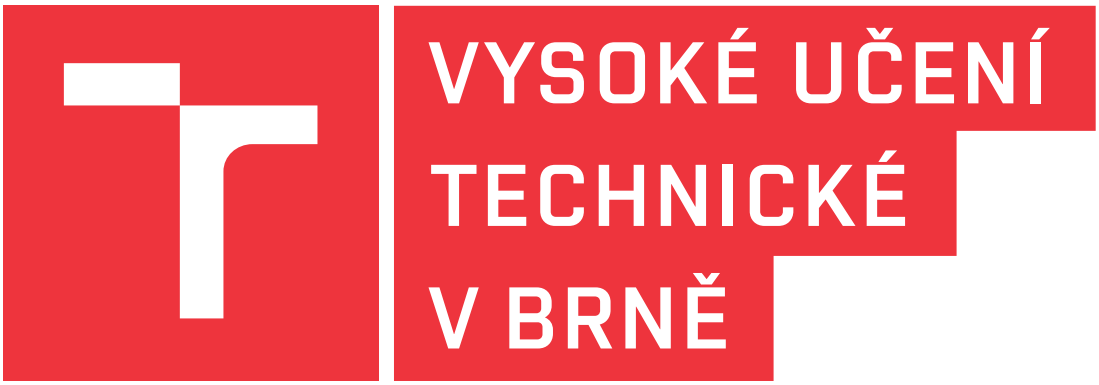


Pokročilá počítačová grafika

Generování terénu

11. ledna 2020

Autor: Petr Flajšingr, xflajs00@stud.fit.vutbr.cz
Fakulta Informačních Technologií
Vysoké Učení Technické v Brně

Úvod

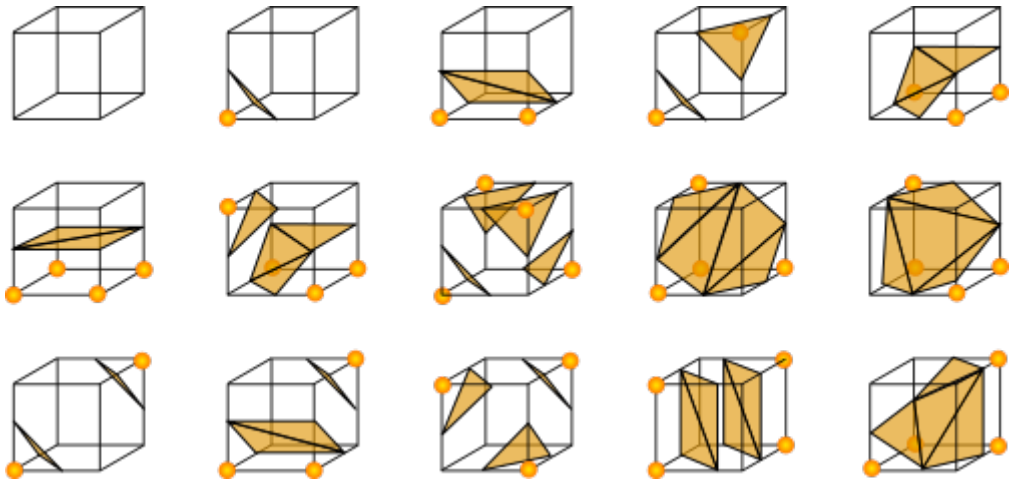
Tato práce se zabývá implementací generování procedurálního terénu na grafické kartě. Generování a vykreslování terénu je implementováno pomocí c++20 a OpenGL. V práci jsou dále popsány další využité metody jako cascaded shadow mapping, level of detail a další.

Generování terénu

Pro generování terénu existuje mnoho přístupů. Nejpopulárnějším jsou asi výškové mapy, kdy je hodnota šumu přímo přenášena do výškových souřadnic. Jednoduchou implementaci výškových map můžete najít na adrese `https://github.com/PetrFlajsingr/TerrainAnimation`. Tento přístup má ale jeden hlavní nedostatek; neumožňuje vykreslovat více vrstev terénu (jeskyně, převisy...). Aby byl schopen můj engine zobrazovat i složitější útvary zvolil jsem algoritmus marching cubes.

Marching cubes

Algoritmus marching cubes pracuje na principu transformace voxelů do polygonální sítě. Pro výpočet je prostor rozdělen na několik malých částí (kostek) a v nich je vytvořeno 0 až 5 trojúhelníku – podle konfigurace voxelu. Celkový počet možných konfigurací v rámci jedné kostky je 256. Na obrázku 1 je ukázka několika možných konfigurací.



Obrázek 1: Ukázka konfigurací

Algoritmus generování

Celé generování probíhá kompletně na grafické kartě. Algoritmus je rozdělen do několika kroků z nichž každý je reprezentován samostatným programem pro GPU. Níže najdete popis generování sítě pro jeden chunk – tento proces je opakován pro všechny chunky, které je nutné vytvořit.

Prostor je pro každý chunk dělen na oblasti o velikosti 32x32x32 jednotek – velikost těchto jednotek je určena konfiguračním souborem.

V algoritmu je využíváno look-up tabulek pro konfigurace buněk a vztahy mezi hranami a vrcholy kostek.

1. Generování hustoty terénu – v této fázi jsou do bufferu, který reprezentuje voxely v prostoru, zapsány hodnoty funkce určující rozložení terénu. Hodnoty nižší než 0 reprezentují oblast mimo terén, vyšší než 0 uvnitř terénu a 0 je povrch. Tento krok je realizován pomocí compute shaderů. Funkce pro generování šumu je založena na článku [1].

2. Streaming markerů pro voxely – prokaždý voxel je vytvořen flag reprezentující jeho konfiguraci (obrázek 1). Konfigurace je určena existencí vertexu na hraně kostky. Jedná se tedy o 8 bitů kde 0 reprezentuje existenci vertexu na dané hraně. V této fázi jsou zahozeny veškeré prázdné voxely, pokud tedy nevygenerujeme žádné hodnoty nemusíme pokračovat dále a můžeme chunk označit jako prázdný. Realizováno pomocí geometry shaderů a transform feedback.

3. Streaming markerů pro hrany – snadno můžeme rapidně snížit množství zpracovávaných dat tím, že pro každou kostku budeme zpracovávat pouze hrany jí vlastní, abychom se vyhnuli vícenásobnému generování vertexů na stejné pozici pro různé voxely. Vybereme tedy pouze hrany napojené na nultý vrchol a pro ně vystreamujeme marker pokud se na nich nachází vrchol. Realizováno pomocí geometry shaderů a transform feedback.

4. Generování vertexů/normál – pro každý marker z minulého kroku je vygenerován vertex následovně: Pomocí hodnot hustoty terénu ve vrcholech kostky sousedící se zpracovávanou hranou vypočteme finální pozici vertexu (rovnice 1).

$$v = v_0 + (v_1 - v_0) * \frac{density(v_0)}{abs(density(v_0) - density(v_1))} \tag{1}$$

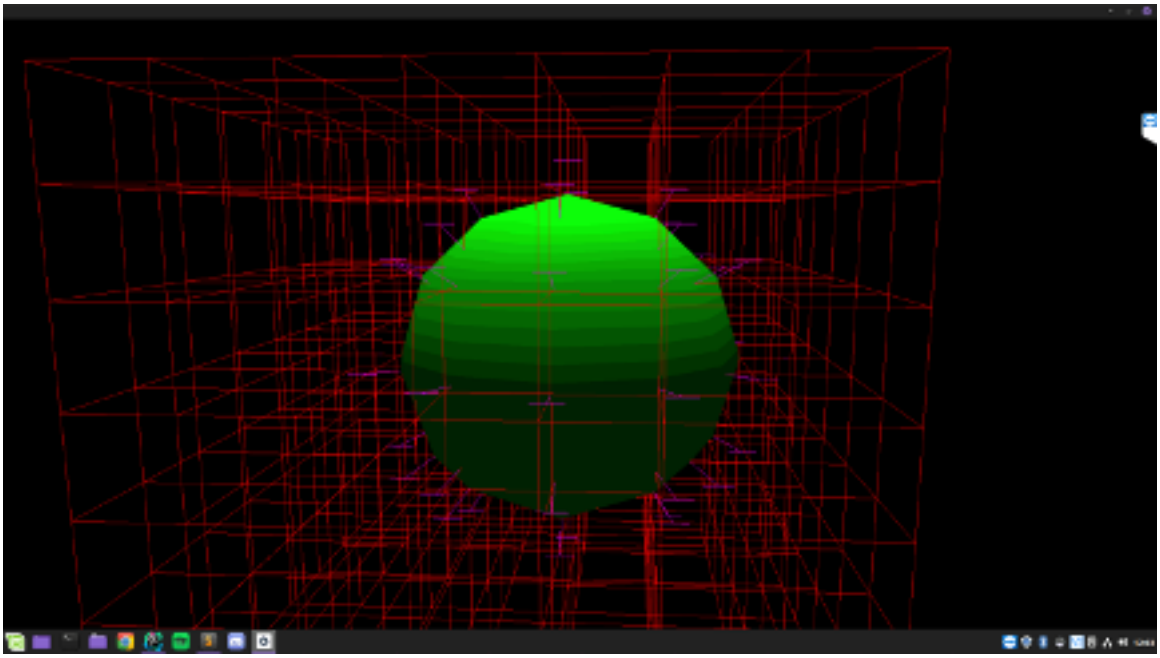
   Zároveň generujeme normály pro jednotlivé vertexy. Pro výpočet normál je opět využito funkce hustoty. Samplujeme funkci všemi směry čímž získáme její gradient. Po jeho normalizaci dostaneme přesnou normálu povrchu.

   Realizováno pomocí vertex shaderů a transform feedback.

5. Ukládání vertex IDs – jelikož potřebujeme vytvořit element buffer abychom mohli síť vykreslit, je potřebné podniknout několik kroků. Prvním y nich je uložení informací o indexech vertexů. Do bufferu odpovídající velikosti chunku (32x32x32 * počet hran pro voxel (3)) jsou uloženy indexy na odpovídající pozici. Tento krok je prováděn za využití markerů hranu z kroku 3. Realizováno pomocí vertex shaderů za využití `gl_VertexID`.

6. Generování element bufferu – po přípravě potřebných informací je možné vygenerovat element buffer pro vykreslování. Průchod probíhá podle markerů pro kostky vygenerovaným v kroku 2. Pro každou kostku, která není prázdná, jsou dohledány indexy v bufferu připraveném v předchozím kroku a jsou vystreamovány do výstupního bufferu. Realizováno pomocí geometry shaderů a transform feedback.

Po provedení těchto kroků můžeme použít buffer hustoty pro další chunk a máme připraveny 3 buffery pro vykreslování – vertexy, normály a elementy. Na obrázku 2 je koule vygenerovaná pomocí marching cubes s růžově vyznačenými normálami.

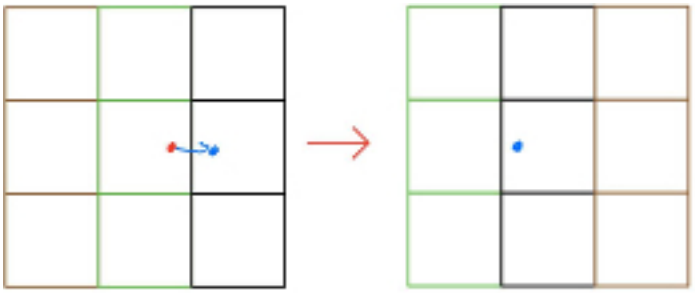


Obrázek 2: Koule vygenerovaná pomocí marching cubes

Jak již bylo naznačeno, nedochází ke generování pouze jednoho chunku, ale velkého množství. Tohle je blíže popsáno v následujících sekcích.

Nekonečný svět

Jelikož se jedná o procedurálně generovaný terén tak je vhodné, aby se mohl načítat do nekonečna. Princip funkčnosti je následující. Oblast kolem pozice kamery je rozdělena na 27 podoblastí – 3x3x3, přičemž středová oblast reprezentuje tu, ve které se nachází kamera. Snad jasněji na obrázku 3, kde v levé části došlo k pohybu kamery mezi oblastmi, což vedlo k jejich přeskládání.



Obrázek 3: Přesun okolí podle pohybu kamery (ve 2D pro jednoduchost)

Každá z těchto oblastí obsahuje pevný počet chunků. Při přesunu je tedy nutné všechny využité chunky ve smazané oblasti recyklovat a chunky v nových oblastech vypočítat.

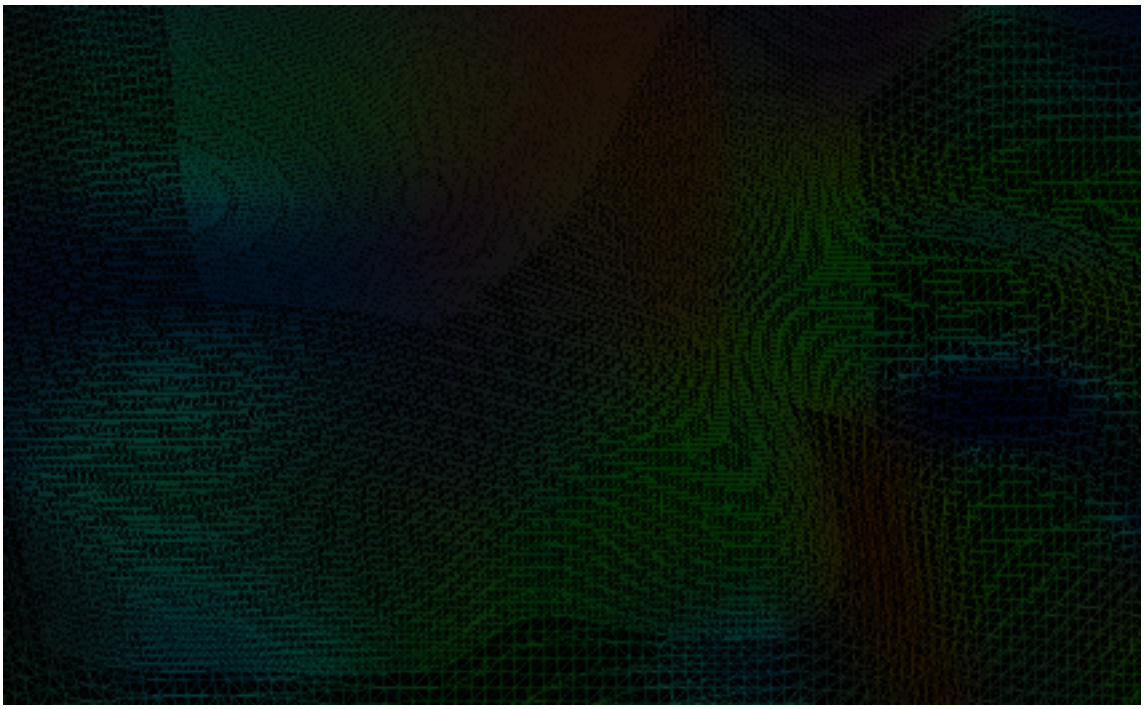
Chunky jsou zobrazovány podle vzdálenosti jejich bounding sphere od pozice kamery. Jelikož můžeme vytvořit bounding sphere i kolem celé podoblasti obsahující velké množství chunků, můžeme kontrolovat i jejich vzdálenost a tím ignorovat velké množství vzdálených chunků.

Každý chunk v těchto oblastech také obsahuje informace o tom, jestli byl již spočítán a zda obsahuje nějaké vertexy. Pokud neobsahuje tak je v dalších snímcích ignorován.

Level of detail

Vzhledem k tomu, že chceme zobrazovat terén do větší vzdálenosti, je nutné implementovat level of detail. Pro marching cubes lze řešit jednoduchým způsobem – můžeme rozdělit chunk na 8 menších chunků při překročení hraniční vzdálenosti vůči nějakému obalujícímu tělesu. Zároveň těchto hranic můžeme určit více a tedy přidat několik úrovní detailu.

V rámci projektu je level of detail pro marching cubes implementován pomocí stromu (z nátury marching cubes vyplývá octree). Každý chunk v sobě obsahuje strom, v němž každá úroveň reprezentuje jednu úroveň LOD. Při přesažení vzdálenosti je chunk rozdělen na menší. Ukázka je na obrázku 4. Díky tomuto přístupu můžeme chunk v různých částech rozdělit do různých úrovní LOD.



Obrázek 4: Tři úrovně LOD

Při procházení stromu pro již existující chunky může nastat několik situací:

1. LOD pro větev je stejný – nic se nemění.
2. LOD pro větev by měl být snížen (nižší detail) – recyklujeme chunky použité pro vyšší LOD a spočteme mesh pro úroveň nižší.
3. LOD pro větev by měl být zvýšen (vyšší detail) – recyklujeme chunky použité pro nižší LOD a spočteme mesh pro úroveň vyšší.

Tento proces je nutné v každém snímku provést pro všechny neprázdné chunky.

Detaily implementace

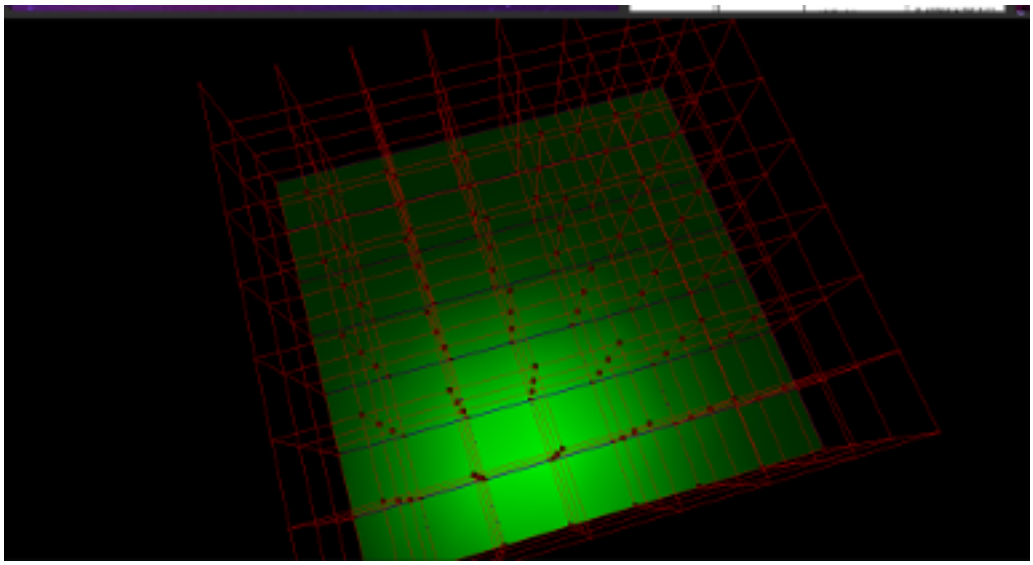
Vzhledem k tomu, že grafická karta bohužel nemá neomezené množství operační paměti, je nutné omezit množství možných chunků ve scéně. Proto je vytvořen pool chunků, které se střídavě používají. Pokud nejsou ve snímku žádné dostupné tak není možné generovat žádný další terén – až po recyklaci některých využitých, ať už pomocí LOD nebo vzdálenosti, je možné pokračovat v dalším generování. Zároveň je také omezeno množství chunků, které je možné v každém snímku přidat do fronty výpočtu z důvodu udržení snesitelného FPS.

Vykreslování

Krom níže zmíněných technik je použito např. View frustum culling. Jedná se ale o základní techniky, které tu nebudu popisovat.

Osvětlení

Osvětlení je řešeno podle Phongova modelu[1]. Jelikož vykreslovaná oblast je poměrně velká je ignorována pozice světla a je použito pouze jeho směru. Na obrázku 5 je vidět vliv osvětlovacího modelu.



Obrázek 5: Ukázka osvětlení

Stíny

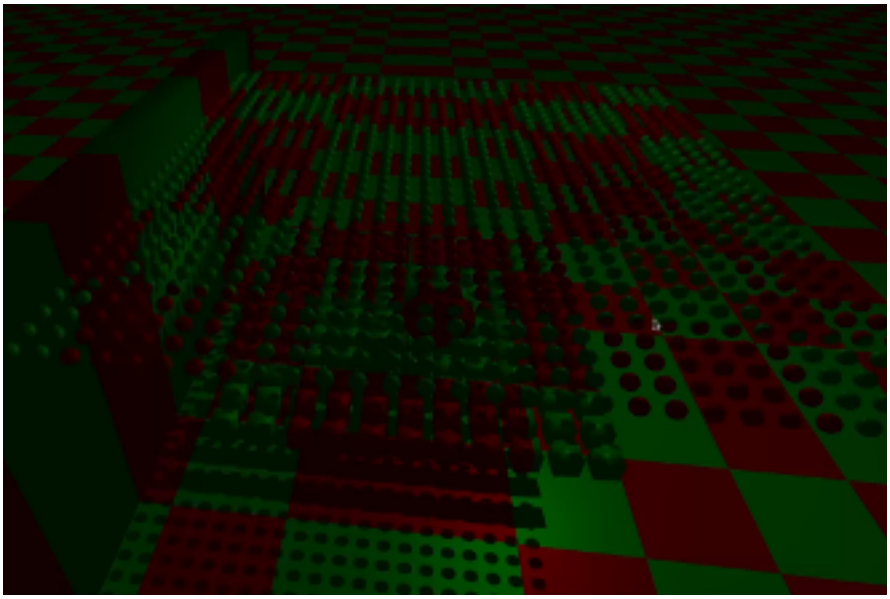
Jelikož terén obsahuje velké množství polygonů na velkém prostoru zvolil jsem shadow mapping jako metodu pro tvorbu stínů.

Shadow mapping

Shadow mapping funguje na principu vykreslení hloubkové mapy scény z pohledu světla a následném využití této mapy při vykreslování, kdy pokud je fragment dále od světla než hodnota

[1] https://en.wikipedia.org/wiki/Phong_reflection_model

uložená v stínové mapě, je ve stínu.



Obrázek 6: Stíny pomocí shadow mapping

Jelikož při samplování textury a vykreslení stínů vznikají chyby, je nutné je nějak kompenzovat. Prvním problémem jsou okraje stínů, které mohou být "hranaté". To lze vyřešit aplikováním jednoduchého blur kernelu nad texturu, což rozmaže okraje.

Dalším problémem je "sebe stínění". Opět problém, který lze vyřešit přidáním tzv. "bias", což je hodnota odečítaná od vzdálenosti objektu od světla. Tím efektivně posuneme fragment blíže ke světlu a nedochází k tomuto jevu.

Pro terén samotný je ale tento přístup nedostatečný. Pokud bychom chtěli pokrýt dostatečně velkou oblast, měly by stíny velmi malé rozlišení a docházelo by ke grafickým glitchům.

Cascaded shadow mapping

Jedná se o rozšíření shadow mapping. Namísto pevně dané oblasti pro stínovou mapu dělíme view frustum kamery do několika oblastí (obrázek 7). Pro každou z těchto oblastí vytvoříme ortogonální projekční matici a vykreslíme do ní stínovou mapu scény. Scéna musí být opakovaně vykreslena tolikrát kolik máme kaskád.

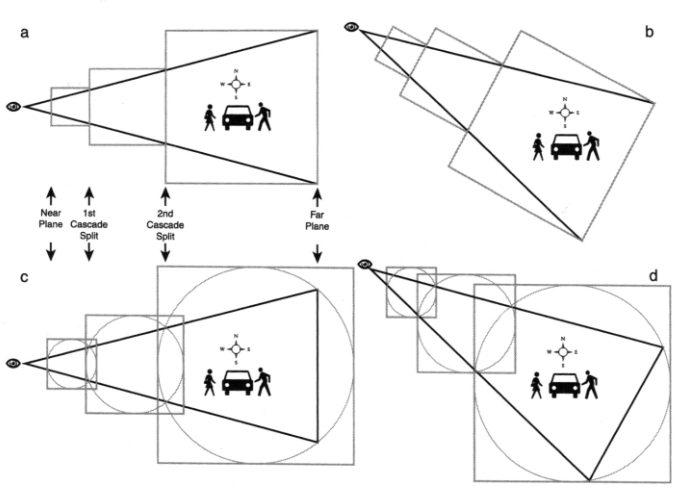
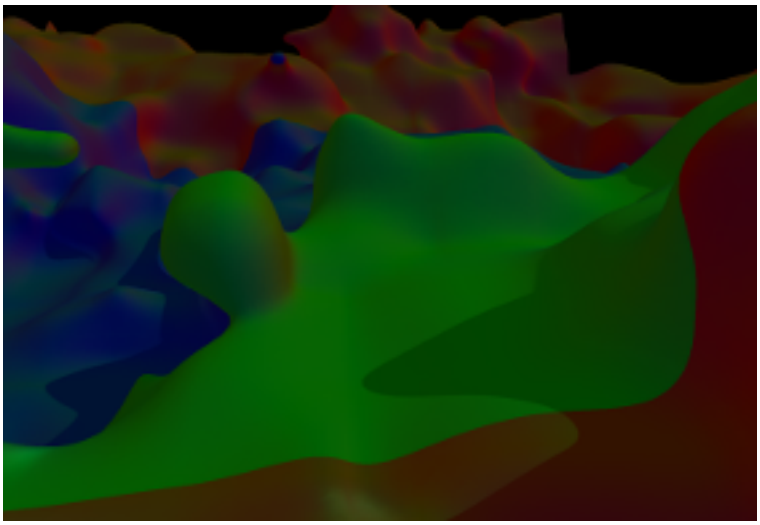


FIGURE 4.1.2 The view frustum in world space split into three cascade frustums and their corresponding shadow map coverage. We use a top view with the light direction pointing straight down the horizontal world plane.

Obrázek 7: Dělba view frusta na kaskády, zdroj: https://johanmedestrom.wordpress.com/2016/03/18/opengl-cascaded-shadow-maps/

Samotné vykreslování s kaskádami probíhá obdobně jako v obyčejném shadow mapping pouze s tím rozdílem, že vybíráme ze které textury samplujeme na základě vzdálenosti fragmentu od kamery. Na obrázku 8 jsou barevně vyznačeny oblasti jednotlivých kaskád – v tomto případě se jedná o 4 kaskády.

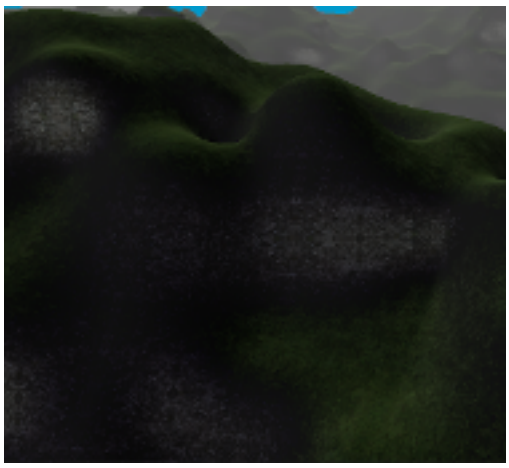


Obrázek 8: Rozdělené scény na kaskády - každá kaskáda je barevně odlišena

Texturování terénu

Jelikož k terénu není možné vytvořit texturovací koordináty jsou naše možnosti značně omezeny. Jednou možností je aplikace procedurální textury. Ukázka tohoto přístupu je na obrázku 6.

Alternativním přístupem je nanášení textur na základě orientace normály povrchu – trilineární texturování. Můžeme tedy například použít 3 textury, kdy každá z nich je použita pro jeden směr (X, Y, Z). S tímto přístupem můžeme vytvořit poměrně věrohodný povrch. Texturovací koordináty jsou dány pozicí fragmentu v prostoru (tedy například pro osu X se bude jedna o složky y a z) a váha samplované textury je určena délkou normály v dané ose. Výsledek je na obrázku 9.



Obrázek 9: Aplikovaná textura

Prostředí

Pro větší realismus jsem implementoval 2 jednoduché úpravy prostředí.

Mraky

Mraky jsou vykreslovány pomocí mapování normál ze zdrojového šumu. Pro každý fragment je samplován šum pro získání jeho gradientu, ze kterého normalizací vytvoříme normálu. Po získání normály vypočteme hodnotu difuzní složky osvětlovacího modelu. Ta je použita jako barevná hodnota a zároveň hodnota alfa kanálu. Jedná se o velmi jednoduchý přístup, ale výsledek je poměrně uspokojivý. Vhodným rozšířením by bylo využití skybox namísto pouze roviny nad kamerou.

Obrázek 10: Mraky

Mlha

V projektu je také implementována mlha. Hodnota mlhy pro fragment je dána rovnicí 2.

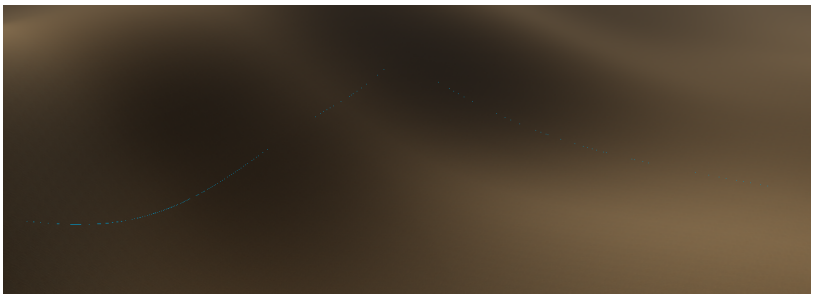
$$f = \frac{fog_{end} - vertexCameraDistance}{fog_{end} - fog_{start}} \tag{2}$$

Obrázek 11: Mlha

Nedostatky

Hlavním nedostatkem implementace je rapidní pád FPS při přesunu mezi oblastmi okolí. V tomto případě se musí spočítat velké množství chunků z nově nastavené oblasti, po krátké chvíli se ale počet snímků za sekundu ustálí.

Druhým nedostatkem jsou přechody mezi úrovni level of detail – tento přechod je vidět na obrázku níže.



Obrázek 12: Přechod mezi LOD

Použité nástroje a knihovny

Nástroje

- CMake[2]
- CLion[3]
- Git[4]
- ClangFormat[5]
- Perf[6]
- Valgrind[7]
- Google sanitisers[8]

Knihovny

- geGL[9]
- sdl2cpp[10]
- SDL2[11]
- {fmt}[12]
- FreeType[13]
- FreeType GL[14]
- glm[15]
- nlohmann::json[16]
- observable[17]

[2] https://cmake.org/
[3] https://www.jetbrains.com/clion/
[4] https://git-scm.com/
[5] https://clang.llvm.org/docs/ClangFormat.html
[6] http://man7.org/linux/man-pages/man1/perf.1.html
[7] https://valgrind.org/
[8] https://github.com/google/sanitizers
[9] https://github.com/dormon/geGL
[10] https://github.com/dormon/SDL2CPP
[11] https://www.libsdl.org/
[12] https://github.com/fmtlib/fmt
[13] https://www.freetype.org/index.html
[14] https://github.com/rougier/freetype-gl
[15] https://glm.g-truc.net/0.9.9/index.html
[16] https://github.com/nlohmann/json
[17] https://github.com/ddinu/observable

- tinyobjloader[18]
- tinyxml2[19]

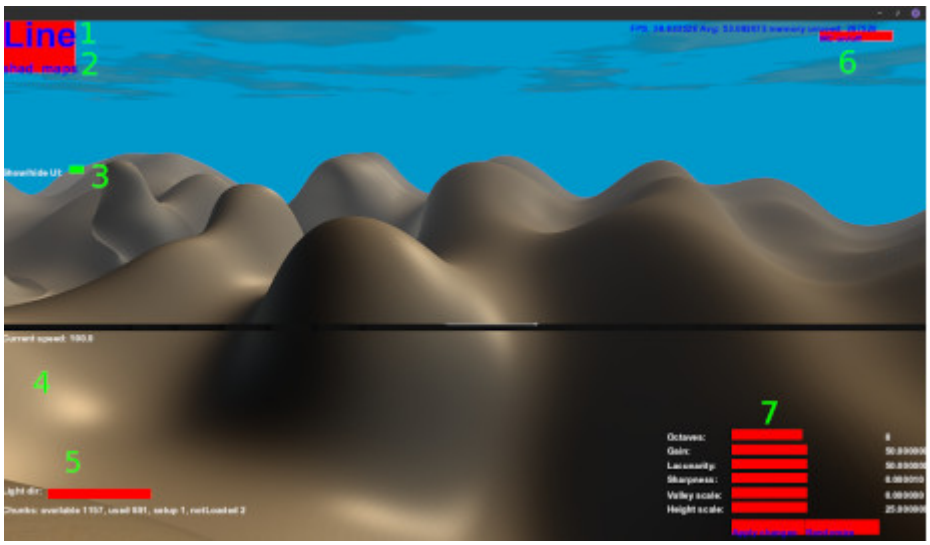
Struktura projektu

- `assets`
  - `gui` – soubory pro uživatelské rozhraní
    - `layout` – XML definice UI (nedokončeno)
    - `fonts` – *.ttf soubory
  - `models` – OBJ soubory pro modely
  - `scenes` – XML soubory pro definici sceny (částečně dokončeno)
  - `textures` – textury
- `include` – zdrojové soubory třetích stran
- `rendering`
  - `environment` – vykreslování okolí (mraky, voda…)
  - `marching_cubes` – třídy pracující s chunky (LOD, chunk management, nekonečný svět…)
  - `models` – třídy pro načítání a práci s modely
  - `shaders` – GLSL shadery
  - `shadow_maps` – třídy pro (cascaded) shadow mapping
  - `textures` – třídy pro načítání a debugging textur
- `ui` – knihovna pro uživatelské rozhraní (více v další sekci)
- `utils` – utility knihovna (více v další sekci)

[18] https://github.com/syoyo/tinyobjloader
[19] https://github.com/leethomason/tinyxml2

Překlad a ovládání programu

Pro návod k překladu se prosím obraťte na soubor `ReadMe.md`, který se nachází u zdrojových souborů. V souboru `config.md` najdete informace o významu jednotlivých položek konfiguračního souboru.



Obrázek 13: Ovládací prvky programu

1. Tlačítko pro přepnutí mezi polygon/line vykreslováním.
2. Tlačítko pro zobrazení texture cascaded shadow mapping
3. Switch pro skrytí/zobrazení UI
4. Slider pro nastavení rychlosti pohybu kamery
5. Slider pro nastavení směru světla
6. Tlačítko pro vypnutí/zapnutí generování nových chunků a LOD
7. Možnosti nastavení funkce pro generování
    - Octaves – počet oktáv FBM
    - Gain – modifikátor amplitudy
    - Lacunarity – modifikátor frekvence
    - Sharpness – záporné hodnoty přidávají váhu ridge noise, kladné billow noise
    - Valley scale – modifikátor velikosti údolí
    - Height scale – škálování výšky
    - Randomize – náhodně nastaví hodnoty a vygeneruje nově nastavený terén
    - Apply changes – vygeneruje nově nastavený terén

Kamera je ovládána pomocí myši a klávesnice. Pro zachycení myši do kamery držte pravé tlačítko myši v libovolné oblasti místo ostatní prvky UI. Kamera udržuje focus dokud nepoužijete jiný UI prvek, poté opět vyžaduje například kliknutí do zmíněné volné oblasti.

Pomocné nástroje vytvořené v rámci projektu

UI

Vrstva postavená nad sdl2cpp. Zpracovává příchozí eventy z SDL, transformuje je a provádí distpatch pro UI komponenty (implementovány jsou například tlačítka, camera controller, switch, label…). Také podporuje focus systém, viditelnost, překrývání atp.

Využívá knihovny observable pro snadnou práci s měnícími se hodnotami (reaktivní prorgamování).

Pomocí připravených rozhraní lze snadno implementovat nové komponenty. Knihovna nabízí rozhraní pro interakci s myší a klávesnicí. Podporuje i vykreslování textu a využívání různých fontů.

V plánu je rozšířit možnosti definice UI prvků pomocí XML souboru a také změna vykreslování na policy based design kdy bude moct programátor vytvořit vlastní vykreslovací funkce pro komponenty bez nutnosti redefinice jejich chování.

Vykreslování modelů

Pro testovací účely jsem vytvořil rozhraní pro načítání a vykreslování modelů načtených z OBJ souborů. Knihovna podporuje definici jednoduché scény za pomocí XML souboru (scéna na obrázku 6 byla definována takto).

Utilities

Utility knihovna obsahující třídy pro geometrii (bounding volumes, view frustum…), výjimky pro error handling s využitím `std::source_location`, tree strukturu využitou pro level of detail, nástroje pro meta programování a mnoho dalších (např. obdoba Python `range` funkce[20]).

Zhodnocení

Testováno na AMD Ryzen 5 3600, NVidia GTX 660 Ti.

Rychlost

Průměrná doba výpočtu jednoho chunku je přibližně 0.5 ms. Pokud bychom tedy pouze generovali, dokážeme vygenerovat až 2000 chunků za sekundu.

Doba výpočtu kontroly viditelnosti a level of detail je odlišná podle vzdálenosti kamery od povrchu a samozřejmě vykreslovací vzdálenosti a mnoha další parametrů. Pro testovací scénu jsou hodnoty následující:

Pokud není žádný chunk rozdělen na vyšší LOD je průměrná doba kontroly přibližně 2.1 ms. V případě rozdělení terénu pomocí LOD se tato doba zvedla o přibližně 40% na 2.8ms.

Celková průměrná doba vykreslení jednoho snímků s využitím stínů (4 kaskády), osvětlení a okolí (mraky, mlha…) a 800 chunků terénu je v klidovém stavu přibližně 6ms. Při běžném pohybu kamery je doba výrazně delší, přibližně 35ms, kvůli zvýšenému množství kontrol, přepočítávání chunků a jejich případné recyklace. Se zvyšující se rychlostí pohybu se samozřejmě zvyšuje množství změn v každém snímku a tím se zvyšuje doba výpočtu.

[20] https://docs.python.org/3/library/functions.html#func-range

Při běžných rychlostech se tedy FPS pohybuje v intervalu 30-60.

Paměťová náročnost

VRAM

Pro každý chunk je možné, že využije maximálního množství vertexů, které lze v oblasti vygenerovat. V každém voxelu může vzniknout až 15 vertexů, tedy i 15 normal. Tím je dána maximální možná paměťová náročnost na chunk $max = 32 * 32 * 32 * 15 * 2$. Reálně k takovému využití paměti nedojde a většina vypočtených chunků je prázdná či obsahuje maximálně pětinu maximální kapacity. Reálně je tedy náročnost na chunk rovna $\frac{32*32*32*15*2}{5} = 196608$ float hodnot, tedy 786432 bytů/chunk. Buffery jsou implementovány se sparse buffer rozšířením pro možnost dynamické práce s pamětí.

Při výpočtech jsou nárazově využity buffery pro uložení hustoty, které mají velikost 413696 bytů/aktuálně počítaný chunk.

Další náročnou položkou jsou textury stínových map. Každá textura má velikost 4096x4096 pixelů s jedním float kanálem, tedy 67108864 bytů/texturu.

Dále samozřejmě případně další modely ve scéně.

RAM

Využití operační paměti se, podobně jako VRAM, odvíjí od dohledové vzdálenosti, nastavení LOD atp. Je ale výrazně nižší, než využití VRAM. Nejnáročnější na operační paměť jsou stromové struktury pro LOD, kdy s každou narůstající LOD úrovní vzniká 8x více uzlů stromu než v předchozí úrovni.

Pro každou oblast musíme udržovat informace o bounding volumes a pozici chunku a další řídící hodnoty. Pro každý uzel stromu tedy zabereme 22 bytů (nepočítaje samotnou stromovou strukturu, která přidává 8 bytů pro každý uzel). Výsledná velikost stromu v paměti pro jeden chunk se třemi LOD úrovněmi je tedy 2190 bytů. V případě, že máme v každé okolní oblasti 10 chunků dostáváme 27000 chunků. Po vynásobení se dostáváme na 59130000 bytů pro celé okolí.

Aplikace samozřejmě využívá spoustu paměti pro další operace. Za pomocí valgrind jsem zjistil, že využívá cca 300 MB RAM paměti.

Závěr

Určitě by bylo vhodné nějak snížit paměťovou náročnost na GPU, aby bylo možné generovat rozsáhlejší terén. Dále by bylo vhodné zvolit lepší osvětlovací model a také přesunout kontrolu LOD úrovní na GPU. Na druhou stranu je dle mého generování poměrně svižné, v terénu nedochází ke grafickým glitchům – krom přechodů mezi LOD – a s trochou další práce by bylo možné přechody mezi oblastmi výrazně zrychlit.

Literatura

[1] Gabriel Costa Backes and Tiago Augusto Engel. Real-time massive terrain generation using procedural erosion on the gpu. 2018.

[2] Hubert Nguyen. *GPU gems 3*. Addison-Wesley, 2008.

[3] NVidia. Cascaded shadow mapping. Navštíveno 18.12.2019, https://docs.nvidia.com/gameworks/content/gameworkslibrary/graphicssamples/opengl_samples/cascadedshadowmapping.htm.