 11 Pro 64-bit

Pro spouštění testů a samotné měření času byla použita knihovna `BenchmarkDotNet` [24]. Všechny testy byly spouštěny několikrát a výsledný čas byl průměrován.

4.1.1 FastNoise

Test rychlosti verzí knihoven FastNoise probíhal vygenerováním 2D šumu o velikosti 1024x1024 pixelů. Do výsledného času se nezapočítává inicializace knihovny, v případě FastNoise 2 včetně tvorby generovacího grafu. Knihovna poskytuje metodu generování šumu nejen pro konkrétní souřadnici, ale i pro celé rovnoměrné pole najednou.

Tabulka 4.1 Test verzí knihovny FastNoise

Knihovna	Perlin	Simplex	Voronoi
FastNoise Lite	11,2 ms	14,6 ms	31,6 ms
FastNoise 2	11,1 ms	12,8 ms	46,7 ms
FastNoise 2 hromadně	1,3 ms	1,7 ms	5,8 ms

I přes to, že implementace FastNoise 2 je výrazně rychlejší než FastNoise Lite, je v případě opakovaného volání funkce pro jednotlivé souřadnice stále pouze srovnatelná s verzí Lite. Toto je

způsobeno režií spojenou s voláním nativního kódu z managed prostředí C#. V případě metody hromadného generování je tato režie zanedbatelná a tím pádem i výkonnostní rozdíl výrazně vyšší.

4.1.2 3D iterace

V průběhu generování terénu je potřeba často iterovat přes všechny voxely chunku. Standardní tříúrovňový cyklus je neergonomický a tím pádem náchylný k chybám, byla tedy otestována řada aternativ. Tento test je zároveň relevantní pro porovnání generování terénu blok po bloku oproti iteraci celého chunku. V testu „Lambda“ totiž měří i režii opakovaného volání anonymní funkce.

Tabulka 4.2 Test 3D iterace

	32^3	64^3	128^3
Vnořené cykly	9 μs	67 μs	590 μs
Zploštěný cyklus	8 μs	63 μs	507 μs
Iterátor	276 μs	2686 μs	21597 μs
Lambda	66 μs	446 μs	4415 μs

Z těchto výsledků je patrné, že vysokoúrovňové konstrukce jako jsou iterátory a lambdy jsou výrazně pomalejší a tedy nevhodné pro kritické sekce generátoru.

4.1.3 Voxelizace

Jak bylo zmíněno již v kapitole 4.1.4, povrchová voxelizace by měla být algoritmicky méně časově náročná, než kolizní. Tento test ověřuje, zda je toto tvrzení pravdivé, a pokud ano, jaký je rozdíl v rychlosti. Test probíhal na modelech Crystal, Tree a Rock, které byly použity i v rámci prototypu.

Tabulka 4.3 Test voxelizace

Model	Povrchová	Kolizní
Crystal	11 ms	428 ms
Tree	23 ms	935 ms
Rock	0,8 ms	13 ms

Výsledky testů potvrzují předpoklad, že povrchová voxelizace je výrazně rychlejší než kolizní. Největší rozdíl je u modelů s nízkým počtem polygonů, které ale zabírají velký objem.

4.1.4 Interpolace biomů

Pro ilustraci rozdílu mezi interpolací a rozmazáváním byl vytvořen test, který měří čas potřebný pro uhlazení přechodů mezi buňkami voronoi šumu v závisloti na poloměru rozmazání.

Tabulka 4.4 Test interpolace

Metoda	10	20	40	80
Rozmazání	1,3 ms	5 ms	19,3 ms	76,3 ms
Interpolace	2,8 ms	2,8 ms	2,8 ms	2,8 ms

Pro malé poloměry rozmazaní znamená vysoká výpočetní cena tvorby Voronoi diagramu, že jednoduché rozmazaní je i přes svou výpočetní komplexitu rychlejší. Při zvyšování poloměru se však čas potřebný pro rozmazaní rychle vymyká kontrole a interpolace se stává v porovnání výrazně rychlejší.

Závěr

Tato bakalářská práce se zabývala analýzou a vývojem prototypu procedurálního generátoru voxelového terénu. Hlavním cílem bylo vytvořit systém generování, který dokáže tvořit zajímavý a plynulý terén s přijatelnou rychlostí generování a zároveň být dostatečně flexibilní pro další vývoj.

Jako součást práce byly zkoumány různé metody procedurálního generování, jako jsou různé typy šumů a algoritmy generování terénu. Dále byla zkoumána reprezentace voxelů a způsob, jakým je možné je efektivně uložit v paměti. Také proběhla analýza frameworků, knihoven a dalších technických možností pro samotnou implementaci, v závislosti na které byl vybrán herní engine Godot Engine.

V praktické části byl na základě výsledků analýzy úspěšně vytvořen prototyp generátoru, který plně splňuje stanovené cíle. Díky rozdělení generování do několika kroků a chunkování dokáže systém generovat terén s využitím libovolného počtu vláken a to bez omezení velikosti generovaného světa. Do terénu dále dokáže vložit předem definované modely s ohledem na prostorové omezení, čímž je zajištěna nehomogennost terénu. Vývojáři poskytuje vývojové rozhraní, které dokáže vizualizovat jednotlivé části generátoru.

V budoucnu by bylo možné rozšířit projekt o další funkce, například dynamické generování vegetace a vodních ploch nebo implementaci interakce hráče s terénem. Také by bylo zajímavé prozkoumat další možnosti optimalizace algoritmů pro generování a vykreslování terénu, aby bylo možné generovat ještě rozsáhlejší a detailnější světy. Tato práce se navíc nezabývala streamováním terénu z disku nebo přes síť. Pro budoucí aplikace a vývoj by bylo vhodné věnovat pozornost i této oblasti.

Bibliografie

1. SAMET, Hanan. The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR)*. 1984, roč. 16, č. 2, s. 187–260.
2. LAINE, Samuli; KARRAS, Tero. Efficient sparse voxel octrees–analysis, extensions, and implementation. *NVIDIA Corporation*. 2010, roč. 2, č. 6.
3. LINDEN, Roland van der; LOPES, Ricardo; BIDARRA, Rafael. Procedural Generation of Dungeons. *IEEE Transactions on Computational Intelligence and AI in Games*. 2014, roč. 6, č. 1, s. 78–89. Dostupné z DOI: 10.1109/TCIAIG.2013.2290371.
4. MIJAILOVIC, Vidak. *A Graph-Based Approach to Procedural Terrain*. 2015. TRITA-ICT-EX, č. 2015:72.
5. HYTTINEN, Tuomo. *Terrain synthesis using noise*. 2017. Dostupné také z: https://urn.fi/URN:NBN:fi:uta-201705081539. Dipl. pr.
6. SPJUT, Josef B.; KENSLER, Andrew E.; BRUNVAND, Erik. Hardware-accelerated gradient noise for graphics. In: *ACM Great Lakes Symposium on VLSI*. 2009.
7. AUBURN. *FastNoise 2 - Noise Tool*. 2023-03-01. 0.9.7-alpha. Dostupné také z: https://github.com/Auburn/FastNoise2/releases/tag/v0.9.7-alpha/. Windows / Linux 64-bit / macOS 64-bit.
8. ALEKSANDROV, Mitko; ZLATANOVA, Sisi; HESLOP, David J. Voxelisation algorithms and data structures: A review. *Sensors*. 2021, roč. 21, č. 24, s. 8241.
9. *Unity documentation* [online]. 2023-04-15. Dostupné také z: https://docs.unity.com/.
10. *Unreal engine 5.1 documentation* [online]. 2023-04-15. Dostupné také z: https://docs.unrealengine.com/5.1/en-US/.
11. *Godot Docs – 4.0 branch* [online]. 2023-04-15. Dostupné také z: https://docs.godotengine.org/en/stable/.
12. *FastNoise Lite* [online]. 2023-04-15. Dostupné také z: https://github.com/Auburn/FastNoiseLite/.
13. *FastNoise 2* [online]. 2023-04-15. Dostupné také z: https://github.com/Auburn/FastNoise2/.
14. *FastNoiseOO* [online]. 2023-04-15. Dostupné také z: https://github.com/deR1t/FastNoiseOO/.
15. *FastNoise 2 C# bindings* [online]. 2023-04-15. Dostupné také z: https://github.com/Auburn/FastNoise2Bindings/.
16. EPHTRACY. *MagicaVoxel*. 2023-03-01. Ver. 0.99.7.0. Dostupné také z: https://ephtracy.github.io/. Windows 64-bit.

17. *VoxReader* [online]. 2023-04-15. Dostupné také z: https://github.com/sandrofigo/VoxReader/.
18. *MagicaVoxel format documentation* [online]. 2023-04-15. Dostupné také z: https://github.com/ephtracy/voxel-model.
19. *Dear ImGui* [online]. 2023-04-15. Dostupné také z: https://github.com/ocornut/imgui.
20. *imgui-godot* [online]. 2023-04-15. Dostupné také z: https://github.com/pkdawson/imgui-godot.
21. *ImGui.NET* [online]. 2023-04-15. Dostupné také z: https://github.com/mellinoe/ImGui.NET.
22. *Reactive Extensions* [online]. 2023-04-15. Dostupné také z: https://github.com/dotnet/reactive.
23. *ReactiveUI* [online]. 2023-04-15. Dostupné také z: https://github.com/reactiveui/reactiveui.
24. *BenchmarkDotNet* [online]. 2023-04-15. Dostupné také z: https://github.com/dotnet/BenchmarkDotNet.

Obsah přiloženého média

- `thesis.pdf` ..... text práce ve formátu PDF
- `bin` ..... adresář s předem sestaveným projektem
- `src` ..... zdrojové kódy implementace
  - `Sidegen` ..... Godot Engine projekt
  - `Benchmarks` ..... výkonnostní testy
  - `Godot.Extensions` ..... rozšíření Godot API
  - `FastNoiseOO` ..... modifikovaná knihovna FastNoiseOO