VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

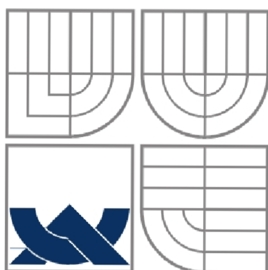
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

GENEROVÁNÍ KOMPLEXNÍCH PROCEDURÁLNÍCH TERÉNŮ NA GPU

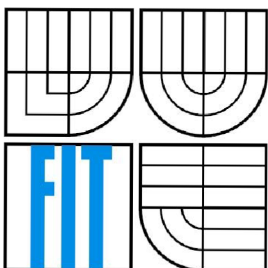
SEMSTRÁLNÍ PROJEKT

AUTOR PRÁCE Bc. Jan Ryba

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

GENEROVÁNÍ KOMPLEXNÍCH PROCEDURÁLNÍCH TERÉNŮ NA GPU

SEMESTRÁLNÍ PROJEKT

AUTOR PRÁCE Bc. Jan Ryba

VEDOUCÍ PRÁCE Doc. Ing. Adam Herout, Ph.D.

BRNO 2010

Abstrakt

Generování komplexních plně prostorových terénů je velmi náročnou činností, buďto datově nebo výpočetní, případně obojí. Datovou náročnost můžeme značně omezit, nebo plně eliminovat použitím procedurálního přístupu, kdy však vyvstává problém výpočetní náročnosti. Zde vstupuje platforma CUDA. Výpočty prováděné paralelně na grafických akcelerátorech mohou výrazně snížit čas nutný pro výpočet. Takto můžeme dosáhnout generování velmi komplexních terénů v reálném čase. Jelikož je metoda plně prostorová, dává nám to možnost navázat generování na jakákoliv volumetrická data. Pro využití v herním, či filmovém průmyslu.

Abstract

Generating fully 3D terrains is a dificult task, meaning that we need to store a lot of data or do a lot of computing or both. We can reduce or completly eliminate the data srorage by using a procedural approch, but this ir where the problem gets realy computationaly costly. This is where the CUDA platform comes in. CUDA kernels run parallely on graphic accelerators can rapidly decrease time needed for execution, allowing even these complex calculations to work in real time or even better. Finding its use in game or movie industry.

Klíčová slova

Procedurální, generování, CUDA, paralelizace, GPU, grafické alcelerátory, volumetrická data, terény, perlinův šum, marching cubes, supercomputing

Keywords

Procedural generating, CUDA, paralelization, GPU, graphics accelerators, volume data, terrains, perlin noise, marchnig cubes, supercomputing

Citace

Jan Ryba: Generování procedurálních terénů na GPU, semestrální projekt, Brno, FIT VUT v Brně, 2010

Generování komplexních procedurálních terénů na GPU

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Doc. Ing. Adama Herouta, Ph.D.

Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Jan Ryba
11.1.2010

Poděkování

Chtěl bych poděkovat vedoucímu své bakalářské práce Ing. Adamu Heroutovi, Ph. D. za jeho cenné rady, doporučení a odbornou pomoc, kterou mi poskytoval v průběhu řešení.

© Jan Ryba, 1010
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Obsah....1

1 Úvod....2

2 Generování terénů....3

2.1 Midpoint algoritmus....3

2.1.1 Generování výškové mapy diamod-square algoritmem....5

2.2 Perlinův šum....7

2.2.1 Způsoby interpolace šumových funkcí....11

3 Vizualizace prostorových dat....14

3.1 Trojúhelníky pro vizualizaci....14

3.2 Marching cubes....15

3.2.1 Generování trojúhelníků v objemové buňce....16

4 nVidia CUDA....18

4.1 Motivace....18

4.2 Model zařízení CUDA....20

4.2.1 Spuštění a organizace vláken....20

4.2.2 Paměťový model....21

4.2.3 Model běhu....22

4.3 Možnosti optimalizace a využití pamětí....23

4.3.1 Optimalizace instrukcí....23

4.3.2 Globální paměť....24

4.3.3 Paměť konstant....24

4.3.4 Paměť textury....24

4.3.5 Sdílená paměť....24

4.3.6 Registry....26

4.3.7 Počet vláken v bloku vláken a počet bloků vláken....26

4.3.8 Přenosy dat mezi host a device....26

4.3.9 Interoperabilita s OpenGL....27

5 Generování procedurálních terénů na GPU....27

5.1 Marching cubes na CUDA....27

5.2 Funkce pro popis terénu....29

6 Závěr....35

Literatura....36

Seznam příloh....37

1 Úvod

Generování komplexních plně prostorových terénů je velmi náročnou činností, ať už datově nebo výpočetně, případně obojí. Datovou náročnost můžeme značně omezit, nebo plně eliminovat použitím procedurálního přístupu, kdy však vyvstává problém výpočetní náročnosti. Většina dnešních metod ke generován terénů, využívá pouze systému výškových map, tedy deformace terénu v jedné ose. My se však snažíme o plně prostorový přístup, kdy je možno generovat převisy, tunely a jeskyně. A to za pomocí čistě matematického popisu, případně možnosti navázání na objemová data. Můžeme tak generovat různé variace terénů, nebo jiných prostorových útvarů, případně jednoduše přidávat detaily na již existující prostorová uskupení. Tato technika může mít široké využití v herním nebo filmovém průmyslu.

V dnešní době můžeme sledovat rychlý vývoj výpočetní výkonosti grafických akcelerátorů. Jejich výpočetní výkonnost předčí dnešní procesory primárně v oblasti výpočtů s plovoucí řádovou čárkou. Pro využití výpočetního výkonu grafických akcelerátorů za účelem obecných výpočtů vznikla CUDA (*Compute Unified Device Architecture*), umožnující využívat širokou škálu grafických akcelerátorů společnosti nVidia. CUDA nachází své uplatnění hlavně v oblasti náročných výpočetních aplikací a hromadného zpracování. Využití je možné všude, kde je možno výpočty značně paralelizovat. Přes svůj výpočetní výkon jsou grafické akcelerátory stále cenově dostupné. V různých pracovních stanicích, mohou být již instalovány ale jejich výpočetní výkon nemusí být využit.

Úvodní část (2) je věnována popisu požadavků a problematice procedurálního generování terénů s ukázkou dvou algoritmů: Midpoint algoritmu a Perlinova šumu. A metodám interpolace šumových hodnot.

Část (3) popisuje vizualizaci objemových (volumetrických) dat. Primárně je věnována generování trojúhelníků algoritmem Marching cubes.

V následující části (4) je popsána architektura CUDA, její význam, spouštění programů, použití a význam různých paměťový prvků a základní možnosti optimalizace.

Hlavní část práce (5) je tvořena popisem průběhu vývoje a výsledků experimentů při implementaci Marching cubes na CUDA a proč byla zvolena právě tato architektura. Zvláštní pozornost je věnována vývoji a možnostem funkce pro popis prostorového terénu.

2 Generování terénů

V této kapitole se zaměříme na různé přístupy ke generování terénů. Prvním základním přístupem bude mid-point algoritmus. Následovaný Perlinovím šumem. Naším cílem je dosáhnout přirozeného terénu s vlastnostmi *fraktálu*[15]. Tedy takový terén, který je v jistém smyslu sám sobě podobný v různých měřítkách.

Příkladem fraktálního systému může být například lidský kardiovaskulární systém. Obsahuje stejný vzor od největších artérií, přes žíly, až po nejmenší kapilární žilky. Vezměme si kupříkladu kouli jako příklad ne-fraktálního systému. Při značném přiblížení přestává být vnímána jako koule, ale podobá se spíše rovině. Země se při pohledu z povrchu nejeví příliš jako kulová plocha. Kouli proto nejlépe popisuje Eukleidovská geometrie a ne fraktální systém.

Fraktální ovšem matematicky rigorózně znamená ne sobě podobný ale stejný v různých měřítkách. Nejznámějším příkladem fraktálu je *Mandelbrotova množina*[9]. V přírodě se ovšem takovéto fraktály příliš nevyskytuj, naším cílem tedy bude dosáhnout spíše pseudo-fraktálního systému.

Pro generování terénů ovšem neexistují žádná objektivní měřítka kvality, vždy záleží na subjektivním vjemu pozorovatele.

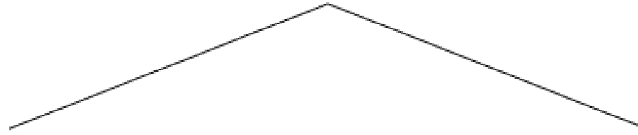
2.1 Midpoint algoritmus

Většina textu v této kapitole byla převzata z [6].

Pro nastínění problematiky začneme popisem algoritmu pro 1D. Který se hodí spíše pro zobrazení hor na horizontu, než generování terénů. Základ algoritmu můžeme popsat takto:

```
začni s jednou horizontální úsečkou
opakuj po dostatečný počet kroků
  { opakuj pro každou úsečku scény
     { najdi střed úsečky
       rozděl úsečku
       střed posuň v ose Y o náhodnou hodnotu
     }
    sniž rozsah pro náhodná čísla
  }
```

Podívejme se na příklad. Začneme s úsečkou jdoucí v souřadnici ose X od -1 do +1, v obou případech se souřadnicí v ose Y rovnu 0. Nastavme generátor čísel na rozsah od -1 do +1. Provedení prvního kroku algoritmu, tak vypadá jako na Obr. 2.1.



Obr. 2.1[6]: První krok Midpoint algoritmu

Při druhém průchodu cyklem máme už dva segmenty, každý samozřejmě poloviční velikosti původní úsečky. Snižme rozsah pro generování náhodných čísel na -0,5 až +0,5. Celý proces opakujeme pro dva středy segmentů z předchozího kroku. Dostaneme výsledek podobný Obr. 2.2. V následujícím kroku dostaneme už 4 segmenty, snížíme rozsah náhodných čísel na -0.25 až +0,25 a výsledek bude jako na Obr. 2.3. Další pokračování může vyprodukovat až Obr. 2.4. Výhodou je navazování, jelikož okrajové body jsou ve shodné souřadnici na ose Y.

Obr. 2.2[6]: Druhý krok algoritmu

Obr. 2.3[6]: Třetí krok algoritmu

Obr. 2.4[6]: Výsledek po dostatečném množství kroků

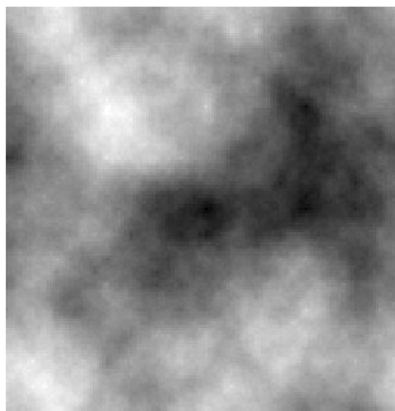
Můžeme si všimnou, že celý proces je iterativní. Dává nám to možnost generovat prakticky do nekonečna, což by odpovídalo definici fraktálu. Algoritmus je nesmírně jednoduchý přitom umožňuje generovat komplexní výstupy bohaté na detaily. Na základě těchto zjištění, že jen několik málo instrukcí dokáže generovat komplexní obrazy, vznikl obor s názvem *Fraktální obrazová komprese*. Jehož myšlenkou je ukládání jednoduchých rekurzivních procedur a ne samotného obrazu. Teoreticky tento přístup lze použít i pro zakódování požadovaného terénu, aby výsledek neměl úplně náhodnou formu.

Otázkou je jak moc snižovat rozsah pro výběr náhodných čísel. Vždy záleží jak „drsně“ vypadající fraktál chceme generovat. V případě, že rozsah nebudeme redukovat, dostaneme výsledek s mnoha ostrými skoky, v opačném případě může být výsledek příliš hladký. Pojem „drsnost“ můžeme vyjádřit konstantou přímo úměrnou snižování rozsahu a nepřímo úměrnou míře „drsnosti“. Konstantu pojmu drsnost označme písmenem **H**, tato udává jakou mírou se sníží rozsah vždy v následujícím kroku. Míru snížení rozsahu v určitém kroku **k** můžeme vypočítat jako $h_{(k)}=H^k$ chceme-li zapsat rekurzivně $h_{(k+1)}=h_{(k)}*H$, $h_{(1)}=H$. Obr. 2.5 zobrazuje vygenerované průběhy pro různé hodnoty **H**.

Obr. 2.5[6]: Průběhy pro různé hodnoty H

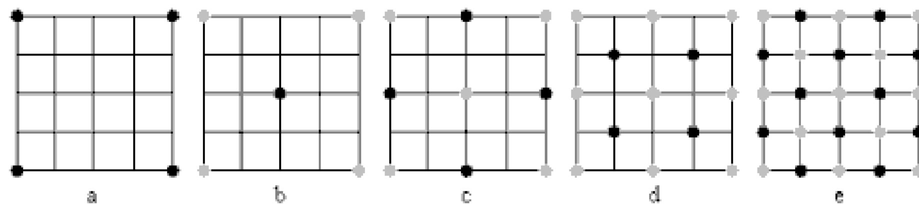
2.1.1 Generování výškové mapy diamod-square algoritmem

Pro generování terénů musíme algoritmus převést do prostoru, kdy výsledek lze chápat jako diskrétní funkci *I(x,y)* dvou souřadnic x a y s funkční hodnotou udávající výšku terénu v daném bodě. Představíme-li si tento systém jako obraz, kde z složku transformujeme do světlosti dostáváme výsledek jako na Obr. 2.6.



Obr. 2.6[6]: Znázornění výškové mapy

Algoritmus začíná na dvou rozměrném diskrétním systému o rozměrech $x, y=2^{I+2}+1$, kde **I** značí počet provedených iterací. Ukažme si průběh algoritmu pro hodnoty *x,y=5*.

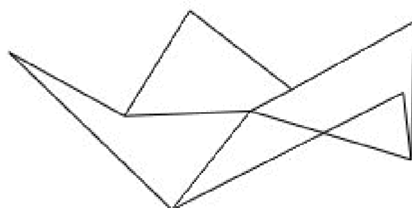


Obr. 2.7[6]: Ukázka průběhu dimond-sqare algoritmu na poli 5x5

Celý iterativní proces sestává ze dvou základních kroků:

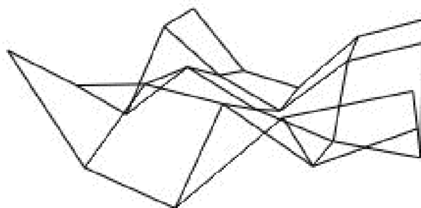
1. **Diamantový krok** – Bere 4 hodnoty z rohů čtverce. V bodě, kde se setkávají diagonály je vygenerována náhodná hodnota a přičte se průměrná hodnota ze 4 hodnot v rozích čtverce
2. **čtvercový krok** – Doplňuje body do pravidelné čtvercové mřížky. Při zachování stejného postupu při generování průměru a každému bodu přičteme náhodnou hodnotu a průměr z okolních bodů.

Přejděme zpět k Obr. 2.7 a popišme si průběh algoritmu. První krokem je nastavení hodnot v rozích (Obr. 2.7(a)). Provedeme diamantový krok s rozsahem pro náhodná čísla -1 až +1 nová hodnota se na Obr. 2.7(b) zobrazuje jako černý bod. Pro čtvercový krok použijeme stejný rozsah náhodných hodnot. Nově spočtené hodnoty jsou zobrazeny černou barvou na Obr. 2.7(c). Tímto máme hotovu první iteraci algoritmu a náš terén by mohl vypadat jako na Obr 2.8.



Obr. 2.8[6]: Výsledek po první iteraci

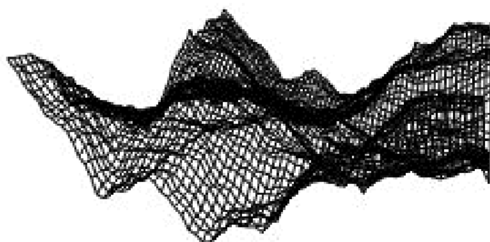
V druhé iteraci opět začínáme diamantovým krokem, rozdílem je, že začínáme se čtyřmi čtverci namísto jednoho a snižujeme rozsah pro generování náhodných čísel podle výše definovaného



Obr. 2.9[6]: Výsledek po druhé iteraci

H. Diamantový krok vygeneruje 4 nové hodnoty viz. Obr. 2.7(d) a následný čtvercový krok přidá 12 nových hodnot viz. Obr. 2.7(e). Celkově tak dostáváme 25 elementů a terén může vypadat jako na

Obr. 2.9. Kdybychom měli na začátku větší pole a pokračovali v iterování, například po pěti iteracích dostaneme terén podobný Obr. 2.10. Takto vygenerovaný terén na sebe ovšem nenavazuje. Návaznost můžeme zařídit když při generovaní bodu na okraji, zkopírujeme hodnotu bodu i na opačnou stranu.



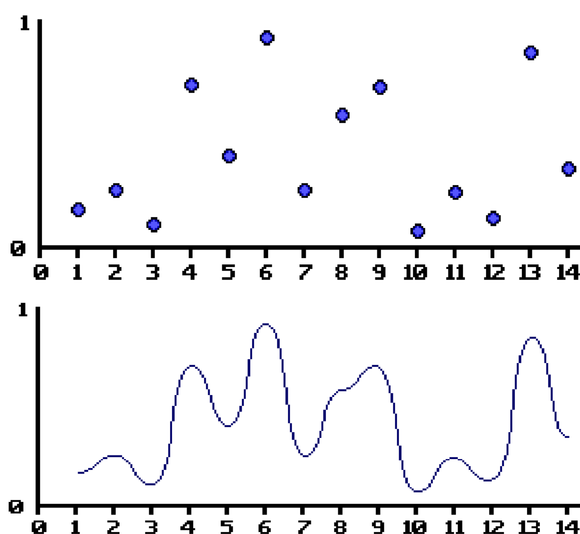
Obr. 2.10[6]: Výsledek po pěti iteracích.

2.2 Perlinův šum

Většina textu a obrázků v této kapitole převzata z [14] a [7].

Perlinův šum generuje výstup podobně jako předchozí algoritmus na základě náhodných čísel. Samotná vygenerovaná posloupnost čísle je ovšem příliš „drsná“ co se týká vizuálního vjemu. Proto se Perlinův šum skládá z více šumových funkcí s různými měřítky. Základem je šumová funkce generující šum a interpolační funkce.

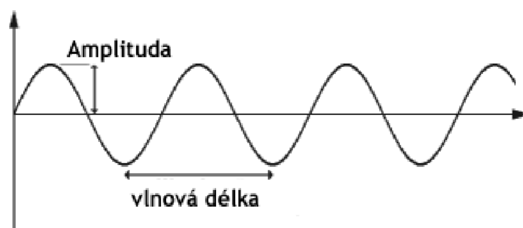
Šumová funkce je generátor pseudonáhodných čísel s iniciační hodnotou a vstupní hodnotou, na základě těchto dvou hodnot vygeneruje číslo. Pokud Dáme funkci stejné parametry znovu, vygeneruje shodný výsledek, což je nesmírně důležité. Dovoluje nám to produkovat shodné výsledky napříč spuštěními programu a i napříč spektrem HW pokud dodržují určitý standard. Šumovou funkci



Obr. 2.11[14]: Výstup šumové funkce (nahoře) a interpolované hodnoty (dole)

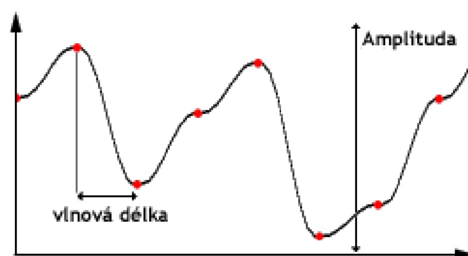
můžeme definovat jako diskrétní funkci $Noise(x, seed); x, seed \in \mathbb{Z}$ kde x je vstupní hodnota a *seed* iniciační hodnota, která se ovšem obvykle neuvádí, jelikož předpokládáme, že je stále stejná. Iniciačních hodnot může být i více.

Pro další pokračování si definujme pojmy **amplituda** a **frekvence**. U funkce Sinus je vlnová délka vzdálenost mezi opakujícími se body funkce a amplituda je rovna maximální hodnotě, kterou je funkce schopna vygenerovat viz. Obr. 2.12. Frekvenci definujeme jako $\frac{1}{amplituda}$. U šumové



Obr. 2.12[14]: Graf funkce Sinus

funkce definujeme vlnovou délku, jako vzdálenost mezi diskrétně vygenerovanými body a amplitudu jako absolutní rozdíl maximálně vygenerovatelných hodnot, viz Obr. 2.13. Frekvenci spočteme jako v předchozím případě. Podobně jako u goniometrických funkcí bude mít šumová funkce s nižší frekvencí opticky hladší průběh než s frekvencí vyšší.

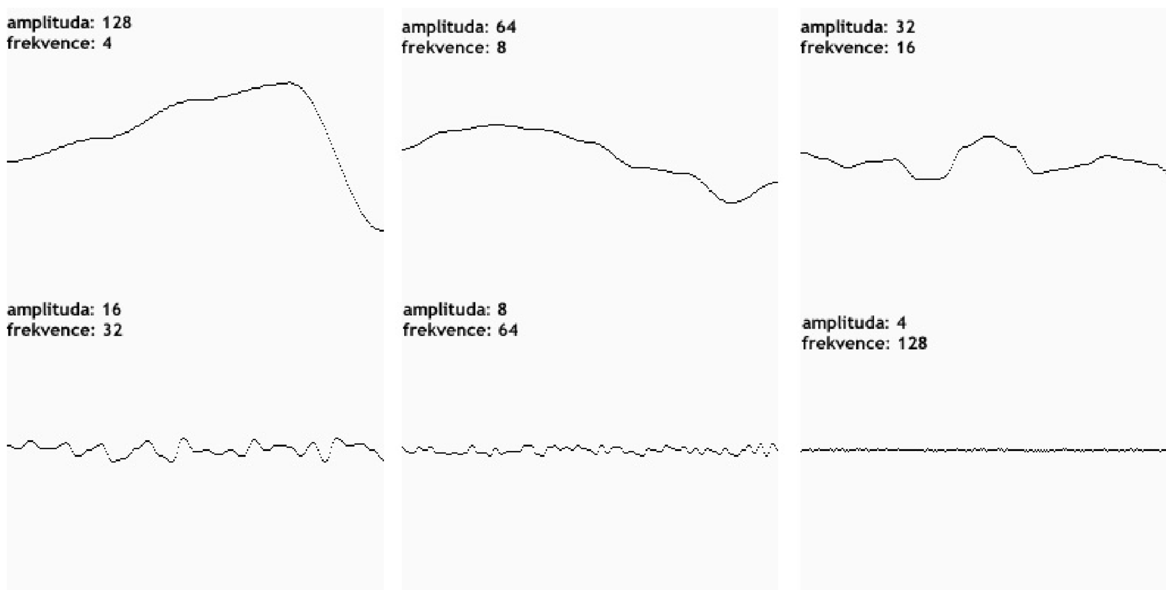


Obr. 2.13[14]: Graf šumové funkce

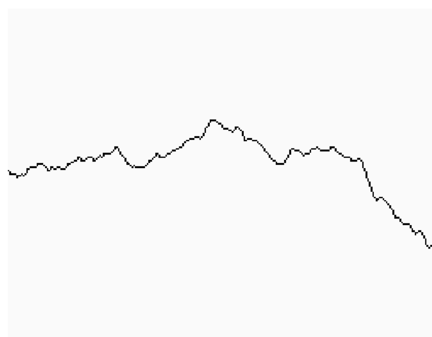
Základem Perlinova šumu je sumace (součet) několika šumových funkcí lišících se v amplitudě a frekvenci. Pro dosažení spojitosti, jsou hodnoty šumových funkcí interpolovány. Samotná šumová funkce by již sama o sobě stačila na vytvoření některých přirozeně vypadajících jevů, ale snažíme se dosáhnout fraktálního vzezření, jak bylo definováno výše. Perlinův šum tedy definujeme následovně (*N* odpovídá počtu šumových funkcí, které chceme použít):

$$f(x) = \sum_{t=0}^{N-1} \frac{1}{2^t} * Noise(2^t * x) \qquad (2.1)$$

Pokud vezmeme několik interpolovaných šumových funkcí o určitých amplitudách a frekvencích viz. Obr. 2.14. Jejich složením dostaneme výsledný průběh Perlinova šumu Obr. 2.15.



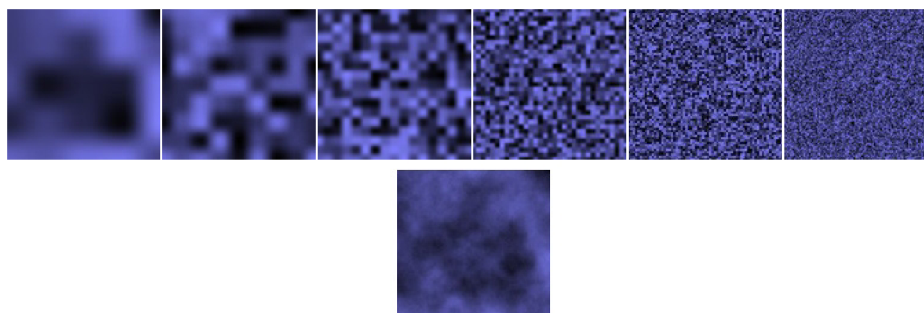
Obr. 2.14[14]: průběhy interpolovaných šumových funkcí o různých amplitudách a frekvencích, tak jak se objevují ve vzorci (2.1)



Obr. 2.15[14]: výsledný průběh Perlinova šumu podle vzorce (2.1)

Perlinův šum můžeme vytvořit i v 2D. Kde definujeme šumovou funkci *Noise2D(x,y)* například jako ve vzorci (2.2). Konstanta 51 může být libovolné prvočíslo. Můžeme tak získat výsledek jako na Obr 2.16.

$$Noise2D(x, y) = Noise(x + y * 51) \tag{2.2}$$



Obr. 2.16[14]: Ukázka šumových funkcí ve 2D (nahoře) a jejich výsledného sečtení (dole)

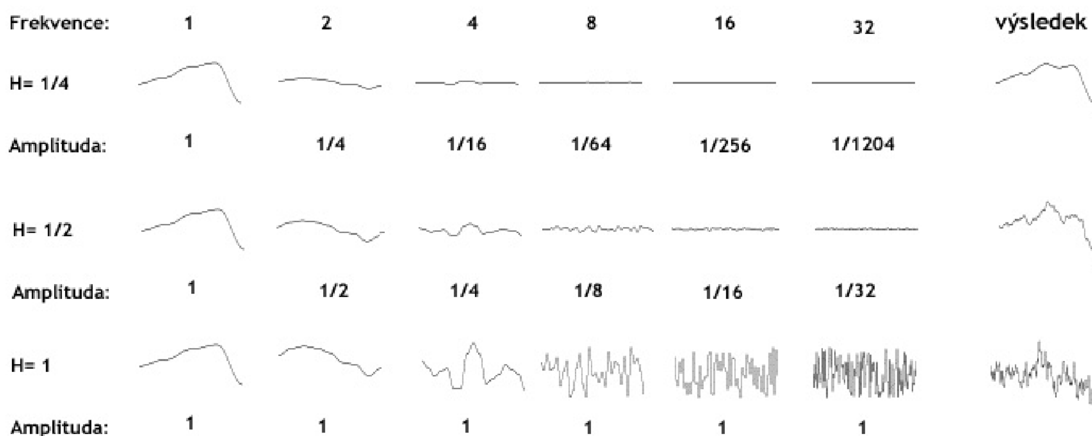
Podobně jako u Midpoint algoritmu můžeme definovat míru „drsnosti“ **H**. Často je označována jako *persistence*. Pojem persistence definoval Mandelbrot[] pro určení obsahu vysokých frekvencí ve fraktálním šumu, jeho definice považuje šum s nízkou persistencí za šum s velkým počtem počtem vysokých frekvencí. My však budeme uvažovat opačnou definici, tedy **H** přímo úměrné obsahu vysokých frekvencí. Pro každou dílčí šumovou funkci jednoduchého Perlinova šumu definujeme frekvenci a amplitudu následovně (*t* odpovídá pořadovému číslu šumové funkce jako ve vzorci (2.1)):

$$\begin{aligned} frekvence &= 2^t \\ amplituda &= H^t \end{aligned} \tag{2.2}$$

Pokud tedy použijeme persistenci jako určující faktor, stačí nám k nastavení Perlinova šumu znát jen její hodnotu a počet šumových funkcí (*oktáv*).

$$f(x) = \sum_{t=0}^{N-1} H^t * Noise(2^t * x) \tag{2.3}$$

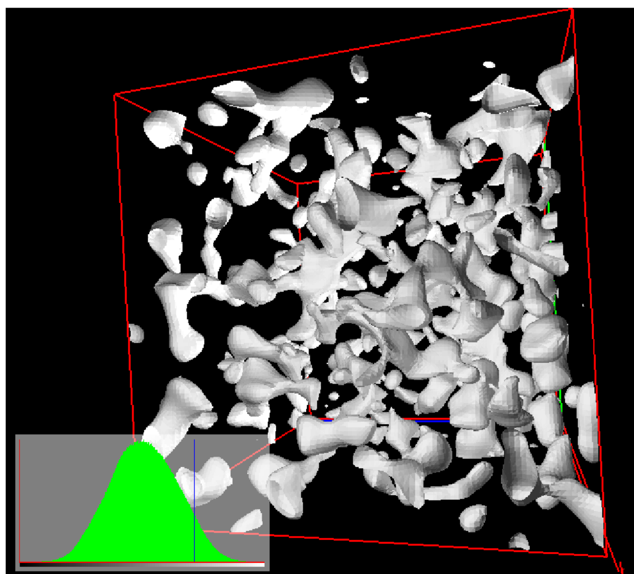
Na Obr. 2.17 můžeme vidět jak ovlivňuje hodnota persistence **H** výsledný vzhled.



Obr. 2.17[14]: Znázornění ovlivnění výsledného vzhledu hodnotou persistence

Každá následující šumová funkce je nazývána oktávou, díky podobnosti s pojmem oktáva používaným v hudební nauce[]. Každá funkce pracuje s dvojnásobnou frekvencí oproti předchozí. Počet oktáv záleží pouze na případu užití. Nemusíme se striktně držet návrhu na dvojnásobnou frekvenci při každé další šumové funkci, zajímavých variací můžeme dosáhnout i s jinými rozestupy. Míra nejvyšší frekvence ovšem obvykle závisí na vzorkování výstupu, jelikož výstup požadujeme obvykle diskrétní, tak nejvyšší frekvence by neměla přesáhnout vzorkovací frekvenci.

Perlinův šum můžeme rozšířit i do prostoru a dalších dimenzí. Problémem ovšem je vizualizace. U prostorového vyjádření dostáváme spojitý 3D signál, pro vizualizaci je tak nutno použít některý z prostředků zobrazování volumetrických dat. Příklad jedné oktávy Perlinova šumu můžeme vidět na Obr. 2.18.



Obr. 2.18[7]: Jedna oktáva Perlinova šumu v prostoru

2.2.1 Způsoby interpolace šumových funkcí

Většina texů a obrázků v této kapitolo převzata z [8], [13], [14] a [2]

Interpolace je matematický proces, kdy z určité diskrétní množiny pomocí aproximace dopočítáme ostatní body funkce. Jelikož je šumová funkce v základu diskrétní, musíme ji rozšířit na spojitou. Existuje několik typů interpolace, které se liší mírou aproximace a výpočetní složitostí.

Lineární interpolace

Je nejméně výpočetně náročným typem interpolace. Metoda je založena na doplnění množiny hodnot funkce za pomocí lineárních mnoho členů. Mezi dvěma sousedními body položíme přímku, pak této přímce říkáme lineární interpolace. Rovnici pro lineární interpolaci získáme z obecné rovnice přímky (2.4) určené body $v_1=[x_1, y_1]; v_2=[x_2, y_2]$ vyjádřením *y* dostáváme rovnici Lineární interpolace (2.5).

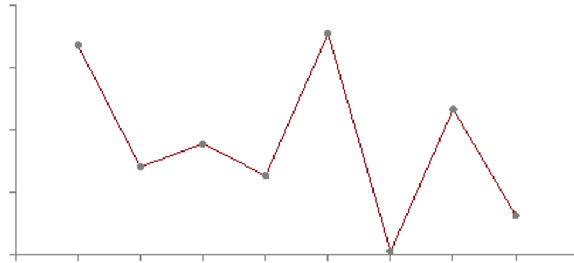
$$\frac{y-y_1}{x-x_1}=\frac{y_2-y_1}{x_2-x_1} \tag{2.4}$$

$$I_{Linear}(v1, v2)=y_1+\frac{(x-x_1)*y_2-y_1}{x_2-x_1} \tag{2.5}$$

Pro náš případ Perlinova šumu je vzdálenost v ose *x* vždy rovna jedné díky této vlastnosti můžeme rovnici zjednodušit. V rovnici (2.6) hodnoty v_1 a v_2 odpovídají hodnotám z šumové funkce a $x \in \langle 0,1 \rangle$ určuje pozici mezi těmito body.

$$I_{linear}(v_1, v_2, x)=v_1*(1-x)+v_2*x \tag{2.6}$$

Velkou nevýhodou této interpolace je její velká nepřesnost. Přechody mezi jednotlivými intepolanty jsou ostré, což způsobuje nespojitost už v první derivaci. Graf interpolovaných hodnot máme zobrazen na Obr. 2.19.

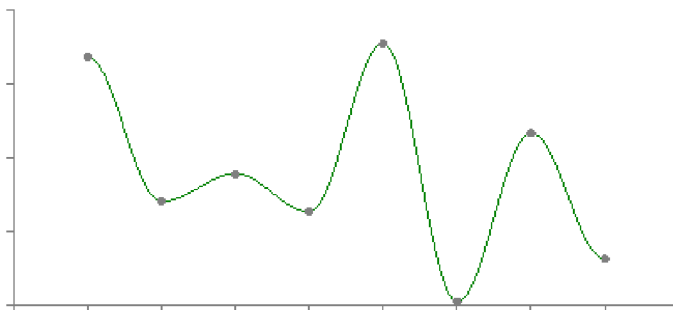


Obr. 2.19[8]: Graf hodnot interpolovaných pomocí Lineární intepolace

Kosinová interpolace

Je vylepšením Lineární interpolace. Metoda je výpočetně náročnější, ale řeší problém nespojitosti v první derivaci, nevýhodou může být, že v každém s interpolovaných bodů je derivace rovna nule. Jedná se o jednoduché řešení založené na mapování části funkce kosinus. Po drobné úpravě rovnice (2.5) dostáváme rovnici (2.6).

$$F(x)=(1-\cos(x*\pi))*0{,}5$$
$$I_{cosine}(v_1, v_2, x)=v_1*(1-F(x))+v_2*F(x) \tag{2.6}$$



Obr. 2.20[8]: graf hodnot interpolovaných pomocí Kosinové interpolace

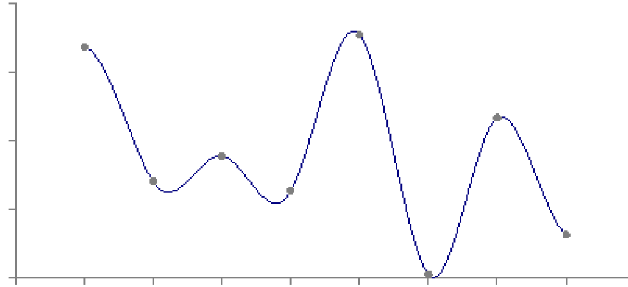
Kubická interpolace

Je nejjednodušší metodou poskytující pravou kontinuitu signálu. Na rozdíl od předchozích vyžaduje metoda 4 body z okolí hledané funkční hodnoty. Využívá několik polynomů nižšího řádu *k* mezi každými dvěma body. Tyto polynomy pak kombinuje, tak aby vytvořili hladkou interpolační křivku, kde je v referenčních bodech zajištěna spojitost až do (*k-1*) derivace. Existují dva základní typy spojitosti pro kubickou interpolaci. Pro spojitost C1 je nutné, aby tečné vektory v místě napojení byli stejné, C2 pak požaduje aby i derivace těchto vektorů byli shodné. Pro vytvoření oblouku mezi dvěma body jsou tak nutné další dva krajní body. Pokud tedy chceme interpolovat mezi body v_1 a v_2

potřebujeme ještě bod před v_0 a za v_3. Pro zjednodušení rovnice (2.8) Kubické interpolace definujeme pomocné hodnoty P,Q,R,S viz. (2.7).

$$\begin{aligned} P &= (v_3 - v_v) - (v_0 - v_1) \\ Q &= (v_0 - v_1) - P \\ R &= v_2 - v_0 \\ S &= v_1 \end{aligned} \tag{2.7}$$

$$I_{cubic}(v_0, v_1, v_2, v_3) = P * x^3 + Q * x^2 + R * x + S \tag{2.8}$$



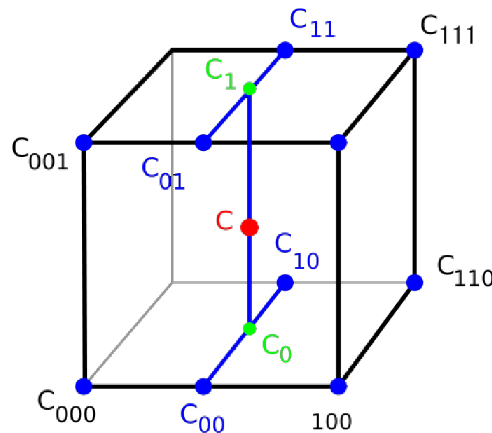
Obr. 2.21[8]: graf hodnot interpolovaných Kubickou interpolací

Trilineární interpolace

Je rozšířením Lineární interpolace do prostoru. Operuje na diskrétní množině skalárních hodnot vygenerovaných šumovou funkcí hodnot kterou můžeme definovat jako (2.9) vracející hodnotu z množiny.

$$Noise3D(x, y, z); x, y, z \in \mathbb{Z} \tag{2.9}$$

Základem je získání 8 hodnot z prostoru v blízkosti bodu pro který chceme provádět interpolaci. Jak můžeme vidět na Obr. 2.22 jedná se o soustavu Lineárních interpolací. Pro výpočet si definujme (2.11) a pomocné rovnice (2.10). korespondence s Obr. 2.22 je na základě $C_{xyz} = Noise3D(x, y, z)$.



Obr. 2.22[13]: zobrazení výpočtu Trilineární interpolace

$$x_d = x - |x|$$
$$y_d = y - |y|$$
$$z_d = z - |z|$$

$$\begin{aligned} C_{00} &= C_{000} * (1 - x_d) + C_{100} * x_d \\ C_{01} &= C_{001} * (1 - x_d) + C_{101} * x_d \\ C_{10} &= C_{010} * (1 - x_d) + C_{110} * x_d \\ C_{11} &= C_{011} * (1 - x_d) + C_{111} * x_d \end{aligned} \qquad [2.10]$$

$$C_0 = C_{00} * (1 - y_d) + C_{10} * y_d$$
$$C_1 = C_{01} * (1 - y_d) + C_{11} * y_d$$

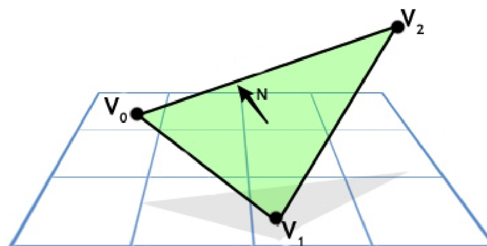
$$C = C_0 * (1 - z_d) + C_0 * z_d$$
$$I_{linear3D}(x, y, z) = C ; x, y, z \in \mathbb{R} \qquad [2.11]$$

3 Vizualizace prostorových dat

Snažíme se o generovaní plně prostorových terénů, k čemuž nám nepostačují pouze 2D výškové mapy. U prostorových (*objemových či volumetrických*) dat je však problém s vizualizací. Musíme tak data převést do jiné lépe prostorově vizualizovatelné formy, jako například převod na polygony, nebo trojúhelníky, které můžeme snadno zobrazovat pomocí grafických akcelerátorů. Toho docílíme například pomocí algoritmu Marching cubes.

3.1 Trojúhelníky pro vizualizaci

Základním prvkem v prostorové vizualizaci je trojúhelník definovaný třemi vrcholy (*vetexy*) $V_0, V_1, V_2 \in \mathbb{R}^3$. Definujeme si pojem *normálový vektor* nebo-li *normála,* která je kolmá na rovinu trojúhelníku. Určuje orientaci trojúhelníku a rozděluje trojúhelník na přivrácenou a odvrácenou stranu z pohledu pozorovatele. Zjednodušeně můžeme říci, že pokud jsou vrcholy seřazeny v protisměru hodinových ručiček, pak je trojúhelník přivrácen, v opačném případě odvrácen.



Obr. 3.1: Trojúhelník v prostoru se zobrazenými vrcholy a normálovým vektorem

Normálový vektor **N** můžeme vypočítat pomocí vzorce (3.1), kdy vytvoříme dva vektory z dvou hran trojúhelníku a hledáme vektor na oba kolmý. Využití nalézá například při výpočtu osvětlení a normálový vektor můžeme definovat v každém vrcholu trojúhelníku, kde je pak možné normálové vektory interpolovat na ploše trojúhelníku.

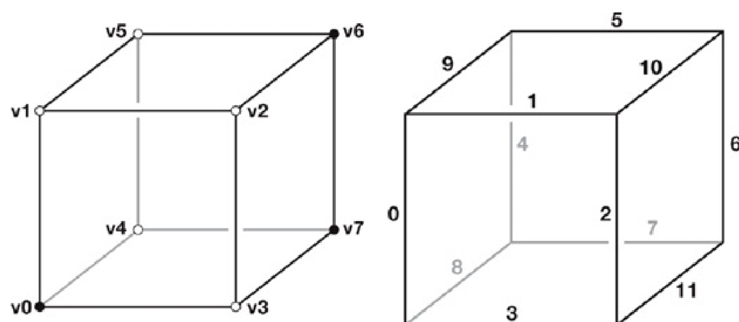
$$
\begin{aligned}
&E_1 = V_1 - V_0 ; E_1 \in \mathbb{R}^3 \\
&E_2 = V_2 - V_0 ; E_2 \in \mathbb{R}^3 \\
&N = E_1 \times E_2 ; N \in \mathbb{R}^3 \\
&(\times \textit{ je operace vektorového součinu})
\end{aligned}
\tag{3.1}
$$

3.2 Marching cubes

Většina textů a obrázků převzata z [2] a [5].

Kombinuje jednoduchost a nízkou výpočetní náročnost, díky tomu že je postaven na vyhledávání v tabulkách. Je použitelný například na vizualizaci prostorových medicínských dat z magnetické rezonance uspořádaných do regulární mřížky, kdy data korespondují s vrcholy objemového tělesa, které používáme v algoritmu. Metoda podporuje i vizualizaci matematicky definovaných prostorových funkcí. V tomto případě je funkce definována pro všechny body $\mathbb{R}^3$ ale vzorkujeme ji pouze ve vrcholech objemového tělesa (krychle) použitého v algoritmu.

Základním problémem je aproximace *implicitní plochy*, napříč diskrétní prostorovou množinou skalárních hodnot. Implicitní plochou se myslí všechny body v prostoru, kde je funkční hodnota rovna nule, nebo jiné hodnotě již zvolíme, jak u matematicky definovaných objemových těles, tak i u objemových dat. Marching cubes umožňuje korektní generování trojúhelníků aproximujících implicitní plochu v jedné diskrétní prostorové buňce s pomocí hodnot z osmi vrcholů. Výstupem je žádný až čtyři trojúhelníky. Pokud například ve všech vrcholech je kladná hodnota, je jasné že buňka je mimo implicitní plochy, a nevygeneruje se žádný trojúhelník. V ostatních případech kdy znaménka ve vrcholech liší, je jasné, že implicitní plocha protíná buňku a generujeme jeden až čtyři trojúhelníky.



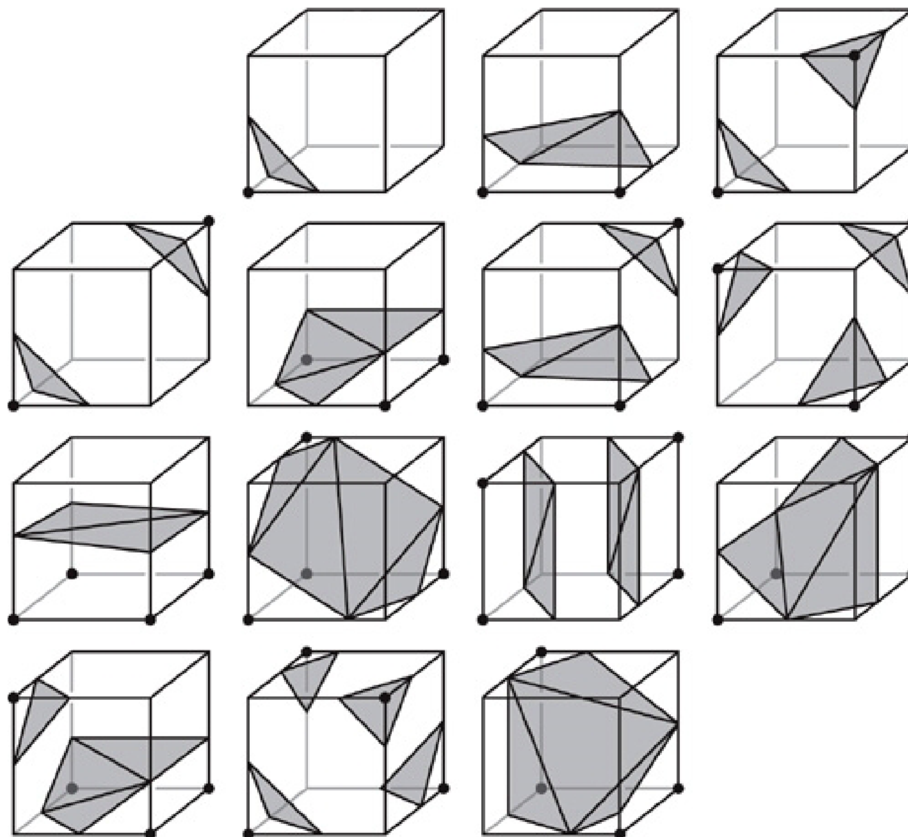
Obr. 3.2[2]: Označení vrcholů a hran objemové buňky

3.2.1 Generování trojúhelníků v objemové buňce

Uvažujme označení vrcholů a hran objemové buňky jako na Obr. 3.2. Jak již bylo řečeno každý vrchol obsahuje jednu skalární hodnotu, buďto pozitivní nebo negativní. Z každé hodnoty učiníme jeden bit a provedeme konkatenaci, podle vzorce (2.2). Pokud je hodnota záporná, nastavíme bit na na hodnotu logická 0, je-li pozitivní nastavíme bit na logickou 1. Konečným výsledkem je osmi bitové číslo, tedy číslo v rozsahu 0-255. V případě hodnot 0 a 255 je buňka kompletně mimo implicitní plochy a žádné trojúhelníky nejsou vygenerovány. V ostatních případech bude vygenerováno určité množství trojúhelníků.

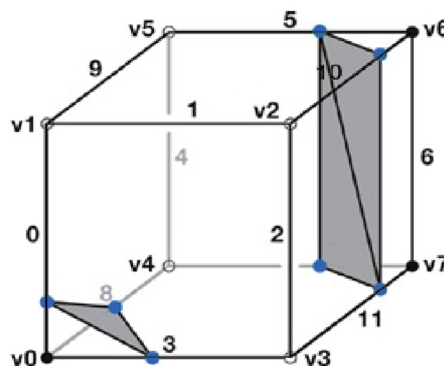
$$T = v7|v6|v5|v4|v3|v2|v1|v0 \tag{2.2}$$

Číslo **T** použijeme pro vyhledávání tabulce okrajů, která vrací 12-bitovou hodnotu, kde každý bit koresponduje s jednou hranou objemové buňky, je-li hodnota rovna log. 1, pak je hrana protnuta implicitní plochou. Vyhledávací tabulku můžeme zjednodušit na tabulku počtu generovaných trojúhelníků, která obsahuje počty trojúhelníků, které se budou generovat pro různé případy **T**. Každý trojúhelník je tvořen propojením tří bodů ležících na třech různých hranách objemové buňky. Obr. 3.3 zobrazuje různé možnosti generování trojúhelníků v objemové buňce (vyznačené vrcholy udávají kladnou hodnotu, nevyznačené naopak).



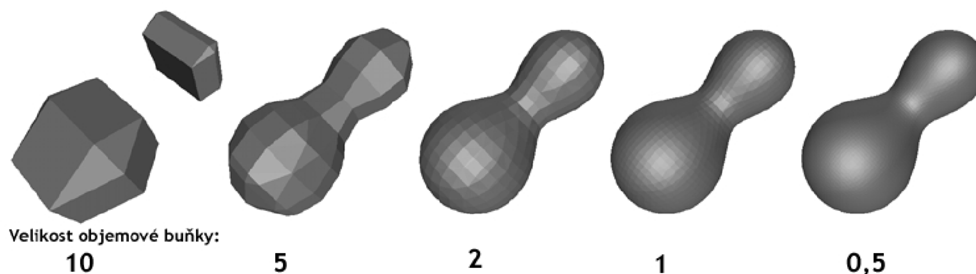
Obr. 3.3[2]: Zobrazení různých možností generování trojúhelníků v objemové buňce

Kde přesně je umístěn bod na hraně zjistíme interpolací. Bod se nalézá v místě na hraně, kde je funkční hodnota rovna nule. Máme-li například hodnotu ve vrcholu *v0* rovnu 0,1 a ve vrcholu *v3* rovnu -0,3, bude bod umístěn na hraně 3 zhruba ve vzdálenosti 25% z celkové délky hrany od bodu *v0*. Soustavy hran pro generování trojúhelníků určuje tabulka pro generování trojúhelníků (*triTable*). Tabulka obsahuje pro každou hodnotu **T** 16 čísel označujících hrany, na kterých jsou umístěny body pro spojování trojúhelníků. Každé 3 po sobě jdoucí hodnoty generují jeden trojúhelník. Samozřejmě ne všechny hodnoty musí být využity. Mějme jako příklad situaci na Obr. 2.4. Hodnota **T** jest 193. Z tabulky počtu generovaných trojúhelníků zjistíme hodnou 3 a *triTable* nám dává údaje (11,5,10,11,7,5,8,3,0,-1,-1,-1,-1,-1,-1). Z čehož vyplývá, že první trojúhelník vznikne spojením bodů na hranách 11, 5 a 10, druhý spojením bodů na hranách 11,7,5 a třetí spojením 8,3,0, jelikož máme generovat jen 3 trojúhelníky, zbylé hodnoty můžeme ignorovat. Seznam hodnot v tabulkách nalezneme například v [5].



Obr. 2.4[2]: Situace pro T=193

Důležitým aspektem je rozlišení mřížky, při příliš malém rozlišení (velký rozměr objemových buněk) může docházet k aliasingu[10] nebo k ne příliš kvalitnímu zobrazení. Spojíme-li si to s šumovou funkcí neměli bychom generovat šum o vyšších frekvencích než je frekvence rozmístění objemových buněk. Některé případy vzorkování matematické funkce v prostoru uvádí Obr. 3.4.

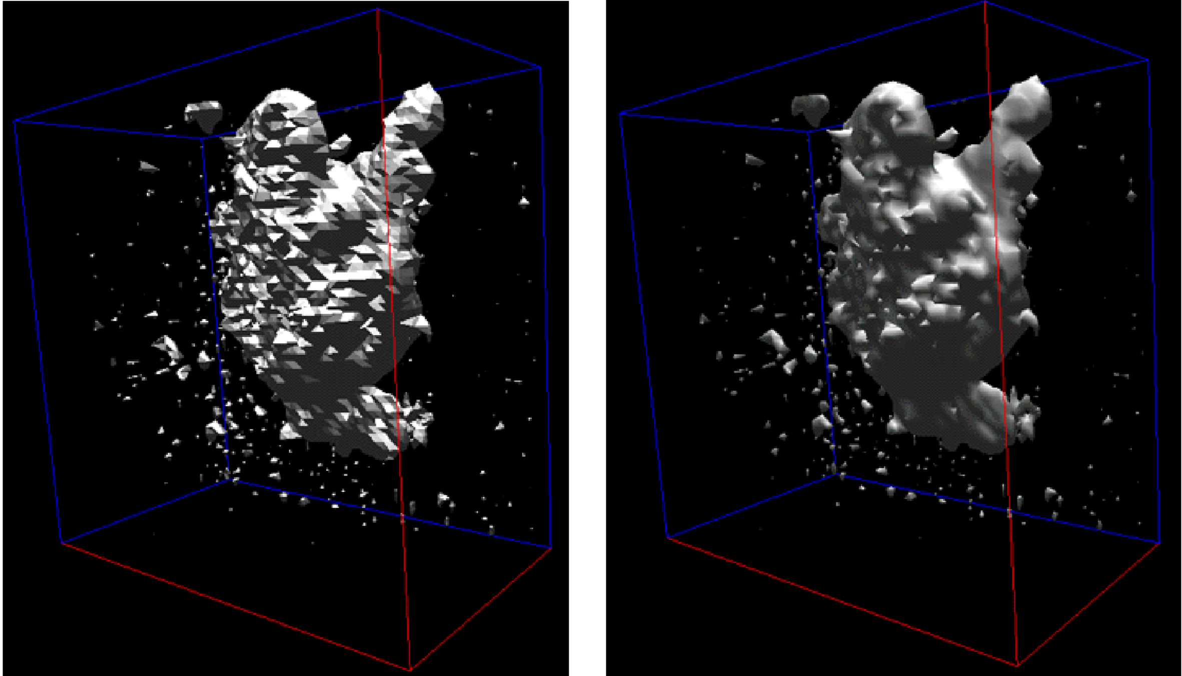


Obr. 3.5[5]: Různé případy vzorkování matematické funkce v algoritmu Marching cubes

Pro lepší zobrazování můžeme definovat normálový vektor v každém vrcholu generovaného trojúhelníku, na rozdíl od použiti jednoho normálového vektoru pro trojúhelník jako v kapitole 3.1. Základním přístupem je definování normálového vektoru v každém vrcholu objemové buňky a jeho

lineární interpolace v bodě generování trojúhelníku. Normálový vektor ve vrcholu je aproximací gradientu použité funkce, např. Perlinova šumu. Normálový vektor definujeme jako ve vzorci (3.3)

$$N(x,y,z)=\left(\frac{\partial f(x,y,z)}{\partial x},\frac{\partial f(x,y,z)}{\partial y},\frac{\partial f(x,y,z)}{\partial z}\right);N\in\mathbb{R}^3 \quad (3.3)$$



Obr. 3.5[5]: zobrazení bez výpočtu normálových vektorů ve vrcholech (vlevo) a s (vpravo)

4 nVidia CUDA

CUDA je zkratkou pro *Copute Unified Device Architecture.* Jedná se o novou hardwarovou a softwarovou architekturu pro vykonávání a správu výpočtů na grafických akcelerátorech bez nutnosti programovat výpočty grafickými API. CUDA je dostupná pro širokou škálu grafických akcelerátorů od firmy nVidia a také pro řadu operačních systémů jako Windows, Linux a Mac OS.

Grafickým akcelerátorem rozumíme kompletní hardwarovou sestavu GPU (*Graphic Processing Unit, neboli grafický procesor*) a RAM integrovanou na desku grafického akcelerátoru.

Tato kapitola vychází z oficiální dokumentace [11] a [12]

4.1 Motivace

V průběhu několika let se programovatelné grafické akcelerátory vyvinuly ve velmi výpočetně výkonné jednotky, jak je ilustrováno na Obr. 3.1, poskytující velké možnosti pro grafické a negrafické operace.

Obr. 4.1[11]: Graf srovnání výpočetního výkonu GPU a CPU

Hlavním důvodem takové evoluce je, že GPU je specializováno pro náročné a vysoce paralelní výpočty, jako je vykreslování 3D grafiky. Více tranzistorů na čipu je věnováno na výpočetní jednotky, než na *cache (rychlá vyrovnávací paměť)* a řízení, jak je znázorněno na Obr.4.2.

Obr. 4.2[11]: Ilustrace přidělení více prostoru na čipu výpočetním jednotkám.

GPU je dobře vybaveno pro řešení problémů specifikovatelných jako datově-paralelní výpočty s vysokou výpočetní intenzitou (vysokým poměrem výpočetních operací k paměťovým operacím). Stejný program může být prováděn na jednotlivých datových elementech současně, čímž jsou sníženy nároky na řízení. Díky paralelnímu provádění s vysokou výpočetní intenzitou se může zpoždění při provádění paměťových operací překrývat s časem provádění aritmetických operací. Vyhneme se tak použití velkých pamětí cache.

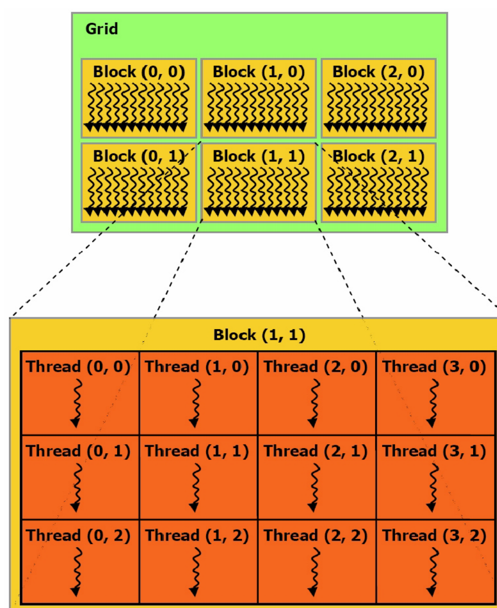
Dá se říci, že v této architektuře může být budoucnost „supercomputingu“. Společnost nVidia vyrábí i grafické akcelerátory značené TESLA, určené výhradně pro účely obecných výpočtů. Jak bylo proneseno na jedné z přednášek společnosti nVidia: „Vize budoucnosti je taková, že CPU bude využíváno spíše pro řízení celého systému, a hlavní výpočetní síla bude v grafických akcelerátorech“.

Hlavním účelem CUDA je dát uživateli možnost využívat vysoce paralelní výpočetní síly a přitom klást menší nároky na učeni se nových věcí u programátorů využívajících standardní programovací jazyky jako je C/C++.

4.2 Model zařízení CUDA

CUDA se odkazuje na hardware PC jako *host (tj. CPU a paměť RAM)* a hardware grafické karty jako *device (tj. GPU a paměť RAM na grafické kartě).*

4.2.1 Spuštění a organizace vláken



Obr 4.3[11]: Ukázka možnosti dvou rozměrné organizace bloků a vláken v nich

CUDA v programovacím jazyce C používá pro spuštění funkcí na GPU tzv. *kernels (není zaveden ustálený český výraz a označení není snadno přeložitelné).* Kernel zprostředkovává vykonání dané funkce na GPU. K jeho spuštění je nutné definovat organizaci a počet vláken.

Blok vláken (*angl. thread block,* dále jen blok) je organizovaná skupina samostatných vláken (*angl. threads*), které spolu mohou spolupracovat skrze efektivní a rychlou sdílenou paměť (*angl. shared memory*) a mohou synchronizovat provádění kódu pro koordinovaný přístup k datům. Celkové uskupení bloků vláken se nazývá *grid*.

Každé vlákno je definováno vlastním indexem (*thread ID*), který specifikuje unikátní číslo v rámci bloku. Každý blok má unikátní index (*block ID*) v rámci mřížky (*angl. grid*) nebo kernelu, který je přístupný všem vláknům v bloku. Účelem je pomoc při komplexnějším adresování. Možností je také specifikace více dimenzionálního rozměru mřížky na dvě (jak je ukázáno na Obrázku 4.3) nebo tři dimenze.

Počet vláken v jednom bloku je omezen (512 vláken a 1024 vláken u compute capability 2.x). Ovšem počet odpovídajících si bloků v jedné mřížce nebo kernelu může být mnohem vyšší. Tento model umožňuje efektivní provádění kernelů na různých zařízeních s různými schopnostmi paralelizace bez nutnosti rekompilace kódu. Počet současně běžících bloků se jednoduše přizpůsobí možnostem konkrétního GPU.

4.2.2 Paměťový model

CUDA rozlišuje paměť na základní dva paměťové prostory *host memory,* odpovídající operační paměti v PC a *device memory* reprezentující paměťový prostor grafického akcelerátoru. Tyto dva prostory jsou fyzicky odděleny, je tedy nutné provádět operace pro přenos dat z jednoho paměťového prostoru do druhého.

Popíšeme si typy pamětí, které může CUDA využívat:

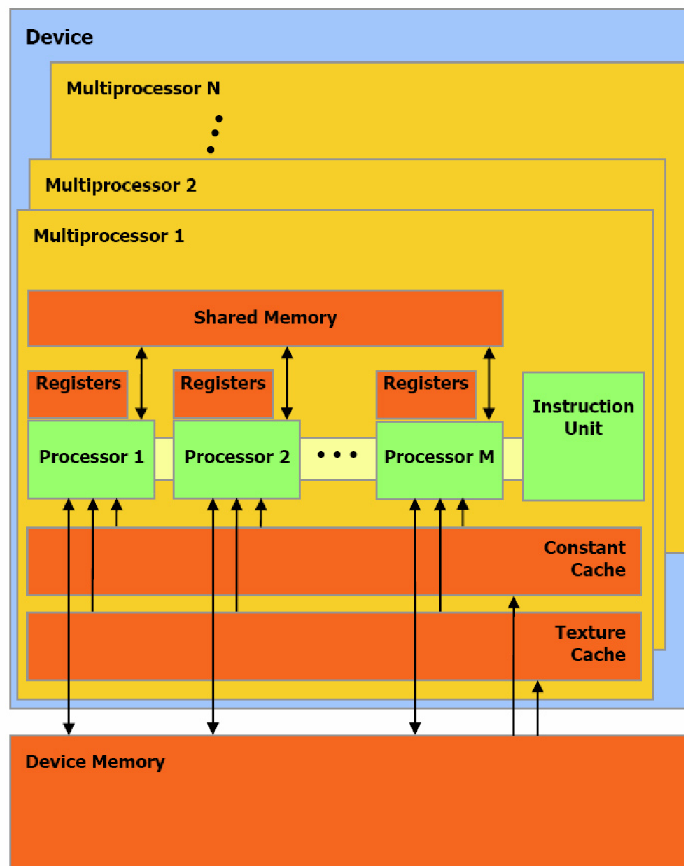
- Sada registrů (*angl. registers*) pro čtení a zápis s velmi rychlým přístupem příslušná každému vláknu.
- Lokální paměť (*angl. local memory*) pro čtení a zápis příslušná každému vláknu, je ovšem mapována do globální paměti a tak značně neefektivní.
- Sdílená paměť (*angl. shared memory*) pro čtení a zápis o velikosti 16kB příslušná každému bloku vláken. Při zarovnaném přístupu dosahuje vysoké propustnosti. Díky této paměti mohou vlákna v rámci jednoho bloku efektivně sdílet data.
- Globální paměť (*angl. global memory*) pro čtení a zápis s pomalým přístupem příslušná všem blokům vláken. Fyzicky se jedná o paměť RAM na grafické kartě. Pouze s touto pamětí je možné uskutečňovat přenosy dat z a do *host memory.*
- Paměť konstant (*angl. constatnt memory*) pouze pro čtení příslušná všem blokům vláken. Je mapována v globální pamětí. Značnou výhodou je využití lokální cache při čtení z tohoto paměťového prostoru, ideální využití nachází pokud všechna vlákna přistupují k jedné paměťové buňce současně.
- Paměť textury (*angl. texture memory*) pouze pro čtení příslušná všem blokům vláken. Má stejné vlastnosti jako paměť konstant. Navíc nabízí několik možností adresování, a také interpolaci dat pro některé specifické formáty dat.

Globální paměť, paměť konstant a paměť textury představují paměť RAM grafického akcelerátoru, a mohou být čteny a přepisovány operacemi z *host,* uložená data jsou perzistentní napříč spuštěními kernelů stejnou aplikací.

Obsáhlejší popis a způsob práce s těmito typy pamětí je v sekci 4.3

4.2.3 Model běhu

Device je složeno ze sady multiprocesorů viz Obrázek 4.4 Každý multiprocesor je postaven na architektuře „jedna instrukce, vícenásobná vlákna" (pod zkratkou SIMT – single instruction, multiple threads). Multiprocesor mapuje jedno vlákno na jeden skalární procesor, všechny procesory v multiprocesoru posléze vykonávají stejnou instrukci, ale obecně nad různými daty. Multiprocesorová SIMT jednotka vytváří, spravuje a provádí vlákna ve skupinách po 32 vláknech zvané *warps* (*half-warp* je první nebo druhá polovina warpu). Jednotlivá vlákna sestavená do SIMT warpu začínají společnou adresou programu, posléze mohou nezávisle provádět skoky a různé instrukce, za cenu nutnosti serializace.



Obr. 4.4[11]: Model hardwaru

Každému multiprocesoru je dáno provedení jednoho nebo více bloků. Bloky jsou rozděleny na warpy, takto rozděleny zůstávají jednotlivé bloky přiděleny jednotlivým multiprocesorům, pro zachování obsahu registrů a sdílené paměti.

Plánovač může přepínat vykonávání jednotlivých warpů, pro maximální využití výpočetní síly multiprocesoru, například při čekání na paměťové operace. Plného výkonu multiprocesoru je využito, když všech 32 vláken warpu jde společnou cestou v kódu. Při divergenci cest v kódu vláken v rámci jednoho warpu je nutno vykonávání kódu serializovat dokud opět nejdou stejnou cestou v kódu. V rámci jednoho bloku můžeme specifikovat body v kódu pomocí funkce `__syncthreads()`, kde se čeká než všechna vlákna z bloku dosáhnou tohoto bodu, a poté dále pokračují ve své činnosti. Samotné bloky vláken však není možné mezi sebou synchronizovat, jedinou možností je ukončení kernelu.

4.3 Možnosti optimalizace a využití pamětí

4.3.1 Optimalizace instrukcí

Pro vykonání instrukce ve warpu musí multiprocesor učinit následující kroky:

- Načtení operandů instrukce pro všechna vlákna z warpu.
- vykonání instrukce.
- Zápis výsledků všech vláken z warpu.

Efektivní zpracování instrukcí závisí na době zpracování určité instrukce, zpoždění pamětí a datové propustností pamětí. Efektivitu maximalizujeme díky následujícím bodům:

- Minimálním využíváním instrukcí s delší dobou zpracování
- Maximální využití datové propustnosti každé z pamětí.
- Umožnit plánovači vyplňovat čas pro paměťové transakce matematickými výpočty vyžadující program s vysokou aritmetickou intenzitou a velkým počtem vláken na blok vláken (potažmo na multiprocesor)

Většina aritmetických instrukcí zabírá 4 hodinové cykly, ale instrukce typu: 32-bitové násobení, dělení a zbytek po dělní v pevné řádové čárce; *log*; *sin*; *cos*; atd. zabírají 16 hodinových cyklů. Je výhodnější se jim vyhnout nebo je nahradit. Nahradit je můžeme například odpovídajícími instrukcemi se sníženou přesností na 24 bitů u operací v pevné řádové čárce, a nebo operacemi v plovoucí řádové čárce.

Instrukce řízení toku (`if, switch, do, for, while`) mohou mít při nesprávném použití značný vliv na efektivitu zapříčiněním divergence mezi vlákny ve warpech, což jak již bylo

popsáno způsobuje serializaci provádění vláken. Při používání instrukcí řízení toku je nutné myslet na minimalizaci možných cest pro divergenci.

Instrukce pro operace s pamětí zahrnující čtení nebo zápis zabírají multiprocesoru 4 hodinové cykly při operacích se sdílenou pamětí. U operací s globální a lokální pamětí musíme připočítat 400 až 600 hodinových cyklů zpoždění. Značná část zpoždění se může ukrýt při aritmetických operacích jak již bylo zmíněno.

4.3.2 Globální paměť

Pro globální paměť neexistuje cache, je tak nutné dodržovat jisté zásady přístupu. Hlavní je dodržování zarovnaného přístupu při zarovnávání paměti na 4, 8 nebo 16 bytů. Takto je vhodné postupně jdoucími vlákny operovat s postupně jdoucími adresami. Přičemž počáteční adresa je zarovnána na určité místo. Ve vyšší architekturách s *compute capability(viz. [3])* 1.2 a vyšší postačuje číst data z velkých zarovnaných bloků pro dosažení maximální paměťové propustnosti.

4.3.3 Paměť konstant

Využívá lokální cache o velikosti 8kB na multiprocesor. Pro všechna vlákna v half-warp je přístup rychlý jako u registru, pokud současně přistupují k jedné adrese.

4.3.4 Paměť textury

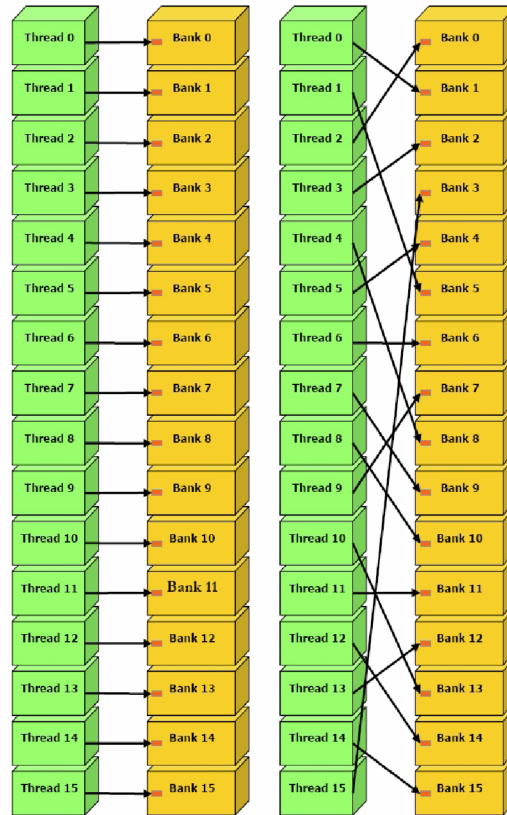
Paměťový prostor textury využívá lokální cache paměť o velikosti 6-8kB na multiprocesor. Čtení z paměti textury stojí jedno čtení z globální paměti pouze v případě *cache miss (*patřičná data se nenacházejí v cache), v opačném případě stojí pouze jedno čtení z paměti cache. Paměť cache textury je specializována na dvou rozměrný, případně tří rozměrný souřadný systém a umožňuje využít HW lineární a trilineární interpolace. Warp kde jsou žádána data v těsné blízkosti dosáhne nejlepších výsledků. Používání paměti textury tak může být velmi výhodné ve srovnání se čtením čistě z globální paměti.

4.3.5 Sdílená paměť

Z důvodu instalace sdílené paměti „na čipu GPU“ je mnohem rychlejší než lokální, či globální paměť. V případě současného přístupu všech vláken warpu ke sdílené paměti je přístup rychlý jako u registrů, ovšem pouze při dodržení zarovnaného přístupu bez konfliktů bank, jak je popsáno níže.

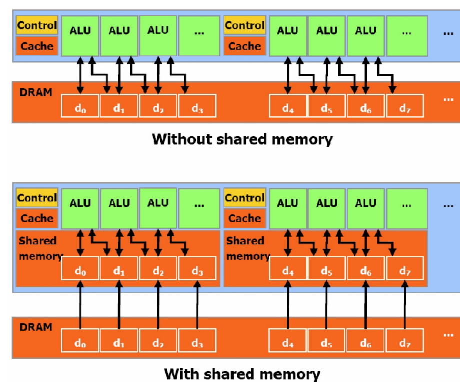
Sdílená paměť je rozdělena do paměťových modulů o stejné velikosti zvané *banky* (*angl. banks*), které mohou pracovat současně. Pokud máme paměťové operace, založeny na *n* různých adresách, které spadají do *n* různých bank, tak mohou být operace provedeny současně s *n* násobně

vyšší paměťovou propustností než je paměťová propustnost jednoho modulu. V případě, že dvě adresy spadají do jedné banky vzniká konflikt bank a přístup musí být serializován, což výrazně snižuje celkovou paměťovou propustnost. Je nutné snažit se mapovat současně žádané adresy na různé banky.



Obr 4.5[11]: Ukázky zarovnaných přístupů bez konfliktů bank ve sdílené paměti

Ve sdílené paměti jsou banky organizovány jako postupná 32-bitová slova. V případě, kdy postupně jdoucí vlákna požadují data z postupně jdoucích adres vzdálených 32-bitů nevznikají žádné konflikty bank, jak je ilustrováno na Obr. 4.5. Postupně jdoucí vlákna nemusí pracovat jen s postupně jdoucími adresami, je však důležité vyhnout se konfliktům, jak také ukazuje Obr. 4.5.



Obr. 4.6[11]: Znázornění sdílené paměti přinášející data blíže k výpočetním jednotkám

Sdílenou paměť lze chápat i jako cache plně kontrolovanou programátorem, přinášející data blíže k výpočetním jednotkám, jak znázorňuje Obr. 4.6.

4.3.6 Registry

Obecně přístup k registru nevyžaduje žádný přístupový čas navíc, zpoždění mohou nastat díky *read-after-write* závislostem a konfliktům bank registrů. Zpoždění způsobeno *read-after-write* závislostmi může být multiprocesorem úspěšně maskováno při použití nejméně 192 vláken na blok. Překladač a plánovač pracují na minimalizaci konfliktů bank u registrů, nejlepší výsledky vykazují když je počet vláken na blok násobkem 64. Doporučení je používat co nejméně registrů na vlákno, multiprocesor tak může obsluhovat více bloků vláken, což přispívá ke zrychlení.

4.3.7 Počet vláken v bloku vláken a počet bloků vláken

Počet vláken v bloku a počet bloků se volí pro maximální využití výpočetních možností GPU. Bloků má být nejméně tolik jako multiprocesorů.

Při běhu jen jednoho bloku na multiprocesoru se nevyhneme zpoždění při synchronizaci a paměťových operacích, pokud není dostatečný počet vláken na blok pro zakrytí zpoždění. Je efektivnější aby pro jeden multiprocesor byl vyšší počet aktivních bloků, pro přepínání mezi bloky při zpožděních. Z toho vyplývá doporučení použít nejméně dvakrát více bloků než je multiprocesorů. Důležité je například využití sdílené paměti. Množství využívané paměti jedním blokem by nemělo být větší než polovina celkové sdílené paměti, která je k dispozici pro jeden multiprocesor, aby druhá polovina mohla být současně využívána jiným blokem. Shodným způsobem ovlivňuje rychlost i počet využívaných registrů.

64 vláken na blok je minimální rozumný počet vláken na blok, vhodný ovšem jen při velkém množství bloků na multiprocesor, přičemž bloky musí být minimálně náročné na sdílenou paměť a počet registrů. Doporučený počet vláken na blok je 192 a vyšší, pro rozumné využití výkonu.

4.3.8 Přenosy dat mezi host a device

Datové přenosy mezi paměťmi na device jsou značně rychlejší než přenosy host-device a device-host. Pro minimalizaci přenosů dat mezi host a device se doporučuje přesunout více výpočtů na device i za cenu nižší paralelizace.

4.3.9 Interoperabilita s OpenGL

OpenGL[16] (*Open Graphics Library*) je průmyslový standard specifikující multiplatformní rozhraní (API) pro tvorbu aplikací počítačové grafiky. Používá se při tvorbě počítačových her, CAD programů, aplikací virtuální reality či vědeckotechnické vizualizace apod.

Chceme-li výstup z našeho CUDA programu použít pro zobrazování na GPU bylo by značně neefektivní data přenášet z host na device a následně je pomocí příkazů OpenGL posílat zpět na GPU. V OpenGl existují *buffer*, *texture* a *renderbuffer* objekty struktury mapující data určena k zobrazení, ať už je jedná o souřadnice vertexů, normálové vektory, texturovací souřadnice, textury nebo obraz vykreslené scény, do RAM na GPU. Toho můžeme využít registrováním těchto zdrojů pro CUDA, kdy s nimi pracujeme jako s globální pamětí. Po odregistrování může s objekty opět pracovat OpenGL.

5 Generování procedurálních terénů na GPU

Některé z poznatků a principů v této kapitole vycházejí z [2], [3] a [4].

GPU je ideálním prostředkem pro provádění vysoce náročných paralelních výpočtů. Můžeme tedy použít systém CUDA pro programování s správu těchto výpočtů. Ideálním je toto řešení i proto, že veškerá výstupní data pro zobrazení jsou generována přímo do RAM grafického akcelerátoru, nemusíme tak vůbec žádná data přenášet z host na device. Nemáme ani žádná vstupní data pro výpočet, nebo jen minimální, a tak je plně využit výpočetní výkon bez zpoždění zapříčiněnými paměťovými operacemi. Nechceme se omezovat pouze na výškové mapy, ale chceme dosáhnou plně prostorového terénu s prostorovými výběžky, tunely, jeskyněmi a podobně.

5.1 Marching cubes na CUDA

Jako základní prostředek zvolme algoritmus Marching cubes vizualizující námi zvolenou funkci (definovanou dále), jako implicitní plochu. Zvolme základní sémantiku, kde hodnoty menší než 0

považujeme za vzduch nebo vodu a pozitivní hodnoty za pevnou půdu nebo skály. Hranice mezi pozitivní a negativní hodnotou (tedy hodnota nula) je povrch terénu, kde budeme generovat trojúhelníkovou síť reprezentující tento povrch.

Pro zpracování na CUDA přidělme jednomu vláknu výpočet jedné objemové buňky. Důležitou volbou je i počet vláken na blok a celkový počet bloků pro pokrytí námi požadovaného prostoru. CUDA umožňuje definovat blok vláken třemi rozměry, ale celkový výpočetní grid je možno definovat pouze dvěma rozměry. Nejlepší volbou tak bude když zvolíme i indexaci vláken v bloku dvou rozměrnou. A pozici vlákna v prostoru vypočítáme následovně. Definujme si pro blok vláken velikost 16x16. Můžeme se to představit jako dlaždici o velikosti 16x16 a výšce 1. Celkový výpočetní grid definujme o velikosti 64x80. Představit si to můžeme jako když pro jednu hodnotu souřadnice z vyskládáme prostor x a y 64-mi dlaždicemi o velikosti 16x16 získáme tak plochu o velikosti 128x128, druhý index z gridu požijeme jako hodnotu v souřadnici z. Celkový počet elementů pro tuto konfiguraci je 128x128x80. Výpočet pozice pro jedno vlákno (užitím standardní CUDA indexace vláken a bloků) tak provádíme následovně:

$$
\begin{aligned}
&blockDim.x = q^2 ; q \in \mathbb{N} \\
&x = (blockIdx.x \circ \sqrt{blockDim.x}) * blockDim.x + threadIdx.x \\
&y = (blockIdx.x / \sqrt{blockDim.x}) * blockDim.x + threadIdx.y \\
&z = blockIdx.y \\
&(operace \circ značí\ zbytek\ po\ celočíselném\ dělění\ a\ /\ operaci\ celočíselnoho\ dělení)
\end{aligned}
\tag{5.1}
$$

Pro přístup k hodnotám vyhledávacích tabulek můžeme využít paměť textury. Vyhledávání prvků v tabulce je náhodné, právě v těchto případech má paměť textury největší výhody oproti ostatním typům pamětí.

Samotný výpočet probíhá následovně:

- Výpočet hodnoty funkce a normálového vektoru ve všech vrcholech objemové buňky.
- Sestavení hodnoty **T**. Podle hodnoty **T** můžeme výpočet i ukončit pokud se nebudou generovat žádné trojúhelníky. Může tím vznikat velká divergence, ale je dosti pravděpodobné, že díky lokalitě vláken pujdou všechny vlákna z warpu nebo celého bloku stejnou programovou cestou, tedy negenerují žádné trojůhelníky. Divergenci tak můžeme pokládat za přiměřeně minimální.
- Pro minimalizaci další divergence vypočteme interpolaci pozice a normálového vektoru na všech hranách, i těch nevyužitých, dochází tím k velké redundanci výpočtu, ale minimalizaci divergence. Jelikož nemůžeme předpokládat které a v jakém pořadí budeme hrany používat musíme vypočíst všechny pro minimalizaci divergence. Daní za to je velká paměťová náročnost.

- Dalším krokem je získání adresy pro ukládání dat pomocí atomických operaci. Nevíme předem které konkrétní vlákno bude ukládat kolik trojúhelníků. Aby nevznikali proluky nebo překryvy, pomocí atomických operací získá každé vlákno, které bude ukládat určitý počet trojúhelníků, vlastní adresu odlišnou od všech ostatních vláken.
- Poslední část výpočtu sestává z ukládání přepočítaných hodnot vrcholů trojúhelníků, normálových vektorů pro vrcholy a texturovacích souřadnic do buffer objektů. Tato větev je také poněkud divergentní, ale ne markantně.

5.2 Funkce pro popis terénu

Definujme si funkci ***F(x,y,z)*** popisující terén, nazvěme ji *densitou*[2]. *Noise3D(x,y,z, frekvence)* je šumovou funkcí interpolovanou trilineárně, parametry funkce jsou pozice v prostoru a frekvence šumové funkce podle definice v kapitole 2.2. Výstupem funkce jsou hodnoty v rozsahu -1 až +1. Chceme-li změnit amplitudu, hodnotu funkce vynásobme požadovanou amplitudou, respektive její polovinou, když uvažujeme i záporné hodnoty.

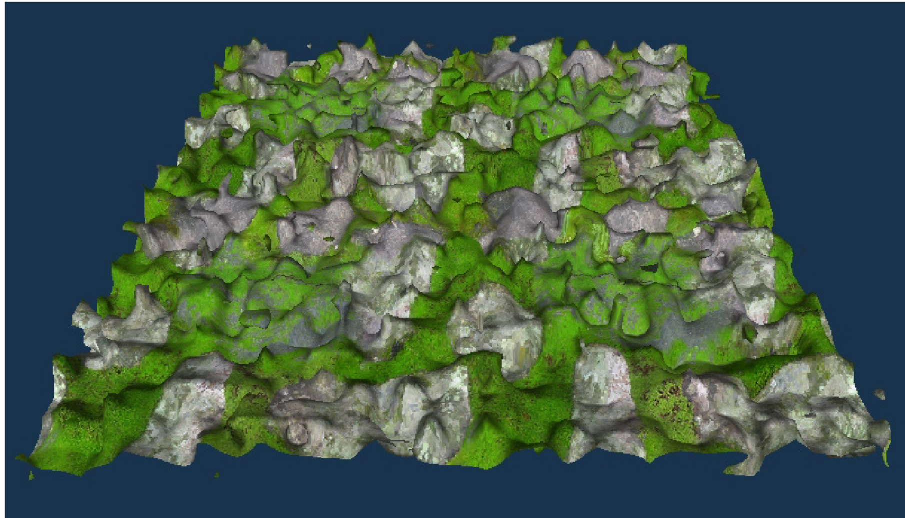
Jako základ můžeme vytvořit plochý terén podle vzorce (5.2). Dojde k rozpůlení prostoru v ose z na kladná a záporná čísla. Konstantou jako je *0,5* můžeme nastavit pro jakou hodnotu v ose z dojde k rozpůlení. Obr. 5.1 zobrazuje výsledný terén.

$$F(x,y,z) = -z + 0{,}5 \tag{5.2}$$

Obr. 5.1: Zobrazení terénu pro rovnici (5.2)

Zkusme přidat terénu nějakou variaci použitím šumové funkce o frekvenci *16* a amplitudě *0,2* viz. vzorec (5.3).

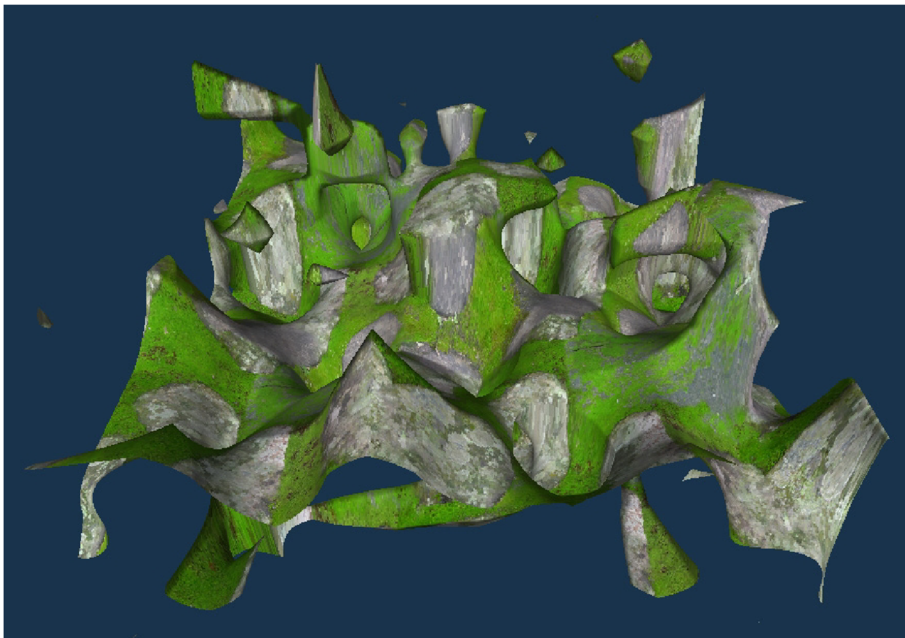
$$F(x,y,z) = -z + 0{,}5 + 0.2 * Noise3D(x,y,z,16) \tag{5.3}$$



Obr. 5.2: Zobrazení terénu pro rovnici (5.3)

Pro další pokus zkusme využít nižší frekvence o vyšší amplitudě.

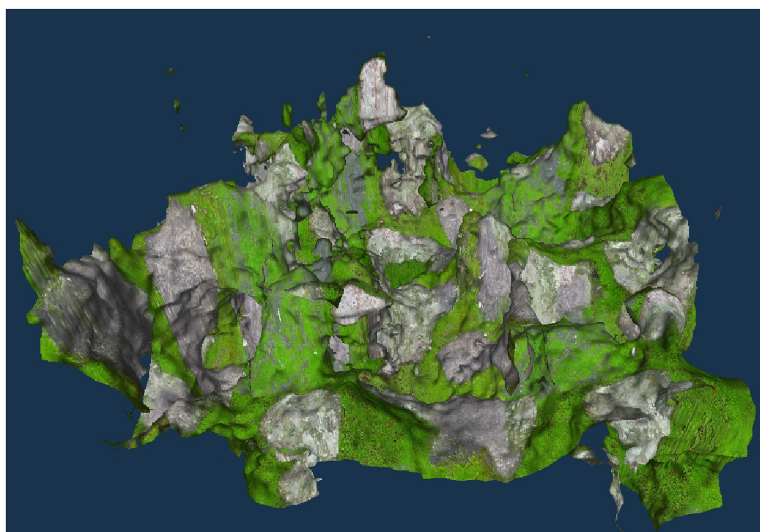
$$F(x, y, z) = -z + 0{,}5 + 0{,}8 * Noise3D(x, y, z, 4) \qquad (5.4)$$



Obr. 5.3: Zobrazení terénu pro vzorec (5.4)

Náš terén už vypadá mnohem lépe není však stále ještě dostatečně zajímavým Přidáme proto jako v případě Perlinova šumu další oktávy s vyššími frekvencemi a nižší amplitudou. Nemusíme se striktně držet přístupu jako v Perlinově šumu. Frekvenční rozestupy a amplitudy můžeme volit jakkoliv nám přijde vhodné pro docílení zajímavého vzezření terénu.

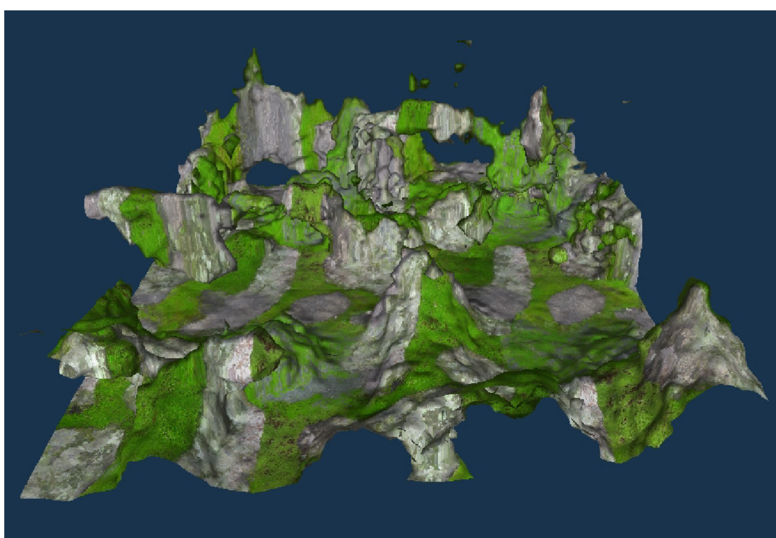
$$F(x, y, z) = -z + 0{,}5 + 0{,}8 * Noise3D(x, y, z, 4) + 0{,}1 * Noise3D(x, y, z, 16) + 0{,}05 * Noise3D(x, y, z, 32) \qquad (5.5)$$



Obr. 5.4: Zobrazení terénu pro vzorec (5.5)

Náš terén už vypadá o mnoho zajímavěji. Pro vyšší míru realističnosti můžeme přidat i pevné dno. Do vzorce musíme zakomponovat, že od určité hodnoty v ose *z* se objevují vysoké kladné hodnoty pro vytvoření pevné země. K tomuto účelu využijeme funkci *saturate(x)* s jednou funkční hodnotou. Funkce ořezává funkční hodnotu do rozsahu 0 až +1, je-li funkční hodnota nižší než 0, pak hodnota funkce je rovna 0, je-li funkční hodnota vyšší než 1, je hodnota funkce rovna 1. Sestavíme tak vzorec (5.6), kde hodnota 0,5 v argumentu funkce *saturate* značí v jaké výšce má vzniknout pevná zem. Další přidané hodnoty už jen určují s jakou intenzitou má dno vzniknout, tedy jak ploché má být.

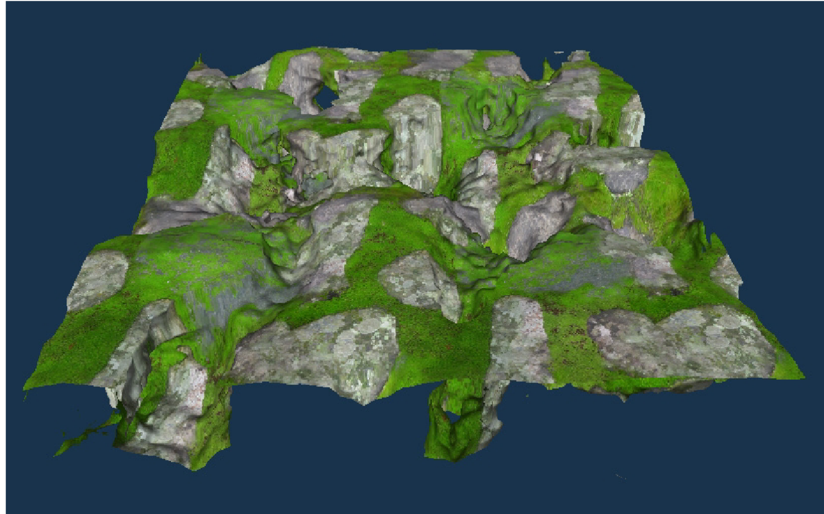
$$\begin{aligned} F(x,y,z) = & -z+0{,}5+0{,}8*Noise3D(x,y,z,4)+0{,}1*Noise3D(x,y,z,16) \\ & +0{,}05*Noise3D(x,y,z,32)+saturate((0{,}5-pos.z)*4)*4 \end{aligned} \quad (5.6)$$



Obr. 5.5: Zobrazení terénu pro vzorec (5.6)

Naopak chceme-li vytvořit rovný terén s prohlubněmi můžeme použít funkci *saturate* naprosto obráceně.

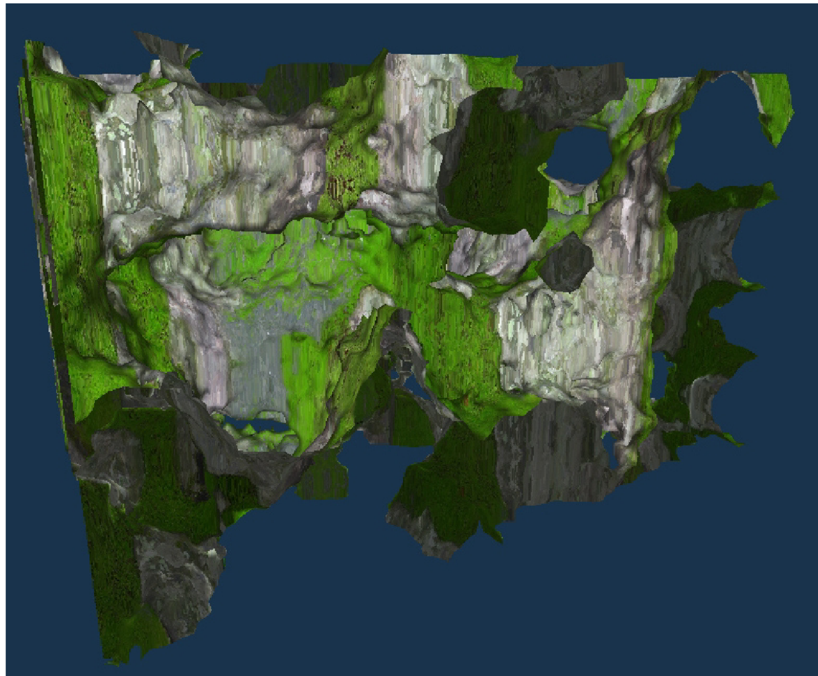
$$F(x,y,z)=-z+0{,}5+0{,}8*Noise3D(x,y,z,4)+0{,}1*Noise3D(x,y,z,16) \\ +0{,}05*Noise3D(x,y,z,32)-saturate((pos.z-0{,}5)*4)*4 \quad (5.7)$$



Obr. 5.6: Zobrazení terénu pro vzorec (5.7)

Pokud vypustíme kompletně část funkce pro rozdělení prostoru (5.2) i část funkce pro pevné dno dostáváme zajímavý jeskynní systém v celém prostoru. (ořezání na Obr. 5.7 je způsobeno tím, že nevizualizujeme celý prostor ale jen jeho část)

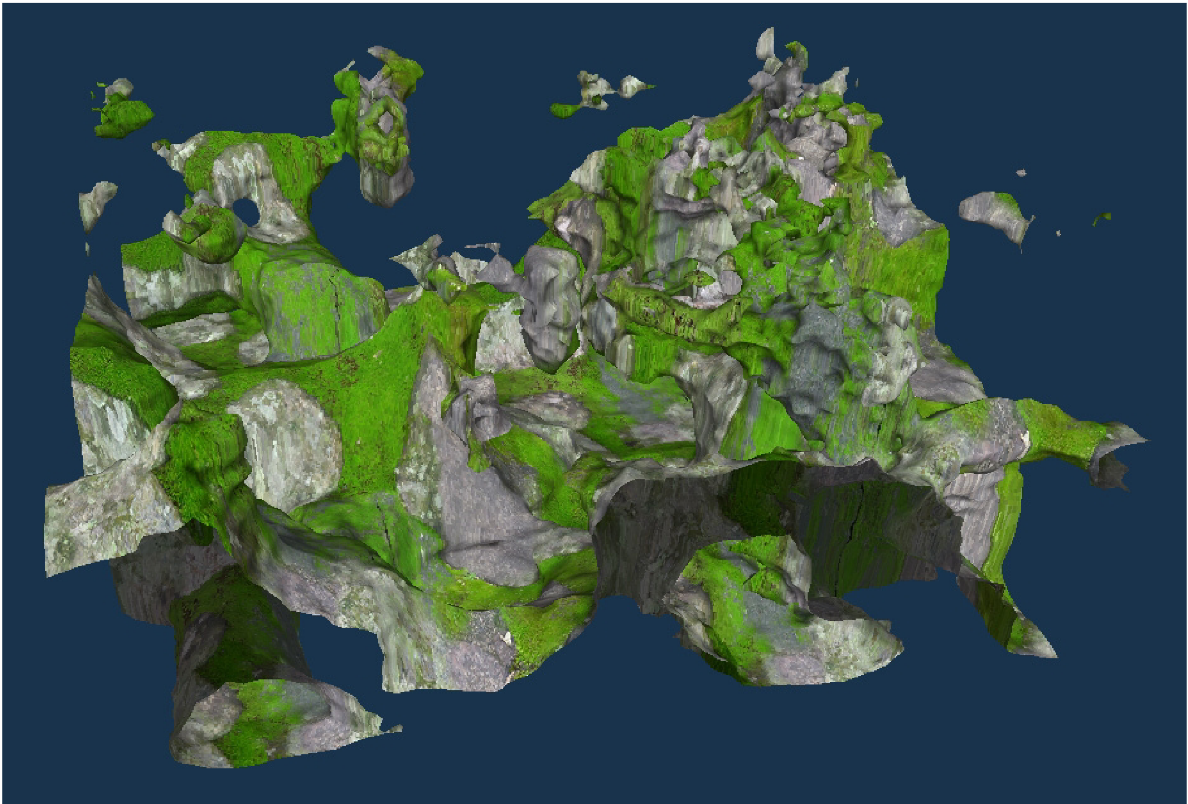
$$F(x,y,z)=0{,}8*Noise3D(x,y,z,4)+0{,}1*Noise3D(x,y,z,16) \\ +0{,}05*Noise3D(x,y,z,32) \quad (5.8)$$



Obr. 5.7: zobrazení terénu pro vzorec [5.8]

Pro dosažení ještě zajímavější variací terénu můžeme využít další funkce jako *sin*, *cos*, *ln*, a podobně. Obr. 5.8 zobrazuje užití první oktávy šumu jako argumentu funkce *sin*.

$$\begin{aligned} F(x,y,z) = -z + 0{,}2 + 0{,}8 * \sin(\mathit{Noise3D}(x,y,z,4)) + 0{,}1 * \mathit{Noise3D}(x,y,z,16) \\ + 0{,}05 * \mathit{Noise3D}(x,y,z,32) + \mathit{saturate}((0{,}2 - \mathit{pos.z}) * 4) * 4 \end{aligned} \qquad (5.9)$$



Obr. 5.8: Zobrazení terénu pro vzorec (5.9)

Důležitým prvkem přidávajícím na reálnosti výsledku je osvětlení řešeno normálovými vektory ve vrcholech trojúhelníků, jak bylo popsáno výše a textura na planární projekci v ose *z*. Rychlost toho řešení dosahuje téměř potřeb pro zpracovávání v reálném čase, jelikož čas zpracování je cca 0,2s na grafickém akcelerátoru nVidia GeForce 9800GT při 128x128x80 objemových buňkách a cca 540 000 vygenerovaných trojúhelníků pro Obr. 5.8.

6 Závěr

Práce se zabývá jak obecnou problematikou procedurálního generování terénů, tak i generování plně prostorových terénů a jejich vizualizací a akcelerací na architektuře CUDA. Cílem bylo také prozkoumat architekturu CUDA pro využití grafických akcelerátorů za účelem nejen urychlování procedurálního generování terénů, ale i dalších obecnějších výpočtů. Důležitou součástí práce je i vytvoření vhodné funkce respektive množiny funkcí vhodných pro popis terénů. Použitím čistě procedurálního přístupu jsme kompletně potlačili jakoukoliv datovou náročnost a navršení výpočetního času díky výpočetní náročnosti se podařilo eliminovat použitím architektura CUDA. Při využití výkonných grafických akcelerátorů je možno dosáhnout i zpracování v reálném čase. Z provedených experimentů vyplynulo, že základní doporučené postupy jsou nejlepším přístupem. Konkrétně myšlenka jednoho vlákna počítající jediný element. A myšlenka kombinace několika šumových funkcí o různých amplitudách a frekvencích, případně obohacena o další funkce, je dobrou volbou pro popis komplexních terénů.

Práce je přínosná ve smyslu nabídnutí zajímavého přístupu ke generování komplexních terénů a značného urychlení jejich vizualizace, případně urychlení vizualizace jiných funkcí nebo objemových dat.

Pokračování je možné velkým množstvím směrů. Může se jednat o další prozkoumání možností popisu terénů různými funkcemi. Například pro generování pilířů, oblouků nebo dalších zajímavých možností využití jednoduchých matematických popisů. Pro zlepšení vizuální kvality můžeme využít tri-planárního texturování s měkkými přechody a některou z metod globální osvětlení jako je Ambient occlusion. A další možností urychlení, jako například předgenerování hodnot šumů a využití paměti textury pro jejich rychlé načítání a HW podporu trilieární interpolace. Jelikož generování šumu a jeho interpolace je nejvíce výpočetně náročnou částí. Pak například navázání generování na volumetrická data pro určení základního reliéfu terénu a jeho real-time editaci.

Literatura

[1] EBERT, David S.; MUSGRAVE, F. Kenton; REACHEY, Darwyn; PERLIN, Ken; WORLEY Steve. *Texturing & Modeling : A procedural approch*. 2nd. ed. San Diego, CA, USA : Academic Press, 1998. 415 s. ISBN 0-12-228730-4.

[2] GEISS , Ryan. Generating Complex Procedural Terrains Using the GPU : Chapter 1.. In NGUYEN, Hubert. *GPU Gems 3*. Boston, Massachusetts, USA : Addison-Wesley Professional, 2007. s. 1008. Dostupné z WWW: <http://http.developer.nvidia.com/GPUGems3/gpugems3_ch01.html>. ISBN 13-978-0321515261.

[3] GEISS, Ryan,; THOMPSON, Michael. 2007. "NVIDIA Demo Team Secrets—Cascades." Prezentace. Developers Conference 2007. Dostupné z WWW: <http://developer.download.nvidia.com/presentations/2007/gdc/CascadesDemoSecrets.zip>

[4] NVIDIA Corporation. 2007. "Cascades." Demo. Dostupné z WWW: <http://www.nzone.com/object/nzone_cascades_home.html>.

[5] BOURKE, Paul. *Geometry, Surfaces, Curves, Polyhedra* [online]. 1994 [cit. 2011-01-10]. Polygonising a scalar field. Dostupné z WWW: <http://local.wasp.uwa.edu.au/~pbourke/geometry/polygonise/>.

[6] MARTZ, Paul. *Game programing and graphics programing* [online]. 2004 [cit. 2011-01-10]. Generating Random Fractal Terrain. Dostupné z WWW: <http://gameprogrammer.com/fractal.html>.

[7] BOURKE, Paul. *Paul Bourke - Personal Pages* [online]. 2000 [cit. 2011-01-10]. Perlin Noise and Turbulence. Dostupné z WWW: <http://local.wasp.uwa.edu.au/~pbourke/texture_colour/perlin/>.

[8] BOURKE, Paul. *Paul Bourke - Personal Pages* [online]. 1999 [cit. 2011-01-10]. Interpolation methods. Dostupné z WWW: <http://local.wasp.uwa.edu.au/~pbourke/miscellaneous/interpolation/>.

[9] Mandelbrotova množina. In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, 2. 10. 2005, last modified on 24. 3. 2009 [cit. 2011-01-10]. Dostupné z WWW: <http://cs.wikipedia.org/wiki/Mandelbrotova_množina>.

[10] Aliasing. In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, 25 November 2002, last modified on 27 December 2010 [cit. 2011-01-10]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Aliasing>.

[11] nVidia. *CUDA C Programing Guide, v 3.2* [online]. Santa Clara, California, USA : 11. 9. 2010 [cit. 2011-01-10]. Dostupné z WWW: <http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf>.

[12] Jan Ryba: Detekce zájmových bodů na CUDA, bakalářská práce, Brno, FIT VUT v Brně, 2009

[13] Trilinear interpolation. In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, 23 May 2004, last modified on 10 November 2010 [cit. 2011-01-10]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Trilinear_interpolation>.

[14] ELIAS, Hugo. *The good-looking textured light-sourced bouncy fun smart and stretchy page* [online]. 2003 [cit. 2011-01-10]. Perlin Noise. Dostupné z WWW: <http://freespace.virgin.net/hugo.elias/models/m_perlin.htm>.

[15] Fraktál. In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, 12. 9. 2005, last modified on 19. 11. 2010 [cit. 2011-01-10]. Dostupné z WWW: <http://cs.wikipedia.org/wiki/Fraktál>.

[16] KHRONOS Consortium. *OpenGL : The Industry's Foundation for High Performance Graphics* [online]. 1997 [cit. 2011-01-10]. Dostupné z WWW: <http://www.opengl.org/>.

Seznam příloh

Příloha 1. CD se zdrojovými texty a manuálem.