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

NÁSTROJ PRO NÁVRH PROCEDURÁLNÍCH TEXTUR

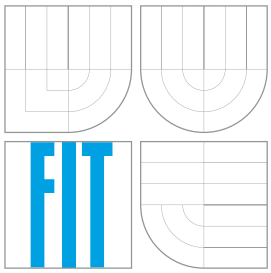
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MILOSLAV ČÍŽ

BRNO 2014

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

NÁSTROJ PRO NÁVRH PROCEDURÁLNÍCH TEXTUR

TOOL FOR DESIGNING PROCEDURAL TEXTURES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MILOSLAV ČÍŽ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. TOMÁŠ MILET

BRNO 2014

Abstrakt

Tato práce se zabývá implementací programové knihovny pro návrh a generování dvourozměrných procedurálních textur a podpůrného GUI nástroje pro tuto knihovnu. Pozornost byla věnována jak klasickým metodám, tak i přístupům z jiných oblastí počítačové grafiky se snahou o modifikaci algoritmů za účelem dosažení originálních výsledků. Dále je prezentován efektivní způsob reprezentace textur a jeho konkrétní realizace.

Abstract

This thesis deals with implementation of a program library for designing and generating two-dimensional procedural textures and its GUI supportive tool. Attention was paid to common methods as well as to other computer graphics areas with effort for modifying the algorithms in order to achieve unique results. An effective way of texture representation and its concrete implementation is also presented.

Klíčová slova

textury, procedurální, Perlinův šum, Fault Formation, Voroného diagram, L-systémy, částicové systémy, celulární automaty, grafický filtr, k-d strom, želví grafika, Substrate, osvětlení, teselace, graf, QT, XML, C, C++, OOP, grafika

Keywords

textures, procedural, Perlin noise, Fault Formation, Voronoi diagram, L-systems, particle systems, cellular automata, graphic filter, k-d tree, turtle graphics, Substrate, illumination, tessellation, graph, QT, XML, C, C++, OOP, graphics

Citace

Miloslav Číž: Nástroj pro návrh procedurálních textur, bakalářská práce, Brno, FIT VUT v Brně, 2014

Nástroj pro návrh procedurálních textur

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Tomáše Mileta.

.....
Miloslav Číž
12. května 2014

Poděkování

Děkuji vedoucímu práce Ing. Tomáši Miletovi za pravidelně poskytované konzultace, nápady, rady a materiály, díky nimž tato práce mohla vzniknout.

© Miloslav Číž, 2014.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **2**

2 Procedurální textury v počítačové grafice **3**
2.1 Základní postup generování textur 3

3 Použité metody **5**
3.1 Šum 5
3.2 Voroného diagramy 7
3.3 Obecné bitmapové operace 10
3.4 Modulace šumem 15
3.5 Substrate algoritmus 15
3.6 Obrazové transformace 15
3.7 Celulární automaty 17
3.8 Teselace 19
3.9 Částicové systémy 21
3.10 Osvětlovací modely a normálové mapy 22
3.11 L-systémy 24

4 Implementace **27**
4.1 Knihovna 27
4.2 Grafický nástroj 30

5 Závěr **33**

A Vygenerované textury **37**

B Ukázka kódu **38**

Kapitola 1

Úvod

Textury hrají v počítačové grafice významnou roli, především při snaze zachytit reálný svět. Tendence dosahovat realističtějšího zobrazení vede ke stále větším a detailnějším texturám nesoucím čím dál větší množství informací. Zabývat se texturami má z těchto důvodů smysl, ať už jde o jejich syntézu, popis, či efektivitu uchovávání.

Cílem této práce bylo nastudovat a analyzovat dosavadní poznatky ve zmíněné oblasti a na jejich základě implementovat multiplatformní knihovnu umožňující snadný popis a vytváření navazujících dvourozměrných procedurálních textur a rovněž nástroj s grafickým uživatelským rozhraním pro podporu práce s knihovnou. Knihovna měla nabídnout programátorské rozhraní pro snadnou aplikaci základních i pokročilých algoritmů generování procedurálních textur a možnost ukládat a načítat popis textur jako malé textové soubory, jež je možné interaktivně editovat pomocí grafického nástroje. Ten však není pro práci s knihovnou nezbytný a slouží pouze jako podpůrný software.

Pozornost byla věnována způsobu popisu textury. Ten je navržen jako graf, kde každý uzel představuje operaci nad texturou, ať jde o generování šumu, vykreslení L-systému nebo aplikaci osvětlovacího modelu. Tento přístup dovoluje část popisu textury dynamicky upravit a znovu přepočítat pouze ty části, které je nutné. Popis se snaží být nezávislým na rozlišení, pozice jsou tedy většinou zadávány v procentech místo pixelů, takže lze snadno manipulovat s rozměrem textury.

Předpokládaný přínos práce spočíval hlavně v možnosti umožnit programátorům jednoduše generovat unikátní textury a rovněž radikálně snížit velikost programu, který tyto textury využívá ve vysokém rozlišení. Knihovna takovému programu umožní dynamicky generovat textury po spuštění či kdykoli za běhu bez nutnosti je trvale ukládat, a to pouze na základě malého textového souboru. Existuje však širší využití a lze si představit, že uživatel použije nástroj např. k vygenerování textury pozadí pro své webové stránky či při tvorbě uměleckých 3D modelů. Snahou také bylo do určité míry experimentovat, upravovat klasické algoritmy a dosahovat nových výsledků.

Kapitola 2

Procedurální textury v počítačové grafice

Tato kapitola osvětluje obecné principy procedurálního generování textur a definuje základní pojmy používané dále v textu.

2.1 Základní postup generování textur

Pojmem bitmapa budeme rozumět dvourozměrné pole pixelů skládajících se ze složek R, G a B (červená, zelená a modrá), ač je někdy namísto toho pojmu používán termín pixelmapa a bitmapa označuje dvourozměrné pole bitů [10]. Každá složka může nabývat celočíselné hodnoty z intervalu $\langle 0, 255 \rangle$, v některých případech však budeme pracovat s normalizovanými hodnotami v rozsahu $\langle 0, 1 \rangle$. Platí-li, že pro každý pixel mají složky R, G a B stejnou hodnotu, pak budeme tuto hodnotu označovat jako intenzita a celou bitmapu nazveme šedotónovou.

Texturu definujeme jako bitmapu nesoucí informace o určitém povrchu. Implicitně bude touto informací barva, je však přípustná celá řada dalších možností, např. směr normál povrchu, míra jeho odlesků apod.

Aby působila textura realisticky, je většinou vytvářena na základě dat získaných z reálného světa, jako je např. digitální fotografie. Tento přístup při snaze o realistické zobrazení převládá, ale má své nevýhody. Vytváření textur tímto způsobem trvá relativně dlouho a musí jej provádět umělec. [1] Textura musí být dále ukládána jako bitmapa v souboru, který zabírá poměrně hodně místa, a její rozlišení nelze zvýšit. Určité aplikace mohou požadovat eliminaci těchto nevýhod.

Řešením je použít algoritmus, který podle vstupních parametrů vygeneruje texturu v daném rozlišení. Tento způsob budeme nazývat procedurálním generováním textur a rozdělíme jej do tří fází, které mohou být prováděny opakovaně a v různém pořadí v závislosti na složitosti textury:

- generování základu – Na základě malého počtu parametrů algoritmem vygenerujeme bitmapu, která poslouží jako základ textury.
- aplikace transformací – Aplikujeme algoritmus, jímž modifikujeme již vytvořenou bitmapu.

[1] Existuje snaha o automatizaci tohoto procesu zvaná syntéza textur [18]. Stále je však většinou jednodušší a spolehlivější přenechat tuto práci člověku.

- kombinování textur – Spojíme určitým způsobem dvě nebo více již vytvořených bitmap do jedné.

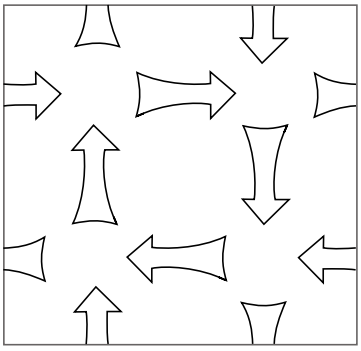
Zajištění návaznosti

Často vyžadujeme, aby textura navazovala sama na sebe v oblasti okrajů bez znatelné nespojitosti. U procedurálních textur bude tento požadavek platit vždy. Důvodem je předpoklad, že textury často mapujeme na povrch tak, že se opakují. Našim cílem je nyní najít způsob, jak obecně zajistit, aby textura sama na sebe navazovala.

Tohoto lze dosáhnout zavedením tzv. wrap-around (modulárních) souřadnic [8]. Máme-li souřadnici x určitého rozměru textury a rozlišení v tomto rozměru je N, pak wrap-around souřadnici x' vypočítáme jako:

$$x' = \begin{cases} N - (|x+1| \bmod N) - 1 & x < 0 \\ x \bmod N & \text{jinak} \end{cases}$$

Přístup k datům bitmapy budeme dále uvažovat výhradně pomocí wrap-around souřadnic. Bude-li algoritmus zapisovat nebo číst obrazová data na souřadnicích $[x,y]$, vždy se tyto souřadnice nejdříve přepočítají na $[x',y']$ a až poté se použijí k přístupu. Algoritmy se tak nemusí starat o kontrolu přístupu za hranice bitmapy a zároveň zajistíme spojitý přechod z jednoho okraje textury na opačný.



Obrázek 2.1: wrap-around souřadnice

Kapitola 3

Použité metody

Následující podkapitoly zmiňují konkrétní metody generování textur použité v knihovně.

3.1 Šum

Jako vhodný základ pro texturu lze použít dvourozměrný šum, tedy funkci dvou proměnných vykazující určitý náhodný charakter. Šum bude rovněž důležitý u metod založených na modulaci. Přirozeně se jako první možnost nabízí použít klasický, jednoduše generovatelný bílý šum. Ten pro naše účely ale většinou není vhodný, jelikož není spojitý. Je proto třeba použít jiné druhy šumů.

Perlinův šum

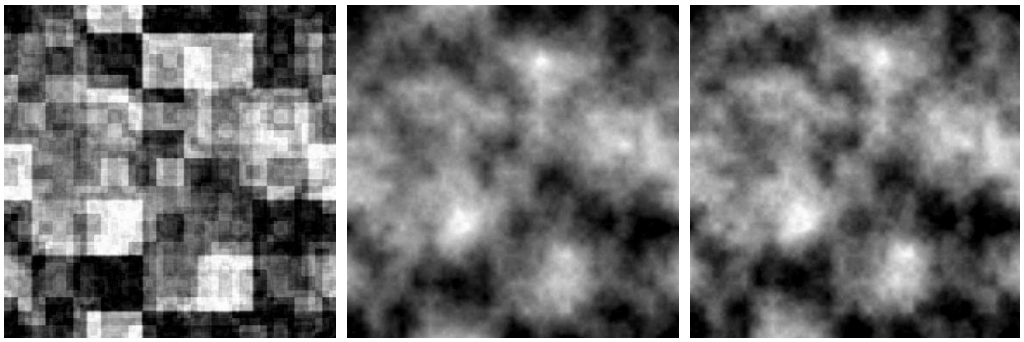
Perlinova šumová funkce původně označovala jeden konkrétní způsob generování šumu o dané frekvenci, později se však takto začal označovat součet těchto funkcí s postupně vzrůstající frekvencí a snižující se amplitudou, potažmo mohlo jít i o součet jiných funkcí, než původně definoval Perlin [6]. V tomto nejširším významu budeme pojem Perlinova šumu používat my.

Parametry Perlinova šumu pro nás budou počáteční frekvence f_0, amplituda a_0 a interpolační metoda. Postupně provedeme iterace generování šumových funkcí, které budeme sčítat. Po každé iteraci se hodnota frekvence zdvojnásobí a hodnota amplitudy sníží na polovinu, přičemž klesne-li pod nejmenší rozlišitelnou hodnotu, generování šumu končí. V první iteraci tedy generujeme šum o frekvenci f_0 s amplitudou a_0 tak, že rozdělíme plochu textury na pravidelnou čtvercovou síť o $f_0 \times f_0$ bodech a každému bodu přiřadíme náhodnou hodnotu z intervalu $\langle -a_0, a_0 \rangle$. Hodnoty mezi body interpolujeme vhodnou metodou, např. lineárně nebo funkcí sinus. Ve druhé iteraci postupujeme obdobně s hodnotami $f_1 = 2f_0$ a $a_1 = \frac{a_0}{2}$ atd.

Perlinův šum lze použít jako bitmapu pro základ textury, především pak mraků nebo kouře, jimž se velmi podobá. Velké uplatnění však nachází při aplikaci jiných technik, které různým způsobem využívají šum, a to především díky svému charakteru turbulence [6].

Fault Formation šum

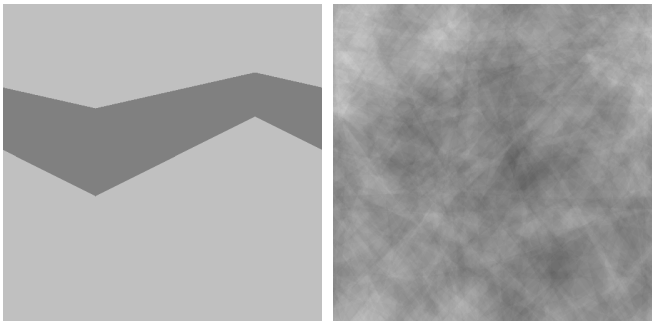
Fault Formation šum se používá při generování výškových map některých typů terénu. Následující odstavec vysvětlí princip algoritmu podle článku o generování terénu [2].



Obrázek 3.1: Perlinův šum: interpolace nejbližším sousedem, lineární a sinová

Začneme s konstantní výškou terénu o hodnotě 0 a v postupných iteracích jej budeme rozdělovat na poloviny a ty vůči sobě vzájemně výškově posouvat. V první iteraci tedy zvolíme rozdělení terénu náhodně umístěnou přímkou. Oběma vzniklým polovinám přiřadíme náhodný výškový posun z určitého intervalu (kladný či záporný). Ve druhé iteraci stejným způsobem opět rozdělíme terén na poloviny a provedeme jejich výškový posun, ten však tentokrát generujeme z užšího intervalu. Náhodný výškový posun se takto lineárně snižuje, až dosáhne nuly a generování šumu skončí.

Je patrné, že takto generovaný šum nenavazuje na okrajích a nesplňuje tak naše požadavky pro generování textur. Lze však provést úpravy, které návaznost šumu zajistí. Toho dosáhneme tehdy, pokud v každé iteraci rozdělíme terén způsobem, jenž zachová návaznost. To lze provést několika způsoby – buď zvolíme pro půlení pouze přímky zachovávající návaznost, nebo použijeme místo přímek jinou křivku. První způsob by nám dovolil rozdělovat terén pouze vodorovnými nebo svislými přímkami, což by se negativně projevilo na přirozenosti vzhledu, neboť by tyto dva směry byly v generovaném šumu znát. Zvolíme proto druhý způsob a k rozdělování použijeme lomenou přímku. Výsledný šum je na obrázku 3.2.



Obrázek 3.2: Fault Formation – rozdělení obrazu v jedné iteraci a výsledek (200 iterací)

Bodový šum

Bodový šum (anglicky spot noise) je definovaný jako součet různě násobené a posunuté funkce $h(x)$ [20]:

$$\text{spot noise}(x) = \sum_i a_i h(x - x_i),$$

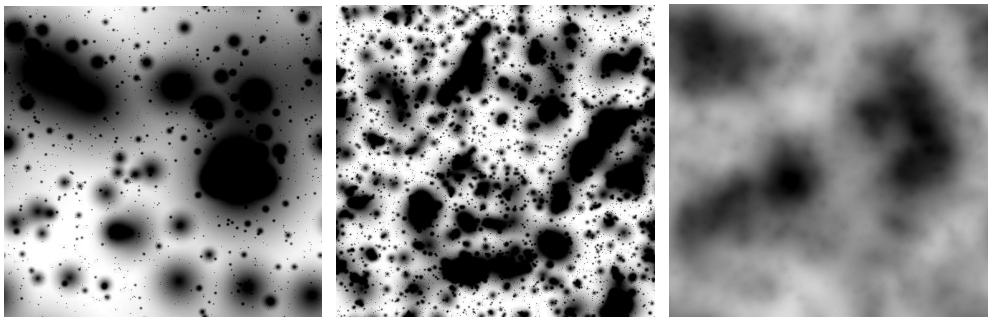
kde x_i tvoří náhodnou posloupnost. Definujme funkci

$$\text{bump}(x, r) = \begin{cases} 0 & |x| \geq r \\ 1 - sin(\frac{|x|}{r}\frac{\pi}{2}) & \text{jinak} \end{cases},$$

kde r je zvolený poloměr. Upravme nyní funkci spot noise na funkci

$$\text{bump noise}(x) = \sum_i a_i \text{bump}(x - x_i, r_i).$$

Tato funkce tvoří šum jako součet různě vysokých a širokých nerovností povrchu, jejichž tvar je popsán funkcí bump. Změna velikosti nerovností je ovlivněna posloupností koeficientů a_i (výška) a r_i (poloměr). Pokud nastavíme např. koeficient a_i jako konstantní, budou všechny nerovnosti stejně vysoké, zatímco nastavíme-li jej jako klesající, budou se v šumu nacházet vysoké i nízké nerovnosti. Obrázek 3.3 ukazuje dvourozměrnou variantu tohoto šumu s různě nastavenými koeficienty.



Obrázek 3.3: bump noise s různými parametry – konstantní amplituda a proměnlivý poloměr, konstantní amplituda a proměnlivý poloměr z užšího intervalu, proměnlivá amplituda a proměnlivý poloměr

3.2 Voroného diagramy

Voroného diagram je způsob dělení metrického prostoru na základě množiny bodů S a může sloužit ke generování základu textury. V literatuře [3] je Voroného diagram definován takto:

Máme-li konečnou množinu řídících bodů prostoru S, pak dominancí bodu p nad bodem q patřících do S rozumíme množinu bodů

$$dom(p, q) = \{x \in R^2 | \delta(x, p) \leq \delta(x, q)\},$$

kde δ značí Euklidovskou vzdálenost. Oblast bodu $p \in S$ je množina

$$reg(p) = \bigcap_{q \in S - \{p\}} dom(p, q).$$

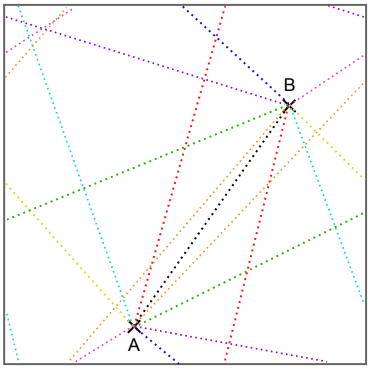
Oblasti všech bodů z S rozdělují prostor na konvexní polygony a společně tvoří Voroného diagram.

Počet řídících bodů množiny S a jejich pozice lze určit několika způsoby. Můžeme je generovat náhodně, pravidelně (např. po obvodu kružnice, pomocí L-systému, ...) nebo explicitně udat polohu každého z nich.

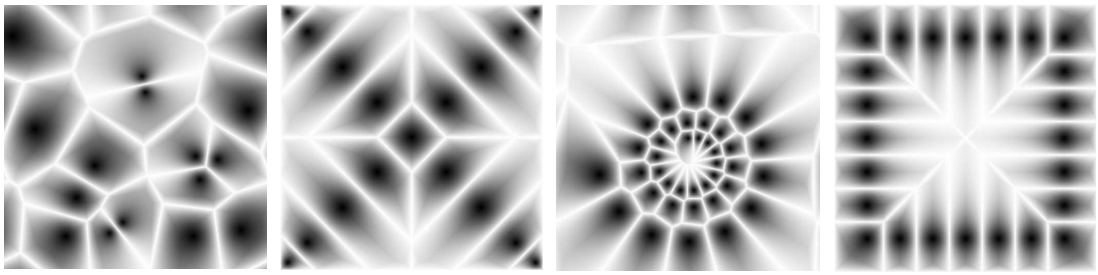
Zbývá definovat způsob, jakým Voroného diagram vizualizujeme jako bitmapu. Za tímto účelem je třeba určit pravidlo, jak vypočítat intenzitu každého pixelu podle informace z Voroného diagramu. Existuje mnoho způsobů, z nichž uveďme dva základní:

- Intenzita pixelu odpovídá vzdálenosti k nejbližšímu řídícímu bodu.
- Intenzita pixelu odpovídá poměru vzdáleností ke dvěma nejbližším řídícím bodům.

Dodejme, že měříme-li vzdálenost dvou bodů za použití wrap-around souřadnic, musíme spočítat minimum ze vzdáleností uvedených na obrázku 3.4.



Obrázek 3.4: vzdálenosti bodu A a B ve wrap-around souřadnicích



Obrázek 3.5: různé způsoby rozmístění řídících bodů Voroného diagramu

Metriky

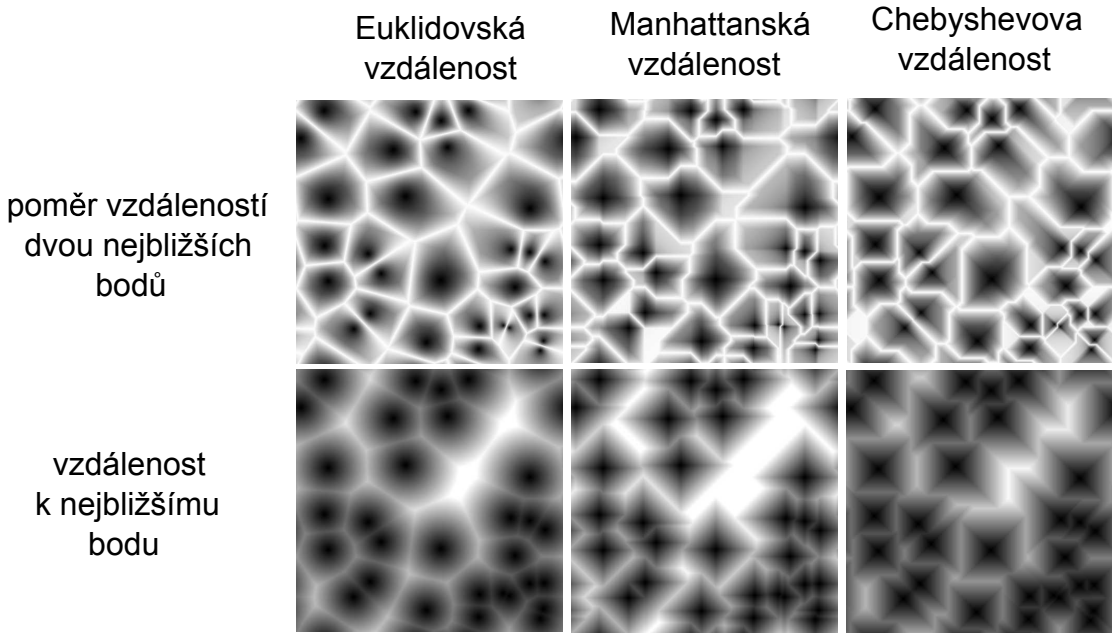
Význam výrazu metrika se může v různých publikacích a kontextech lišit [19]. V této práci jím budeme rozumět způsob měření vzdálenosti $\rho(p, q)$ dvou bodů, p a q, ve dvourozměrném prostoru. Jelikož při konstrukci Voroného diagramů počítáme vzdálenosti bodů, můžeme zavedením různých metrik dosáhnout lišících se výsledků. Nejčastějšími metrikami jsou:

Euklidovská [4]	$\rho(p, q) = \sqrt{(p_x - q_x)^2 - (p_y - q_y)^2}$
Manhattanská [1]	$\rho(p, q) = \lvert p_x - q_x \rvert + \lvert p_y - q_y \rvert$
Chebyshevova [1]	$\rho(p, q) = max(\lvert p_x - q_x \rvert, \lvert p_y - q_y \rvert)$

Optimalizace pomocí k-d stromu

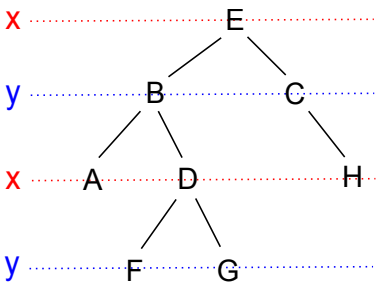
Často prováděným úkonem při generování Voroného diagramu je hledání nejbližšího sousedního bodu z množiny S k pozici aktuálního pixelu (tzv. nearest neighbour problém). Urychlení tohoto procesu vede ke zrychlení generování diagramů s větším množstvím řídících bodů.

Tohoto cíle můžeme dosáhnout použitím datové struktury zvané k-d strom k uchovávání bodů Voroného diagramu. Tato datová struktura definuje hierarchické dělení prostoru pomocí řezů kolmých na souřadnicové osy, přičemž ty se střídají s každou iterací [6].



Obrázek 3.6: různé metriky a metody určování intenzity pixelu u Voroného diagramu

Ilustrujme princip k-d stromu na příkladu, kdy máme body rozmístěné způsobem, který znázorňuje obrázek 3.8. Nejdříve provedeme sestavení stromu. Začínáme tím, že bereme v úvahu celou oblast, tedy všechny body, a rozdělení této oblasti provedeme podle souřadnice x náhodně zvoleného bodu na levou a pravou podoblast. [1] V našem případě jsme vybrali bod E. Všechny body nalevo od tohoto bodu budou v levé větvi stromu, všechny ostatní v pravé. Nyní rekurzívně postupujeme stejným způsobem pro obě nově vzniklé oblasti pouze s tím rozdílem, že je budeme rozdělovat podle souřadnice y. Souřadnice střídáme stejným způsobem, než umístíme všechny body do stromu znázorněného na obrázku 3.7.

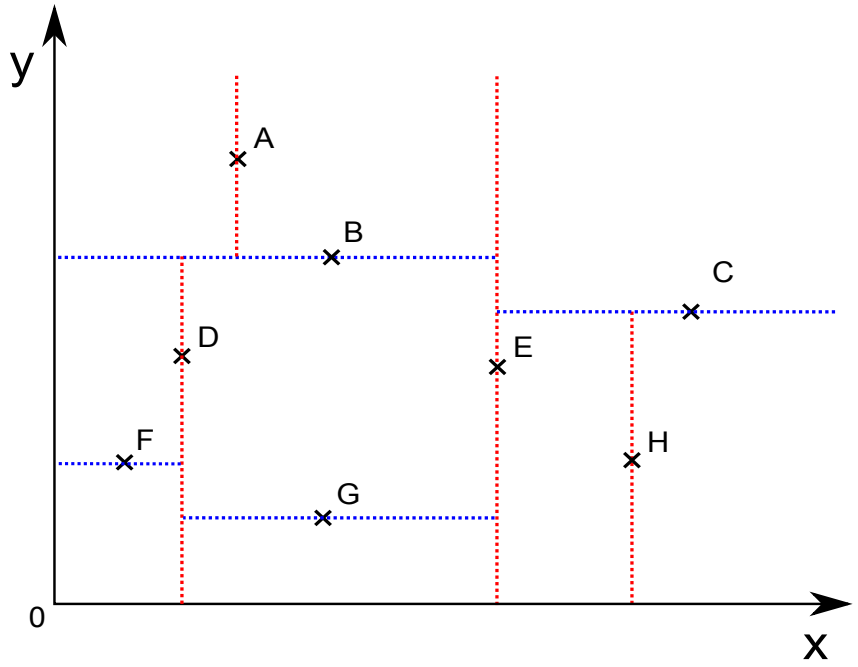


Obrázek 3.7: k-d strom

Pokusme se nyní vyhledat nejbližšího souseda bodu I (obrázek 3.9). Postupujeme stejně jako při vkládání nového bodu – procházíme strom od kořene a cestu volíme podle toho, zda námi hledaný bod leží v levé či pravé oblasti, které jsou určeny aktuálním uzlem. Takto se dostaneme k bodu, který označíme za aktuálně nejlepšího kandidáta (v našem případě G). Nyní provedeme zpětný průchod všemi uzly, jimiž jsme prošli, a u každého provedeme následující:

- Je-li aktuální uzel bližší než aktuálně nejlepší kandidát, označíme jej za nového nejlepšího kandidáta.
- Zkontrolujeme, zda může existovat bližší bod v opačné oblasti, než ve které se nacházíme. Test provedeme sestrojením kružnice se středem v bodě I o poloměru rovném

[1] Pro maximální efektivitu k-d stromu bychom měli brát bod se souřadnicí, která je mediánem, čímž zajistíme rovnoměrné rozdělení prvků do levé a pravé větve. Pro jednoduchost si však vystačíme s náhodně vybíranými body.



Obrázek 3.8: sestavování k-d stromu

aktuálně nejkratší vzdálenosti (vzdálenosti k nejlepšímu kandidátovi). Pokud tato kružnice nezasahuje do opačné oblasti, nemůže v ní existovat bod bližší, než je aktuálně nejlepší kandidát, a proto nemusíme druhou větev vůbec procházet (čímž k-d strom získává svou efektivitu). Pokud kružnice do oblasti zasahuje, jako na obrázku 3.9, musíme provést celý proces hledání nejbližšího souseda i pro druhou větev. V našem případě takto za nového nejlepšího kandidáta označíme bod F, který už dále zůstane nejbližším bodem, jenž jsme hledali.

Barvení šedotónových bitmap

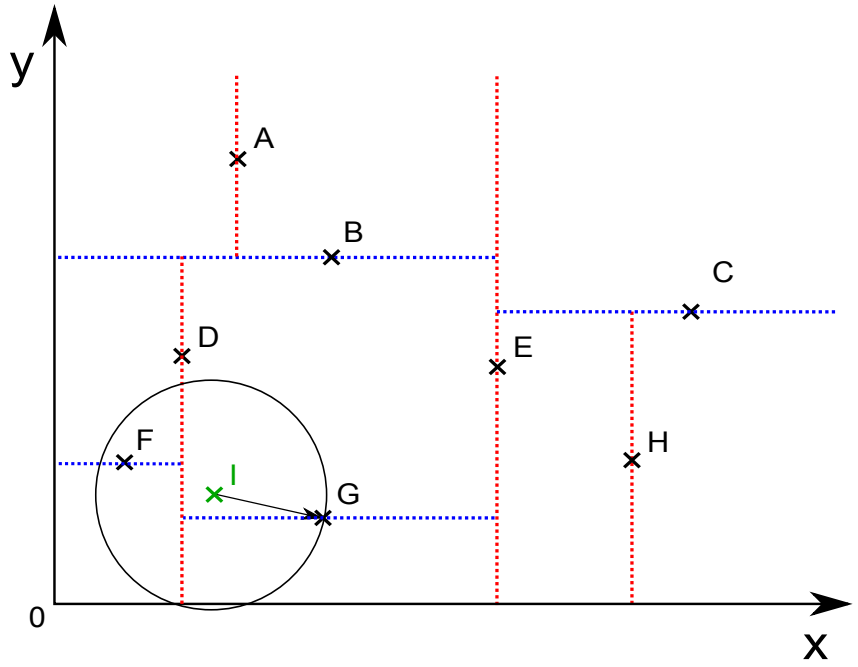
Výše popsané šumy a Voroného diagramy generují šedotónové bitmapy. Abychom jim dodali barvu, můžeme použít např. barevný přechod. Jedná se o tabulku, která přiřazuje každé intenzitě pixelu barvu, jíž bude nahrazena. Tento pojem se také podle kontextu nazývá barevná mapa nebo barevná paleta [6].

Další možností, jak získat barevnou bitmapu, je vygenerovat tři různé šedotónové bitmapy, každou pro jednu ze složek R, G a B, a spojit je do jednoho obrazu. Oba způsoby jsou vidět na obrázcích 3.10 a 3.11.

3.3 Obecné bitmapové operace

Nástroj pracující s texturami by měl umožnit základní operace definované pro obecný obraz, jako je např. editace jasu a kontrastu nebo vyvážení barev. Operace pracující pouze s jednotlivými pixely, např. inverze barev, lze aplikovat bez jakékoli modifikace, u komplexnějších algoritmů typu grafický filtr však musíme zajistit, aby pracovaly s wrap-around souřadnicemi [2] a nenarušily návaznost textury.

[2] viz část 2.1



Obrázek 3.9: test pomocí kružnice

Grafické filtry

Grafickým filtrem budeme rozumět filtr založený na dvourozměrné konvoluci [16]. Tyto filtry jsou jednoduše specifikovatelné a implementovatelné a umožňují realizaci základních operací, jako je rozostření, detekce hran apod. Nevýhodou je časová náročnost provádění dvourozměrné konvoluce pro velké konvoluční matice, avšak v běžných případech vystačíme s maticemi dostatečně malých rozměrů. Knihovna umožní uživateli specifikovat vlastní konvoluční matice.

Barevné modely

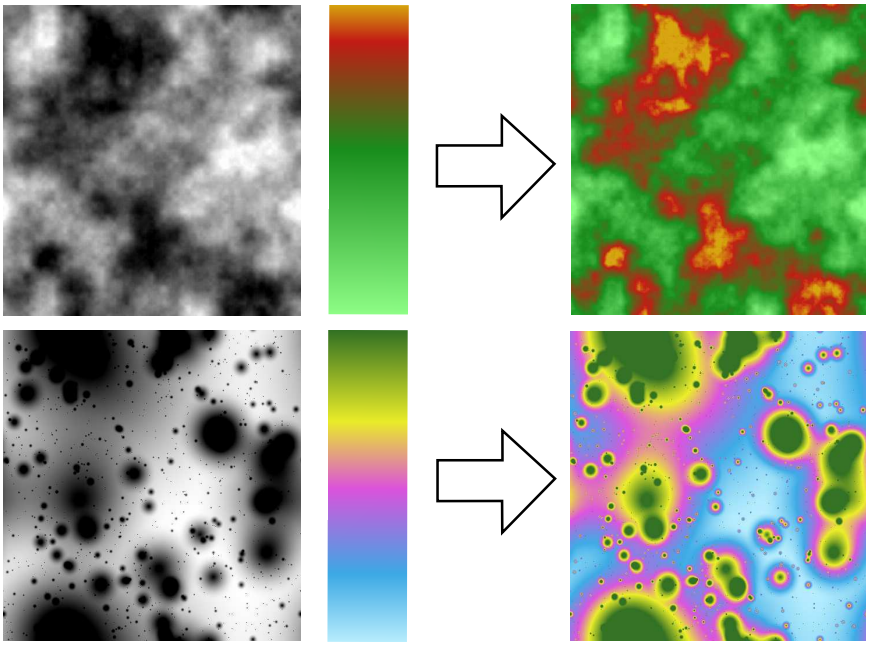
Bitmapa je typicky uchovávána v barevném modelu RGB, který je nativním pro většinu zobrazovacích zařízení a při vykreslování tudíž nedochází ke složitým výpočtům. Z hlediska vnímání barvy člověkem ale nemusí být tento model vhodný. Uživatel často vyžaduje možnost práce v jiných dimenzích, než mu nabízí složky RGB modelu.

Z tohoto důvodu vznikly další modely, jako např. HSL reprezentující barvu hodnotami tónu (hue), sytosti (saturation) a světlosti (lightness) [6]. Pro editaci těchto složek převedeme obraz existujícími algoritmy z RGB do HSL, provedeme úpravy a zpětný převod do RGB.

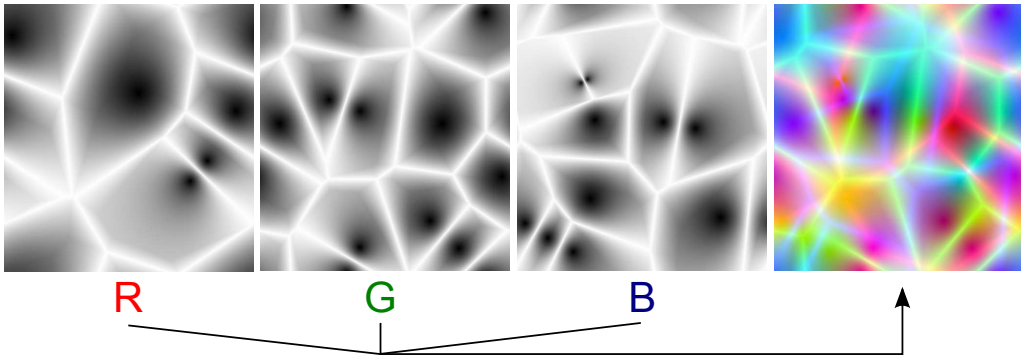
Omezení barevného prostoru

Omezení barevného prostoru při zachování rozlišení se nazývá rozptylování nebo dithering [6]. Tato operace se využívá pro úpravu obrázků, které mají být zobrazovány na zařízeních podporujících omezené množství barev. Nejvyužívanějšími metodami jsou: [3]

[3] Podrobné informace uvádí literatura [6].



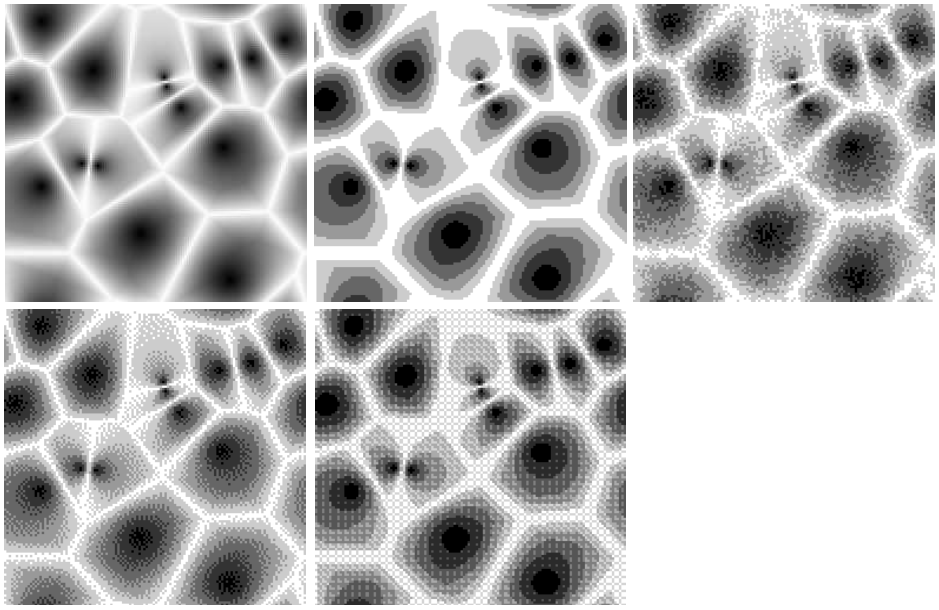
Obrázek 3.10: barevné přechody



Obrázek 3.11: míchání RGB kanálů

- prahování – Hodnota každého pixelu se zaokrouhlí nezávisle na okolních pixelech k nejbližší barvě výsledné množiny barev. Jde o jednoduchou metodu nepříliš vhodnou např. pro fotografie.
- náhodné rozptýlení – Zda se hodnota pixelu zaokrouhlí nahoru či dolů, se rozhoduje vždy náhodně, přičemž pravděpodobnost, že zaokrouhlení proběhne nahoru, je tím vyšší, čím je bod této hodnotě blíže. Výsledný obraz je u běžného obrazu kvalitnější než při prahování.
- rozptýlení s distribucí chyby – Hodnota pixelu se zaokrouhluje buď nahoru nebo dolů s ohledem na chyby při zaokrouhlování předchozích pixelů. Tato metoda dává velmi dobré výsledky.
- pravidelné rozptýlení – Tato metoda využívá předem definované matice, jimiž se obraz pravidelně pokryje a které následně rozhodují o směru zaokrouhlení hodnoty každého

pixelu. Pravidelné rozptýlení často používáme při tisku, jelikož díky maticím vznikají v obraze vzory, které jsou z hlediska tisku výhodné. U opakujících se textur však hrozí riziko vzniku drobných nenávazností těchto vzorů na okrajích textury, není-li její rozměr dělitelný rozměrem matice.



Obrázek 3.12: dithering: původní obraz, prahování, náhodné rozptýlení, distribuce chyby, pravidelné rozptýlení (Všechny metody lze aplikovat i na barevný obraz.)

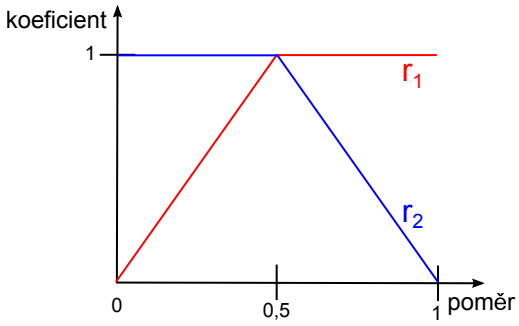
Prolínání bitmap

Při prolínání dochází určitým způsobem k vykreslování dvou obrazů přes sebe. Většinou se tak děje na základě tzv. alfa kanálu, který je spolu s kanály R, G a B další složkou každého pixelu a určuje jeho průhlednost.[4] Dále budeme uvažovat několik metod prolínání dvou obrazů, a to jak s možností definovat alfa kanál v samostatné šedotónové bitmapě pro každý pixel zvlášť, tak i bez explicitně určeného alfa kanálu jenom na základě zadaného poměru prolnutí. Vstupem algoritmu jsou dvě bitmapy a jeden z následujících parametrů:

- poměr prolnutí r v intervalu $\langle 0, 1\rangle$ definující rovnoměrnou hodnotu alfa kanálu pro každý pixel
- šedotónová bitmapa $r(x, y)$ definující hodnotu alfa kanálu pro jednotlivé pixely zvlášť, potenciálně nerovnoměrně

Operace probíhá nad jednotlivými pixely, přičemž pro pixel na souřadnicích $[x_0, y_0]$ se vždy určí dva koeficienty r_1, resp. r_2 na základě aktuální hodnoty alfa kanálu (buď hodnoty r nebo $r(x_0, y_0)$), kterými se vynásobí hodnoty získané z pozice $[x_0, y_0]$ v první, resp. druhé prolínané bitmapě. Výpočet koeficientů r_1 a r_2 byl zvolen tak, jak znázorňuje obrázek 3.13.

[4] 0 značí úplnou průhlednost, 255 úplnou neprůhlednost [10].



Obrázek 3.13: výpočet koeficientů prolínání

Díky těmto vahám je možné vytvořit plynulý přechod z jednoho obrázku do druhého při změně hodnoty alfa kanálu. Výpočet konečné intenzity v složky pixelu je dán zvolenou metodou (v_1, resp. v_2 značí hodnotu pixelu první, resp. druhé bitmapy):

- součet: $v = r_1 \cdot v_1 + r_2 \cdot v_2$
- rozdíl: $v = r_1 \cdot v_1 - r_2 \cdot v_2$
- průměr: $v = \frac{r_1 \cdot v_1 + r_2 \cdot v_2}{r_1 + r_2}$
- součin: $v = (r_1 \cdot v_1 + 1 - r_1) \cdot (r_2 \cdot v_2 + 1 - r_2)$

Supersampling

Častým problémem, s nímž se v počítačové grafice setkáváme, je tzv. aliasing, neboli zkreslení způsobené vzorkováním s příliš malou frekvencí vzhledem ke vzorkovanému signálu. Tomuto jevu můžeme částečně zabránit, pokud pro výpočet hodnoty každého pixelu použijeme více než jeden vzorek. Tato technika se nazývá supersampling [10].

Stupeň supersamplingu udáme celým číslem větším než nula. Označme toto číslo s. Hodnota $s = 1$ znamená, že se supersampling nepoužije (i když se takto někdy označuje vzorkování v rozích pixelů, ke kterému zde však nedochází [10]). Při vyšší hodnotě se bitmapa s požadovaným rozlišením $x \times y$ nejdříve vygeneruje v rozlišení $sx \times sy$ a následně je zmenšena tak, že se každému pixelu přiřadí průměrná hodnota s^2 okolních pixelů.

Popis parametrů textury by měl být navržen tak, aby byl nezávislý na rozlišení textury a aby měla změna hodnoty supersamplingu vliv pouze na aliasing, nikoli na generovaný obraz. Z tohoto důvodu by měl veškerý popis využívat relativní rozměry a souřadnice, nikoli absolutní hodnoty v pixelech. Nezávislost na rozlišení je však v některých případech obtížné zajistit (např. u celulárního automatu, který pracuje s pixely jako s buňkami).

Geometrické transformace

Geometrické transformace budou takové transformace, které nezmění celkový vzhled obrazu, ale pouze jeho umístění (při dodržení wrap-around souřadnic). Příklady těchto transformací jsou např. posunutí všech pixelů v daném směru, otočení obrazu o celočíselný násobek 90 stupňů, překlopení podle osy nebo opakování obrazu v určitém směru (viz obrázek 3.14).

Tyto operace mohou být vhodné k drobným úpravám, avšak lze je také použít jako základ složitějších operací, např. sečtení obrazu s posunutou kopií sebe sama k vytvoření iluze stínu.

Obrázek 3.14: geometrické transformace – původní obraz, posunutí, překlopení, otočení a opakování

3.4 Modulace šumem

Modulace šumem je jedním z nejčastěji používaných způsobů generování procedurálních textur a využívá se většinou k napodobení dřeva či mramoru. Podstata metody spočívá v použití šumu, popsaného v části 3.1, k modulaci velmi jednoduchého základního obrazce (tzv. rampa), například kružnice nebo svislých pruhů [6].

Jednoduchou texturu mramoru vytvoříme např. ze svislých pruhů generovaných funkcí sinus souřadnice x. Musíme samozřejmě zajistit návaznost obrazce a proto generujeme celočíselný násobek periody funkce sinus. Dále vygenerujeme šum, např. Perlinův, a iterací přes všechny pixely provedeme modulaci:

$$bitmap(x, y) \leftarrow bitmap(x + i \cdot noise(x, y), y).$$

Proměnná i značí intenzitu modulace. Zde uvedené přiřazení provede horizontální modulaci, vertikální modulace se provede přičtením hodnoty šumu k souřadnici y namísto x. Obdobně lze definovat i diagonální směry.

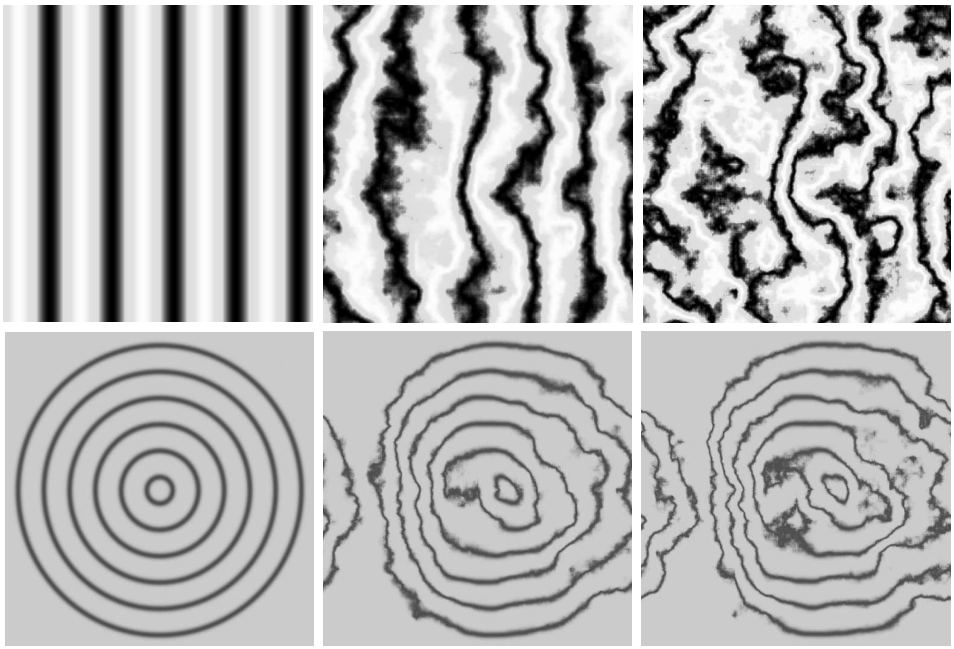
3.5 Substrate algoritmus

Přestože nejčastěji používáme jako základ textury šum nebo Voroného diagram, je možné využít i méně tradiční metody. Jednou z nich je algoritmus Substrate, který definoval J. Tarbell [28].

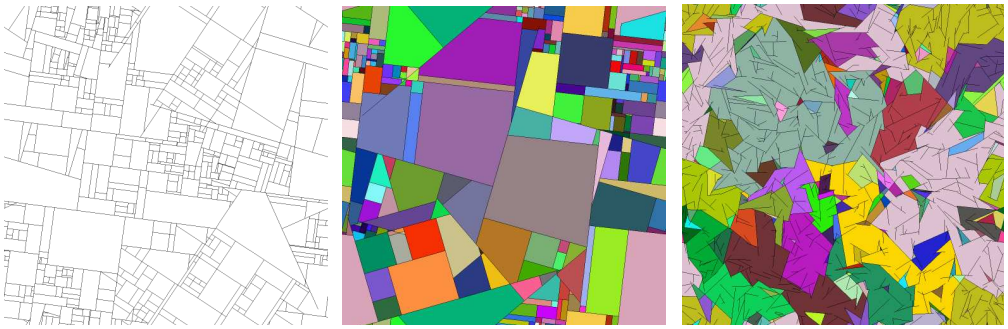
Algoritmus se ve své původní verzi snaží napodobit kresbu vodovými barvami. Postup generování spočívá v kreslení úseček z náhodně zvolených počátečních bodů a v náhodně zvolených počátečních úhlech, přičemž každá úsečka je kreslena tak dlouho, dokud nenarazí na jinou úsečku. Kolmo od kreslené úsečky se navíc v náhodných intervalech začínají kreslit nové úsečky. Vzniklé oblasti lze vybarvit např. semínkovým vyplňováním.

3.6 Obrazové transformace

K docílení rozmanitosti generovaného obrazu bychom měli mít k dispozici řadu transformací, které bude možné aplikovat na vygenerovaný základ textury a změnit tak určitým způsobem jeho celkový vzhled. Konkrétně hledáme algoritmy s malým počtem vstupních



Obrázek 3.15: modulace prováděná Perlinovým šumem s postupně zesilujícím vlivem



Obrázek 3.16: modifikovaný Substrate algoritmus: bez výplně, s náhodnou výplní a bez iterací

parametrů proveditelné nad bitmapou, přičemž pokud je tato bitmapa navazující na sebe sama, zůstane navazující i po provedení transformace.

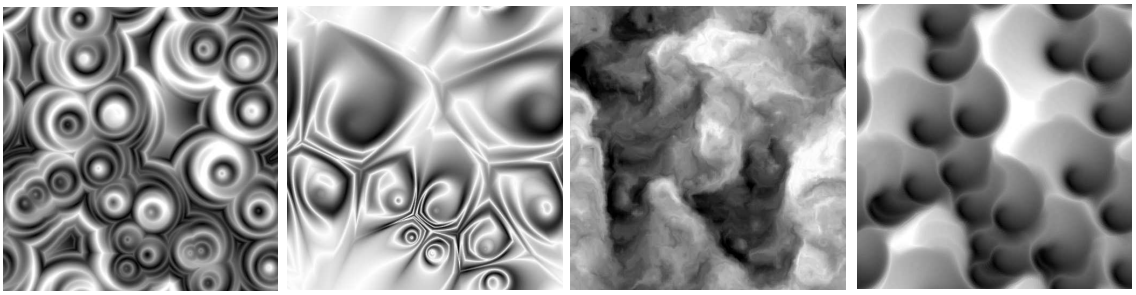
Nejjednodušším případem jsou per-pixel transformace (jen s jednotlivými pixely). Mezi ty patří např. aplikace goniometrické funkce na hodnotu každého pixelu. Lepších výsledků ale dosahujeme variacemi algoritmů použitých v práci zabývající se 3D intrem [15].

Tyto transformace pracují s šedotónovou bitmapou a fungují na podobném principu jako modulace šumem popsaná v části 3.4. K modulaci však není využit šum, ale samotný modulovaný obraz. Transformace se vždy provádí iterací nad všemi pixely, které jsou podle určitého pravidla nahrazovány pixely v blízkém okolí. Toto nahrazení je v původní verzi definováno takto:

$$v \leftarrow bitmap(x, y)$$
$$bitmap(x, y) \leftarrow bitmap(x + r \cdot cos(2\pi \cdot v), y + r \cdot sin(2\pi \cdot v)).$$

Proměnná r je zvolený poloměr a $bitmap(x, y)$ označuje hodnotu v intervalu $\langle 0, 1 \rangle$ pixelu

na souřadnicích $[x, y]$. Transformace v podstatě nahradí každý pixel hodnotou, která se získá skokem o daný poloměr z pozice aktuálního pixelu v úhlu, který je přímo úměrný intenzitě aktuálního pixelu. Toto přiřazení lze různě modifikovat, např. zavedením nekonstantní délky poloměru, vícenásobného skoku apod.



Obrázek 3.17: obrazové transformace

3.7 Celulární automaty

Definujme celulární automat jako systém diskrétního času skládající se z buněk, z nichž každá může nabývat konečného počtu stavů [17]. Buňky mění s časem svůj stav podle stanovených pravidel, která berou v potaz okolí buňky, tzn. stav buněk v okolí. Zavedeme-li ekvivalenci mezi buňkou a obrazovým bodem, můžeme takové systémy použít k obrazovým transformacím.

Existuje množství celulárních automatů lišících se uspořádáním buněk, použitým okolím, počtem stavů i přechodovými pravidly. V tomto textu se dále budeme zabývat dvourozměrnými celulárními automaty s buňkami uspořádanými do klasické čtvercové mřížky, neboť ty odpovídají použité reprezentaci obrazu. [5]

Dále budeme pro buňku na souřadnicích $[x, y]$ uvažovat vždy jedno ze dvou tradičně zaváděných okolí:

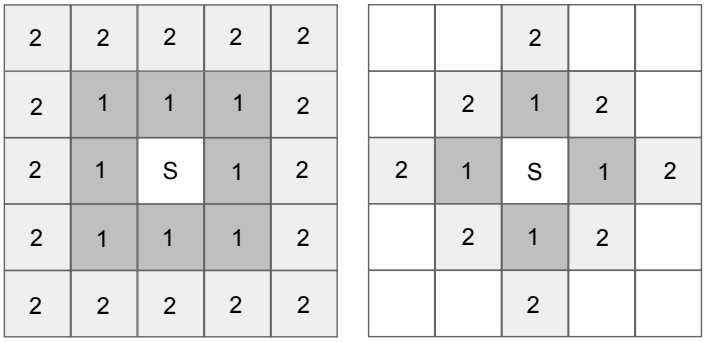
- von Neumannovo – buňky $[x + 1, y]$, $[x - 1, y]$, $[x, y + 1]$ a $[x, y - 1]$
- Mooreovo – von Neumannovo okolí plus buňky $[x+1, y+1]$, $[x-1, y+1]$, $[x+1, y-1]$ a $[x - 1, y - 1]$

Samotnou buňku, pro niž okolí definujeme, za součást tohoto okolí nepovažujeme, i když v některých publikacích tomu tak je. Někdy bude možné specifikovat rovněž velikost okolí; v takovém případě znamená hodnota 1 okolí tak, jak je uvedeno výše, a hodnota $n + 1$ značí okolí vzniklé sjednocením všech okolí buněk okolí velikosti n (buňku $[x, y]$ opět nezahrnujeme).

Cyklický celulární automat

Publikace [5] uvádí tyto parametry cyklického celulárního automatu: počet stavů k, práh θ a druh okolí. Buňka ve stavu n postoupí do stavu $n + 1 \bmod k$, pokud má ve svém okolí alespoň θ buněk ve stavu $n + 1 \bmod k$, jinak se její stav nemění.

[5] Obraz lze generovat i jednorozměrnými celulárními automaty za předpokladu, že časová osa odpovídá ose y. Takto generovaný obraz však nenavazuje ve vertikálním směru a proto se touto možností nebudeme zabývat.



Obrázek 3.18: okolí celulárního automatu: Mooreovo (vlevo) a Von Neumannovo

Kámen, nůžky, papír

Tento automat se zkráceně nazývá RPS (rock, paper, scissors). Je založený na soupeření buněk, při němž o vítězství rozhoduje sada pravidel vycházejících ze hry kámen, nůžky, papír. Takové chování se vyskytuje např. mezi bakteriemi a proto jej můžeme využít k napodobení organického vzhledu určitých povrchů [9].

Ve hře se vyskytují klasicky tři hodnoty: kámen, nůžky a papír, avšak je možné provést zobecnění na jakýkoli vyšší lichý počet. Počet musí být lichý, neboť každá hodnota vyžaduje stejný počet různých soupeřů, s nimiž zvítězí a s nimiž prohraje. Je-li tento počet roven hodnotě n, pak je celkový počet odlišných soupeřů roven $2n$. Po přičtení hodnoty samotné dostáváme celkový počet hodnot $2n + 1$.

Parametrem automatu je tedy počet hodnot účastnících se hry (pro klasickou variantu rovno třem). Každá buňka může nabývat jedné z těchto hodnot a navíc může být i prázdná. Každá neprázdná buňka má dále úroveň v rozsahu 0 až 9, sloužící k omezení rozpínání na prázdném prostoru (úroveň nemá vliv na výslednou intenzitu pixelu). Uveďme nyní popis automatu pomocí algoritmu jeho řízení: [6]

```
for (i in iterations)
  for (c in cells) {
    enemy = random_cell_in_the_neighbourhood(c);

    if (c.state == empty && enemy.state != empty) {
      if (enemy.level < 9) {
          c.new_state = enemy.state;
          c.new_level = enemy.level + 1;
        }
      }
    else if (c.state != empty && enemy.state != empty)
      if (winner(c.state,enemy.state,total_states) == enemy.state) {
          c.new_state = enemy.state;
          c.new_level = enemy.level + 1;
        }
      }
```

Ještě zbývá uvést algoritmus určení vítěze zobecněné hry kámen, nůžky, papír:

[6] Algoritmus je převzatý z webové stránky [27].

```
winner(value1,value2,total) {
    if (value1 > value2)
      swap(value1,value2);

    half = total / 2;

    if (value2 - value1 > half)
      return value1;
    else
      return value2;
  }
```

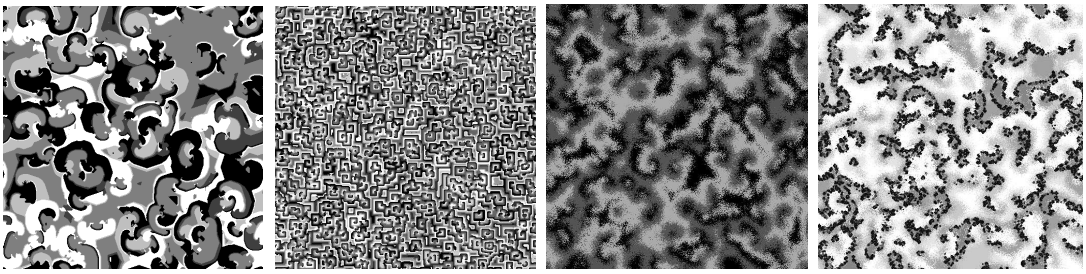
Obecný binární celulární automat

Binární celulární automat je automat mající dva možné stavy buněk, 1 a 0. Obecný binární celulární automat umožňuje specifikaci přechodových pravidel ve formě pravdivostní tabulky, v níž proměnné představují buňky okolí. Pro každý možný stav okolí určité buňky tedy definujeme, do jakého stavu buňka přejde. Povolíme tyto možnosti:

- Buňka přejde do stavu 1.
- Buňka přejde do stavu 0.
- Buňka zůstane ve stavu, v němž se nachází.

Pro automat uvažující N buněk okolí bude tabulka definovat 2^N pravidel. Je proto vhodné držet se malé velikosti okolí.

Chceme-li nějakým způsobem zavést více stavů buněk, abychom generovali obraz s vyšším počtem barev než dvě, ale zároveň nechceme, aby rostla velikost tabulky pravidel, můžeme zavést speciální pravidlo setrvačnosti [21]. Podle tohoto pravidla rozdělíme stav 0 na více jednotlivých stavů $0_0, 0_1, \cdots 0_n$. Tyto stavy vystupují z hlediska pravidel jako stav 0. Pokud má buňka přejít do stavu 0, přejde do stavu 0_0 a v každém dalším kroku, kdy má v tomto stavu setrvat, přejde do stavu 0_{i+1} až po stav 0_n.



Obrázek 3.19: celulární automat: cyklický, cyklický, RPS, obecný binární (se setrvačností)

3.8 Teselace

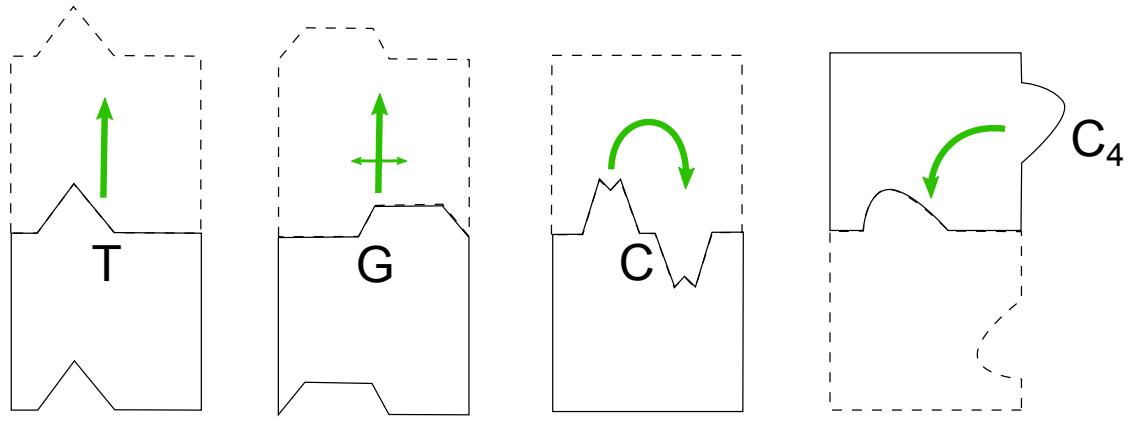
Termínem teselace budeme rozumět periodické dláždění dvourozměrného prostoru jednoduchým obrazcem, např. polygonem [10]. Následující odstavce vychází z článku zabývajícího se Escherovými mozaikami [12].

Počet způsobů, jakými můžeme teselaci provádět, je velmi rozsáhlý. Jelikož teselace není hlavní náplní této práce, stanovíme předpoklady, jimiž výsledný systém zjednodušíme, aby byl reálně implementovatelný, avšak stále dobře použitelný.

První předpoklad se týká mřížky dláždění, která bude čtvercová – základem každé dlaždice bude čtverec. Každou z jeho čtyř stran bude možné upravit výčtem bodů $[x,y]$, kde x udává pozici na straně a y posunutí nahoru nebo dolů. Hodnoty mezi body se následně lineárně interpolují a vytvoří nový tvar strany. Takto bude možné vytvářet originální dlaždice. Teselace bude provedena vždy použitím pouze této jedné, různě prostorově orientované dlaždice.

Dále zavedeme množinu přípustných prostorových transformací, které je možné přiřadit každé straně dlaždice a které společně určí způsob, jakým dlaždice vyplní prostor. Každá transformace určí podmínky pro použití dalších transformací a pro tvar stran dlaždice. Těmito transformacemi jsou:

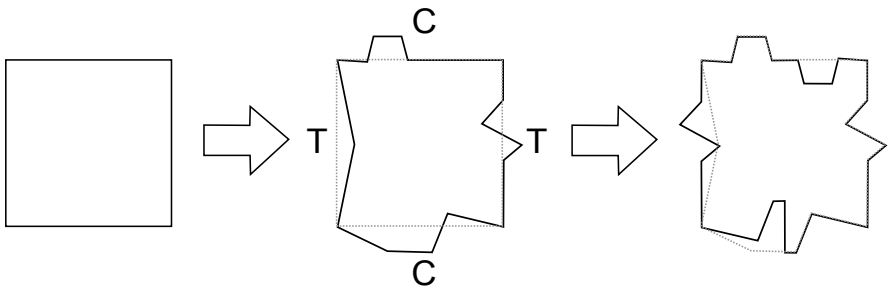
- translace (T) – Dlaždici pouze posuneme v daném směru. Strana protilehlá straně s touto transformací musí mít rovněž nastavenou translaci a musí mít odpovídající tvar. Vzor se ve směru strany opakuje s periodou 1.
- translace s překlopením (G) – Stejně jako u translace dlaždici posuneme, ale navíc dojde ještě k jejímu překlopení. Protilehlá strana musí mít nastavenu stejnou transformaci a odpovídající tvar. Vzor se opakuje s periodou 2.
- rotace kolem středu strany (C) – Dlaždici otočíme o 180 ° kolem středu strany, takže strana s touto transformací bude přiléhat na stejnou stranu druhé dlaždice. Tvar strany musí být symetrický kolem svého středu. Vzor se opakuje s periodou 2.
- rotace kolem vrcholu (C_4) – Dlaždici otočíme o 90 ° kolem jejího krajního bodu. Je-li tato transformace použita pro některou stranu, musí být použita i pro všechny ostatní. Vzor se opakuje s periodou 2.



Obrázek 3.20: teselace – transformace

S těmito předpoklady lze implementovat systém pro specifikaci teselace. Tento systém umožní zadat tvar stran dlaždice a typ transformace pro každou stranu. Jelikož ne každá specifikace je validní, je nutné, aby systém ověřoval výše zmíněné podmínky přípustných typů transformací a aby vynucoval pravidla týkající se tvaru dlaždice jejich úpravou, pokud

je to nezbytné. Pokud je například pro danou stranu zadána transformace C, musí se její tvar upravit tak, aby byl symetrický kolem svého středu (viz obrázek 3.21). Systém potom vykreslí dlaždice jako bitmapu, která se může použít např. jako maska pro prolnutí dvou (či více) textur.



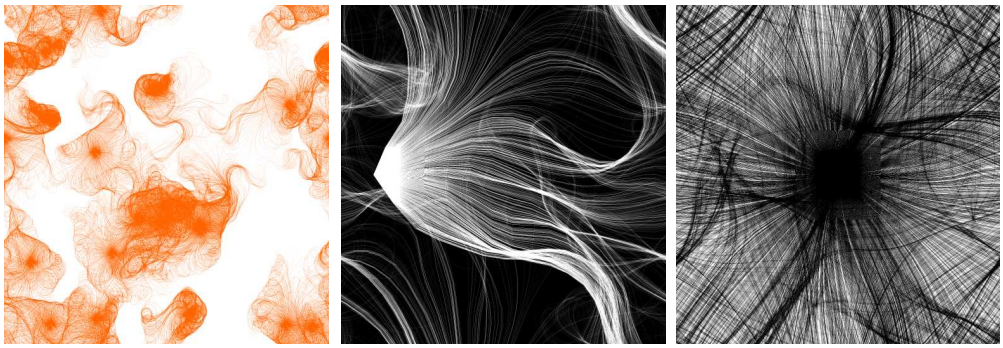
Obrázek 3.21: vytváření dlaždice

Obrázek 3.22: výsledky teselace, zleva $TTTT$, $CGCG$, $C_4C_4C_4C_4$

3.9 Částicové systémy

Částicový systém je definován množinou bodů (částic) rozvíjejících své chování v čase [11]. Tyto systémy jsou v počítačové grafice využívány např. k simulaci obrazu ohně, kouře, deště apod. Jejich použití pro generování textur není zcela typické, avšak může dát výsledkům abstraktnější vzhled.

Základními parametry je pozice zřídla (tj. počáteční pozice částic), počet částic, jejich počáteční rychlost a směr. Každá částice má aktuální velikost a směr rychlosti a intenzitu (jas). Částice následně necháme, aby se pohybovaly, přičemž jejich rychlost a intenzita postupně klesají, než se úplně zastaví a vyhasnou. Klíčovou myšlenkou popsaných systémů je použití šumu, popsaného v kapitole 3.1, k modulaci pohybu částic. Pohyb se totiž musí jevit částečně nahodile, avšak kdybychom pohyb každé částice modifikovali izolovaně, nezávisle na poloze a čase, výsledný obraz by vypadal chaoticky. Překrytím obrazu spojitě se měnícím šumem jakožto modifikátorem rychlosti a směru částic dosáhneme stejné spojité změny u pohybu částic a částice nacházející se blízko sebe v prostoru budou ovlivňovány podobně, což je chování, jaké známe z reálného světa.



Obrázek 3.23: částicové systémy

3.10 Osvětlovací modely a normálové mapy

Realistického zobrazení prostředí, v němž se vyskytuje světlo, dosahujeme pomocí počítačů díky matematickým aproximacím chování světla v reálném světě, jež se nazývají osvětlovací modely [6]. Textury při výpočtu osvětlovacího modelu dříve nehrály velkou roli, avšak v dnešní době je možné, aby se tohoto procesu účastnily použitím tzv. normálových map.

Výpočet osvětlovacího modelu probíhá za pomoci normálové mapy většinou až ve fázi zobrazování a je prováděn zobrazovacím systémem, např. 3D renderovacím softwarem. Mohou se ale vyskytnout systémy, které osvětlovací modely nepodporují, zejména při práci s 2D grafikou. Je proto vhodné poskytnout možnost výpočtu osvětlení ve fázi vytváření textury a informaci o osvětlení do ní přímo zanést (čímž zároveň získáme rychlejší zobrazování, samozřejmě za cenu možnosti dynamicky upravovat parametry osvětlení).

Normálové mapy

Normálová mapa je bitmapa asociovaná s texturou a lze na ni nahlížet jako na samostatnou texturu, která k hlavní textuře nese určité informace navíc. Tyto informace se týkají směru povrchových normál v každém pixelu textury a je proto možné je využít při výpočtu osvětlovacího modelu. Možnost modifikovat normály povrchu texturou umožňuje docílit zdánlivého zvýšení jeho členitosti bez nutnosti použít složitější geometrii, podmínkou je pouze osvětlovací model zohledňující normály. Tato metoda se nazývá Bump Mapping [10]. Technická realizace uložení normál využívá složek R,G a B, které nabývají významu souřadnic x, y a z normalizovaného vektoru normály pro každý pixel.

Phongův osvětlovací model

Tento empirický model reprezentuje světlo odražené od povrchu jako součet intenzit ambientní, difúzní a spekulární složky [6]:

$$I = I_s + I_d + I_a.$$

Ambientní složka reprezentuje rozptýlené světlo a její hodnota je konstantní:

$$I_a = I_{a0}.$$

Difúzní složka představuje světlo odražené rovnoměrně do všech směrů v intenzitě, která závisí na úhlu mezi normálou povrchu a příchozím paprskem. Čím přímějším světlem je

povrch osvětlen, tím více světla odráží:

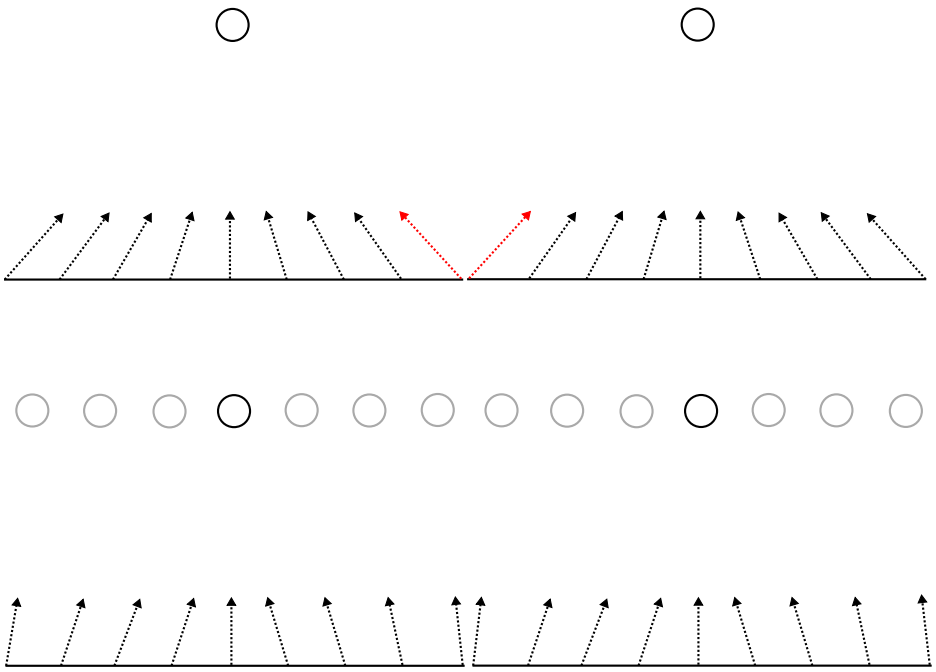
$$I_d = I_{d0}(\vec{l} \cdot \vec{n}).$$

Vektor $\vec{l}$ je normalizovaný vektor směřující z bodu dopadu paprsku ke zdroji světla, $\vec{n}$ je normála. A konečně spekulární (zrcadlová) složka je reprezentací odlesků, tedy odrazů, které závisí na pozici světelného zdroje, normále povrchu a rovněž pozici pozorovatele:

$$I_s = I_{s0} r_s (\vec{v} \cdot \vec{r})^h.$$

Hodnota r_s je míra zastoupení zrcadlových odrazů, h jejich ostrost, vektor $\vec{v}$ je normalizovaný vektor směřující k pozorovateli a vektor $\vec{r}$ je normalizovaný vektor ideálního směru odrazu.

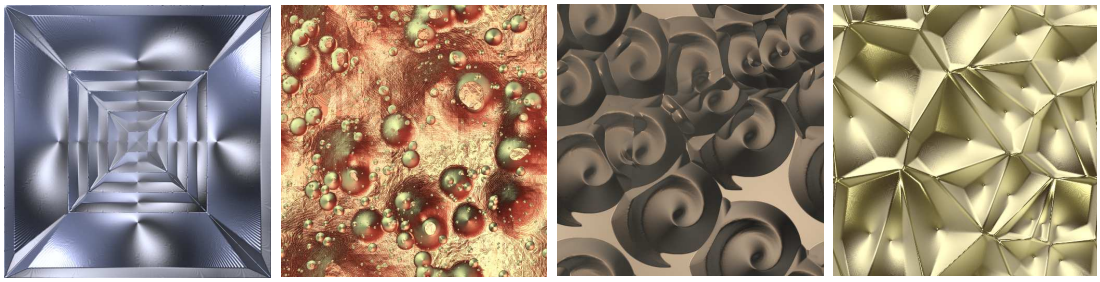
U spekulární složky vyvstává problém, který znázorňuje obrázek 3.24. Jelikož počítáme s pozicí pozorovatele, přesněji s vektorem směřujícím k pozorovateli, pak je-li jeho pozice fixní, dojde při přechodu přes okraj textury ke skokové změně vektoru a tudíž k nenávaznosti v textuře. Tento problém nastává jen v případě, že osvětlovací model počítáme v době výpočtu textury. Na obrázku je také naznačeno řešení spočívající v interpolaci pozice pozorovatele, jakmile se přiblížíme okraji textury.



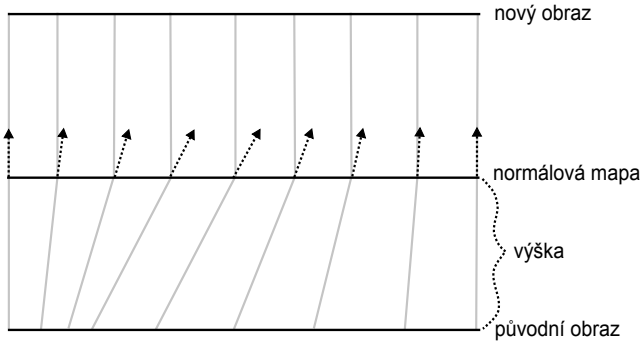
Obrázek 3.24: Phongův osvětlovací model způsobuje nenávaznosti na okraji textury způsobené spekulární složkou, řešením je interpolace pozice pozorovatele.

Lom světla

Normálové mapy nacházejí využití i jinde, než jen u osvětlovacích modelů. Promítneme-li obraz skrze normálovou mapu, která změní směr promítacích paprsků, dosáhneme efektu lámání světla, jako by se obraz nacházel např. za pokřiveným sklem. Tento postup naznačuje obrázek 3.26. Čím výše normálovou mapu nad modifikovaný obraz umístíme, tím bude efekt znatelnější.



Obrázek 3.25: Phongův osvětlovací model



Obrázek 3.26: Průchod promítacích paprsků normálovou mapou.

3.11 L-systémy

Podle literatury [6][11][10] definujme L-systémy (Lindenmayerovy systémy) jako systémy založené na přepisování řetězců umožňující popisovat komplexní geometrické tvary pomocí jednoduchých pravidel, jež lze reprezentovat gramatikami. L-systémy jsou vhodné pro generování struktur, jež lze nalézt v přírodě, většinou pak rostlin.

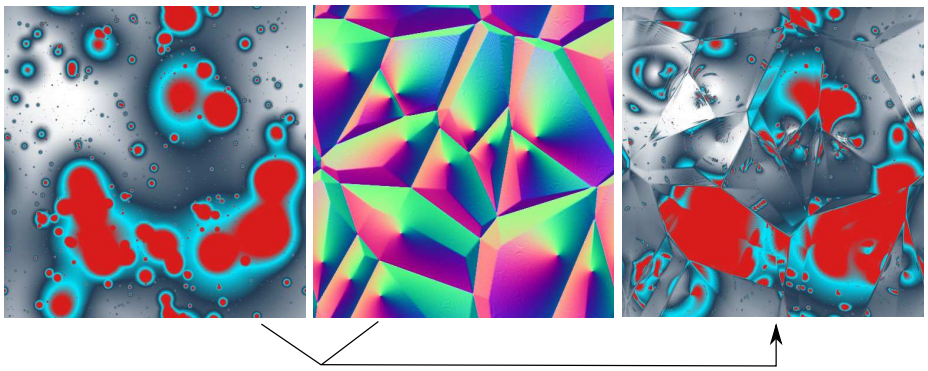
L-systémy lze různými způsoby rozdělit dle jejich vlastností [14]. Námi uvažovaný L-systém bude:

- bezkontextový – Používá bezkontextovou gramatiku.
- stochastický – V gramatice může existovat více pravidel se stejnou levou stranou. Které pravidlo se použije při přepisování, se určí náhodně (pravidla mohou mít explicitně uvedené rozdělení pravděpodobnosti).
- parametrický – Symboly gramatiky mohou mít určitý počet celočíselných parametrů.

Možnost parametrizovat symboly gramatiky umožňuje popsat postupné spojité změny tvaru nebo barvy.

Želví grafika

Želví grafika slouží ke grafické interpretaci řetězce generovaného gramatikou L-systému a jde o jeden z tradičních způsobů programování grafiky [6][10]. Název je odvozen ze způsobu,



Obrázek 3.27: efekt lámání světla – původní obraz, normálová mapa a výsledek

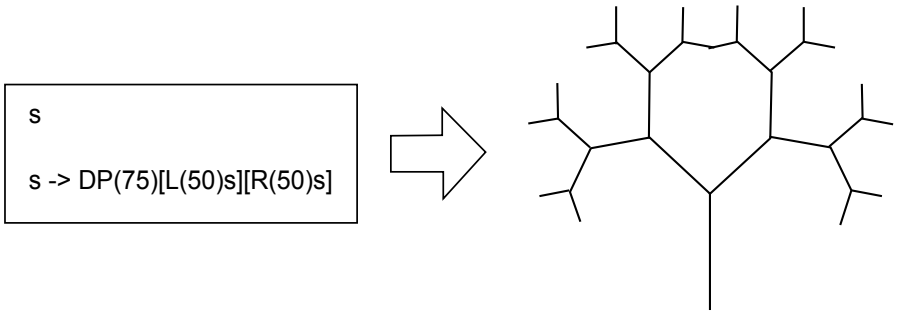
jakým k vykreslování řetězce dochází. Při něm využíváme tzv. želvu, nacházející se na dvourozměrném povrchu, definovanou svou polohou a úhlem natočení. Želva postupně čte znaky řetězce a interpretuje je jako kreslicí příkazy typu proveď krok vpřed s kreslením, otoč se o daný úhel apod. Také je schopná ukládat svůj stav na zásobník a později jej obnovovat, což je klíčová vlastnost při vykreslování některých fraktálních tvarů.

Želvu lze dále rozšířit o proměnné týkající se např. vykreslovacího štětce (tloušťka, barva, styl čáry apod.). Tímto v kombinaci s parametry gramatiky snadno dosahujeme např. plynulých barevných přechodů. Želva rozumí následujícím příkazům (jiné symboly, malá písmena a příkazy s nedostatečným počtem parametrů neinterpretuje):

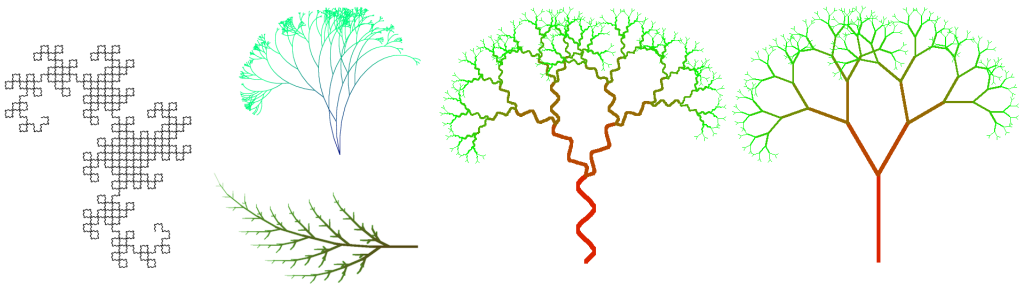
příkaz	**význam**
[	Ulož aktuální stav na zásobník.
]	Obnov stav ze zásobníku.
A(x)	Nastav úhel otočení na x stupňů.
L(x)	Otoč se o x stupňů doleva.
R(x)	Otoč se o x stupňů doprava.
C(r,g,b)	Nastav barvu na (r,g,b).
D	Kresli čáru dlouhou jeden krok.
G	Jdi jeden krok dopředu bez kreslení.
P(x)	Nastav délku kroku na x procent aktuální délky.
B(x)	Nastav délku kroku na x · 0,001 šířky obrázku.
I(x)	Zvětši délku kroku o x pixelů.
M(x)	Nastav délku kroku na x pixelů.
W(x)	Nastav tloušťku čáry na x · 0,001 šířky obrázku.
F(x)	Nastav tloušťku čáry na x pixelů.
S(x,y)	Nastav styl čáry na x a přerušování na y · 0,001 šířky obrázku.
T(x,y)	Nastav styl čáry na x a přerušování na y pixelů.

Styl čáry může nabývat hodnot:

hodnota	**význam**
0	rovná čára
1	klikatá čára
2	vlnitá čára
3	šipka
4	dvojitá čára
5	vykreslování částicemi



Obrázek 3.28: jednoduchý L-systém (5 iterací)

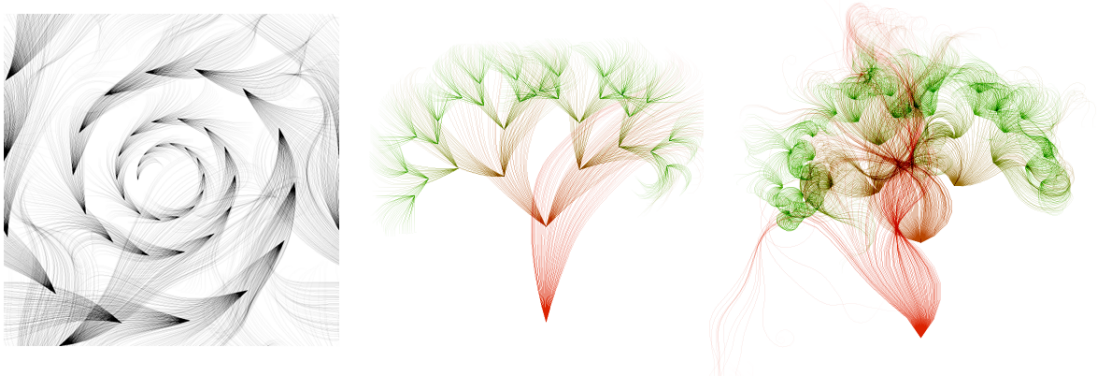


Obrázek 3.29: L-systémy

Kombinace s částicovými systémy

Zajímavou možností je pokusit se zkombinovat L-systémy s částicovými systémy. Lze např. umožnit použití částicového systému k vykreslení L-systému tak, že želví grafika namísto kreslení čar nechá proudit částice ve směru kreslení s počáteční rychlostí odpovídající délce čáry. Technicky lze tuto možnost implementovat jako speciální styl čáry, který želví grafika nastaví při přečtení definovaného symbolu z interpretovaného řetězce.

Problémem se ukázalo být nalezení takových parametrů částicového systému, které umožní rozpoznat strukturu vykreslovaného obrazce a přesto zachovají částečně nahodilý vzhled. Vliv šumu, jímž modulujeme pohyb částic, nesmí být příliš velký, ani příliš malý.



Obrázek 3.30: L-systémy a částicové systémy

Kapitola 4

Implementace

Tato kapitola probírá implementační část práce a uvádí použité technologie.

4.1 Knihovna

Jádro knihovny PT Designer je psáno v jazyku C, k dispozici je však rozhraní jak pro jazyk C, tak i C++, přičemž C++ rozhraní využívá výhody objektového přístupu, poskytuje vyšší úroveň abstrakce a je deklarativně orientované. Rozhraní jazyka C je orientováno imperativně.

Knihovna využívá dvou externích, volně dostupných knihoven, konkrétně LodePNG [25] pro práci s PNG soubory a RapidXml [23] jako XML parser. K automatické dokumentaci kódu je využit systém Doxygen [22], ke správě kódu Git [24] (dostupný na adrese `https://github.com/drummyfish/proctextures`).

Seznam zdrojových souborů

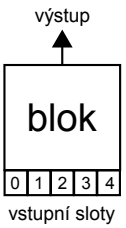
Jádro knihovny tvoří procedury modulu *proctextures.*$\{c|h\}$ realizující metody generování textur popsané výše v teoretické části. Tento zdrojový kód využívá další moduly, jako např. modul pro práci s bitmapami *colorbuffer.*$\{c|h\}$. Celkově se knihovna skládá z těchto zdrojových souborů:

Obrázek 4.1: logo knihovny vygenerované knihovnou

modul	popis
colorbuffer.{c|h}	základní práce s bitmapou (ukládání a načítání PNG, přístup k pixelům, ...)
colortransition.{c|h}	práce s barevnými přechody
general.{c|h}	obecné funkce a datové struktury (konstanty, transformace souřadnic, výpočet vzdáleností, ...)
grammar.{c|h}	práce s gramatikami
kdtree.{c|h}	práce s k-d stromy
linelist.{c|h}	pomocný lineární seznam využívaný některými algoritmy
lodepng.{c|h}	LodePNG knihovna
matrix.{c|h}	práce s maticemi
proctextures.{c|h}	algoritmy procedurálního generování textur
ptdesigner.{cpp|h}	C++ rozhraní knihovny, grafový popis textury
rapidxml.hpp	RapidXml knihovna
rapidxml_iterators.hpp	RapidXml knihovna
rapidxml_print.hpp	RapidXml knihovna
rapidxml_utils.hpp	RapidXml knihovna

Grafový popis textury

Texturu můžeme popsat pomocí grafu, v němž uzly, nebo také bloky, představují elementární operace, jakými jsou např. generování Voroného diagramu nebo aplikace barevného přechodu. Z formálního hlediska se jedná o orientovaný acyklický graf [7], jenž má navíc vstupy každého uzlu uspořádané pomocí tzv. slotů. Sloty představují vstupy bloku a jsou očíslovány, takže je možné je rozlišit. Blok má dále buď jeden nebo žádný výstup. [1]



Obrázek 4.2: blok

Výstupy bloků lze připojovat na sloty jiných bloků a vytvořit tak graf, jak prezentuje obrázek 4.3. Každý slot může mít připojen maximálně jeden vstup, avšak každý výstup může být připojen na neomezený počet slotů za podmínky, že v grafu nevznikne cyklus. Cykly je nutné detekovat při každém vzniku propojení a případné spojení neuskutečnit, jinak by se během výpočtu vyskytl nekonečný cyklus. Jedním ze způsobů, jak detekovat cyklus, je rekurzívní dohledávání předchůdců bloku, jehož výstup byl právě připojen – propojení způsobí cyklus právě tehdy, je-li po jeho provedení propojovaný blok svým vlastním předchůdcem.

Formát ukládání grafového popisu do souboru popisuje sekce 4.1.

Třídy C++ rozhraní

Rozhraní jazyka C++ nabízí třídy pro snadný návrh textur. Tyto třídy využívají procedury implementované v jazyku C a navíc se starají o konzistenci popisu textury jako celku, tzn. automaticky provádí kroky výpočtu textury ve správném pořadí, rozlišení, s minimem redundance atd.

Základní třídou je abstraktní třída *c_block* představující blok, tedy uzel grafu popisující jednu operaci. [2] Blok má jeden výstup a určité množství vstupů, díky čemuž je možné

[1] Příklady bloků bez výstupů jsou např. uložení do souboru nebo tzv. koncový blok sloužící k programovému přístupu k výsledné textuře.

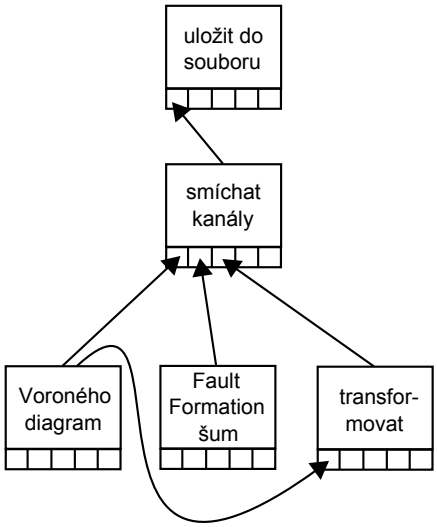
[2] viz část 4.1

vytvářet mezi nimi spojení. Blok disponuje schopností rozpoznat a odmítnout propojení, které by způsobilo cyklus v grafu.

Dále má blok určité parametry, které jsou závislé na tom, jakou operaci představuje. Parametry bloku realizujícího osvětlovací model mohou být např. barva a směr světla. Jelikož je s parametry nutné pracovat i mimo samotný programovací jazyk, např. když jsou uchovávány v souboru, využívá třída *c_block* k jejich správě třídu *c_parameters*. Díky ní je možné zjistit, kolik parametrů blok očekává, jak se jmenují a jaké jsou jejich datové typy, což je velmi dobře využitelné při implementaci grafického nástroje, který může na základě těchto informací automaticky generovat dialogy pro editaci parametrů. Třída *c_parameters* podporuje čtyři datové typy: celé číslo, číslo v plovoucí řádové čárce, pravdivostní hodnotu a řetězec znaků.

Od třídy *c_block* dále dědí třídy *c_graphic_block* a *c_special_block*. Třída *c_graphic_block* reprezentuje blok, který uchovává vlastní bitmapu, k níž je možné přistupovat. Jedná se o základní třídu pro většinu bloků. Třída *c_special_block* představuje blok, který vlastní bitmapu nepotřebuje a poskytuje na výstupu jiný typ dat. Jedná se např. o barevný přechod nebo L-systém poskytující pouze vygenerovaný textový řetězec.

Konkrétní bloky, jako jsou např. *c_block_perlin_noise* nebo *c_block_voronoi_diagram*, jsou potomky buď třídy *c_graphic_block* nebo *c_special_block*. Programátor může tímto způsobem velmi jednoduše přidávat vlastní bloky a knihovnu tak rozšiřovat. Jediné, co musí udělat, je zdědit třídu svého bloku od jedné z výše uvedených tříd a předefinovat kód virtuálních metod *execute* a *set_default_parameters*. Metoda *set_default_parameters* nastaví implicitní parametry bloku a metoda *execute* slouží k výpočtu výstupu bloku.



Obrázek 4.3: příklad grafového popisu

Sjednocující třídou je třída *c_texture_graph*, která se stará o propojené bloky a zajišťuje jejich spolupráci. Uchovává informace o textuře, jako je její rozlišení nebo úroveň supersamplingu, a je schopná poznat, které bloky je v případě opětovného výpočtu po provedené změně třeba přepočítat. Více informací o třídách znázorňuje UML diagram 4.6.

Formát popisového souboru

Třída *c_texture_graph* umožňuje ukládat a načítat popis textury jako XML soubor odpovídající následujícímu DTD popisu [13]:

```
<!ELEMENT texturegraph (block*)>
<!ELEMENT block ((input | parameter)*)>
<!ELEMENT input EMPTY>
<!ELEMENT parameter EMPTY>

<!ATTLIST texturegraph width NMTOKEN #REQUIRED>
<!ATTLIST texturegraph height NMTOKEN #REQUIRED>
<!ATTLIST texturegraph seed NMTOKEN #REQUIRED>
```

```
<!ATTLIST texturegraph supersampling NMTOKEN #REQUIRED>

<!ATTLIST block id ID #REQUIRED>
<!ATTLIST block type CDATA #REQUIRED>

<!ATTLIST input id NMTOKEN #REQUIRED>
<!ATTLIST input slot NMTOKEN #REQUIRED>

<!ATTLIST parameter name CDATA #REQUIRED>
<!ATTLIST parameter type (int | double | string | bool) #REQUIRED>
<!ATTLIST parameter value CDATA #REQUIRED>
```

Gramatika

Implementovaný L-systém čte pravidla gramatiky ze souboru. První řádek představuje axiom (počáteční řetězec), následuje prázdný řádek a řádky pravidel, jejichž syntaxe je dána gramatikou:

<START>	→	<LEFT> <CHANCE> -> <RIGHT>
<LEFT>	→	<LETTER> | <LETTER> (<PARAM_LIST>) <CONDITION>
<PARAM_LIST>	→	<LETTER> | <LETTER>, <PARAM_LIST>
<CONDITION>	→	[<EXPRESSION>] | ”
<RIGHT>	→	” | <RIGHT_NONEMPTY>
<RIGHT_NONEMPTY>	→	<SYMBOL> | <SYMBOL> <RIGHT_NONEMPTY>
<SYMBOL>	→	<LETTER> | <LETTER> (<PARAM_ASSIGN>)
<PARAM_ASSIGN>	→	<EXPRESSION> | <EXPRESSION>, <PARAM_ASSIGN>
<CHANCE>	→	: <NUMBER> | ”
<EXPRESSION>	→	<OPERAND> | <OPERAND> <OPERATOR> <OPERAND>
<OPERATOR>	→	+ | - | * | / | & | ’|’ | ! | > | < | =
<OPERAND>	→	<NUMBER> | <LETTER> | (<EXPRESSION>)
<NUMBER>	→	1 | 2 | 3 | . . . | 10 | 11 | . . .
<LETTER>	→	a | A | b | B | . . . | z | Z | { | }

4.2 Grafický nástroj

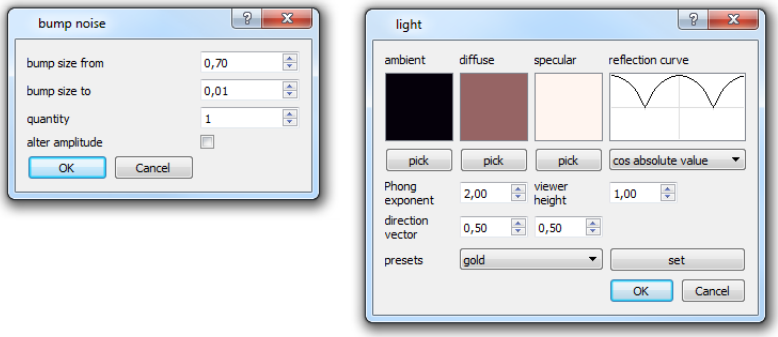
Aplikace pro návrh textur (Texturemaker) byla implementována v prostředí QT [26] verze 5.2.0. [3] Je ji možné zkompilovat na všech platformách, které toto prostředí podporují; mezi tyto platformy patří operační systémy Windows a Linux. [4] Uživatel pomocí ní může interaktivně navrhnout texturu použitím grafového popisu zmíněného v části 4.1. Během návrhu je možné vidět všechny bloky, jejich propojení, stav a další detaily včetně náhledu jejich výstupu. Nástroj dále umožňuje nastavovat parametry bloků a globální informace, jako jsou rozlišení, úroveň supersamplingu nebo seed náhodného generátoru.

K editaci parametrů bloků slouží modální dialogy. Nejdříve byl implementován implicitní univerzální dialog umožňující editaci parametrů libovolného bloku formou tabulky, která je schopná díky třídě *c_parameters* zjistit, jaké parametry blok vyžaduje a jakých jsou datových typů. Tento dialog je možné využít například tehdy, přidá-li programátor do knihovny vlastní blok a nepotřebuje pro něj implementovat nový dialog. Parametry některých bloků jsou však komplikovanější a je výhodnější mít pro ně zvláštní dialog. Proto byly

[3] Prostředí využívá modifikovanou verzi jazyka C++.

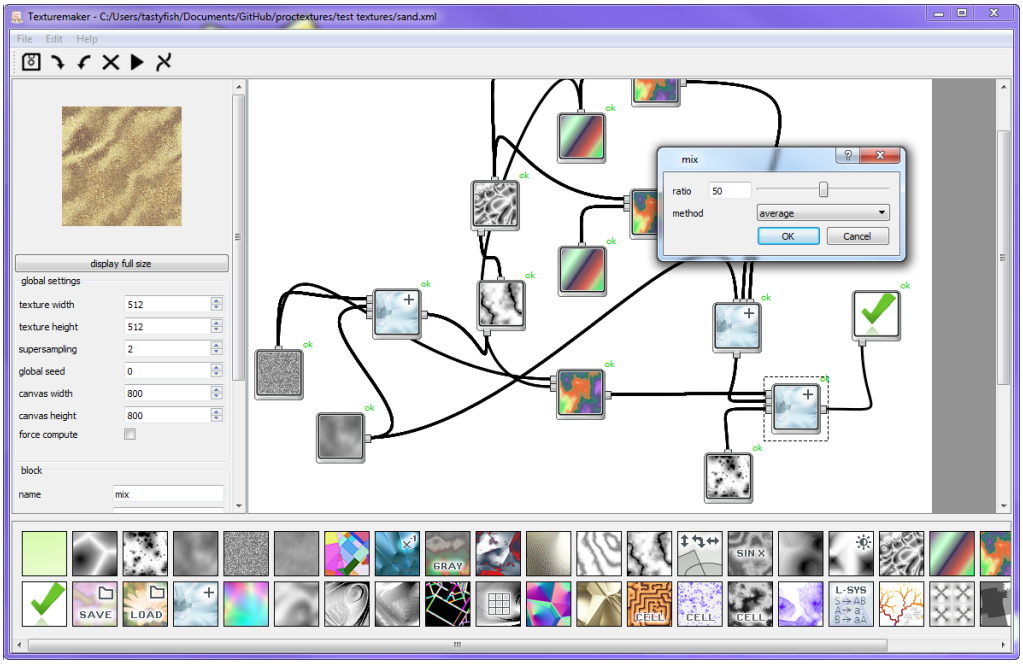
[4] Při kompilaci pro systém Windows je doporučeno sestavení se statickým linkováním (nejprve se musí staticky zkompilovat jádro QT).

pro většinu bloků implementovány speciální dialogy umožňujících např. vlastní rozmístění bodů Voroného diagramu nebo návrh tvaru dlaždice u teselace.

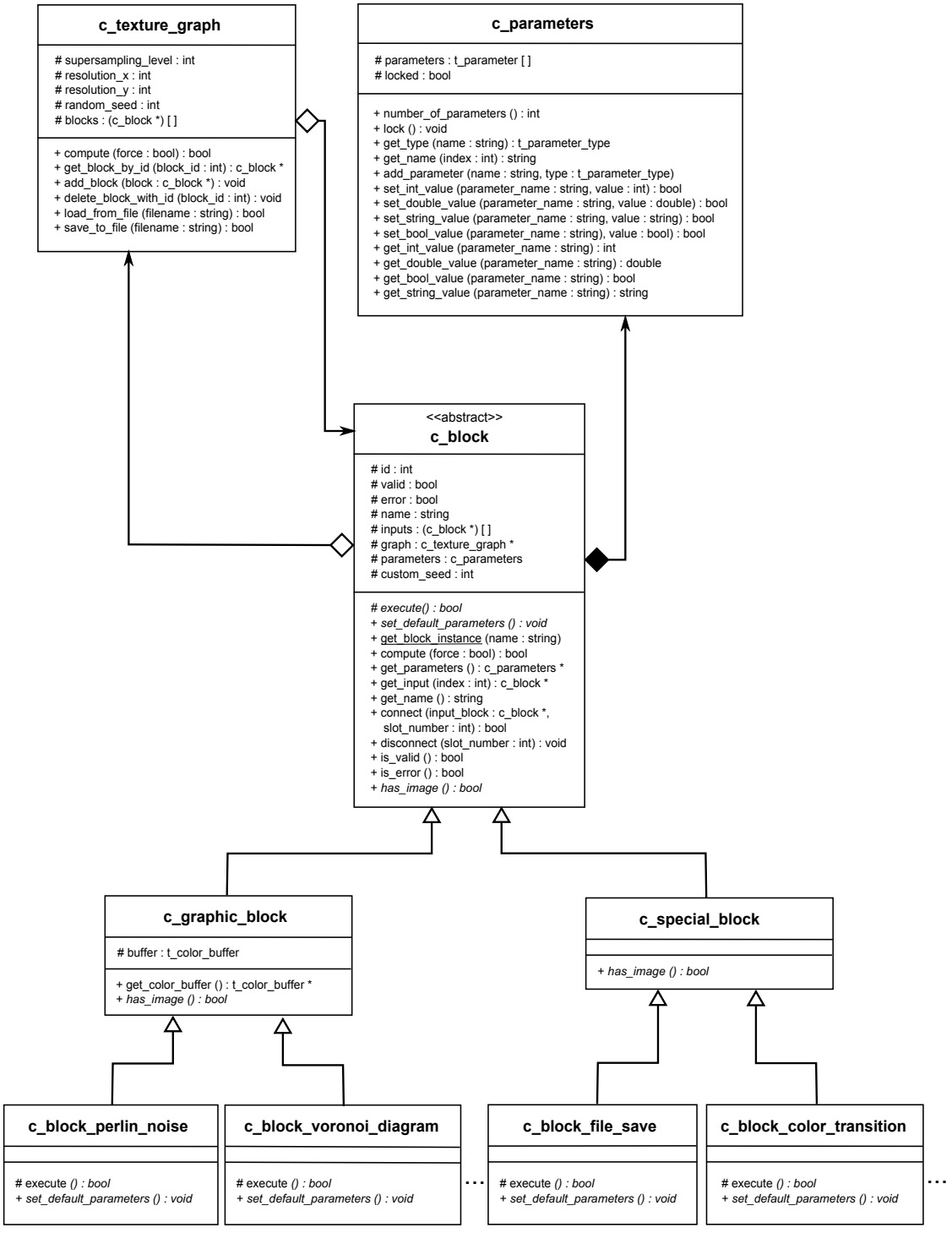


Obrázek 4.4: implicitní versus specializovaný dialog

Texturu lze během návrhu uložit jako XML popisový soubor nebo přímo spustit její generování, které umožní prohlížení výsledku či jeho uložení do PNG souboru. Generování může v závislosti na komplexnosti textury trvat nezanedbatelnou dobu, proto probíhá v odděleném vlákně. Během výpočtu se zobrazuje jeho průběh. Jelikož v programu existuje jeden globální objekt třídy *c_texture_graph*, k němuž může být přistupováno současně z více míst v kódu (vlákno výpočtu, náhled, dialogy editace parametrů apod.), musí být nějakým způsobem zajištěn výlučný přístup. Tento problém řeší třída *QMutex* nabízená prostředím QT.



Obrázek 4.5: grafický nástroj pro návrh textur Texturemaker



Obrázek 4.6: diagram tříd knihovny

Kapitola 5

Závěr

Knihovna i aplikace jsou implementovány způsobem splňujícím všechna hlavní kritéria zadání, přičemž byl brán ohled i na praktickou použitelnost, rozšiřitelnost a přehlednost kódu.

Knihovna PT Designer má řádově okolo 10 000 řádků vlastního kódu a nabízí celkem 44 bloků využitelných pro tvorbu textur. Nabízí dvě rozhraní na různé úrovni abstrakce a poskytuje následující metody generování textur:

- obecné grafické operace – filtry, rozptylování, RGB-HSL převod, jas/kontrast, barevné přechody, prolínání bitmap, míchání RGB kanálů, supersampling, ...
- netradiční algoritmy – Substrate, transformace, promítání přes normálovou mapu, ...
- generování různých druhů šumů – Perlinův, Fault Formation, ...
- generování Voroného diagramů – různé metriky, způsoby vizualizace a rozmisťování řídících bodů (náhodné, pravidelné, výčtem, pomocí L-systému)
- popis a vykreslování L-systémů a možnost jejich kombinace s částicovými systémy
- vykreslování částicových systémů
- výpočet normálových map a Phongova osvětlovacího modelu
- různé druhy celulárních automatů – cyklické, RPS, obecné binární
- popis a vykreslování teselace pomocí Escherových mozaiek

Grafický nástroj Texturemaker umožňuje relativně snadný návrh textur, uživateli nabízí všechny bloky knihovny a jejich parametry se dají jednoduše měnit pomocí dialogů, z nichž 20 je ručně navržených a zbytek je automaticky generovaný.

Kromě ladění a optimalizace kódu by práce samozřejmě mohla pokračovat různými rozšířeními. V úvahu přichází např.:

- přidávání nových algoritmů (např. více druhů šumů)
- možnost generovat 3D textury zobecněním použitých algoritmů
- podpora přidávání bloků formou plug-inů (bez nutnosti znovu kompilovat zdrojové kódy)
- možnost generovat animované textury:

  - specifikováním hodnot parametrů nikoli neměnnou hodnotou, ale funkcí času
  - generováním sekvence postupných průřezů vícerozměrnou texturou
- optimalizace rychlosti specializovanými procesorovými instrukcemi (MMX, SSE)
- využití grafické karty a shaderů pro výpočty
- jednoduchá syntéza textur
- podpora více grafických formátů (v současnosti pouze PNG)
- oproti generování pouze obdelníkových textur přidat možnost specifikovat tvar textury a místa, kde má navazovat, pro mapování na konkrétní složitější geometrii
- vylepšení uživatelského rozhraní nástroje (akce zpět/vpřed, označování více bloků apod.)

V současné podobě se dá software úspěšně použít např. pro generování pozadí ve 2D grafice nebo texturování ne příliš složitých 3D objektů typu terén, zdi apod. Složitější objekty, jako je např. koule, není jednoduché generovanou 2D texturou pokrýt, což by vyřešily 3D textury pomocí výše zmíněného rozšíření. Existuje také možnost načíst již existující obrázek ve formátu PNG a dále s ním pracovat, proto se dá knihovna použít třeba jen k úpravě již existujících, např. člověkem vytvořených textur, nebo čistě jako obecná knihovna pro práci s bitmapami. Celá práce bude veřejně k dispozici pod licencí GPL.

Literatura

[1] AKBAR, M. A.; FAROOQ, M.: RTP-miner: a real-time security framework for RTP fuzzing attacks. In *NOSSDAV '10*.

[2] ALUNING, R. C. A.; HERMOCILLA, J. A. C.: Terra: A 3D Terrain Generator and Visualizer. In *NCITE 2010*.

[3] AURENHAMMER, F.: Voronoi diagrams – a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 1991.

[4] BARTSCH, H. J.: *Matematické vzorce*. Mladá Fronta, třetí vydání, 2002, ISBN 80-204-0607-7.

[5] BARYSHNIKOV, Y. M.; COFFMAN, E. G.; KWAK, K. J.: High Performance Sleep-Wake Sensor Systems Based on Cyclic Cellular Automata. In *IPSN '08*.

[6] BENEŠ, B.; FELKEL, P.; SOCHOR, J.; aj.: *Moderní počítačová grafika*. Computer Press, druhé vydání, 2010, ISBN 80-251-0454-0.

[7] DEMEL, J.: *Grafy a jejich aplikace*. Academia, 2002, ISBN 80-200-0990-6.

[8] DUFF, B.: Gen #26: the mod attribute. *ACM SIGAda Ada Letters*, 2009.

[9] ESTEBAN, P. G.; PATÓN, A. R.: Simulating a rock-scissors-paper bacterial game with a discrete cellular automaton. In *IWINAC'11*.

[10] F. S. HILL, J.: *Computer Graphics Using OpenGL*. Prentice Hall, druhé vydání, 1990, ISBN 0-02-354856-8.

[11] FOLEY, J. D.; van DAM, A.; FEINER, S. K.; aj.: *Computer Graphics: Principles and Practice*. Addison-Wesley, druhé vydání, 2003, ISBN 0-201-84840-6.

[12] KAPLAN, C. S.; SALESIN, D. H.: Escherization. In *SIGGRAPH '00*.

[13] KOSEK, J.: *XML pro každého: podrobný průvodce*. Grada Publishing, 2000, ISBN 80-7169-860-1.

[14] KVITA, J.: Generátor 3D objektů s využitím L-systémů. 2013.

[15] MILET, T.: Grafické intro 64kb s použitím OpenGL. 2012.

[16] QADEER, W.; HAMEED, R.; SHACHAM, O.; aj.: Convolution engine: balancing efficiency & flexibility in specialized computing. In *ISCA '13*.

[17] SCHIFF, J. L.: *Cellular Automata: A Discrete View of the World*. Wiley, 2008, ISBN 978-0-470-16879-0.

[18] SOUSEDÍK, C.: Syntéza textur. 2010.

[19] VANÍČEK, J.; PAPÍK, M.; PERGL, R.; aj.: *Teoretické základy informatiky*. Kernberg Publishing, 2007, ISBN 978-80-903962-4-1.

[20] van WIJK, J. J.: Spot noise texture synthesis for data visualization. In *SIGGRAPH '91*.

[21] Cellular automata rules lexicon - 1-dimensional totalistic. http://psoup.math.wisc.edu/mcell/rullex_1dto.html, viděno 10.4.2014.

[22] Doxygen: Main page. http://www.stack.nl/~dimitri/doxygen/index.html, viděno 27.1.2014.

[23] RapidXml. http://rapidxml.sourceforge.net/, viděno 15.1.2014.

[24] Git. http://git-scm.com/, viděno 9.2.2014.

[25] LodePNG. http://lodev.org/lodepng/, viděno 15.1.2014.

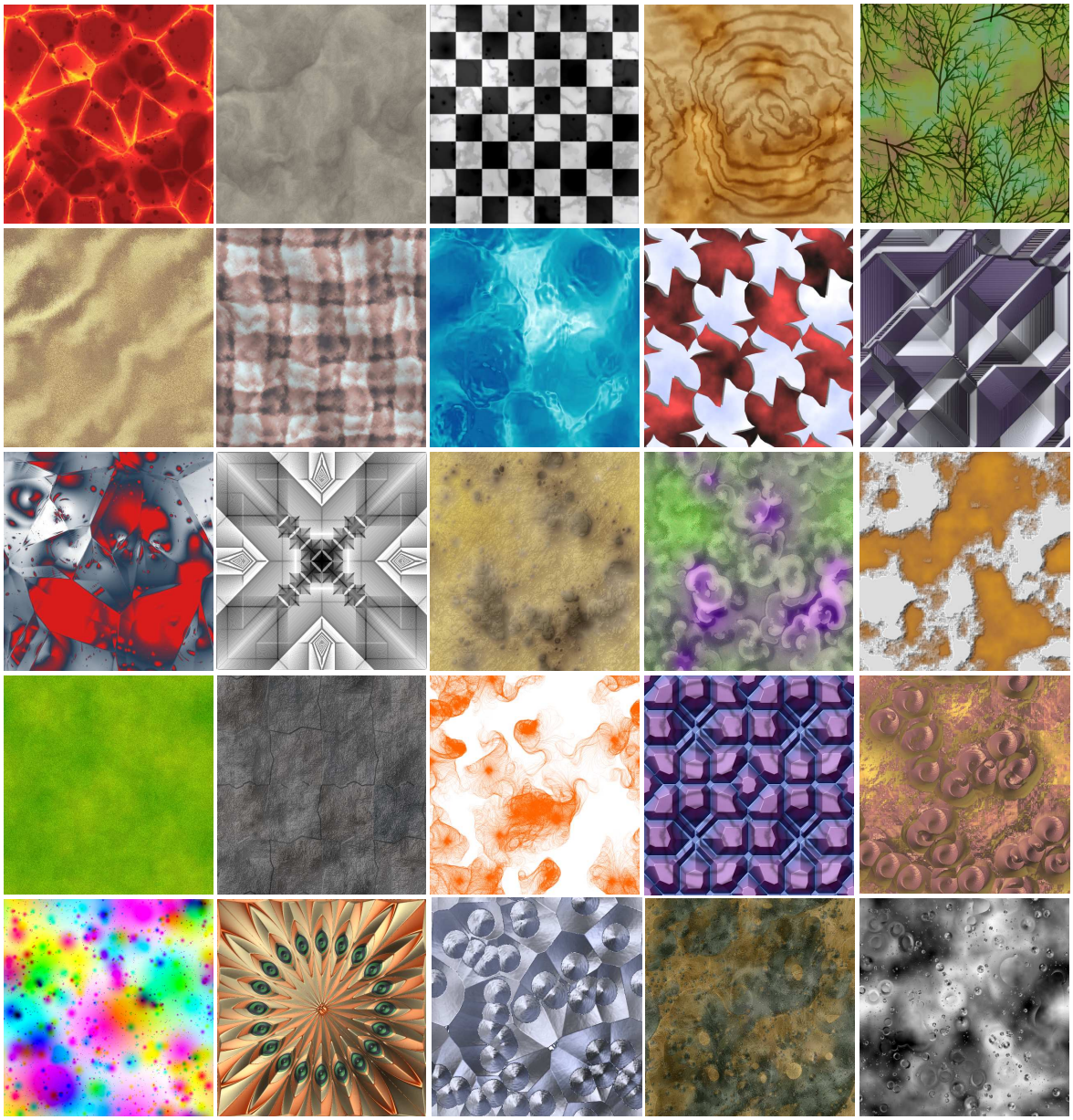
[26] Qt Project. http://qt-project.org/, viděno 15.1.2014.

[27] Another cellular automaton video - GameDev.net. http://www.gamedev.net/blog/844/entry-2249737-another-cellular-automaton-video/, viděno 10.4.2014.

[28] Substrate — Gallery of Computation. http://www.complexification.net/gallery/machines/substrate/, viděno 15.1.2014.

Dodatek A

Vygenerované textury



Dodatek B

Ukázka kódu

Následuje ukázka použití knihovny k vygenerování textury v rozhraní pro jazyk C a ekvivalentní kód v C++ rozhraní.

```c
#include "proctextures.h"

int main()
{
  t_color_buffer buffer1, buffer2, buffer3, buffer4;
  color_buffer_init(&buffer1,512,512);
  color_buffer_init(&buffer2,512,512);

  pt_bump_noise(&buffer1,0.7,0.01,1,0,0);
  pt_voronoi_diagram_simple(0,20,&buffer2);
  pt_transformation_circle(&buffer2,5,1,&buffer3);
  pt_mix_channels(&buffer1,&buffer2,&buffer3,&buffer4);

  color_buffer_save_to_png(&buffer4,"texture.png");

  color_buffer_destroy(&buffer1);
  color_buffer_destroy(&buffer2);
  color_buffer_destroy(&buffer3);
  color_buffer_destroy(&buffer4);

  return 0;
}
```

```c
#include "ptdesigner.h"
using namespace pt_design;

int main()
{
  int ids[5];
  c_texture_graph graph;

  graph.set_resolution(512,512);
```

```
  ids[0] = graph.add_block(new c_block_bump_noise());
  ids[1] = graph.add_block(new c_block_voronoi_diagram());
  ids[2] = graph.add_block(new c_block_circle_transform());
  ids[3] = graph.add_block(new c_block_mix_channels());
  ids[4] = graph.add_block(new c_block_file_save());

  graph.get_block_by_id(ids[1])->get_parameters()->
    set_int_value("number_of_points",20);

  graph.connect_by_id(ids[0],ids[3],0);
  graph.connect_by_id(ids[1],ids[3],1);
  graph.connect_by_id(ids[2],ids[3],2);
  graph.connect_by_id(ids[1],ids[2],0);
  graph.connect_by_id(ids[3],ids[4],0);

  graph.compute(false);
  return 0;
}
```