Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

BAKALÁŘSKÁ PRÁCE

Voxelový terén a nástroje jeho editování

2020

Daniel Bazala

Vedoucí práce: RNDr. Eduard Bartl, Ph.D.

Studijní obor: Aplikovaná informatika, prezenční forma

Bibliografické údaje

Autor:	Daniel Bazala
Název práce:	Voxelový terén a nástroje jeho editování
Typ práce:	bakalářská práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2020
Studijní obor:	Aplikovaná informatika, prezenční forma
Vedoucí práce:	RNDr. Eduard Bartl, Ph.D.
Počet stran:	36
Přílohy:	1 CD/DVD
Jazyk práce:	český

Bibliograhic info

Author:	Daniel Bazala
Title:	Voxel terrain and tools editing it
Thesis type:	bachelor thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2020
Study field:	Applied Computer Science, full-time form
Supervisor:	RNDr. Eduard Bartl, Ph.D.
Page count:	36
Supplements:	1 CD/DVD
Thesis language:	Czech

Anotace

Cílem bakalářské práce bylo naprogramovat realistickou 3D vizualizaci terénu pomocí voxelové reprezentace. S využitím voxelové reprezentace poté umožnit objemovou editaci terénu za běhu programu. Algoritmus programu terén automaticky generuje podle nastavitelných parametrů, řeší převod voxelové reprezentace na polygonovou a dále zajišťuje texturování a materiály terénu. Celý program je koncipovám jako modul herního engine.

Synopsis

The bachelor thesis goal was to develop a realistic 3D terrain visualization benefiting of voxel representation. Furthermore to enable volumetric terrain editation using the voxel representation in application real time. The program algorithm automatically generates terrain according to the settable parameters and it solves the conversion of voxel to polygonal representation. It also handles terrain textures and materials. The whole program has been designed as a game engine module.

Klíčová slova: terén; editace; simulace; 3D; voxel; marching cubes; unreal engine

Keywords: terrain; editable; simulation; 3D; voxel; marching cubes; unreal engine

Děkuji vedoucímu této práce RNDr. Eduardu Bartlovi, Ph. D. za podněty k tématu a pomoc se zpracováním problematiky. Dále bych chtěl také poděkovat celému pedagogickému sboru KI PřF UPOL za kvalitní a individuální přístup při vzdělávání.

Místopřísežně prohlašuji, že jsem celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

datum odevzdání práce podpis autora

Obsah

Seznam obrázků

1 Úvod

Převážná většina moderních počítačových her a simulací pracující s otevřeným světem řeší problém terénu. Přístupů, jak řešit realistickou simulaci terénu, je více. Obvykle se volí ten přístup, který nejvíce vyhovuje potřebám aplikace s ohledem na výpočení a paměťové nároky cílových zařízení, ale také s ohledem na náročnost implementace řešení.

Prvním a nejjednodušším řešením je simulovat terén jakožto statickou síťovinu o $n \times n$ bodech, kde jsou sousedící body navzájem propojené hranami. Takto strukturovaný terén můžeme následně zvrásnit nastavením různých výškových souřadnic jeho bodů. Gradientní změna výšky bodů zajišťuje pozvolnou změnu sklonu terénu. Toto se obvykle v případě statického terénu dělá v editoru herního engine za pomoci nástrojů upravujících výšku terénu v určitém poloměru podle matematických funkcí. Běžně se používá také tzv. výšková mapa [10], která jednotlivým bodům terénu přiřazuje jejich výškovou souřadnici. Výhodou tohoto řešení je nízká výpočetní i paměťová náročnost a jednoduchá implementace.

Druhým o něco složitějším řešením je terén dynamicky generovat za běhu programu. Struktura terénu může být podobná jako v prvním řešení, nicméně s tím rozdílem, že terén musí být rozdělen na menší části (obvykle čtvercového půdorysu), které na sebe navazují a generují se v místech, kde to simulace zrovna vyžaduje. Při tomto řešení už terén není tvarován uživatelem v editoru herního engine, ale souvislým algoritmem zajišťujícím návaznost a různorodost terénu. Výhodou tohoto řešení je nižší režie návrháře terénu, ale za cenu náročnější implementace a mírně vyšší výpočetní náročnosti.

Je možná ještě varianta statického terénu, který je rozdělen na menší části z důvodu optimalizace výkonu a vykreslování pouze viditelných částí.

Třetím a nejnáročnějším řešením, kterým se bude zabývat i tato práce je řešení pomocí voxelové reprezentace. Voxel je trojrozměrnou analogií ke grafickému pojmu pixel. Při tomto řešení již nepovažujeme terén pouze za síťovinu o $n \times n$ bodech s různou výškou, ale bereme v úvahu i jeho objem. Tím se ale celé řešení komplikuje, protože většina současných herních engine neumí voxelovou reprezentaci vykreslovat. Z tohoto důvodu je tedy třeba před vykreslením ještě převést voxelovou reprezentaci terénu na polygonovou reprezentaci, se kterou běžně pracuje většina moderních herních engine a umí ji vykreslit ve 3D prostoru. Nevýhody tohoto řešení: algoritmy generování jsou mnohem náročnější na výkon i implementaci, protože je třeba generovat i vnitřní strukturu terénu (časová složitost min. v kubickém čase), potřeba provádět polygonizaci a také podstatně vyšší paměťová náročnost. Výhodou řešení je možnost simulace terénu tak, jak reálně existuje i s jeho objemem a dále snadná editace terénu za běhu programu pouhým přidávám či odebíráním voxelů pomocí prostorových štětců.

Výstupem této práce je modul herního engine, který zajišťuje generování voxelového terénu, jeho polygonizaci, materiál s texturami a vykreslování pomocí standardních metod pro polygonovou reprezentaci. Terén lze za běhu aplikace plně editovat pomocí prostorových štětců a nastavením parametrů generátoru terénu.

2 Herní engine

Jak již bylo výše zmíněno, aplikace voxelového terénu, kterou se tato práce zabývá, je koncipovaná jakožto modul herního engine. Rozhodl jsem se tak z důvodu co největší jednoduchosti a úspory času. Kdybych měl programovat řešení kolizí, vykreslování, texturování nebo pohybu hráče bez pomoci sofistikovaného software, který již tato řešení v sobě zahrnuje, tato práce by svým obsahem byla mnohonásobně rozsáhlejší a složitější. Právě takovým softwarem usnadňujícím programování videoher a simulací jsou herní engine.

2.1 Obecná definice

Herní engine je software sloužící k vývoji videoher. Před příchodem herních engine se často stávalo, že se u podobných typů her opakoval stejný kód jako např. počítání kolizí, zobrazování modelů nebo vykreslování scény. Herní engine tyto základní problémy řeší a umožňuje tak oddělit kód jádra od kódu samotné hry. Vytváří tedy jakousi abstrakční bariéru, která usnadňuje herním vývojářům práci tím způsobem, že jim nabízí k dispozici funkce a nástroje, které při tvorbě her využívají [20].

Můžeme se setkat s více či méně specializovanými engine. Na této míře specializace vždy závisí, kolik práce s tvorbou videohry budeme mít. Pokud zvolíme engine specializovaný na náš typ hry, ušetří nám to čas i práci, protože většinu funkcionality engine již bude obsahovat v sobě. Zároveň se také ztenčí hranice mezi kódem engine a kódem samotné hry. Někdy tato hranice vůbec nemusí být jasná a kód může splývat. Případně může být nutné zasáhnout do kódu samotného engine (pokud to umožňuje) a upravit jej k vlastním specifickým potřebám hry [22].

Současné populární herní engine často nabízí velké množství nástrojů a funkcí, které jejich uživatel při tvorbě videohry ani všechny nevyužije. Engine se tak stávají velice rozsáhlé a je nutné jejich jednotlivé nástroje rozdělovat do modulů. Tento přístup přináší řadu výhod, např. moduly, které nejsou součástí jádra a nepotřebujeme je k vývoji, můžeme deaktivovat a ušetřit tak výkon a paměť aplikace i velikost výsledné hry. Další výhodou je jednodušší přidávání uživatelsky vytvořeného kódu ke kódu samotného engine.

2.2 Unreal Engine 4

Unreal Engine 4 (dále jen UE4) je čtvrtá generace tohoto engine, poprvé představená v roce 2012 firmou Epic Games. Oproti předchozím generacím má několik výhod a nových technologií. Mezi ty nejdůležitější patří globální osvětlení v reálném čase, které umožňuje efektivně počítat dynamické osvětlení bez nutnosti osvětlení předpočítat, pokročilé vizuální skriptování, nástroje pro tvorbu a editaci krajiny, podpora virtuální reality, integrace pro multiplatformní vývoj včetně mobilních platforem či nový systém návrhu UI a animací [5].

Jako programovací jazyk UE4 využívá C++. Předchozí generace engine byla napsaná v C++, ale samotné programování probíhalo ve speciálně vyvinutém jazyku nazvaném UnrealScript [6]. Tento jazyk umožnoval snadné a rychlé programování her, protože byl vyvinut a přizpůsoben přesně k tomuto účelu. Nevýhodou ale bylo, že skutečné jádro engine, napsané v jazyku C++, bylo uzavřené a běžnému programátorovi nepřístupné. Pokud programátor při vývoji narazil na nutnost upravit fungování samotného engine, musel si velice draze zakoupit licenci na zdrojové kódy, nebo požádat o úpravu samotné vývojáře engine. Tento problém v UE4 zcela odpadá, protože jádro engine je otevřené a jak samotný engine, tak herní vrstva jsou programovány v jazyku C++. V případě potřeby tedy není žádný problém zasáhnout do kódu engine a překompilovat jej do vlastní verze.

Rozhodl jsem se tento engine zvolit z důvodu otevřeného zdrojového kódu, obsáhlé dokumentace a také kvůli faktu, že pro nekomerční účely je zcela zdarma. Navíc v tomto engine vyvíjím již několik let, a proto s ním mám zkušenosti. Nutno ale poznamenat, že aplikace by stejně dobře mohla být vytvořena i v konkurečních engine.

3 Struktura terénu

Reálný terén, se kterým se setkáváme při procházce krajinou v přírodě, je souvislý, má objem a můžeme jej upravovat po téměř libovolně malých částech objemu. Pokud bychom chtěli být důslední, museli bychom ještě vzít v úvahu fakt, že reálný terén je zakřivený a ve skutečnosti tvoří geoid. Tento fakt ovšem většina simulací terénu zanedbává a zanedbám jej i já v této práci. Nelze jej ovšem ignorovat ve hrách nebo simulacích zobrazující terén z velké výškové vzdálenosti (např. simulátorech letectví, vesmírných simulátorech). Takovým případem může být například hra Space Engineers, která tento problém zohledňuje [14].

Další kompromis, který musíme při návrhu simulace udělat, je aproximace objemu. Reálný terén se skládá z velice malých částic, se kterými lze manipulovat. Taková simulace je na současných osobních počítačích nemožná jak z výpočetních, tak paměťových nároků. Musíme si totiž uvědomit, že pokud bychom chtěli simulovat výsek terénu o velikosti $1km \times 1km \times 1km$ pomocí částic o velikosti zrn hrubého písku (velikost částice cca. 1 mm), zabralo by nám to v paměti počítače $10^6 \times 10^6 \times 10^6$ bitů = 125 000 TB (pokud uvažujeme, že se částice na místě buďto nachází, anebo ne). Toto je samozřejmě krajní případ bez jakékoliv optimalizace a komprese, ale pro ilustraci postačuje. Tento případ by bylo možné výrazně optimalizovat např. pomocí metod oktalových stromů, které takto jemné částice zobrazují jen v místech, kde to simulace vyžaduje a na ostatních místech zobrazují násobky velikosti částic [18].

Vhodným měřítkem aproximace objemu zmenšíme paměťovou i výpočetní náročnost simulace. Tuto konstantu budu dále nazývat velikost voxelu. Velikost voxelu může být i 100cm a simulace bude stále vypadat poměrně realisticky. Toho je docíleno tzv. adaptační funkcí (více v kapitole Polygonizace). Aby ovšem

adaptační funkce fungovala, nestačí voxel v paměti reprezentovat pouze jedním bitem, ale je potřeba jej reprezentovat číslem s plovoucí řádovou čárkou. I přesto je nyní paměťová náročnost přijatelná: $1km \times 1km \times 1km$ terénu při voxelové velikosti 100cm a velikosti 4B na voxel zabírá terén 4GB místa v paměti. Případnou kompresí a optimalizací lze toto číslo ještě snížit.

Protože simulace terén generuje a zobrazuje dynamicky za běhu aplikace, je nezbytné terén rozdělit na menší výseky stejné velikosti zvané chunky. Toto rozdělení umožňuje efektivně generovat a zobrazovat terén jen v místech světa, kde je to potřeba (obvykle v okolí kamery uživatele). Z praktických důvodů jsem zvolil tvar chunků jako kvádry se čvercovou základnou a výškou několikanásobně větší než je velikost základny. Simulace totiž nevyžaduje skládat chunky směrem do výšky, ale stačí terén skládat v horizontální rovině. Má simulace je tedy limitována minimální a maximální výškou terénu, zatímco rozloha terénu je teoreticky nekonečná. Bylo by ale možné udělat i variantu s nekonečnou výškou. Typickou vlastností chunku je jeho rozlišení udávané v počtu voxelů ve třech prostorových osách a velikost voxelu v jednotkách vzdálenosti. Tyto konstanty jsou v kódu nastavitelné.

3.1 Voxelová reprezentace

3.1.1 Definice voxelu

Voxel je složenina anglických slov volumetric, česky objemový, a element, neboli prvek. Označuje částici objemu představující hodnotu v pravidelné mřížce tří-dimenzionálního prostoru. Jak již bylo zmíněno v úvodu, voxel je prostorovou analogií ke grafickému pojmu pixel. Stejně jako pixel, neuchovává si informaci o své poloze, ale pouze svou hodnotu. Tato hodnota bývá obvykle skalár nebo vektor. Poloha je dána souřadnicemi voxelu v datové struktuře (např. třírozměrném poli) [9].

Voxel si můžeme představit také jako krychli, jejíž střed obsahuje hodnotu. Střed krychle je místem, kde byl odebrán vzorek objemu, a na tomto místě tedy obsahuje přesnou hodnotu. Čím více se ovšem blížíme k okrajům této krychle, tím nepřesnější hodnotu dostáváme (celý voxel je reprezentovám hodnotou jeho středu). Tento problém lze minimalizovat pomocí polynomiální interpolace jako např. trilineární interpolace nebo trikubická interpolace s okolními voxely.

Voxel si také může uchovávat další užitečné hodnoty pro renderování, jako normálový vektor k povrchu nebo barvu.

3.1.2 Voxelová mřížka

Voxelová mřížka je část 3D prostoru, která aproximuje objemové vlastnosti tohoto prostoru pomocí vzorků (voxelů). Tato mřížka je pravidelná, definují ji vlastnosti jako rozlišení voxelové mřížky a její rozměry. Rozlišení uvádníme v počtu voxelů ve třech souřadnicových osách, např. rozlišení 32x32x128 voxelů. Rozměry

voxelové mřížky se pak vypočítají jako její rozlišení krát velikost voxelu. Tyto rozměry uvádíme v jednotkách vzdálenosti, např. 16x16x64 metrů.

Platí pravidlo, že čím větší stanovíme velikost voxelu, tím nepřesnější aproximace objemu bude. V případě simulace terénu jsem vypozoroval přijatelnou hodnotu velikosti voxelu v rozmezí 10 – 100 cm. Pokud byla velikost menší než 10 cm, terén byl velmi detailní, ale za cenu paměťové i výpočetní náročnosti. Naopak pokud byla velikost větší než 100 cm, terén už byl hrubý a viditelně hranatý. Velikost voxelu se stanovuje v interních jednotkách UE4 zvané unreal units, nicméně platí převod, že 1 unreal unit = 1 cm.

3.2 Polygonová reprezentace

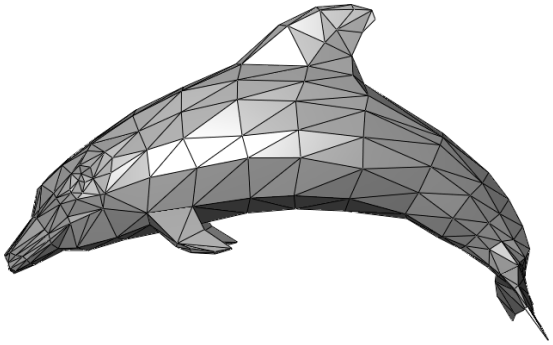
Polygonová grafika je založená na aproximaci povrchu objektů pomocí mnohoúhelníků (polygonů). Skupině vzájemně propojených mnohoúhelníků, které dohromady reprezentují model, říkáme polygonová síť. Naprostá většina současných herních engine pracuje výhradně a pouze s touto reprezentací z důvodu efektivních algoritmů nad polygony. Nejběžnější bývá reprezentace pomocí trojúhelníků. Pokud se model skládá z vyšších mnohoúhelníků, často bývá převeden na trojúhelníky [13].

Tak tomu je i v případě UE4, který kromě importu polygonových modelů do editoru, navíc obsahuje modul pro konstrukci polygonové síťoviny za běhu programu. Tento modul se nazývá Procedural Mesh Component [7] a lze jej povolit v nastavení editoru Plugins. V mojí aplikaci jsem využil modul třetí strany Realtime Mesh Component [4], který funguje na podobném principu, ale navíc obsahuje optimalizace. Součástí modulu je funkce, jejíž vstupní parametry jsou pole vrcholů (bodů ve 3D) a pole indexů těchto vrcholů, kde každé tři po sobě jdoucí indexy tvoří trojúhelník.

Například čtverec ve 3D pomocí této funkce definujeme následovně:

```
vertices = [(1, 1, 0), (1, -1, 0), (-1, -1, 0), (-1, 1, 0)]
triangles = [0, 1, 2, 0, 2, 3]
```

K polygonové reprezentaci se váží ještě doplňující informace potřebné pro správný výpočet osvětlení a vykreslování materiálu. Pro výpočet osvětlení jsou důležité normálové vektory vrcholů. Výše zmíněný modul obsahuje funkci, která normálové vektory vypočítá a přiřadí je k jejich vrcholům. Pro správné mapování textury na síťovinu se používá tzv. UV mapování. UV mapování jsem nicméně vůbec nepoužil, protože využívám jinou techniku zvanou triplanární projekce, viz. sekce 7.2.2.



Obrázek 1: Delfín vymodelovaný pomocí polygonů [11]

3.3 Vnější a vniřní struktura

Z pohledu implementace má každý chunk terénu vnitřní (voxelovou) a vnější (viditelnou) strukturu. Vnitřní struktura - voxelová mřížka je definovaná jako multidimenzionální statické pole čísel s plovoucí řádovou čárkou. Protože pracujeme ve 3D, toto pole má dimenzi třetího řádu. Díky tomu může algoritmus pracují s tímto polem přistupovat k voxelu na souřadnících $[x, y, z]$ v konstantním čase, a být tak efektivní.

Z hlediska vnější struktury, tedy viditelné, se jedná o polygonovou síť, kterou algoritmus vygeneroval z informací vnitřní voxelové struktury. UE4 neumí vykreslovat jinou než polygonovou reprezentaci, proto je tento převod nezbytný. Vnější reprezentace obsahuje dynamické pole vektorů se třemi složkami definující vrcholy polygonové sítě, dále dynamické pole indexů těchto vrcholů definující trojúhelníky a další podpůrné informace pro vykreslování jako normálové vektory vrcholů nebo informace o kolizích. Každý chunk si pak nese informaci o aplikovaném materiálu, který v kombinaci s dynamickým osvětlením dodává polygonové reprezentaci realistický vzhled.

3.4 Skládání terénu

Jelikož terén není generován jako jeden kus, ale rozdělen na chunky, ze kterých se skládá, implementace simulace se stává složitější. Souvislost terénu je jen zdánlivá a algoritmus musí řešit načítání a návaznost sousedních chunků tak, aby přechod mezi nimi nebyl viditelný.

Z hlediska generování voxelové struktury je toto zajištěno jednoduše - jsou použity metody grafické šumu, které jsou už ve své podstatě kontinuální. Mírně komplikovanější situace nastává při generování polygonové sítě - algoritmus potřebuje znát i informace o voxelech sousedních chunků. Z tohoto důvodu, pokud chceme načíst a vykreslit určitý chunk terénu, musíme nejprve vygenerovat voxely sousedních chunků. Funkci poté musíme předat sousední chunky zároveň s chunkem pro načtení.

Dále je ještě potřeba implementovat funkci, která vrátí hodnotu voxelu podle jeho světových souřadnic. Tato funkce musí nejprve určit, ve kterém chunku se daný voxel nachází, a poté ještě, na jakých lokálních souřadnicích voxelové mřížky je umístěn. Při použití této funkce už skládání terénu není problém, protože máme vygenerovanou voxelovou strukturu okolních chunků a můžeme zjistit i informace o okolí načítaného chunku.

4 Generování terénu

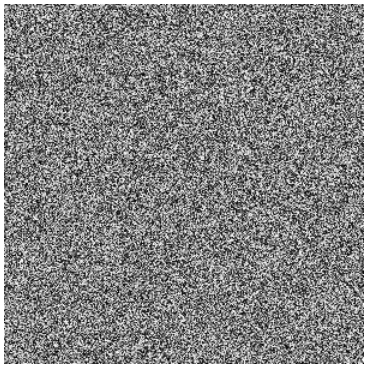
Předtím než můžeme terén převést na polygonovou strukturu a vyrenderovat v hlavním vlákně engine, musíme nejprve vygenerovat jeho objemovou strukturu. Generování terénu probíhá na úrovni voxelů. Každý voxel si nese informaci o tom, jak hluboko v terénu nebo jak daleko mimo terén se nachází. Platí pravidlo, že voxely ležící přesně na povrchu terénu mají hodnotu 0. Voxely nacházející se uvnitř terénu (pod povrchem) mají kladnou hodnotu a tato hodnota je tím větší, čím hlouběji uvnitř terénu se nacházejí. Naopak voxely ležící mimo objem terénu mají zápornou hodnotu, která je tím menší, čím dále se nacházejí od nejbližšího bodu povrchu terénu. Tyto skalární hodnoty uložené ve voxelech nám pomáhají k realistické aproximaci povrchu terénu.

O samotné generování terénu se stará samostatná třída generátor terénu. Tato třída nemusí znát informace o chuncích ani další informace o terénu, jako například materiál nebo textury. Stačí, když tato třída poskytuje funkce pro získání hodnoty voxelu podle jeho světových souřadnic v prostoru. Rozdělení voxelů do chunků a skládání terénu již řeší ostatní třídy terénu.

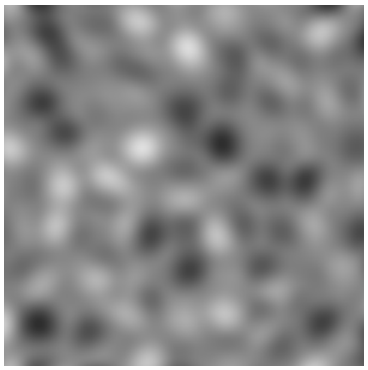
Aby byl terén souvislý, musíme se postarat o návaznost funkce generující voxely. Nejjednodušším, a také mnou zvoleným řešením, je použití fraktálů a metod grafického šumu, které jsou již ve své podstatě spojité. K tomuto účelu jsem využil knihovnu třetí strany, nazvanu Accidental noise. Nejprve obecně popíši metody grafického šumu, fraktály a poté samotnou knihovnu a její implementaci v generátoru terénu. Zmíním se také o parametrech generátoru upravujících vzhled terénu.

4.1 Metody grafického šumu

Grafický šum je posloupnost náhodných čísel, které mohou být vygenerované například algoritmem generátoru náhodných čísel. Výchozí grafický šum je nespojitý, byly ale vyvinuty techniky pro generování spojitého šumu. Jedním z takových spojitých šumů je algoritmus Perlin noise (Perlinův šum). Ten využívá interpolaci mezi sousedními hodnotami, a tak získává vyhlazenou spojitou funkci. Perlinův šum může být generován v různé dimenzi [12]. Příklad nespojitého a Perlinova šumu ve 2D:



Obrázek 2: Nespojitý grafický šum



Obrázek 3: Perlinův šum ve 2D

Funkce spojitého šumu mohou být využity jako tzv. výšková mapa terénu. Výšková mapa funguje na principu, kdy terén nastavuje svou výškovou souřadnici z, bodu (x, y, z) podle hodnoty pixelu na souřadnicích (x, y) výškové textury. Výškové mapy bývají většinou šedotónové, protože stačí jeden kanál udávající výšku terénu na jeho souřadnicích. Maximální hodnota pixelu (bílá) se mapuje na maximální amplitudu terénu, minimální hodnota (černá) se mapuje na minimální amplitudu terénu.

Ve hrách a simulacích terénu se využívají buďto předgenerované výškové textury, které slouží pro předchystání statických terénů, anebo se do programu implementuje přímo funkce spojitého grafické šumu, a výška terénu se tak může zjišťovat za běhu programu voláním této funkce. Druhý zmíněný případ jsem při návrhu programu využil i já, jelikož celý můj terén se generuje dynamicky.

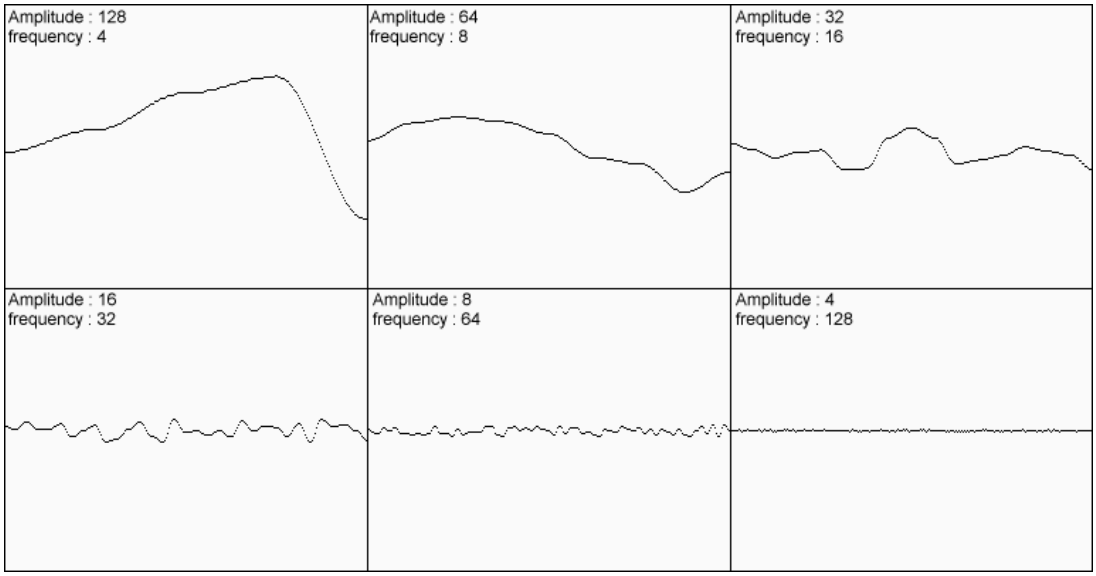
Perlinův šum nicméně není úplně přirozená funkce, které by přesně vystihovala hornatost terénu tak, jak ji můžeme vidět běžně v přírodě. Ken Perlin, autor původního algoritmu, vyvinul v roce 2001 vylepšení tohoto algoritmu, jež nazval Simplex noise. Simplex noise má oproti předchozímu algoritmu několik výhod: je efektivnější, má menší časovou složitost a řeší nechtěně artefakty, které v algoritmu nastávaly.

Ani Simplex noise ale stále neposkytuje dostatečně realistický vzhled terénu. Ten se mi podařilo docílit až s využitím fraktálů, které popíši v následující kapitole.

4.2 Využití fraktálů

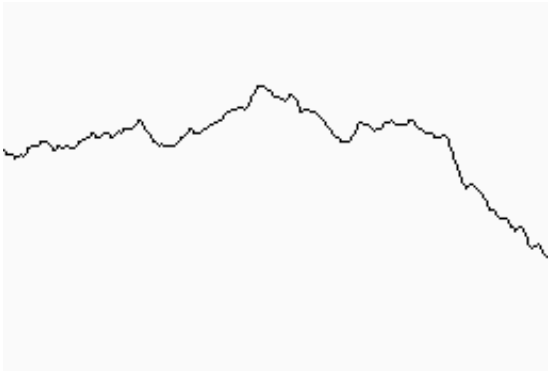
Fraktál poprvé definoval Benoît Mandelbrot roku 1975. Můžeme jej neformálně definovat jako geometrický objekt, který vykazuje soběpodobnost bez ohledu na přiblížení, kterým se na něj díváme. Fraktály mívají na první pohled velmi složitý tvar, který je ale generován opakovaným použitím jednoduchých pravidel. Typicky fraktály používáme pro počítačové modelování stromů, mraků, sněhových vloček, řek nebo hor.

Poměrně realistického vzhledu terénu lze docílit kombinací fraktálů a metod grafického šumu. Rekurzivní opakování funkcí grafického šumu se zvyšující se frekvencí a snižující se amplitudou umožňuje makroskopické i mikroskopické zvrásnění terénu. Počet rekurzivních opakování funkce se u fraktálů nazývá počet oktáv. Každá oktáva je specifická svou frekvencí a amplitudou.



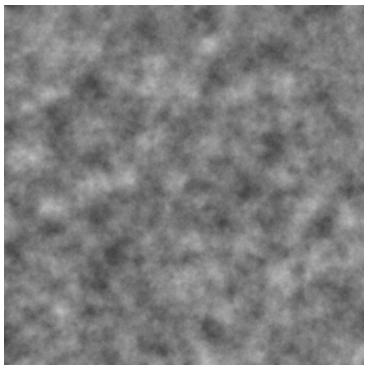
Obrázek 4: Šest oktáv spojitého grafického šumu [1]

Poté, co máme takto vygenerované oktávy, je všechny sečteme a tím získáme výsledný fraktálový šum:



Obrázek 5: Výsledný součet všech šesti oktáv [1]

Využití fraktálů při konstrukci terénu vychází ze skutečnosti, že i reálný terén vykazuje známky sebepodobnosti. Menší části terénu vypadají podobně jako větší části, a naopak. Díky této skutečnosti lze rekurzivním opakováním fraktálové funkce vytvořit simulaci terénu, která se velmi podobá reálnému terénu. Skutečnost, že reálný terén vykazuje znaky fraktálů, byla předmětem mnoha studií [19].



Obrázek 6: Fraktálový šum ve 2D

4.3 Knihovna Accidental noise

Na internetu existuje velké množství knihoven implementujících grafické šumové funkce. Abych tyto funkce nemusel programovat od základu, rozhodl jsem se jednu z těchto knihoven využít. Tato knihovna se jmenuje Accidental noise library [21] a implementuje různé druhy šumů v druhé, třetí, čtvrté a šesté dimenzi. Navíc poskytuje rozhraní pro řetězení těchto funkcí do jedné výsledné funkce. Knihovna má otevřené zdrojové kódy a je napsaná v jazyce C++, není tedy žádný problém ji do UE4 projektu přidat.

Knihovna obsahuje přímo funkci pro získání fraktálového šumu. Tato funkce je polymorfní, šum generuje v té dimenzi, kolik parametrů jí předáváme. Pokud

chceme generovat výškovou mapu terénu, předáváme jí pouze parametry (x, y) a funkce nám vrací skalár značící výšku v tomto bodě terénu.

Objekt implementující fraktálovou funkci má dále nastavitelné parametry, upravující vlastnosti šumu. Jedná se o frekvenci, amplitudu, počet oktáv, použitý algoritmus a funkci použitou pro interpolaci. V mém případě jsem parametry frekvence, amplitudy a počtu oktáv nechal nastavitelné uživatelem aplikace, zatímco jako použitý algoritmus jsem zvolil algoritmus Fractional Brownian motion (frakcionální Brownův pohyb). Interpolaci jsem zvolil kvintickou.

Takto nastavený objekt, implementující fraktálový šum, můžu následně využít k samotnému generování voxelové struktury terénu.

4.4 Algoritmus

Formálně můžeme matematický předpis generující strukturu terénu definovat jakožto zobrazení:

$$f\colon \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \to \mathbb{R}.$$

Množina $\mathbb{Z}$ označuje množinu všech celých čísel, množina $\mathbb{R}$ označuje množinu všech reálných čísel. Toto zobrazení přiřazuje každému diskrétnímu bodu 3D prostoru (voxelu) skalární hodnotu v závislosti na poloze vůči terénu (princip vysvětlen v úvodu kapitoly). Zobrazení by ale stejně tak mohlo být definováno jako:

$$f\colon \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \to \mathbb{R} \times A.$$

Takové zobrazení by každému diskrétnímu bodu prostoru navíc mohlo ještě přiřazovat určitý materiál z množiny materiálů A, kde A by mohlo být například $A = \{hlína, kámen, písek\}$. Materiály terénu jsem nicméně v aplikaci nevyužil.

Z hlediska implementace algoritmus závisí na funkci fraktálového šumu *noise* z knihovny Accidental noise. Samotný kód algoritmu je navržen takto:

Algoritmus 1 Algoritmus generování terénu

```
1:  procedure GENERATE(x1, y1, x2, y2, h)
2:      DECLARE VoxelGrid[x2 − x1 + 1][y2 − y1 + 1][h + 1]
3:      for x ← x1 to x2 do
4:          for y ← y1 to y2 do
5:              height ← h · amplitude · noise(x, y)
6:              for z ← 0 to h do
7:                  VoxelGrid[x][y][z] ← (height − z)
8:              end for
9:          end for
10:     end for
11:     return VoxelGrid
12: end procedure
```

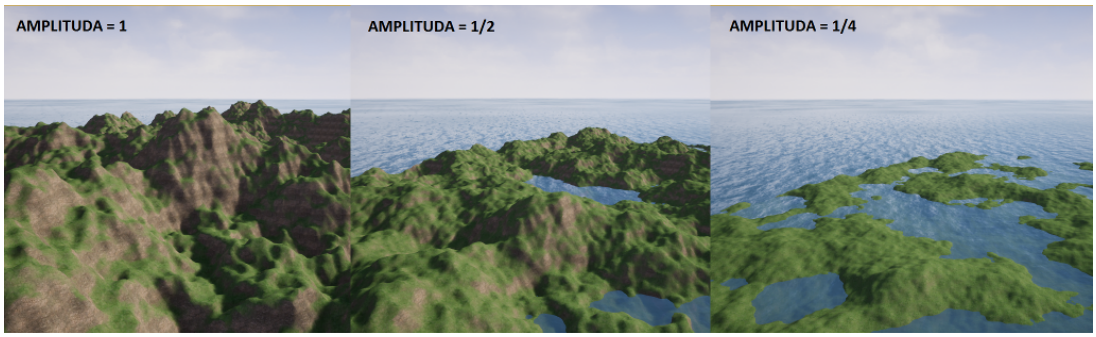
Parametry x_1, y_1 jsou počáteční světové souřadnice, od kterých začínáme generovat, souřadnice x_2, y_2 jsou koncové světové souřadnice, kde generování končí. Parametr h je maximální výška terénu. $VoxelGrid$ je 3D voxelová mřížka skalárních hodnot, kam se ukládají hodnoty voxelů a kterou funkce na konci vrací. Pro každou polohu (x, y) se nejprve vypočte výška *height* terénu, kterou je potřeba vynásobit maximální výškou terénu h a amplitudou terénu *amplitude*, jelikož funkce *noise* vrací hodnoty v intervalu $[0, 1]$. Třetí zanořený cyklus poté generuje sloupec terénu až do maximální výšky terénu. Pokud je souřadnice z pod touto výškou, hodnota voxelů je kladná, pokud je nad výškou terénu, hodnota voxelů je záporná. Ale jak jsem zmínil již v úvodu, pro následnou polygonizaci je důležitá i velikost tohoto skaláru.

Jednotlivé chunky terénu pak můžou tuto funkci volat a získat tak voxelovou mřížku definující jejich objemovou strukturu. Díky faktu, že funkce *noise* je spojitá, pak chunky terénu na sebe budou automaticky navazovat, i když je generujeme každý zvlášť. Tímto postupem jsme tedy docílili spojitého terénu, složeného z navazujících kusů - chunků.

4.5 Parametry generátoru

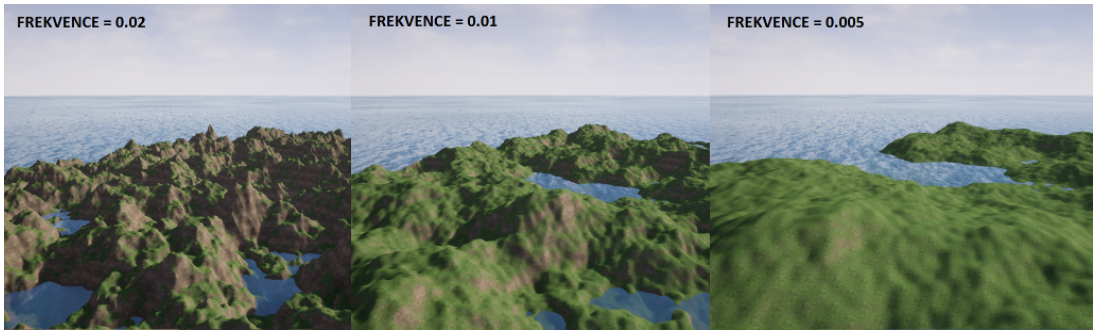
Uživatel aplikace může z uživatelského rozhraní ovlivňovat parametry generátoru, a tedy i výsledný vzhled terénu. V případě, že se některý z nich změní, je potřeba celý terén (všechny načtené chunky) znovu přegenerovat podle nových vlastností terénu. Mezi nastavitelné parametry generátoru patří amplituda, frekvence a počet oktáv.

Amplituda ovlivňuje maximální výšku terénu, nesmí být větší než jedna, jinak by terén byl vertikálně useknutý (byl by mimo rozsah voxelů).



Obrázek 7: Změna amplitudy

Frekvence ovlivňuje, jak rychle se mění hornanost terénu. Velká frekvence značí velké množství malých kopců, malá naopak velmi velké kopce.



Obrázek 8: Změna frekvence

Počet oktáv určuje počet rekurzivních opakování fraktálového šumu. Malý počet oktáv redukuje zvrásnění terénu pouze na makroskopické, větší počet oktáv generuje i mikroskopické zvrásnění.

Obrázek 9: Změna počtu oktáv

Všechny tři parametry lze libovolně kombinovat a vytvářet tak různé zajímavé vzhledy terénů.

5 Polygonizace

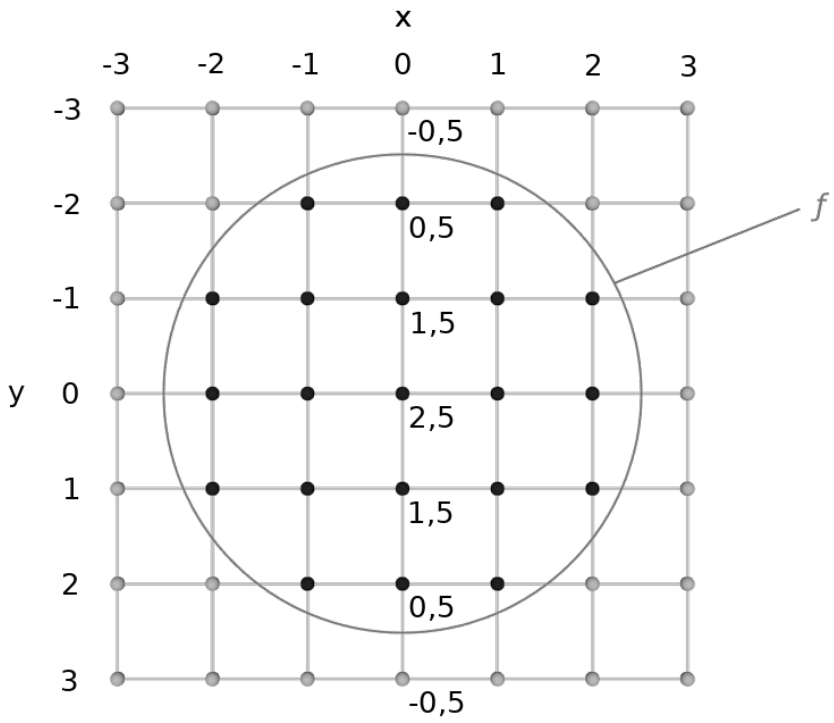
Voxelová reprezentace terénu zachycuje jeho objem. Taková data není jednoduché vykreslit v rámci 3D simulace. Existují sice přímo voxelové engine, které umí vykreslit voxelová data, ale cílem této práce bylo zpřístupnit voxelový terén populárním engine, které žádnou přímou podporu pro voxelové renderování nemají. Z tohoto důvodu je nutný převod voxelové reprezentace na polygonovu, zvaný polygonizace.

Pro převod voxelů na polygony existuje více algoritmů lišících se svým přístupem, výsledkem i efektivitou. Mezi dva nejznámější algoritmy patří algoritmus zvaný Marching cubes a algorimus zvaný Dual Contouring. V mé práci jsem využil první zmíněný algoritmus, zaměřím se tedy především na něj. Předtím ale

ještě popíši jeho 2D variantu zvanou Marching squares, která nám pomůžeme fungování algoritmu Marching cubes lépe pochopit.

5.1 Algoritmus Marching squares

Algoritmus Marching squares (česky můžeme volně přeložit jako Pochodující čtverce) slouží k tvorbě obrysů z 2D skalárního pole. Používáme jej v situaci, kdy 2D skalární pole aproximuje obsah jednoho či více 2D objektů a hledáme obrysy těchto objektů [2]. Příklad 2D skalárního pole aproximující obsah kruhu:



$$f(x, y) = 2.5 - \sqrt{x^2 + y^2}$$

Obrázek 10: Vizualizace skalárního pole aproximující obsah kruhu

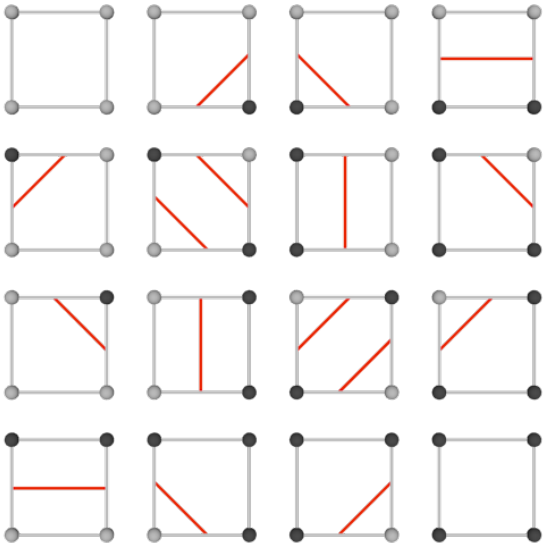
Hodnoty skalárního pole můžou být určeny nějakým zobrazením:

$$f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{R}.$$

Množina $\mathbb{Z}$ označuje množinu všech celých čísel, množina $\mathbb{R}$ označuje množinu všech reálných čísel. Zobrazení f jednotlivým diskrétním bodům skalárního pole přiřazuje kladné hodnoty, pokud leží v obsahu tvaru, a záporné hodnoty pokud leží mimo tvar objektu. Čím blíže středu objektu bod leží, tím větší hodnotu bude mít, naopak čím dále se od objektu nachází, tím menší hodnotu bude mít.

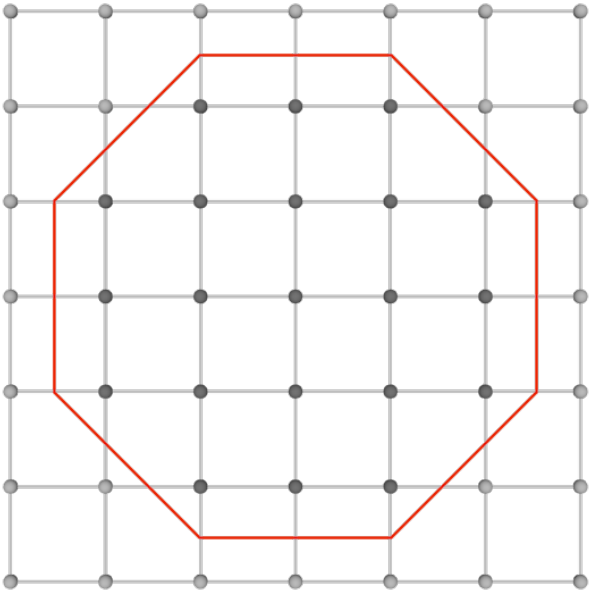
Popis algoritmu:

1. Prostor si rozdělíme na stejné buňky čtvercového tvaru. Každá buňká má čtyři vrcholy určující jejich hodnotu ve 2D skalárním poli.
2. V cyklu procházíme každou buňku zvlášť. Pro každý vrchol buňky zjišťujeme, zda leží uvnitř, anebo mimo obsah objektu.
3. V případě, že jeden vrchol buňky leží uvnitř objektu a druhý nikoliv, bude se na hraně mezi nimi nacházet bod obrysu.
4. Body obrysů spojíme podle tabulky kombinací (viz. obrázek 11)



Obrázek 11: Tabulka možných kombinací obrysů

Všechny získané úsečky poté spojíme a vykreslíme výsledek. I když procházíme každou buňku zvlášť, algoritmus vygeneruje souvislý obrys tvaru:



Obrázek 12: Výsledný obrys tvaru

Výsledek příliš nevypadá jako původní tvar definovaný zobrazením f. Tato odchylka je způsobená faktem, že každý bod obrysu leží přesně uprostřed mezi vrcholy buněk. Pokud chceme získat přesnější aproximaci obrysu, musíme ještě určit polohu bodu mezi vrcholy. K tomuto účelu slouží adaptační funkce.

5.2 Adaptace

Znaménko hodnoty ve 2D skalárním poli nám určuje, zda tento bod leží uvnitř útvaru, anebo mimo něj. Hodnota samotná nám však zároveň říká, jak hluboko uvnitř nebo daleko mimo něj se bod nachází. Pokud je hodnota velké kladné číslo, víme, že se bod nachází hluboko uvnitř útvaru. Pokud je hodnota naopak velmi malé záporné číslo, víme, že se bod nachází daleko mimo útvar. Tohoto faktu můžeme využít k přesnějšímu určení obrysu útvaru.

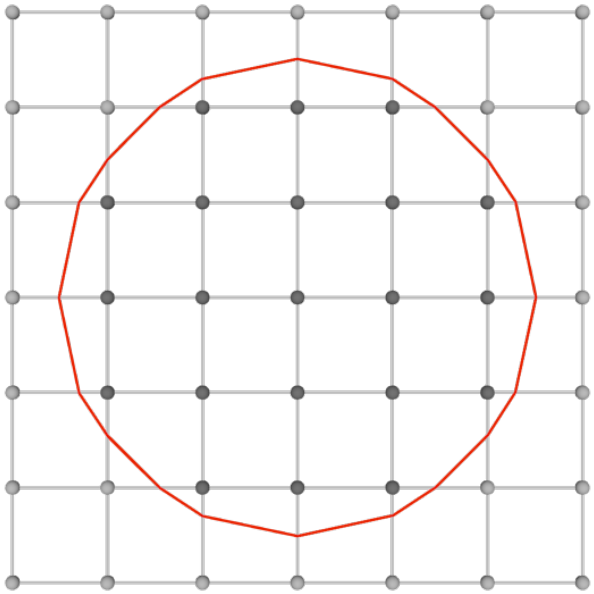
K určení přibližené polohy bodu mezi dvěma vrcholy můžeme použít lineární interpolaci. Lineární interpolace se používá k určení přibližné hodnoty funkce v případě, že známe jiné dvě hodnoty této funkce a hledaná hodnota leží mezi těmito dvěma body. Tyto podmínky náš případ splňuje. Koeficient k můžeme nalézt rovnicí:

$$k = \frac{v_0}{v_1 - v_0},$$

kde v_0, v_1 jsou hodnoty vrcholů a $v_0 \neq v_1$. Koeficient k nám určuje kolikrát je námi hledaný bod vzdálen od vrcholu v_0. V případě $k = 0$ je hledaný bod totižný

s vrcholem v_0, pro $k = 1$ je totožný s vrcholem v_1. Pokud $k \in (0, 1)$, leží hledaný bod na úsečce mezi vrcholy v_0 a v_1.

S využitím adaptační funkce nyní získáváme daleko přesnější výsledek:

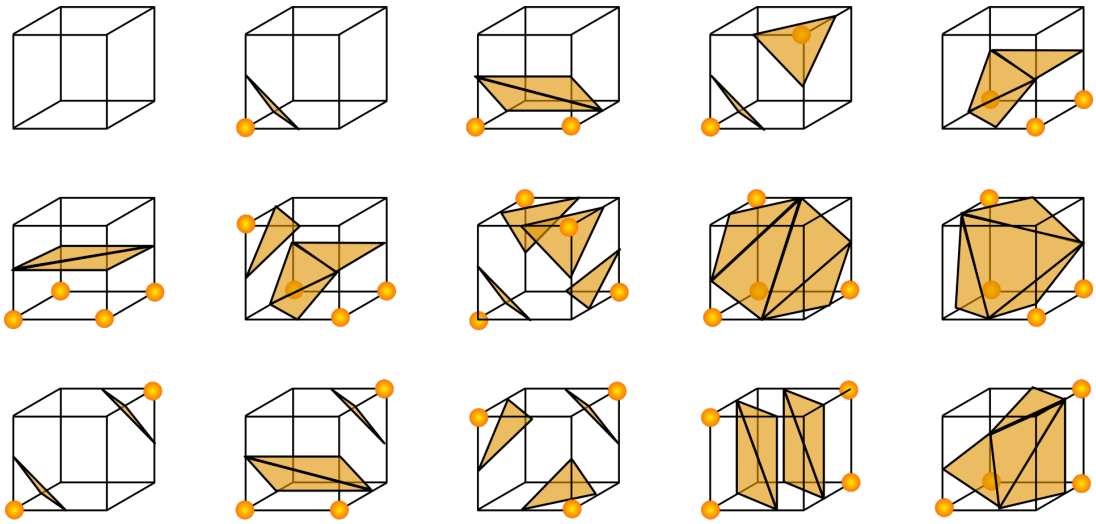


Obrázek 13: Výsledný obrys tvaru s adaptační funkcí

5.3 Algoritmus Marching cubes

Algoritmus Marching cubes (česky můžeme volně přeložit jako Pochodující kostky) je 3D varianta algoritmu Marching squares. Slouží k extrakci polygonové síťoviny ze 3D skalárního pole (voxelové mřížky). Tato metoda byla poprvé představena na konferenci SIGGRAPH v roce 1987 [16]. Od té doby prošla několika úpravami a vylepšeními, řešící možné problematické situace, které v algoritmu můžou nastat. Metoda se často používá k vizualizaci medicínských dat získaných pomocí CT nebo MRI.

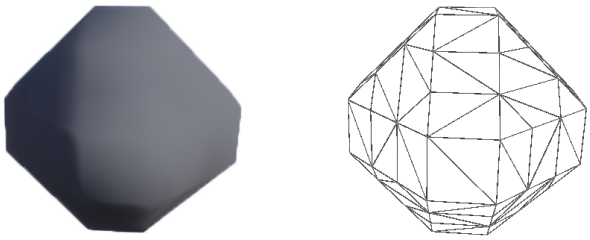
Zatímco u algoritmu Marching squares jsme procházeli pole po čtvercích stejné velikosti, ve 3D prostoru budeme pole procházet pomocí krychlí. V případě, že všechny vrcholy krychle leží uvnitř objemu, nebo mimo objem voxelů, neprochází touto krychlí žádný polygon povrchu objektu. V opačném případě krychlí prochází jeden či více polygonů, které chceme vygenerovat. V závislosti na vrcholech, které jsou uvnitř nebo vně objektu, můžou nastat různé kombinace průchodů krychle polygony. Ve 2D (čtverec má 4 vrcholy) těchto kombinací bylo $2^4 = 16$. Krychle má ovšem 8 vrcholů a ve 3D tedy těchto možných kombinací máme celkem $2^8 = 256$. Většina kombinací je ovšem pouze rotace nebo zrcadlení (nebo oboje) jiných případů. Pokud odstraníme redundantní případy, získáme minimální tabulku 15 možných případů:



Obrázek 14: Minimální tabulka kombinací algoritmu Marching cubes [15]

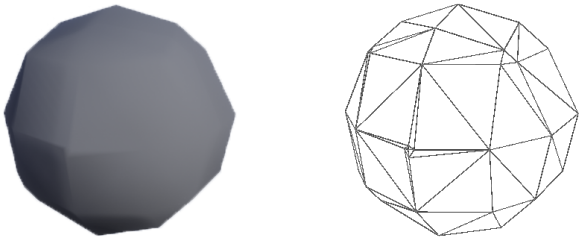
Musíme ale poznamenat, že z důvodu optimalizace výkonu algoritmu se tyto transformace v kódu neprovádějí, ale je zde staticky předdefinováno všech 256 možných kombinací ve statickém poli s konstantním časovým přístupem. Tímto postupem se algoritmus značně zrychlí.

Výsledek algoritmu, aproximující kouli, bez použití adaptační funkce:



Obrázek 15: Rekonstrukce koule pomocí Marching cubes bez adaptační funkce

Stejně jako v případě Marching squares můžeme použít adaptační funkci, která pomocí lineární interpolace určí přibližnou polohu hledaného bodu mezi vrcholy krychle. Výsledek algoritmu s použitím adaptační funkce:



Obrázek 16: Rekonstrukce koule pomocí Marching cubes s adaptační funkcí

Mnou použitá implementace algoritmu Marching cubes pochází z roku 1994 od Paula Bourkeho [3].

6 Nástroje editování terénu

Voxelová struktura terénu nabízí možnost editovat jeho objem. K tomuto účelu slouží tzv. prostorové štětce. Jedná se o 3D geometrické útvary, které upravují voxely ve svém okolí podle matematické funkce, jež je definuje. Prostorové štětce mohou voxely buďto přidávat, odebírat, nebo upravovat.

6.1 Prostorové štětce

Prostorové geometrické útvary bývají obvykle definované pomocí matematické funkce, která nabývá hodnoty 0 právě při povrchu útvaru. Tohoto faktu můžeme přímo využít při editaci voxelů. Pokud chceme voxely štětcem přidávat, funkce musí přiřazovat kladné hodnoty uvnitř útvaru a záporné hodnoty mimo útvar. Naopak pokud chceme štětcem voxely odebírat, funkce musí uvnitř útvaru přiřazovat záporné hodnoty a kladné mimo útvar. Dále při přidávání voxelů platí pravidlo, že do voxelu štětcem zapíšeme hodnotu jen v případě, že je tato hodnota větší než je aktuální hodnota voxelu. Tímto pravidlem zabráníme přepsání voxelů, které jsou k povrchu terénu blíže než voxely přidávané štětcem. Naopak u odebírání voxelů platí pravidlo, že do voxelu zapíšeme hodnotu jen v případě, že je tato hodnota menší než je aktuální hodnota voxelu.

Protože jednotlivé štětce mohou pozměnit voxely nacházející se na více než jednom chunku terénu, je po každé úpravě terénu štětcem potřeba zjistit, které chunky byly úpravou ovlivněny. Nad těmito chunky je posléze potřeba provést opětovnou polygonizaci a jejich pozměněnou polygonovou strukturu aktualizovat ve vykreslovacím vlákně engine, aby se změny v aplikaci viditelně projevily.

Do projektu jsem implementoval tři základní štětce upravující terén. Jedná se o štětce ve tvaru krychle a koule, které umožňují přidávat a odebírat voxely,

a o štětec ve tvaru koule, který umožňuje vyhlazování voxelového terénu. Všem štětcům lze měnit velikost a tedy celkový objem terénu, který ovlivňují. Tyto štětce nyní stručně popíši.

6.1.1 Koule

Matematicky můžeme kouli v 3D prostoru definovat rovnicí:

$$(x-k)^2 + (y-m)^2 + (z\ -n)^2 = r^2,$$

kde k, m, n jsou prostorové souřadnice středu koule a r je její poloměr. Z této rovnice můžeme odvodit funkci f prostorového štětce:

$$f_1(x,y,z) = (x-k)^2 + (y-m)^2 + (z\ -n)^2 - r^2.$$

Tato funkce nabývá záporných hodnot u středu koule, nulových při povrchu koule a kladných mimo kouli. Právě proto je vhodná pro odebírání voxelů. Pro přidávání voxelů musíme použít funkci $f_2(x,y,z) = -f_1(x,y,z)$, která je naopak uvnitř koule kladná a mimo kouli záporná.

6.1.2 Krychle

Definovat krychli v 3D prostoru matematickým předpisem není tak jednoduché, proto jsem pro implementaci tohoto štětce použil sérii podmíněných výrazů. V ní rozlišuji polohu voxelu vůči krychli na třech souřadnicových osách. Při odebírání voxelů nastavuji voxelům uvnitř krychle záporné hodnoty, voxelům na povrchu krychle nulouvou hodnotu a voxelům mimo krychli kladné hodnoty. U přidávání voxelů je to analogicky jako u koule naopak. U krychle není potřeba gradientní funkce, protože krychle má ostré hrany a nepotřebuje přesné hodnoty pro adaptaci.

6.1.3 Vyhlazování terénu

Třetím štětcem je štětec pro vyhlazování terénu, který má tvar koule. Tento štětec je 3D analogie grafického algoritmu konvoluce [23] známého z obrázkových filtrů. Štětec si nejprve dočasně uloží všechny voxely, které ovlivní. Poté je v cyklu prochází a hodnotu každého voxelu průměruje s okolními voxely. Dalo by se tedy říci, že se jedná o 3D analogii konvoluce s 3D jednotkovou maticí, tedy jakési "rozostření terénu". Pokud jsou někde v terénu ostré hrany nebo výběžky, tento štětec je pěkně zarovná a terén tak působí přirozeněji.

7 Textury a materiál terénu

Textury a materiál terénu hrají ve výsledném vzhledu důležitou roli. Terén s výchozím materiálem bez textur by byl pouze šedý a nikterak zajímavý. Proto

je pro něj potřeba vytvořit nový materiál, určující jak se na něj mají textury mapovat a jak má terén ve výsledku vypadat.

7.1 Textury

Základem kvalitního materiálu jsou kvalitní textury. Důležité je jejich rozlišení, světelná vyváženost a v případě terénu také bezešvost. Čím větší rozlišení textur zvolíme, tím kvalitněji a detailněji modely můžou vypadat. Větší textury nicméně zvyšují paměťovou náročnost na grafickou kartu. Platí pravidlo, že čím větší má model velikost, tím větší potřebujeme rozlišení textur. V případě terénu, který je teoreticky nekonečný, používáme techniku opakování textur. Právě z tohoto důvodu je důležitá bezešvost textur, aby při skládání textur nebyl vidět přechod mezi koncem a začátkem textury.

V materiálu jsem použil celkem pouze dvě textury a to texturu trávníku a texturu hlíny. Obě dvě textury jsou bezešvé a rozlišení mají 2048x2048 pixelů. Textura trávníku je určena k mapování na povrch terénu, textura hlíny na místa, kde je terén vidět z boční nebo spodní strany. Obě textury můžou jedna v druhou pozvolně přecházet.

7.2 Materiál

Materiál modelu určuje, jak bude výsledný model vypadat, když na něj dopadnou paprsky světla ze zdroje osvětlení. Může obsahovat textury, konstanty, barvy, ale také matematické operace a výpočty, určující jak má osvětlení interagovat s povrchem modelu. Materiál v UE4 se většinou sestavuje pomocí speciálního editoru pro vizuální skriptování [8].

7.2.1 Triplanární projekce

U běžných modelů se ve většině případů používá tzv. uv mapování. Touto technikou se každému 3D vrcholu modelu přiřadí u a v souřadnice na 2D textuře. Proto se tato technika nazývá uv mapování. Jelikož se terén generuje procedurálně za běhu programu, není jednoduché správně přiřadit uv souřadnice modelu tak, aby na něm textura vypadala hezky a nebyla nikde nepřirozeně natažená.

Z tohoto důvodu jsem využil techniku zvanou triplanární projekce [17]. Tato technika vůbec nepracuje s uv mapou. Triplanární projekce funguje na principu, kdy máme tři k sobě navzájem kolmé roviny. Každá rovina může mít svou vlastní texturu. Dále provádíme projekci každé roviny na model ve směru normálového vektoru roviny. Dle normálového vektoru každého 3D vrcholu modelu poté rozlišujeme, která z textur rovin bude na tomto vrcholu převažovat. Pokud například budeme mít vrchol s normálovým vektorem $\overrightarrow{n} = (1, 0, 0)$, bude na tomto vrcholu převládat textura roviny, která je kolmá k ose x. Stejný případ nastane u vektoru $\overrightarrow{n} = (-1, 0, 0)$. Pokud ovšem bude mít vrchol normálový vektor $\overrightarrow{n} = (0.7071, 0.7071, 0)$, dojde na tomto vrcholu k blendování (míchání) textur z roviny kolmé k ose x a textury z roviny kolmé k ose y v poměru 1:1.

Tato technika zajišťuje správné mapování textur na povrch terénu, podle sklonu terénu. Zároveň umožňuje mapovat texturu hlíny z boční a spodní strany terénu a texturu trávníku pouze na povrch terénu.

7.2.2 Blendování textur

Blendování (míchání) textur je zajištěno matematickou funkcí lineární interpolace. UE4 je přímo pro tento účel vybaven funkcí, jejíž vstup jsou dvě textury a konstanta nebo textura, podle které se má interpolace provádět. Triplanární projekce této funkce využívá a míchá textury do jedné výsledné podle normálového vektoru každého vrcholu. Tento postup zajišťuje pěkné pozvolné přechody mezi texturami mapovanými z různých stran.

8 Optimalizace

Optimalizace programu je důležitá, aby aplikace běžela plynule, bez zbytečného trhání obrazu a dlouhých prodlev při generování chunků. Samotný algoritmus Marching cubes již je velmi dobře optimalizován. Největším problémem aplikace ovšem bylo znatelné zamrznutí obrazu při pohybu uživatele, v případě, kdy se generovalo větší množství chunků. Než se totiž chunk může zobrazit, je nejprve potřeba vygenerovat jeho voxelouvou strukturu, provést polygonizaci, vypočítat normály vrcholů a nakonec zobrazit model v renderovacím vlákně engine. Z tohoto důvodu jsem problém vyřešit prováděním těchto výpočtů v jiném než hlavním herním vlákně. Tomuto pomocnému vláknu předám všechna potřebná data, vlákno provede zmíněné výpočty, a poté se synchronizuje s hlavním herním vláknem. Zobrazení samotného modelu je nicméně potřeba provést v hlavním herním vlákně, jinak by aplikace skončila fatální chybou. Tímto řešením je zajištěna plynulost aplikace po celou dobu používání a uživatelův zážitek nenaruší žádné větší prodlevy mezi snímky simulace. V budoucnu by bylo možné program ještě vylepšit o výpočty ve více paralelních vláknech, a tak využít vícejádrové procesory.

9 Uživatelská příručka

Přiložené DVD obsahuje složky VoxelTerrain32, VoxelTerrain64, VoxelPlugin a soubor `Instrukce.txt`. Složka VoxelTerrain32 obsahuje 32-bitovou verzi aplikace, která demonstruje výsledky této práce. Složka VoxelTerrain64 obsahuje 64-bitovou verzi. V souboru `Instrukce.txt` jsou popsány instrukce k aplikaci. Samotná aplikace ke svému běhu potřebuje knihovnu DirectX, verzi alespoň 10. Všechny prerekvizity by aplikace měla nainstalovat při svém prvotním spuštění (vyžaduje úroveň oprávnění administrátora).

Po spuštění aplikace se uživatel ocitá v menu, které obsahuje dva herní módy, informace o aplikaci a ukončení aplikace. Mód první osoby umožňuje procházení po terénu z pohledu první osoby, včetně kolizí a štětců pro editování terénu. Mód letadla umožňuje volný pohyb ve 3D prostoru a nastavení parametrů generátoru terénu.

Ovládání aplikace je následující:

`W, S, A, D` - pohyb v 3D prostoru
`Levé tlačítko myši` - přidávání voxelů štětcem
`Pravé tlačítko myši` - odebírání voxelů štětcem
`Kolečko myši` - změna prostorového štětce
`Levý shift` + kolečko myši - změna velikosti štětce
`Tlačítka +/- na numerické klávesnici` - změna velikosti štětce
`Esc` - návrat do menu

10 Programátorská příručka

Složka `VoxelPlugin` obsahuje UE4 projekt s modulem voxelového terénu, který tato práce implementovala, včetně zdrojových kódů a všech dalších souvisejících položek. Tento projekt je kompatibilní s verzí Unreal engine 4.22. Dále budu popisovat strukturu této složky.

Složka obsahuje soubor `VoxelPlugin.uprojekt`. Toto je samotný soubor s projektem spustitelným v Unreal engine. Složka Source obsahuje zdrojové kódy projektu, které implementují funkce aplikace, nikoli však samotný voxelový terén. Ten je navržen jako modul engine, proto jeho zdrojové kódy nalezneme ve složce Plugins.

Složka Plugins obsahuje dva moduly - modul RuntimeMeshComponent, který realizuje procedurální generování modelů, a modul VoxelTerrainPlugin. VoxelTerrainPlugin obsahuje samotnou implementaci této práce a závisí právě na zmíněném modulu RuntimeMeshComponent.

Složka VoxelTerrainPlugin obsahuje složku Source se zdrojovými kódy modulu. Nalezneme zde složky Noise - implementaci knihovny Accidental noise library, složku Private a Public. Složka Public zahrnuje hlavičkové soubory jednotlivých tříd, složka Private zahrnuje `.cpp` zdrojové kódy.

Zde je implemenováno několik základních tříd. Třída MarchingCubes implementuje algoritmus Marching Cubes pro polygonizaci voxelů. Třída TerrainChunk realizuje jednotlivé kusy terénu s jejich vlastnostmi a funkcemi. Třída TerrainGenerator dále implementuje funkce generování voxelové struktury a úpravu vlastností generátoru. A nakonec samotná hlavní třída VoxelTerrain všechny tyto třídy spojuje a realizuje terén samotný, včetně jeho načítání, skládání a štětců, které jej editují.

Závěr

Práce splnila zadaná očekávání. Aplikace je plně funkční, simulující realistický terén generovaný za běhu programu. Terén obsahuje voxelovou reprezentaci, umožnující terén libovolně editovat pomocí prostorových štětců. Program dále voxelovou reprezentaci převádí na polygonovou, čímž umožňuje terén renderovat pomocí standardních technik pro vykreslování polygonových modelů. Generátor terénu má nastavitelné parametry, kterými může uživatel ovlivňovat výsledný vzhled terénu. Terén je náležitě otexturován, umožňuje více textur a pozvolný přechod mezi nimi. Algoritmus je efektivní, zajišťuje plynulé načítání terénu v reálném čase podle pohybu uživatele. Program je dále koncipován jako modul herního engine, a tak umožňuje jeho jednoduché nasazení dalším případným uživatelům engine.

Conclusions

The assigned goals of the thesis were achieved. The application is fully working, simulating the realistic terrain generated in real time. The terrain includes voxel representation enabling free editing using 3D space brushes. The program also converts the voxel representation to polygonal one which enables rendering the terrain using standard techniques for polygonal models rendering. The terrain generator has settable parameters which can be used by the users to alter the final terrain look. The terrain is properly textured, it enables more textures and gradual switching between them. The algorithm is effective, it provides smooth loading of the terrain in real time according to the user's movement. The program is also designed as a plug-in of a gaming engine and thus enables its other potential engine users an easy set up.

A Obsah přiloženého CD/DVD

Přiložené CD/DVD obsahuje:

`VoxelTerrain32/`
: Složka obsahuje 32-bitovou verzi aplikace.

`VoxelTerrain64/`
: Složka obsahuje 64-bitovou verzi aplikace.

`VoxelPlugin/`
: Složka obsahuje samotný UE4 projekt s modulem a zdrojovými kódy.

`Instrukce.txt`
: Soubor obsahuje instrukce a požadavky aplikace.

Literatura

[1] BIAGIOLI, Adrian. Understanding Perlin Noise [online]. 2014 [cit. 2020-05-24] Dostupné z: https://adrianb.io/2014/08/09/perlinnoise.html

[2] BorisTheBrave.com. Marching Cubes Tutorial [online]. 2008 [cit. 2020-05-20]. Dostupné z: https://www.boristhebrave.com/2018/04/15/marching-cubes-tutorial/

[3] BOURKE, Paul. Polygonising a scalar field [online]. 1994 [cit. 2020-05-06]. Dostupné z: http://paulbourke.net/geometry/polygonise/

[4] CONWAY, Chris. RuntimeMeshComponent, UE4 plugin [online]. 2016 [cit. 2020-05-21]. Dostupné z: https://github.com/Koderz/RuntimeMeshComponent

[5] EPIC GAMES, INC, Unreal Engine 4 Features [online]. Cary, North Carolina: [cit. 2020-05-02]. Dostupné z: https://www.unrealengine.com/en-US/features

[6] EPIC GAMES, INC, UDK [online]. Cary, North Carolina: [cit. 2020-05-02]. Dostupné z: https://docs.unrealengine.com/udk/Three/UnrealScriptHome.html

[7] EPIC GAMES, INC, ProceduralMeshComponent, Unreal Engine 4 Documentation [online] [cit. 2020-05-21] Dostupné z: https://docs.unrealengine.com/en-US/API/Plugins/ProceduralMeshComponent/UProceduralMeshComponent/index.html

[8] EPIC GAMES, INC, Materials, Unreal Engine 4 Documentation [online] [cit. 2020-05-24] Dostupné z: https://docs.unrealengine.com/en-US/Engine/Rendering/Materials/index.html

[9] FOLEY, James D.; ANDRIES van Dam; HUGHES, John F.; FEINER, Steven K. Spatial-partitioning representations; Surface detail. Computer Graphics: Principles and Practice. The Systems Programming Series. Addison-Wesley. ISBN 978-0-201-12110-0.

[10] HAN, JungHyun. 3D graphics for game programming. Boca Raton: Chapman and Hall/CRC, 2011. Print. ISBN 9781439827383.

[11] HOSAM, Osama. Developing Simple Games with OpenGL [online]. 2015 [cit. 2020-05-24] Dostupné z: https://www.researchgate.net/publication/283255927_Developing_Simple_Games_with_OpenGL

[12] JAKUBÍKOVÁ, Radka. Perlinovy šumové funkce pro generování textur [online]. 2012, Brno: Vysoké učení technické v Brně [cit. 2020-05-22]. Dostupné z: https://core.ac.uk/download/pdf/30281734.pdf

[13] KARLÍK, Ondřej. Úvod do problematiky polygonového modelování [online]. Praha: České vysoké učení technické v Praze [cit. 2020-05-05]. Dostupné z: http://keymaster.powermac.cz/temp/b.pdf

[14] KEEN SOFTWARE HOUSE S.R.O., Space Engineers [online]. [cit. 2020-05-03]. Dostupné z: https://www.spaceengineersgame.com

[15] LAPRAIRIE, Mark J W, HAMILTON, Howard J. Isovox: A Brick-Octree Approach to Indirect Visualization [online] [cit. 2020-05-24] Dostupné z: https://www.researchgate.net/publication/228454597_Isovox_A_Brick-Octree_Approach_to_Indirect_Visualization

[16] LORENSEN, William E., CLINE, Harvey E. Marching cubes: A high resolution 3D surface construction algorithm. ACM SIGGRAPH Computer Graphics [online]. 1987 [cit. 2020-05-21]. Dostupné z: https://dblp.org/db/conf/siggraph/siggraph1987

[17] PALKO, Martin. Triplanar Mapping [online]. 2014 [cit. 2020-05-24]. Dostupné z: https://www.martinpalko.com/triplanar-mapping/

[18] PROCHÁZKA, Pavel. Optimalizace vykreslování voxelové grafiky [online]. Olomouc: Univerzita Palackého v Olomouci [cit. 2020-05-05]. Dostupné z: https://library.upol.cz/i2/i2.entry.cls?ictx=upol&plang=cs&pretty=csg&repo=upolrepo&key=31741994467

[19] RICHARDSON, Lewis Fry. The Problem of Contiguity, General systems: Yearbook of the Society for the Advancement of General Systems Theory, 1961.

[20] SKALSKÝ, Michal. Eclipse engine [online]. Plzeň: Západočeská univerzita v Plzni, 2010. [cit. 2020-05-01]. Dostupné z: http://graphics.zcu.cz/files/101_DP_2010_Skalsky_Michal.pdf

[21] TIPPETTS, Joshua. Accidental Noise Library [online]. 2011 [cit. 2020-05-23]. Dostupné z: http://accidentalnoise.sourceforge.net/index.html

[22] WARD, Jeff. What is a Game Engine? Game Career Guide [online]. 2008. [cit. 2020-05-01]. Dostupné z: https://www.gamecareerguide.com/features/529/what_is_a_game_.php

[23] ŽÁRA, Jiří. Moderní počítačová grafika. 2., přeprac. a rozš. vyd. Brno: Computer Press, 2004. ISBN 80-251-0454-0.