VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

GRAFICKÉ INTRO 64KB S POUŽITÍM OPENGL

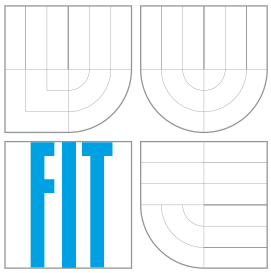
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. TOMÁŠ MILET

BRNO 2012

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

GRAFICKÉ INTRO 64KB S POUŽITÍM OPENGL

GRAPHICS INTRO 64KB USING OPENGL

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. TOMÁŠ MILET
AUTHOR

VEDOUCÍ PRÁCE — doc. Ing. ADAM HEROUT, Ph.D.
SUPERVISOR

BRNO 2012

Abstrakt

Diplomová práce se zabývá tvorbou intra s omezenou velikostí. Práce popisuje metody pro omezení velikosti výsledné aplikace. Hlavní část práce popisuje metody pro generování a animaci grafického obsahu. Zabývá se tvorbou textur a geometrie. Další částí jsou fyzikální simulace částicových a elastických systémů.

Abstract

This thesis deals with the creation of the intro with limited size. This work describes methods for reducing the size of the final application. The main part describes methods for generating graphic content and methods for its animation. It deals with creation of textures and geometry. Another part is aimed on the physical simulation of particle and elastic systems.

Klíčová slova

grafické intro, OpenGL, 64kB, omezená velikost, parametry překladače, exe packer, generování šumu, Perlinův šum, vyšší dimenze, Voroného diagram, barvený přechod, textura, bump mapping, triplanární texturování, skybox, Marching cubes, Marching tetrahedra, částicové systémy, elastické systémy, Eulerova numerická metoda, Runge Kutta metoda, kolize, voda, terén, pohyb kamery, Catmull-Rom, hudba

Keywords

graphics intro, OpenGL, 64kB, limited size, compiler arguments, exe packer, noise generation, Perlin noise, higher dimension, Voronoi diagram, color gradient, texture, bump mapping, Tri-Planar texturing, skybox, Marching cubes, Marching tetragedrons, particle system, elastic system, Euler numerical method, Runge Kutta method, collision, water, terrain, camera motion, Catmull-Rom, music

Citace

Tomáš Milet: Grafické intro 64kB s použitím OpenGL, diplomová práce, Brno, FIT VUT v Brně, 2012

Grafické intro 64kB s použitím OpenGL

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením doc. Ing. Adama Herouta Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Tomáš Milet
21. května 2012

Poděkování

Rád bych poděkoval doc. Ing. Adamovi Heroutovi Ph.D. za jeho vedení a rady v průběhu práce. Dále bych poděkoval své matce za korekci některých gramatických chyb. Nakonec bych poděkoval Dormonovi za jeho extraordinární nápady.

Obsah

Kapitola 1

Úvod

Cílem této práce je vytvoření grafického intra s omezenou velikostí. Práce pojednává o technikách minimalizace velikosti spustitelného souboru pomocí nastavení překladu, komprimování výsledného souboru, vhodného programátorského stylu a generování obsahu podle šablon. Zabývá se generováním grafického obsahu jako jsou textury a geometrie. V práci je popsán způsob generování vícerozměrného šumu. Dále se zabývá vytvořením vícerozměrných Voroného diagramů. Další část práce se zabývá tvorbou textur pomocí šumu, Voroného diagramů, barevných přechodů a transformačních operací. Následuje sekce o generování geometrie intra. V této části je popsán převod objemové reprezentace tělesa na povrchovou reprezentaci. Spadají sem algoritmy Marching cubes a Marching tetrahedra. Dále sem patří vyhlazení povrchu a optimalizace ukládání dat produkovaných těmito algoritmy. Poté se práce zabývá bump mappingem, texturováním, tvorbou skyboxu, částicovými a elastickými systémy, vodou, terénem a pohybem kamery.

1.1 OpenGL

OpenGL je v práci využito pro vykreslování veškeré grafiky. OpenGL („Open Graphics Library“) je nízkoúrovňová grafická knihovna. Za vytvořením knihovny stojí společnost Silicon Graphics, Inc., jež ji uvedla v roce 1992. Knihovna od počátku vyšla v několika verzích, přičemž verze jsou zpětně kompatibilní. Obsahuje stovky funkcí umožňujících specifikaci a manipulaci s grafickými daty. V dřívějších dobách byla kreslící pipeline fixní a nebylo tedy možné ji jakkoliv upravit. Od verze OpenGL 2.0 je k dispozici jazyk GLSL, kterým lze kreslicí pipeline naprogramovat a změnit její chování.

1.2 Intro

Grafické intro bývá krátké video vytvořené programem. Video vytvořené aplikací nebývá nijak interaktivní. Důležitou charakteristikou programu je jeho malá velikost. Existuje několik kategorií velikostí od 1KB přes 64KB až po neomezenou velikost. Program obvykle nepotřebuje ke svému běhu žádná externí data, jako jsou obrázky textur a soubory s modely či hudbou. Klíčovým prvkem intra je kontrast mezi malou velikostí programu a množstvím a silou účinku prezentovaných grafických a zvukových informací. Tento kontrast většinou poukazuje na schopnosti daného programátora nebo skupiny, kteří intro vytvořili. U větších skupin vznikají i různé podpůrné nástroje například pro komponování a přehrávání hudby. Zdrojové kódy programů bývají většinou tajné a autoři si je bedlivě střeží. Intra se objevují

nejčastěji jako programy vytvořené pro zábavu či soutěž. Vyskytují se ale i jako komerční prezentace.

Obsah intra může být krátký příběh, předvedení myšlenky, prezentace schopností autorů, ukázání svých uměleckých schopností, propagace k filmu, reklama nebo jiné věci. Pokud se jedná o intro vytvořené pro zábavu (nejčastější případ) není obsah intra nikterak svázán a náplň je zcela v rukou autora a záleží jen na něm, co bude jeho intro zobrazovat. Časté jsou abstraktní scény, které nemají mnoho společného s realitou, poskytují však vydatný grafický zážitek. Abstraktní intra se nejčastěji objevují u velmi malých velikostí programů. Nevýhodou těchto typů inter bývá, že ač jsou grafické efekty sebelepší časem, s vývojem grafického hardwaru, stejně zastarávají. Proto bývá důležitým prvkem intra příběh. Silný příběh může hodnotu intra značně zvýšit. Nicméně to platí i naopak.

Grafická intra jsou fenoménem. Internet jich obsahuje tisíce a komunita kolem inter je velmi živá. Pořádají se různé soutěže s oceněními. Do budoucna lze očekávat, že tvorba inter bude pokračovat. S přibývajícím výkonem zobrazovacího hardwaru porostou kvalita vizuálních efektů a možnosti inter, avšak vždy bude nejvíce záležet na schopnostech autora.

1.3 Velikost 64KB

Velikost 64KB se může zdát pro aplikaci zobrazující grafiku velmi málo, když i velmi jednoduché programy mohou po kompilaci zabírat velikost několika megabajtů. Toto se často týká vysokoúrovňových jazyků a vysokoúrovňových programovacích prostředků. Tyto vysokoúrovňové nástroje poskytují programátorský komfort, ale obvykle nedbají na výslednou velikost aplikace a ani na její rychlost. V podstatě je mezi kódem programátora a výslednými instrukcemi procesoru několik mezivrstev. V samotných aplikacích se tak vyskytují spousty nevyužitého kódu a statických dat. Když se řekne počítačová grafika, většině uživatelů se vybaví počítačové hry a ty běžně zabírají gigabajty dat, což je i sto tisíckrát více než obvyklá velikost inter. Toto je jedna z nejdůležitějších (ne-li nejdůležitější) charakteristik intra.

Základem malé velikosti inter je tedy odstranění všech nevyužitých částí. Toho lze dosáhnou vhodným (nízkoúrovňovým) programovacím jazykem jako je jazyk C nebo assembler. Dále vhodným nastavením překladače a vhodným programátorským stylem. Další velkou úsporu místa lze dosáhnout generováním grafického obsahu. Místo, aby se data jako textury a modely ukládaly do konstantních dat, uloží se jen způsob jejich vytvoření. Poslední možností je výslednou aplikaci zkomprimovat.

1.4 Příběh

Od začátku jsem chtěl intro, které zobrazuje přírodu. Nechtěl jsem intro, které je příliš abstraktní a mechanické. Intro, ve kterém se vyskytují nereálné mechanické objekty. Proto jsem zvolil mírný motiv - přírodu. V intru jsou nakonec čtyři scény. První z nich je vysokohorská scéna se zasněženými vrcholky štítů. Hory jsou snímány z ptačího pohledu a jsou nasvíceny ranním sluncem. Následuje přímořská scéna. V ní je mírná pahorkatina a mořská hladina. Scéna je osvícena zapadajícím sluncem. Ve scéně se kamera ponoří do stráně. Následuje scéna s tunelem, který směřuje dolů, do kopce. Tunel je zarostlý a vyskytuje se v něm pavučina. Na konci je voda. Poslední scéna je v jeskyni. Jeskyně je z části zatopena. Do jeskyně proniká voda v podobě několika vodopádů. Jsou zde zavěšeny tři barevné koberce, pod kterými jsou rozházeny bedny. Scéna ke konci intra vybouchne.

Kapitola 2

Techniky pro omezení velikosti

Jak již bylo v úvodu řečeno, velikost aplikace je u inter důležitá. Technik pro zmenšení výsledné aplikace je několik. Je to nastavení překladače tak, aby odstranil všechna nepoužívaná data, a aby svůj překlad optimalizoval na výslednou velikost. Dalším způsobem je výslednou aplikaci zkomprimovat speciálními balícími programy - takzvanými Exe packery. Ty aplikaci zkomprimují a vloží na začátek algoritmus rozbalení a spuštění. Další úsporu místa lze dosáhnou zvoleným jazykem a vhodnými konstrukcemi v něm. Velmi důležitou část také tvoří generování grafického obsahu, protože nemůžeme mít uložena prakticky žádná data. Mezi ostatní metody zmenšování velikosti programu patří obfuskace kódu, který je uložen ve statických datech a volba platformy, na které má aplikace běžet (rozdíl ve velikostech aplikace lze pozorovat u třiceti dvou bitových a šedesáti čtyř bitových verzích systému).

2.1 Nastavení překladače

Pro překlad této práce byl použit program GCC. Ten umožňuje několik nastavení pro zmenšení velikosti. Významné parametry jsou:

Parametr -s Kompilátor při využití tohoto parametru zahodí veškerý nevyužívaný kód programu. Pokud se v programu vyskytne kód nebo data na které není odkazováno, nemají význam pro běh aplikace. Parametr také zajistí odstranění všech statických údajů pro ladění programu. Tyto údaje jsou například názvy funkcí a procedur. Proto je vhodné tento parametr nepoužívat při ladění. Tento parametr může výslednou velikost zmenšit i na třicet procent původní velikosti.

Parametr -Os Tento parametr donutí kompilátor optimalizovat kód přeuspořádáním a hledáním výhodnějších konstrukcí. Příkladem může být spojení dvou po sobě jdoucích cyklů se stejným počtem opakování. Díky tomuto parametru může kompilátor obě smyčky spojit dohromady. Nesmí však dojít ke změně významu. Využitím tohoto parametru se aplikace může zmenšit až o několik stovek bajtů.

Parametr -nostdlib Tento parametr odstraní veškeré standardní knihovny z aplikace a zanechá jenom nutné minimum. Parametr se obvykle neobejde bez dalšího parametru: *-Wl,-emain*, kterým se specifikuje vstupní bod aplikace. Nesmíme zapomenout, že aplikováním tohoto parametru se nám znepřístupní většina běžně používaných funkcí jako je například alokace paměti. Proto je nutné tyto chybějící funkce dopsat a to

obvykle v inline assembleru. Příkladem funkce, kterou je nutné dopsat v inline assembleru je funkce exit, bez které se program neukončí korektně. Využitím parametru je možné aplikaci zmenšit i o 5KB, nicméně toto zmenšení sebou přináší řadu nepříjemností v podobě nutného dopsání základních funkcí.

2.2 Exe packer

Exe packer je aplikace na komprimování zkompilovaných programů. Výsledkem jeho aplikování je komprimovaný spustitelný soubor. Těchto komprimačních programů existuje několik. Liší se svými algoritmy pro kompresi. Některé pracují jen v některých operačních systémech a některé jsou multiplatformní. Úzce zaměřené poskytují většinou větší komprimační poměr.

Mezi významnější komprimační programy patří UPX [12] ve verzi 3.08. Tento program je multiplatformní a jsou k dispozici jeho zdrojové kódy. Tím, že je multiplatformní jej lze s výhodou použít pro multiplatformní intra. Tímto komprimačním programem lze mimo spustitelné aplikace komprimovat i dynamicky linkované knihovny. Poskytuje menší komprimační poměr. Vytvářený program může zkomprimovat i na třicet procent jeho původní velikosti.

Další komprimační program se jmenuje kkrunchy [6]. Tento exe packer vyvinula německá skupina Farbrausch. Program běží v operačním systému Windows 7 a poskytuje jeden z nejlepších komprimačních poměrů. Proto bývá často využívám při tvorbě inter. Nastavením parametru při spuštění programu lze zvolit požadovanou výslednou velikost.

2.3 Programátorský styl a konstrukce

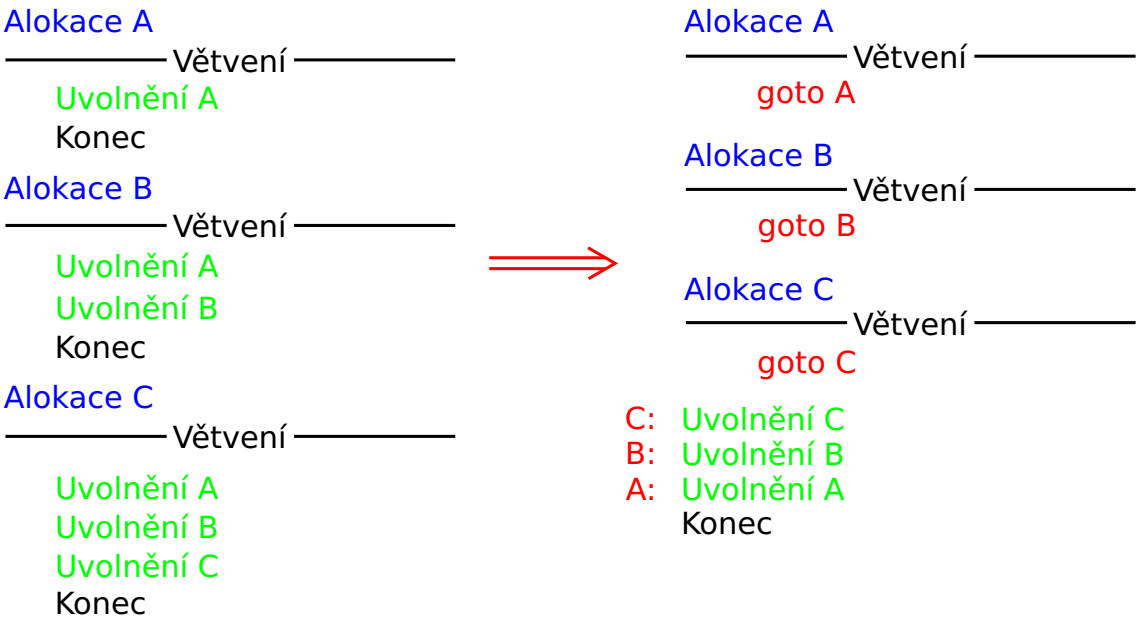
Vhodným programátorským stylem lze ušetřit také nějaké místo. Existuje několik výhodných konstrukcí, které jsou v této práci používány:

- Využívání funkcí. Vhodné je často využívat funkce na části programu, které se opakují. Toto patří k dobrému programátorskému stylu jako způsob dekompozice problému. Nicméně, dobrým návrhem a hledáním míst programu, které lze sloučit do funkce lze dosáhnout zmenšení kódu i o stovky bajtů. Důležité je dbát na to, aby se velikost aplikace zmenšila. Nevhodným použitím funkcí může velikost naopak narůst. Pokud máme jen několik řádků kódu, které používáme na mnoha místech (například sčítání dvou dvojrozměrných vektorů), nemusí mít vytváření funkce pro tyto řádky z pohledu velikosti smysl. Volání funkce pro sečtení dvou vektorů může samo o sobě zabírat více místa. Proto je výhodnější dané řádky kódu neobalovat funkcí. Dobré z pohledu velikosti tak čitelnosti může být využití preprocesoru či inline funkcí, které se na místě volání rozbalí.

- Využití rekurze. Programátorskou konstrukcí, která zmenšuje výslednou velikost je také rekurze. Rekurze je sice pomalejší na vyčíslení, ale její vhodné použití může pomoci aplikace zmenšit. Rekurze je zvláště vhodná pro procházení seznamů, zásobníků, front a stromů, kde se oproti řešení bez rekurze značně zmenšuje výsledná velikost algoritmů.

- Využití příkazu goto. Příkaz goto (nepodmíněný skok) už většinou k dobrému programovacímu stylu nepatří. Bývá doporučeno tento příkaz nevyužívat, kvůli zanesení

nepřehlednosti do kódu. Nicméně, pokud potřebujeme ušetřit i desítky bajtů, může být použití tohoto příkazu přínosné. Výhodné je použít jej na vyskočení ze zanořené smyčky. Další využití může být pro uvolňování paměti. Uvažme případ, kdy si funkce v průběhu výpočtu alokuje na několika místech paměť. Zároveň se funkce větví a může skončit na různých místech. Před ukončením funkce je záhodné veškerou pomocnou alokovanou paměť uvolnit. Vzhledem k tomu, že máme vícero ukončovacích míst, je nutné uvolnění dočasně alokované paměti provést na každém z nich, přičemž každé uvolnění se mírně liší. Toto lze vyřešit vytvořením uvolňovací funkce. Funkce bude mít tolik parametrů, kolik je proměnných k uvolnění. Navíc uvolňovací funkce bude kontrolovat, zda je proměnná naplněná (proměnná nemusela být v daném ukončovacím místě původní funkce alokovaná). Nevýhodou tohoto řešení je, že neušetří moc místa (pokud vůbec nějaké). Další nevýhodou je, že uvolňovací funkce se využívá jen pro tuto funkci. Výhodnější je na konec funkce uvést uvolnění proměnných v opačném pořadí něž jak jsou alokovány a za ně vložit jediný ukončovací bod funkce. V místech, kde původní funkce končila se uvede příkaz goto na příslušné místo do uvolňovací sekvence. Situace je znázorněna na obrázku 2.1.



Obrázek 2.1: Využití příkazu goto

- Vytýkání kódu. Vytýkáním lze také ušetřit několik bajtů. Vytknout lze (mimo jiné) také prolog a epilog části kódu. Použití vytýkání epilogu můžeme vidět u příkladu využití goto znázorněného na 2.1. O vytýkání kódu se snaží i kompilační program (při využití parametru *-s*), nicméně programátor má veškeré informace, tak tuto činnost dělá efektivněji.

- Rozbalení smyček. U smyček jejichž tělo je dostatečně malé a počet opakování nízký, může vést rozbalení k uspoření místa.

- Obecné abstraktní datové typy. Vytvořením obecných abstraktních datových typů, jako je například seznam, můžeme program také zmenšit. V případě, kdy seznam potřebujeme na vícero místech, může být vytvoření obecného seznamu dobrým krokem.

Obecná implementace abstraktního datového typu obvykle zabere více místa než implementace konkrétní, avšak, pokud je tento typ využíván na vícero místech programu, může být jeho použití výhodné. Zmenšení velikosti totiž spočívá ve sdílení obslužných funkcí.

- Využívání bitových a jiných operací. Části některých algoritmů lze efektivně zmenšit pomocí série operátorů. Výsledkem je úspora místa za cenu přehlednosti. Příkladem nechť je inicializace matice M na jednotkovou matici:

```
for(int i=0;i<16;++i)
  M[i] = i%4 == i/4;
```

- Inline assembler. Assembler se často využívá u malých inter, protože poskytuje velkou úsporu místa. Jeho nevýhodou je však ztráta přehlednosti a do jisté míry i přenositelnosti kódu.

- Jiné. Mezi další techniky můžeme zařadit špatné programátorské návyky. V případě, že se snažíme ušetřit každý bajt, se můžeme rozhodnout neuvolňovat námi alokovanou paměť a doufat, že potřebnou dealokaci provede operační systém. Další možností je vytvoření vícero zkompilovaných verzí aplikace, a tím odstranit nutnost zpracování argumentů po spuštění (ve kterých může být například šířka a výška okna).

Při implementaci algoritmů pro intro se často setkáme se situací, kdy se musíme rozhodnout, mezi velikostí, rychlostí, znovupoužitelností a čitelností. Všechny tyto vlastnosti jdou proti sobě až na čitelnost, kterou lze téměř vždy zajistit. Záleží jen na situaci a schopnostech programátora předpovídat, která z implementací daného algoritmu je vhodnější. Případně lze naprogramovat vícero variant, mezi kterými se v případě potřeby přepne preprocesorem.

2.4 Šablony, generování grafického obsahu

Bez generování grafického obsahu by se intra neobešla. Generování a šablony jsou základem všech inter. Díky generování obsahu podle šablon lze ušetřit nejvíce místa. Lze ušetřit desítky i stovky megabajtů. Namísto, aby se v aplikaci ukládaly textury a modely, uloží se jen způsob, jak je vytvořit. Například nebudeme ukládá body kružnice, uložíme si jen poloměr a střed a algoritmus na vygenerování bodů kružnice. Tento způsob uložení má několik výhod. První z nich je velikost, která je o mnoho menší. Další výhodou je parametrizování generování. Můžeme si zvolit kvalitu (počet bodů na kružnici), střed i poloměr. Neuložili jsme si tedy jen jednu kružnici, ale všechny možné kružnice ve všech možných kvalitách a to s konstantní velikostí dat. Výsledné intro lze tedy dobře parametrizovat a jen malými změnami parametrů markantně měnit grafický obsah. Generování grafického obsahu je věnována celá kapitola 3.

2.5 Další techniky, obfuskace

Mezi další techniky minimalizace velikosti patří obfuskace [10]. Obfuskace je využití syntaxe jazyka tak, že se použijí co nejkratší textové konstrukce pro daný program. Obvykle to zahrnuje odstranění komentářů, zkrácení identifikátorů a zrušení formátování. Toto lze použít jen ve speciálních případech, kdy je program uložen nezkompilovaný ve statických datech. Týká se to programů v jazyce GLSL. U rozsáhlejších programů v GLSL (shaderech) lze dosáhnou úspory až stovek bajtů. Nevýhodou obfuskace je absolutní ztráta čitelnosti.

Kapitola 3

Generování grafického obsahu

Jak již bylo řečeno v předcházející kapitole, generování grafického obsahuje je pro intra klíčové. Generují se textury, generují se modely, generuje se geometrie. Generuje se rozmístění modelů ve scéně. Generování grafických objektů sebou přináší výhody malé velikosti a parametrizování generování. Můžeme si tedy například uložit jednu šablonu stromu a vhodným měněním parametrů generovat pokaždé jiný strom. Nevýhodou generování je delší nahrávací čas. Vytvoření kvalitní textury může být časově velmi náročné, z tohoto důvodu mívají intra delší inicializační časy.

3.1 Šum

Šum je náhodný signál. Se šumem se často setkáváme v elektronice. Zde se jedná o nežádaný jev, který zkresluje výsledky. Setkáváme se s ním také u digitálních fotografii, kde se například projevuje jako změna hodnoty daného pixelu. V informatice se však šum využívá (například jako zdroj náhodných čísel). Náhodná čísla (skutečně náhodná čísla) se využívají v kryptografii. Běžně většinou nemáme k dispozici generátory skutečně náhodných čísel (je k tomu obvykle potřeba specializovaný hardware), proto se používají kongruentní generátory produkující pseudonáhodná čísla. Seznam hodnot vygenerovaných tímto generátorem je pak považován za šum. V počítačové grafice se setkáváme s různými druhy generátorů šumu založených na různých druzích generátorů pseudonáhodných čísel. Na generátory čísel je kladena řada požadavků z pohledu statistiky. Tyto požadavky je nutné dodržet v případech, kdy hrají důležitou roli (například v simulacích). V počítačové grafice si však často vystačíme i s jednoduchými implementacemi. U inter hraje nejdůležitější roli velikost a rychlost.

3.1.1 Generátory pseudonáhodných čísel

Existuje vícero druhů generátorů. Některé v podobě kongruentního generátoru, jiné v podobě funkce, která má vstupní parametry a vždy pro stejné nastavení parametrů produkuje stejný pseudonáhodný výstup. Tyto generátory produkují celočíselný výstup s určitým rozsahem. Je vhodné generátory upravit tak, aby vracely čísla s plovoucí řádovou čárkou v rozsahu $\langle 0, 1\rangle$. S tímto rozsahem se pak později lépe pracuje.

- Kongruentní generátor. U kongruentního generátoru se vyskytuje inicializační hodnota (takzvané semínko). Generátor na základě tohoto semínka vygeneruje novou hodnotu a tu do semínka uloží. Lineární kongruentní generátor má tvar $x_{n+1} = (a \cdot x_n + b)\%m$, kde a, b, c jsou vhodně zvolené konstanty. Semínko je pak x_0.

- Generátor založený na funkci. Tento typ generátoru vrací stejná čísla na základě stejných celočíselných vstupů. Nalezení takové funkce může být obtížné. Základem je operace zbytku po celočíselném dělení (%), která se při generování náhodných čísel často používá. Tvar této funkce, kterou jsem v práci použil je $f(x) = (x^5 + x^4 + x^3 + x^2 + x^1)\%N$. Číslo N je hodnota $2^n, n > 0$. Tento zápis lze pomocí Hornerova schématu zefektivnit: $f(x) = (x(x(x(x(x + 1) + 1) + 1) + 1))\%N$. Pro vyčíslení jedné hodnoty potřebujeme jen čtyři násobení, čtyři sčítání a jednu operaci modulo. Experimentálně jsem si ověřil, že pro $N = 2^{24}$ funkce vrací pro vstup $x \in \langle 0, N - 1\rangle$ pokaždé jinou hodnotu. Tento poznatek bychom mohli využít pro konstrukci inverzní funkce.

3.1.2 Perlinův šum

Perlinův šum posaný v [13] a v [3] je šum založený na šumové funkci. V počítačové grafice se hojně využívá. Perlinův šum vrací pro stejné souřadnice stejnou hodnotu v rozsahu $\langle -1, 1\rangle$. Výsledná hodnota šumu na daných souřadnicích je získána jako součet šumových funkcí o různých frekvencích a amplitudách. Frekvence se obvykle volí jako násobek základní frekvence f_0 mocninou dvě. Amplituda jednotlivých frekvencí je určena počáteční amplitudou a perzistencí. Pokud bychom zvolili počáteční amplitudu $A = 1$, perzistenci $P = 1/2$, tak pro frekvenci $f_0 \cdot 2^n$ je amplituda $a = A \cdot P^n$. Perlinův šum budeme používat pro generování skyboxů.

3.2 Generování šumu půlením intervalu

Kdybychom vygenerovali sérii hodnot pomocí generátoru náhodných čísel, získali bychom šum, který není pro grafiku příliš vhodný. Můžeme jej vidět v levé části obrázku 3.1. Tento šum má hodnoty v sousedních bodech příliš odlišné. Pro další generování (například textur) je tedy vhodné šum generovat jinak. Důležité je, aby rozdíly v sousedních bodech byly malé. Jedním ze způsobů jak to zajistit, je generovat hodnoty s ohledem na již vygenerované hodnoty. Další vlastností, kterou chceme zaručit je, aby šum na sebe navazoval. Toto lze zajistit algoritmem půlení intervalu, jehož výsledek je v pravé části obrázku 3.1. Jak je z tohoto obrázku patrné, šum produkovaný tímto algoritmem je vhodnější pro další generování (například mraků).

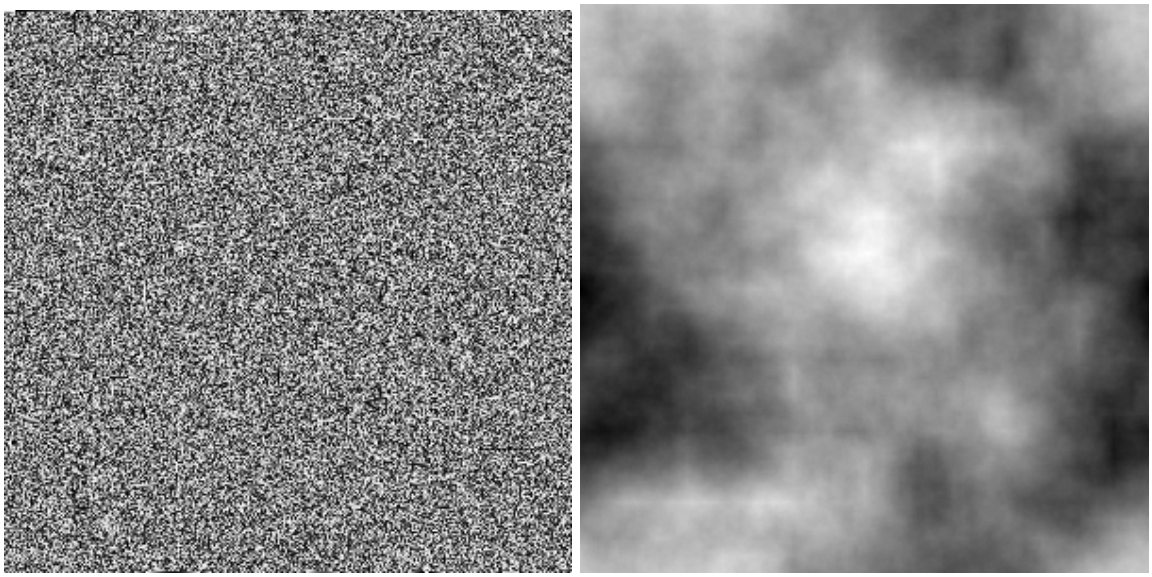
Generování šumu pomocí půlení intervalu je dopodrobna popsáno v bakalářské práci [11]. Základem je generátor čísel (v podobě kongruentního generátoru). Předpokládejme, že chceme vygenerovat jednorozměrný šum reprezentovaný polem o N prvcích s indexy $0, 1, \ldots, N - 1$. Máme k dispozici základní rozsah generátoru d a faktor vyhlazování m. Algoritmus pracuje následovně (jeho vizualizace je v levé části obrázku 3.2 a výstup v pravé části obrázku 3.2):

1. Nejprve se vygeneruje náhodná hodnota s rozsahem generátoru $\langle -d, d\rangle$ a uloží se do prvku s indexem 0.

2. Nastaví se nový rozsah $d := d/m$.

3. Určí se index S vhodného středu. Je to největší možné číslo 2^n takové, že je menší než počet prvků pole (N).

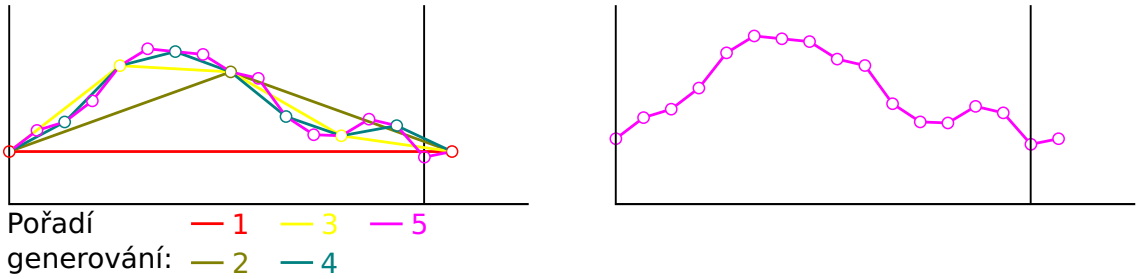
4. Prvek s indexem S má hodnotu součtu váženého průměru a nově vygenerované hodnoty s upraveným rozsahem. Vážený průměr je počítán z hodnot nejbližších, již vygenerovaných prvků zleva a zprava od S, přičemž hodnota váhy je určená vzdáleností



Obrázek 3.1: Vlevo: šum jako prostá sekvence náhodných čísel. Vpravo: šum vygenerovaný algoritmem půlení intervalu.

jejich indexů od S. Čím je prvek od S vzdálenější, tím má jeho hodnota menší váhu. Součet vah je roven 1. Pokud žádný vygenerovaný prvek zprava neexistuje, použije se prvek s indexem 0 a vzdálenost je spočítána jako vzdálenost k poslednímu prvku pole plus jedna. Tímto je zajištěno opakování.

5. Krokem 4 nám vznikla dvě pole: jedno s indexy $0, \ldots, S$ a druhé s indexy $S, \ldots, N-1$. Na tato pole se aplikuje tento algoritmus znovu od kroku 2. Pole však musí mít více než dva prvky.



Obrázek 3.2: Vlevo: kroky algoritmu půlení intervalu. Vpravo: výsledek algoritmu půlení intervalu.

Výše popsaný postup je určen pro jednorozměrný šum. Více rozměrná varianta je obdobná. Výpočet hodnoty středního bodu se například pro dvojrozměrný šum nepočítá ze dvou ale ze čtyř okolních bodů. Podrobně je obecný algoritmu popsán v bakalářské práci [11].

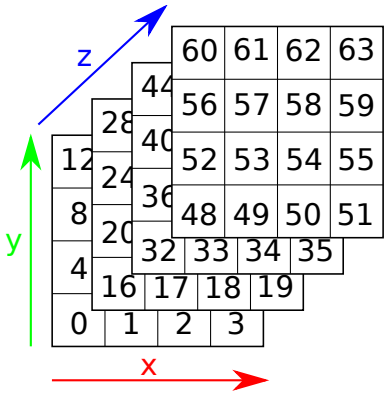
3.3 Více dimenzí

Jelikož se v této práci často pracuje s vícerozměrnými daty (příkladem může být výše zmíněný vícerozměrný šum), předvedeme si zde i postup, jak s těmito daty jednotně pracovat. Vícerozměrná data budeme reprezentovat jednorozměrným seznamem (polem) prvků. Způsob uspořádání prvků vícerozměrných dat do jednorozměrného seznamu je ukázán na obrázku 3.3, kde je předvedeno uspořádání trojrozměrných dat. Index do d rozměrného pole $I = (i_1, i_2, \ldots, i_d)$ se skládá z d složek (čísel) i_n, z nichž každé indexuje jinou dimenzi (složka i_1 nejnižší a složka i_d nejvyšší). Rozměry (výška, šířka, ...) dat označme $r = (r_1, r_2, \ldots, r_d)$. Přepočet na index J do jednorozměrného pole je dán vztahem:

$$J = i_1 + i_2 \cdot r_1 + i_3 \cdot r_1 r_2 + \ldots = \sum_{k=1}^{d} \left(i_k \prod_{h=1}^{k-1} r_h \right) \tag{3.1}$$

Vztah 3.1 můžeme zefektivnit a převést na rekurentní vztah:

$$\begin{aligned} J_1 &= i_d \\ J_n &= J_{n-1} \cdot r_{d-n+1} + i_{d-n+1} \\ J &= J_d \end{aligned}$$



Obrázek 3.3: Uspořádání 3D dat

3.4 Voroného diagram

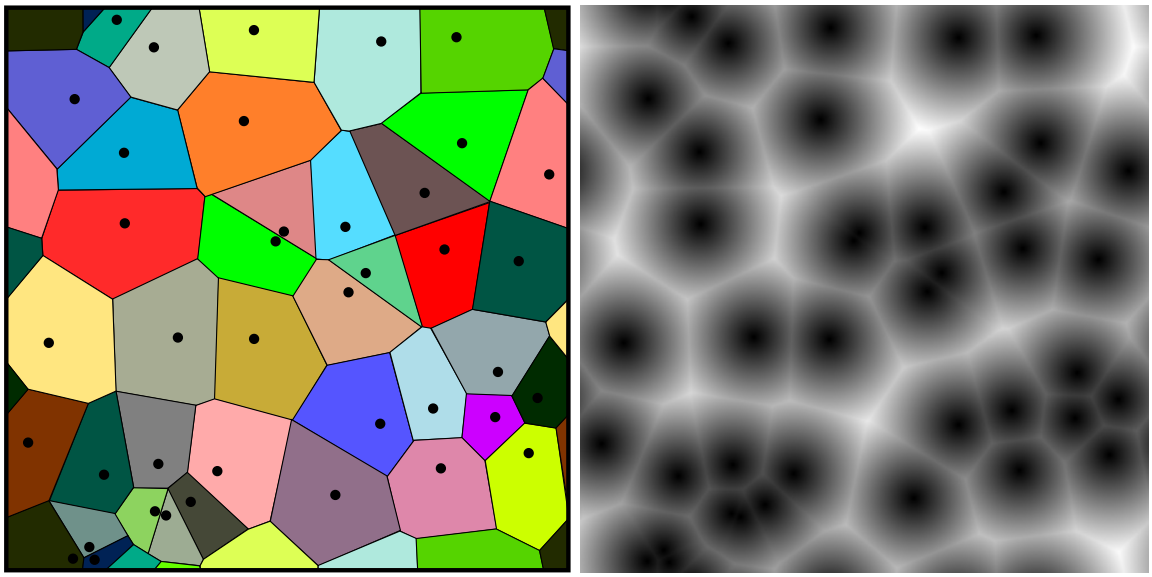
Voroného diagram popsaný v [1] je způsob dekompozice metrického prostoru. Dekompozice je určena vzdálenostmi k dané diskrétní množině objektů v prostoru, například diskrétní množinou bodů. V levé části obrázku 3.4 můžeme vidět příklad rozdělení dvourozměrného prostoru. Pro rozdělení byla použita množina náhodných bodů (reprezentovány černými tečkami). Jednotlivé barevné oblasti patří k danému černému bodu, který se v nich nachází. Všechny body z jedné barevné oblasti mají společnou vlastnost: vzdálenost k černému bodu ležící v této oblasti je menší než vzdálenost ke všem ostatním černým bodů. Matematicky si můžeme, pro naše potřeby, Voroného diagramy popsat následovně: mějme metrický prostor (M, ρ) dimenze d, který reprezentuje všechny body Voroného diagramu. Zobrazení $\rho : M \times M \to \mathbb{R}$ je v našem případě Eulerova vzdálenost. Dále mějme množinu

indexů K a body $P_k \in M, k \in K$. Oblast (nebo buňka) Voroného diagramu, která je asociována k bodu P_k označme množinou R_k. Pak jsou body v jedné množině dány vztahem $R_k = \{x \in M | \forall i \in K \setminus \{k\} : \rho(x, P_k) \leq \rho(x, P_i)\}$. Počet buněk diagramu je $|K|$.

Využití Voroného diagramu může být například v texturách nebo při generování geometrie. Voroného diagramy dělí prostor na buňky, proto poskytují dobrý základ k buněčným texturám. Příkladem může být textura dlažby nebo textura vysušené půdy.

Aby se s diagramem dobře pracovalo je vhodné jej převést na normalizované vzdálenostní. Vzdálenostní pole můžeme vidět v pravé části obrázku 3.4. Každý prvek pole (pixel textury) si místo příslušnosti do dané oblasti nese informaci o normalizované vzdálenosti ke středu oblasti do které patří. Pro výpočet vzdálenosti použijeme Eulerův výpočet vzdálenosti pro dimenzi prostoru d:

$$\rho(x, y) = \sqrt{\sum_{k=1}^{d} (x_k - y_k)^2} \tag{3.2}$$



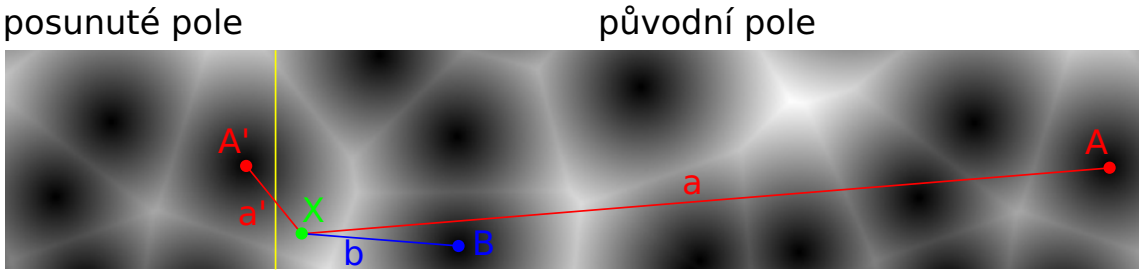
Obrázek 3.4: Vlevo: Voroného diagram, Vpravo: vzdálenostní pole.

3.4.1 Navazování

Navazování Voroného diagramu a tedy i vzdálenostního pole, které je popsáno v [14] je důležitá vlastnost. Aby se dala textura vytvořená na základě vzdálenostního pole použít opakovaně vedle sebe, je nutné zajistit navazování. Navazování lze u vzdálenostního pole zajistit úpravou výpočtu vzdálenosti. Označme d dimenzí prostoru. Bod v prostoru $x = (x_1, x_2, \ldots, x_d)$. Rozměry pole (výška, šířka, ...) označme $r = (r_1, r_2, \ldots, r_d)$. Předpokládejme, že všechny body vzdálenostního pole mají nezáporné souřadnice. Upravený výpočet vzdálenosti 3.2, který respektuje navazování, je:

$$\rho'(x, y) = \sqrt{\sum_{k=1}^{d} \min(|x_k - y_k|, r_k - |x_k - y_k|)^2} \tag{3.3}$$

Funkce min vrací minimum ze dvou hodnot. Jak je z upraveného vypočtu patrné, absolutní velikost dílčího rozdílu $|x_k - y_k|$ nemůže být větší než $r_k/2$. V případě, kdy je $|x_k - y_k| > r_k/2$ se použije pro výpočet bod, který leží v sousedním poli. Sousední pole je totožné, jen je posunuté o r_k. Názorně je to vidět na obrázku 3.5.

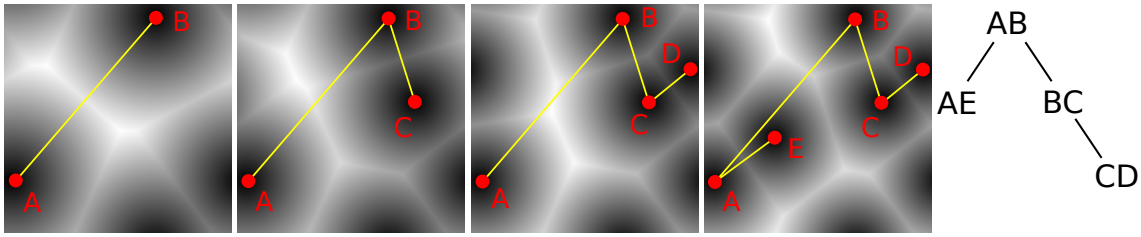


Obrázek 3.5: Výpočet vzdálenosti. Počítáme vzdálenost k bodu X. Vzdálenost ke středu buňky A v neopakujícím poli je $a = |XA|$. Buňka, která je nejblíže k bodu X v neopakujícím poli je B (se vzdáleností $b = |XB|, b < a$). V opakujícím poli je nejblíže bodu X střed buňky A', který je totožný s A jen leží v posunutém poli. Vypočítaná vzdálenost je tedy $a' = |XA'|$.

3.4.2 Optimalizace

Výpočet vzdálenostního pole je časově poměrně náročná záležitost. Pro pole šířky w a výšky h a počtu buněk b je potřeba provést $w \cdot h \cdot b$ výpočtů vzdáleností abychom zjistili nejmenší vzdálenost bodů ke středu buněk. To znamená, z každého bodu pole spočítat vzdálenost ke všem středům buněk a vybrat tu nejmenší. Kdybychom pro všechny body věděli, ke které buňce patří bylo by potřeba jen $w \cdot h$ výpočtů vzdáleností. Optimalizací výpočtů se budeme snažit tomuto počtu výpočtů vzdáleností přiblížit.

Způsob, jak toho dosáhnout je rozmístit středy buněk do binárního stromu a počítat vzdálenosti jen k části stromu. Binární strom má v každém uzlu dvojicí bodů. Tato dvojice bodů dělí prostor na dvě části podle vzdálenosti k nim. Vytvořit strom je snadné. Nejprve se vytvoří kořenový uzel z prvních dvou středů buněk. Poté se do stromu postupně přidávají další středy. Pokud je nově vkládaný střed blíže k prvnímu bodu uzlu, vloží se do levého podstromu. Pokud je blíže druhému bodu uzlu, vloží se do pravého podstromu. Pokud podstrom neexistuje, vytvoří se nový uzel z dvojice, kterou tvoří nově vkládaný střed a první nebo druhý bod rodičovského uzlu. Postupná tvorba stromu je znázorněna v levé části obrázku 3.6. Výsledný binární strom je pak v pravé části obrázku 3.6.

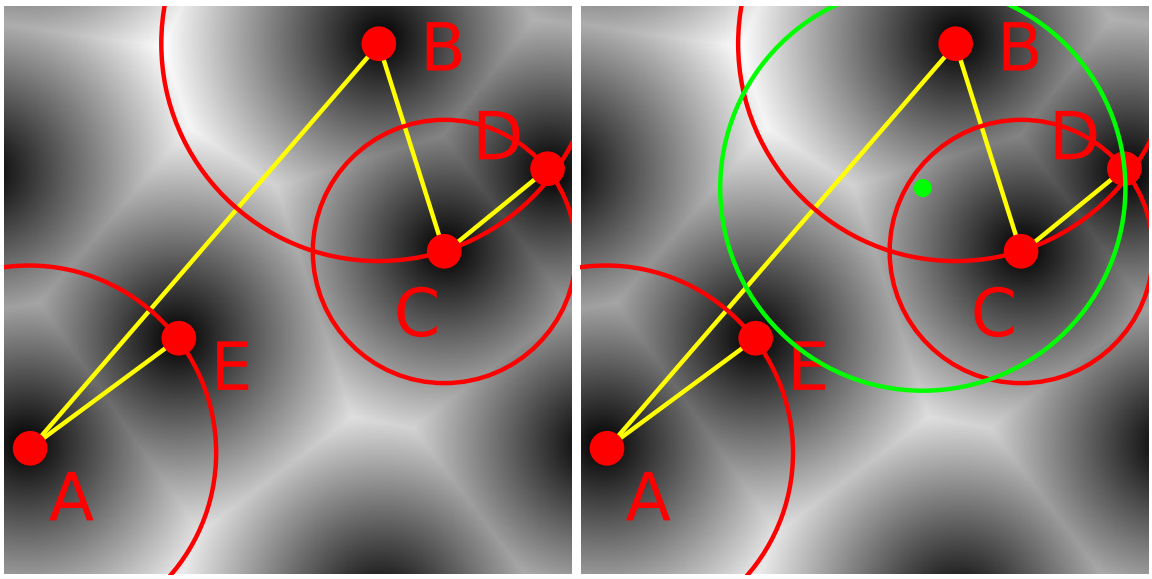


Obrázek 3.6: Vlevo: Postupné vkládání bodů do stromu. Vpravo: Výsledný strom

Když máme vytvořený strom pro všechny středy buněk, je ještě potřeba spočítat k oběma bodům každého uzlu stromu vzdálenosti k nejvzdálenějším potomkům a ty do daného

uzlu uložit. Vzdálenosti jednotlivých bodů uzlů jsou znázorněny kružnicemi v levé části obrázku 3.7.

Nalezení nejkratší vzdálenosti pro každý bod pole pak spočívá ve zvolení poloměru kolem zkoumaného bodu. V prvním kroku zvolíme poměr jako vzdálenost ke středu náhodné buňky, pravá část obrázku 3.7. V dalších krocích budeme volit poloměr ze vzdálenosti ke středu poslední nejblíže nalezené buňky. Pokud se kružnice kolem aktuálně zkoumaného bodu nepřekrývá s danou kružnicí v aktuálním uzlu, není třeba prohledávat jeho podstrom. Pokud ano, prohledá se podstrom uzlu. Pokud je vzdálenost k bodu uzlu menší než aktuální poloměr kolem zkoumaného bodu, zmenšíme poloměr na tuto vzdálenost. Tím se nám neustále zmenšuje poloměr kolem zkoumaného bodu, než nalezneme hledanou vzdálenost.



Obrázek 3.7: Vlevo: Kružnice kolem uzlů. Vpravo: Kružnice kolem zkoumaného bodu s poloměrem k náhodně vybrané buňce D.

3.5 Generování textur

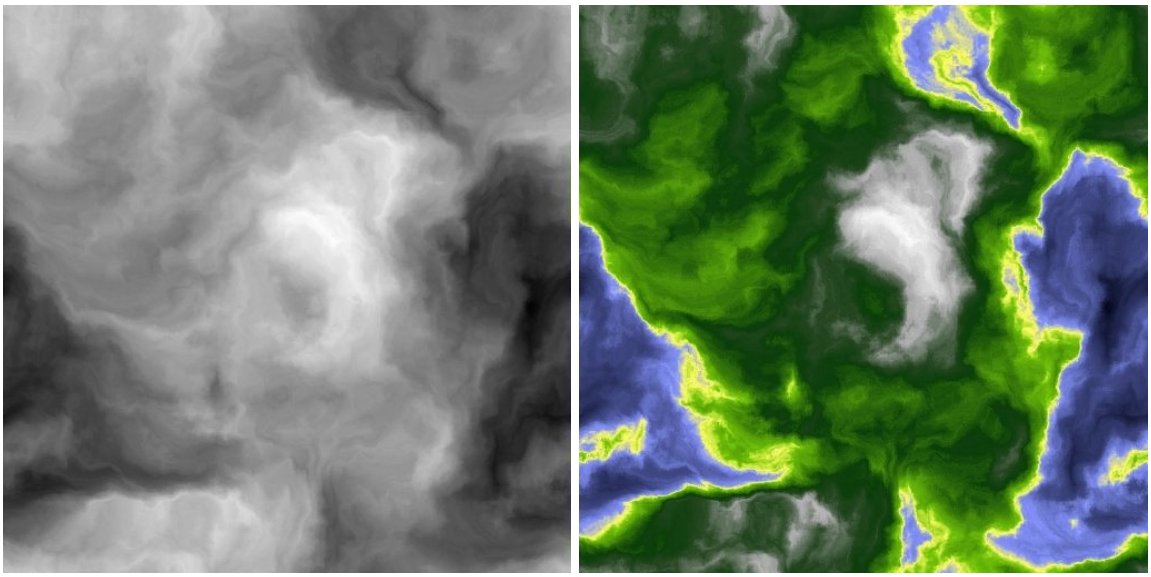
Textura slouží pro detailní popis struktury povrchu. Vyskytují se nejčastěji ve formě dvojrozměrných obrázků. Můžeme se ale setkat i s jednorozměrnými, trojrozměrnými nebo dokonce čtyřrozměrnými texturami. Trojrozměrné textury slouží pro popis objemu a čtyřrozměrné pro popis objemu měnícího se v čase (například plameny ohně, kouř). Textury si díky jejich velikosti nemůžeme do programu uložit, a tak je musíme generovat. Vytvoření textury bývá obvykle časově náročné, neboť obsahují velké množství dat. Základem pro vytvoření textur v této prací je šum, který je popsaný v části 3.1 a vzdálenostního pole vytvořené z Voroného diagramu, který je popsaný v části 3.4. Tyto dvě metody nám produkují d rozměrné pole. Všechny hodnoty těchto polí jsou v rozsahu $\langle 0, 1 \rangle$. Příklad šumu můžeme vidět v pravé části obrázku 3.1 a příklad vzdálenostního pole v pravé části obrázku 3.4. Nejdůležitějším vstupem algoritmu pro vytvoření šumu je faktor vyhlazení, který nám určuje poměr nízkých frekvencí a vysokých frekvencí ve výsledném šumu. U Voroného diagramu je nejdůležitějším vstupem rozmístění a počet středů buněk. Nicméně tyto možnosti nám neposkytují dostatečnou variabilitu textur. Proto je vhodné pole hodnot produkované

těmito postupy ještě dále upravit. Upravit takové pole můžeme prostým obarvením pomocí barevného přechodu, globální transformací jako je například vyhlazení, lokální transformací a složením několika vstupů. Jednotlivé úpravné operace si popíšeme v dalších částech práce.

3.5.1 Barevné přechody

Příklad barevného přechodu je zobrazen na obrázku 3.8. Barevný přechod (nebo též gradient) jsou v podstatě tři funkce $R(x), G(x), B(x)$ (čtyři v případě přítomnosti alfa kanálu). Tyto funkce jsou definované na intervalu $\langle 0, 1 \rangle$. Jejich vstup je hodnota celkové intenzity barvy a výstupem intenzita červené respektive zelené respektive modré barvy. Intenzity jsou taktéž v rozsahu $\langle 0, 1 \rangle$. Barevné přechody můžeme výhodně použít na obarvení obrázků v odstínech šedé. Znázorněné to můžeme vidět na obrázku 3.9. Pro uložení barevného přechodu stačí uložit jen kontrolní barvy a hodnoty mezi nimi interpolovat. Například u barevného přechodu na Obrázek 3.8 stačí uložit pouze barvy: tmavě modrá, světle modrá, žlutá, světle zelená, tmavě zelená, šedá a bílá (v pořadí zleva doprava). Tímto můžeme zredukovat paměťovou náročnost. Další možností je vytvořit algoritmus na generování barevných přechodů. Musíme však zvážit, jestli potřebujeme tolik různých barevných přechodů, že se nám je oplatí generovat.

Obrázek 3.8: Barevný přechod, který je možné využít na obarvení výškové mapy.



Obrázek 3.9: Vlevo: vstupní obrázek v odstínech šedé. Vpravo: obarvený obrázek z leva barevným přechodem z obrázku 3.8

3.5.2 Globální transformace

Globální transformací budeme v textu rozumět operaci, která transformuje vstupní d rozměrné pole hodnot na výstupní d rozměrné pole hodnot. Pro výpočet jednoho prvku výstupního pole s indexem I budeme potřebovat hodnoty okolí vstupního pole kolem indexu I. Jedná se tedy o d rozměrnou konvoluci s d rozměrným konvolučním jádrem. Pokud si vytvoříme obecnou konvoluci, můžeme ji použít pro řadu transformací. Příkladem těchto transformací je vyhlazení a detekce hran. Konvoluční jádro pro vyhlazení obsahuje hodnoty, které jsou všechny stejné a jejichž součet je roven jedné.

3.5.3 Lokální transformace

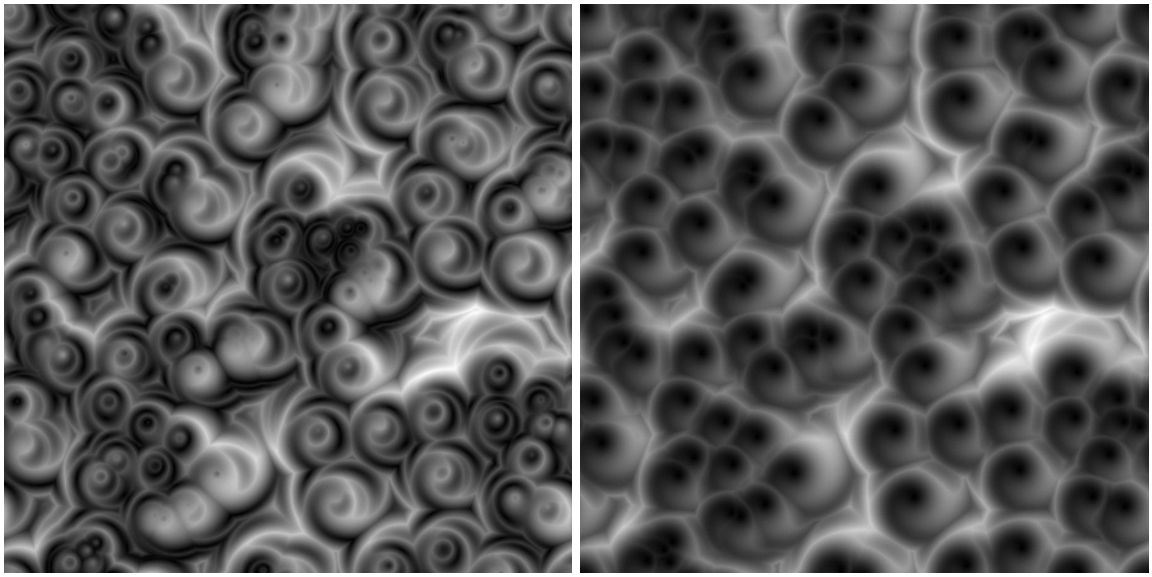
Lokální transformací budeme v textu rozumět (podobně jako globální transformace) operaci, která transformuje vstupní pole na výstupní pole hodnot. Pole mají rozměr d. Pro výpočet jednoho prvku výstupního pole s indexem I budeme potřebovat několik hodnot ze vstupního pole. Výběr těchto hodnot závisí na transformaci. Možnosti lokální transformace jsou velké. Vše závisí na transformační funkci. Důležité je, aby nám transformace zachovávala opakování. Kdykoliv si transformace zažádá hodnotu pole na souřadnicích, které leží mimo rozsah pole, budeme muset souřadnice přepočítat. Souřadnice bodu pole je $I = (i_1, i_2, \ldots, i_d)$ a velikost pole (výška, šířka, ...) je $r = (r_1, r_2, \ldots, r_d)$. Pokud je složka souřadnic $i_n < 0 \vee i_n \geq r_n$ (hodnoty pole indexujeme od nuly), je potřeba ji přepočítat podle vztahu: $i'_n = ((i_n \% r_n) + r_n) \% r_n$.

V levé části obrázku 3.10 můžeme vidět velmi jednoduchou transformaci. K výpočtu výstupní hodnoty na souřadnicích $I = (i_1, i_2)$ jsou zapotřebí dvě hodnoty vstupního pole. První hodnota vstupního pole, označme ji a , je na totožných souřadnicích. Druhá hodnota, označme ji b, je na souřadnicích $(i_1 + r \cdot \cos(2\pi a), i_2 + r \cdot \sin(2\pi a))$. Hodnota r udává poloměr. Hodnota b je výsledná hodnota výstupního pole na souřadnicích I. Můžeme vidět, že hodnota a udává normalizovaný úhel. Spolu s poloměrem r vytváří vektor. Když tento vektor přičteme k původním souřadnicím, máme souřadnice bodu, jehož hodnota je použita pro výstup – tedy hodnotu b. Na dalších obrázcích je postup transformace obdobný. V pravé části obrázku 3.10 je výše popsaný postup opakován několikrát. Místo abychom použili dva body ze vstupního pole, použijeme jich několik. Poslední bod udává hodnotu výstupu. Další příklady lokálních transformací můžeme vidět na obrázku 3.12

3.5.4 Skládání operací

Nyní, když máme základ všech textur v podobě distančního pole a šumu, lokální a globální transformace a barevné přechody, můžeme se pustit do vytváření textur. Vytvoření jedné textury si můžeme představit jako strom. Uzly stromu představují operaci. Operace uložena v uzlu bere své vstupy ze svých potomků a výstup posílá rodiči. V listech tohoto stromu se nachází buď distanční pole nebo šum. V uzlech stromu je lokální nebo globální transformace nebo jedna z následujících operací:

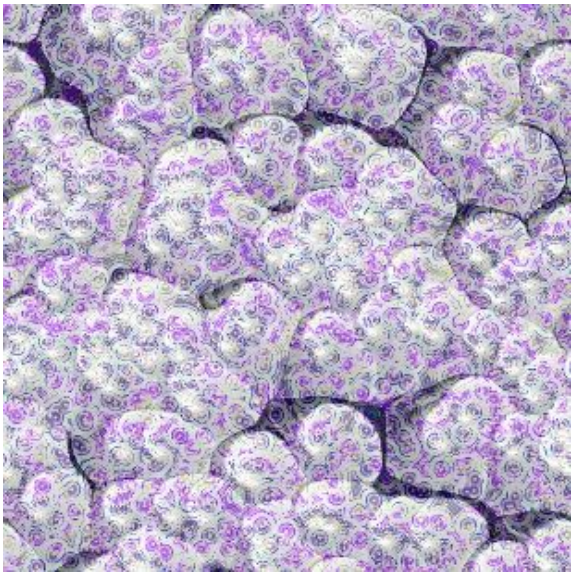
- Součet hodnot vstupů.
- Rozdíl hodnot vstupů.
- Násobení hodnot vstupů.
- Součet hodnot vstupů s ohledem na alfa kanál.



Obrázek 3.10: Lokální transformace

- Kompozice vstupů do vícekanálové textury.

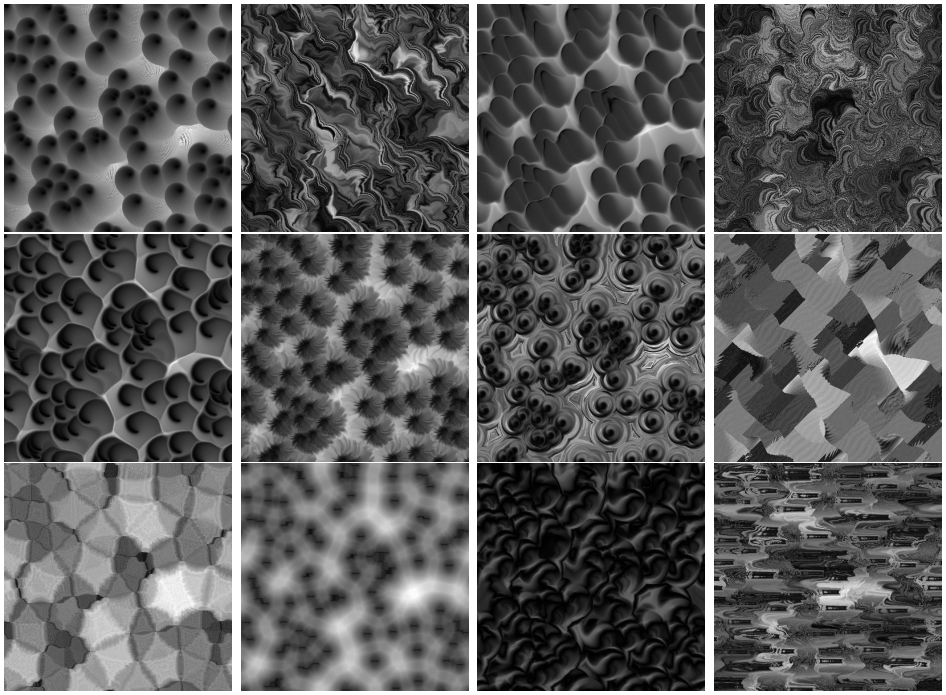
Příklad textury složené pomocí několika operací můžeme vidět na obrázku 3.11



Obrázek 3.11: Sestavená textura

3.6 Generování geometrie

Pro generování geometrie můžeme použít několik postupů. V práci budeme používat algoritmus, který převádí objemovou reprezentaci tělesa na povrchovou. Pro vygenerování objemové reprezentace můžeme použít šum nebo vzdálenostní pole. Můžeme použít také



Obrázek 3.12: Příklady lokálních transformací aplikovaných na vzdálenostní pole.

transformace (lokální, globální), které nám objemovou reprezentaci upraví. Pro převod objemové reprezentace budeme používat algoritmus Marching tetrahedra.

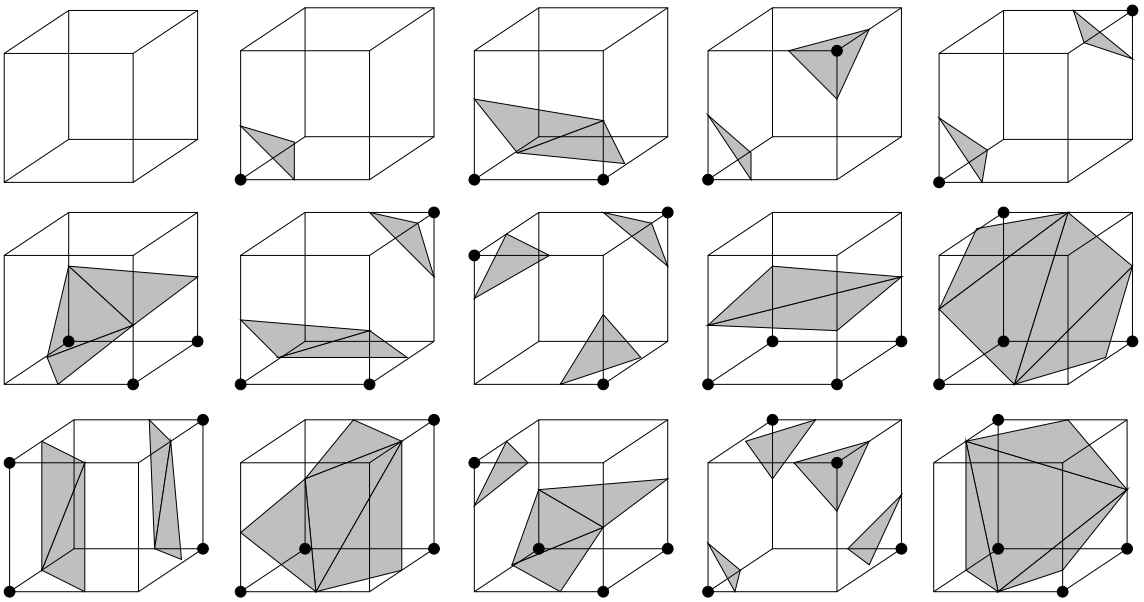
3.6.1 Převod do povrchové reprezentace

V dnešní počítačové grafice se nejčastěji setkáme s povrchovou reprezentací těles. Důvod je ten, že současný hardware je pro to optimalizovaný. Občas je však vhodné specifikovat objekty objemem nebo pomocí implicitních ploch. Příkladem je trojrozměrný šum, reprezentující hustotu objektu v daném místě prostoru. Proto potřebujeme takto reprezentované objekty převést do povrchové reprezentace. Existuje několik algoritmů pro převod. Jedním z nejznámějších je Marching cubes. Další používaný algoritmus je Marching tetrahedra.

3.6.2 Marching cubes

Marching cubes algoritmus popsán v [9] je v češtině algoritmus pochodujících krychlí. Krychle stejné velikosti jsou pravidelně rozmístěny v prostoru, kde se nachází objekt, který chceme převést. Abychom mohli algoritmus použít, musíme mít k dispozici funkci: $f : \mathbb{R}^3 \rightarrow \{0, 1\}$. Funkce zjistí, zda bod na daných souřadnicích leží uvnitř objektu (funkce vrací 1) nebo vně (funkce vrací 0). Pro jednu krychli vyhodnotíme tuto funkci pro všechny její rohy. Získáme jednu z 2^8 možných ohodnocení rohů. Existuje patnáct unikátních ohodnocení (konfigurací), které můžeme transformacemi převést na všechny ostatní.

Na hranách krychle se vyskytuje vrchol geometrie (trojúhelníku) v případě, kdy ohodnocení rohů krychle, které hrana sdílí, není stejné (na jednom rohu je 0 a na druhém 1). Vznikne tak několik hran, na kterých leží vrcholy trojúhelníků. Vhodným propojením vrcholů vznikne trojúhelníková síť pro jednu konfiguraci krychle. Patnáct trojúhelníkových sítí pro patnáct unikátních konfigurací můžeme vidět na obrázku 3.13. Spojením všech



Obrázek 3.13: Unikátní konfigurace, obrázek převzat z [2]

trojúhelníkových sítí ze všech krychlí rozmístěných v prostoru vznikne výsledná síť.

Marching cubes algoritmus má jednu nevýhodu. Některé konfigurace krychlí spolu nejsou kompatibilní podle [9]. Pokud bychom nekompatibilní stavy neřešili, ve výsledné trojúhelníkové síti by vznikly díry. Obslužný kód by nám zbytečně zvětšil aplikaci, a tak není algoritmus Marching cubes v této podobě v práci použit.

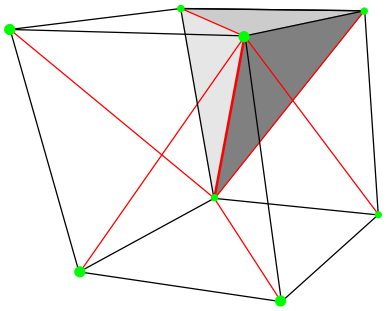
3.6.3 Marching tetrahedra

Marching tetrahedra algoritmus (kráčející čtyřstěn) už netrpí problémem nekompatibilních konfigurací jako Marching cubes. Nicméně produkuje větší množství geometrie. Stejně jako u předcházejícího algoritmu potřebujeme funkci, která rozhodne, zda daný bod leží uvnitř nebo vně objektu. Čtyřstěn má čtyři rohy, proto je celkový počet konfigurací roven 2^4. Maximální počet trojúhelníků v jedné konfiguraci je 2. Vzhledem k těmto malým číslům si můžeme jednotlivé konfigurace vhodně uložit do konstantních dat programu.

Problém tohoto algoritmu spočívá v rozmístění čtyřstěnů v prostoru s objektem. Pokud bychom použili čtyřstěn, který má všechny hrany stejně dlouhé, rozmístění v prostoru by nebylo úplně snadné (tak jako u krychle u předešlého algoritmu). Algoritmus na rozmístění čtyřstěnu by nám zbytečně obsadil místo v programu. Proto rozmístíme šest čtyřstěnů do krychle. Tuto krychli budeme pravidelně rozmísťovat do prostoru stejně jako u algoritmu Marching cubes. Rozmístění čtyřstěnů v krychli můžeme vidět na obrázku 3.14. Pro vrcholy krychle spočteme, zda leží uvnitř či vně objektu. Poté pro všech šest čtyřstěnů vybereme z tabulky správnou trojúhelníkovou síť.

3.6.4 Vyhlazení

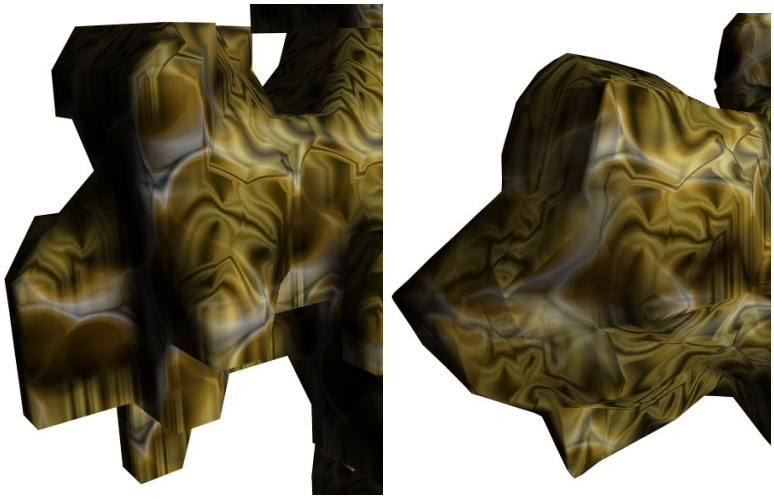
Pokud bychom u algoritmů uvedených výše umísťovali vrcholy trojúhelníků přímo doprostřed hrany na které leží, výsledná trojúhelníková síť by byla hranatá. Hranatou trojúhelníkovou síť můžeme vidět v levé části obrázku 3.15. Vhodným posunutím vrcholů na



Obrázek 3.14: Rozmístění čtyřstěnů do krychle

hranách můžeme dosáhnout vyhlazení (pravá část obrázku 3.15. Body na hranách můžeme posunou podle toho, jak daleko jsou konce hrany od předpokládaného povrchu.

Pokud použijeme objemovou reprezentaci (například šum), hodnota v prostoru nám určuje hustotu. Hodnotu hustoty budeme brát v rozsahu $\langle 0, 1\rangle$. Když hodnota hustoty na daných souřadnicích překročí hodnotu prahu T, nachází se v místech souřadnic těleso. Hustotu v bodech na souřadnicích S_a, S_b na koncích hrany označme H_a, H_b. Zároveň platí podmínka, že právě jeden bod konce hrany leží v tělese:$(H_a \geq T \wedge H_b < T) \vee (H_a < T \wedge H_b \geq T)$. Souřadnice vrcholu na hraně jsou dány vztahem: $(1-t)\cdot S_a + t \cdot S_b$. Parametr t je vypočítán podle vztahu: $t = \frac{T-H_a}{H_b-H_a}$.



Obrázek 3.15: Marching tetrahedra. Vlevo: bez vyhlazení. Vpravo: s vyhlazením.

3.6.5 Zmenšení počtu vrcholů

Výše uvedené algoritmy produkují mnoho geometrie. Pokud budeme data ukládat do společného pole, nebudeme mít informaci o tom, který vrchol je sdílen vícero trojúhelníky. Daný bod na hraně jedné krychle je vygenerován i jinou sousední krychlí a je uložen do společného pole několikrát. Navíc toto způsobuje problém při výpočtu normál. Proto je vhodné vrcholy, které jsou duplicitní uložit do pole jen jednou. Toto lze zajistit jednoduše. Budeme vrcholy do společného pole vkládat postupně. Vždy porovnáme souřadnice právě vkládaného vrcholu se všemi vrcholy, které jsou již uloženy.

Toto řešení má nevýhodu a tou je rychlost. Pro velké množství vrcholů je potřeba provést velké množství porovnání. Proto budeme vrcholy ukládat do stromu. Pokud budeme chtít zjistit, zda byl již daný bod uložen, stačí prohledat jen malou část stromu. Toto algoritmus urychlí.

3.6.6 Normály

Normály se v počítačové grafice používají pro stínování objektů a práci se světlem. Můžeme si ukládat normálu ke každé plošce. Normála je v tomto případě stejná pro všechny body plošky (polygonu). Tímto dosáhneme jednolitého stínování. Jednolité stínování můžeme použít pro ploché objekty jako jsou stěny, podlaha nebo například bedny. Další možností je ukládat si normálu ke každému vrcholu polygonu. Poté budeme osvětlení počítat pro všechny vrcholy polygonu a uvnitř osvětlení interpolovat. Vizuálně lepší výsledky ale dostaneme, pokud budeme interpolovat normály. Pro každý bod polygonu tak budeme mít k dispozici interpolovanou normálu, kterou použijeme pro výpočet osvětlení. Díky interpolaci normály nepotřebujeme tolik polygonů (trojúhelníků) pro reprezentaci hladkého objektu. Výpočet normály pro vrchol, který je sdílen k trojúhelníky s normálami a obsahy $\overline{n}_i, s_i, i \in \{1, \ldots, k\}$ je dán vztahem:

$$n = nor\left(\sum_{i=1}^{k} \overline{n}_i \cdot s_i\right) \tag{3.4}$$

Funkce *nor* normalizuje vstupní vektor. Můžeme se obejít bez výpočtu obsahu trojúhelníků a vztah 3.4 zjednodušit na:

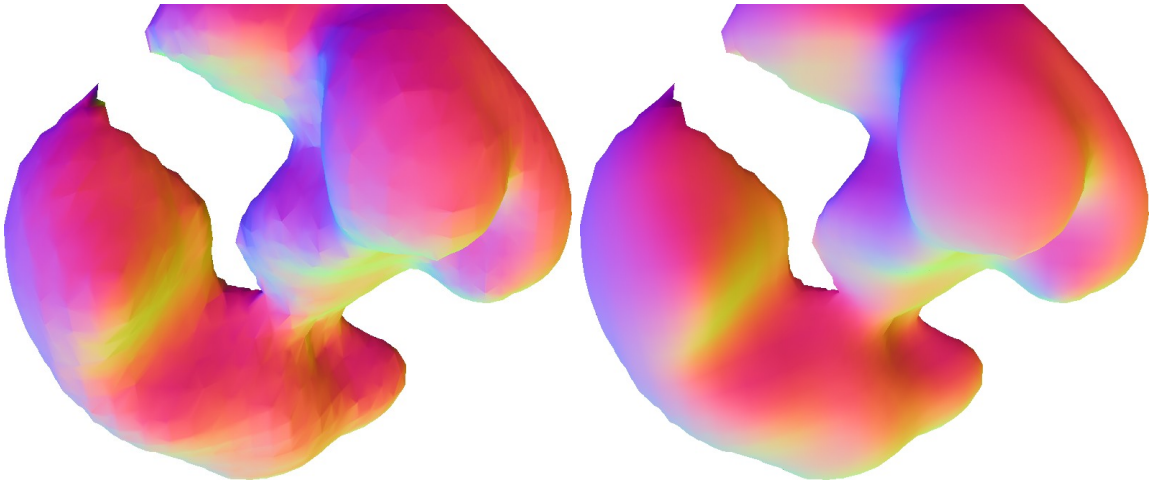
$$n = nor\left(\sum_{i=1}^{k} \overline{n}_i\right) \tag{3.5}$$

Problém nastane, pokud jsou v objektu velmi úzké trojúhelníky. U velmi úzkých trojúhelníků je hlavní část vyhlazení normál zastoupena v malé oblasti. Povrch objektu je sice vystínován plynule, ale z větší vzdálenosti se již tak nejeví. V levé části obrázku 3.16 můžeme pozorovat hranice trojúhelníků, které sousedí s velmi úzkými trojúhelníky. Algoritmus marching tetrahedra bohužel produkuje tyto úzké trojúhelníky pokud je zapnuto vyhlazování. Buď budeme mít stejně velké trojúhelníky a vizuálně hladkou interpolaci normál, ale hranatý objekt nebo hladký objekt a viditelné hrany trojúhelníků. Řešení můžeme vidět v algoritmu, který by pro daný vrchol nepočítal normály jen z trojúhelníků, které tento vrchol sdílí, ale z většího okolí. Implementováním tohoto řešení bychom ale zbytečně zabrali místo v aplikaci.

Elegantní řešení spočívá v použití gradientu (směru růstu) trojrozměrného šumu, který jsme použili pro vygenerování geometrie, postup je blíže popsán v [5]. Normály reprezentované pomocí gradientu můžeme vidět v pravé části obrázku 3.16. Gradient trojrozměrného šumu si můžeme uložit do trojrozměrné textury a využít jej také pro počítání kolizí. Výpočet gradientu v bodě o souřadnicích $I = (i_1, i_2, i_3)$ získáme pomocí vztahu:

$$g = (h(I + x) - h(I - x), h(I + y) - h(I - y), h(I + z) - h(I - z)) \tag{3.6}$$

Ve vztahu 3.6 reprezentuje x respektive y respektive z trojici $(1, 0, 0)$ respektive $(0, 1, 0)$ respektive $(0, 0, 1)$. Funkce $h(I)$ vrací hodnotu šumu na souřadnicích I. Výpočet normály z gradientu je jednoduchý stačí gradient znormalizovat.

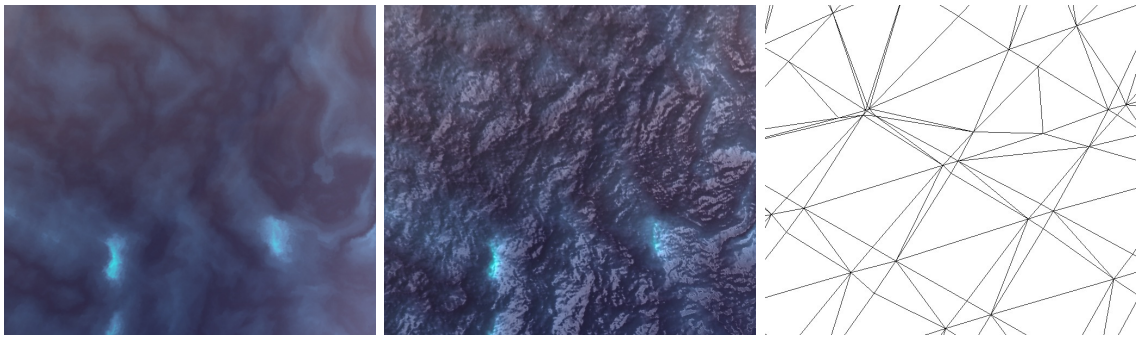


Obrázek 3.16: Zobrazení normál. Vlevo: normály pro vrcholy trojúhelníků. Vpravo: normály reprezentované gradientem.

3.7 Bump mapping

Bump mapping je metoda pro zvýšení detailu povrchu. V sekci normály 3.6.6 jsme si popsali interpolaci normály v bodě trojúhelníku. Tímto přístupem získáme hladký povrch. Zvýšení úrovně detailu bychom mohli dosáhnout zvětšením počtu polygonů. Pokud bychom ale chtěli mít vyšší detaily uvnitř trojúhelníku, bez přidání dalších trojúhelníků, můžeme zvolit bump mapping. Bump mapping ovlivňuje normálu v bodě trojúhelníku a tím i výsledné stínování. Pro ovlivňování normály se používá bump mapa.

Bump mapa je textura, která místo barvy nese informaci o normále. Pokud bump mapu naneseme na trojúhelník a budeme pomocí ní ovlivňovat interpolovanou normálu, získáme vizuálně detailnější stínování. Výsledek bump mappingu je znázorněn na obrázku 3.17. Bump mapa je tříkanálová textura. Místo barev (r, g, b) obsahuje složky normály (u, v, w). Bump mapu budeme vytvářet z textury těsně před obarvením. Texturu v odstínech šedé budeme brát jako výškovou mapu a normála v daném bodě je na ní kolmá.



Obrázek 3.17: Vlevo: Stínování s vyhlazením normál. Uprostřed: stínování s aplikovaným bump mappingem. Vpravo: trojúhelníková síť.

3.8 Texturování

Postup vytvoření textur jsme si popsali v sekci 3.5. Nyní nastává otázka, jak texturu nanést na povrch. Pokud použijeme trojrozměrnou texturu je nanesení jednoduché. Souřadnice vrcholů polygonů jsou zároveň i souřadnice do textury (texturovací koordináty). Texturovací koordináty pro trojrozměrnou texturu můžeme snadno transformovat pomocí transformační matice. Texturu můžeme roztahovat, otáčet a posouvat. Navíc můžeme objekt, na který je nanesena trojrozměrná textura různě deformovat, vytvářet do něj díry nebo přidávat polygony. Nemusíme se přitom starat o přepočet koordinát. Nevýhoda trojrozměrné textury spočívá v její velikosti. Povrch objektu probíhá jen zlomkem celkového objemu textury. Z velikosti plyne i doba na vygenerování takové textury. Další nevýhoda spočívá v nesnadné konstrukci bump mapy. Povrch objektu může bodem textury procházet různě natočen, proto bychom potřebovali pro každé natočení jinou bump mapu. U dvojrozměrných textur je výpočet bump mapy jednoduší. Dvojrozměrné textury mají pouze dvě složky texturovacích koordinát, proto je nutné je vhodně rozmístit po povrchu objektu. Rozmístění texturovacích koordinát ale není jednoduché. Obvykle bývají rozmísťovány ručně pomocí grafika. Algoritmus pro rozmístění koordinát na netriviální objekt by byl příliš složitý a zabíral by místo. Proto budeme textury na trojrozměrné objekty nanášet jinak.

3.8.1 Triplanární texturování

Triplanární mapování textur popsané v [5] používá pro obarvení objektu tři různé textury. Textury jsou nanášené podél tří souřadných osy (x, y, z). Pokud je normála povrchu v daném bodě (x, y, z) například $(1, 0, 0)$, použije se textura pro osu x. Souřadnice pro tuto texturu budou poté (y, z). Přičemž složka y představuje u složku texturovací koordináty (směr v ose x v textuře). Pro normálu $(0, 1, 0)$ se použije textura pro osu y. Texturovací koordináty budou v takovém případě $(u, v) = (x, z)$. Pokud normála povrchu není rovnoběžná s jednou ze souřadných os, použije se vícero textur, které se mezi sebou smíchají. Uvažujme bod povrchu objektu $P = (x, y, z)$ s normálou $N = (a, b, c)$ a tři textury T_x, T_y, T_z pro tři souřadné osy. Dále uvažujme funkci $f : T \times R^2 \to \langle 0, 1\rangle^3$. Funkce f vrací pro texturu T a souřadnice reprezentované dvojicí barvu. Výpočet barvy je pak dán následujícím vztahem:

$$C = \frac{|a|}{S} \cdot f(T_x, (y, z)) + \frac{|b|}{S} \cdot f(T_y, (x, z)) + \frac{|c|}{S} \cdot f(T_z, (x, y)) \tag{3.7}$$

Ve vztahu 3.7 je $S = |a| + |b| + |c|$ normalizace vah. Složky a, b, c normály udávají váhy pro vážený průměr.

Triplanární texturování můžeme modifikovat. Místo abychom požívali tři textury pro tři souřadné osy, můžeme použít šest a rozlišovat směr osy. Pro otexturování jeskyně a přívodního tunelu ale zvolíme jiný postup. Použijeme také tři textury, ale ne v souřadných osách. Jednu texturu použijeme na stropy (kladná osa y). Další na zem (záporná osa y). Poslední textura je mapována na stěny. Vztah pro výpočet výsledné barvy pro takto rozmístěné textury je obdobný vztahu 3.7.

3.9 Skybox

Skybox je vzdálené okolí od scény, které nereprezentujeme pomocí polygonů, ale jen pomocí obrázků. Skybox můžeme použít pro reprezentaci mraků, vzdálených krajů nebo třeba hvězd ve vesmíru. Obvykle bývá skybox ve formě krychle. Na všechny stěny krychle je

nanesen jiný obrázek. Obrázky na sebe navazují a tvoří tak dojem okolí. Příklad rozvinutého krychlového skyboxu můžeme vidět na obrázku: 3.18. Skybox můžeme vytvořit fotoaparátem a programem na skládání fotografií. Stačí vyfotit okolí ve všech směrech a fotografie následně složit dohromady. V našem případě si ale musíme skybox vygenerovat. V práci budeme používat dvě metody generování skyboxu.

Obrázek 3.18: Skybox zobrazující dopolední oblohu

První je obdobná tvorbě skyboxu pomocí fotoaparátu. Z jednoho vybraného místa si scénu vykreslíme do šesti směrů a výsledky si uložíme. Zorný uhel kamery nastavíme na 90 stupňů a šířku a výšku vykreslené oblasti zvolíme totožnou. Tímto dokážeme vytvořit jednoduché skyboxy. V práci budeme takto vytvořený skybox používat pro odraz na hladině vody. Na složitější skybox (například oblohy) tato metoda ale nestačí.

Druhá metoda vykresluje stěny skyboxu zároveň. Metodu budeme využívat pro oblohy. Ze středu krychle se pro zvolený pixel a zvolenou stěnu krychle určí paprsek. Pro velikost obrázků na stěnách krychle (w, h) potřebujeme $6 \cdot w \cdot h$ paprsků. Paprsek můžeme reprezentovat pouze normalizovaným vektorem u. Tento vektor si můžeme představit jako jistou třísložkovou souřadnici. Pokud budeme chtít vykreslit do skyboxu jistý element, předáme mu tento vektor a element nám vrátí barvu (r, g, b, a). Číslo a je průhlednost v intervalu $\langle 0, 1 \rangle$ a používá se pro míchání nově příchozí barvy $c = (r, g, b)$ s předcházející $cp = (r_p, g_p, b_p)$. Míchání je řízeno vztahem:

$$cp = a \cdot c + (1 - a) \cdot cp \tag{3.8}$$

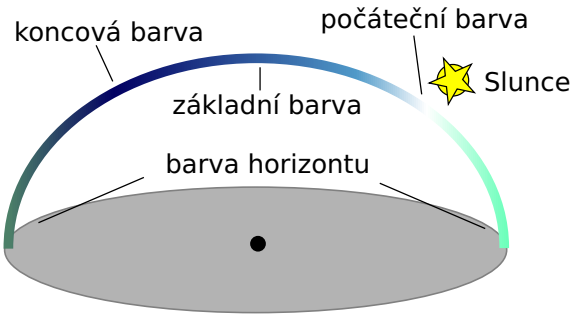
Algoritmus generování skyboxu obohy je následující:

1. Nastavíme všechny barvy obrázků na nulu $cp = (0, 0, 0)$.
2. Pro všechny stěny krychle skyboxu a všechny jejich pixely určíme normalizovaný vektor u.
3. Pro všechny elementy $e_i, i \in 1, \ldots, n$, které chceme vykreslit získáme barvu c_i.
4. Výsledná barva pro vektor u (a tedy i pro konkrétní pixel) je cp_n.

$$\begin{aligned} cp_1 &= a_1 \cdot c_1 + (1 - a_1) \cdot cp \\ cp_i &= a_i \cdot c_i + (1 - a_i) \cdot cp_{i-1} \end{aligned}$$

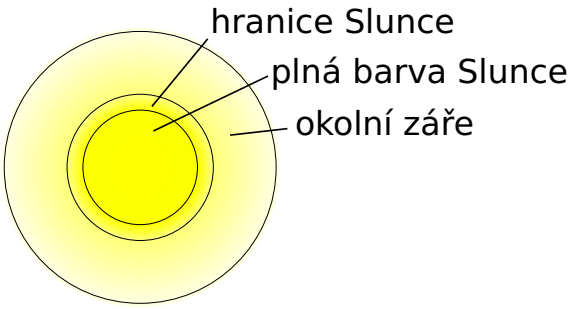
Oblohu si budeme reprezentovat pomocí tři základní elementů:

- Barevný přechod oblohy. Barevný přechod oblohy udává barvu pozadí. Udává barvu atmosféry při východu slunce, západu slunce nebo v průběhu odpoledne. Barevný přechod je určen barvou horizontu, počáteční, koncovou a základní barvou, vektorem ke slunci a exponentem. Počáteční barva oblohy je barva blízko slunce, koncová barva oblohy je barva na opačném konci oblohy od slunce. Základní barva je barva mezi počáteční a koncovou. Barva horizontu obarvuje oblohu blízko nad zemí. Rozmístění barev je dobře patrné na obrázku 3.19 Exponent udává rychlost změny barvy horizontu do barvy oblohy.



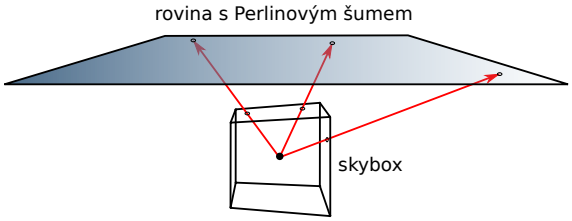
Obrázek 3.19: Rozmístění barev oblohy.

- Slunce. Slunce je určeno vektorem pozice, velikostí, barvou, exponentem a velikostí okolní záře. Velikost slunce je ve steradiánech. Exponent udává, jak ostrá je hranice slunce. Velikost okolní záře udává, do jaké vzdálenost od slunce se šíří barva slunce obrázek: 3.20.
- Mraky. Mraky jsou reprezentované nekonečnou rovinou, která je umístěna nad scénou, obrázek: 3.21. Směrový vektor u pro daný pixel skyboxu protne tuto rovinu v jistých souřadnicích $I(x, y)$. Souřadnice I použijeme jako vstup funkce Perlinova šumu popsaného v podsekci 3.1.2. Parametry mraků jsou tedy: Amplituda, frekvence, perzistence a počet sčítání. Tyto parametry jsou předány funkci Perlinova šumu. Další parametry jsou hustota, krytí a barva mraků. Pro věrnější reprezentaci mraků budeme obvykle potřebovat vícero vrstev mraků. Problém teto metody spočívá v aliasing efektu na horizontu. Průsečík směrového vektoru s rovinou mraků je pro pixely na horizontu velmi daleko. Jednotlivé souřadnice pro Perlinův šum pro pixely, které leží vedle sebe v této oblasti jsou velmi odlišné. Tím nám vzniká problém, že barva

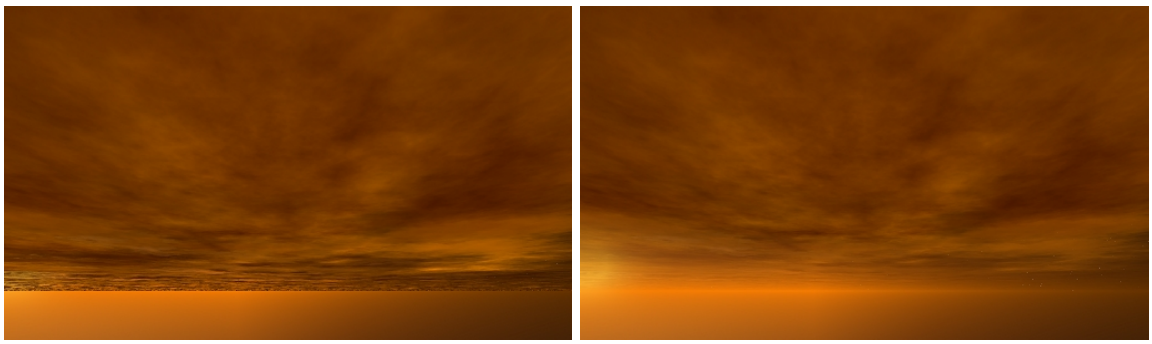


Obrázek 3.20: Hranice slunce a okolní záře.

mraků na horizontu se prudce střídá v sousedních pixelech. Řešení by mohlo spočívat v použití jiného než Perlinova šumu. Naším řešením ale bude zvyšování průhlednosti mraků směrem k horizontu. Mraky se tak rozplynou v barvě horizontu, obrázek: 3.22.



Obrázek 3.21: Vykreslování mraků.



Obrázek 3.22: Vlevo: mrak bez opravy aliasing efektu. Vpravo: mrak s rostoucí průhledností k horizontu.

3.10 Částicové systémy

Částicové systémy se v počítačové grafice používají k nejrůznějším účelům. Pomocí částicových systémů reprezentujeme fyzikální jevy, které by se jinou cestou špatně vizualizovaly nebo simulovaly. Jedním z příkladů může být simulace sněžení. Další příklady jsou písek, ohňostroj nebo voda. Částicový systém je složen s velkého množství samostatných částic. Každá částice se chová samostatně podle určitých pravidel. Částice mají svoji pozici a

Obrázek 3.23: Skybox zobrazující západ slunce

rychlost. Tyto parametry slouží k popisu pohybu. Rovnice pro popis pohybu jsou 3.10, 3.12 a 3.13 a blíže si je popíšeme v sekci 3.11 o elastických systémech.

Částice budeme vykreslovat jako plošky s nanesenou texturou. Budeme rozlišovat dva druhy. Jedny částice se neustále natáčejí ke kameře. Tento druh částic budeme používat pro vykreslení vodopádu, bublinek ve vodě a listů liánové rostliny. Druhý druh si udržuje svoji orientaci. V intru jej budeme používat pro vykreslení chomáče rostlin. Mimo pozice a rychlosti má u sebe částice také čas. Pokud čas částice překročí určitou hodnotu, částice zanikne. U částicového systému si budeme definovat i emitor. V emitoru se vytváří částice. Pokud částice zanikne objeví se nová s počátečním nastavením v emitoru.

3.11 Elastické systémy

Elastické systémy slouží k simulaci těles jako je látka, tkanina, lana, ale i rosol. Elastický objekt mění svůj tvar v průběhu času. Proto je výpočet jeho pohybu odlišný od výpočtu dynamiky tuhého tělesa. Pohyb tuhého tělesa si můžeme rozložit na posuvnou rychlost a rotační rychlost. Pokud na takové těleso působíme silou, jsou body tělesa přímo ovlivňovány. Elastické těleso je složeno z bodů a vazeb, přičemž každý bod má vlastní výpočet dynamiky. Pokud tedy na elastické těleso působíme silou, účinky síly jsou postupně distribuovány pomocí vazeb do bodů.

Elastický systém si můžeme představit jako speciální obdobu částicového systému. Částice (uzly) představují vrcholy objektu, vrcholy trojúhelníků. Každá částice má svoji

hmotnost, pozici a rychlost. Oproti částicovým systémům obsahuje částice i seznam vazeb, které ovlivňují její pohyb. Vazba je spoj mezi dvěma částicemi. Nese informaci o své počáteční délce a své tuhosti. Dále obsahuje informaci, které dva uzly spojuje. Elastický systém je tedy složen ze dvou základních druhů elementů: uzlů a spojů.

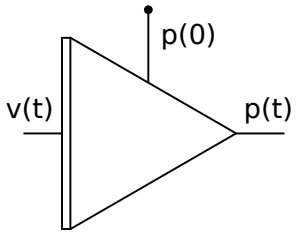
Pohyb elastického systému si popíšeme diferenciálními rovnicemi. Každá částice má svoji pozici p a rychlost v. Pro výpočet nové pozice $p(t + dt)$ potřebujeme rychlost $v(t)$ a krok času dt. Pro výpočet nové rychlosti $v(t + dt)$ potřebujeme zrychlení $a(t)$ a krok času dt. Diferenciální rovnice pro výpočet pozice:

$$\frac{\partial p}{\partial t} = v \tag{3.9}$$

Rovnici 3.9 zintegrujeme a dostaneme tvar:

$$p(t) = p(0) + \int_0^t v(t)dt \tag{3.10}$$

V rovnici 3.10 je $p(0)$ počáteční pozice částice v čase $t = 0$. $p(t)$ respektive $v(t)$ je pozice respektive rychlost v čase t. Rovnici 3.10 si můžeme vyjádřit schématem s integrátorem na obrázku 3.24. Diferenciální rovnice pro výpočet rychlosti:



Obrázek 3.24: Integrátor reprezentující rovnici 3.10

$$\frac{\partial v}{\partial t} = a \tag{3.11}$$

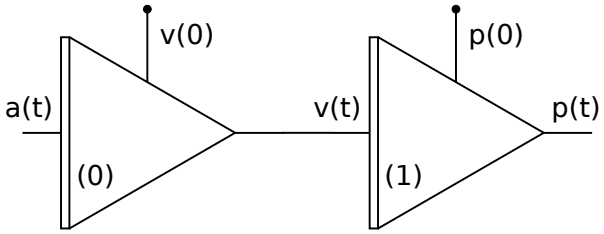
Rovnici 3.11 zintegrujeme a výsledný tvar je:

$$v(t) = v(0) + \int_0^t a(t)dt \tag{3.12}$$

$a(t)$ v rovnici 3.12 reprezentuje zrychlení, $v(0)$ počáteční rychlost a $v(t)$ rychlost v čase t. Schéma pro rovnici 3.12 je obdobné schématu 3.24. Schémata můžeme spojit v jedno, které je znázorněné na obrázku 3.25 Jelikož vytváříme elastický systém v trojrozměrném prostoru je zrychlení, rychlost a pozice třísložkový vektor. Schéma 3.25 nám již dokáže vypočítat pozici jedné částice v čase t. Zbývá zjisti zrychlení, tedy vstup integrátoru (0). Zrychlení částice se odvíjí od druhého Newtonova zákonu o působení síly na hmotné těleso:

$$a = \frac{1}{m} \cdot F \tag{3.13}$$

Nyní již víme, jak se částice (uzel elastického systému) zachová, pokud na ni zapůsobíme silou. Síla je složena z gravitační síly $F_g = m \cdot g$, která je pro daný uzel konstantní a síly



Obrázek 3.25: Schéma představující spojení rovnic 3.10 a 3.12

vyvolané spoji daného uzlu. Spoj mezi dvěma uzly si můžeme představit jako pružinu. Sílu, kterou pružina působí na dva uzly si vyjádříme vztahem:

$$F = -k \cdot x \tag{3.14}$$

Ve vztahu 3.14 je k tuhost pružiny (nebo spoje) a $x = l_0 - l$ je výchylka pružiny z rovnovážného stavu. l_0 je počáteční délka spoje a l je aktuální délka spoje v čase t. Absolutní velikost síly vyvolané spoji na uzel i je:

$$|F_i| = \sum_{j=1}^{n} -k_{i,j} \cdot (l_{0i,j} - |p_i p_j|) \cdot s_{i,j} \tag{3.15}$$

n je počet uzlů. $l_{0i,j}$ je počáteční délka spoje mezi uzly i, j. p_i, p_j jsou pozice uzlů a $s_{i,j} = 1$ pokud je mezi uzly i, j spoj. Jinak je $s_{i,j} = 0$. Stejně jako pro pozici a rychlost, je i síla třísložkový vektor. Označme $e_{i,j} = (p_j - p_i)/|p_i p_j|$ normalizovaný vektor spoje z uzlu i do j. Výsledná síla působící na uzel i je:

$$F_i = \sum_{j=1}^{n} -k_{i,j} \cdot (l_{0i,j} - |p_i p_j|) \cdot s_{i,j} \cdot e_{i,j} = \sum_{j=1}^{n} -s_{i,j} \cdot k_{i,j} \left(\frac{l_{0i,j}}{|p_i p_j|} - 1 \right) (p_j - p_i) \tag{3.16}$$

Nyní již známe vše, abychom mohli vytvořit schéma obecného elastického systému. Schéma je znázorněné na obrázku 3.26

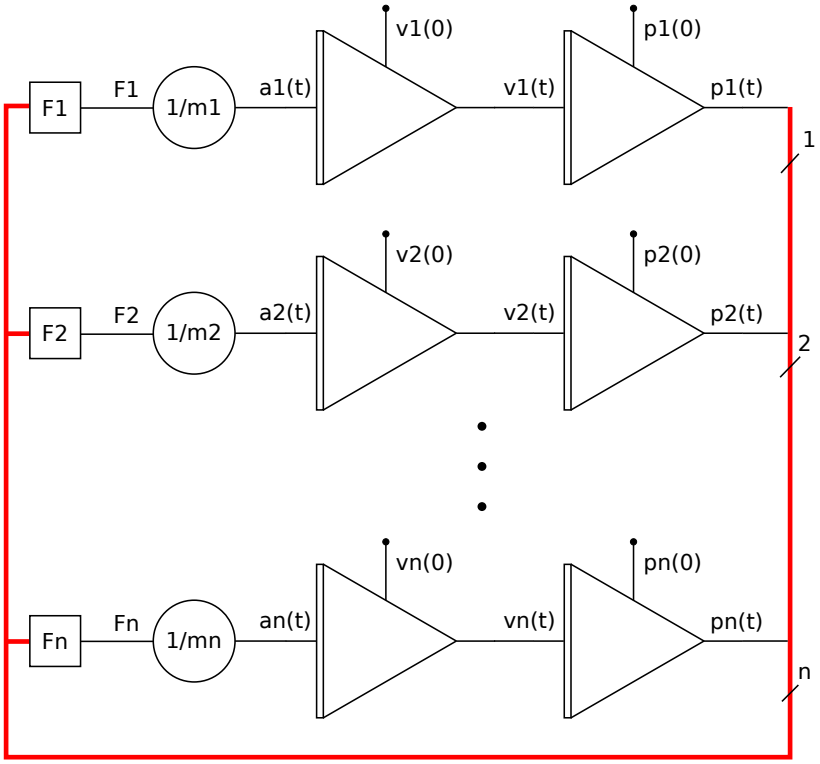
Diferenciální rovnice musíme vyřešit pomocí numerických metod. Existuje několik druhů numerických metod. Nejznámější je Eulerova metoda. Přesnost numerických metod závisí na velikosti kroku času dt a stupně metody.

3.11.1 Eulerova metoda

Eulerova metoda je jednoduchá numerická metoda pro řešení diferenciálních rovnic. Je dána vztahem:

$$\begin{aligned} y_0 &= y(0) \\ y_{n+1} &= y_n + dt \cdot f(t_n, y_n) \end{aligned}$$

$f(t_n, y_n)$ je aproximace derivace y'_n. V našem případě je to vstup integrátorů ve schématu 3.26. Eulerova metoda je rychlá na výpočet a jednoduchá na implementaci. Proto může být vhodná pro intra. Její nevýhodou je malá přesnost. Proto je nutné volit menší krok času než u jiných metod.



Obrázek 3.26: Schéma obecného elastického systému. Jednotky $F_1 - F_n$ počítají sílu podle vztahu 3.16

3.11.2 Runge Kutta metoda

Runge Kutta metoda čtvrtého stupně je dána vztahem:

$$\begin{aligned} y_0 &= y(0) \\ y_{n+1} &= y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \\ k_1 &= dtf(t_n, y_n) \\ k_2 &= dtf(t_n + dt/2, y_n + k_1/2) \\ k_3 &= dtf(t_n + dt/2, y_n + k_2/2) \\ k_4 &= dtf(t_n + dt, y_n + k_3) \end{aligned}$$

Runge Kutta metoda je obecně přesnější než Eulerova metoda. Je ale výpočetně náročnější. Pro výpočet následující hodnoty integrátoru je zapotřebí čtyřikrát vyčíslit jeho vstup.

3.11.3 Stabilita numerických metod

Numerické metody nemusí být stabilní. Pro některé rovnice se rozkmitají. Náš elastický systém se pro nevhodně zvolené koeficienty rozkmitá a následně rozpadne. Musíme správně zvolit koeficienty tuhosti spojů a hmotnosti uzlů. Velký vliv na stabilitu má krok času dt. Elastický systém je tuhý systém, pokud jsou hmotnosti uzlů malé nebo pokud jsou tuhosti spojů velké. Tato skutečnost vyplývá z rovnic 3.13 a 3.14. Pokud obě rovnice spojíme získáme koeficient $c = k/m$, kde k je tuhost spoje a m hmotnost uzlu. Tento koeficient přibližně popisuje zesílení zpětné vazby vyznačené červenou barvou v obrázku 3.26.

V programu jsem nejprve naimplementoval Eulerovu numerickou metodu. Její největší nevýhoda byla nestabilita. Pokud jsem v průběhu animace zvyšoval tuhosti spojů, v jednom okamžiku se elastický systém choval dle očekávání. Pokud ale tuhost přerostla určitý práh, systém se velice rychle rozkmital a rozpadl a animaci jsem musel pusti znovu. Proto jsem se rozhodl zkusit implemetovat Runge Kutta čtvrtého řádu. U Runge Kutta metody elastický systém reaguje na nevhodně zvolené tuhosti a hmotnosti pomaleji. Proto je možné se ještě z rozkmitaného stavu vrátit zpět. Testoval jsem také rychlost. Věděl jsem, že u Runge Kutta metody potřebuji vyčíslit čtyři koeficienty. Výpočet jednoho kroku je tak přibližně čtyřikrát náročnější než u metody Eulerovy. Experimentálně jsem si srovnal výsledky pro stejný elastický systém s Eulerovou metodou pro krok dt a Runge Kutta metodou pro krok $4dt$. Sledoval jsem, při jakém nastavení tuhosti spojů se systém rozkmitá. Výsledky pro metodu Runge Kutta byly mírně lepší než pro Eulerovu metodu, proto jsem se rozhodl jej ve výsledné aplikaci využívat. Také velmi záleželo na tvaru elastického systému.

3.12 Kolize

Bez kolizí se neobejde téměř žádná fyzikální simulace. Například částicový systém díky kolizím interaguje s okolím. Kolize bývají složité na výpočet a proto je potřeba algoritmy zefektivňovat. Jedním z vylepšení algoritmů detekcí kolizí jsou obalová tělesa. Dalším vylepšením může být hierarchické rozdělení scény.

V intru jsou kolize počítány jen pro částicové a elastické systémy. Tyto systémy kolidují s geometrií jeskyně a tím se odráží od jejího povrchu. Pro jednoduchost budeme kolize počítat jen pro body. Abychom zjistili, zda je bod v kolizi s jeskyní, musíme zjistit, se kterým trojúhelníkem bod koliduje. Tento přístup by nebyl příliš rychlý a implementace algoritmu by zabrala příliš místa. Místo toho použijeme jednoduchý způsob. Souřadnice bodu, pro který počítáme kolizi, můžeme použít jako index do volumetrické reprezentace jeskyně. Pokud hodnota na souřadnicích bodu překročí určitý práh (stejný práh je použit i pro algoritmus Marching Tetrahedra) je v daném místě kolize.

Nejprve jsem navrhl obecný algoritmus, který dokázal získat hodnotu pro neceločíselné souřadnice z d dimenzionálního pole. Tento algoritmus byl rekurzivní, relativně malý a díky své obecnosti znovupoužitelný. Nicméně nebyl příliš rychlý. Proto jsem navrhl algoritmus, který je rychlejší, ale pracuje jen pro trojrozměrné pole. $P_0 = (p_0, p_1, p_2)$ je bod, pro který počítáme kolizi. Nejprve přesuneme bod do rozsahu $r = (r_0, r_1, r_2)$ (šířka, výška, délka) podle vztahu: $P_1 = ((P_0 \% r) + r) \% r$. Operace modulo % pracuje po složkách vektoru. Poté získáme nejvyšší celočíselný index $I = (i_0, i_1, i_2) = \lfloor P_1 \rfloor$, který je menší než P_1. Rozdíl $v = (v_0, v_1, v_2) = P_1 - I$ udává poměr míchání hodnot na indexech o jedničku vyšších než I v některé ose. Hodnota $h(P_0)$ trojrozměrného pole v bodě P_0 je dána vztahem:

$$\begin{aligned}
h_0 &= (1 - v_0)h(I) + v_0 h(I_x) \\
h_1 &= (1 - v_0)h(I_y) + v_0 h(I_{yx}) \\
h_2 &= (1 - v_0)h(I_z) + v_0 h(I_{zx}) \\
h_3 &= (1 - v_0)h(I_{zy}) + v_0 h(I_{zyx}) \\
h_{01} &= (1 - v_1)h_0 + v_1 h_1 \\
h_{23} &= (1 - v_1)h_2 + v_1 h_3 \\
h(P_0) &= (1 - v_2)h_{01} + v_2 h_{23}
\end{aligned}$$

Dolní sufix indexu I udává, ke které složce indexu I je přičtena jednička. Například pro

index $I_{zx} = (i_0 + 1, i_1, i_2 + 1)$.

Abychom mohli správně reagovat na kolizi, musíme znát i normálu povrchu. Normálu povrchu máme uloženou také v trojrozměrném poli, proto můžeme použít stejný algoritmus. Reakce na kolizi spočívá v přesunutí bodu na povrch jeskyně a otočení jeho rychlosti podle normály. Úhel dopadu se přitom musí rovnat úhlu odrazu. Výslednou rychlost ještě zmenšíme o ztráty. Ovlivnění rychlosti u částicových systému je přímočaré. U elastických systémů si ale musíme dát pozor. Ovlivňujeme rychlosti uzlů externě, mimo výpočetní model elastického systému. Toto může potenciálně vést ke vzniku nestability.

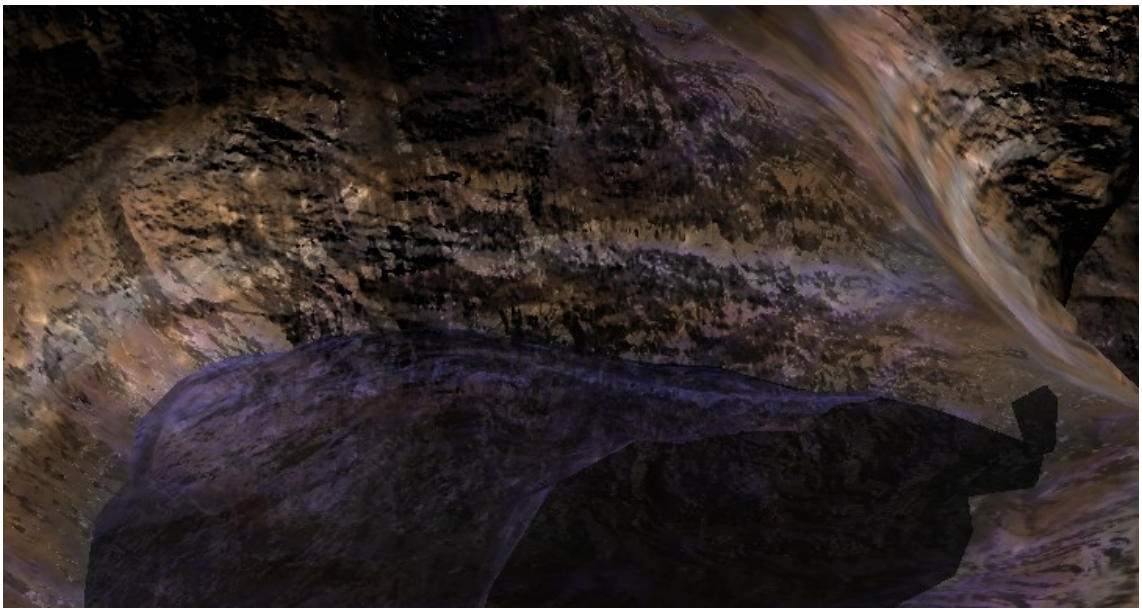
3.13 Voda

Voda se v intru vyskytuje ve dvou formách. V podobě vodní hladiny a v podobě tekoucí vody představující vodopády. Vodní hladina je v intru použita pro reprezentaci moře v části s přímořskou oblastí. Dále pak pro vizualizaci jezírek v přívodním tunelu do jeskyně a v jeskyni. Tekoucí voda se vyskytuje až v poslední části intra - v jeskyni. V jeskyni je několik míst, kde vyvěrá voda. Voda stéká po stěnách, kterou zvlhčuje.

3.13.1 Vodní hladina

Vodní hladina je reprezentována prostým čtvercem. Leskne a odráží se na ni okolí. Odraz na vodní hladině lze vytvořit několik způsoby.

Jedním z nejjednodušších způsobů je zrcadlově převrátit scénu a vykreslit ji znovu. Musíme si dát pozor na to, abychom vykreslovali obraz jen v oblasti vodní plochy. Můžeme toho dosáhnou pomocí stencil testu. Dále musíme vykreslovat jen odraz. Nesmí se stát, aby odraz "vylézal"z vodní plochy. Příklad takto vytvořeného odrazu je vidět na obrázku 3.27. Výhoda metody spočívá v jednoduché implementaci. Nevýhoda je nemožnost přidat na



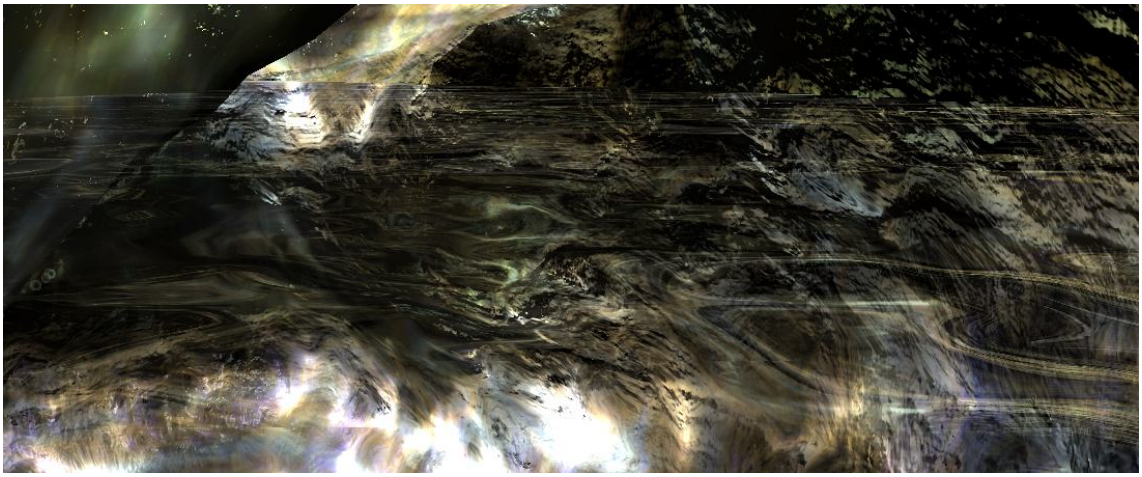
Obrázek 3.27: Jednoduchý odraz na vodní hladině vytvořený pomocí překlopení scény.

hladinu vlnění. Vlnění by se muselo přidávat dodatečně, jinou metodou. Další nevýhoda je

nutnost rasterizace odrazu. Pokud se v hladině odráží celá scéna může nám výkon klesnou i na polovinu.

Jiným způsobem, jak vizualizovat odraz, je vytvořit kolem vodní plochy skybox. Na tomto skyboxu je vykresleno okolí, které se na vodní hladině odráží. Paprsek od kamery se podle normály vodní hladiny odrazí a směruje do určitého místa skyboxu. Tímto získáme barvu. Jelikož pracujeme s fragmenty a ne s celou scénou, lze poměrně snadno vytvořit na hladině vlnění. Pokud vytvoříme skybox při inicializaci a necháme jej statický, ušetří nám to výkon. Nemusíme scénu překreslovat pro vizualizaci odrazu. Nevýhoda metody se skyboxem je, že se na hladině nemůže odrážet pohyblivý předmět. Další nevýhoda spočívá v nepřesnosti odrazu. Skybox si vytvoříme jen pro jeden konkrétní bod hladiny. Správně bychom si jej měli vytvořit pro všechny body hladiny. Čím vzdálenější je zkoumaný bod na hladině od bodu, kolem kterého je vytvořen skybox, tím je odraz zkreslenější. Pokud ale hladinu vody zvlníme, nemusí být tento nedostatek patrný. Metodu se skyboxem budeme v intru používat.

Efekt vlnění na hladině je vytvořen pomocí ovlivňování normály. Podobně jako u bump mappingu je na hladinu namapována bump mapa. Bump mapa nám lokálně ovlivňuje normálu, proto se povrch zdá zakřivený. Abychom hladinu rozpohybovali, potřebovali bychom sekvenci bump map, které na sebe navazují v čase. Sekvenci bump map si můžeme představit jako trojrozměrnou texturu. Trojrozměrná textura obsahuje vrstvy bump map v z souřadnici.



Obrázek 3.28: Vlnění odrazu na hladině s použitím skyboxu s průhledností.

3.13.2 Tekoucí voda

Tekoucí voda je částicový systém. Textura částic je v pravé části obrázku 4.4. Jediný rozdíl je v tom, že textura není zelená ale bílá. Tekoucí voda se vyskytuje v podobě vodopádu v poslední scéně intra.

3.14 Terén

Terén budeme v intru používat ve dvou scénách: hory a přímořská oblast. Geometrie terénu je vytvořena z mřížky trojúhelníků. Body trojúhelníku na ose y reprezentující výšku hor

vychýlíme pomocí výškové mapy. Výšková mapa je dvojrozměrné pole, kde hodnoty v prvcích neudávají barvu ale výšku. Pro vygenerování výškové mapy můžeme použít algoritmy popsané v sekci 3.5 o generování textur. Velikost pole určuje počet trojúhelníků. Pro pole velikost $w \times h$ je potřeba $2 \cdot (w-1) \cdot (h-1)$.

3.15 Pohyb kamery

Aby bylo intro dynamická, potřebujeme v ní pohyb. Jedním ze způsobů, jak toho můžeme dosáhnout je pohybovat s kamerou. Správný pohyb kamery může vytvořit i z nezajímavé scény akční podívanou. Kamera v průběhu času sleduje jistou trajektorii. Jelikož má intro omezenou velikost, nemůžeme si uložit pro každý krok času, pozici kamery. Místo toho, si budeme ukládat jen klíčové body a místa mezi nimi budeme vypočítávat z okolních klíčových bodů.

Jeden klíčový bod si můžeme představit jako deset čísel $K = (p, z, y, a)$. Vektor $p = (p_x, p_y, p_z)$ reprezentuje pozici kamery. Vektor $z = (z_x, z_y, z_z)$ reprezentuje pohledový vektor neboli směr, kam je kamera nasměrována. Vektor $y = (y_x, y_y, y_z)$ je vektor určující směr nahoru. Tento vektor udává natočení kamery podél osy z. Číslo a udává zorný úhel. Sekvence klíčových bodů definuje pohyb kamery.

Mezi klíčovými body budeme interpolovat. Pro interpolaci použijeme Catmull-Rom interpolaci popsanou v [8]. Pro výpočet interpolace z hodnoty v_1 do hodnoty v_2 používá metoda také hodnoty v_0, v_3.

$$v(t) = [t^0, t^1, t^2, t^3] \cdot \frac{1}{2} \cdot \begin{bmatrix} 0 & 2 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 2 & -5 & 4 & -1 \\ -1 & 3 & -3 & 1 \end{bmatrix} \cdot \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{bmatrix} = C(t) \cdot \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{bmatrix} \tag{3.17}$$

Hodnoty v_0, v_1, v_2, v_3 jsou klíčové hodnoty. Hodnota $v(t)$ je interpolovaná hodnota mezi v_1, v_2. Parametr t může nabývat hodnot $t \in \langle 0, 1 \rangle$. Pokud je parametr $t = 0$ je hodnota $v(t) = v_1$. Pokud je parametr $t = 1$ je hodnota $v(t) = v_2$. Funkce $C : \langle 0, 1 \rangle \to \mathbb{R}^4$ vrací vektor vah pro parametr t.

Pokud máme více než čtyři klíčové hodnoty, interpolujeme po částech. Například máme n klíčových hodnot $v_1, v_2, \ldots, v_n$. Celková délka je T_{max}. Pokud budeme potřebovat hodnoty v_{1-a} respektive v_{n+a}, $a \in \mathbb{N}$, použijeme v_1 respektive v_n. Výsledná hodnota v místě t je dána podle vztahu:

$$v(t) = C\left((n-1)\frac{t}{T_{max}} - i\right) \cdot \begin{bmatrix} v_i \\ v_{i+1} \\ v_{i+2} \\ v_{i+3} \end{bmatrix} \tag{3.18}$$

Index $i \in \{0, 1, \ldots, n-2\}$ udává segment, ve kterém se interpoluje z klíčových hodnot. Jeho hodnota je:

$$i = \left\lfloor \frac{t}{T_{max}} \cdot (n-1) \right\rfloor \tag{3.19}$$

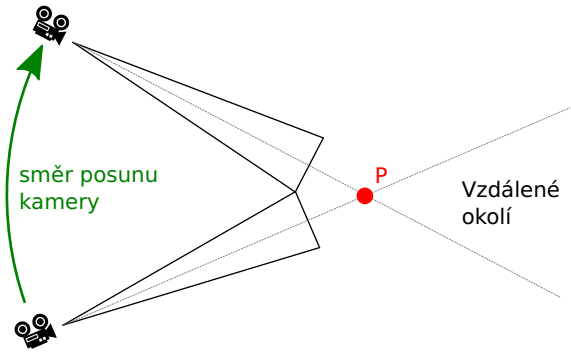
Interpolaci budeme provádět pro všech deset čísel klíčového bodu. Získáme tak pro čas t přesný popis kamery.

3.15.1 Efekty kamery

Pokud již máme vyřešený pohyb kamery, je vhodné jej dobře použít. V intru můžeme použít několik efektů, které vidíme například ve filmech. Prvním z nich je efekt, kdy se kamera přibližuje k objektu a její zorný úhel se zvětšuje. Situace je znázorněna na obrázku 3.29. Objekt by se důsledkem zmenšení vzdálenosti od kamery měl zvětšovat. Pokud budeme ale zároveň zvětšovat zorný úhel, zůstane stejně velký. Okolí kolem objektu se ale bude roztahovat do šíře. Obdobný efekt lze vytvořit obráceně. Místo abychom se k objektu přibližovali, budeme se vzdalovat a zorný úhel zmenšovat. Tento efekt je oblíbený mezi filmaři. Efekt bývá využíván v chodbách, tunelech a podobných protáhlých prostorách, kde je zvýrazněn.

Obrázek 3.29: Efekt kamery s využitím zorného úhlu. Kamera se posouvá zleva doprava. Její zorný úhel se přitom zvětšuje $a_1 < a_2$

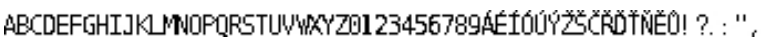
Další efekt je rovněž hojně používaný ve filmech. Kamera se zaměří na jeden objekt, přiblíží na něj pomocí zmenšení zorného úhlu a rotuje kolem něj. Efekt je znázorněn obrázkem 3.30. Objekt je snímán na stejném místě. Vzdálená krajina se ale rychle pohybuje na pozadí. Bývá používán v exteriérech, kdy význačný objekt stojí na vrcholku hory nebo kopce.



Obrázek 3.30: Efekt kamera kdy se kamera zaměří na jeden bod P a rotuje kolem něj.

3.16 Texty

Texty jsou v intru spíše doprovodné prvky. Font textu je uložen ve statických datech. Jedná se o malý bitový obrázek s vybranými znaky. Z obrázku vytvoříme čtverce, na které je namapována textura jednoho písmena. Font, který je v intru použit je zobrazen na obrázku 3.31



Obrázek 3.31: Písmenka fontu.

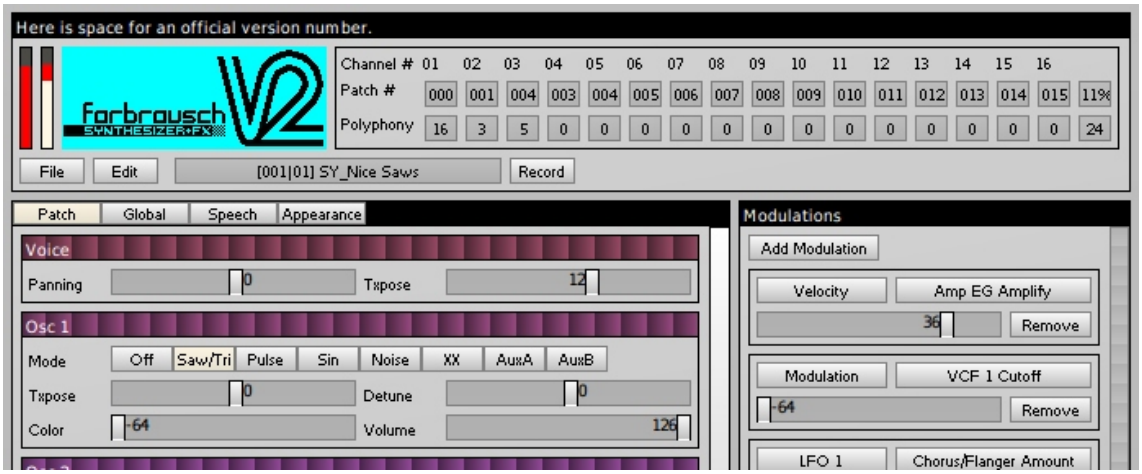
Kapitola 4

Implementace

Většinu projektu jsem implementoval pod systémem Ubuntu Linux. Pro vytvoření okna pod Linuxem je použita knihovna SDL. U systému Windows je to WINAPI. Pro spuštění programu je vyžadováno OpenGL minimálně ve verzi 3.3. Program jsem ladil pro Windows 7. Verze pro Linux neobsahuje hudbu a je také větší. Důvod je ten, že knihovna pro přehrávání hudby byla napsána pro systém Windows. Také komprimační program kkrunchy slouží pro komprimování binárních spustitelných souborů systému Windows.

4.1 Hudba

Pro přehrávání hudby v intru jsem použil knihovnu libv2 od německé skupiny Farbrausch [4]. Ke knihovně je dodáván i příklad přehrávače souborů v2m. Přehrávač byl napsán v jazyce C pro Visual Studio. Některé části kódu byly napsány v jazyce assembler a ty bylo nutné přepsat. Pro skládání hudby je ke knihovně přidán i VSTi plugin, zobrazen na obrázku 4.1. Hudbu jsem skládal v demo verzi programu FL Studio [7] a použil jsem tento VSTi plugin.



Obrázek 4.1: VSTi plugin pro tvorbu hudby pro knihovnu libv2.

4.2 Scény intra

Intro obsahuje čtyři scény. Horskou scénu, přímořskou scénu, tunel a jeskyni. Mezi scénami je plynule přepínáno pomocí zatmívaček. Zatmívačka je černá plocha. Tato plocha je kreslena s vypnutým hloubkovým testem. Ovlivňováním průhlednosti můžeme plynule přejít mezi scénami. V intru se vyskytují tyto objekty: hory, kopce, skybox, vodní hladina, vodopády, bublinky ve vodě, chomáče rostlin, liánové rostliny, bedny, tunel, jeskyně a zavěšené barevné koberce.

4.2.1 Horská scéna

Scéna s horskou scénou je na obrázku 4.2. Pro vytvoření terénu je použita výšková mapa. Výšková mapa je vytvořena ze vzdálenostního pole Voroného diagramů, pravá strana obrázku 3.4. Ve vzdálenostním poli se vyskytují špičky (horské štíty) a propoje (hřebeny). Na hory je namapována bump mapa, která vytváří skály. Barva hor je vytvořena dvojicí barevných přechodů. Jeden barevný přechod je použit stejným způsobem jako na obrázku 3.9. Tímto způsobem získáme barvu s nadmořskou výškou. Druhý barevný přechod je použit pro obarvení srázů. Výběr barvy z přechodu je stejný jako u prvního přechodu - pomocí výšky. Barva je ale přimíchávána pomocí průhlednosti. Průhlednost je určena podle strmosti srázu - podle normály $n = (u, v, w)$. Pokud je složka $|v| = 1$ je průhlednost maximální. Při $|v| = 0$ je průhlednost minimální. Kolem hor je skybox, který je zobrazen na obrázku 3.18. S rostoucí vzdáleností se hory noří do mlhy. Hustota mlhy je vyšší s menší nadmořskou výškou.

Obrázek 4.2: Horská scéna.

4.2.2 Přímořská scéna

Přímořská scéna zobrazena na obrázku 4.3 je vytvořena pomocí výškové mapy vytvořené ze šumu. Stejně jako u hor, jsou pro obarvení použity dva barevné přechody. Skybox kolem

přímořské oblasti můžeme vidět na obrázku 3.23.

Obrázek 4.3: Přímořská scéna.

4.2.3 Tunel, jeskyně

Tunel i jeskyně jsou vytvořeny stejným způsobem, pomocí algoritmu Marching Tetrahedra. Rozdíl spočívá ve vytvoření trojrozměrného pole volumetrické reprezentace. U jeskyně je způsob vytvoření pole jednoduchý. Vygenerujeme trojrozměrný šum pomocí algoritmu půlení intervalu. Šum poté vyhladíme pomocí globální transformace. Poté budeme hodnoty pole násobit koeficientem $k = \langle 0, 1 \rangle$. Hodnota koeficientu k klesá, pokud se vzdalujeme od středu krychle obsahující šum. Tímto zajistíme, aby se jeskyně uzavřela a neměla ve stěnách díry. Výslednou objemovou reprezentaci jeskyně převedeme pomocí algoritmu Marching Tetrahedra na trojúhelníky.

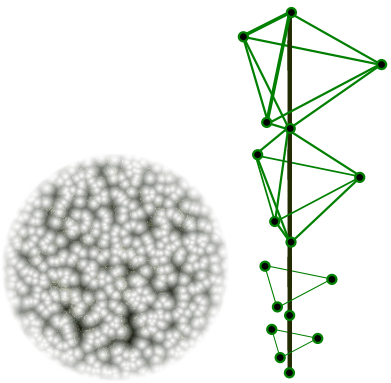
Objemová reprezentace tunelu je vytvořena jiným způsobem. Nejprve vytvoříme trojrozměrný šum stejně jako u jeskyně. Hodnoty šumu zmenšíme tak, aby nebyly větší než práh pro algoritmus Marching Tetrahedra. Pokud bychom teď provedli převod na trojúhelníky, žádný trojúhelník by nevznikl. Nyní do pole vykreslíme tunel. Tunel budeme vykreslovat po bodech. Body leží na křivce podobné jednomu vláknu blesku. Konce křivky umístíme na opačné rohy krychle šumu. Křivka je reprezentována dvěma jednorozměrnými šumy. Jeden ovlivňuje natočení kolem spojnice počátečního a koncového bodu křivky. Druhý ovlivňuje vzdálenost od spojnice. Tuto křivku vykreslíme do pole několikrát a vznikne nám tak tunel zobrazený na obrázku 4.6

Textury v tunelu jsou rozmístěny stejně jako v jeskyni. Jeskyně je obarvena pomocí několika textur: Textury stropu, textury stěn a textury země. Každá z těchto textur má k sobě i bump mapu. Další textura je jednorozměrná a je umístěna vertikálně. Představuje geologické vrstvy. Další textura je trojrozměrná. Je vytvořena pomocí distančního pole z Voroného diagramů a reprezentuje kaustiky. Textura je trojrozměrná, abychom mohli kaustiky animovat. Poslední textura je také trojrozměrná a je dynamicky měněna. Představuje vlhkost stěn jeskyně. Vodopády do ní zapisují hodnoty a textura samotná ovlivňuje

míru odlesků a tmavost povrchu.

V tunelu jsou umístěny chomáče rostlin. Jedná se o částicový systém, jehož částice jsou statické. Textura částic je zobrazena v levé části obrázku 4.4. Na konci tunelu je vodní hladina. V tunelu se vyskytuje i pavučina. Pavučina je složena z elastického systému a můžeme ji vidět uprostřed obrázku 4.6.

V jeskyni jsou zavěšeny liánové rosliny. Rostliny jsou složeny z elastického systému, jehož tvar je v pravé části obrázku 4.4. Jeden článek rostliny je složen ze dvou spojených čtyřstěnů. V bodech je umístěna textura rosliny zobrazena v levé části obrázku 4.4. Dalším objektem jeskyně je zavěšený koberec. Koberec je také složen z elastického systému ve tvaru mřížky. Na koberec je nanesena textura na obrázku 4.5



Obrázek 4.4: Vlevo: textura rostlin. Vpravo: rozložení uzlů a spojů elastického systému liánové rostliny.

Obrázek 4.5: Textura zavěšeného koberce.

4.3 GLSL

V intru jsem použil sedm shader programů. Tři z nich používají i geometry shader. Jsou to shader programy využívané částicovými systémy. O vytvoření částice (plošky) se stará geometry shader. Pro částicové systémy je vyhrazen vlastní shader program. Pomocí uniformu lze nastavit, jestli se částice natáčí ke kameře či udržuje svoji orientaci.

Shader program, který v intru používáme pro vykreslení jeskyně má nejsložitější část fragment shader. Jeskyně obsahuje poměrně malé množství trojúhelníků a přesto je nejsložitější na vykreslení. Snížením počtu trojúhelníků bychom složitost nesnížili. Jednou

Obrázek 4.6: Tunel.

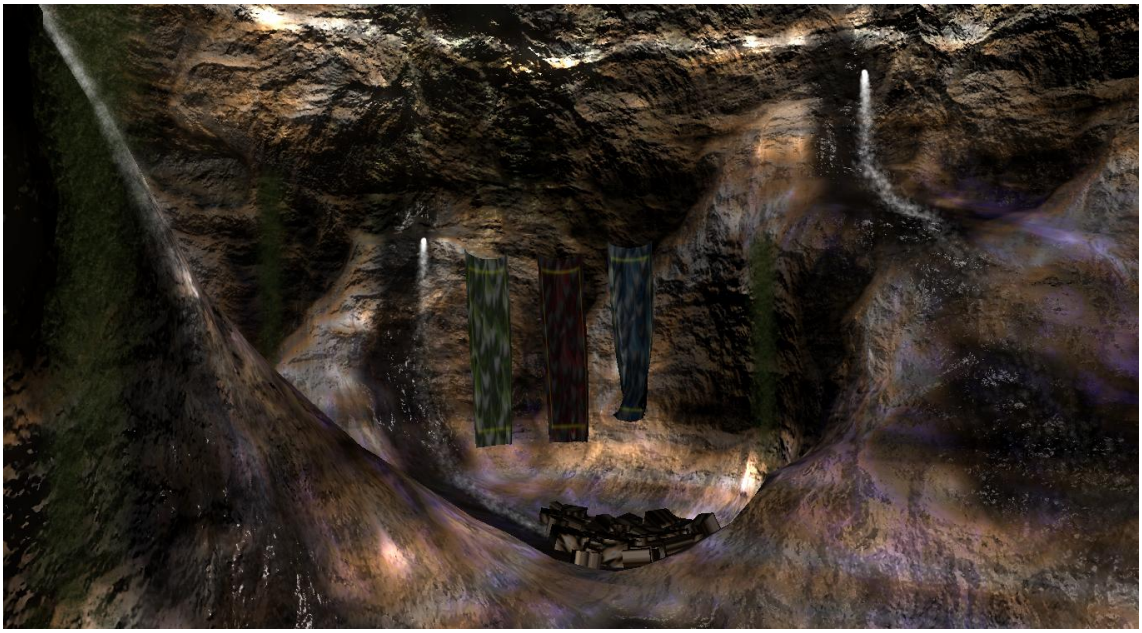
možností je optimalizovat fragment shader. Vhodnou optimalizací může být minimalizace větvení programu použitím vestavěných funkcí. Příklad větvení:

```
if (red_flag == 1){
  color = vec3(1,0,0);
}else{
  color = texture(wall,coord);
}
```

Výše uvedený úsek programu obsahuje větvení pomocí uniformu red_flag. Můžeme se větvení zbavit využitím vestavěné funkce mix.

```
color=mix(vec3(1,0,0),texture(wall,coord),red_flag);
```

Další funkce, které se dají použít pro odstranění větvení jsou $min, max, clip, clamp$ a další. Jinou možností je omezení používání množství normalizační funkce: $normalize$. Urychlení vykreslení scény se složitým fragment shaderem můžeme udělat i jinak. Urychlení spočívá v omezení množství fragmentů, které musíme vykreslit. Při vykreslování se stává, že jsou trojúhelníky kresleny ve špatném pořadí. Vykreslujeme některé fragmenty, které jsou později přepsány novými hodnotami. Řešení může existovat v algoritmu řazení trojúhelníků. Potom stačí vykreslovat od nejbližších trojúhelníků. V případě, že je nějaká část trojúhelníku zakryta, je vykreslování zastaveno hloubkovým testem. Řazení trojúhelníků je ale složité na výpočet a zbytečně by nám zabralo místo v programu. Jeskyně ale obsahuje málo trojúhelníků, proto vykreslíme nejprve hloubkový buffer s vypnutým zápisem barvy. Tím se přeskočí část s fragment shaderem. Poté vykreslíme scénu znovu, tentokrát již s povoleným zápisem barvy. Tímto zajistíme, že se výpočet fragment shaderu provádí jen pro nezbytně nutné množství fragmentů.



Obrázek 4.7: Jeskyně.

4.4 Rozdělení projektu a FPS

Zdrojové kódy přesahují 20 000 řádků kódu, proto je bylo potřeba kvůli přehlednosti rozdělit do logických celků. Projekt je rozdělen do velkého množství knihoven. Ve zdrojových kódech pro Linux jsou knihovny rozděleny do několika složek:

app V této složce jsou funkce pro obsluhu vytvoření okna a časování.

- **winwindow** Knihovna obsahuje funkce pro vytvoření okna, nastavení obslužných funkcí. Dále obsahuje řízení časem. Knihovna je preprocesorem rozdělena do dvou částí. Jedna část je pro systém Windows 7 a obsahuje kód ve WINAPI. Druhá část je pro systém Linux a využívá funkcí knihovny SDL.
- **main** Obsahuje hlavní část programu, většinu inicializací a krokování. Obsahuje kreslící funkci.

adt Složka obsahuje abstraktní datové typy.

- **list2** Obsahuje obecný dvousměrný seznam.
- **relist** Reprezentuje obecné pole s automatickým zvětšování velikost. Oproti **list2** je rychlejší čtení, ale pomalejší vkládání doprostřed pole.
- **ntree** Představuje obecný strom a obslužné funkce.
- **adtfce, adt** Obsahují rozhraní a funkce společné pro abstraktní datové typy.

elastic Složka obsahuje pouze jednu knihovnu, která implementuje elastický systéme a jeho obslužné funkce

enviroment Ve složce nalezneme knihovny pro grafické objekty intra.

- **box_swarm** Knihovna obsahuje funkce pro inicializaci, umístění, vykreslení a přepočítání haldy beden.

bubble Knihovna obsahuje částicový systém reprezentující bublinky ve vodě.

carpet Objekt zavěšeného koberce je obsahem této knihovny.

cave Obsahuje funkce pro vygenerování jeskyně.

collide Obsahuje kolizní systém.

fade Zatmívačka, která slouží pro přechod mezi scénami

font Obsahuje funkce a font pro vykreslení textu.

geometry V knihovně můžeme najít funkce pro výpočet normál a výpočet spojů pro elastický systém.

lake Vodní hladina a obslužné funkce jsou obsaženy v této knihovně.

moss Obsahem knihovny je objekt mechu nebo křoví.

mountain, waterside Obsahují funkce pro vytvoření výškových map hor a přímořských kopců.

object Soubory knihovny reprezentují obecný objekt, který využívá elastický systém.

particle Nalezneme zde obecný částicový systém, který je využíván například v knihovně **bubble**.

plant Rostlina složená z elastického systému.

skybox Obsahuje funkce pro vykreslení skyboxu.

skyboxgenerate Vytvoření skyboxu je součást funkcí v souborech této knihovny.

spiderweb Knihovna reprezentuje pavučinu.

terrain Vykreslení a vytvoření geometrie terénu pomocí výškových map.

tunnel Obdobně jako **cave** obsahuje knihovna funkce pro vygenerování tunelu.

gen Složka obsahuje některé obecné algoritmy pro generování.

color Obsahuje funkce pro práci s HSV barevným modelem.

map, colormap Funkce pro práci s barevným přechodem.

midpoint Algoritmus pro generování šumu pomocí půlení intervalu.

voronoi Generování Voroného diagramů.

gpu Složka obsahuje knihovny pro komunikaci s grafickou kartou.

gpuattribute Funkce pro práci s atributy shader programu.

gpubuffer Obsahuje funkce pro správu bufferů na grafické kartě.

gpushaderprogram Obsahuje obslužné funkce pro shader programy.

gputexture Inicializace a správa textur.

gputextureunit Obsluha texturovacich jednotek.

index Knihovny pro práci s vícerozměrnými daty.

index Knihovna obsahuje obecný d dimenzionální index.

nsize Rozměry d dimenzionálních dat.

marchingtetra Složka obsahuje algoritmus Marching tetrahedra.

mt_core Algoritmus Marching tetrahedra.

pack Algoritmus pro spojování společných vrcholů trojúhelníků.

music Složka obsahuje přehrávač a hudební soubory.

music Funkce pro inicializace a přehrání hudby.

songs Hudební skladby.

v2mplayer Přehrávač hudby.

mymath Složka obsahuje matematické knihovny.

camera Kamera scény.

cameracontrol Systém pro pohyb kamery.

vector, matrix Operace s vektory a maticemi.

stdmath Základní matematické vztahy.

transform Transformace scény.

mymem Složka obsahuje knihovnu pro práci s pamětí a zastřešuje rozdíly mezi systémy Windows a Linux.

shaderprogram Obsahem složky jsou vertex, geometry a fragment shadery.

std Základní funkce. Funkce pro generování náhodného čísla. Funkce pro zobrazení grafu načítání.

texturefactory Složka obsahuje knihovny pro generování textur.

colorbuffer Obsahuje funkce pro práci s d rozměrnými poli pro vytvoření textur.

convolution Obsahuje d rozměrnou konvoluci.

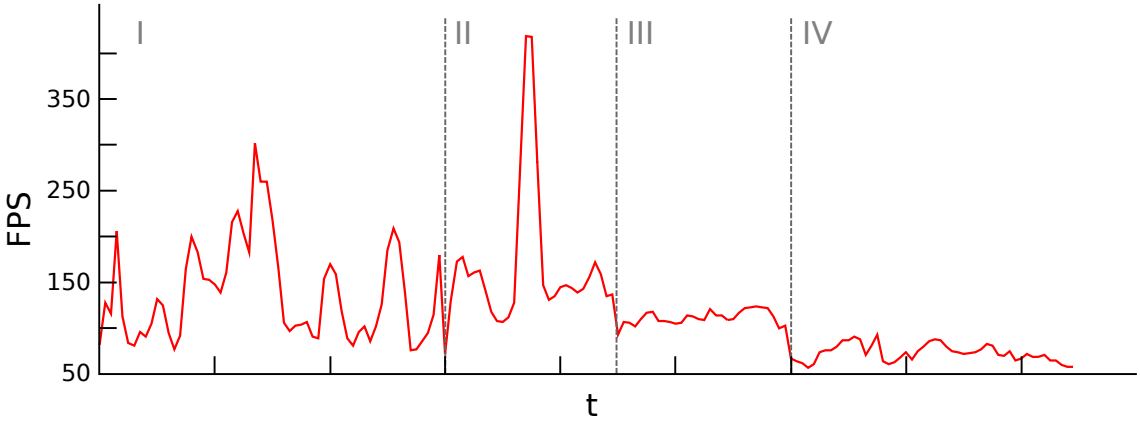
fastgetvalue Obsahuje obtimalizované funkce pro přístup k d rozměrným polím.

globaltransform Obsahuje globální transformační funkce pro tvorbu textur.

localtransform Obsahuje lokální transformační funkce pro tvorbu textur.

4.4.1 FPS

V grafu zobrazeném na obrázku 4.8 můžeme vidět průběh počtu snímku za sekundu (FPS) v čase. V prvních dvou částech grafu FPS hodně kolísá. V ostatních částech FPS udržuje přibližně konstantní hodnotu. První dvě scény zobrazují horskou a přímořskou krajinu. Každá z těchto scén je složena z 130 050 trojúhelníků. Záleží proto na pozici a záběru kamery, jak velká bude hodnota FPS. Lokální maxima poukazují na místa v intru, kde kamera zabírá jen zlomek scény. Příkladem může být špička v druhé scéně. V přímořské oblasti se kamera chvíli dívá jen na skybox. To vysvětluje velký nárůst FPS. V posledních dvou scénách: tunelu a jeskyně je řádově méně trojúhelníků. Proto tolik nezávisí na záběru kamery. První dvě scény jsou náročnější z pohledu množství trojúhelníků. Poslední dvě scény pak z pohledu shader programu, který je složitější.



Obrázek 4.8: Průběh počtu snímku za sekundu v čase. Graf je rozdělen do čtyř částí podle scén.

Kapitola 5

Závěr

Část textu jsem převzal s drobnými úpravami ze semestrálního projektu. Jedná se o části až po sekci Generování geometrie 3.6. Text semestrálního projektu prošel úpravami a byl přepsán do systému Latex. Celková velikost intra nepřesáhla 64kB. Intro obsahuje 4 scény. Každá z nich je ozvučená hudbou, kterou jsem složil.

V průběhu projektu mne napadlo velké množství rozšíření a vylepšení. Některé jsem implementoval a na některé nezbyl čas. Možným rozšířením intra by mohlo být přidání statických a dynamických stínů. Jiné rozšíření by spočívalo v implementaci SSAO (*screen space ambient occlusion*). Další možností by mohlo být generování geometrie v průběhu animace. Naimplementovat Marching Tertahedra algoritmus v shader programu OpenGL. Naimplementovat jiné šumy. Příkladem může být Simplex Noise. Tento šum implementovat v shader programu. Vhodná by mohla být implementace šumu, který netrpí aliasing efektem. Připojit dynamiku rigidních těles a spojit ji s dynamikou elastických těles. Přidat složitější detekci kolizí. Převést výpočet elastických systému do OpenCL a tím jej urychlit. Přesunout časování do zvláštního vlákna. Vyzkoušet další numerické metody pro řešení diferenciálních rovnic. Například implicitní Eulerovu metodu nebo metodu s využitím Taylorova rozvoje. Rozšíření projektu z jiného úhlu pohledu by mohlo spočívat v automatizovaném generování hudby. Vytvoření algoritmů pro generování zvuků. Přidání živých organizmů s umělou inteligencí: ryby ve vodě, pavouk, poletující mušky... Místo rozšiřování intra by bylo možné některé vlastnosti oddělit a vyvíjet je jako samostatný projekt. Napadlo mě několik takových projektů. Efektivní generování textur v shader programu. Fyzikální engine. Prozkoumání d dimenzionální alternativy algoritmu Marching Tetrahedra.

V průběhu projektu jsem se naučil hodně nových věcí a vyzkoušel několik různých alternativních řešení problému. Příkladem nechť je algoritmus Marching tetrahedra, kterým jsem nahradil Marching cubes. Další příkladem je implementace numerické metody Runge Kutta, která nahradila Eulerovu metodu. Implementoval jsem dobrý základ pro generování textur, který je dostatečně obecný, aby byl znovupoužitelný. Implementoval jsem generování Voroného diagramu a šumu obecné dimenze. Vytvořil jsem funkce, které dokáži s d dimenzionálními daty pracovat. Vytvořil jsem obecný elastický a částicový systém. Rozšířil jsem si svoje znalosti jazyka GLSL a správy velkého projektu jako takového. Během práce na projektu jsem řešil řadu problémů v podobě překlepů nebo použití nevhodné metody. Největší problém ale byl ukončit tvůrčí činnost v určitém bodě a projekt dokončit. Napadalo mne hodně vylepšení a rozšíření a bylo těžké se rozhodnout je již do projektu nezahrnovat. Přes všechny problémy jsem se při vyvíjení projektu bavil a co je nejdůležitější - rozšířil jsem si znalosti.

Literatura

[1] Aurenhammer, F.: *Voronoi Diagrams – A survey of a Fundamental Geometric Data Structure*. ACM Computing Surveys, 1991.

[2] Chernyaev, E. V.: *Marching Cubes 33: Construction of Topologically Correct Isosurfaces*. Institute for High Energy Physics, 1995.

[3] Elias, H.: Perlin Noise. http://freespace.virgin.net/hugo.elias/models/m_perlin.htm.

[4] Farbrausch: V2 synthesizer system. http://1337haxorz.de/products.html.

[5] Geiss, R.; Thompson, M.: *Cascades*. NVIDIA Corporation, 2006.

[6] Giesen, F.: kkrunchy. http://www.farbrausch.de/~fg/kkrunchy/.

[7] Image-Line: FL Studio. http://www.image-line.com/index.html.

[8] Joy, K. I.: *Catmull-Rom splines*. 2002.

[9] Lewiner, T.; Lopes, H.; Vieira, A. W.; aj.: *Efficient implementation of Marching Cubes' cases with topological guarantees*. Laboratório MatMídia, 2003.

[10] Mateas, M.; Montfort, N.: *A Box, Darkly: Obfuscation, Weird Languages, and Code Aesthetics. Proceedings of the 6th Digital Arts and Culture Conference*. IT University of Copenhagen, 2005.

[11] Milet, T.: *Grafické intro 64KB s použitím OpenGL [bakalářská práce]*. FIT VUT, 2010.

[12] Molnár, L.; Reiser, J. F.: Ultimate Packer for eXecutables. http://upx.sourceforge.net/.

[13] Perlin, K.: Making Noise. http://www.noisemachine.com/talk1/index.html.

[14] Scott, J.: Making Cellular Textures. http://www.blackpawn.com/texts/cellular/default.html.