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

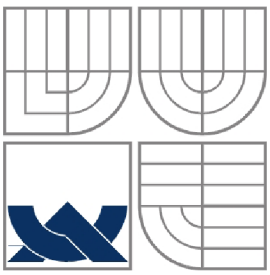
KNIHOVNA PRO VÝPOČET ŠUMŮ POUŽÍVANÝCH V PROCEDURÁLNÍM TEXTUROVÁNÍ

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

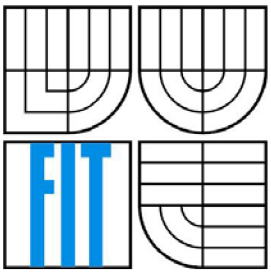
AUTOR PRÁCE
AUTHOR

Ondřej KUČERA

BRNO 2007



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

KNIHOVNA PRO VÝPOČET ŠUMŮ POUŽÍVANÝCH V PROCEDURÁLNÍM TEXTUROVÁNÍ

LIBRARY FOR EVALUATION OF NOISES USED IN PROCEDURAL TEXTURING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Ondřej KUČERA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. Adam HEROUT, Ph.D.

BRNO 2007

Zadání diplomové práce/5625/2006/xkucer35

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2006/2007

Zadání diplomové práce

Řešitel: **Kučera Ondřej**
Obor: Výpočetní technika a informatika
Téma: **Knihovna pro výpočet šumů používaných v procedurálním texturování**
Kategorie: Počítačová grafika

Pokyny:

1. Seznamte se s problematikou procedurálního texturování s využitím šumu.
2. Prostudujte používané šumy, popište jejich vlastnosti, použití a výpočetní a paměťovou náročnost.
3. Navrhněte a zdůvodněte rozhraní knihovny, která by obsahovala efektivní implementace vybraných šumů, zejména v implicitní podobě.
4. Implementujte navrženou knihovnu.
5. Vytvořte sadu demonstračních příkladů a dokumentaci ke knihovně.
6. Zhodnoťte dosažené výsledky a navrhněte možnosti pokračování projektu.

Literatura:

- podle pokynů vedoucího

Při obhajobě semestrální části diplomového projektu je požadováno:

- Bez požadavků.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci ročníkového a semestrálního projektu (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním paměťovém médiu (disketa, CD-ROM), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Herout Adam, Ing., Ph.D.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2006
Datum odevzdání: 22. května 2007

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Pavel Zemčík
vedoucí ústavu

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

1. Pan

Jméno a příjmení: **Ondřej Kučera**
Id studenta: 21608
Bytem: Hradec nad Svitavou 497, 569 01 Hradec nad Svitavou
Narozen: 29. 01. 1982, Svitavy
(dále jen "autor")

a

2. Vysoké učení technické v Brně

Fakulta informačních technologií
se sídlem Božetěchova 2/1, 612 66 Brno, IČO 00216305
jejímž jménem jedná na základě písemného pověření děkanem fakulty:

.....
(dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
diplomová práce

Název VŠKP: Knihovna pro výpočet šumů používaných v procedurálním texturování
Vedoucí/školitel VŠKP: Herout Adam, Ing., Ph.D.
Ústav: Ústav počítačové grafiky a multimédií
Datum obhajoby VŠKP: .....

VŠKP odevzdal autor nabyvateli v:

tištěné formě počet exemplářů: 1
elektronické formě počet exemplářů: 2 (1 ve skladu dokumentů, 1 na CD)

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2
Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti:
   - ☒ ihned po uzavření této smlouvy
   - ☐ 1 rok po uzavření této smlouvy
   - ☐ 3 roky po uzavření této smlouvy
   - ☐ 5 let po uzavření této smlouvy
   - ☐ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3
Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: .....

....

Nabyvatel Autor

Abstrakt

V této práci se čtenář seznámí s problematikou procedurálního texturování a využití šumu pro tvorbu textur materiálů reálného světa. Pozná význam šumu, jeho vlastnosti a způsob využití. Při výpočtech šumu jsou nedílnou součástí náhodná čísla a interpolační metody, proto nechybí kapitoly o generátorech náhodných čísel a interpolačních metodách. Samozřejmostí je také popis vlastností a principů jednotlivých metod výpočtu šumu. Hlavní součástí této práce je efektivní implementace knihovny vybraných šumů. Další kapitoly jsou o výběru metod a návrhu rozhraní, samotné implementaci, a aby byla knihovna efektivní, tak se lze dočíst o různých druzích testů a optimalizací. Na závěr jsou uvedeny dosažené výsledky po optimalizacích a celkové shrnutí práce.

Klíčová slova

Procedurální texturování, šum, Perlinův šum, fraktální suma, turbulence, interpolace, náhodná čísla, textura mramoru, textura dřeva, statická knihovna, hlavičkový soubor, makro

Abstract

The aim of the work is to describe procedural texturing and usage of a noise for creating textures of real materials. The reader will learn the fundamentals of the noise, its properties and a way of using it. Random numbers and interpolation methods are important parts of the noise evaluations, therefore there are chapters about random numbers generating and interpolation methods. Obviously, there is not missing the chapter, which is depicting properties and principles of several methods of creating noise. The main part of this work is effective implementation of library with a chosen noise methods, so next chapters are about choosing methods and design of the interface, implementation and many kinds of tests and optimalizations. Achieved results and final conclusions are at the end of this work.

Keywords

Procedural texturing, noise, Perlin noise, turbulence, interpolation, random numbers, marble texture, wood texture, static library, header file, macro

Citace

Ondřej Kučera: Knihovna pro výpočet šumů používaných v procedurálním texturování, diplomová práce, Brno, FIT VUT v Brně, 2007

Knihovna pro výpočet šumů používaných v procedurálním texturování

Prohlášení

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

……
Jméno Příjmení
Datum

Poděkování

Tímto způsobem bych chtěl poděkovat vedoucímu Ing. Adamu Heroutovi, Ph.D. za kvalitní a odborné vedení při zpracovávání této diplomové práce. Za jeho trpělivost a čas, který mi při konzultacích věnoval.

Obsah

1 Úvod

Pro mnoho lidí je pojem „Knihovna pro výpočet šumů“ nepředstavitelný, ovšem ti, kteří se pohybují v oblasti programování a počítačové grafiky, mají určitě jasno.

Onou knihovnou není samozřejmě myšlena klasická dřevěná knihovna, která je plná zajímavých knih, ale je to soubor důležitých procedur a funkcí s definovaným rozhraním, vyčleněný pro společné použití různými programy. Tyto programy používají právě rozhraní, které definuje implementované funkce v knihovně. Pro uživatele není nutné znát vlastní implementaci, ale pouze názvy jednotlivých funkcí a jejich parametry. Knihovna může také být v jednom hlavičkovém souboru, který je zároveň rozhraním.

Šumu se ve většině odvětví snaží předejít nebo jej odstranit. Jedním z nejčastějších je šum ve zvuku nebo šum ve fotografiích, kde je opravdu nežádoucí. Jedná-li se ale o šum v procedurálním texturování, je vypočítáván za účelem vytvoření náhodnosti, která tvoří podstatnou část textur napodobující materiály reálného světa (dřevo, mramor, …) nebo za účelem náhodného rozmístění a pohybu objektů (mraky, oheň) a ke generování terénu.

Cílem této práce je vytvoření knihovny a jejího rozhraní, která bude implementovat výpočet jednotlivých metod šumu. Tuto knihovnu pak lze libovolně použít v kterémkoli programu pro výpočet procedurálních textur dřeva, mramoru a jiných materiálů, ohně, mraků a dalších různých animací.

Následující kapitola uvede čtenáře do problému procedurálního texturování a šumu. Jsou zde vysvětleny jednotlivé vlastnosti, charakteristiky a požadavky a v neposlední řadě, skládání šumových funkcí.

Třetí kapitola pojednává o výpočtu šumu, který je založen na náhodných číslech a interpolaci. Jednotlivé podkapitoly pojednávají právě o náhodných číslech a jejich generátorech, interpolačních a šumových metodách. Náhodná čísla jsou hlavní součástí všech šumů a tak je třeba vysvětlit jejich vlastnosti a principy generování. Interpolační funkce jsou využity při interpolaci dvou a více hodnot, protože šumová funkce vrací pouze jedno reálné číslo. Je vysvětlen jejich základní princip, efektivnost a využitelnost. Další podkapitola popisuje jednotlivé šumové metody, jejich vlastnosti, nevýhody a výpočetní složitosti. U těch známějších je vysvětlen i podrobný princip výpočtu.

Čtvrtá kapitola nabízí návrh rozhraní pro knihovnu. Jedná se o výběr metod a deklaraci funkcí, které budou implementovány. Tato kapitola byla poslední kapitolou semestrálního projektu a doznala malých změn. Více uvnitř kapitoly.

Pátá kapitola popisuje implementaci jednotlivých funkcí. Začíná výběrem programovacího jazyka, následuje stručný popis principů jednotlivých metod a možnosti využití programovacích struktur a nástrojů jazyka. Nechybí rozbor paměťové náročnosti.

Následující kapitola je největší a nejdůležitější kapitolou celé práce. Pojednává o testování a optimalizacích jednotlivých funkcí, o dosažených výsledcích a vlivu konfigurace testovací sestavy. Na úvod je popsána konfigurace základní sestavy, metodika testování a testovací podmínky. Následují první testy a naměřené hodnoty. Po prvních testech začínají první optimalizace, testy, grafy a zhodnocení výsledků. Velkou optimalizací je poměr mezi float a double, kvůli kterému je knihovna kompletně přeprogramována. Velký vliv v tomto případě má také konfigurace testovací sestavy. I tyto testy byly provedeny.

Sedmá kapitola nabízí možnosti rozšíření knihovny o další funkce spojené s výpočtem šumů. První návrh je o funkcích, které vrací pole hodnot a v druhém návrhu jsou funkce, jejichž návratové hodnoty jsou vektory.

Osmá kapitola ukáže možnosti využití knihovny v praxi. Samozřejmostí jsou ukázky textur materiálů, které jsou vygenerovány pomocí knihovny a přiloženého programu.

Práci ukončuje závěrečná kapitola s celkovým zhodnocením, možnostmi pokračování a doplnění knihovny o další funkce. Na úplný konec nechybí seznam literatury a příloh.

Tato diplomová práce je postavena na předchozí semestrální práci, na kterou navazuje. Ze semestrální práce je převzat teoretický rozbor, popis jednotlivých funkcí a návrh rozhraní. Diplomová práce čerpá právě z těchto informací, proto jsou zde opět uvedeny a nelze je opomenout. Tyto kapitoly byly shrnuty do jedné „třetí“ kapitoly s názvem „Výpočet šumů“. Ve čtvrté kapitole, která byla poslední šestou kapitolou v semestrálním projektu, je výběr metod, které mají být implementovány, a předběžný návrh rozhraní pro knihovnu. Na základě tohoto výběru a rozhraní pokračuje diplomová práce v implementaci funkcí do knihovny.

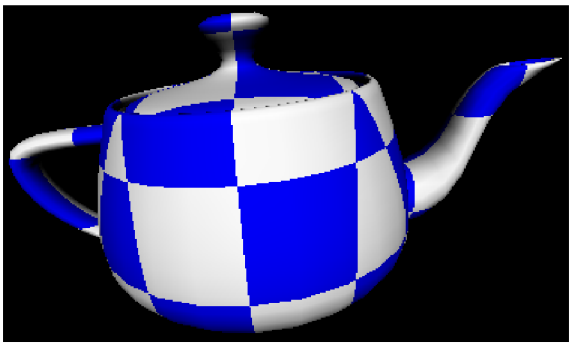
2 Šum v procedurálním texturování

Nejdříve je důležité vysvětlit samotné texturování [3]. Je to metoda, při které se na 2D povrch nebo 3D těleso namapuje textura, která může být brána buď jako obrázek nebo ji lze vytvořit procedurálně za běhu programu. Dále je důležité zdůraznit, že nanášením textur se nikterak nemění geometrické vlastnosti objektů.

Obrázkové textury většinou vznikají namalováním nebo skenováním. Jejich výhodou je, že je vidět jejich vzhled a snadnější přístup k jednotlivým texelům (pixely v textuře). Také jejich nanesení je ve většině případů jednodušší, ale také hodně závisí na tvaru tělesa nebo plochy. Nevýhodou těchto textur je pevná velikost rozlišení a také hlavně velikosti souborů s těmito texturami.

U procedurálního texturování je textura buď předpočítána ještě před samotným nanesením a pak se chová jako klasická textura nebo se její výpočet provádí v reálném čase s okamžitým nanášením. Tyto textury jsou vytvářeny pomocí procedur nebo funkcí, které „obarvují“ jednotlivé pixely podle matematického zápisu. Proto procedurální nebo také matematické textury. Těmito texturami pak lze jednoduše realizovat opakující se vzory, jako je např. cihlová zeď, plot nebo šachovnice (Obrázek 1.).

Výhodou procedurální textury je malá velikost kódu, nemá pevné rozlišení (snadné zobrazení detailů) a může se v průběhu výpočtu měnit. Nevýhodou je složitější programování, není vidět výsledný vzhled a také výpočet může být dosti náročný. Pomocí procedurální textury lze ale vytvářet jakýkoliv obrázek nebo 3D model. Takto se například modelují mraky, oheň, struktura dřeva apod. Vytvoření těchto textur je postaveno na syntéze šumu.

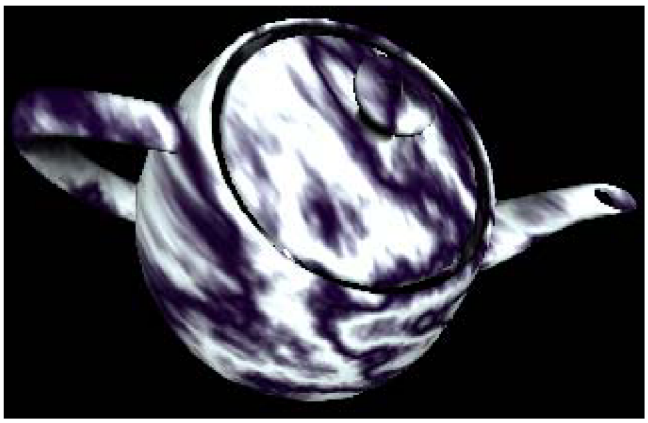


Obrázek 1.: Příklad procedurální textury

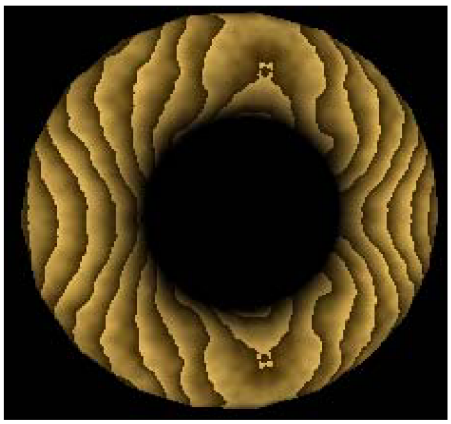
2.1 Šum

V mnoha oborech a odvětvích lze najít různé druhy šumu. Jsou šumy v elektrotechnice, ve zvuku, signálech, obrázcích a taky grafice. Někde jsou potřebné jindy naopak nežádoucí. Tato práce se bude zabývat analýzou a následným výpočtem šumu pro procedurální texturování.

Cílem je tedy subjektivně nahodilý vzhled, který napodobí komplikované přírodní materiály, jako je mramor, dřevo, kámen, atd. (Obrázek 2. a 3.).



Obrázek 2.: Textura mramoru pomocí šumu



Obrázek 3.: Textura dřeva

Šum je vypočítán pseudonáhodným algoritmem a měl by být v libovolném bodě deterministický. Pokud se kdykoli provede výpočet pro dané hodnoty, funkce by vždy měla vrátit stejnou hodnotu. Je tedy opakovatelná.

Vstupem šumu je *n* dimenzionální skutečná souřadnice a výstupem je jedno reálné číslo v určitém intervalu. Běžně jsou používány dimenze první, druhá a třetí. Souřadnice o jedné dimenzi jsou většinou používány k animaci, dvě dimenze pro jednoduché procedurální texturování a tři dimenze pro texturování těles. Čtyři dimenze jsou také využívány, ale ne tak často, protože se většinou jedná o textury, které se mění v čase.

Pokud je na šum pohlíženo jako na signál, pak lze zjistit, že je frekvenčně omezen. Tedy jeho valná část energie je koncentrována v určité části kmitočtového spektra. Vysoké frekvence, představují detaily a nízké frekvence, které mají velmi malou část energie, celkový vzhled. Šum si lze také představit jako blok náhodných hodnot, které jsou vzájemně interpolovány.

2.2 Požadavky na šum pro účely procedurálního texturování

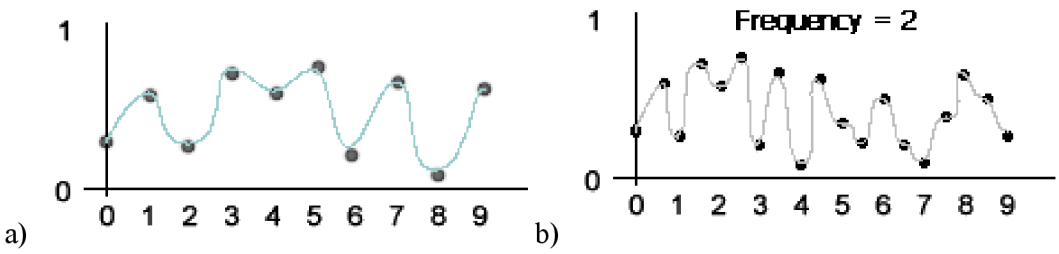
Různé šumové funkce mohou vracet různé výsledky, proto vše záleží jen na výběru funkce a metody a její interpolace. Ale jsou zde jisté zásady, které by měly být dodrženy.

Optimální šumová funkce by měla poskytovat bílý šum, který není ovšem frekvenčně ohraničený, takže poskytuje i ty nejjemnější detaily. Z toho následně vyplývá, že pro jeho výpočet by bylo potřeba velkého výpočetního výkonu. Proto se upustilo od těch nejjemnějších detailů a stanovily se následující požadavky [2,4].

- Pseudonáhodný a opakovatelný
  Mohlo by se zdát, že šum je čistě náhodný, ale není. Pokud by byl náhodný, pak při zavolání v jakémkoliv čase bude výsledek různý. Proto je použita pseudonáhodnost, která jen představuje náhodnost. Zavoláním funkce s parametrem x vrátí vždy stejnou hodnotu y.
- Spojitý
  Šum musí být spojitý, čímž je zaručena určitá maximální frekvence, která by mohla případně produkovat nežádoucí aliasing.
- Frekvenčně omezený
  Pokud je použit neomezený frekvenční rozsah (Bílý šum), pak lze modelovat i nejmenší detaily, ale na úkor vysoké výpočetní náročnosti a také vzniku aliasingu. Proto je vhodné jej frekvenčně omezit, čímž se zvýší také rychlost výpočtu.
- Staticky invariantní k otáčení (Isotropní)
  Šum by měl být isotropní. To znamená, že by měl být z každého směru stejný. Tedy pokud se otočí se souřadným systémem, pak nelze z této pozice otočení odvodit. Při mapování nezáleží na otočení textury.
- Staticky invariantní k posunutí (Stacionární)
  Funkce se nebude staticky měnit při posouvání. Při mapování nezáleží, kde se začne textura mapovat.
- Rozsah
  Funkce by měla vracet čísla v určitém rozsahu. Běžné hodnoty jsou od -1 do 1. V některých případech je použita absolutní hodnota, pak jsou výstupem hodnoty od 0 do 1.

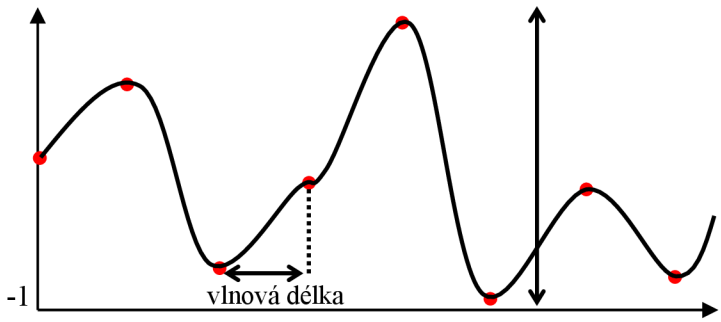
2.3 Frekvence a amplituda

Frekvence [9,10] je velice důležitým faktorem šumových funkcí, protože určuje stupeň zobrazených detailů. Některé šumy mají stále konstantní hodnotu frekvence, ale mnohem používanější je proměnná hodnota v určitém rozsahu, který je určen nejmenší a největší vzdáleností mezi lokálním minimem a maximem. Rozsah frekvence by měl být ale limitován a to obzvláště jeho horní mez. Frekvence je vyjádřena, jako počet šumových bodů, které se spočítají v jednom celočíselném intervalu, jako na obrázku 4., který ukazuje frekvenci o hodnotě 1 a hodnotě 2. Hodnota bývá zpravidla stanovena mocninou dvěmi, tedy 1, 2, 4, 8, atd.



Obrázek 4.: Ukázky frekvence: a) f = 1 b) f = 2 [9]

Amplituda [9,10] je rozmezí mezi maximální a minimální hodnotou, kterou šumová funkce může vrátit. Běžně je to rozsah od -1 do 1 nebo také od 0 do 1. Pro přesnost je uveden ještě jeden obrázek s kompletním vysvětlením frekvence a amplitudy, kde červené body jsou hodnoty, které vrátila šumová funkce. Vlnová délka je vzdálenost mezi dvěma hodnotami, podle které se určí hodnota frekvence dle vztahu 1/(vlnová délka).



Obrázek 5.: Frekvence a amplituda

2.4 Oktávy a persistence

Oktáva [9,10] je jedna vrstva šumu, v podstatě je to množina hodnot šumové funkce se stejnými parametry. Když se sečtou dvě stejné oktávy, zvýší se pouze celkové hodnoty šumu, což není zrovna žádoucí. Pravé uplatnění nacházejí oktávy ve sčítání šumových funkcí, popsané v další kapitole. Pro jednotlivé oktávy se mění parametry výpočtu, kterými jsou frekvence a persistence. Pokud se pokaždé

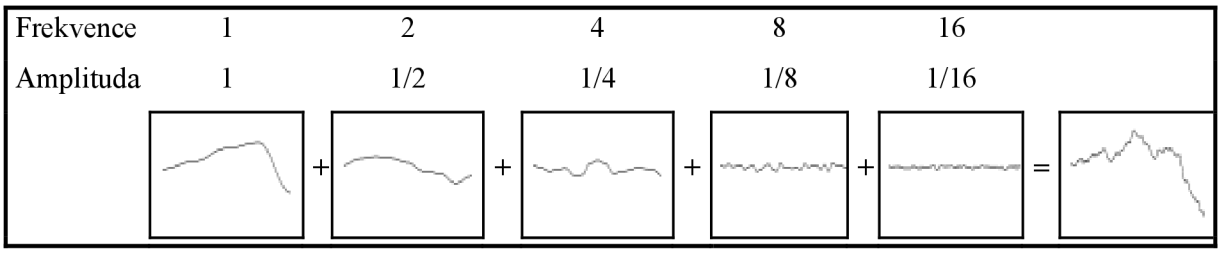
budou sčítat oktávy s vyšší frekvencí, pak lze dosáhnout větších detailů, ale je nutné stanovit hranici, kdy je už zbytečné sčítat další oktávy, protože tak jemné detaily se už nedokáží zobrazit. Některé algoritmy už sčítají jen tolik oktáv, kolik lze při aktuálním rozlišení zobrazit. Dále to také závisí na persistenci.

Persistence [9,10] je hodnota mezi 0 a 1, která specifikuje amplitudu pro každou frekvenci. Určuje prakticky váhu šumové funkce s danou frekvencí. Čím více oktáv je sčítáno, tím větší jsou frekvence a tím více se snižuje hodnota persistence. Všechno lze zapsat dvěma rovnicemi [10]:

$$frekvence = 2^i$$

$$amplituda = persistence^i$$

Kde i je i-tá oktáva, které se počítají od nuly (Obrázek 6.)



Obrázek 6.: Ukázka součtu několika oktáv. Persistence = 0,5 [10]

2.5 Skládání šumových funkcí

V předchozích dvou kapitolách byly objasněny pojmy, které se týkají skládání šumových funkcí a úzce s tímto souvisí. Viz obrázek 6., kde je vidět součet pěti oktáv šumové funkce.

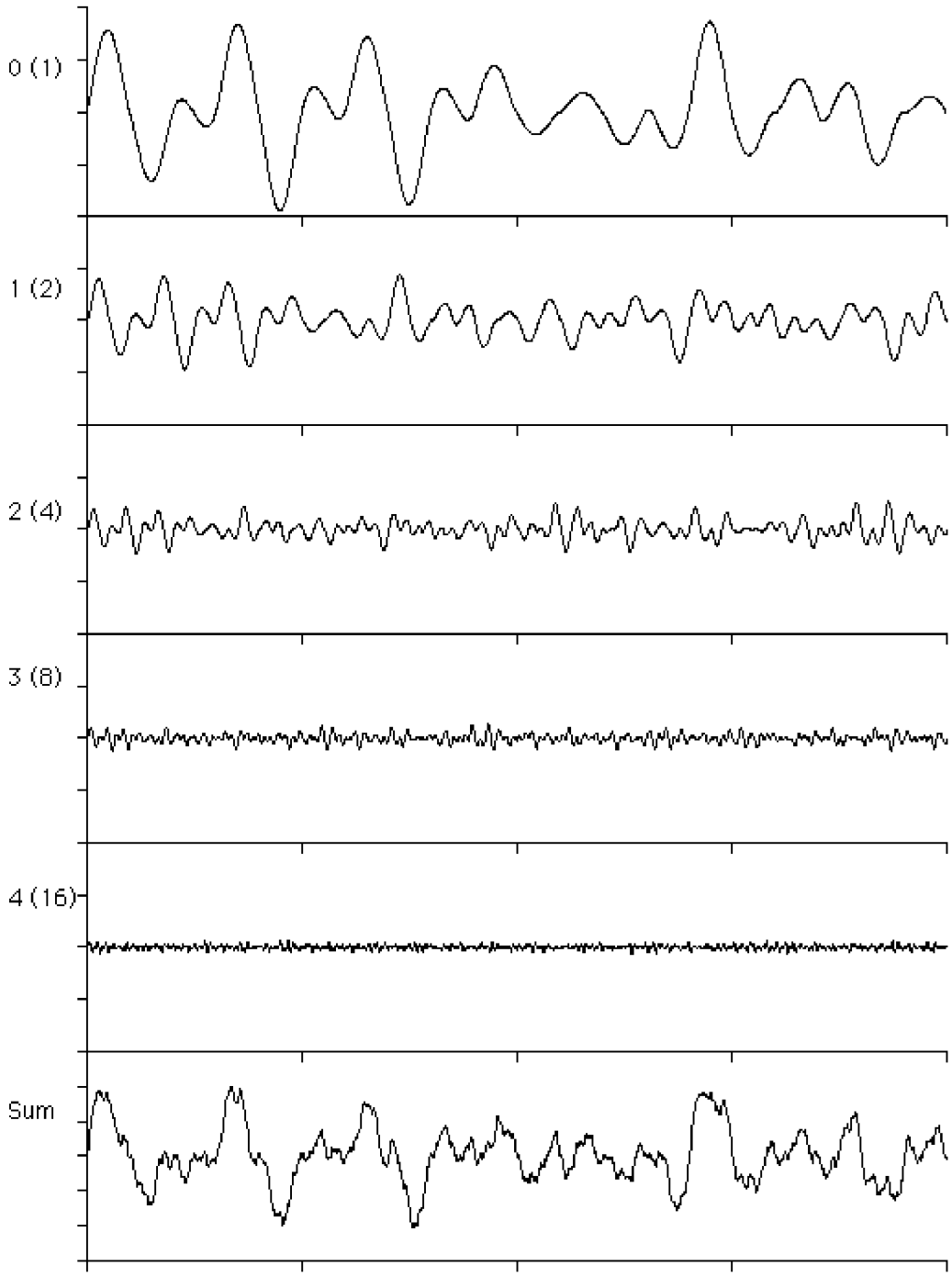
Pomocí skládání šumových funkcí [2,4,6,8] lze vytvářet realistické přírodní jevy. Takto lze vytvořit např. hory, oheň, vodu, dřevo, mramor, atd. Pouhá aplikace jednoho šumu má totiž omezené možnosti. Velice se hodí k aplikaci na objekty, kde je potřeba docílit jemné „špinavosti“ a další použití je při změně spektrální části světla, které dodá dojem „zašlého kovu“. Také ji lze použít jako normálovou mapu při vytváření bump textur. Pro širší použití je lépe využít skládání šumových funkcí.

Složení šumových funkcí je součet funkcí šumu, kde každá má změněnou amplitudu a frekvenci (viz kapitola Oktávy a persistence). Takto se často označuje jako Perlinova funkce, i když je to ve skutečnosti součet Perlinovy funkce nebo jiné. Obecně lze takto totiž složit libovolnou šumovou funkci. Občas je také označována jako Turbulence, což opět není správně. Bude vysvětleno později. Skládání šumových funkcí je pouhý součet jednotlivých oktáv a označuje se jako Fraktální suma [2]:

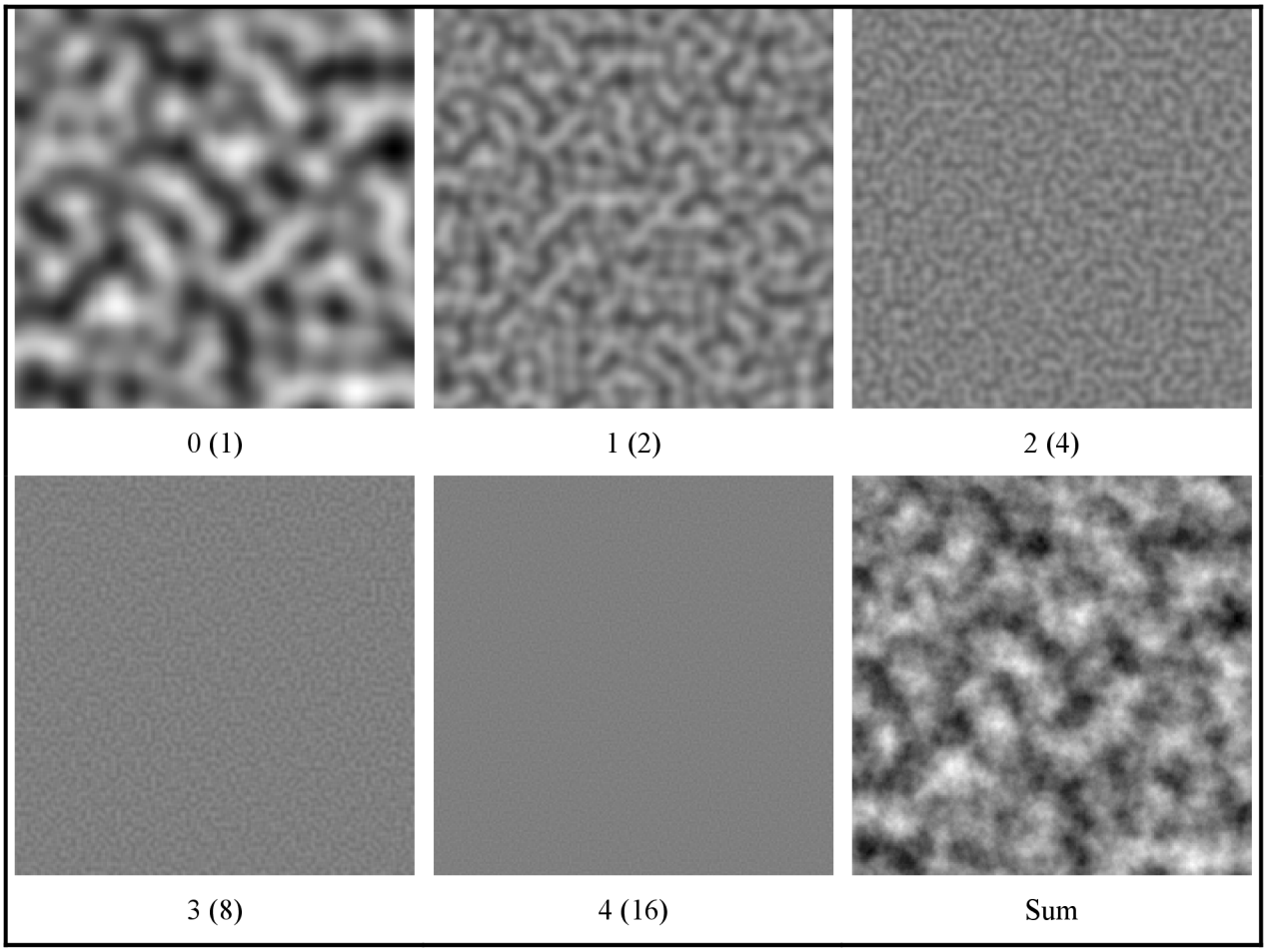
$$fsum(x, y, z) = \sum_{i=0}^{n-1} a_i \cdot noise((x, y, z) \cdot f_i)$$

Kde n je počet oktáv, a je amplituda vypočítaná z persistence. Přičemž počet oktáv určuje většinou rozlišení a jiné parametry. Je zbytečné počítat jemné detaily, když je už nelze zobrazit. Tímto

způsobem se modelují např. mraky, protože má jemné přechody a nabývá větších hodnot. Nejlepší bude ukázka na obrázcích [8].



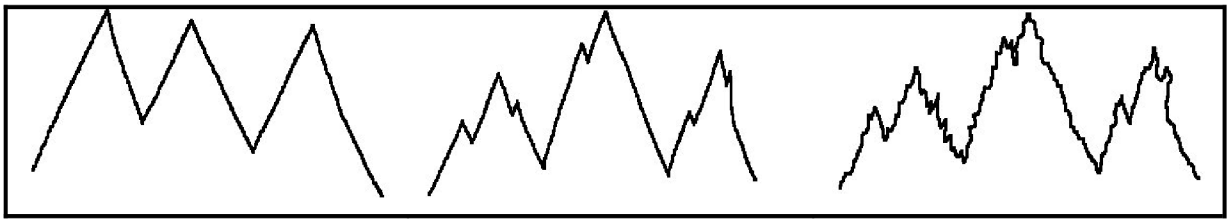
Obrázek 7.: Součet pěti oktáv v 1D. První číslo udává i-tou oktávu a číslo v závorce je hodnota frekvence



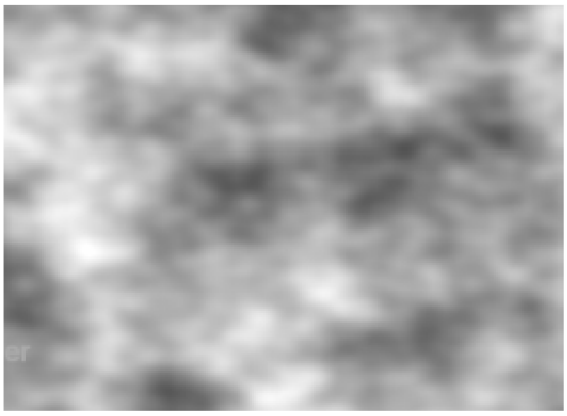
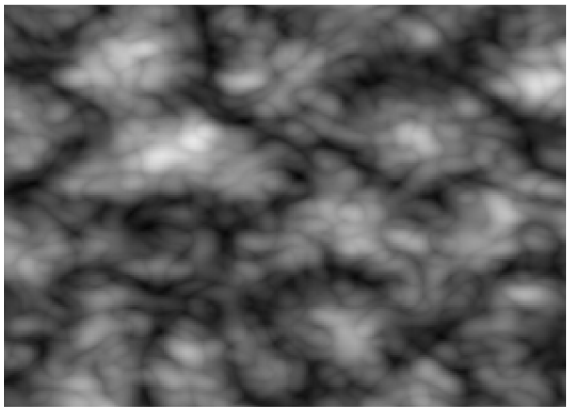
Obrázek 8.: Součet pěti oktáv ve 2D. Zde je již patrné, že výpočet další oktávy by bylo již zbytečné

Ken Perlin ovšem nepoužil jen fraktální sumu, ale provedl součet absolutní hodnoty šumových funkcí. Absolutní hodnota způsobí, že záporné hodnoty se „překlopí“ do kladných a tam kde byly hodnoty okolo nuly, pak vzniknou větší tmavé čáry nebo vrásky. V roce 1984 tuto funkci pojmenoval jako turbulenci [2,4,6,8]. Turbulenci si lze představit jako hory[11], pokud se na ně dívá z dálky, pak lze vidět hrubý obrys, při větším přiblížení lze rozlišit větší kameny a ještě blíže už drobné detaily. Turbulence je nejčastěji využívána k tvorbě textury dřeva a mramoru. Matematicky ji lze zapsat rovnicí [2]:

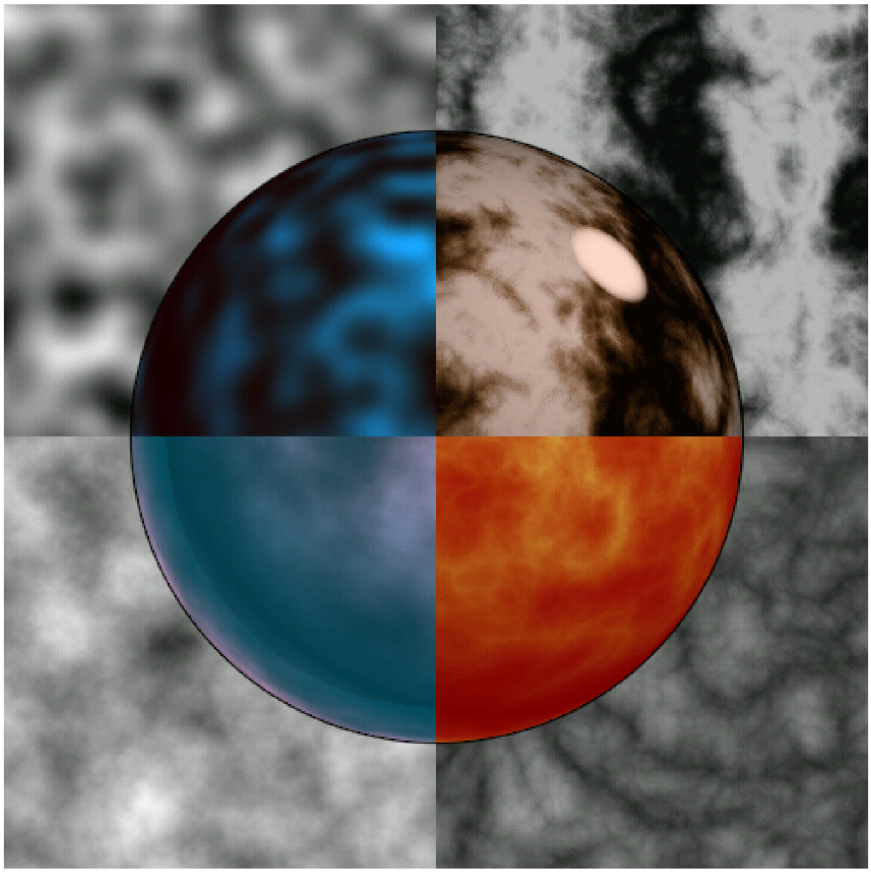
$$turbulence(x,y,z) = \sum_{i=0}^{n-1} a_i \cdot abs(noise((x,y,z) \cdot f_i))$$



Obrázek 9.: Turbulence přirovnaná k horám [11]

a)  b) 

Obrázek 10.: Součet tří oktáv Perlinova šumu. a) fraktální suma, b) turbulence [4]



Obrázek 11.: Příklad čtyř textur aplikované ve 2D a na kouli s barvami. Vlevo nahoře je klasický samotný šum, vlevo dole fraktální suma, vpravo dole turbulence a vpravo nahoře je ukázka mramoru. Jedná se o Perlinův šum [6]

Podle obrázků je patrný rozdíl mezi fraktální sumou a turbulencí. Obě metody sčítání šumových funkcí mají svůj daný vzhled a vlastnosti, které určují jejich použití nebo uplatnění.

3 Výpočet šumů

3.1 Náhodná čísla a generátory

Náhodná čísla jsou hojně využívanou součástí počítačových her, různých druhů simulací, ale například i pro lámání šifer a kódů. Tato práce se jimi bude zabývat pouze v rámci použití pro výpočet šumu použitého v procedurálním texturování [3].

Počítače dokáží výborně generovat pseudonáhodná čísla, ale ne skutečně náhodná čísla. Použití náhodných čísel v deterministickém systému (dokážeme určit stav) není problém, ale problémem je, jak určit stav, když použijeme náhodná čísla. Ve skutečnosti nejsou ale použita jen skutečně náhodná čísla, ale pseudonáhodná, která jen „předstírají“ skutečnou náhodnost.

3.1.1 Skutečně náhodná čísla

Na začátek je nutno dodat, že počítačem vygenerovaná skutečně náhodná čísla neexistují a o tom, jestli vůbec existují se vedou velké diskuse.

Definice posloupnosti náhodných čísel [16,17] říká, že pokud má být daný řetězec čísel náhodný, pak nelze najít kratší způsob jeho zápisu, než je právě řetězec samotný. Proto žádná posloupnost čísel, vzniklá dle nějakého algoritmu, který je určitě deterministický, nemůže být skutečně náhodná. Místo toho se vytváří posloupnost pseudonáhodných čísel, která je založena pouze na prvním náhodném čísle.

Skutečně náhodná čísla musí splňovat některá kriteria. Jsou nepředvídatelná, nedeterministická a neopakovatelná a jsou generována bez jakéhokoliv předchozího výsledku či vstupu. Za skutečně náhodná čísla se dá považovat házení mincí nebo kostkou. Většina náhodných čísel je generována fyzikálními generátory [18] na základě fyzikálních zákonů. Příkladem mohou být šumové generátory využívající vlastnosti polovodičového přechodu nebo kombinace radioaktivního zářiče a detektoru. Tyto generátory se už ale skoro vůbec nepoužívají, protože mají spoustu nevýhod. Generují neopakovatelná čísla, což ztěžuje algoritmizaci a testování a především jsou závislé na vnějších podmínkách.

3.1.2 Pseudonáhodná čísla

Jsou to taková čísla, která vypadají náhodně, ale jsou spočítány pomocí algoritmů, které jsou založeny na předchozím prvotním náhodném čísle. Jak již bylo řečeno, pseudonáhodná čísla jsou vytvářena pomocí generátorů pseudonáhodných čísel, které pracují na základě nějakého deterministického algoritmu. Tyto čísla pak plně dostačují pro libovolné účely.

3.1.3 Generátory pseudonáhodných čísel

V této podkapitole je pojednáno o několika generátorech pseudonáhodných čísel a jejich vlastnostech. Podrobnější informace [16].

3.1.3.1 Lineárně-kongruentní generátory

Jsou založené na výpočtu podle pravidla:

$$N_{k+1} = (l \cdot N_k + m) \bmod M$$

kde *l, m, M* jsou celočíselné parametry. Pokud $N_0 \in \langle 0, M)$, pak toto pravidlo generuje čísla z rozsahu $\langle 0, M-1 \rangle$. Výstupy těchto generátorů jsou náhodné, avšak periodické s periodou nejvýše *M*. Perioda je ale velmi závislá na volbě parametrů. Špatná volba znamená malou periodu a časté opakování stejných čísel. Lineárně-kongruentní generátor se používá s parametry l = 1103515245, m = 12345 a $M = 2^{32}$ například UNIXová funkce `rand()` generující 32-bitová celá čísla. Pro některé aplikace může být tento generátor nedostačující, protože perioda může být na dnešních počítačích vyčerpána za několik sekund. Aritmetika s dvojitou šířkou, pak může být neúnosně pomalá. Další nevýhodou se později ukázala skutečnost, že skupiny generovaných čísel vykazují geometrický vzorec, který odhaluje test na prostorové rozložení *(scatter plot test)*.

3.1.3.2 Kombinované lineárně-kongruentní generátory

Odstraňují problém s velikostí periody a skupiny generovaných čísel tímto generátorem již nevykazují geometrický vzorec. Tyto generátory nepoužívají ke generování pouze jedno předchozí číslo, ale výsledek je tvořen součtem dvou pomocných náhodných čísel. Perioda posloupnosti odpovídá hodnotě $M_1 \cdot M_2$.

$$\begin{aligned} x_i &= (l \cdot x_{i-1} + m_1) \bmod M_1 \\ y_i &= (l \cdot y_{i-1} + m) \bmod M_2 \\ N_i &= (x_i + y_i) \bmod \max(M_1, M_2) \end{aligned}$$

3.1.3.3 Zpožděné Fibonacciho generátory

Jsou založené na stejném principu jako kombinované lineárně-kongruentní generátory, mají však tu výhodu, že náhodné číslo závisí na některém jiném čísle ze stejné posloupnosti a nikoliv na číslech ze dvou pomocných posloupností. Zvyšuje se tím délka periody a snižuje míra korelací mezi prvky posloupnosti.

$$N_i = (N_{i-p} \oplus N_{i-q}) \bmod M$$

kde p a q jsou zpoždění nabývající nezáporných hodnot do velikosti posloupnosti a $\odot$ je aritmetická operace. Výhodou je, že volbou zpoždění měníme také periodu generovaných čísel.

3.1.4 Počáteční hodnota generátorů

Uvedené generátory produkují jeden proud náhodných čísel, které nejsou nikterak přesně určené. Problémem může být také jiné chování v jiných operačních systémech nebo platformách. Řešení nabízí počáteční nastavení vstupní hodnoty pomocí funkce srand(), která je založena na funkci rand(), ale je volána s jedním parametrem, kterému se říká semínko nebo zrníčko (seed) [3]. Toto zrníčko je počátečním činitelem (vstupní hodnotou), podle které se bude generovat proud náhodných čísel. Funkce rand() totiž generuje čísla na základě předchozí hodnoty a semínko plní právě funkci prvního náhodného čísla. Funkce pak jednoduše nabízí determinističnost a zároveň „jakousi“ náhodnost. Protože, pokud zavoláme funkci třeba ze zrníčkem 225, pak při zavolání této funkce se stejným zrníčkem v libovolném dalším čase získáme opět stejný proud hodnot.

3.2 Interpolační metody

Je dána mřížka s pseudonáhodnými hodnotami a číslo, které „spadne“ do některé buňky v mřížce. Toto číslo není z množiny celých čísel, proto ve 2D je potřeba interpolace s okolními čtyřmi body v uzlech mřížky. Záleží jen na výběru interpolační metody a podle toho pak bude vypadat výsledný vzhled, ale také rychlost aplikace. Je zde mnoho interpolačních metod, ze kterých lze vybírat a liší se hlavně poměrem kvalita a výkon.

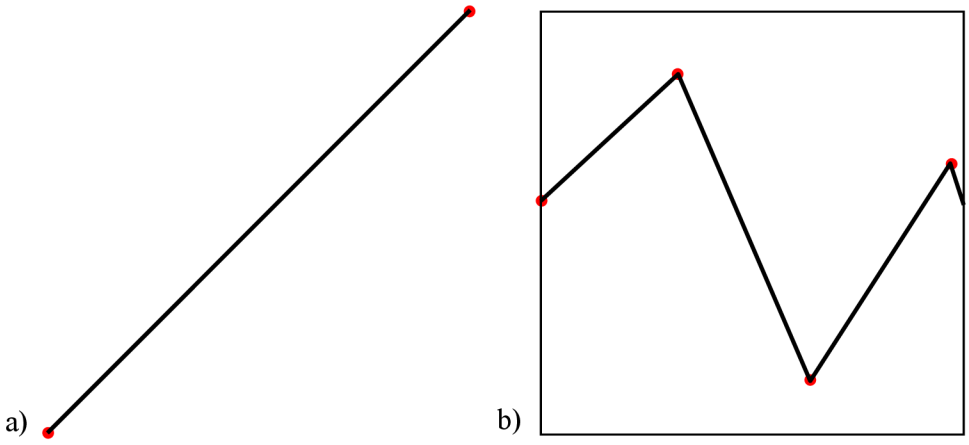
3.2.1 Lineární interpolace

Je to ta nejjednodušší interpolační metoda [4,10]. Má konstantní průběh mezi dvěma hodnotami, které jsou interpolovány. Interpolace je pak ovlivňována váhou jednotlivých bodů. Matematicky lze interpolaci zapsat:

$$linear = A*(1-t)+B*t$$

kde A a B jsou dva interpolované body a t je váha, která je v intervalu <0,1>. Ve většině případů je t reálná část souřadnice bodu, pro který hledáme hodnotu.

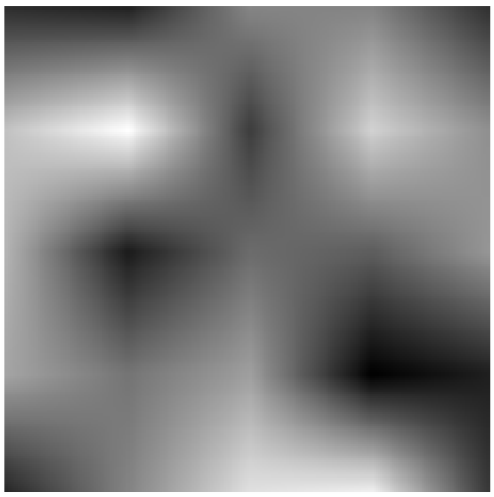
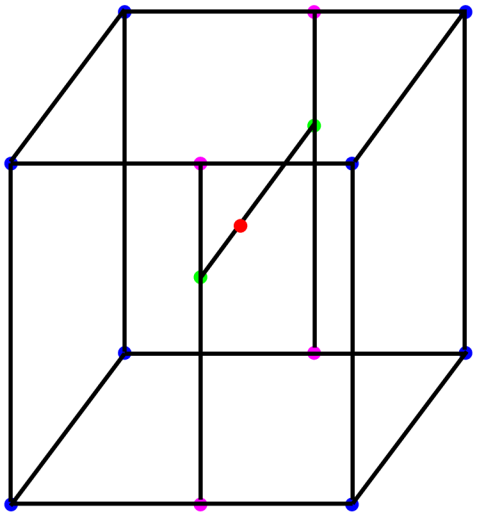
Lineární interpolace je jednoduchá na výpočet, ale vzhledem k ostatním metodám produkuje nejhorší výsledky. Faktem ale také je, že je již hardwarově podporována.



Obrázek 12.: Lineární interpolace. a) od 0 do 1, b) mezi čtyřmi hodnotami

3.2.2 Bilineární a trilineární interpolace

Je 2D, resp. 3D verzí lineární interpolace. Pokud je aplikována ve 3D, pak se provede 7 lineárních interpolací pro 8 okolních bodů.

a)  b) 

Obrázek 13.: a) 2D bilineární interpolace [4], b) 3D trilineární interpolace. Nejdříve se spočítají fialové body pomocí čtyř lineárních interpolací ve směru osy x, poté dva zelené ve směru osy y a poslední výsledný červený bod lineární interpolací ve směru osy z.

3.2.3 Spline interpolace

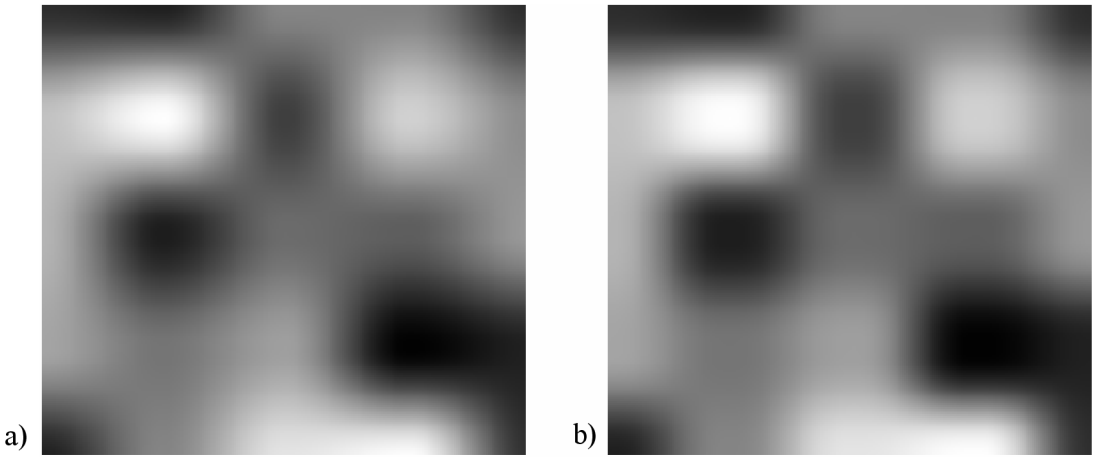
Je možné ji též nazvat S-křivka [4], protože má podobný tvar. Na obou koncích se zaobluje.
Rovnice pro spline interpolaci se dá zapsat následovně:

$$spline = A*\left(1-\left(3t^2+2t^3\right)\right)+B*\left(3t^2+2t^3\right)$$

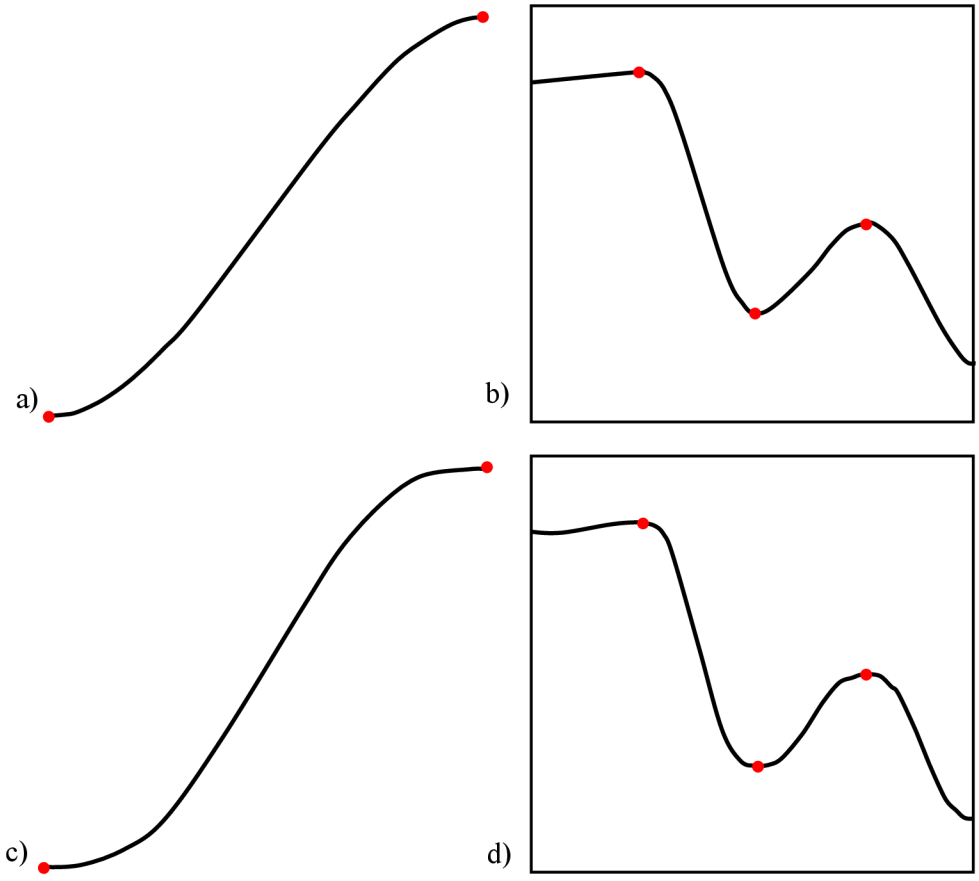
Tak jako v předchozí rovnici je t opět v intervalu <0,1>. Oproti lineární interpolaci zde neexistuje „špičatost“ v hlavních bodech. Přesto to není tak přesně, protože lze najít i případy, kdy toto nastane. Pokud bude t rovno 0 nebo 1, nebo-li interpolovaný bod je roven bodu v mřížce, pak druhá derivace v tomto bodě není nulová. Toto lze vyřešit modifikací rovnice na rovnici:

$$spline = A*\left(1-\left(6t^5-15t^4+30t^3\right)\right)+B*\left(6t^5-15t^4+30t^3\right)$$

Spline interpolace podává mnohem lepší výsledky, ale také je výpočetně náročnější. Modifikovaná spline interpolace je ve výsledku skoro nerozpoznatelná od nemodifikované. Pro jednoduchost stačí tedy použití nemodifikované verze.



Obrázek 14.: a) 2D spline interpolace, b) modifikovaná 2D spline interpolace [4]



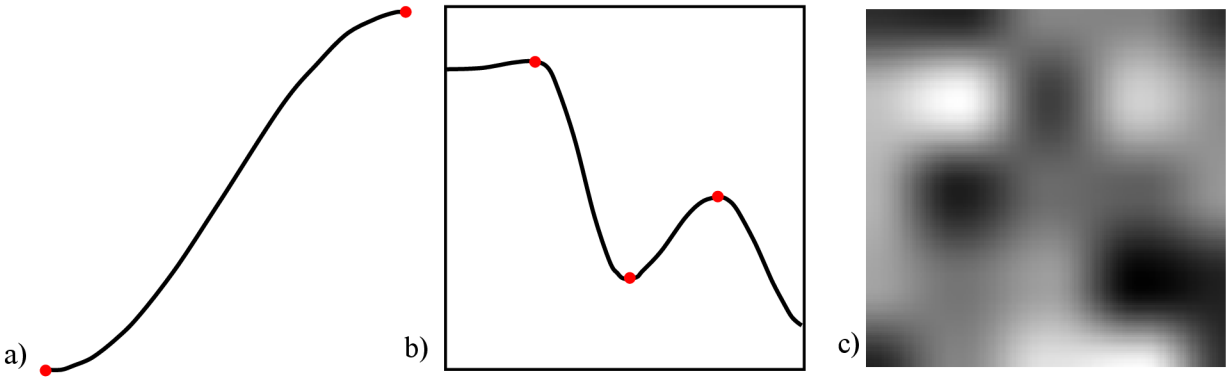
Obrázek 15.: a) a b) 1D Spline interpolace, c) a d) 1D modifikovaná spline interpolace

3.2.4 Kosinová interpolace

Je alternativou ke Spline interpolaci, ale její přechody mezi jednotlivými odstíny nejsou tak jemné. Kosinová interpolace [4,10] je vyjádřena rovnicí [4]:

$$\cos ine = A*(1-((1-\cos(\pi*t))/2))+B*((1-\cos(\pi*t))/2)$$

Ve většině případů je ale Spline interpolace využívána více než Kosinová interpolace, protože výpočetně je pro procesor méně náročná. Ovšem pokud je program aplikován na grafický procesor, tak se výpočetní rozdíly stírají a pak už jen záleží na programátorovi, který z algoritmů použije.



Obrázek 16.: Kosinová interpolace v 1D a 2D

Jednotlivé metody interpolace, které byly dosud zmíněny, používají pro interpolaci 2 resp. 3 vstupní hodnoty pro 2D resp. 3D. U těchto metod se extrémní hodnoty rovnají hodnotám v mřížce náhodných hodnot a to má za následek, že výsledný šum má také mřížkovou strukturu („kostičky“). Proto existuje více interpolačních funkcí, které mají větší počet vstupních hodnot a tím dosahují lepších výsledků.

3.2.5 Kubická interpolace

Je nejjednodušší z interpolací, které využívají více hodnot [4,10], konkrétně čtyři. Rovnice výpočtu [4]:

$$cubic = A*t^3 + B*t^2 + C*t + D$$

kde: A = d – c – a + b

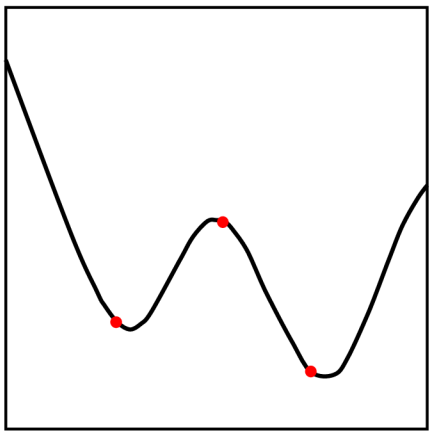
B = a – b – A

C = c – a

D = b

Zde *b* je hlavní bod před *t* a *a* je hlavní bod před *b*. Hodnotový bod *c* přímo po *t* a *d* je bod následující za *c*. Pokud budou zapsána v posloupnosti písmen, pak v následujícím pořadí: *a b t c d*.

Výsledkem je, že lokální maxima a minima již nejsou umístěna v jednotlivých uzlech mřížky, ale jsou posunuta (Obrázek 17.).



Obrázek 17.: 1D kubická interpolace pro tři hodnoty

Mezi další interpolační funkce patří **Catmull-Rom spline interpolace**, která je založena na spline interpolaci. **Hermite interpolace** má zase více parametrů, které dovolují ovládat výsledek interpolace a tím i kvalitu.

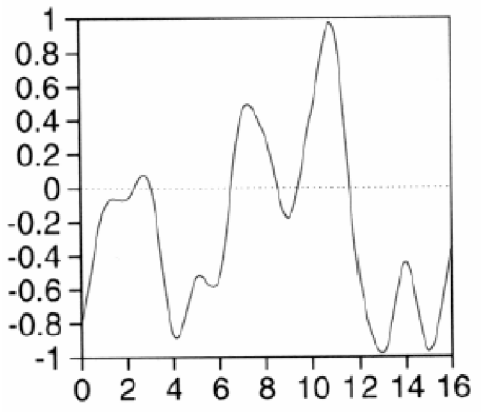
Přestože tyto metody produkují mnohem lepší výsledky, nejsou moc využívány. Je to hlavně pro jejich vysokou výpočetní náročnost. Dalším a možná i důležitějším faktorem je použití několika násobně vyššího počtu vstupních hodnot. Např. ve 3D, kde je použito 64 hodnot místo 8 u lineární interpolace.

3.3 Druhy šumů

Hlavní a nedílnou součástí jsou samozřejmě šumové funkce. Opět záleží jen na výběru a většinou platí stejné pravidlo jako u interpolací, že za kvalitu se platí výkonem. Nejčastěji používaným a zmiňovaným šumem je právě Perlinův šum. Jeho výsledky jsou velice dobré a přitom není nikterak výpočetně náročný. Přehled šumových funkcí a obrázků v literatuře [4].

3.3.1 Lattice Value Noise (Mřížkový šum)

Je nejjednodušším šumem. Základem je mřížka vygenerovaných pseudonáhodných čísel. Výsledkem jsou pak vyhlazené nebo nevyhlazené interpolované hodnoty v mřížce. Princip je takový, že vstupem je bod, pro něhož hledáme hodnotu. Ten je typu float. Jelikož ve většině případů to není uzel mřížky, musí se interpolovat s ostatními čtyřmi okolními uzly a záleží jen na výběru interpolační funkce. Samotná funkce neprovádí žádné korekce ani výpočty a vše spočívá na interpolační funkci.

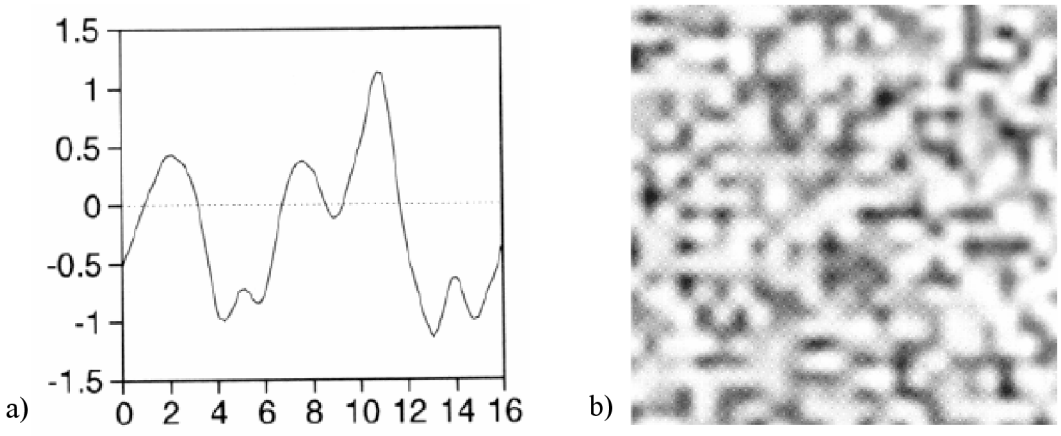
a) 

b) 

Obrázek 18.: a) průběh 1D hodnot Value Noise, b) 2D Value Noise [5]

3.3.2 Lattice Convolution Noise

Je stejný jako Lattice Value Noise, ale používá složitější Catmull-Rom interpolaci. Snaží se odstranit různé nežádoucí vlastnosti Lattice Value Noise, ale výsledek není o moc lepší. Za cenu více vstupních hodnot se tento algoritmus nevyplatí.

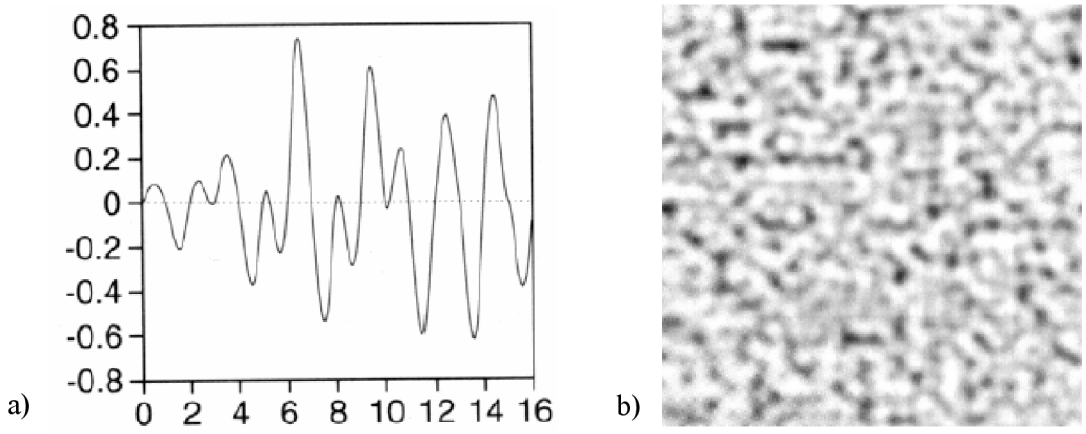


Obrázek 19.: Lattice Convolution Noise a) 1D, b) 2D. Nyní lze snadno porovnat s Value Noise [5].

Mřížkové šumy poskytují docela přesvědčivé a dobré výsledky, ale stále je v nich poznat interpolace podle mřížky. Proto byl vytvořen jiný algoritmus, který používá místo hodnot vektory.

3.3.3 Gradient noise

Místo hodnot jsou v mřížce uloženy jednotkové náhodné vektory. Tyto vektory pak ovlivňují výslednou hodnotu hledaného bodu podle vzdálenosti od mřížkové hodnoty. Hodnota uzlu se získá skalárním součinem vektoru a vektoru vzdálenosti hledaného bodu od uzlu. Většinou je pak využita trilineární interpolace. Na tomto šumu je založen Perlinův šum.



Obrázek 20.: a) průběh 1D Gradient Noise, kde je vidět, že hodnoty v uzlech prochází nulou, b) 2D Gradient Noise [5]

3.3.4 Perlin noise

Nejoblíbenější šumová funkce. Podstatou Perlinova šumu [6] je generování spojitého šumu, který je ovšem vypočítáván z diskrétní mřížky gradientních vektorů. Perlinův šum, tak jako většina šumů,

může být definován v libovolné dimenzi. Parametrem funkce jsou souřadnice pixelu (texelu), které jsou z reálného oboru hodnot. Princip Perlinova šumu [2,4,6,11].

Je tedy dán bod *S*, který má reálné souřadnice $[x, y, z]$. Prvním krokem je určení buňky, do které bod *S* se svými souřadnicemi „zapadne“. Souřadnice buňky se určí tak, že se souřadnice bodu *S* zaokrouhlí na nejbližší dolní celočíselnou hodnotu. Pokud budou souřadnice ve 2D, pak má čtyři okolní body, v případě 3D jich je osm. Obecně pro *n* dimenzí platí 2^n okolních bodů. Zaokrouhlením se získá levý dolní roh a souřadnice zbylých uzlů jsou vždy zvýšeny o jedničku v každém směru.

Než uvedu způsob výpočtu hodnoty, je lépe vysvětlit vygenerování mřížky (ve skutečnosti jen jednorozměrné pole) gradientních vektorů. Je důležité, aby si sousední vektory nebyly příliš podobné, protože ve výsledku by to mohlo znamenat, že v šumu by mohly být znatelné vzory. Proto Perlin navrhnul velikost pole 256 a k tomu permutační pole, které zajistí náhodný přístup do pole gradientů. Velikost 256 by měla být dostačující, aby se odstranila případná opakovatelnost.

Algoritmus generuje pro každý index pole tři náhodné hodnoty z intervalu $\langle -1, 1 \rangle$. Ty odpovídají náhodnému vzorkování jednotkové krychle, ale v jednotkové krychli se nachází i čísla s délkou větší než jedna. Ty nejsou použity a pro daný index se musí generovat nová čísla. Hodnoty, které jsou uvnitř jednotkové koule, se znormalizují (promítnou se na jednotkové kouli) a uloží do pole na daný index. Algoritmus se opakuje, dokud nevygeneruje 256 vektorů, které rovnoměrně pokrývají jednotkovou kouli.

Jak již bylo zmíněno, do pole gradientů se nepřistupuje přímo, ale pomocí permutačního pole *P*. Tabulka je velikosti 256 a obsahuje hodnoty od 1 do 256. Pole *P* se předpočítá pomocí jednoduchého algoritmu. Jedná se o prosté „zamíchání“. Nejdříve se pole naplní hodnotami od 0 do 255, tedy index položky bude vlastní položkou. Poté pro *i* od *0* do *255* vygeneruje pseudonáhodné číslo *k* z intervalu $\langle 0, 255 \rangle$ a provede záměnu *P*[*i*] za *P*[*k*].

Nyní lze přiřadit vektor pro libovolný bod z okolních bodů vstupní souřadnice. Jenže ty jsou *n* rozměrné a proto je použita permutační tabulka na tzv. přeložení (*fold*). Hodnota se získá vztahem:

$$fold(i, j, k) = \left(\left(i + P\left[\left(j + P\left[k\right]\right) \bmod n\right]\right) \bmod n\right).$$

Tento vztah však používá operaci modulo. Tato operace zbytečně prodlužuje dobu výpočtu, proto ji Ken Perlin odstranil a provedl „pouhý“ součet dle vzorce

$$fold(i, j, k) = \left(i + P\left[j + P\left[k\right]\right]\right).$$

Tímto způsobem lze ale snadno překročit hodnotu 255 a tím „sáhnout“ po prvku mimo pole. Jelikož je ale pole o velikosti 256, tedy mocnina dvou, lze bez problémů použít dolních 8 bitů a výsledek bude stejný za kratší dobu.

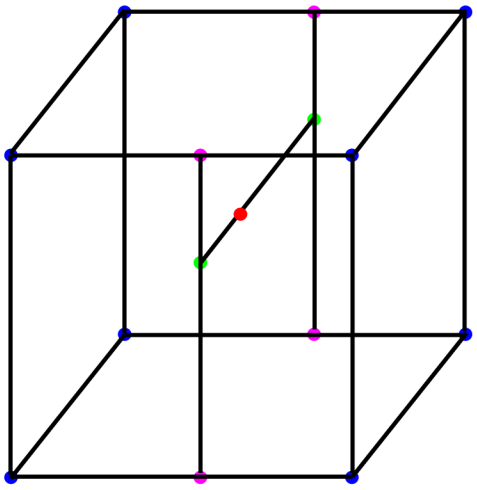
Gradient se získá voláním $G = [fold(i, j, k)]$. Tím je vybrán libovolný náhodný vektor a je splněna podmínka, že se nebude příliš opakovat a jeho rozložení v podstatě vzorkuje jednotkovou kouli.

Posledním krokem je samotný výpočet. Výsledek ovlivňuje vzdálenost vstupní souřadnice od uzlu. Udává váhu daného uzlu ke všem ostatním uzlům a tato váha je vyjádřena vektorem vzdálenosti od uzlu. Je-li vstupní bod $S(u, v, w)$ a uzel mřížky Q, pak se vektor spočítá jako $(S - Q)$. Hodnoty prvků tohoto vektoru jsou v intervalu $\langle -1, 1 \rangle$.

Perlinův šum používá Hermite interpolaci. Nejdříve se spočítá směšovací kubická funkce

$$f(t) = 3t^2 - 2t^3$$

v každé ose, čímž se získá váhový parametr pro následujících sedm interpolací. Konkrétně čtyři v x-ové ose, dvě v y-ové a poslední v z-ové. Jedná se samozřejmě o 3D. Postupnou interpolací je získána výsledná šumová hodnota.



Obrázek 21.: Ukázka trilineární interpolace

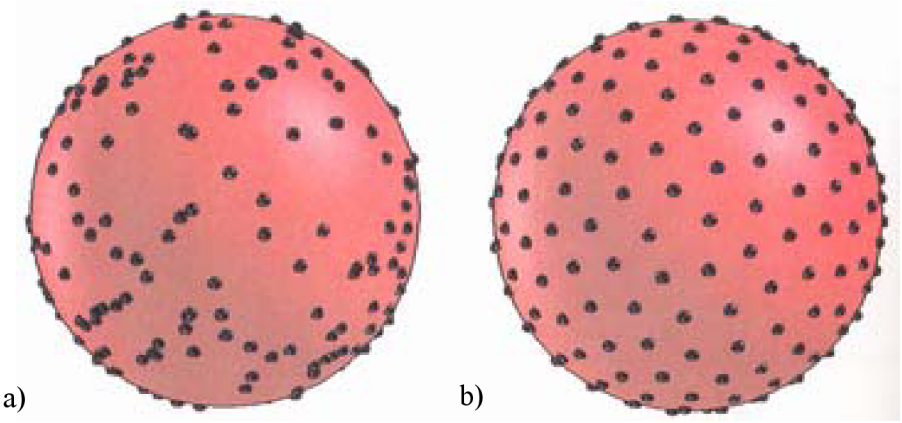
3.3.5 Improved Perlin noise

Perlinův šum měl ovšem dvě chyby, proto Perlin navrhl Improved Perlin noise [1,7]. První chyba byla ve výběru interpolační funkce, která v každé dimenzi způsobí, že druhá derivace obsahuje nenulové hodnoty. Pokud by byl šum použit pro vytváření bump textury, kde je potřeba derivace, tak může způsobit artefakty. Toto lze opět napravit jinou interpolační rovnicí:

$$y = 6t^5 - 15t^4 + 30t^3$$

Tato rovnice již artefakty odstraní, ale také je mnohem více výpočetně náročnější. Tato interpolace se tedy vůbec nevyplatí, nehledě na to, že ony artefakty jsou téměř nerozpoznatelné.

Dalším problémem a nedostatkem je výběr gradientních vektorů. Přece jen není zajištěn rovnoměrný výběr vektorů a tato nerovnoměrnost produkuje nechtěné vyšší frekvence ve výsledné šumové funkci [1].



a) b)

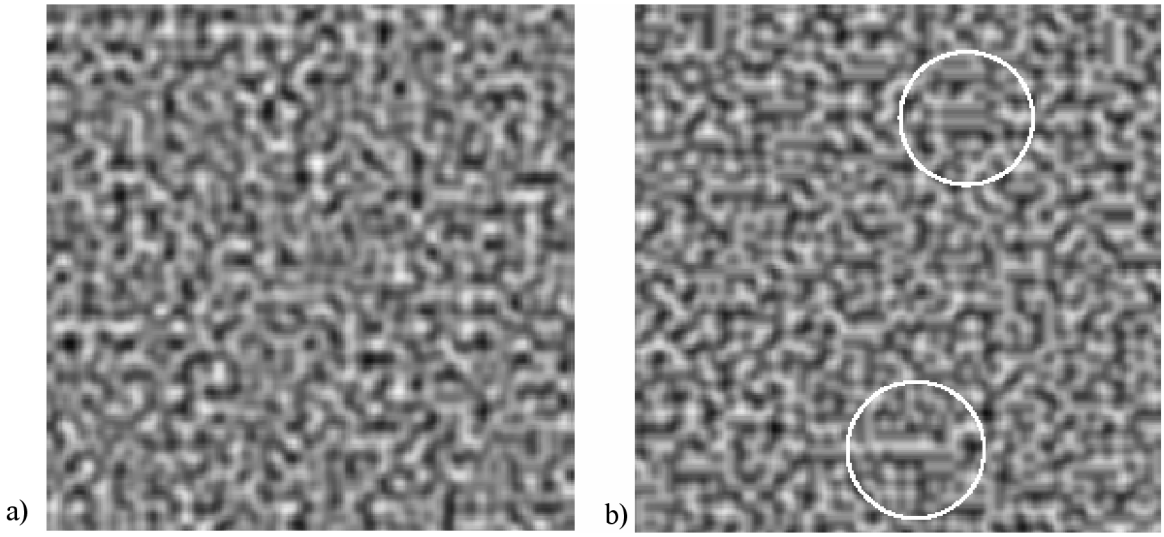
Obrázek 22.: a) nepravidelné rozložení, b) pravidelné rozložení

Řešení nabízí Poissonovo rozložení, které vrátí opravdu pravidelné rozložení. Jenže i zde záleží na použití šumu. Ve většině případů jsou tyto změny nepostřehnutelné.

Pro zmenšení výpočetní náročnosti je tabulka náhodných vektorů zaměněna za menší fixní tabulku vektorů.

$$(1,1,0),(-1,1,0),(1,-1,0),(-1,-1,0),$$
$$(1,0,1),(-1,0,1),(1,0,-1),(-1,0,-1),$$
$$(0,1,1),(0,-1,1),(0,1,-1),(0,-1,-1),$$

Tyto vektory reprezentují diagonální vektory v rovinách krychle, skládající se z okolních osmi bodů. Pokud jsou použity tyto pevně stanovené vektory, je celková výpočetní náročnost menší, protože není potřeba žádné násobení. Ale na druhou stranu je nutno podotknout, že při použití pevně stanovených vektorů, může dojít k jakési pravidelnosti. Toto už záleží jen a pouze na uživateli, jakou šumovou funkci si vybere. Dalším faktorem je, který procesor bude program počítat, protože grafické jádro zpracovává vektory (tři složky u 3D) jako jednu instrukci. Pak se zřejmě vyplatí starší metoda.

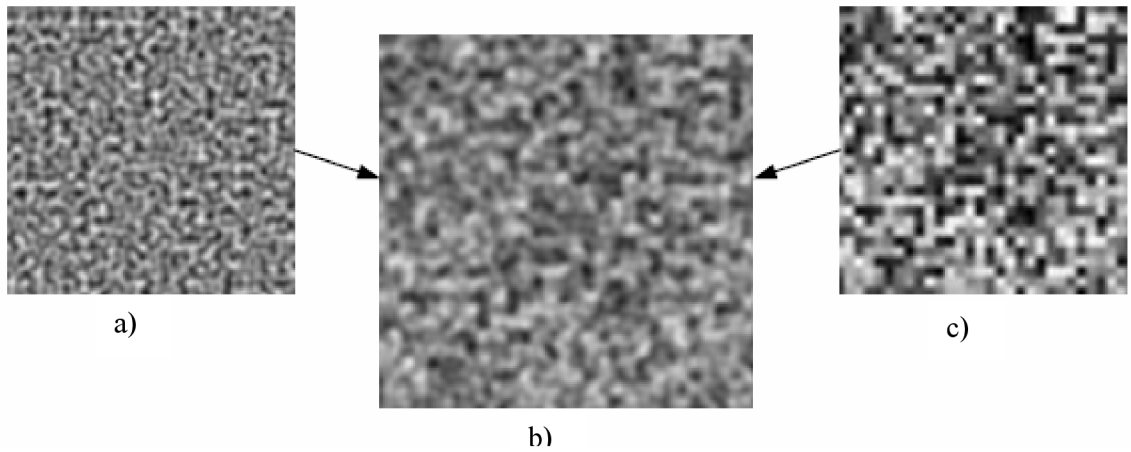


a) b)

Obrázek 23.: Perlinův šum ve vysokých frekvencích. b) kruhy označují pravidelnost u Improved Perlin noise. U náhodného výběru gradientů pravidelnost nelze najít (a) [4].

3.3.6 Value-Gradient Noise

Řeší problém toho, že při použití gradientního šumu vznikají artefakty v uzlech mřížky, přesněji nulové hodnoty. Proto se spojil gradientní šum a mřížkový šum (Lattice Value Noise). Jednotlivé výsledky šumů se vzájemně sčítají a tím se v uzlech mřížky doplní hodnoty z mřížkového šumu.



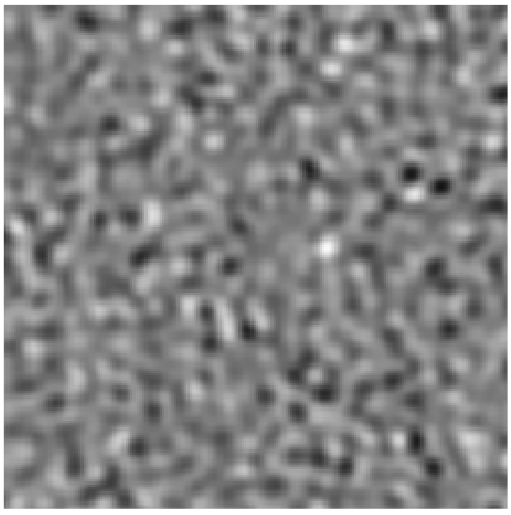
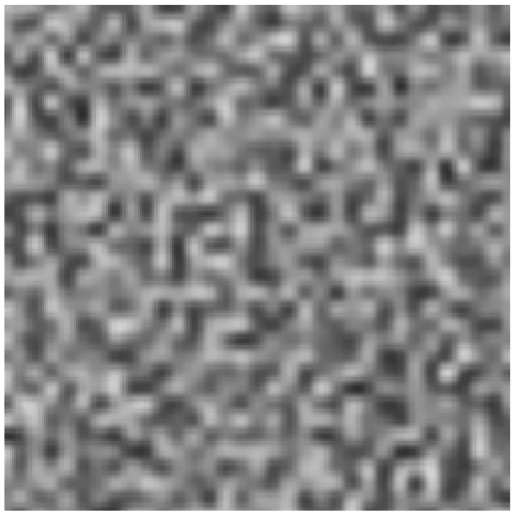
Obrázek 24.: a) Perlinův šum vypočtený na základě gradientního šumu, c) Value noise vypočtený Spline interpolací, b) kombinace těchto dvou šumů [4].

Ve výsledku je jejich součet velice dobrý, ale za kvalitu je opět zaplaceno cenou výpočtu. Nicméně, některé výpočty, týkající se Value noise mohou být přesunuty do výpočtu Perlinova šumu a tím lze zmenšit výpočetní náročnost.

Všechny výše popsané šumy používají jistý druh mřížkových hodnot. Jejich výsledky jsou velice dobré, ale přesto je někdy znát použití pravidelné tabulky hodnot. Řešení lze nalézt u hodnot, které jsou nepravidelně rozmístěny.

3.3.7 Sparse Convolution Noise

Používá tři hodnoty pro výpočet pseudonáhodného pozice mezi uzly mřížky (v případě 3D). Interpolační metoda je stejná jako u Lattice Convolution Noise, ale výsledek tohoto šumu je mnohem lepší. To je dáno pseudonáhodnou pozicí.

a)  b) 

Obrázek 25.: Porovnání a) Sparse Convolution Noise a b) Lattice Convolution Noise [4].

Výsledek je velmi vysoké kvality a staticky izotropní, ale opět za velmi vysokou cenu. Pět uzlových bodů musí být vypočítáno pro jednu dimenzi. Tedy pro tři dimenze je to 125 bodů.

3.3.8 Spot nosie (Bodový šum)

Spot noise [4,13] je úplně odlišný od šumů, které byly doposud zmíněny. Jeho rovnice má tvar:

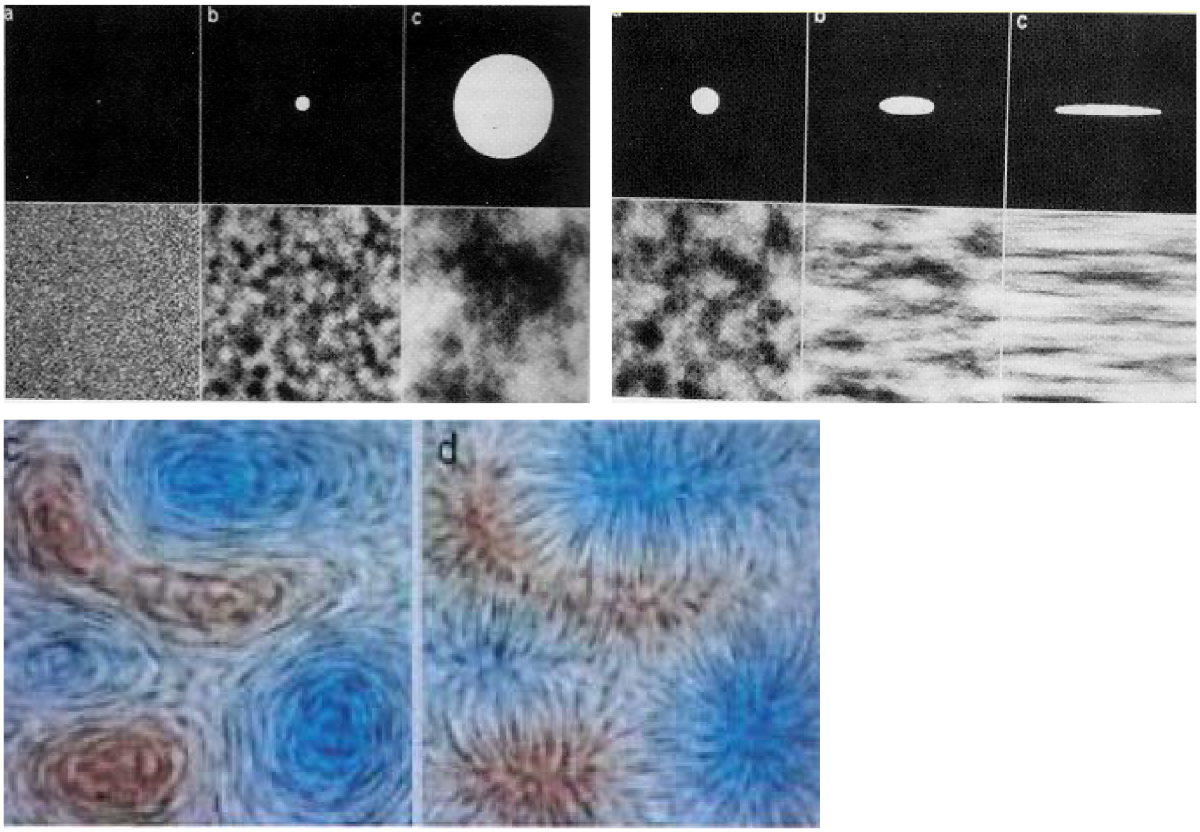
$$f(x) = \sum_i a_i h(x - x_i)$$

kde a_i je pseudonáhodná hodnota určující stupeň šedi, tak jako hodnoty v mřížkových šumech

x_i je pseudonáhodná hodnota pozice.

Tato rovnice byla navržena panem van Wijk v roce 1991. Je frekvenčně založen na bílém šumu a realizuje se pomocí bodů, které mohou mít libovolný tvar podle svého účelu. Funkci si lze představit tak, jako by se na povrch náhodně rozmísťovaly náhodné hodnoty v určitém specifickém tvaru. Výsledek se dostane sečtením jejich hodnot. Samozřejmě je možnost vypočítat šum i ve 3D, kde by se daly body nazývat kapkami.

Tak jak jsou ostatní šumové funkce založeny na druhu interpolační metody a na frekvenci, tak u bodového šumu záleží jen na tvaru a velikosti bodu. Velikost určuje hrubost resp. Jemnost, tvar určí vzhled a směr. Právě použitím elipsoidních tvarů lze vytvářet různé simulace magnetických vln. Přesto, že je Spot noise ze strany výpočtu tou nejnáročnější funkcí, najde právoplatné uplatnění právě při simulaci různých toků (magnetické, elektrické, silové, …).



Obrázek 26.: Různé tvary Spot noise [13]

3.3.9 Worley's Cellular Texture Basis Function (Worleyova funkce založena na buňkách)

Worley představil funkci, která využívá rozložení kontrolních bodů nezávislých na mřížce hodnot. Výsledná hodnota je dána vzdáleností nejbližšího kontrolního bodu. Prvním krokem je nalezení buňky, do které spadá vstupní souřadnice. Souřadnice této buňky jsou pak použity pro transformační funkci, která vrátí jedno číslo. Toto číslo je následně použito jako vstupní semínko generátoru náhodných čísel, který vygeneruje několik náhodných čísel podle Poissonovy náhodné distribuční funkce. Nakonec se spočítá vzdálenost k nejbližšímu vygenerovanému bodu. Ovšem nestačí spočítat hodnoty jen pro jednu buňku, do které padla vstupní souřadnice, ale také pro sousední buňky. Některá z nich může vygenerovat bližší hodnotu ke vstupní souřadnici.

Přestože je funkce pomalejší něž např. Perlinův šum a její výsledek není taky zrovna moc přesvědčivý, může najít uplatnění tam, kde je vyžadována jiná alternativa.

3.3.10 Fractional Brownian Motion (Fraktální Brownův pohyb)

Je jedním z prvních algoritmů pro výpočet šumu používaný v počítačové grafice. Byl popsán jako aproximace Brownova pohybu (uspořádaný pohyb molekul). Metoda může být popsána jako náhodné středové posunutí. U mřížkových šumů se detaily určovaly pomocí oktáv, které byly sčítány (turbulence), u FBM jsou detaily řízeny počtem vzorkovacích bodů. Amplituda je řízena vstupní hodnotou funkce.

U této metody jsou ale dva hlavní problémy. Prvním problémem je, že funkce není skutečnou šumovou funkcí, ale vrací pouze množinu hodnot. Tedy klasická šumová funkce vrátí pro libovolný bod interpolovanou hodnotu, kdežto tato funkce množinu hodnot. Pokud by bylo potřeba stejného výsledku, pak je nutné ještě tuto množinu hodnot interpolovat. Druhým problémem je, že celá množina vzorkovacích bodů v určitém intervalu musí být vypočítána ještě předtím, než je potřeba získat hodnotu z některého bodu. Buď ji předpočítat a uložit nebo počítat za běhu programu, ale pak to bude velice pomalé.

4 Návrh rozhraní knihovny

Cílem této kapitoly je návrh rozhraní tak, aby byly efektivně implementovány šumové funkce. Implementovat všechny druhy šumů by bylo asi zbytečné a možná i náročné, proto je důležité vybrat nejdříve některé metody podle jejich vlastností a náročnosti. Jistou variabilitu nabízí použití fraktální sumy a turbulence.

4.1 Výběr šumových metod

V knihovně budou implementovány následující šumové metody.

- Lattice Value Noise (mřížkový šum)
  Jelikož je to nejjednodušší metoda a je nejméně výpočetně náročná nesmí v knihovně chybět. Bude-li chtít uživatel spočítat šum co nejrychleji, pak určitě využije tuto metodu.
  Samozřejmostí je použití nejjednodušší lineární interpolační metody. Bude také implementována pro případ srovnání s ostatními metodami. A to jak kvalitou tak výpočetní náročností
- Perlin noise (Perlinův šum)
  Nesmí v knihovně chybět, protože je to ten nejpoužívanější šum. Taky nabízí dobré výsledky za velmi dobrou cenu.
- Value-Gradient noise (mřížkový a gradientní šum)
  Tento šum byl zvolen proto, protože prakticky dává dohromady oba předchozí šumy a napravuje jejich nevýhody, i když je výpočetně náročnější.

4.2 Návrh funkcí

Nyní lze navrhnout jména jednotlivých funkcí. Jména jsou volena tak, aby podle názvu vyplynula jejich funkce a jejich použití bylo intuitivní, tak jako jsou např. funkce jazyka OpenGL nebo knihovny Glut (glVertex3f(), glutSolidCube()). Tedy zásada je taková, že se jméno skládá z předpony knihovny, následuje jméno a přípona udává počet a formát parametrů. Stejné zásady se drží i návrh funkcí tohoto rozhraní. Předpona „ns“ jako „noise“, jméno funkce a přípona udává dimenzi, a zároveň i počet parametrů funkce. Parametry jsou stejného datového typu jako návratová hodnota funkce. Za základní datový typ je zvolen double, protože nabízí vysokou přesnost vypočítaných hodnot. Knihovna bude také obsahovat funkce pro fraktální sumu a turbulenci. Bylo by zbytečné počítat fraktální sumu a turbulenci pro 1D verze, tak byly funkce definovány pro každou metodu ve 2D a 3D.

Obecné zásady byly zmíněny, proto bude následovat seznam jednotlivých funkcí a jejich popis.

```
1.  double nsLattice1D(double x)
2.  double nsLattice2D(double x, double y)
3.  double nsLattice3D(double x, double y, double z)
4.  double nsLatticeFractalSum2D(double x, double y)
5.  double nsLatticeFractalSum3D(double x, double y, double z)
6.  double nsLatticeTurbulence2D(double x, double y)
7.  double nsLatticeTurbulence3D(double x, double y, double z)
8.  double nsPerlin1D(double x)
9.  double nsPerlin2D(double x, double y)
10. double nsPerlin3D(double x, double y, double z)
11. double nsPerlinFractalSum2D(double x, double y)
12. double nsPerlinFractalSum3D(double x, double y, double z)
13. double nsPerlinTurbulence2D(double x, double y)
14. double nsPerlinTurbulence3D(double x, double y, double z)
15. double nsValueGradient1D(double x)
16. double nsValueGradient2D(double x, double y)
17. double nsValueGradient3D(double x, double y, double z)
18. double nsValueGradientFractalSum2D(double x, double y)
19. double nsValueGradientFractalSum3D(double x, double y,
    double z)
20. double nsValueGradientTurbulence2D(double x, double y)
21. double nsValueGradientTurbulence3D(double x, double y,
    double z)
```

Názvy prototypů jednotlivých funkcí jsou uvedeny i s parametry a lze podle nich snadno odvodit význam funkce.

Tato kapitola uzavírala semestrální projekt. Při pokračování v diplomovém projektu již byly zjištěny další okolnosti vyplývající ze začátku implementace, tak byla tato kapitola trochu přepracována. Jak se ovšem na závěr ukázalo, došlo k mnohem větším změnám, proto je nutné brát tuto kapitolu opravdu jen jako návrh. Kdyby přeci byla kapitola přepracována až na závěr, kdy rozhraní má jistou podobu, pak už by to nebyl návrh rozhraní.

5 Implementace knihovny pro výpočet šumů

V předchozích kapitolách bylo teoreticky popsáno, co je to šum, jak jej vypočítat a jaké metody budou zařazeny do knihovny. Nyní je důležité vybrat programovací jazyk a zvážit jeho možnosti. Jako první programovací jazyk je zvolen jazyk C. Záměrně zde uvádím slova „Jako první“, protože implementace pomocí jazyka C++ a šablon by mohla být výhodnější. O tom ale až v některé z dalších kapitol.

Jazyk C nabízí jednoduchou a rychlou implementaci knihovny. První verze knihovny obsahovala klasické funkce, které byly implementovány v jazyce C s definovaným rozhraním v hlavičkovém souboru. Tyto funkce odpovídaly návrhu v předchozí kapitole s tím, že došlo k malé změně ve funkcích pro fraktální sumu a turbulenci. U těchto funkcí musely být přidány další parametry pro řízení výpočtu. Také ve funkci pro Lattice Value Noise došlo ke změně. V následujících kapitolách jsou implementace metod rozepsány i s názvy funkcí.

5.1 Lattice Value Noise

Je-li název tohoto šumu doslovně přeložen, tak znamená: „Mřížkový hodnotový šum“. Přesně tak byl i vysvětlen v semestrálním projektu. Při dalším hledání informací na internetu jsem ovšem našel jinou verzi tohoto šumu, která byla rovnou také implementována. Tato „nová“ verze nepočítá s vygenerovanou mřížkou hodnot, ale využívá pouze jednorozměrné pole hodnot a permutační pole. Obě pole mají velikost 256 a princip nalezení hodnot v tabulce hodnot je naprosto stejný s principem u Perlinova šumu. Po nalezení daného počtu hodnot se mezi nimi provede lineární, bilineární nebo trilineární interpolace. Výhoda této verze je zřejmá. Menší paměťová náročnost a odstranění opakovatelnosti. Teoreticky se ovšem už jedná o přístup do dvou polí a to by mohlo znamenat jisté zpomalení. Jelikož knihovna má být efektivní, rozhodnul jsem se zařadit vlastní implementaci tohoto šumu na základě předchozího teoretického výkladu s využitím dvojrozměrného pole. V knihovně vznikly dvě různé implementace pro jeden typ šumu a testováním se rozhodne, která funkce bude v konečné verzi knihovny. Možná to budou obě dvě.

5.1.1 Lattice Noise

Funkce: `nsLattice1D`, `nsLattice2D`, `nsLattice3D`

Tento návrh funkce je založen na dvojrozměrném a trojrozměrném poli hodnot, do kterého se přistupuje přímo podle souřadnic předaných funkci. Přístup k prvkům dvojrozměrného a trojrozměrného pole je přitom stejně rychlý jako přístup do jednorozměrného. Pokud by například

mělo být naplněno pole o velikosti 256x256 hodnotami pomocí šumové funkce, pak tato funkce bude zavolána celkem 65536krát, což není zanedbatelné a mohlo by dojít ke zrychlení oproti verzi s permutačním polem.

Ovšem tato implementace má dvě nevýhody. První nevýhodou je paměťová náročnost. 2D verze s dvojrozměrným polem „zabere“ 256 kB paměti v případě datového typu *float* a 512kB v případě typu *double*. Tato hodnota není nikterak velká, ale pro 3D verzi, kde by bylo použito trojrozměrné pole o velikosti 256x256x256, pak bude potřeba 64MB paměti pro *float* a to už je doslova nepřijatelné. Pro 3D verzi by bylo teoreticky možné použít pouze dvojrozměrné pole a indexy do pole pro další čtyři body, které určuje třetí souřadnice, dopočítávat na základě již zjištěných hodnot v poli. Další možností je zmenšení trojrozměrného pole na velikost 32x32x32, čímž klesne paměťová náročnost na 128kb pro *float* a 256 kB pro *double*. V tomto případě se ale může vyskytnout viditelné opakování. Všechny tyto otázky budou zodpovězeny pomocí jednotlivých testů.

5.1.2 Value Noise

Funkce: `nsValue1D`, `nsValue2D`, `nsValue3D`

Funkce využívající jednorozměrné permutační pole a pole hodnot. Obě pole mají velikost 256, která by měla stačit tomu, aby nedocházelo k viditelnému opakování. Pole hodnot datového typu float je velké 1kB a double 2kB. Permutační pole je datového typu unsigned char o velikosti 256B. Oproti předchozí verzi se jedná opravdu o velkou úsporu paměti. Výhodu by tato funkce měla mít právě v odstranění opakovatelnosti.

5.2 Perlin Noise

Funkce: `nsPerlin1D, nsPerlin2D, nsPerlin3D`

Perlinův šum je velice známý a hodně používaný. Vymýšlet novou verzi implementace Perlinova šumu by bylo asi zbytečné, proto byl implementován originální kód šumu přímo ze zdroje [19] od Kena Perlina, který jej implementoval v roce 1984 v jazyce C.

Perlinův šum využívá permutační pole a pole gradientů, které ovšem musí být ve třech verzích. Je to proto, že pro 1D není hodnotou vektor, ale jen číslo, pro 2D je to pole, které uchovává vektor o dvou složkách a ve 3D jsou v poli uloženy trojsložkové vektory. Všechna pole mají velikost 514. Tato velikost je nastavena naprosto účelně. Při inicializaci se naplní všech 256 hodnot a ty se pak ještě jednou zkopírují do zbytku pole, plus dvě počáteční hodnoty. Při tomto způsobu se totiž ušetří jedna bitová operace and, aby se číslo nedostalo mimo rozsah pole. To znamená malé zrychlení výpočtu.

Paměťová náročnost Perlinova šumu je mnohem větší než u Value noise. Permutační tabulka je datového typu int a má velikost 514, což jsou 2kB. Pole gradientů mají stejné velikosti, ale pro

každou dimenzi obsahují různý počet položek. Pro 1D to je jeden prvek na jeden index a v případě datového typu float to jsou 2kB a 4kB pro double. S počtem dimenzí se velikost zdvojnásobí. Největší paměťová náročnost nastává pro datový typ double a pro výpočet Perlinova šumu je poté zapotřebí zhruba 27kB paměti. Samozřejmě v této hodnotě nejsou započítány lokální proměnné uvnitř funkcí.

5.3 Value-Gradient Noise

Funkce: `nsValueGradient1D, nsValueGradient2D, nsValueGradient3D`

Tento typ šumu spojuje value noise a gradient noise a výsledkem je vážená suma hodnot. Pokud by se měl nejdříve vypočítat value noise a pak Perlin noise, byl by výpočet časově náročný. Proto byla provedena taková implementace, kde jsou společné výpočty provedeny na začátku. Pro Value-Gradient noise jsou přitom použita stejná pole, jako pro Perlin noise a tak paměťová náročnost zůstává stejná. Value noise je vypočítán pomocí permutačního pole a gradientního pole pro 1D, kde jsou uloženy jen čísla tak, jako v poli hodnot pro Value noise. Poté se klasicky vypočítá Perlinův šum. Přitom na začátku jsou provedeny výpočty indexů do permutačního pole, které jsou pro obě metody stejné. Těmito drobnými optimalizacemi je dosaženo co nejkratšího času výpočtu.

5.4 Fraktální suma a turbulence

Funkce: `nsFractalSum2D, nsFractalSum3D, nsTurbulence2D, nsTurbulence3D`

Fraktální sumu a turbulenci lze provést nad libovolným šumem, a proto byly definovány pro všechny 2D a 3D funkce. V knihovně je celkem osm 2D a 3D funkcí a to by znamenalo každou z nich implementovat osmkrát. Celkem by knihovna narostla o 16 funkcí, které mají stejný princip, ale pokaždé by uvnitř volaly různé funkce. Původní návrh počítal pouze se souřadnicovými parametry, ale až implementace ukázala, že je potřeba více parametrů. Celkem přibyly čtyři parametry. Jeden parametr slouží pro identifikaci základní šumové funkce, nad kterou se bude provádět výpočet a další tři parametry slouží k určení persistence, frekvence a počtu oktáv. Těmito parametry se určuje stupeň detailů a velikost vzoru v textuře, proto nemohou být nastaveny „napevno“ uvnitř funkce, protože pro každý typ textury je nutné je individuálně nastavit.

Deklarace funkcí:

```
1. double nsFractalSum2D(NSenum function, double x, double y, double
   alpha, double beta, int n);
2. double nsTurbulence2D(NSenum function, double x, double y, double
   alpha, double beta, int n);
3. double nsFractalSum3D(NSenum function, double x, double y, double
   z, double alpha, double beta, int n);
```

4. `double nsTurbulence3D(NSenum function, double x, double y, double z, double alpha, double beta, int n);`

5.5 Inicializační funkce

Předtím, než se zavolá kterákoliv šumová funkce, musí být permutační pole a pole hodnot a gradientů naplněny. Proto byly vytvořeny inicializační funkce pro jednotlivé metody. Tyto funkce ale uživatel volá nepřímo pomocí jiných funkcí. Inicializace probíhá pomocí generátoru náhodných čísel, kterému se nejdříve vygeneruje hodnota na základě „semínka“, prvotní hodnoty. První funkcí, kterou lze provést inicializaci je `setSeed(unsigned int seed)`, která nastaví hodnotu „semínka“ pro generátor náhodných čísel. Druhou funkcí je `setTabSize(int size)`, která nastaví velikosti permutačních polí a polí hodnot a gradientů. Tato funkce je užitečná tam, kde uživatel chce, aby pole měla menší velikost nebo dosáhnout jisté opakovatelnosti. Přičemž minimální hodnota je 8 a maximální 256. Funkce `int getTabSize()` naopak vrátí aktuální hodnotu velikosti polí.

Jednotlivé funkce byly naprogramovány a vytvořena první verze knihovny. Jak ale stojí v zadání, tak knihovna má obsahovat efektivní a rychlou implementaci. Každá optimalizace je spojena s testováním a měřením výsledků. Další kapitola se zabývá metodikou testování a optimalizací.

6 Testování knihovny a optimalizace

Všechny testy byly prováděny pro zjištění doby výpočtu a vizuálních výsledků všech funkcí. Nejdříve je ovšem nutné zmínit něco o testovacím počítači a metodě testování.

6.1 Testovací sestava

Testování probíhalo na počítači s konfigurací:

Procesor: Intel Celeron 2,4 GHz přetaktovaný na 2,8 GHz, 256kB cache

Paměti: 2 x 256 MB DDR RAM v dual channel na frekvenci 400 MHz

Grafická karta: nVidia GeForce FX5700 256MB VRAM

6.2 Metoda testování

Testování probíhalo v OpenGL vytvářením 2D textury pomocí pole o velikosti 256x256, které bylo naplněno hodnotami zavoláním šumové funkce. Toto pole pak sloužilo jako zdroj dat pro vytvoření pole pro texturu. Pro měření času výpočtu byla použita knihovna <time.h>, ze které byla použita funkce clock(), která zjistí aktuální počet tiků od spuštění programu.

Ve funkci pro naplnění pole hodnotami, byla nejdřív uložena hodnota počtu tiků a pak provedeny dva cykly for, ve kterých se naplnilo pole o velikosti 256x256 čísel. Po skončení cyklů, byl opět přečten počet tiků. Tyto dvě hodnoty pak byly od sebe odečteny a vyděleny hodnotou, která udává počet tiků za sekundu. Tato hodnota je také v knihovně definována. Ovšem po prvním měření bylo zjištěno, že naplnění pole je tak rychlé, že se to nedalo ani změřit, proto byl celý cyklus plnění pole opakován několikrát a to tak, aby celý cyklus probíhal přibližně jednu až dvě sekundy. Výsledný čas byl nakonec počtem opakování zase vydělen a udává, za jak dlouho bylo pole o velikosti 256x256 danou funkcí naplněno po jednom průchodu.

Aby bylo měření přesnější, bylo provedeno pokaždé deset výpočtů hned za sebou a vypočítán průměr. Výsledná hodnota byla zaokrouhlena na tři desetinná místa. Mají-li být výsledky opravdu přesné, pak je nutné také uvést režii smyček for. Tyto smyčky a režie s tím spojená byly změřeny s výsledkem 0,684 ms.

6.3 Podmínky testování

Než budou uvedeny samotné výsledky testů, je dobré napsat, za jakých podmínek bylo testováno a jaký vliv měly tyto podmínky na testování.

Funkce byly testovány v průběhu celého dne, kdy byl počítač využíván nejen za účelem testování, ale i pro zábavu a jiné činnosti. Když byly večer porovnány výsledky testů, které by si měly odpovídat, nebylo tomu tak. Večerní výsledky byly někdy až o dvě milisekundy horší! Je to způsobeno stavem operační paměti a také počtem procesů, které byly v danou chvíli spuštěny. Bohužel se nedalo ohlídat, aby si některý program nespustil nějaký proces navíc a pamatovat si, který to byl a pokaždé před testováním to kontrolovat. Řešením by byl restart počítače, ale to by se testy poněkud prodloužily.

Nicméně to ve výsledku nemá na testy vliv. Pokaždé bylo totiž testováno za nějakým účelem porovnání a všechny tyto testy na daných funkcích probíhaly těsně za sebou za stejných podmínek. Vyhodnocení mezi takovými měřeními je provedeno v procentech, takže vliv operační paměti a procesů je odstraněn. To, že jeden den funkce provedla výpočet za nějaký čas a druhý den za čas o dvě milisekundy horší nevadí, protože nejsou vzájemně srovnávány.

6.4 První výsledky

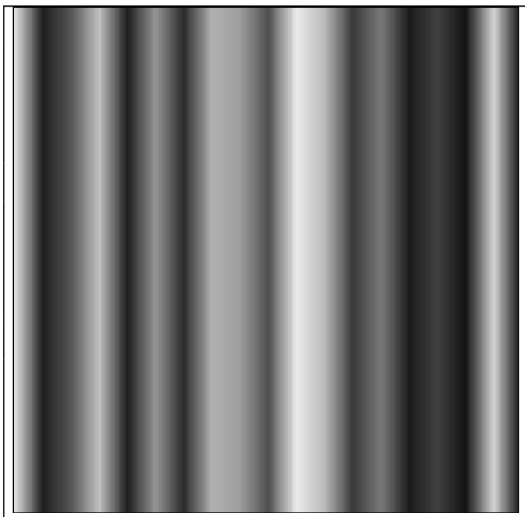
Před optimalizací byly provedeny první vizuální a časové testy. V následujících kapitolách jsou uvedeny jednotlivé funkce po dimenzích s popisem, vizuálními a časovými výsledky.

6.4.1 Jednodimenzionální šumové funkce

Tyto funkce byly tak rychlé, že bylo nutné celý cyklus naplnění dvourozměrného pole opakovat 500krát, aby došlo ke zpřesnění měření.

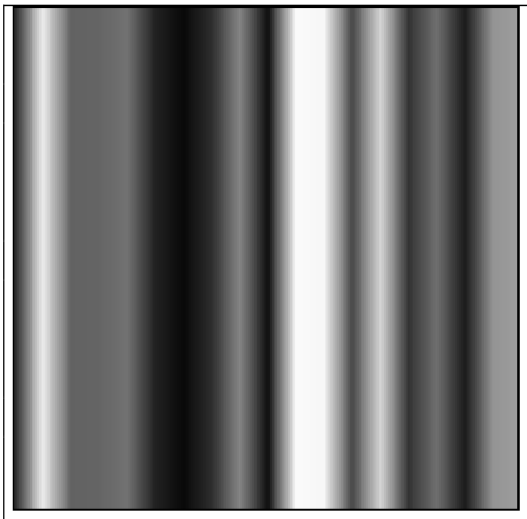
Funkce: `nsLattice1D, nsValue1D, nsPerlin1D, nsValueGradient1D`

Nový návrh řešení pomocí přímého přístupu do dvourozměrného pole s pseudonáhodnými hodnotami. To je funkce nsLattice.

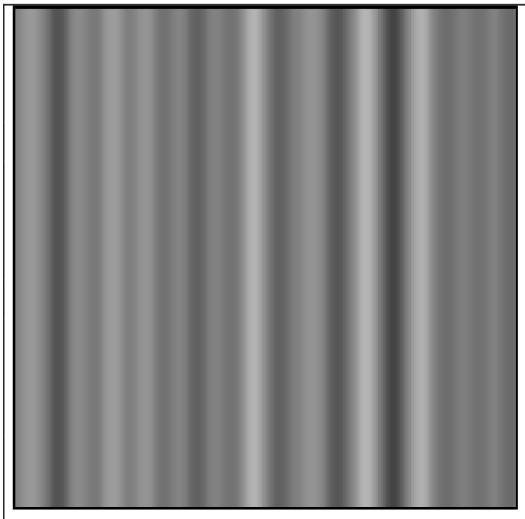


Naměřené hodnoty [ms]:
3,658
3,594
3,592
3,562
3,562
3,624
3,594
3,562
3,562
3,594
Průměrná hodnota: 3,590 ms

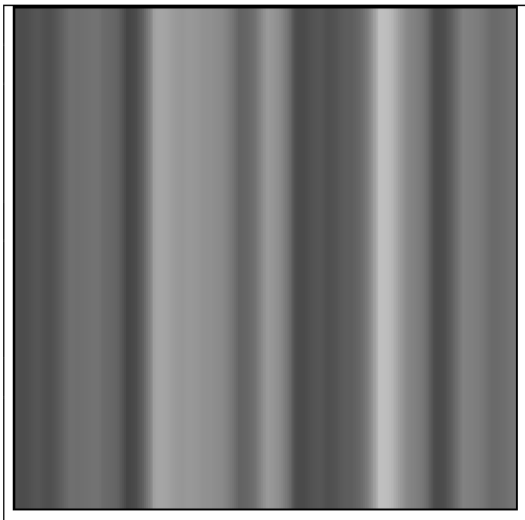
Tabulka 1.: Obrázek a naměřené hodnoty pro nsLattice1D

	Naměřené hodnoty [ms]:
	3,594
	3,562
	3,624
	3,688
	3,564
	3,562
	3,594
	3,564
	3,594
	3,594
	Průměrná hodnota: 3,594 ms

Tabulka 2.: Obrázek a hodnoty pro nsValue1D

	Naměřené hodnoty [ms]:
	4,750
	4,686
	4,658
	4,686
	4,688
	4,656
	4,626
	4,688
	4,626
	4,780
	Průměrná hodnota: 4,684 ms

Tabulka 3.: Obrázek a hodnoty pro nsPerlin1D

	Naměřené hodnoty [ms]:
	6,406
	6,376
	6,374
	6,374
	6,562
	6,594
	6,344
	6,468
	6,374
	6,408
	Průměrná hodnota: 6,428 ms

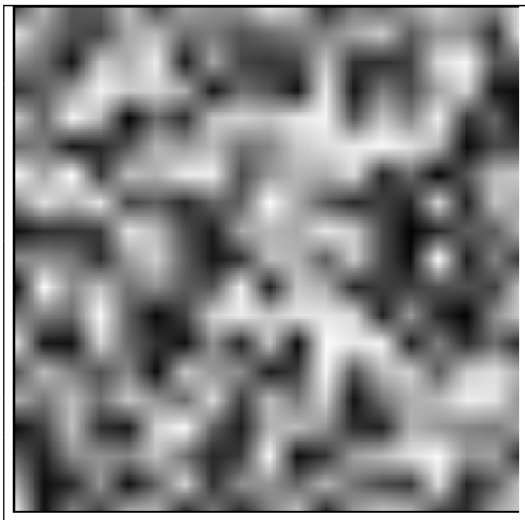
Tabulka 4.: Obrázek a hodnoty pro nsValueGradient1D

Tak jak bylo asi očekáváno, tak časová náročnost jednotlivých funkcí postupně roste. Teď se ale zaměříme na porovnání nsLattice1D a nsValue1D. Předpoklad byl takový, že by mohlo dojít ke zrychlení při přímém přístupu do pole. Jisté zrychlení tu je, ale je velmi mizivé. Pouhé 4 mikrosekundy. Zatím se nová implementace nevyplatí, protože tabulka hodnot zabere mnohem více paměti. Rozhodnou ještě testy ve 2D a 3D.

6.4.2 Dvojdimenzionální šumové funkce

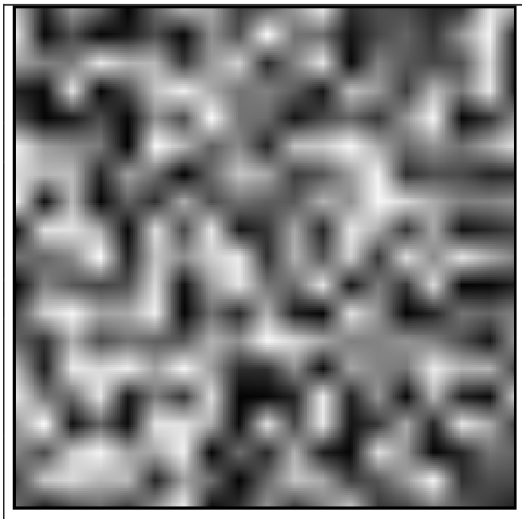
V případě dvojdimenzionálních funkcí již nebylo nutné opakovat cyklus 500krát, ale stačilo pouze 250krát.

Funkce: nsLattice2D, nsValue2D, nsPerlin2D, nsValueGradient2D



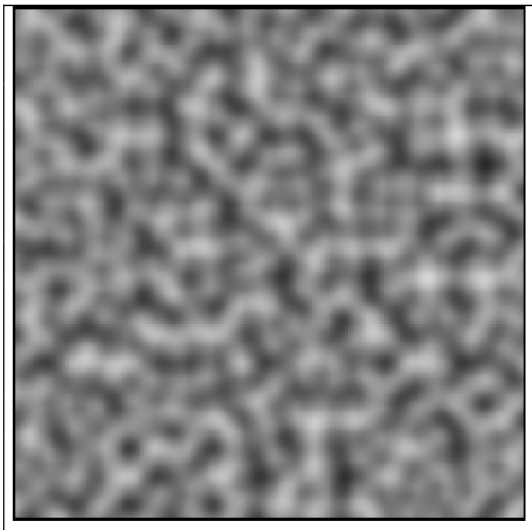
Naměřené hodnoty [ms]:
6,564
6,248
6,312
6,436
6,188
6,376
6,376
6,436
6,252
6,436
Průměrná hodnota: 6,362 ms

Tabulka 5.: nsLattice2D

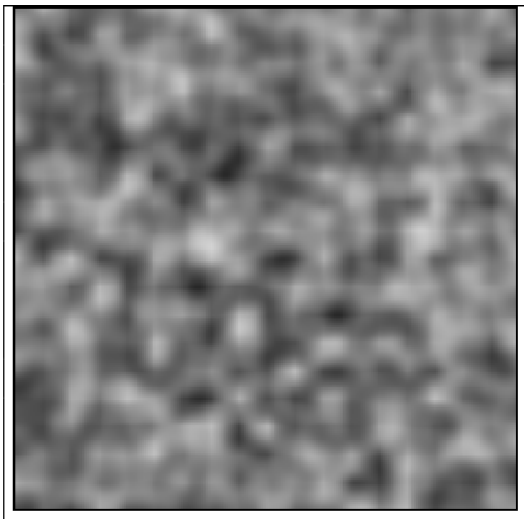


Naměřené hodnoty [ms]:
7,812
7,876
7,812
7,812
7,872
7,812
7,812
7,876
8,000
7,812
Průměrná hodnota: 7,850 ms

Tabulka 6.: nsValue2D

	Naměřené hodnoty [ms]:
	8,684
	8,688
	8,688
	8,560
	8,748
	8,812
	8,752
	8,688
	8,752
	8,812
	Průměrná hodnota: 8,718 ms

Tabulka 7.: nsPerlin2D

	Naměřené hodnoty [ms]:
	11,624
	11,624
	11,748
	11,624
	11,688
	11,748
	11,688
	11,752
	11,560
	11,564
	Průměrná hodnota: 11,662 ms

Tabulka 8.: nsValueGradient2D

Ve 2D je již znát rozdíl mezi přímým přístupem do tabulky hodnot a pomocí permutačního pole. Při porovnání obou implementací je jediný rozdíl v získání indexu do pole hodnot. Pro získání indexu je použito makro, které přičte druhý vstupní parametr k hodnotě získané z permutačního pole z indexu daného prvním parametrem. V tomto případě by pak mohlo dojít k „sáhnutí“ mimo permutační pole, proto musí být tato hodnota upravena do maximální hodnoty velikosti pole. Jako první se naskytne operace *modulo*, která byla také původně implementována. Po prvních testech bylo ale zřejmé, že použití této operace není vhodné.

Výsledky s operací „modulo“:

14,748	13,124	12,876	13,128	13,000	13,064	13,060	13,000	13,000	13,560
Průměrná hodnota: 13,256 ms									

Při dosažení těchto výsledků bylo opravdu nutné použít jinou operaci, protože se jedná o 40% zpomalení. Nejvýhodnější je použití bitové operace „and“, která je v podstatě použita. Jako další možnost optimalizace by mohlo být rozšíření permutačního pole na dvojnásobnou velikost, kde by nebylo potřeba nic upravovat. O tom ale až v některé z následujících kapitol. Nyní následuje testování 3D funkcí.

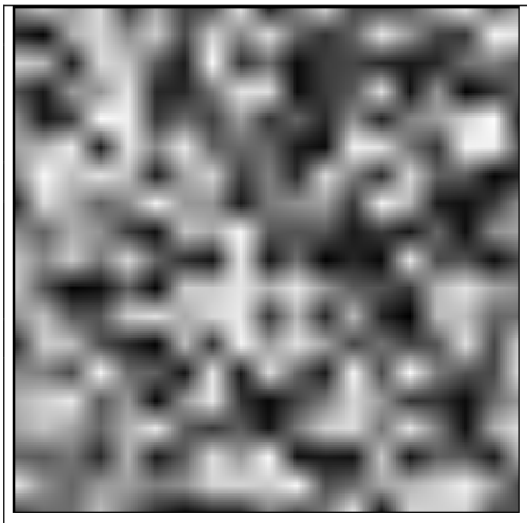
6.4.3 Třídimenzionální šumové funkce

Tyto funkce mají tři vstupní parametry a správně by se měly testovat při vytváření 3D textur tak, že se budou měnit všechny tři parametry ve třech cyklech. To by ovšem znamenalo změnu testovacích cyklů a vytvoření třírozměrné textury a poté její namapování. Tato textura také bude mít velký datový objem. Jelikož předchozí testy probíhaly tak, že se vytvářela 2D textura jak pro dvojdimenzionální, tak i pro jednodimenzionální funkce, tak jsem se rozhodl zachovat stejnou metodu i pro trojdimenzionální funkce.

Režie smyček je stejná a funkce jsou volány 65536krát, aby se naplnilo pole o velikosti 256x256. Může se ale zdát, že při volání funkce je zanedbán jeden parametr a pak je časová náročnost menší. Třetí parametr není zanedbán, ale jeho hodnota je stále nulová a všechny výpočty se musí provést i s touto hodnotou a tak je časová náročnost funkce stejná a lze ji porovnat s výsledky dvojdimenzionálních funkcí. Vizuální výsledky jsou stejné jako 2D verze. Cyklus byl pro 3D funkce opakován 100krát.

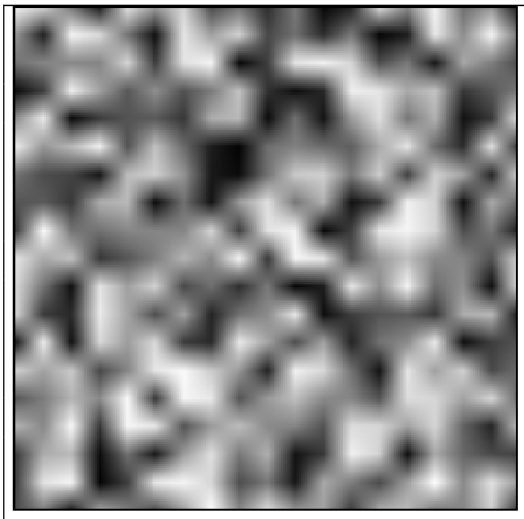
Funkce: `nsLattice3D, nsValue3D, nsPerlin3D a nsValueGradient3D`

V testované verzi nsLattice3D je použito dvojrozměrné pole o velikosti 256x256 a indexy do pole pro další čtyři hodnoty jsou dopočítávány přičtením třetího parametru.

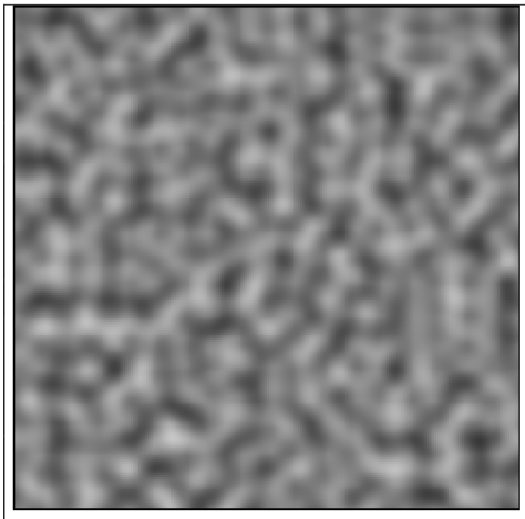


Naměřené hodnoty [ms]:
10,470
10,780
10,470
10,470
10,470
10,460
10,470
10,620
10,630
10,620
Průměrná hodnota: 10,546 ms

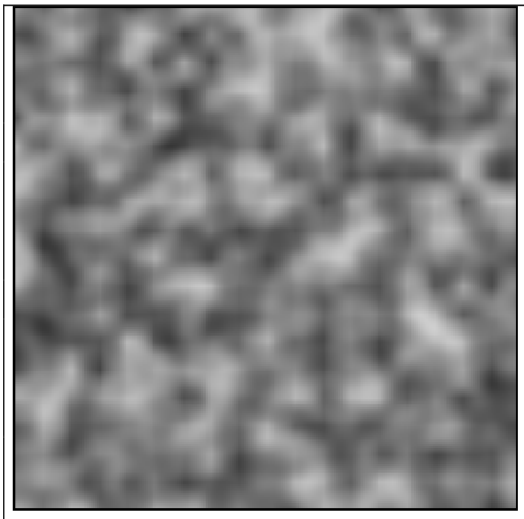
Tabulka 9.: nsLattice3D

	Naměřené hodnoty [ms]:
	12,500
	12,660
	12,500
	12,820
	12,500
	12,650
	12,500
	12,660
	12,500
	13,280
	Průměrná hodnota: 12,657 ms

Tabulka 10.: nsValue3D

	Naměřené hodnoty [ms]:
	16,250
	16,090
	16,090
	16,100
	16,090
	16,100
	16,250
	16,100
	15,940
	15,940
	Průměrná hodnota: 16,095 ms

Tabulka 11.: nsPerlin3D

	Naměřené hodnoty [ms]:
	21,720
	21,720
	21,720
	22,030
	21,880
	21,880
	21,720
	21,720
	22,030
	21,880
	Průměrná hodnota: 21,830 ms

Tabulka 12.: nsValueGradient3D

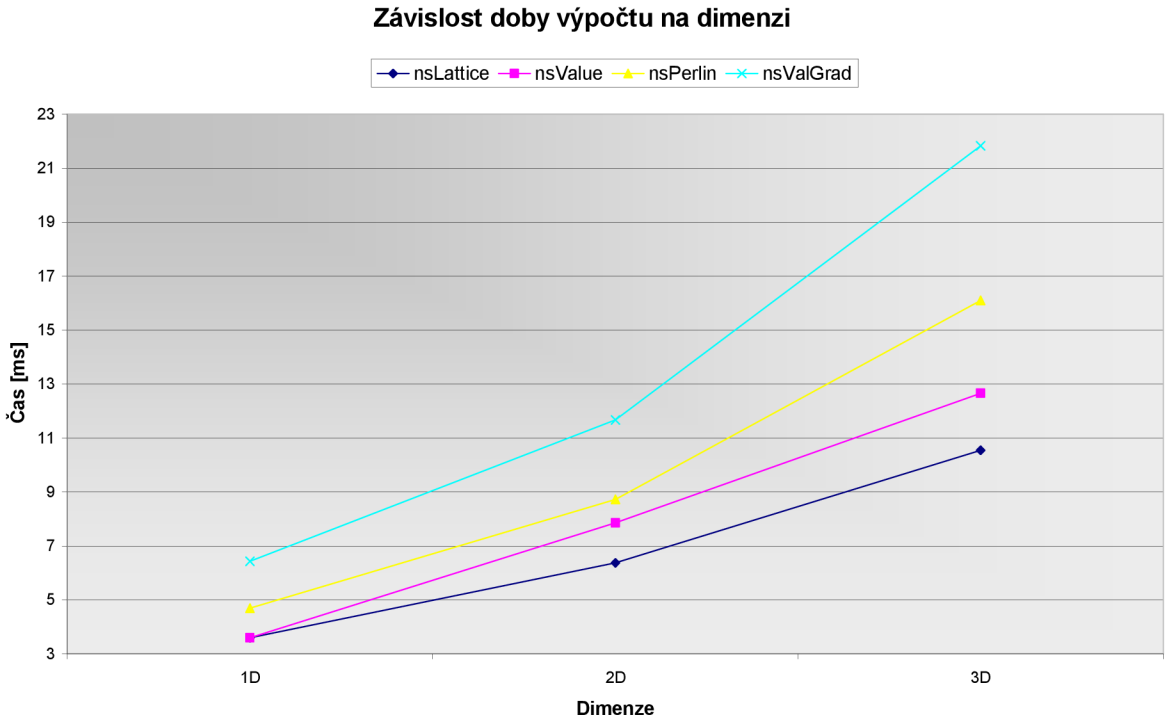
Tím, že byla zachována metoda testování pomocí 2D pole, tak lze také porovnat 3D funkce s 2D verzemi. Výsledné časy ukazují, za jak dlouho bylo pole naplněno každou 3D funkcí (tak jako u předchozích 2D).

Nejdříve jsou ale srovnány výsledky Lattice a Value noise. I ve 3D dochází k viditelnému zrychlení. Pro naplnění pole byla funkce nsLattice3D o 2 ms rychlejší, uvedeno v procentech 16,68%. Samozřejmě optimalizace bude teprve provedena, ale zatím ze „souboje" vychází lépe tato funkce. Ve výsledku ještě také ale musí rozhodnout konečné vizuální výsledky pro 3D verze funkcí provedené ve 3D. Tyto testy budou následovat.

Nyní lze provést předběžný závěr testů. Naměřené hodnoty nejsou asi ničím překvapivým. Tak jak bylo napsáno již dříve o časové náročnosti těchto funkcí, tak naměřené hodnoty tyto předpoklady jen potvrdily.

Je zajímavé pozorovat, že nejnáročnější implementovaný šum Value-Gradient je vždy o málo pomalejší než o dimenzi vyšší šumová funkce Lattice noise.

Nejlépe je vše vystihnuto v následujícím grafu, kde lze vidět průběh jednotlivých funkcí pro všechny dimenze.



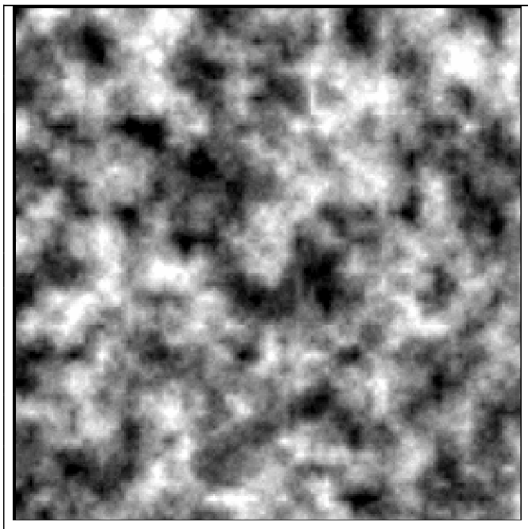
Graf 1.: Závislost doby výpočtu na dimenzi

Z grafu je nejzajímavější křivka pro Value noise. Ta má čistě lineární průběh, což by mohlo vést k dobré optimalizaci pro 3D verzi.

6.4.4 Šumové funkce složené z více oktáv (fraktální suma a turbulence)

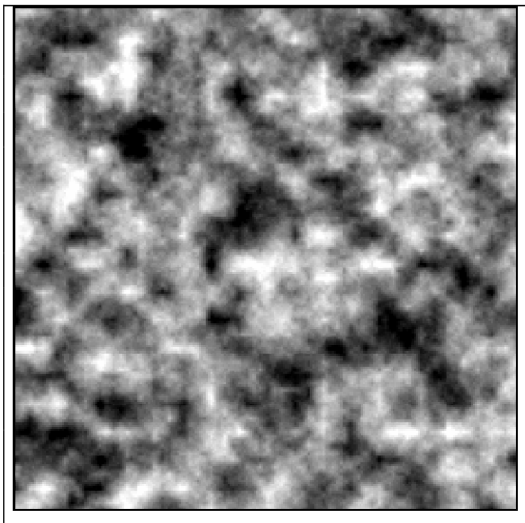
Funkce: `nsLatticeFractalSum2D, nsValueFractalSum2D, nsPerlinFractalSum2D, nsValueGradientFractalSum2D, nsLatticeFractalSum3D, nsValueFractalSum3D, nsPerlinFractalSum3D, nsValueGradientFractalSum3D`

Nejdříve jsou testovány funkce pro fraktální sumu. Záměrně už jsou testovány 2D a 3D verze společně, protože vizuální výsledky jsou ve 2D naprosto stejné, pouze čas bude jiný.



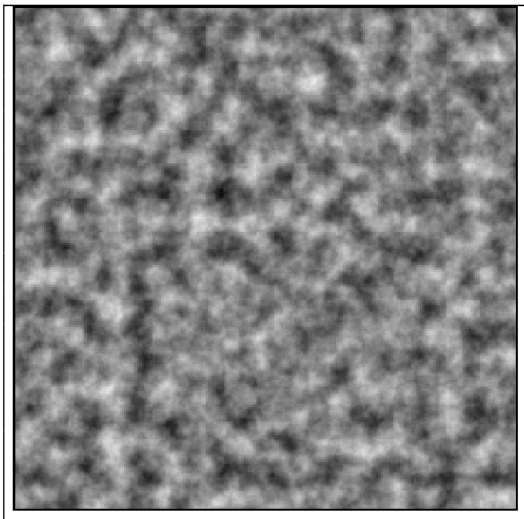
Naměřené hodnoty [ms]:	
nsLatticeFractalSum2D	nsLatticeFractalSum3D
70,300	101,500
67,200	101,600
67,200	100,000
67,200	100,000
65,600	101,500
65,700	101,500
67,200	100,000
65,700	101,500
65,600	100,000
65,600	100,000
Průměrná hodnota: 66,730 ms	Průměrná hodnota: 100,760 ms

Tabulka 13.: nsLatticeFractalSum2D, nsLatticeFractalSum3D



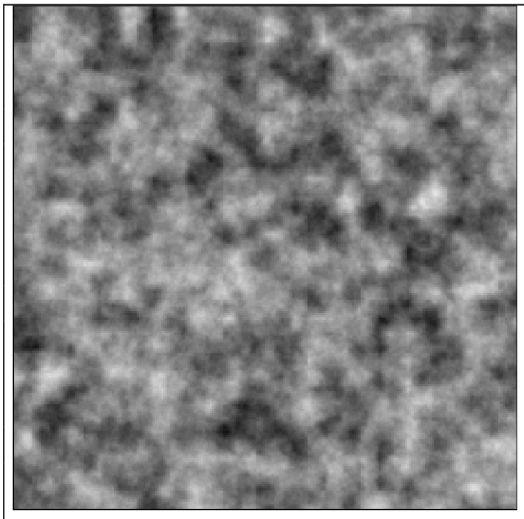
Naměřené hodnoty [ms]:	
nsValueFractalSum2D	nsValueFractalSum3D
76,500	117,200
78,200	118,700
79,700	118,700
78,100	118,800
78,100	118,700
78,100	117,200
78,100	117,200
78,100	117,200
78,200	118,800
78,100	118,800
Průměrná hodnota: 78,120 ms	Průměrná hodnota: 118,130 ms

Tabulka 14.: nsValueFractalSum2D, nsValueFractalSum3D



Naměřené hodnoty [ms]:	
nsPerlinFractalSum2D	nsPerlinFractalSum3D
84,400	146,900
82,800	145,300
82,800	142,200
82,800	142,200
81,300	145,400
81,200	143,700
82,800	142,200
81,200	142,100
82,900	145,300
82,800	145,300
Průměrná hodnota: 82,500 ms	Průměrná hodnota: 144,060 ms

Tabulka 15.: nsPerlinFractalSum2D, nsPerlinFractalSum3D



Naměřené hodnoty [ms]:	
nsValueGradient-FractalSum2D	nsValueGradient-FractalSum3D
107,800	189,100
106,200	187,500
103,100	189,100
103,100	190,600
106,200	190,600
104,700	190,600
104,700	189,100
104,700	190,600
104,700	189,100
106,200	192,100
Průměrná hodnota: 105,140 ms	Průměrná hodnota: 189,840 ms

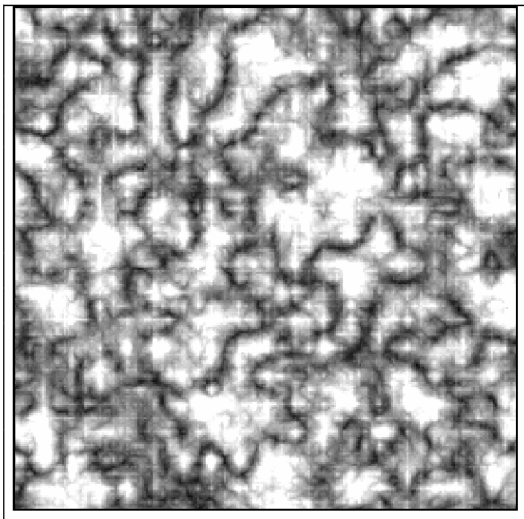
Tabulka 16.: nsValueGradientFractalSum2D, nsValueGradientFractalSum3D

Všechny naměřené hodnoty nejsou ničím novým. Časová náročnost roste a opět se potvrdilo, že šum ValueGradient je přibližně stejně rychlý jako o dimenzi vyšší Lattice šum. Hodnoty nemusí být samozřejmě konečné, protože ještě bude provedena optimalizace základních funkcí.

Následují testy funkcí pro výpočet turbulence.

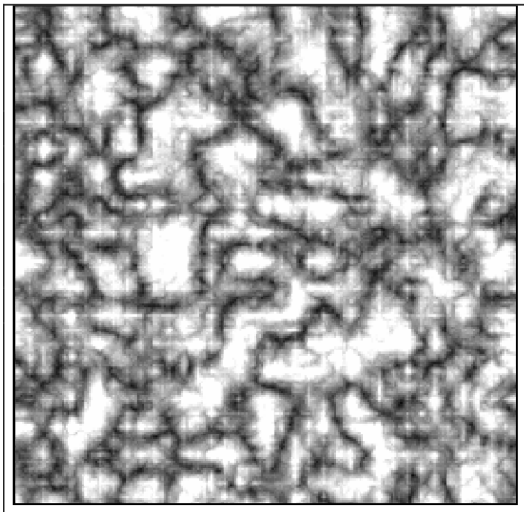
Funkce: `nsLatticeTurbulence2D, nsValueTurbulence2D,`
`nsPerlinTurbulence2D, nsValueGradientTurbulence2D,`
`nsLatticeTurbulence3D, nsValueTurbulence3D,`
`nsPerlinTurbulence3D, nsValueGradientTurbulence3D`

Funkce pro turbulenci jsou implementovány stejně jako pro fraktální sumu, ale výsledek volání šumové funkce je absolutní hodnota, která je zajištěna funkcí *fabs()* z knihovny *math.h*. Jako optimalizace bylo vyzkoušeno rozepsání funkce do podoby podmínky *if*, ale tímto způsobem bylo dosaženo horších výsledků. Cca o 20 milisekund více.



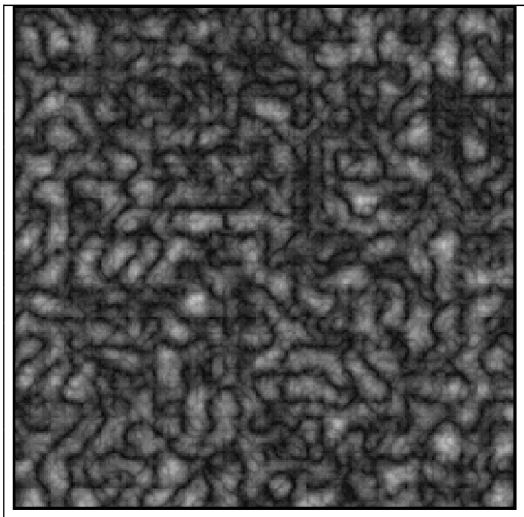
Naměřené hodnoty [ms]:	
nsLatticeTurbulence2D	nsLatticeTurbulence3D
68,800	101,600
67,200	101,600
67,200	100,000
67,200	100,000
67,200	101,500
67,200	103,100
67,200	101,600
68,700	101,500
67,200	101,600
65,700	101,500
Průměrná hodnota: 67,360 ms	Průměrná hodnota: 101,400 ms

Tabulka 17.: nsLatticeTurbulence2D, nsLatticeTurbulence3D



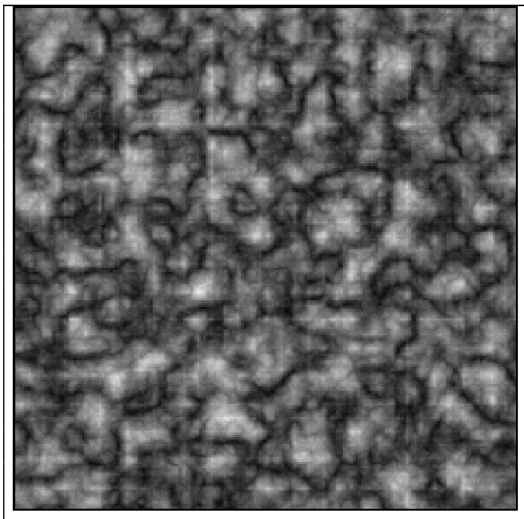
Naměřené hodnoty [ms]:	
nsValueTurbulence2D	nsValueTurbulence3D
79,600	123,500
79,600	117,200
79,700	117,200
79,700	118,700
84,400	118,800
81,200	118,800
79,700	118,800
78,100	118,700
79,700	120,400
78,100	120,400
Průměrná hodnota: 79,980 ms	Průměrná hodnota: 119,250 ms

Tabulka 18.: nsValueTurbulence2D, nsValueTurbulence3D



Naměřené hodnoty [ms]:	
nsPerlinTurbulence2D	nsPerlinTurbulence3D
84,400	145,300
84,400	143,700
84,400	145,300
84,400	146,800
84,300	143,700
82,800	143,800
81,300	143,800
85,900	142,200
84,400	143,800
82,800	143,800
Průměrná hodnota: 83,910 ms	Průměrná hodnota: 144,220 ms

Tabulka 19.: nsPerlinTurbulence2D, nsPerlinTurbulence3D

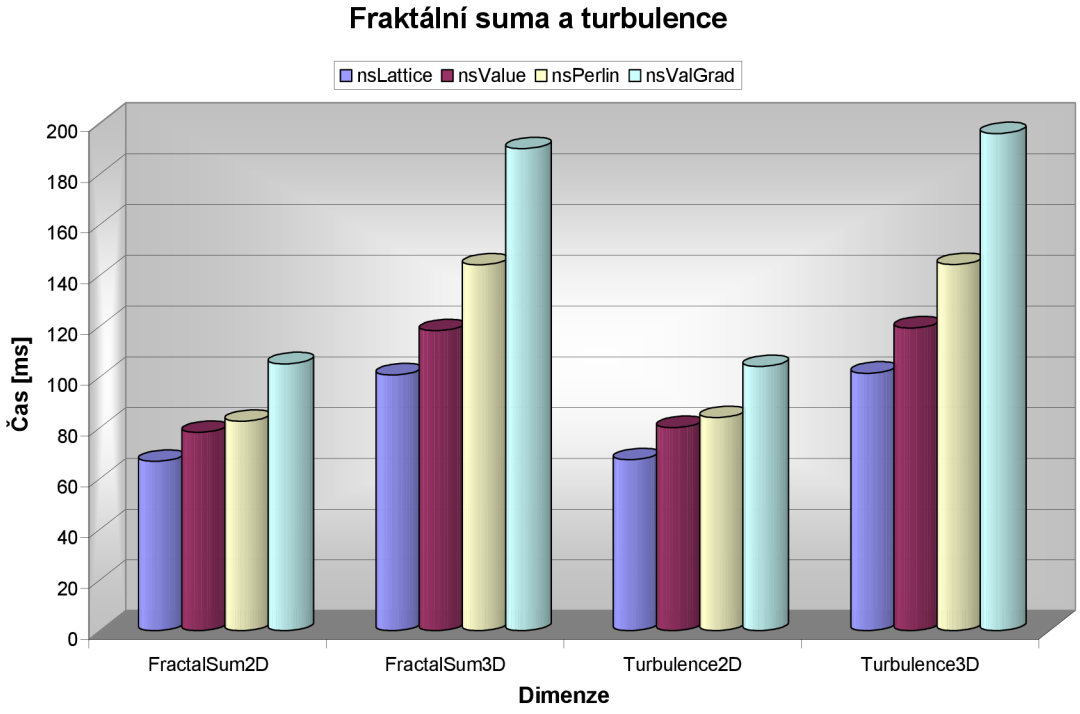


Naměřené hodnoty [ms]:	
nsValueGradient-Turbulence2D	nsValueGradient-Turbulence3D
109,400	198,400
104,700	195,400
103,100	196,900
101,600	195,300
104,700	195,300
104,700	193,800
101,600	196,800
103,100	195,300
103,100	195,300
104,700	195,400
Průměrná hodnota: 104,070 s	Průměrná hodnota: 195,790 s

Tabulka 20.: nsValueGradientTurbulence2D, nsValueGradientTurbulence3D

Jsou-li srovnány naměřené hodnoty s fraktální sumou, pak lze říci, že funkce fabs() má na časovou náročnost velmi malý vliv. Jedná se o stovky mikrosekund. Turbulence i fraktální suma byly tvořeny celkem osmi oktávami, tedy základní šumové funkce byly volány (256 * 256 * 8)krát. Spočítáním časové náročnosti samotné šumové funkce nsLattice2D, která by byla zavolána ještě osmkrát více, pak je dosaženo průměrné hodnoty: 6,362 * 8 = 50,899. Při srovnání s hodnotou funkce fraktální sumy (66,730) je snadno zjištěna režijní doba funkcí pro fraktální sumu nebo turbulenci, která odpovídá přibližně 15,831 milisekundám.

Výsledky jsou opět ukázány pomocí grafu.



Graf 2.: Závislost doby výpočtu na metodě a dimenzi

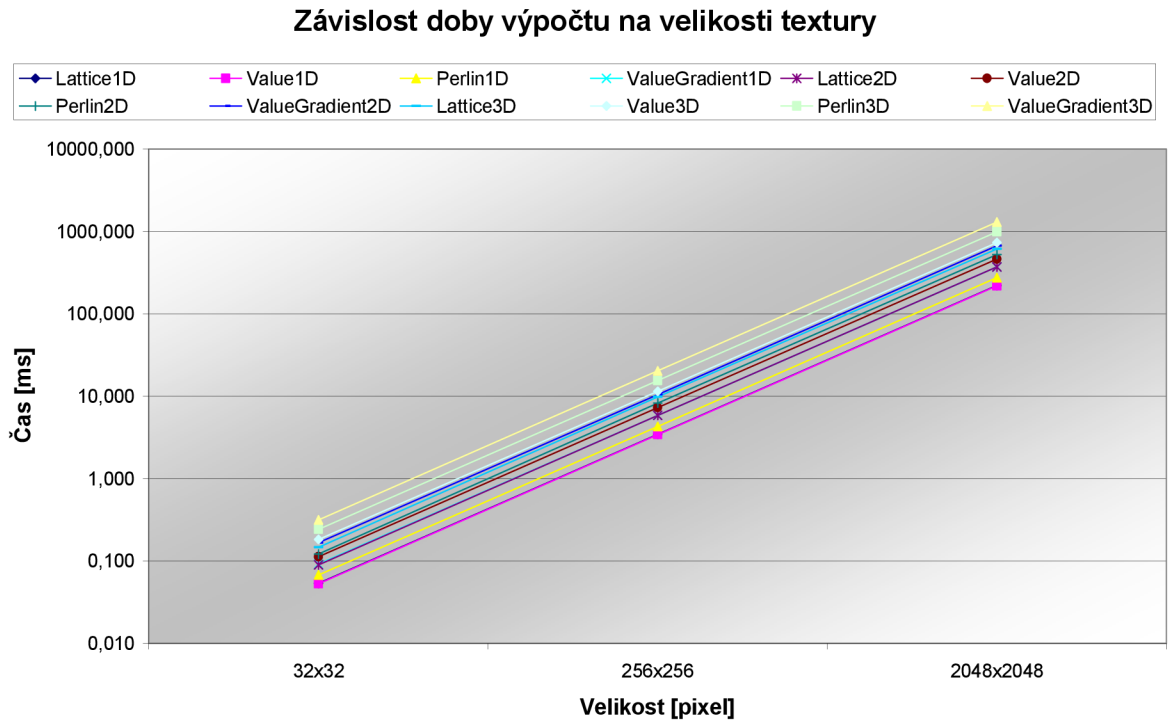
6.5 Různé velikosti textur

Dalším zajímavým testem by mohlo být porovnání času, kdy by byly vytvářeny textury o různých velikostech. Od malých (32x32) přes střední (256x256) až po velké (2048x2048).

Podle předpokladu by se funkce měly chovat stejně a přibližně „lineárně“. Po provedení testů a vytvoření grafu se tyto předpoklady splnily.

Funkce	Velikost textury		
	32x32	256x256	2048x2048
nsLattice1D	0,053	3,448	220,100
nsValue1D	0,053	3,420	218,986
nsPerlin1D	0,068	4,298	273,686
nsValueGradient1D	0,091	5,866	371,214
nsLattice2D	0,090	5,856	370,528
nsValue2D	0,112	7,268	462,043
nsPerlin2D	0,121	8,258	525,686
nsValueGradient2D	0,167	10,497	671,628
nsLattice3D	0,146	9,789	614,071
nsValue3D	0,182	11,440	731,457
nsPerlin3D	0,242	15,475	990,400
nsValueGradient3D	0,314	20,327	1302,443

Tabulka 21.: Hodnoty výpočtů pro různé velikosti textur v milisekundách



Graf 3.: Závislost doby výpočtu na velikosti textury

Z grafu je zřetelně vidět chování funkcí pro různé velikosti textur. Jelikož složitost algoritmů je logaritmická a graf je vytvořen s logaritmickým měřítkem, jsou výsledky stejné „lineární“.

Po provedení prvních testů se zaměřím na možnosti zrychlení algoritmů, menší paměťovou náročnost nebo vizuální výsledky. Funkce v knihovně by měly být efektivní jak po časové stránce tak i pokud možno málo paměťově náročné. Vizuální výsledky, časová a paměťová náročnost spolu úzce souvisí a tak bude nutné najít určitý kompromis.

6.6 Value noise – rozšíření permutačního pole

Tak jak bylo napsáno, tento šum využívá permutační pole a pole hodnot. Tyto pole mají velikost 256 a hodnoty v permutačním poli od 0 do 255. Vstupní parametry je nutné vždy přepočítat do určitého rozsahu, protože nelze předpovědět, jaké hodnoty to budou. Pokud jsou převedeny do rozsahu od 0 do 255, pak u jednodimenzionální verze už nedochází k dalším výpočtům a přímo se přistoupí do permutačního pole, naleznou se indexy do pole hodnot, provede se interpolace a vrátí výsledek.

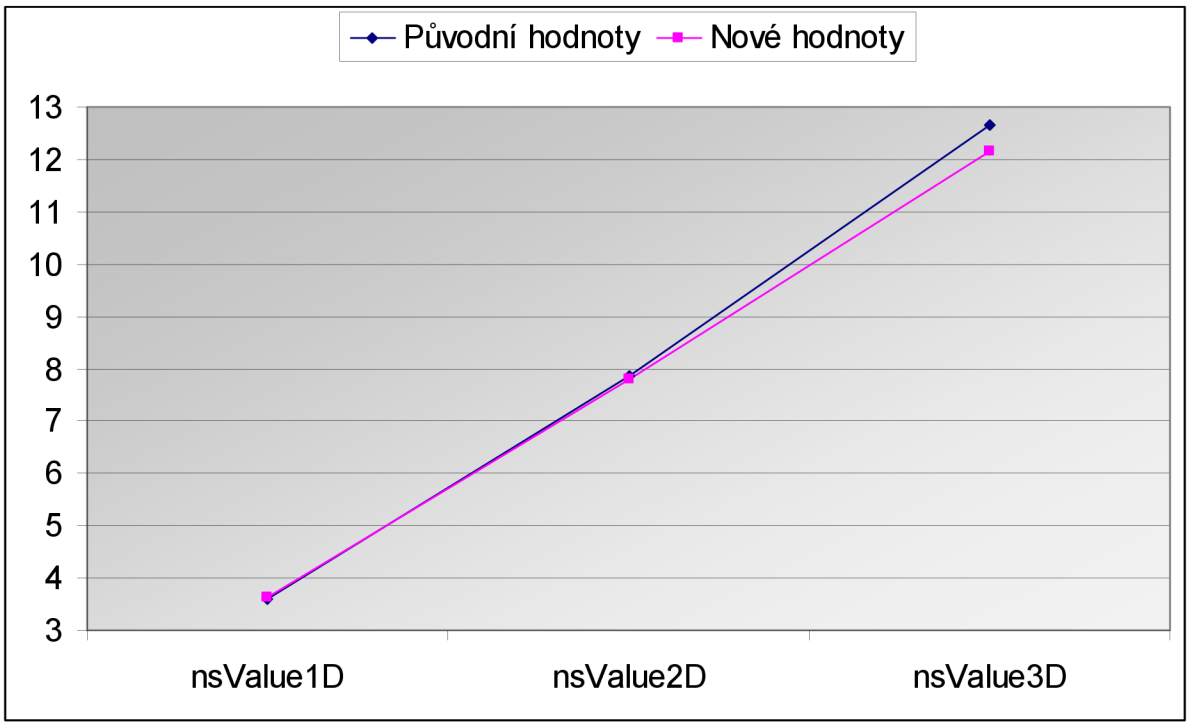
U dvojdimenzionální verze se zjistí index z prvního parametru, a k hodnotě na tomto indexu se přičte druhý parametr a opět se zjistí hodnota z permutačního pole na tomto novém indexu. Jenže přičtením druhého parametru může být hodnota vyšší než 255 a tím by došlo k přístupu mimo pole. Proto permutační pole zvětším na dvojnásobnou velikost a nebude nutné tuto hodnotu upravovat.

Ve 3D by se muselo toto pole zvětšit ještě jednou, ale výhodnější bude provést zvětšení pole hodnot na dvojnásobnou velikost a po přičtení třetího parametru již nepřistupovat do permutačního pole, ale přímo do pole hodnot.

Naměřené hodnoty:

	nsValue1D	nsValue2D	nsValue3D
Původní hodnoty	3,594 ms	7,850 ms	12,657 ms
Nové hodnoty	3,618 ms	7,806 ms	12,157 ms

Tabulka 22.: Hodnoty před a po optimalizaci funkce nsValue



Graf 4.: Srovnání doby výpočtu pro optimalizovanou verzi funkce

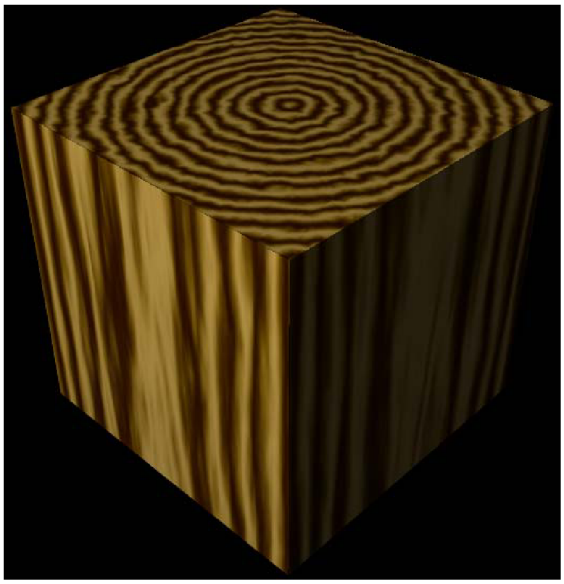
U 1D verze zůstaly výsledky stejné, což se dalo očekávat, protože v 1D nedošlo k žádným změnám. Je zde velmi malé zhoršení, které může být způsobeno kopírováním hodnot do druhé poloviny polí.

Ve 2D je výsledek prakticky také stejný. Došlo k minimálnímu zlepšení v podobě 44 mikrosekund, neboli 0,56%. Velmi zanedbatelné zlepšení.

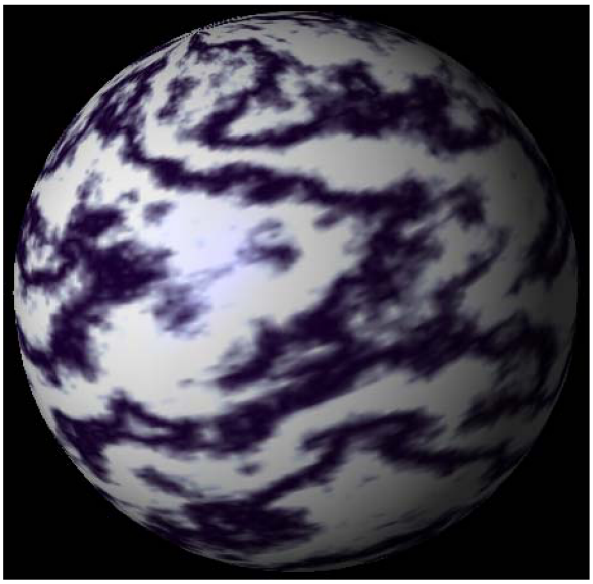
U trojdimenzionální funkce se jedná o zlepšení cca o 500 mikrosekund při počtu zavolání 65536. Toto není nikterak velké zlepšení, 3,95%. Nyní vezmu do úvahy změnu paměťové náročnosti. Permutační pole je datového typu unsigned char o velikosti 256 prvků. To činí 256B paměti, takže nové pole zabere 512B paměti. Pole hodnot je datového typu double, který je dvakrát větší a zabere 4kB paměti. Tyto hodnoty jsou v dnešní době velice malé, takže se zatím budu přiklánět k nové verzi řešení. Záměrně říkám zatím, protože ještě budou provedeny 3D vizuální testy, které mohou odhalit pravidelné vzory při vynechání výpočtu indexů z permutačního pole.

Z tohoto testu hlavně vyplynula velmi vysoká rychlost bitové operace *and* a každá operace *modulo*, která je stále obsažena ve funkcích pro naplnění polí daty, je nahrazena bitovým *and*. Po provedení několika testů jsou výsledky stejné. Zlepšení nastalo v desítkách mikrosekund, což je dáno tím, že se pole na začátku naplní jen jednou.

6.7 Vizuální testy ve 3D



Obrázek 27.: Textura dřeva generovaná pomocí knihovny



Obrázek 28.: Textura mramoru nanesená na kouli vypočítaná pomocí knihovny

Časová optimalizace je nutná, ale nemůže být na úkor vizuálních výsledků. V předchozích kapitolách jsem u Lattice noise a Value noise poznamenal, že záleží na vizuálních výsledcích ve 3D.

Proto byla v hlavním programu naprogramována nová funkce pro testování 3D funkcí, ve které je plněno trojrozměrné pole o velikosti 128x128x128 pomocí 3D šumových funkcí. To probíhá ve třech vnořených cyklech for, kde se postupně měnily hodnoty parametrů funkce. Výsledné pole je následně použito pro vytvoření 3D textury a její „namapování“ na krychli.

Při testování jsem také narazil na dosti omezující limit, kdy maximální velikost jedné strany 3D textury je maximálně 128 texelů. Samozřejmě lze použít i větší rozměry, protože velikost je závislá na velikosti RAM paměti a VRAM paměti na grafické kartě, ale z praktického hlediska je pak program nepoužitelný. Textura totiž při této velikosti zabere 256*256*256*3*4 = 201 MB za použití datového typu float. Tato hodnota je skutečně příliš vysoká a program je neefektivní.

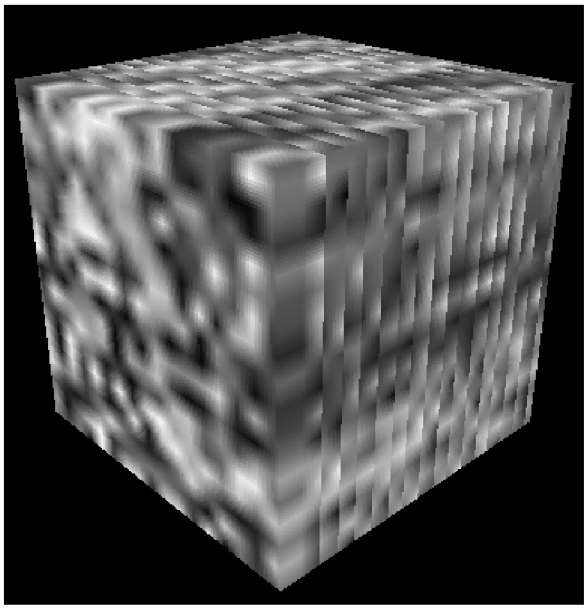
Na začátku kapitoly jsou uvedeny dva příklady 3D procedurálních textur a určitě by vizuální testy měly být provedeny právě generováním textur materiálů. Následující testy jsou ovšem provedeny se základními funkcemi a prostým zobrazením šumu. Toto zobrazení má dvě opodstatnění.

Když jsou generovány procedurální textury, tak se většinou jedná o fraktální sumu nebo turbulenci a pokud je měřena jejich časová náročnost, tak je mnohem vyšší. Samotný výpočet textury obsahuje další výpočty a trvá také určitou dobu, takže časové výsledky by nebyly adekvátní.

Dalším důvodem jsou možná opakování. Pokud je generována textura, tak její výpočet je nastaven tak, že pravděpodobnost opakování vzoru je velmi malá. Proto vizuální testy jsou provedeny se základními funkcemi a zobrazují se přímo hodnoty, které funkce vrací. U funkce pro Latttice noise to bylo nutností. Bylo nutné vidět, jak se funkce ve 3D chová a jaké vrací hodnoty.

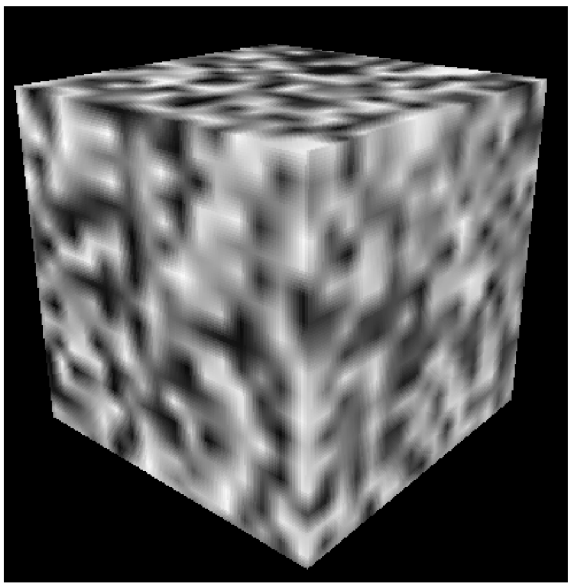
6.7.1 Lattice noise

U trojdimenzionální verze Lattice noise jsem se nejdříve pokoušel vyhnout aplikaci trojrozměrného pole, protože zabírá velmi hodně paměti. Snažil jsem se to vymyslet tak, že by bylo použito dvojrozměrné pole a za použití třetího parametru funkce bych pomocí jednoduchého výpočtu dopočítával další čtyři hodnoty pro trilineární interpolaci. Ve 2D testech se potvrdilo, že dopočítáváním indexů se časová náročnost nikterak výrazně nezhoršila (křivka v grafu by měla mít téměř lineární průběh). Pro 3D testy činila 316,42 ms, kterou samozřejmě nelze srovnávat s předchozími hodnotami, kde se plnilo dvojrozměrné pole. Vizuální výsledky bohužel už nepřinesly dobré výsledky. Při dopočítávání hodnot vždy došlo k dosažení konce pole a to znamenalo viditelný přechod v textuře. Zkoušel jsem více možností, ale výsledek byl vždy stejný.



Obrázek 29.: Lattice noise 3D s dopočítáváním indexů

Nakonec jsem tedy použil trojrozměrné pole, které jsem omezil na velikost 32x32x32. Tato velikost je maximální přípustnou velikostí, protože kdybych použil pole o velikosti 64x64x64 a datový typ double, pak by pole „spolknulo“ 2 MB paměti. To není zrovna málo. Tím, že bylo použito takto malé pole, bude určitě při větších texturách vidět opakovatelnost, ale na druhou stranu funkce zůstala stále rychlejší než Value noise s časem 316,26 ms (srovnání všech časů je na konci kapitoly).



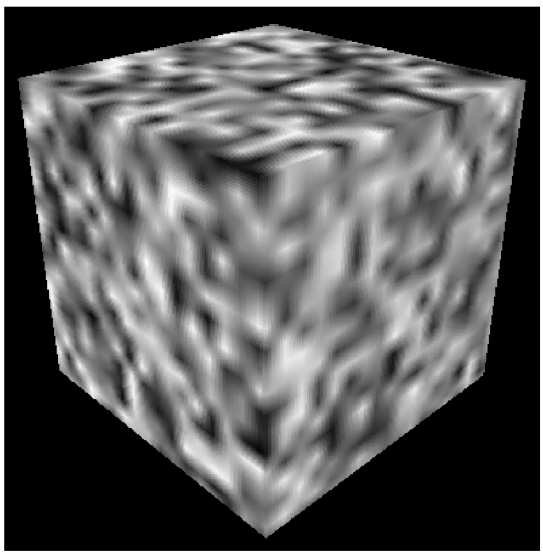
Obrázek 30.: Lattice noise 3D s trojrozměrným polem

Při takto malé textuře nedošlo ještě k opakování, které se na obrázku nenachází. Z časových výsledků je také vidět, že přístupem do trojrozměrného pole se časová náročnost nezvyšuje.

6.7.2 Value noise

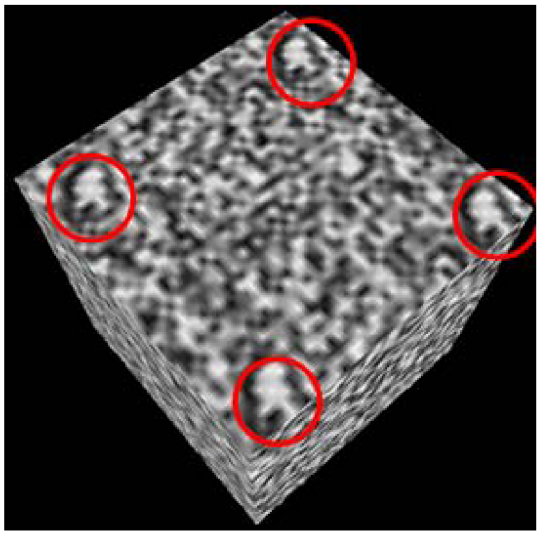
V předchozí kapitole je možné se dočíst o změně velikosti permutačního pole a pole s hodnotami. Bylo také zmíněno, že tato změna by mohla mít negativní vliv na vizuální výsledky. To se ovšem nepotvrdilo a výsledky byly stejné jako v případě, kdy se všechny indexy přepočítávaly přes permutační pole o velikosti 256.

Časový rozdíl ve 3D testu je oproti 2D testu výraznější s hodnotami 382,96 ms (starší verze) a 370,47 ms (nová verze). Rozdíl činí 12,49 milisekund, což odpovídá 3,26%. Závěrem lze říci, že je určitě výhodné použít tuto novou verzi funkce.



Obrázek 31.: Value noise 3D optimalizovaná verze

Nyní lze ještě porovnat Lattice noise s Value noise. Tak jak jsem je srovnával i v předchozích dimenzích, tak je dobré je porovnat i nyní. Vizuální výsledky jsou naprosto stejné, což se dalo očekávat. Funkce pro Lattice noise tuto výhodu ovšem ztratí, pokud parametry funkce překročí hodnotu 32. V tuto chvíli dojde k opakování hodnot tak jako na obrázku 32. K takovému opakování by samozřejmě došlo i v případě Value noise, ale až po překročení hodnoty 256.

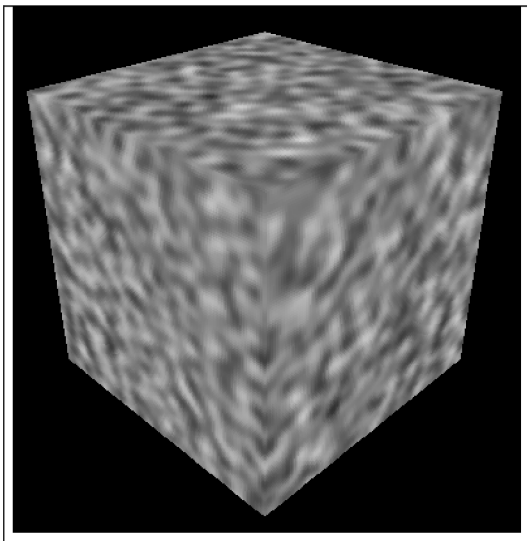
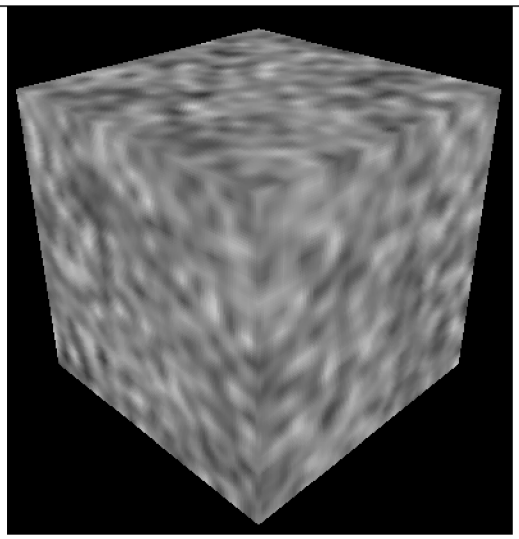


Obrázek 32.: Opakovatelnost u funkce nsLattice3D

Vizuální nedostatek je zase vyvážen menší časovou náročností. Rozdíl mezi funkcemi je 54,21 ms, vyjádřeno v procentech 14,63%. To není zrovna malá hodnota, takže se funkce vyplatí.

U Perlinova a Value-Gradientního šumu nedošlo ke změnám, které by mohly ovlivnit vzhled ve 3D, ale přesto výsledky uvedu a to i kvůli časové náročnosti.

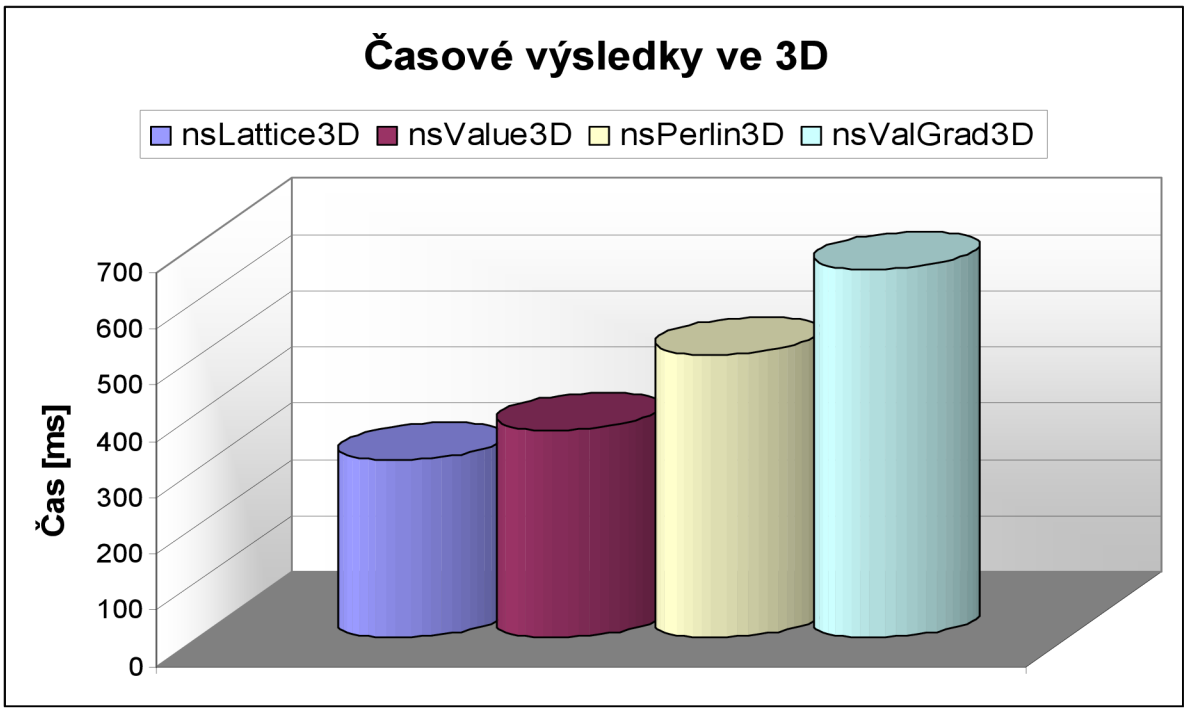
6.7.3 Perlin noise a Value-Gradient noise

	
Perlin noise Časová náročnost: 503,723 ms	Value-Gradient noise Časová náročnost: 656,723 ms

Tabulka 23.: Vizuální a časové výsledky pro Perlin a Value-Gradient noise

Ve 3D je vidět velký rozdíl v časové náročnosti. Jelikož byly funkce volány 2097152krát, aby se naplnilo pole 128*128*128 položek, projevila se jejich časová náročnost a vznikly velké rozdíly. Mezi Lattice a Value noise je rozdíl 14,63%. Mezi Value noise a Perlin noise je rozdíl 133,253 milisekund a to je 26,45%. Toto určitě není zanedbatelná hodnota.

Porovnám-li teď nejrychlejší Lattice noise s Value-Gradient noise, pak dostanu následující hodnoty: rozdíl 0,34 sekund, v procentech 51,84%. Vše ukazuje závěrečný graf.



Graf 5.: Naměřené hodnoty doby výpočtu ve 3D testech

Jednoznačně lze za vítěze označit Lattice noise, ale vše záleží na tom, k jakému účelu šum generujeme. V knihovně nakonec zůstanou obě implementace Lattice Value Noise, protože ne každý šum se hodí pro všechny typy textur materiálů nebo specifický typ použití. O tomto tématu je taky jedna z dalších kapitol.

6.8 Float vs. Double

Funkce v knihovně jsou naprogramovány tak, že každá proměnná pro uchování desetinných čísel je datového typu ***double***. Zde by se mohla položit otázka, zda by nebylo výhodnější použít datový typ ***float***.

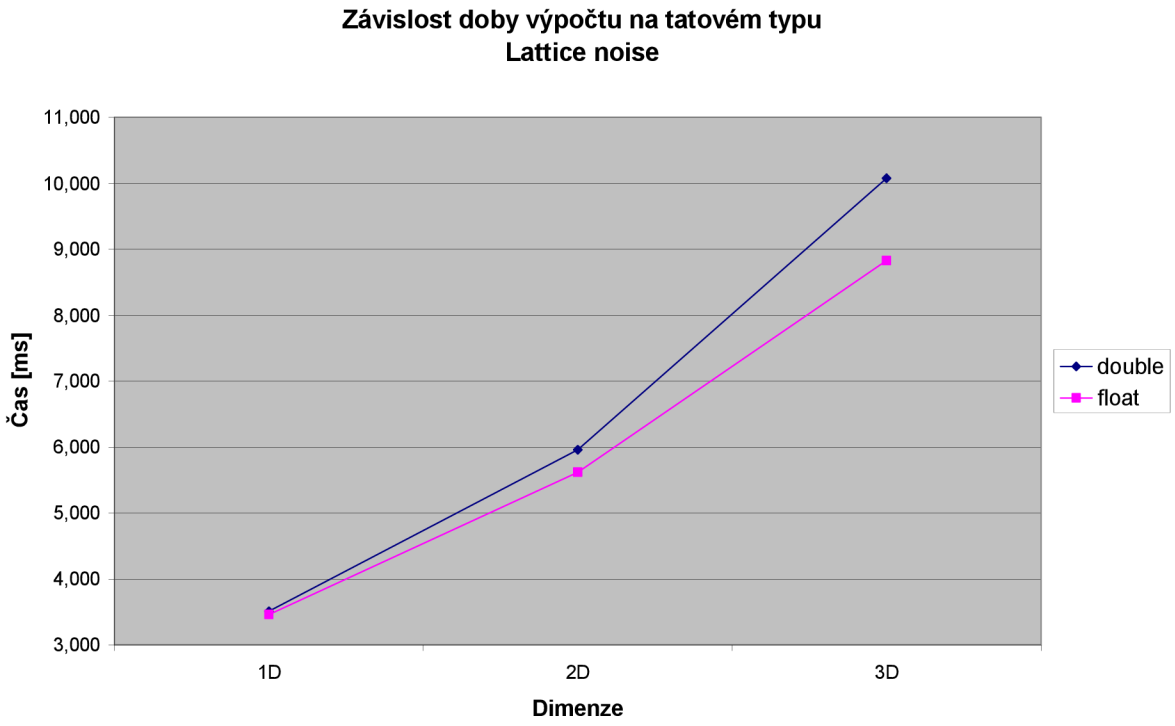
Float má velikost 32 bitů, což jsou čtyři bajty. Double pracuje s dvojnásobnou přesností a tedy i s dvojnásobnou velikostí. Podle předpokladu by tedy výpočet ve float měl být rychlejší asi o 15%. Je to jen předpoklad, který je ovšem nutné dokázat. Proto byly funkce v knihovně přetypovány (přetíženy) na float a provedeny testy s následujícími výsledky. Testovány přitom byly jen základní šumové metody, protože fraktální suma a turbulence vycházejí z těchto funkcí a tak bylo zbytečné je testovat. Testy proběhly na stejné testovací sestavě, Intel Celeron 2,88 GHz s 512 MB RAM.

6.8.1 Lattice noise

U Lattice noise došlo k nárůstu výkonu postupně s počtem dimenzí, tak jak je vidět v následující tabulce a grafu.

Lattice noise	***double*** [ms]	***float*** [ms]	**rozdíl** [ms]	**rozdíl** [%]	**Celkový rozdíl [%]**
1D	3,509	3,460	0,049	1,41%	**6,50%**
2D	5,956	5,619	0,337	5,67%	
3D	10,079	8,828	1,251	12,41%	

Tabulka 24.: Naměřené hodnoty pro Lattice noise



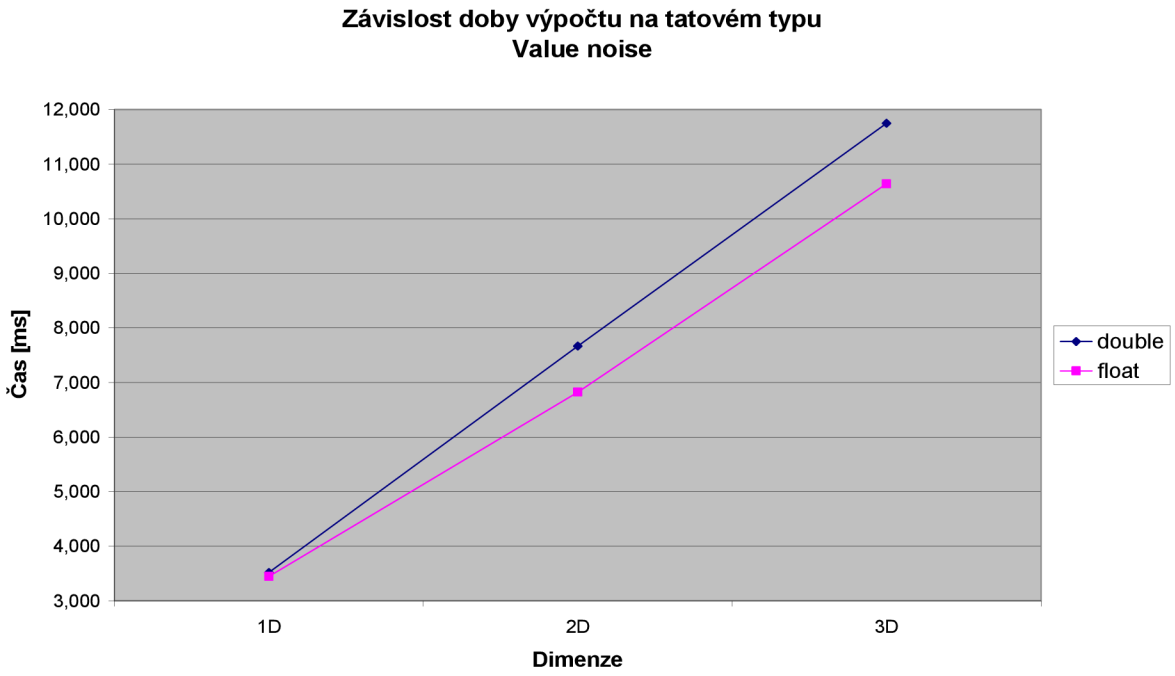
Graf 6.: Závislost doby výpočtu na datovém typu

6.8.2 Value Noise

Value noise je trochu odlišný od Lattice noise. U tohoto šumu došlo při použití typu float k největšímu nárůstu výkonu a je zvláštní, že největší nárůst zaznamenala 2D verze.

Value noise	***double*** [ms]	***float*** [ms]	**rozdíl** [ms]	**rozdíl** [%]	**Celkový rozdíl [%]**
1D	3,516	3,447	0,069	1,96%	**7,47%**
2D	7,668	6,825	0,844	11,00%	
3D	11,750	10,640	1,110	9,45%	

Tabulka 25.: Naměřené hodnoty pro Value noise



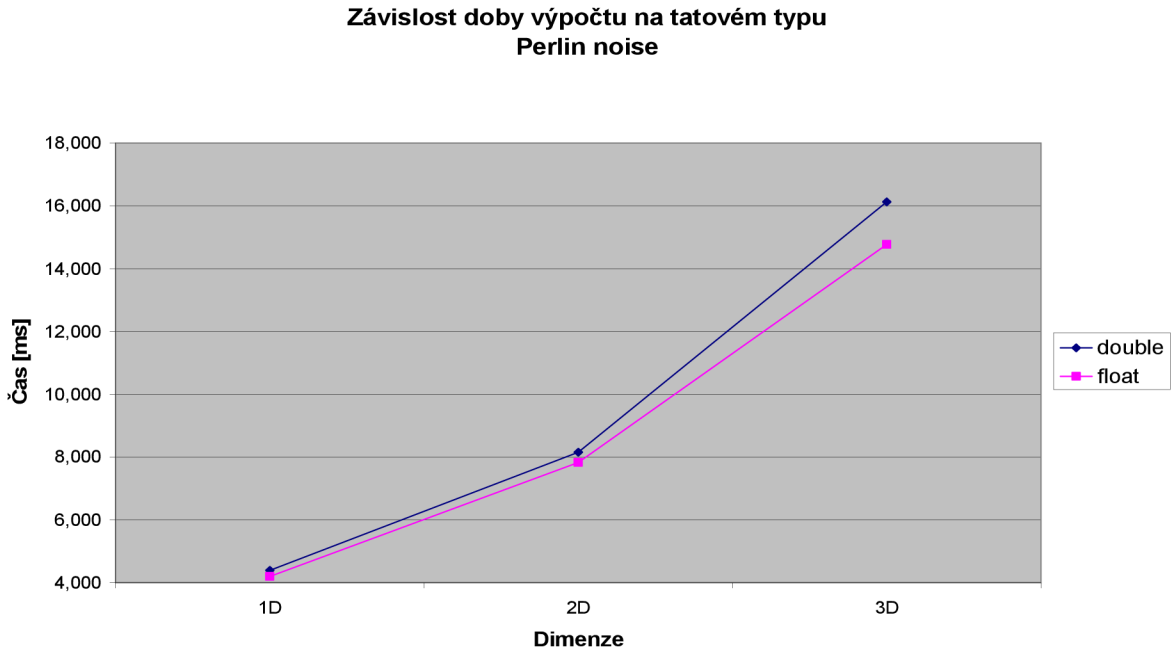
Graf 7.: Závislost doby výpočtu na datovém typu pro Value noise

6.8.3 Perlin noise

Tento typ šumu má nejmenší celkový nárůst výkonu, ale zlepšení u 1D a 2D je stejné.

Perlin noise	***double*** [ms]	***float*** [ms]	**rozdíl** [ms]	**rozdíl** [%]	**Celkový rozdíl [%]**
1D	4,391	4,200	0,191	4,34%	**5,55%**
2D	8,149	7,831	0,318	3,90%	
3D	16,125	14,767	1,358	8,42%	

Tabulka 26.: Naměřené hodnoty pro Perlin noise



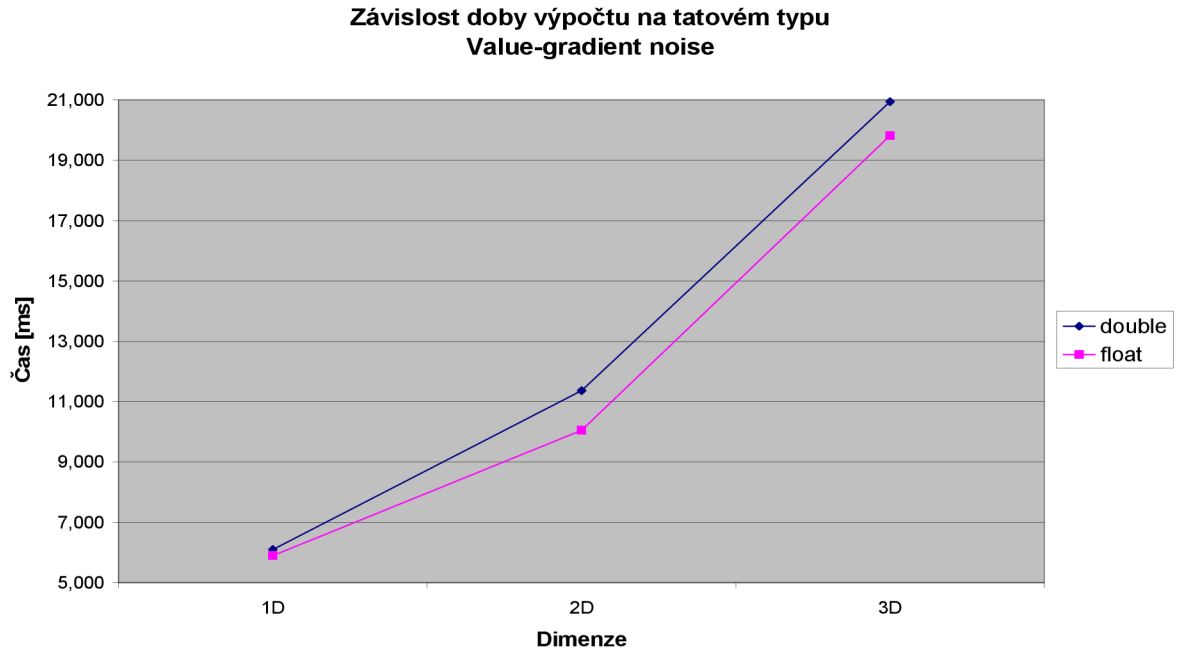
Graf 8.: Závislost doby výpočtu na datovém typu pro Perlin noise

6.8.4 Value-Gradient noise

Value-Gradient noise je kombinací Value noise a Perlin noise a změnou datového typu byly naměřeny tyto poněkud zvláštní výsledky.

Value-Gradient noise	***double*** [ms]	***float*** [ms]	**rozdíl** [ms]	**rozdíl** [%]	**Celkový rozdíl [%]**
1D	6,094	5,897	0,197	3,23%	**6,71%**
2D	11,360	10,051	1,309	11,52%	
3D	20,939	19,813	1,126	5,38%	

Tabulka 27.: Naměřené hodnoty pro Value-Gradient noise



Graf 9.: Doba výpočtu pro Value-Gradient noise

6.8.5 Celkové srovnání

Celkem	rozdíl v dimenzích [%]	celkový rozdíl (všechny metody) [%]
1D	2,73%	**6,56%**
2D	6,25%	
3D	8,91%	

Tabulka 28.: Celkové naměřené výsledky v procentech

Jak je z tabulek vidět, předpoklad byl pravdivý, ale zrychlení není 15%, nicméně hodnota okolo 10% u některých funkcí určitě taky není zanedbatelná. Nejdříve, než provedu závěrečné hodnocení výsledků, tak bych se chtěl nad nimi pozastavit.

Naměřené výsledky mě tak trochu i překvapily. Začal jsem pátrat po tom, proč to tak je, že třeba Value noise má takové velké zlepšení ve 2D a 3D, Perlin noise toto zlepšení má minimální a ve Value-Gradient noise je hlavní zlepšení ve 2D, když je to prakticky kombinace Value a Perlin noise. Naprosto přesnou odpověď jsem nebyl schopen najít, ale z teoretických znalostí a různých diskuzí na internetu jsem dospěl k následujícímu závěru.

Výsledky jsou ovlivněny hlavně architekturou procesoru, velikostí cache paměti a rychlostí datových přenosů mezi procesorem a pamětí RAM. Další a malý vliv má také samozřejmě překladač a optimalizace překladače.

Dnešní moderní procesory jsou stavěné na 64 bitové architektuře, ale i starší procesory na 32 bitové architektuře jsou postavené tak, aby počítaly rychle i datovým typem double. Procesor Pentium 4 má 128 bitovou SIMD architekturu a na tomto procesoru nepřináší použití datového typu float žádné zlepšení. Po naprogramování knihovny do maker (bude vysvětleno později) jsem provedl testy s typem float a double na notebooku s procesorem Intel PentiumM III 1GHz s 512MB RAM. Výsledky pro oba typy byly stejné a dokonce datový typ float byl o 1% pomalejší. Tento procesor má velikost cache 512kB.

Starší procesory x86 dokáží provést výpočet pro double a float stejnou dobu, protože počítají s 80ti bity a na jeden takt „natáhnou“ z paměti 4B. Float se ale do cache paměti dostane na jeden takt, ale double potřebuje dva takty. Takže teoreticky by float měl být 2x rychlejší. Datový typ double v tomto případě ztrácí výkon při paměťových operacích.

Velikost cache paměti dokáže také hodně ovlivnit výkon. Pokud se data nevejdou do této malé paměti, pak se právě musí neustále přenášet z hlavní paměti a tak dochází neustále k I/O operacím. Toto by vedlo právě ke zlepšení výkonu na některých architekturách při použití typu float. Jinde se lze zase dočíst, že float je pomalejší, protože dochází k zaokrouhlování.

S jistotou lze jen říci, že použitím datového typu float nebude potřeba tolik paměti, ale výkon bude vždy závislý na použité architektuře procesoru, typu instrukcí, velikosti cache paměti a rychlostí datových přenosů mezi cache pamětí a RAM.

Jelikož jsou tyto informace z různých zdrojů od uživatelů a z teoretických znalostí, tak nejsou ničím podložené a mohou být vyvratitelné a nedůvěryhodné. Proto jsem se rozhodl provést nové testy pro double a float na jiných počítačích a potvrdit variabilní výsledky závislé na použité testovací sestavě. Kapitola s těmito testy je z chronologických důvodů uvedena později.

Nastává tedy otázka, zda funkce naprogramovat tak, aby pracovaly jen s datovým typem float nebo tak, aby bylo možné použít jak float tak i datový typ double nebo jen double. Pokud by funkce byly jen datového typu float, pak by byla ztracena přesnost, kterou nabízí double a to by přeci byla škoda, protože v některých případech je potřeba i přesnost. Kdyby byly jen typu double, pak by na některých platformách float mohl být zase rychlejší. Je to dost sporné, ale rozhodl jsem se funkce naprogramovat tak, aby byly schopny počítat s datovým typem float i double a přitom bude použita i další optimalizace. Viz následující kapitola.

6.9 Typová nezávislost

V předchozí kapitole jsem dospěl k názoru, že by bylo dobré funkce naprogramovat tak, aby byly schopny počítat s datovým typem float i double. Nejdříve tedy představím několik způsobů, jak toho docílit. Některé metody nabízí jazyk C a jiné vyspělejší jazyk C++.

6.9.1 Makra

Pomocí maker se dá dosáhnout datové nezávislosti a také velkého zrychlení výpočtu. Makro je totiž zpracováno preprocesorem před samotným kompilátorem. Tam, kde je makro voláno, tam preprocesor nahradí toto volání konkrétním textem makra přímo do kódu programu. Makra většinou slouží k odstranění režie volání funkce. Je-li totiž funkce krátká s rychlým výpočtem, pak se ji vyplatí napsat jako makro, protože se tím odstraní režie spojená s voláním funkce. Tato metoda má ovšem i nevýhodu. Jelikož se makro „rozbaluje“, tak se zvětšuje velikost programu. Nyní je tedy na programátorovi, zda si vybere kratší, ale pomalejší program nebo delší, ale rychlejší.

Jazyk C++ pak nabízí následující dvě možnosti typové nezávislosti (polymorfismus).

6.9.2 Přetížení funkcí

V C++ mohou dvě a více funkcí sdílet jedno jméno, pokud se liší počtem parametrů nebo jejich datovými typy anebo se liší obojí. Tyto funkce se pak nazývají přetížené. V praxi to vypadá tak, že se funkce naprogramuje ve více verzích se stejným jménem, ale s různým datovým typem nebo s jiným počtem parametrů a překladač automaticky vybere správnou funkci podle typu nebo počtu parametrů.

6.9.3 Generické funkce (šablony)

Taková funkce definuje obecnou sadu operací, které budou použity na rozmanité typy dat. Takovou funkci lze pak zavolat s různým datovým typem a překladač sám pak generuje správný kód dle typu dat. Určení typu zpracovávaných dat se funkci předává jako parametr. V podstatě lze říci, že v generické funkci je vytvořena funkce, která může přetížit sama sebe.

Každá z uvedených metod má svoje výhody a nevýhody. Tím, že se makro „rozbalí“ do kódu programu, tak nabízí zrychlení, ale zase zvětší velikost programu. Makro kód je také velmi špatně čitelný a ještě hůř se provádí ladění takového kódu.

Přetížení funkcí nabídne jednoduchou typovou nezávislost, neovlivní rychlost programu a nezvětší jeho velikost, ale zvětší se velikost knihovny.

Šablony pak nabídnou ještě jednodušší typovou nezávislost, nezvětší velikost programu ani knihovny, ale může dojít ke zpomalení programu způsobené vytvářením funkcí při volání.

6.9.4 Testy možných implementací

Na základě předchozího rozboru jsem provedl naprogramování několika funkcí do makra a do šablony a provedl několik testů. Přetížení funkcí jsem záměrně netestoval, protože tato metoda byla již testována při porovnání double a float, nicméně výsledky jsou porovnávány s časy klasických

funkcí. Testování probíhalo stejnou metodou jako předchozí testy ve 2D. Bylo plněno pole o velikosti 256x256 pomocí šumových metod implementované v makrech, šablonách a klasických funkcích. Všechny testy proběhly pro datový typ double.

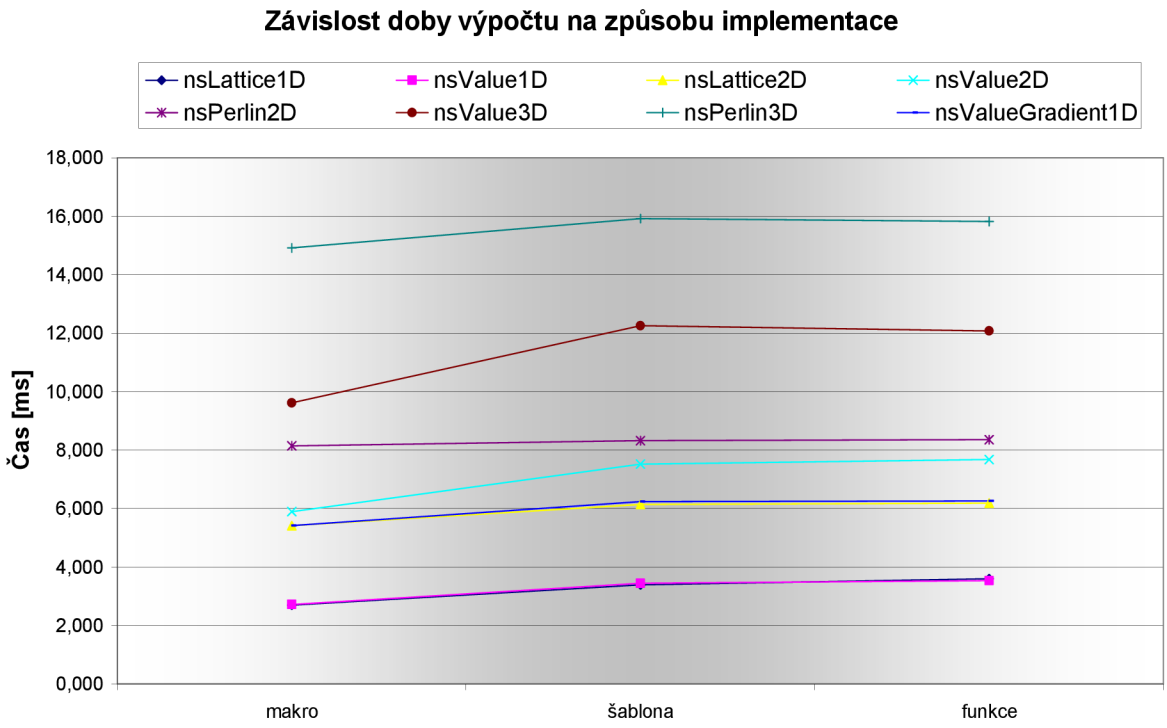
Účelem následujícího testu nebylo otestovat, zda je float výkonnější než double v jiných implementacích, ale test byl proveden, aby bylo rozhodnuto, jak budou metody v knihovně implementovány.

V tabulce naměřených hodnot nejvíce vyniká funkce Value3D a celkově tato metoda opět dosáhla nejlepších výsledků.

Funkce	Způsob implementace			Zlepšení
	makro	šablona	funkce	
nsLattice1D	2,690	3,389	3,598	20,63%
nsValue1D	2,718	3,442	3,536	21,04%
nsLattice2D	5,417	6,136	6,180	11,72%
nsValue2D	5,892	7,514	7,672	21,58%
nsPerlin2D	8,143	8,322	8,352	2,15%
nsValue3D	9,618	12,253	12,072	21,51%
nsPerlin3D	14,909	15,912	15,821	6,30%
nsValueGradient1D	5,414	6,233	6,263	13,13%

Tabulka 29.: Naměřené hodnoty při testech makra a šablony

Sloupec „Zlepšení“ udává procentuální zlepšení doby výpočtu makra oproti výpočtu pomocí šablon a ve sloupci „funkce“ jsou hodnoty funkcí, které již byly implementovány dříve. Jsou zde uvedeny pro celkové porovnání a pro srovnání s šablonami. Je vidět, že šablony byly někdy lepší a někdy horší, ale průměrně si časy odpovídaly. Nejlépe je to znázorněno v následujícím grafu.



Graf 10.: Doba výpočtu v závislosti na implementaci

Jednoznačně lze označit implementaci pomocí maker za nejlepší řešení. Tímto způsobem je dosaženo nejrychlejšího výpočtu a typové nezávislosti. Nevýhodou tohoto řešení je velikost výsledného programu. Knihovna bude kompletně přeprogramována do maker tak, že vlastní jádro výpočtu zůstane nezměněno, ale dojde pouze ke změně hlavičky, kde návratová hodnota je ve formě parametru. Také bude obsahovat datový typ, se kterým bude celý výpočet probíhat.

6.10 Opětovné testy double vs. float

V kapitole 6.8 se lze dočíst o rozdílu výpočtu mezi double a float, který je zřejmě způsoben rozdílnou architekturou a konfigurací testovacího počítače. Takové informace jsou založeny na základě internetových diskuzích a teoretických znalostí, které nejsou ověřeny a je nutné je podložit.

Záměrně jsem provedl opětovné testy pro porovnání double a float až po přeprogramování do maker, protože makra přinesly velké zlepšení a výsledná knihovna bude tvořena makro funkcemi. Když je volána klasická funkce, uloží se aktuální proměnné do zásobníku, předají se parametry, provede se výpočet funkce a po návratu z funkce se aktualizují hodnoty proměnných ze zásobníku. Jelikož se makra „rozbalí“ přímo do kódu, není potřeba zásobník a tak je ušetřen čas. V případě funkcí mohla mít tato procedura se zásobníkem vliv na rozdíl mezi double a float a proto byly opětovné testy provedeny až nyní.

Časové výsledky nebudou samozřejmě odpovídat naměřeným hodnotám v kapitole 6.8. To ale nevadí, protože nejde o celkovou dobu výpočtu, nýbrž o časový rozdíl ve výpočtu s datovým typem float a double na jednotlivých sestavách. To také ukáže, zda implementace pomocí makra má vliv.

6.10.1 Testovací sestavy

Pro jednoduchost byly jednotlivé sestavy očíslovány a v tabulkách jsou použity jejich čísla. Testovací sestavy mají následující konfigurace:

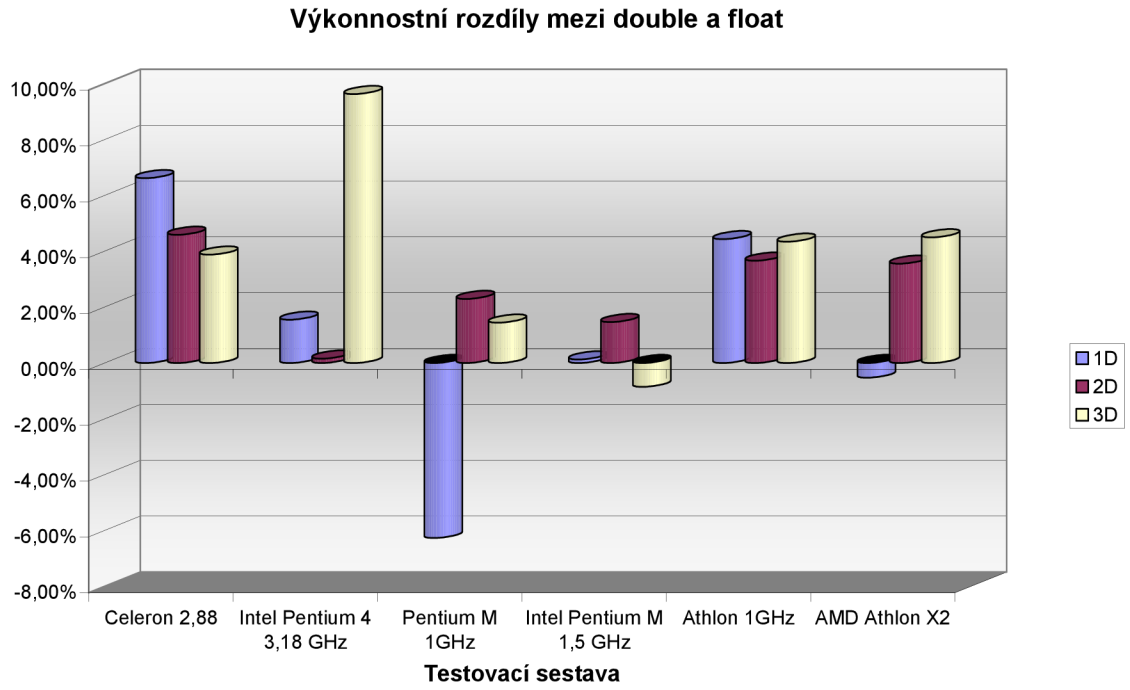
1. **Intel Celeron 2,88GHz** s 256kB cache, 2 x 256 MB DDR RAM s frekvencí 400MHz
   Jedná se o stálou testovací sestavu.
2. **Intel Pentium 4 3,18GHz** s 1MB cache, 2 x 256 MB DDR RAM s frekvencí 366MHz
   Měl jsem možnost vyzkoušet „čistý" Intel procesor s jádrem Prescott. Takže jsem jej vyměnil a provedl testy. Jelikož byla jiná frekvence sběrnice, změnila se také frekvence pamětí na 366MHz.
3. **Intel Pentium III-M 1GHz** s 512kB cache, 512 MB SO-DIMM RAM s frekvencí 266MHz
   Notebook s mobilním procesorem a technologií SpeedStep.
4. **Intel Pentium M 1,5GHz** s 1MB cache, 512 MB DDR RAM s frekvencí 266MHz
   Notebook s mobilním procesorem
5. **AMD Athlon 1GHz** s 256kB cache, 512 MB DDR s frekvencí 400MHz
   Stolní počítač se starším Athlonem s jádrem Thunderbird
6. **AMD Athlon 64 X2 3800+** (2GHz) s 512kB cache v každém jádře, 2GB DDR2 RAM
   Stolní počítač s dvoujádrovým procesorem. Test ovšem probíhal pouze na jednom jádře.

Údaje o procesorech a pamětech byly zjištěny pomocí programu CPU-Z, který je volně dostupný na internetu (http://www.cpuid.com/cpuz.php) a jedná se o freeware.

6.10.2 Lattice noise

Sestava	1D			2D			3D		
	float	double	rozdíl	float	double	rozdíl	float	double	rozdíl
1	2,457	2,631	6,62%	4,685	4,911	4,59%	9,165	9,536	3,88%
2	2,207	2,242	1,55%	4,379	4,385	0,15%	7,381	8,167	9,63%
3	8,199	7,716	-6,26%	15,935	16,310	2,30%	25,925	26,305	1,44%
4	5,790	5,797	0,13%	11,431	11,601	1,47%	18,444	18,289	-0,85%
5	6,594	6,900	4,43%	12,341	12,811	3,67%	21,536	22,514	4,35%
6	2,811	2,796	-0,53%	5,773	5,987	3,56%	9,984	10,455	4,50%

Tabulka 30.: Naměřené hodnoty pro Lattice noise



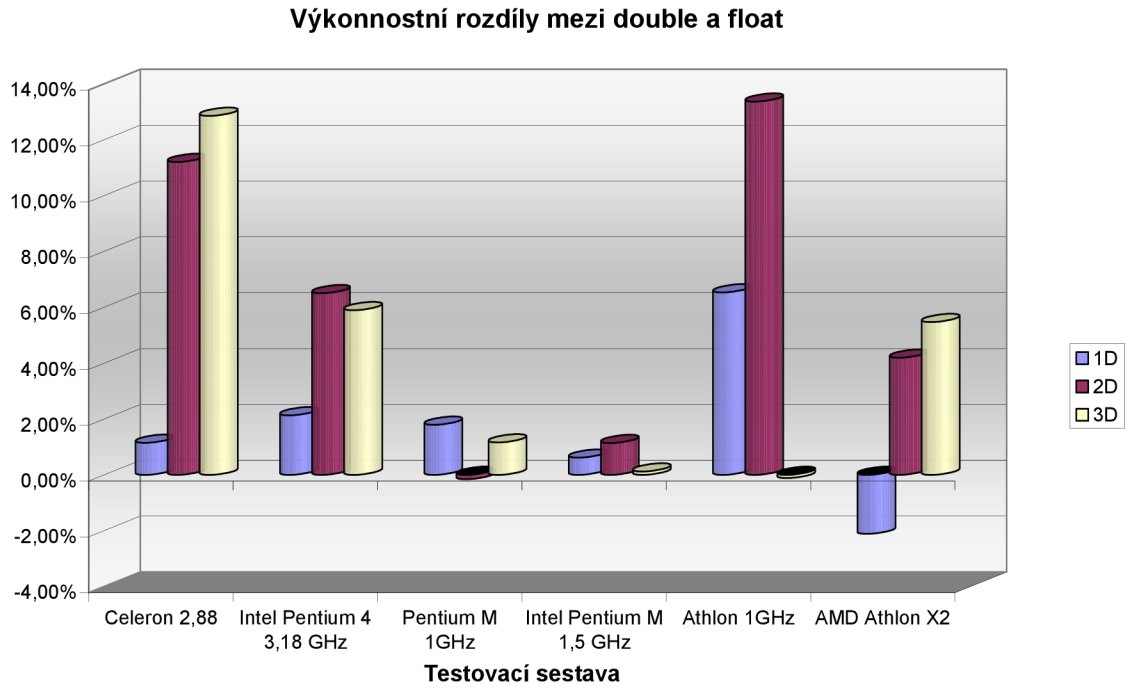
Graf 11.: Rozdíly výpočtu mezi double a float u jednotlivých sestav

Z tabulky a grafu je vidět, že každý počítač se choval trochu jinak. Oproti předchozím testům u Celeronu nyní s počtem dimenzí rozdíl klesá. Pentium 4 dosáhlo velkého nárůstu ve 3D verzi. Notebook s procesorem Pentium III-M 1GHz má výpočet v datovém typu float horší o 6% oproti double. Výsledky druhého notebooku jsou téměř stejné pro oba datové typy a Athlon 1GHz dosahuje v datovém typu float ve všech dimenzích přibližně stejného zrychlení.

6.10.3 Value noise

Sestava	1D			2D			3D		
	float	double	rozdíl	float	double	rozdíl	float	double	rozdíl
1	2,539	2,568	1,14%	5,037	5,672	11,20%	9,003	10,330	12,85%
2	2,286	2,335	2,13%	4,796	5,130	6,50%	7,911	8,406	5,89%
3	8,311	8,462	1,79%	17,068	17,041	-0,16%	26,134	26,441	1,16%
4	5,921	5,958	0,62%	12,329	12,471	1,14%	19,121	19,143	0,12%
5	7,102	7,599	6,53%	15,910	18,364	13,37%	24,663	24,634	-0,12%
6	3,053	2,990	-2,11%	7,319	7,640	4,19%	11,446	12,108	5,47%

Tabulka 31.: Hodnoty pro Value noise



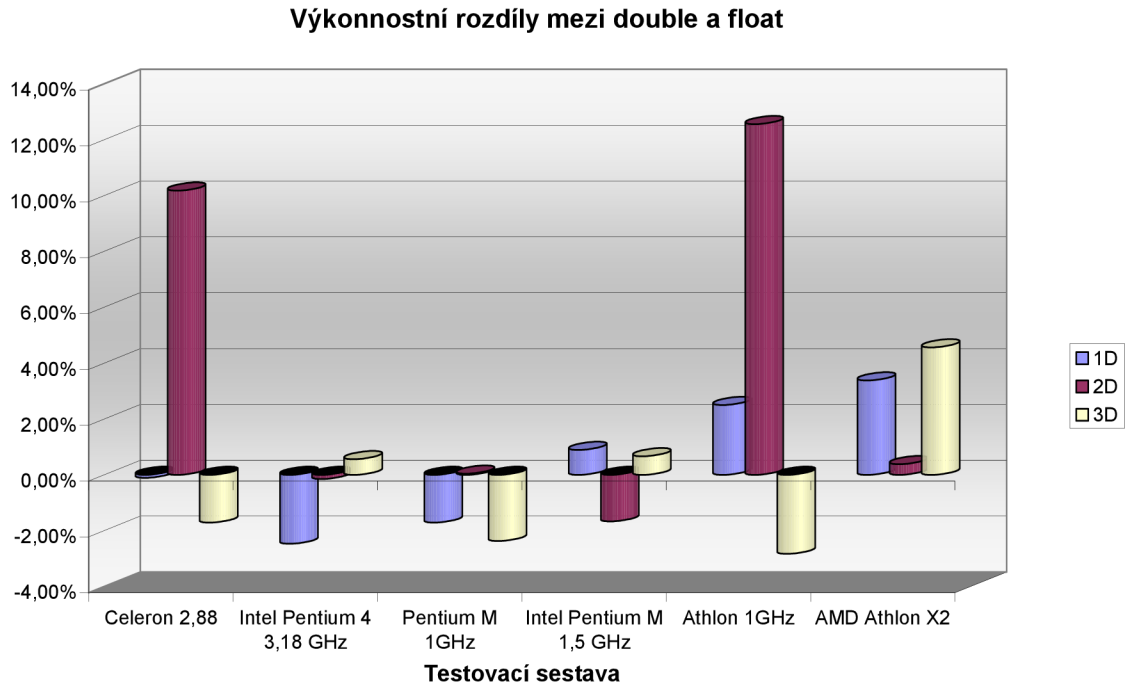
Graf 12.: Výkonnostní rozdíly mezi double a float u jednotlivých sestav

U Value noise je situace zase jiná. Zde dosáhl Athlon 1GHz největšího nárůstu výkonu pro float a také Celeron dosáhnul stejných výsledků jako v kapitole 6.8. Oba notebooky se držely malého zlepšení výkonu ve šech dimenzích. Athlon X2 prohloubil lepsi výkonnost pro double v 1D a ve 2D a 3D zase zlepšil výkon pro float.

6.10.4 Perlin noise

Sestava	1D			2D			3D		
	float	double	rozdíl	float	double	rozdíl	float	double	rozdíl
1	3,350	3,347	-0,11%	8,099	9,016	10,17%	14,192	13,953	-1,71%
2	3,575	3,489	-2,47%	6,921	6,910	-0,15%	13,171	13,245	0,56%
3	9,245	9,089	-1,71%	20,479	20,487	0,04%	37,257	36,395	-2,37%
4	6,739	6,799	0,89%	14,783	14,540	-1,67%	25,199	25,367	0,66%
5	8,140	8,349	2,50%	19,238	22,000	12,55%	36,419	35,416	-2,83%
6	3,747	3,878	3,38%	8,981	9,015	0,38%	17,736	18,583	4,56%

Tabulka 32.: Hodnoty naměřené pro Perlin noise



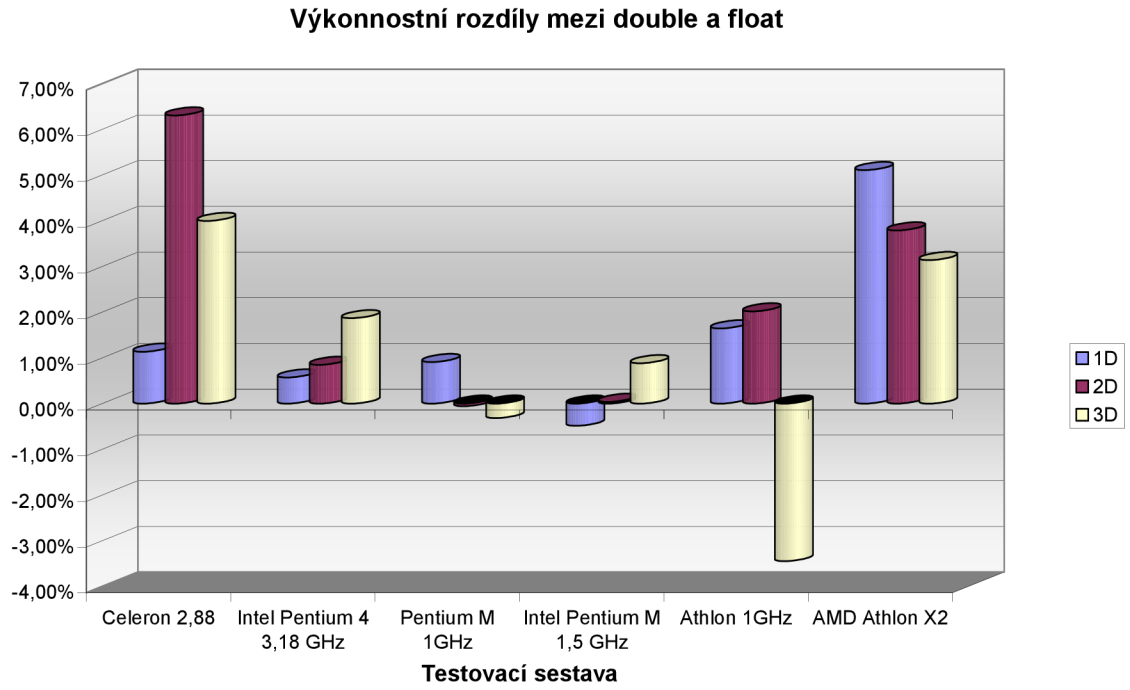
Graf 13.: Rozdíly mezi double a float

U Perlinova šumu je výsledek obrácený. Celkově dopadly testy pro float hůř než pro double. Je zajímavé, že oba procesory s 256 kB cache paměti dosáhly vysokého zlepšení výkonu ve 2D verzi. Další procesor, který dosáhl zlepšení ve float byl Athlon X2. Ostatní sestavy byly rychlejší, když počítaly s datovým typem double.

6.10.5 Value-Gradient

Sestava	1D			2D			3D		
	float	double	rozdíl	float	double	rozdíl	float	double	rozdíl
1	4,770	4,825	1,14%	10,395	11,094	6,31%	18,832	19,617	4,00%
2	4,907	4,936	0,57%	9,031	9,108	0,85%	17,598	17,934	1,87%
3	10,853	10,952	0,91%	23,604	23,592	-0,05%	43,872	43,735	-0,31%
4	7,793	7,756	-0,48%	16,801	16,808	0,04%	29,824	30,090	0,88%
5	10,074	10,243	1,65%	22,423	22,885	2,02%	43,929	42,466	-3,44%
6	4,456	4,696	5,11%	11,180	11,621	3,79%	21,959	22,672	3,14%

Tabulka 33.: Naměřené hodnoty pro Value-Gradient noise



Graf 14.: Rozdíly výpočtu mezi double a float u jednotlivých sestav

U tohoto šumu nejvíce vynikal opět Celeron, který měl stejné tendence zvýšení výkonu tak, jako u měření v kapitole 6.8. Slušného nárůstu výkonu ve výpočtu ve float dosáhl také Athlon X2. Nejhůře skončil Athlon 1GHz, který lépe počítá 3D verzi v double.

Naměřené výsledky jsou hodně proměnlivé a u některých metod se podstatně liší. Informace o tom, že záleží na architektuře, pamětech a frekvenci se tedy potvrdily. Vždyť už jenom samotnou výměnou Celeronu za „čisté“ Pentium a změnou frekvence je dosaženo úplně jiných rozdílů mezi výpočty ve float a double. Přitom zbytek sestavy zůstal nezměněn.

Přeprogramování do maker mělo a má určitě také vliv na výsledné rozdíly ve výpočtu v double a float, ale i po těchto testech lze jednoznačně říci, že se typová nezávislost u některých funkcí vyplatí.

7 Rozšíření knihovny

Ještě než byla knihovna přeprogramována do maker, byly naprogramovány další klasické funkce, které by rozšířily knihovnu o další možnosti využití šumu.

Samotné šumové funkce vrací jen jedno číslo float nebo double. Co kdyby ale vracely rovnou pole hodnot nebo vektory? Muže dojít ke zrychlení výpočtu nebo jiných výhod.

7.1 Funkce vracející jednorozměrné pole

V některých případech by mohlo být výhodnější, kdyby funkce vracely jednorozměrné pole hodnot. Pole se vytvoří v základním programu před zavoláním funkce a pak se jen předá ukazatel jako parametr na toto pole a velikost pole.

V tomto řešení by mohla být výhoda v rychlosti, protože funkce je volána přímo v knihovně. Proto byla naprogramována pro tři šumové funkce nsLattice1D, nsValue2D a nsPerlin3D a časové výsledky porovnány s časy, kdy se pole 256x256 plnilo v hlavním programu.

Funkce	Návratová hodnota		Zlepšení
	číslo	pole	
nsLattice1D	3,631	3,594	1,01%
nsValue2D	7,742	7,329	5,34%
nsPerlin3D	15,250	14,906	2,26%

Tabulka 34.: Naměřené hodnoty pro funkce vracející pole

V případě nsLattice1D je zrychlení zanedbatelných 1,01%. U nsValue2D je rozdíl docela značný, činí 413 mikrosekund, přepočteno na procenta 5,34%. Tato hodnota je poněkud vyšší, tak bylo provedeno ještě jedno kontrolní měření, ve kterém byla naměřena hodnota 2,53%. U poslední měřené funkce je zrychlení také zanedbatelných 2,26%.

Nakonec jsem provedl přeprogramování tak, aby mohly být otestovány i další šumové funkce. Výsledky neukázaly nic nového. Nárůst výkonu se pohyboval mezi 1% až 5%, přičemž oněch pět procent bylo naměřeno jen jednou. Průměrná hodnota zlepšení byla 1,89%.

Závěrem lze jen říci, že efekt zrychlení je mizivý a toto řešení nepřináší žádné výhody. Jelikož není vhodné knihovnu plnit zbytečnými funkcemi, bylo od implementace funkcí vracející pole upuštěno.

7.2 Funkce vracející vektory

Podle nadpisu by se dalo pochybovat o tom, proč vlastně vytvářet funkce, které budou vracet vektor s náhodnými čísly. Vždyť mohu zavolat 3x šumovou funkci a získám také vektor.

Náhodný vektor se může hodit pro „náhodný pohyb“, který přitom není tak úplně náhodný. Jednou z hlavních vlastností šumu je, že je pseudonáhodný a opakovatelný. Pokud je funkce zavolána se stejnými parametry, tak návratovou hodnotou bude zase stejný vektor. Šumové vektory jsou využívány v Ray-tracerech, bump mappingu a určitě najdou i další uplatnění.

Příkladem může být třeba některá hra, kde se ozbrojený „lump“ pohybuje z nějakého výchozího bodu. Pokud je zneškodněn, tak po určitém čase je tato postava opět obnovena. Je-li vytvořen zase na stejném místě, bude svoje procházky po mapě provádět s naprosto stejnými kroky jako předtím. Nemusí se mu tedy předem specifikovat každý jeho krok, ale stačí vědět výchozí bod.

Na začátku bylo řečeno, že na to není potřeba zvláštní funkce, že stačí třikrát (pro třísložkový vektor) zavolat šumovou funkci. V podstatě by to tak mohlo být, ale pokud nebudou změněny parametry funkce, tak návratová hodnota bude stále stejná a výsledkem bude stále stejný vektor. Cílem této funkce je tedy vytvořit vektor pomocí dvou nebo tří šumových funkcí, ale tak, aby byly složky vektoru pseudonáhodné.

První myšlenka byla taková, že stačí pouhé zpřeházení pořadí jednotlivých souřadnic jako parametry jednotlivých volání funkcí. Toto řešení odstranilo stejné hodnoty ve vektoru, ale za jedné podmínky. Souřadnice nesmí mít stejné hodnoty. Pokud jsou stejné, pak zamíchání nemá smysl a hodnoty prvků vektorů budou stejné. Z toho důvodu je nutné hodnoty parametrů nějakým způsobem změnit tak, aby byly zaručeny různé hodnoty prvků vektoru.

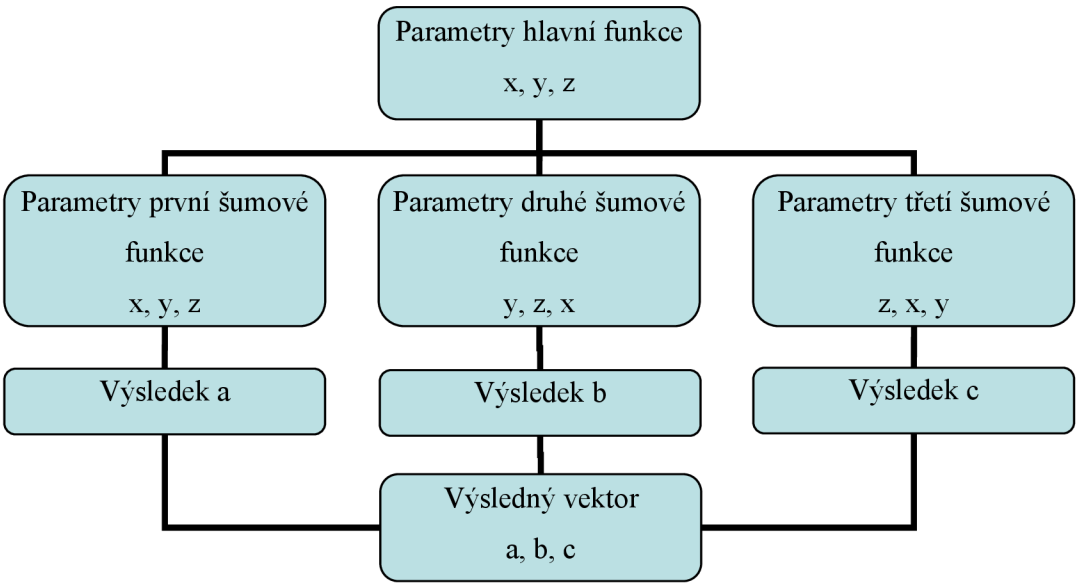


Diagram 1.: Zpřeházení pořadí jednotlivých souřadnic

Další možností je zachování pořadí parametrů, ale ke každé souřadnici u všech šumových funkcí je přičtena libovolná hodnota. Ty zaručí, že šumová funkce bude mít vždy různé parametry a vrátí jinou hodnotu. Přičítaná hodnota je číslo s několika desetinnými místy. Pravděpodobnost, že mohou parametry nabývat stejných hodnot je poté mizivá. Nastává tu ale jiný problém. Tím, že souřadnice nejsou promíchány a měnila by se hodnota jen jednoho parametru, tak by došlo k tomu, že by se měnila jen jedna složka vektoru.

Kombinací předchozích dvou návrhů dostaneme požadovaný výsledek. Přičtením hodnot se odstraní možnost stejných parametrů a jejich zamícháním zaručíme změnu výsledného čísla každé složky vektoru.

Toto je nejjednodušší a nejefektivnější možnost, jak toho dosáhnout. Samozřejmě by šlo souřadnice ještě násobit a provádět nad nimi i jiné operace tak, aby došlo k naprostému zamíchání, ale tím by také vzrostla výpočetní náročnost. Následující diagram ukazuje, jak jsou parametry měněny.

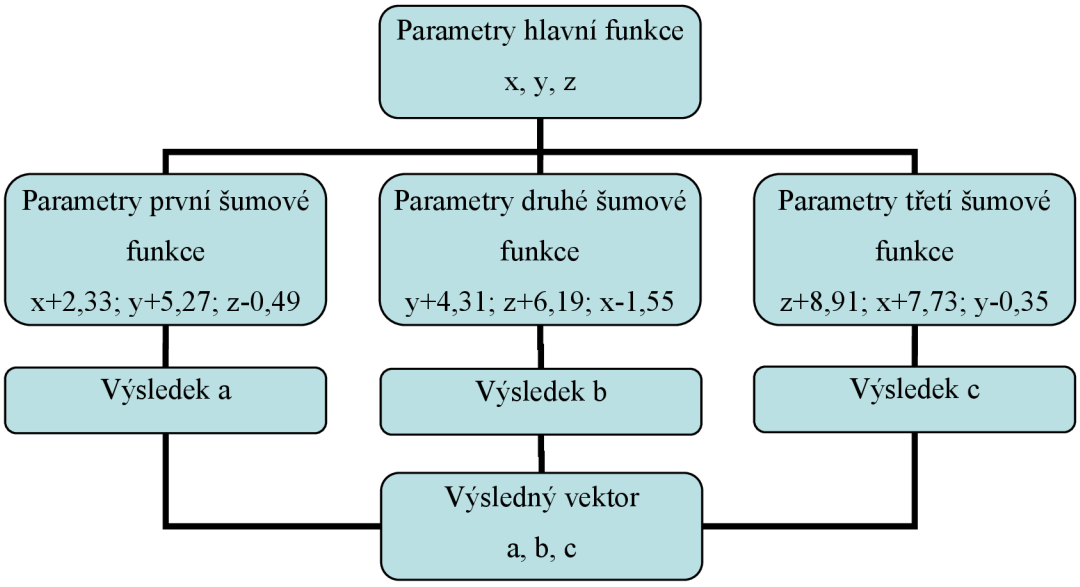


Diagram 2.: Výsledné zamíchání s parametry jednotlivých funkcí

8 Možnosti využití a ukázky

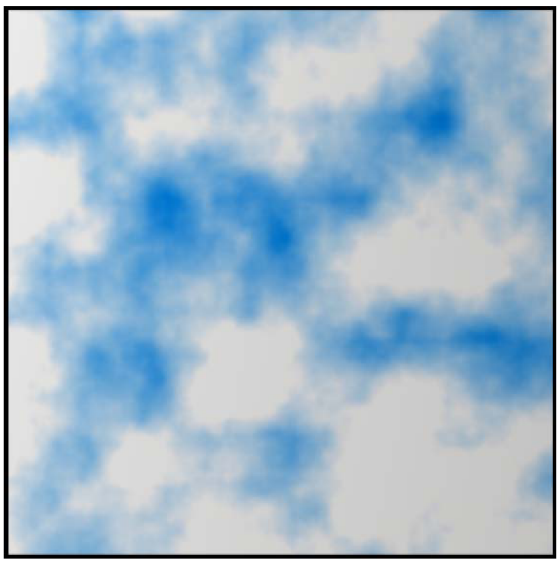
Celá tato práce se zabývala tím, jak šum vygenerovat. Již v úvodu bylo napsáno, že hlavním účelem generování šumu je použití v počítačové grafice a ve vytváření textur různých materiálů. Proto byl vytvořen program, který využívá vytvořenou knihovnu a takové textury vytváří.

8.1 Mramor, dřevo, oblaka

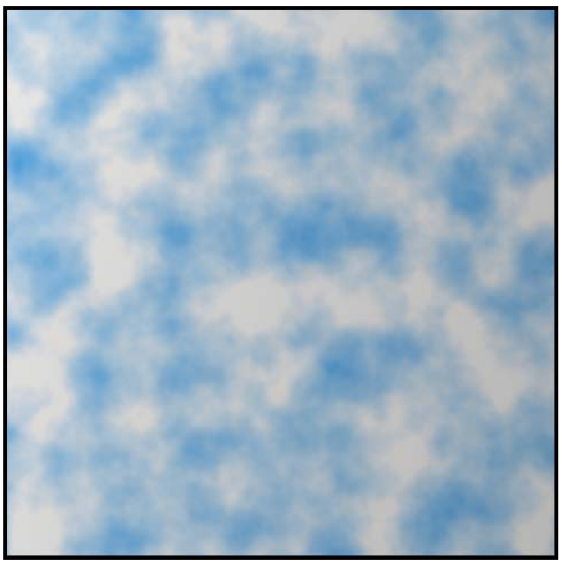
Pro vytvoření těchto textur je potřeba fraktální sumy nebo turbulence.

Mramor je vytvořen funkcí sinus a přidáním náhodných hodnot a souřadnic do této funkce dojde k tomu, že rovné čáry funkce sinus se „rozbijí“. Textura dřeva také využívá pro soustředné kruhy funkce sinus, ale k tomu je také započítána vzdálenost od středu a v případě 3D rozměru se také mění souřadnice hloubky. V případě mraků stačí jen použití fraktální sumy.

Jenže dle vlastních zkušeností musím poznamenat, že vytvoření textury není jednoduché. Výpočet záleží na správně zvolené metodě a správném nastavení několika parametrů výpočtu. Zásadní je výběr šumové metody. Ne každá metoda se hodí pro výpočet jakékoliv textury. Příkladem může být vytvoření textury mraků pomocí Value noise a Perlin noise, tak jak to ukazují následující obrázky.



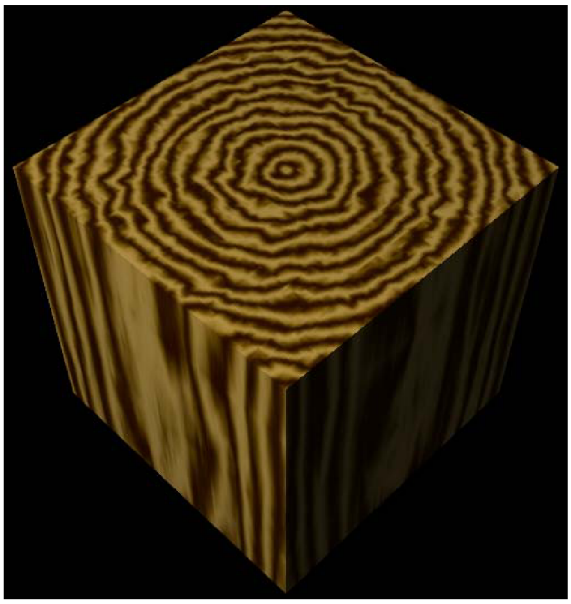
Obrázek 33.: Oblaka pomocí fraktální sumy Value noise



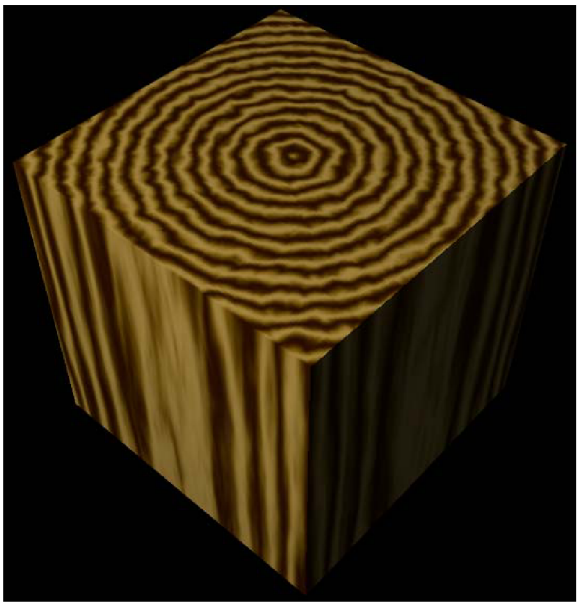
Obrázek 34.: Oblaka pomocí fraktální sumy Perlinova šumu

Oblaka pomocí Perlinova šumu jsou více „rozbité“ a nejsou tak celistvé jako v případě Value noise. Dalším aspektem je také nastavení jednotlivých parametrů fraktální sumy, které může být někdy docela obtížné.

Uvedu ještě jeden příklad, kdy je lépe použít naopak Perlinův šum a turbulenci. Jedná se o texturu dřeva.



Obrázek 35.: Textura dřeva generovaná turbulencí Value noise

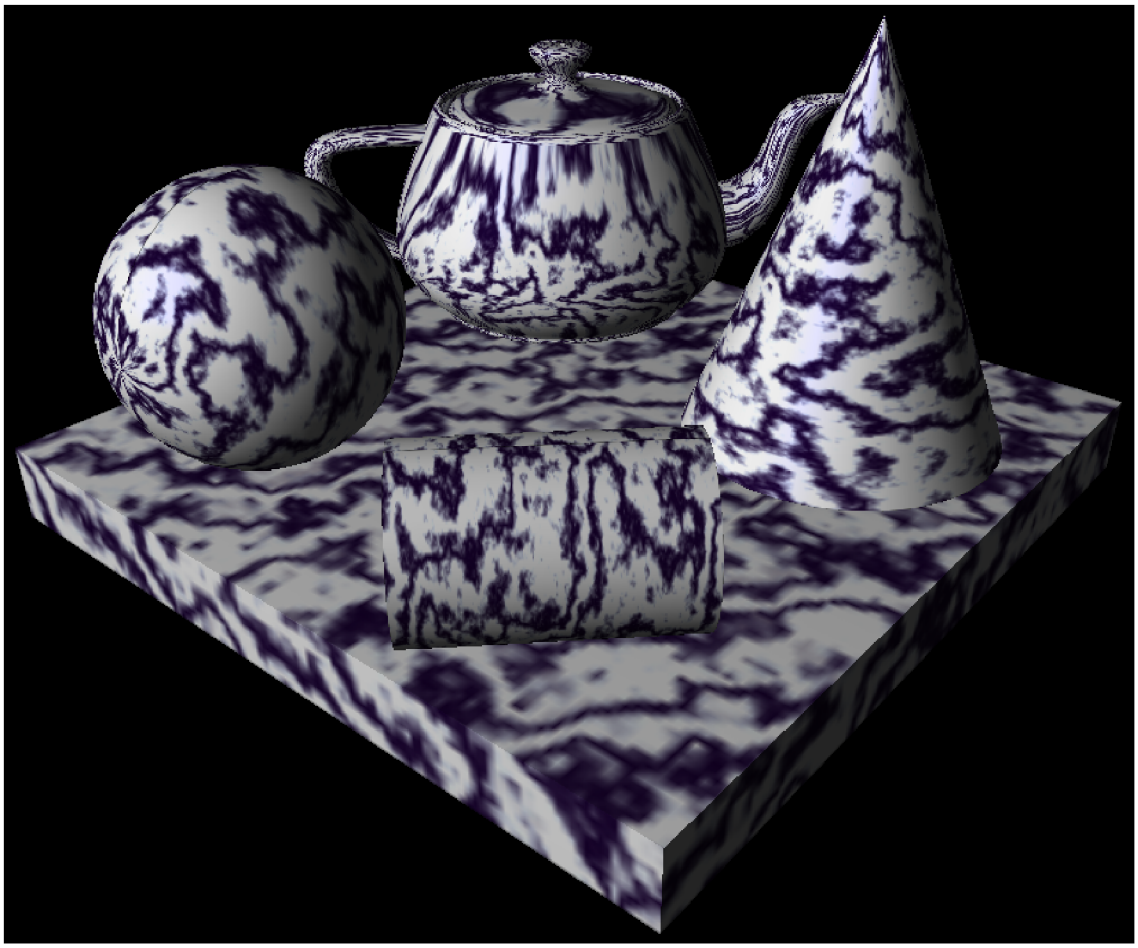


Obrázek 36.: Textura dřeva generovaná turbulencí Perlin noise

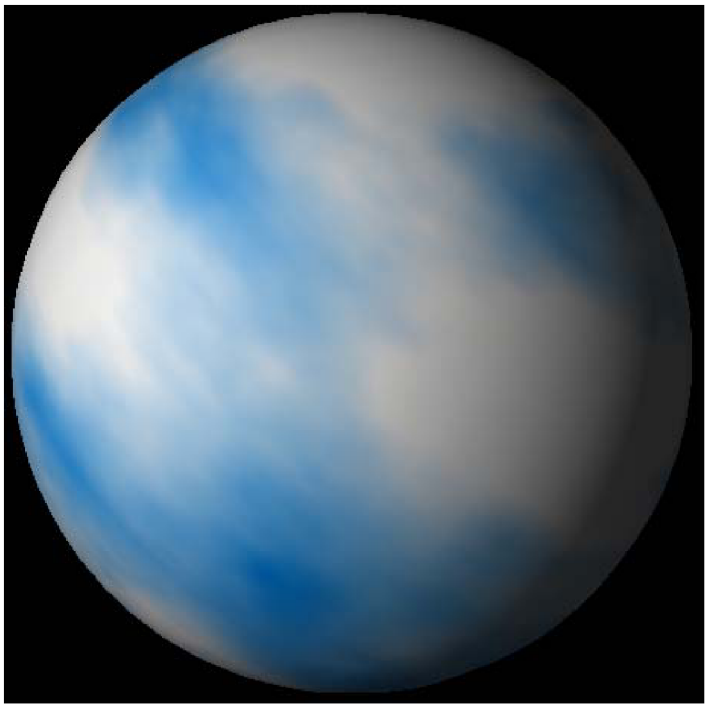
U textury dřeva je zase lepší použít Perlinův šum, protože ten netvoří pravidelné čáry (rovné úseky) a tvary tak jako Lattice Value noise, který je generován z mřížky. Perlinův šum je více uhlazený a nepravidelný. Určitě by se dala vygenerovat textura dřeva i pomocí jiné metody, ale pak je nutné opět hledat správné nastavení změn souřadnic a parametrů turbulence a to ne vždy podaří.

Pomocí šumu a přiloženého programu v OpenGL lze vygenerovat následující obrázky.

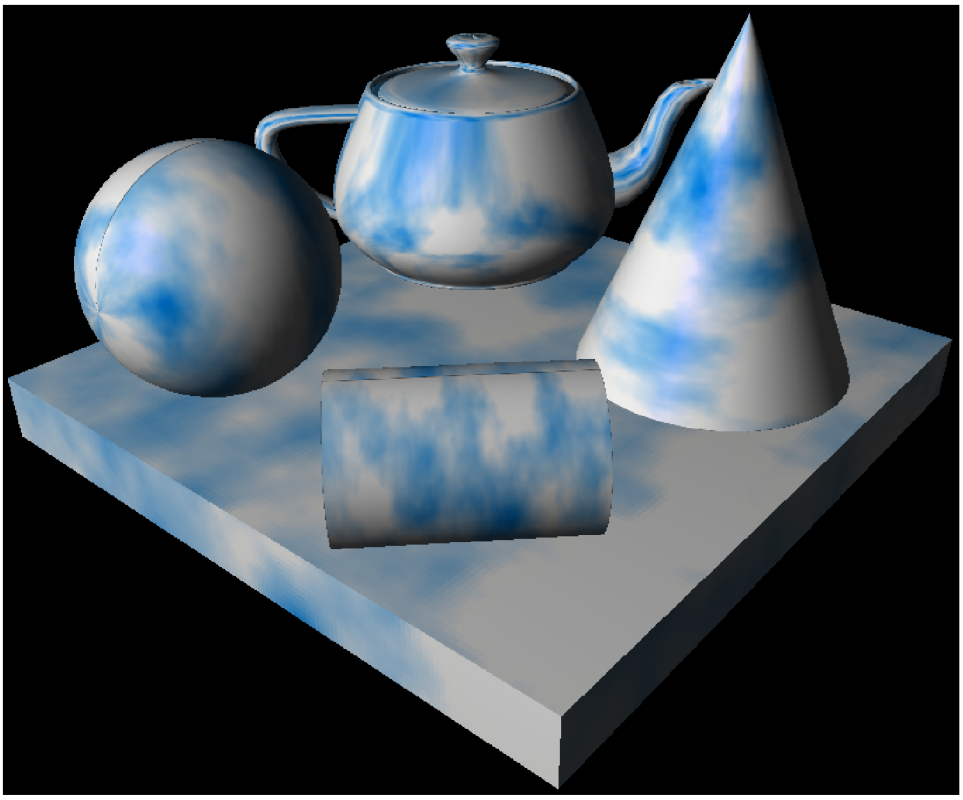
Obrázek 37.: Konvička s texturou mramoru



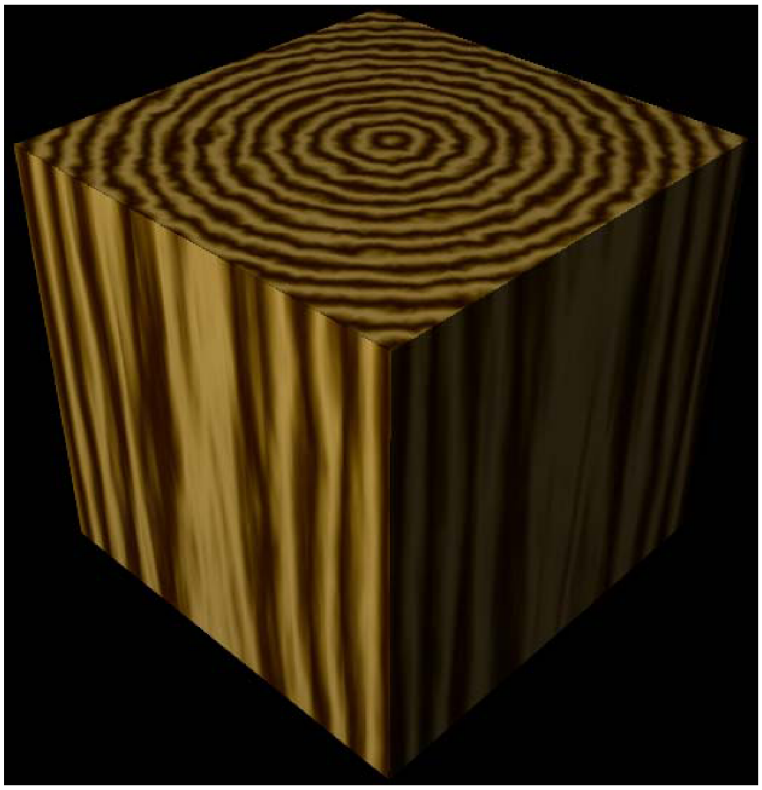
Obrázek 38.: Sestava více objektů s texturou mramoru



Obrázek 39.: Koule s texturou oblaků



Obrázek 40.: Sestava s „oblačnými“ objekty



Obrázek 41.: Dřevěný špalek

Obrázek 42.: Dřevěná sestava různých objektů

8.2 Další možnosti využití

Další využití je změna tvaru povrchů, různé textury a náhodný pohyb. Následuje několik textur jako další příklady použití.

Obrázek 43.: 3D rozměr v oblacích [20]

Obrázek 44.: Vodopád virtuální vody [20]

Obrázek 45.: Vodopád lávy [20]

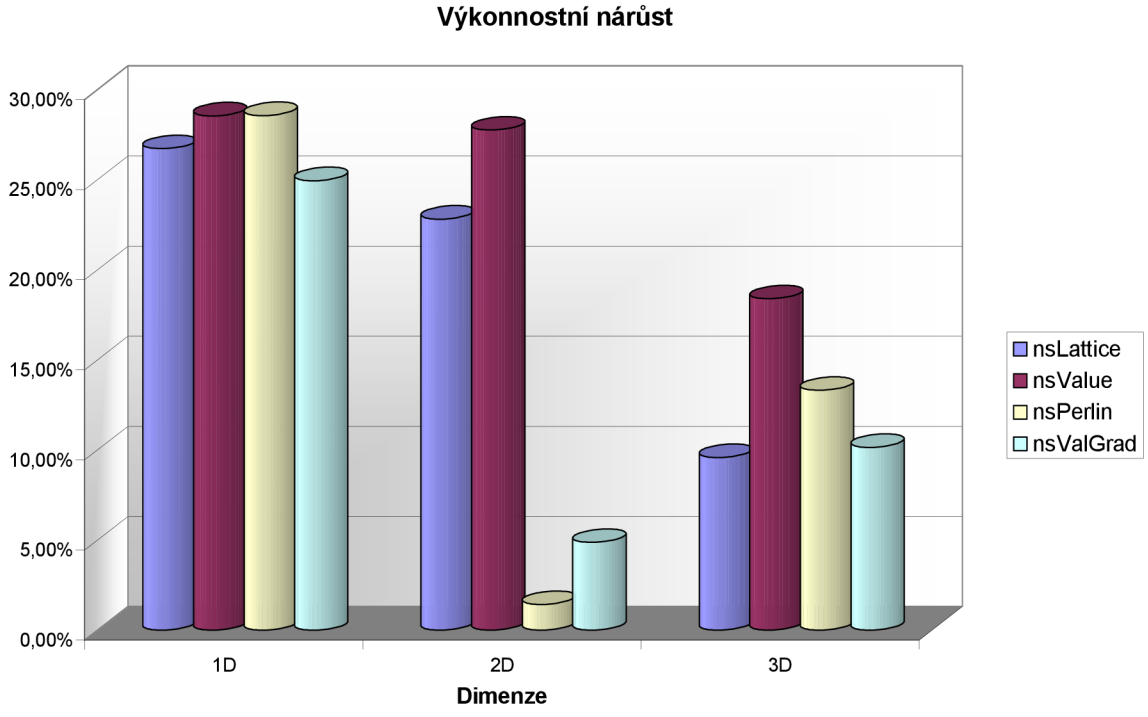
9 Závěrečné výsledky a srovnání

Původní návrh rozhraní počítal s funkcemi, které jsou implementovány v knihovně. Tato knihovna měla být v odděleném souboru, a aby mohly být jednotlivé funkce zpřístupněny, je nutné použít hlavičkový soubor. Nic z toho ovšem nakonec nebylo realizováno a funkce mají také trochu odlišné parametry než v původním návrhu.

Jak již bylo napsáno v úvodu, není nutnou podmínkou, aby byla knihovna ve zvláštním souboru s vlastním rozhraním. Příkladem může být STL (Standard Template Library), která je implementována pouze v hlavičkových souborech. Stejným způsobem je řešena i tato knihovna. Důvodem tohoto řešení jsou makra, do kterých byly funkce přeprogramovány, a která nepovolují kvůli svým vlastnostem vytvoření knihovny. Makra pak také stojí za změnou parametrů funkcí.

Nevím, jestli mohu mluvit o funkcích, ale jsou to spíše procedury. Funkce má návratovou hodnotu. Makra nemají návratové hodnoty, takže proměnná, do které je výsledek ukládán, je předávána jako parametr. Proto byla nutná změna hlaviček jednotlivých funkcí (procedur). Ty se také změnily kvůli typové nezávislosti. Více informací si lze přečíst v dokumentaci ke knihovně.

Teď to vypadá, že přeprogramováním funkcí do maker má více nevýhod než výhod. Hlavním důvodem přeprogramování je typová nezávislost a efektivnost výpočtu, tak jak ukázaly předcházející testy. Na závěr jsem tedy vytvořil poslední graf nárůstu výkonu jednotlivých funkcí (procedur) oproti prvním verzím. V 1D je rozdíl opravdu velký a proto se makro vyplatí.



Graf 15.: Graf nárůstu výkonu oproti první implementaci

10 Závěr

V procedurálním texturování hraje šum velice důležitou roli a pro zobrazení materiálů reálného světa je nepostradatelný. Procedurální texturování je velmi mocný nástroj pro tvorbu realtime textur, kde není nutné neustále nahrávat velké obrázky textur a složitě je nanášet na objekt. Vzhledem k tomu, že v dnešní době nabízí grafické procesory samostatné programování a vysoký výkon, pak není důvod váhat nad jejich použitím. Ne však všechny metody lze použít, protože jejich výpočetní náročnost již není malá.

Díky této práci jsem se blíže seznámil s problematikou procedurálního texturování a použití šumu. Snažil jsem se co nejlépe popsat nastudovanou problematiku texturování, šumu a jednotlivých metod, které se používají. Byly také popsány kapitoly, které se šumem úzce souvisí.

Na základě teoretického rozboru a navrženého rozhraní byla knihovna implementována. První implementace byla pomocí klasických funkcí, ale po provedení několika testů a zvážení dalších možností implementace byly funkce přeprogramovány do maker, pomocí kterých bylo dosaženo velkého zrychlení výpočtu. O tomto tématu už jsem psal v předchozí kapitole a tak by bylo zbytečné se nadále opakovat.

Za vlastní přínos pokládám dvojí implementaci Lattice Valu Noise, kde si uživatel může vybrat mezi paměťově nenáročnou funkcí nsValue nebo časově méně náročnější nsLattice. Dalším přínosem jsou funkce, které vracejí vektory a také spousta provedených testů.

Na této práci by se dalo pokračovat rozšířením o další funkce. Místo použitého „klasického“ generátoru náhodných čísel by se mohl naprogramovat jiný generátor. V knihovně by mohly být obsaženy také funkce pro přímý výpočet textur známých materiálů, jako je dřevo a mramor. Samozřejmě by se dala rozšířit o další funkce jiných metod, které jsou mnohem více výpočetně náročnější, ale generují kvalitnější šum. S tím je pak spojená možnost výpočtu v GPU. V dnešní době se rychle rozvíjí programy počítající nad GPU, protože nabízí velmi vysoký výkon.

Literatura

[1] NVIDIA Corporation, FERNANDO, R. (ed.). *GPU Gems : Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Second printing. Boston : NVIDIA Corporation. 2004. ISBN 0-321-22832-4.

[2] ŽÁRA, Jiří, BENEŠ, Bedřich, SOCHOR, Jiří, FELKEL, Petr. *Moderní počítačová grafika*. Redaktor Tomáš Tůma. 2. přeprac. rozšiř. vyd. Brno : Computer Press, c2004. 609 s. ISBN 80-251-0454-0.

[3] *Game Programming Gems 2*. Edited by Mark A. DeLoura. Hingham(Massachussets) : Charles River Media, 2003. 575 s., 1 CD-ROM. ISBN 15-845-0054-9.

[4] ERIKSSON, Erik. *Noise in Real-time 3D Graphics* [online]. [s.l.], 2004. 45 s. School of Computer Science and Engineering, Royal Institute of Technology. Vedoucí diplomové práce Lars Kjelldahl. [cit. 2005-12-29]. Dostupný z WWW: <http://www.nada.kth.se/utbildning/grukth/exjobb/rapportlistor/2004/rapporter04/eriksson_erik_04161.pdf>.

[5] HEROUT, Adam. Počítačová grafika : Procedurální textury, texturování a modelování. Ústav počítačové grafiky a multimédií, Fakulta informačních technologií, Vysoké Učení Technické Brno. 2005. [cit. 2005-12-29].

[6] PERLIN, Ken. *Making noise* [online]. [1999] [cit. 2005-12-29]. Angličtina. Dostupný z WWW: <http://www.noisemachine.com/talk1/>.

[7] PERLIN, Ken. *Improving noise* [online]. c2002. Media Research Laboratory, Dept. of Computer Science, New York University. [cit. 2005-12-29]. Dostupný z WWW: <http://mrl.nyu.edu/~perlin/paper445.pdf>.

[8] BOURKE, Paul. *Perlin Noise and Turbulence* [online]. 2000 [cit. 2005-12-29]. Angličtina. Dostupný z WWW: <http://astronomy.swin.edu.au/~pbourke/texture/perlin/>.

[9] LONG, James. *Perlin Noise and its fractal nature* [online]. 2005 [cit. 2005-12-29]. Angličtina. Dostupný z WWW: <http://www.animeimaging.com/asp/PerlinNoise.aspx>.

[10] ELIAS, Hugo. *Perlin Noise* [online]. 1998 , 22:18 - 4 Feb 2000 [cit. 2005-12-29]. Angličtina. Dostupný z WWW: <http://freespace.virgin.net/hugo.elias/models/m_perlin.htm>.

[11] VANDEVENNE, Lode. *Lode's Computer Graphics Tutorial : Perlin Noise* [online]. 2004 [cit. 2005-12-29]. Angličtina. Dostupný z WWW: <http://www.student.kuleuven.ac.be/~m0216922/CG/perlinnoise.html>.

[12] LEWIS, J. P., *Algorithms for Solid Noise Texturing* [online]. 1989. Computer Graphics Laboratory, New York Institute of Technology. [cit. 2005-12-29]. Agličtina. Dostupný z WWW: <http://www.idiom.com/~zilla/Work/Crunge/crunge.pdf>.

[13] *Flow Visualization : Mapping Techniques* [online]. [cit. 2005-12-29]. Dostupný z WWW: <http://www.cs.purdue.edu/homes/sun/Teach/530_04F/Notes/07d_flowVis_mapTechnique.pdf>.

[14] HART, John C. *Perlin Noise Pixel Shaders* [online]. [2000]. University of Illinois, Urbana-Champaign. [cit. 2005-12-29]. Agličtina. Dostupný z WWW: <http://graphics.cs.uiuc.edu/~jch/papers/pixelnoise.pdf>.

[15] KAMEYA, Masaki, HART, John C. *Bresenham Moise* [online]. School of Electrical Engineering and Computer Science, Washington State University. [cit. 2005-12-29]. Agličtina. Dostupný z WWW: <http://graphics.cs.uiuc.edu/~jch/papers/bresnoise.pdf>.

[16] *Náhodné jevy* [online]. 2001 [cit. 2005-12-29]. Český. Dostupný z WWW: <http://cml.fsv.cvut.cz/~kupca/qc/node26.html>.

[17] *Kryptografie a počítačová bezpečnost : Náhodné a pseudonáhodné generátory* [online]. [cit. 2005-12-29]. Český. Dostupný z WWW: <http://www.cs.vsb.cz/ochodkova/courses/kpb/KPB_8_05.pdf>.

[18] GUŠTAR, Milan. Generování náhodně proměnných veličin v metodě Monte carlo [online]. 2000. Celostátní konference SPOLEHLIVOST KONSTRUKCÍ, Dům techniky Ostrava. [cit. 2005-12-29]. Dostupný z WWW: <http://www.noise.cz/SBRA/doc/2000a.pdf>. ISBN 80-02-01344-1.

[19] PERLIN, Ken. *Ken's Academy Award : Noise and Turbulence* [online]. [cit. 2007-4-20]. Angličtina. Dostupný z WWW: <http://mrl.nyu.edu/~perlin/doc/oscar.html>.

[20] PERLIN, Ken, NEYRET Fabrice. *Flow Noise* [online]. [cit. 2007-5-2]. Angličtina. Dostupný z WWW: <http://mrl.nyu.edu/~perlin/flownoise-talk/#0>

Seznam příloh

Příloha 1

Dokumentace ke knihovně

Knihovna pro výpočet šumů je implementována v jednom hlavičkovém souboru „**noise.h**“.

Pro využití funkcí z knihovny stačí, aby byl hlavičkový soubor knihovny vložen do souboru se zdrojovým kódem, kde budou funkce volány. Vložení probíhá pomocí direktivy *#include "noise.h"* na začátku programu.

Než budou jednotlivé funkce použity, je potřeba inicializovat všechna pole pro výpočet šumů. Nabízí se dvě možnosti. První možnost je zavolání funkce „**setTabSize(x)**“, kde parametr „x“ udává velikost tabulek, nebo zavoláním funkce „**setSeed(y)**“ pro nastavení semínka pro generátor náhodných čísel. Obě funkce mají za následek inicializaci polí. Nyní je vše připraveno a lze se pustit do výpočtu šumu.

Seznam symbolických konstant a struktur v knihovně

Symbolická konstanta	Přiřazená metoda	Použití
NS_LATTICE_2D	2D Lattice noise	Fraktální suma a turbulence, funkce pro výpočet vektoru
NS_LATTICE_3D	3D Lattice noise	
NS_LATTICE_FS_2D	2D fraktální suma Lattice noise	Funkce pro výpočet vektoru
NS_LATTICE_TB_2D	2D turbulence Lattice noise	
NS_LATTICE_FS_3D	3D fraktální suma Lattice noise	
NS_LATTICE_TB_3D	3D turbulence Lattice noise	
NS_VALUE_2D	2D Value noise	Fraktální suma a turbulence, funkce pro výpočet vektoru
NS_VALUE_3D	3D Value noise	
NS_VALUE_FS_2D	2D fraktální suma Value noise	Funkce pro výpočet vektoru
NS_VALUE_TB_2D	2D turbulence Value noise	
NS_VALUE_FS_3D	3D fraktální suma Value noise	
NS_VALUE_TB_3D	3D turbulence Value noise	
NS_PERLIN_2D	2D Perlin noise	Fraktální suma a turbulence, funkce pro výpočet vektoru
NS_PERLIN_3D	3D Perlin noise	
NS_PERLIN_FS_2D	2D fraktální suma Perlin noise	Funkce pro výpočet vektoru
NS_PERLIN_TB_2D	2D turbulence Perlin noise	
NS_PERLIN_FS_3D	3D fraktální suma Perlin noise	
NS_PERLIN_TB_3D	3D turbulence Perlin noise	

NS_VALGRAD_2D	2D Value-Gradient noise	Fraktální suma a turbulence, funkce pro výpočet vektoru
NS_VALGRAD_3D	3D Value-Gradient noise	
NS_VALGRAD_FS_2D	2D fraktální suma Value-Gradient noise	Funkce pro výpočet vektoru
NS_VALGRAD_TB_2D	2D turbulence Value-Gradient noise	
NS_VALGRAD_FS_3D	3D fraktální suma Value-Gradient noise	
NS_VALGRAD_TB_3D	3D turbulence Value-Gradient noise	

Tabulka 35.: Tabulka symbolických konstant

Symbolické konstanty byly do knihovny přidány za účelem zjednodušení volání některých funkcí. Ty mají totiž stále stejný výpočet, ale uvnitř jsou volány různé šumové metody. Jedná se o funkce pro fraktální sumu a turbulenci a také funkce pro výpočet vektoru. Aby nemusela být funkce definována pro každou základní metodu, tak jeden z parametrů je symbolická konstanta, která udává, se kterou metodou se bude počítat.

Struktury jsou pouze dvě a slouží jako návratová hodnota pro funkce vracející vektor.

Struktura	Členové	Příklad deklarace proměnné
VECTOR2	**double** x, y	`VECTOR2 myVector2D;`
VECTOR3	**double** x, y, z	`VECTOR3 myVector3D;`

Tabulka 36.: Tabulka struktur

Seznam a popis inicializačních funkcí

Následuje krátký seznam inicializačních funkcí, jejich deklarace, seznam parametrů a popis.

- `void setTabSize(int size);`

  Funkce nastaví velikost permutačních polí a polí hodnot a gradientů na velikost *`size`*. Výjimkou je trojrozměrné pole pro Value noise, které má konstantní velikost 32.
- `int getTabSize(void);`

  Návratovou hodnotou funkce je aktuální velikost tabulek
- `void setSeed(unsigned int s);`

  Funkce pro nastavení semínka na hodnotu *`s`*. Semínko je použito jako počáteční hodnota pro generátor náhodných čísel.

Makro funkce (procedury)

Makra nemají návratovou hodnotu a nejsou to ani funkce, spíše procedury. Z formálního hlediska je ale budu stále nazývat funkcemi. Jednotlivé funkce jsou seřazeny podle metod. Všechny funkce obsahují parametr *`type`*. Ten udává datový typ, nad kterým bude výpočet probíhat a může nabývat dvou hodnot, konkrétně ***float*** a ***double***. Všechny ostatní parametry by pak měly být stejného datového

typu, jaký je zadán parametrem *type*. Souřadnice jsou čísla libovolné velikosti. Návratová hodnota většiny funkcí je v intervalu <-1, 1> kromě turbulence, která vrací čísla v intervalu <0, 1>.

- **Lattice noise**
  - `nsLattice1D(type, x, r);`

    Výpočet Lattice noise v 1D pro datový typ *type* ze souřadnice *x*. Výsledek se uloží do *r* datového typu *type*.
  - `nsLattice2D(type, x, y, r);`

    Funkce spočítá Lattice noise ve 2D pro datový typ *type* dle souřadnic *x*, *y* a výsledek uloží do *r*.
  - `nsLattice3D(type, x, y, z, r);`

    3D verze Lattice noise s datovým typem *type*. Výpočet je podle vstupních souřadnic *x*, *y*, *z* a výsledek je uložen v *r*.
- **Value noise**
  - `nsValue1D(type, x, r);`

    Výpočet Value noise v 1D pro datový typ *type* ze souřadnice *x*. Výsledek se uloží do *r* datového typu *type*.
  - `nsValue2D(type, x, y, r);`

    Funkce spočítá Value noise ve 2D pro datový typ *type* dle souřadnic *x*, *y* a výsledek uloží do *r*.
  - `nsValue3D(type, x, y, z, r);`

    3D verze Value noise s datovým typem *type*. Výpočet je podle vstupních souřadnic *x*, *y*, *z* a výsledek je uložen v *r*.
- **Perlin noise**
  - `nsPerlin1D(type, x, r);`

    Výpočet Perlin noise v 1D pro datový typ *type* ze souřadnice *x*. Výsledek se uloží do *r* datového typu *type*.
  - `nsPerlin2D(type, x, y, r);`

    Funkce spočítá Perlin noise ve 2D pro datový typ *type* dle souřadnic *x*, *y* a výsledek uloží do *r*.
  - `nsPerlin3D(type, x, y, z, r);`

    3D verze Perlin noise s datovým typem *type*. Výpočet je podle vstupních souřadnic *x*, *y*, *z* a výsledek je uložen v *r*.

- **Value-Gradient noise**
  - `nsValueGradient1D(type, x, r);`

    Výpočet Value-Gradient noise v 1D pro datový typ *type* ze souřadnice *x*. Výsledek se uloží do *r* datového typu *type*.
  - `nsValueGradient2D(type, x, y, r);`

    Funkce spočítá Value-Gradient noise ve 2D pro datový typ *type* dle souřadnic *x*, *y* a výsledek uloží do *r*.
  - `nsValueGradient3D(type, x, y, z, r);`

    3D verze Value-Gradient noise s datovým typem *type*. Výpočet je podle vstupních souřadnic *x*, *y*, *z* a výsledek je uložen v *r*.
- **Funkce pro skládání šumu**

  Funkce pro fraktální sumu a turbulenci mají navíc čtyři parametry.
  - Parametr *function* je symbolická konstanta, která udává metodu, se kterou bude funkce počítat. Pokud se jedná o fraktální sumu nebo turbulenci ve 2D, pak možnými hodnotami jsou NS_LATTICE_2D, NS_VALUE_2D, NS_PERLIN_2D a NS_VALGRAD_2D. Pro 3D se změní poslední součást parametru.
  - Podle parametru *alpha* (typ **float**) je vypočítávána amplituda (váha) jednotlivých oktáv, které se sčítají. Standardně je to hodnota 2,0. Pokud je hodnota větší, amplituda klesá rychleji a tím se snižuje váha oktáv s většími detaily. Pokud je hodnota menší, amplituda neklesá tak rychle a oktávy s vyšší frekvencí mají větší vliv na výsledek. To má za následek více detailů.
  - Parametr *beta* (typ **float**) udává základní číslo pro výpočet frekvence. Standardně je tato hodnota 2,0. Pokud je menší, frekvence s počtem oktáv nebude tolik růst a dojde k většímu vlivu první oktávy a ztrátě detailů. Pokud je hodnota větší, výsledek obsahuje více detailů a textura je celkově větší. Výpočet frekvence se dá vyjádřit vzorcem *frekvence* = $beta^i$, kde *i* je i-tá oktáva počítaná od nuly.
  - Posledním parametrem pro výpočet je *n*. Tato hodnota datového typu **int** udává počet oktáv. Maximální doporučený počet je 10. Při větším počtu nejsou detaily této oktávy už zobrazitelné a nemají vliv na výsledek.
  - `nsFractalSum2D(type, function, x, y, alpha, beta, n, r);`

    Fraktální suma ve 2D, která je provedena nad základní šumovou funkcí specifikovanou symbolickou konstantou *function*. Výpočet probíhá podle souřadnic *x*, *y* a je řízen třemi parametry *alpha*, *beta* a *n*. Výsledek je uložen v *r*.

    příklad: nsFractalSum2D(**double**, NS_VALUE_2D, u, v, 2.0, 2.0, 8, result);
  - `nsTurbulence2D(type, function, x, y, alpha, beta, n, r);`

    Turbulence ve 2D, která je provedena nad základní šumovou funkcí specifikovanou

symbolickou konstantou *function*. Výpočet probíhá podle souřadnic *x*, *y* a je řízen třemi parametry *alpha*, *beta* a *n*. Výsledek je uložen v *r* a nabývá hodnot v intervalu <0, 1>.

- `nsFractalSum3D(type, function, x, y, z, alpha, beta, n, r);`
  Fraktální suma ve 2D, která je provedena nad základní šumovou funkcí specifikovanou symbolickou konstantou *function*. Výpočet probíhá podle souřadnic *x*, *y*, *z* a je řízen třemi parametry *alpha*, *beta* a *n*. Výsledek je uložen v *r*.
- `nsTurbulence3D(type, function, x, y, z, alpha, beta, n, r);`
  Turbulence ve 2D, která je provedena nad základní šumovou funkcí specifikovanou symbolickou konstantou *function*. Výpočet probíhá podle souřadnic *x*, *y*, *z* a je řízen třemi parametry *alpha*, *beta* a *n*. Výsledek je uložen v *r* a nabývá hodnot v intervalu <0, 1>.

- **Funkce vracející vektor**
  - `nsVecNoise2D(type, function, u, v, r);`
    Funkce vypočítá vektor pomocí 2D šumové metody definované konstantou *function* na základě souřadnic *u*, *v* a výsledek uloží do složek struktury *r*, která je definována jako VECTOR2.
  - `nsVecOctaveNoise2D(type, function, u, v, alpha, beta, num, r);`
    Funkce vypočítá vektor pomocí 2D funkce, která skládá více oktáv. Fraktální suma nebo turbulence. Parametr *function* může nabývat například NS_LATTICE_FS_2D, NS_VALUE_TB_2D, NS_PERLIN_TB_2D, atd. Ostatní parametry jsou stejné jako pro funkce skládající šum. Výsledek je uložen do struktury *r*.
  - `nsVecNoise3D(type, function, u, v, r);`
    Funkce vypočítá vektor pomocí 3D šumové metody definované konstantou *function* na základě souřadnic *u*, *v* a výsledek uloží do složek struktury *r*, která je definována jako VECTOR3.
  - `nsVecOctaveNoise3D(type, function, u, v, w, alpha, beta, num, r);`
    Funkce vypočítá vektor pomocí 3D funkce, která skládá více oktáv. Fraktální suma nebo turbulence. Parametr *function* může nabývat například NS_LATTICE_FS_3D, NS_VALUE_TB_3D, NS_PERLIN_TB_3D, atd. Ostatní parametry jsou stejné jako pro funkce skládající šum. Výsledek je uložen do struktury *r*.

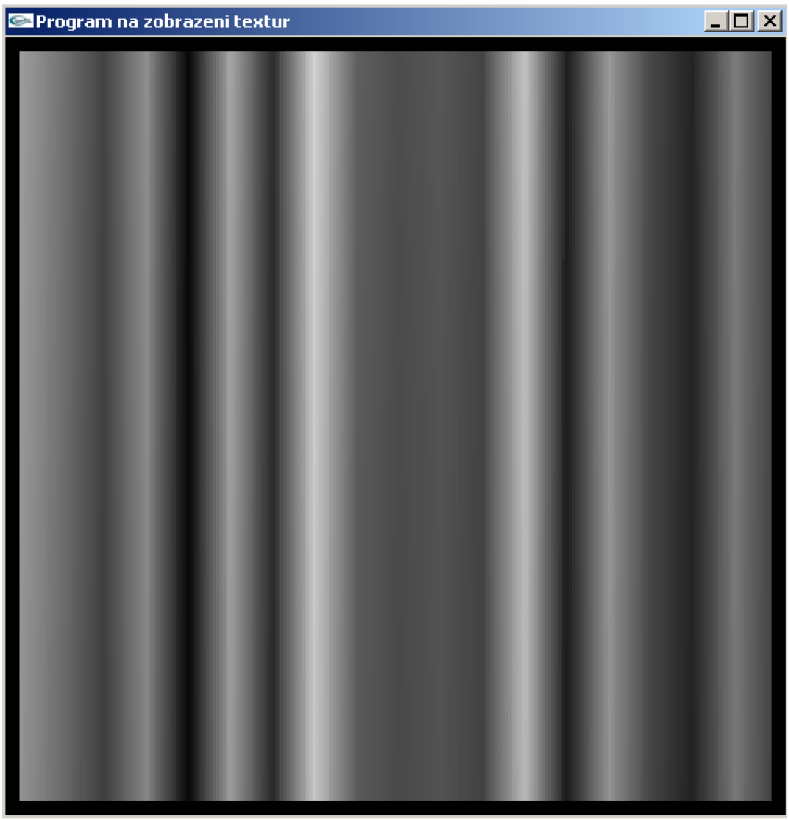
Příloha 2

Manuál k přiloženému programu

Na přiloženém CD lze nalézt binární (spustitelnou) a zdrojovou (zdrojový kód) verzi programu.

Jelikož je program vytvořen v OpenGL, je nutné, aby v počítači byly nainstalovány potřebné knihovny. Operační systém Windows XP automaticky instaluje dvě potřebné knihovny opengl32.dll a glu32.dll (adresář system32 v kořenovém adresáři OS). V programu je ovšem použita i knihovna glut32.dll, která není součástí OS, proto je také přiložena u tohoto programu.

Po spuštění programu se provádí výpočet jednotlivých procedurálních textur. Doba výpočtu je závislá na frekvenci procesoru a může trvat i několik desítek sekund. Po vypočítání všech textur se objeví okno s 2D plochou (viz obrázek), která zobrazuje 1D šum.



Obrázek 46.: Úvodní okno programu

V adresáři se zdrojovým kódem programu **(program.cpp)** lze nalézt další dva soubory.

nosie.h je knihovna s funkcemi pro výpočet šumů.

Makefile je skript pro přeložení programu do binární podoby. Pokud jsou v počítači správně nastavené cesty k souborům překladače, stačí v příkazové řádce (konzoli) zadat příkaz „***make***“. Pokud jsou v počítači všechny potřebné knihovny a hlavičkové soubory, dojde k vytvoření spustitelného souboru.

Zdrojová podoba je na CD přiložena proto, aby uživatelé jiného operačního systému než Windows mohli také tento program vytvořit. Knihovny OpenGL jsou totiž přenositelné, takže programy lze vytvářet a spouštět v OS Linux.

Ovládání programu

Program se ovládá pomocí klávesnice a myši. Běžící program je rozložen do dvou oken. Jedním oknem je grafický výstup a druhým je příkazová řádka nebo konzole. Do této konzole jsou vypisovány informace o aktuálně zobrazené textuře.

Ovládání pomocí klávesnice

Klávesa	Význam
l [el]	Zapne / Vypne osvětlení a vyvolá přepočítání textur
s [es]	Změní hodnotu semínka a vyvolá inicializaci tabulek a přepočítání všech textur
MEZERNÍK, d [dé]	Inkrementuje/dekrementuje index šumové metody. Pokud je aktuálně zobrazen samotný šum, je překreslen šumem s následujícím/předchozím indexem. Typ šumu je vypisován do konzole.
t [té]	Změna zobrazené 2D textury (čistý šum, mramor, dřevo, oblaka)
u [ú]	Změna zobrazené 3D textury aplikované na krychli a kvádru (mramor, dřevo, oblaka)
a [á]	Změna zobrazovaného objektu (2D plocha, krychle, koule, konvička, sestava)
1, 2, 3	Změna zobrazení objektů (body, čáry, vyplněné plochy)
ESC, q, x	Ukončení programu
f [ef], w	Přepnutí mezi fullscreen a zpět do okna

Tabulka 37.: Tabulka ovládání programu

Ovládání pomocí myši

Přidržením levého tlačítka v okně grafického výstupu a pohybem myši dojde k rotaci objektu kolem jeho středu v libovolné ose.

Přidržením pravého tlačítka a pohybem myši nahoru a dolů dojde k zazoomování nebo odzoomování objektu.

Příloha 3

Obsah CD

Na přiloženém CD lze nalézt binární a zdrojovou verzi programu, knihovnu a její dokumentaci a také tuto diplomovou práci.

Struktura adresářů na CD

- ***Library*** – Vytvořená knihovna a její dokumentace

  Soubory: **Dokumentace.doc**, **Dokumentace.pdf**, **noise.h**

- ***Program_bin*** **–** Binární verze programu. Spustitelná pod OS Windows

  Soubory: **glut32.dll, manual.doc, manual.pdf, program.exe**

- ***Program_src*** **–** Zdrojový kód programu

  Soubory: **Makefile**, **manual.doc, manual.pdf, noise.h**, **program.cpp**

- ***Masters_Thesis*** **–** Diplomová práce

  Soubory: **Diplomova_prace.doc**, **Diplomova_prace.pdf**