VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

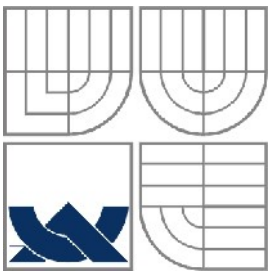
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

REAL-TIME VIZUALIZACE POVĚTRNOSTNÍCH VLIVŮ V TERÉNU

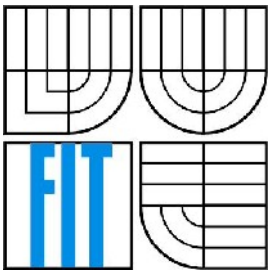
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE BC. ADAM VLČEK
AUTHOR

BRNO 2009



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

REAL-TIME VIZUALIZACE POVĚTRNOSTNÍCH VLIVŮ V TERÉNU

REALTIME WEATHER IN A LANDSCAPE VISUALISATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

BC. ADAM VLČEK

VEDOUCÍ PRÁCE
SUPERVISOR

ING. MICHAL SEEMAN

BRNO 2009

Abstrakt

Díky obrovskému výpočetnímu výkonu se virtuální realita stává stále pestřejší a dynamičtější. Tato práce si klade za cíl prozkoumat vybrané povětrnostní vlivy v terénu, možnosti jejich simulace a dynamického zobrazení v reálném čase na současných osobních počítačích. Cílem je spíš nalezení dobře vypadajících rychlých aproximací než dokonalý fyzikální model. Je zde rozebráno použití moderních programovatelných GPU nejen pro účely zobrazování, ale také jako velmi silný výpočetní prostředek pro simulaci přírodních dějů. Práce je zaměřena zejména na pohyb vody terénem a její vliv na něj. Jedná se například o tání sněhu, erozi nebo výskyt různých druhů rostlin podle preferované vlhkosti. Pro tyto účely je využito dynamické texturování terénu a algoritmy umožňující rychlou úpravou zobrazované geometrie včetně počítání normál.

Abstract

Thanks to the increasing computation power the complexity and dynamism of virtual reality is continuously improving. This work aims to examine influences of weather in a landscape and the means to simulate and dynamically visualize them in real time on the current personal computer hardware. The main goal is to find quick well looking approximations rather than a complex physically correct simulation. The work covers using modern programmable GPU not only for visualization but also as a powerful simulation instrument. The main topic is water movement in the terrain and its effects on it like erosion, snow melting and moisture impact on vegetation. This requires dynamic terrain texturing and algorithms supporting fast geometry and normals updates.

Klíčová slova

povětrnostní vlivy, počasí, terén, výšková mapa, výpočet normál, voda, eroze, déšť, sníh, mlha, vizualizace, simulace, interaktivita, reálný čas, shader, GPU, GPGPU

Keywords

weather effects, terrain, height map, normal computation, water, erosion, rain, snow, fog, visualization, simulation, interactivity, real time, shader, GPU, GPGPU

Citace

Adam Vlček: Real-time vizualizace povětrnostních vlivů v terénu, diplomová práce, Brno, FIT VUT v Brně, 2009

Real-time vizualizace povětrnostních vlivů v terénu

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Michala Seemana. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Adam Vlček
25.5.2009

Poděkování

Rád bych poděkoval svému vedoucímu Ing. Michalu Seemanovi za vstřícný přístup a podnětné rady při řešení tohoto projektu.

Obsah

Obsah....1
1 Úvod....3
2 Simulace a vizualizace....5
2.1 Simulace vody....5
2.1.1 Eroze....6
2.2 GPGPU....7
2.2.1 Výhody....8
2.2.2 Limity....8
2.2.3 Programovací API....9
2.3 Vizualizace....10
2.3.1 Geometrie....10
2.3.2 Výpočet normál....12
2.3.3 Texturování....15
3 Terén....16
3.1 Reprezentace....16
3.2 Geometrie....17
3.3 Generování a modifikace terénu....17
3.4 Textury....18
3.5 Fragment shader terénu....18
3.5.1 Výpočet koeficientů....19
3.5.2 Určení výsledné barvy fragmentu....20
4 Další vizuální efekty....21
4.1 Vegetace....21
4.2 Mlha....23
4.2.1 Terén, voda a vegetace....23
4.2.2 Obloha....24
4.3 Stíny....24
4.4 Obloha....25
4.5 Postprocessing....26
5 Implementace....27
5.1 ZYLib....27
5.2 OpenGL 3.0, rozšíření a GPGPU....27
5.2.1 GPGPU....28
5.3 GLSL shadery....29

5.3.1 Načítání .sp souborů....29
5.3.2 GLSL....30
5.4 Terén a voda....30
5.4.1 Datové textury....30
5.4.2 Zobrazení....31
5.4.3 Shadery....31
5.5 Vegetace....32
5.6 Obloha....33
5.7 Čtení dat z grafické karty....33
5.8 GUI....34
5.9 Problémy....34
6 Výsledky....35
6.1 Simulace vody....35
6.2 Texturování terénu a vody....38
6.3 Vegetace....40
6.4 Mlha....42
7 Další práce....44
8 Závěr....46
Literatura....48
Seznam zkratek....50
Seznam příloh....51
A. Ukázky shaderů....52
B. Manuál....58

1 Úvod

Výkon počítačů stále roste, a tak se virtuální realita stává stále propracovanější a pestřejší. Z původních prakticky statických scén se dnes stává skutečně živé prostředí, které díky simulaci věrohodně reaguje na činnost uživatele, nebo se vyvíjí nezávisle na něm. Cílem této práce je prozkoumat možnosti vizualizace měnících se povětrnostních vlivů v terénu v reálném čase a nakolik je tyto jevy pro přesvědčivý zážitek potřeba simulovat.

Pojem povětrnostní vlivy je poměrně široký, a tak jsem se pokusil již v rámci semestrálního projektu [1] rozebrat jeho význam, vybrat důležité jevy, vyhodnotit možnosti jejich vizualizace a simulace a identifikovat společné rysy vybraných jevů, které pak bude možno využít při implementaci.

Velmi významný činitel v terénu je bezpochyby voda. Její simulací a vizualizací jsem se zabýval již ve své bakalářské práci [2] a nyní je její působení na terén dále rozvíjeno. Kromě tvorby řečišť a jezer tu již má voda na terén také erozivní účinky, rozpouští sníh, určuje množství bahna v daném místě a ovlivňuje výskyt rostlin. Lokální ráz terénu je kromě vody ovlivněn také nadmořskou výškou a sklonem svahů. Na základě těchto několika vstupních údajů je terén texturován pomocí komplexního fragment shaderu za účelem vytvoření iluze krajiny určitého typu, který lze v reálném čase modifikovat jednoduchou změnou několika čísel, a tak terén změnit například ze zasněžených prosluněných kopců v písečnou bouří zmítanou poušť.

Jelikož je simulace jedním z podkladů pro zobrazení, je jí věnována hned následující druhá kapitola, která začíná rozborem metod použitelných pro rychlou simulaci vody a eroze v rozsáhlém terénu. Následují informace o výpočtech na grafické kartě (GPGPU), která dnes dokáže mnohé simulační problémy zvládnout ve zlomcích času spotřebovaného klasickým procesorem, zejména pokud zde probíhá celá simulace a její výsledky jsou kartou rovnou vizualizovány bez potřeby přenosu na CPU, čemuž se věnuji v závěru kapitoly. Součástí je i popis velice rychlého algoritmu pro výpočet normál ve 2D mřížce výškové mapy.

Kapitola třetí je věnována terénu a zejména jeho texturování. Nejdříve je popsána geometrie terénu, jeho generování a procedurální tvorba základních textur. Hlavní náplní kapitoly je popis způsobu, jak využít výstupy simulace pohybu vody krajinou v kombinaci s určováním lokálního rázu terénu podle nadmořské výšky a sklonu svahů pro vytvoření pestrého terénu fragment shaderem.

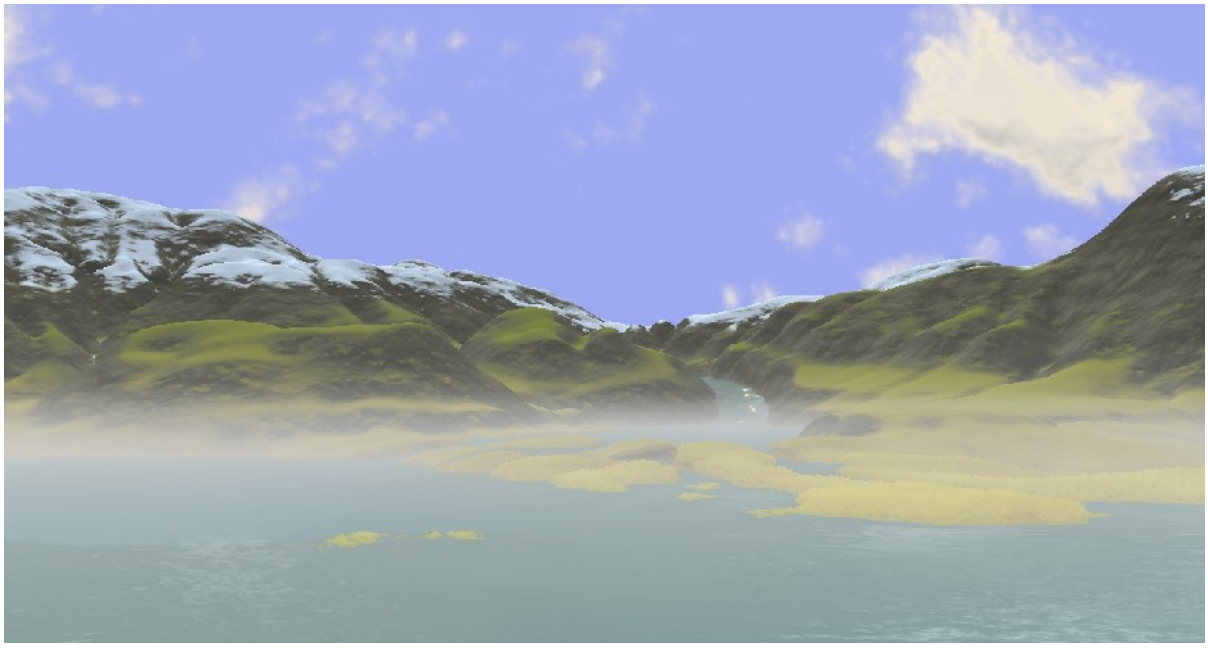
Další vizuální efekty, které spíše doplňují scénu, jsou popsány v kapitole čtvrté. Jedná se zejména o vizualizaci travní vegetace, mlhy, oblohy, stínů a post-processing celé scény. Zobrazení oblohy je postaveno na mém projektu do předmětu PGR (Pokročilá grafika) [3] a post-processing těží z předmětů MUL (Multimédia) [4] a VIN (Výtvarná informatika) [5].

Pátá kapitola shrnuje implementaci demonstračního programu, zejména témata okolo OpenGL, výpočetních shaderů, GPGPU a zobrazení výsledků. V této kapitole je také popsán formát souboru použitý pro definici shaderů a na závěr také zmiňuji problémy a limity demonstračního programu.

Výsledky jsou detailně rozebrány v šesté kapitole, která obsahuje jak textové shrnutí důležitých poznatků, tak mnoho obrázků z demonstrační aplikace, mezi které patří i obrázek 1 na této stránce.

Podobné projekty je možné prakticky neustále rozvíjet, a tak je sedmá kapitola věnována možným vylepšením a předpokládanému dalšímu vývoji projektu.

Závěrečná osmá kapitola již jen shrnuje obsah celé práce a upozorňuje na zvlášť významné výsledky.

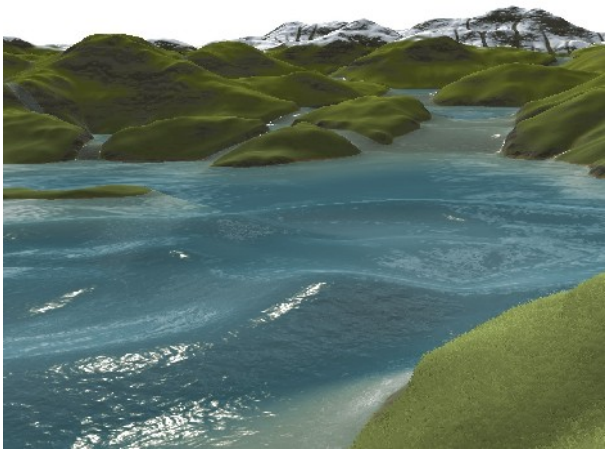


Obrázek 1: Výstup demonstračního programu.

2 Simulace a vizualizace

Pokud chceme dostat realistické vizuální výstupy, je mnohdy potřeba zobrazované děje simulovat. V oblasti virtuální reality však není cílem vytvořit dokonalý fyzikální model, jehož výpočty budou trvat měsíce. Spíše je potřeba najít řešení, které daný problém aproximuje dostatečně věrohodně při minimálních požadavcích na náročnost výpočtů. Vždy je potřeba hledat rovnováhu mezi složitostí výpočtů simulace, náročností vizualizace výsledků a nechat dostatečnou rezervu v prostředcích na ostatní součásti programu.

2.1 Simulace vody



Obrázek 2: Voda z dešťových srážek stéká po terénu, formuje toky a jezera. Do scény může také interaktivně zasahovat uživatel.

Hlavním objektem simulace je zde voda tekoucí terénem. Jelikož terén sám je velmi dynamický, může být měněn uživatelem a voda může působit erozi, není možné využít pro vodu předem spočítaná data a simulaci se proto nevyhneme. Způsobem zde použité simulace se zabývám ve své bakalářské práci [2], takže ho zde nebudu rozebírat příliš do hloubky a spíš upozorním na případné změny.

Zvolenou metodou simulace je jistá obdoba celulárního automatu (CA) [6], kdy každý bod ve 2D mřížce reprezentuje jednu buňku. Tato buňka má svůj stav v podobě výšky podkladového terénu, výšky vodního sloupce a aktuální rychlosti vody. Každá buňka pak ke své činnosti potřebuje kromě svého stavu znát ještě stavy buněk ve svém okolí. Změnu stavu buňky v čase lze popsat funkcí *f*:

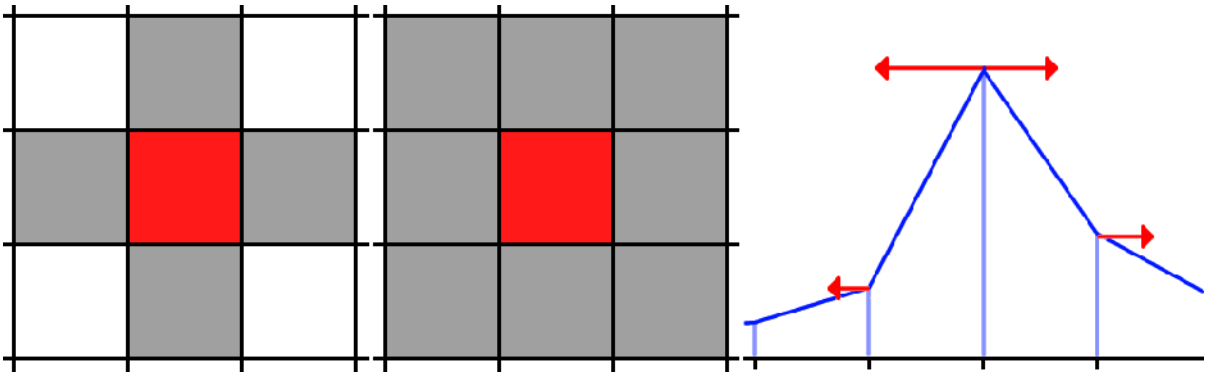
$$s(t+1) = f(s(t), N_S(t)) \quad (1)$$

kde $s(t+1)$ je nově počítaný stav buňky, $s(t)$ aktuální stav buňky a $N_S(t)$ jsou aktuální stavy okolních buněk.

Ve 2D čtvercové sítí je potřeba pracovat nejméně se čtyřmi okolními buňkami ležícími na osách, s takzvaným Von Neumannovým okolím. O něco náročnější alternativou je Mooreovo okolí, kdy se počítá navíc i s buňkami dotýkajícími se počítané buňky svými rohy, tedy dohromady s osmi

buňkami. Tyto dva typy okolí ilustruje obrázek 3. Samozřejmě je možné využít pro výpočty i širší okolí a existují i jiné typy, pro jednoduchou simulaci však dobře poslouží i základní čtyři body.

Každá buňka si pamatuje, jakou rychlostí z ní odtéká voda do okolních buněk. Rychlost přítoku lze stanovit podle rychlostí odtoku ze sousedů. Pro jednu buňku s Von Neumannovým okolím tak dostaneme na dvou osách čtyři rychlosti, které musí být všechny nezáporné. Jedna rychlost pro každou osu nestačí, protože při rozlévání do protilehlých směrů by se rychlosti navzájem rušily. Potřebu dvou rychlostí pro jednu osu ilustruje pravá část obrázku 3. Navíc je takto možné do určité míry aproximovat vícevrstvé proudění a rozvířenost vody.

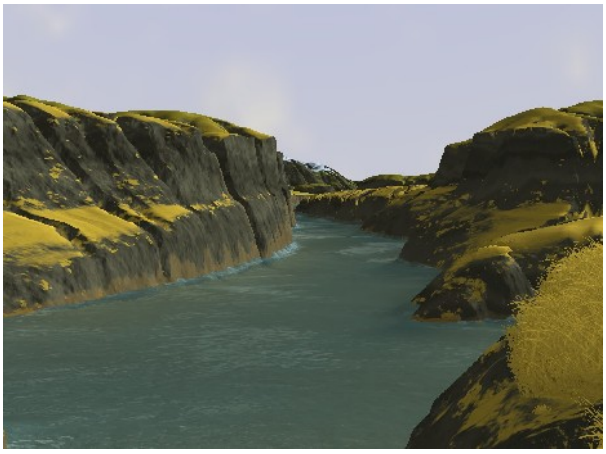


Obrázek 3: Vlevo Von Neumannovo okolí, uprostřed Mooreovo. Vpravo je ukázka rychlostí přelévání vody v buňkách pro 1D.

Rychlosti je potřeba v každém simulačním kroku upravit podle rozdílů hladin v okolních buňkách a následně použít při výpočtu množství přesouvané vody. Každá buňka si spočítá, kolik vody do ní přitéká a kolik odtéká pryč. Přítok je pak přičten k výšce hladiny, zatím co odtok je odečten. Jedná se o klasickou numerickou integraci v podobě Eulerovy metody, která má díky požadavku na interaktivní počet snímků za vteřinu zajištěn dostatečně malý krok a vykazuje i dobrou stabilitu. Přesun samotný navíc vyvolává předávání kinetické energie do cílové buňky, což se opět projevuje změnou rychlostí.

2.1.1 Eroze

Určitých úspěchů je možno dosáhnout prostým ubíráním terénu v místech s rychle proudící vodou, ovšem nebudou se nikde vytvářet náplavy. Přesnější je využít podobný způsob jako při simulaci základní vody, kdy kromě výšky hladiny uchovává každá buňka i množství unášeného materiálu. Ten je ubírán z terénu v místech rychle proudící málo saturované vody a ukládán v oblastech s nižší rychlostí. Předávání materiálu pak probíhá současně s přenosem vody.



Obrázek 4: Eroze může vytvářet zajímavé kanály a kaňony. Jejich hloubku a sklon svahů je však potřeba pro realistický vzhled omezit.

U obou způsobů je potřeba vyřešit přílišné zařezávání koryt a tvorbu nepřirozeně členitého terénu. Navíc má pravidelná mřížka tendenci tvořit kanály převážně podél svých os. Přílišné zařezávání se dá redukovat vyhlazováním terénu v okolí proudící vody, nicméně u větších koryt tento způsob nemusí stačit a je potřeba zavést limit sklonu svahu, od kterého bude docházet k sesunu půdy do údolí. Sesun lze simulovat jednoduchým vyhlazovacím filtrem, jehož intenzita se určí z lokálního svahu, množství vody a její rychlosti. Tento způsob neodpovídá realitě, je ovšem velmi rychlý, nevyžaduje další datové struktury a dává přijatelné vizuální výsledky. O něco náročnější je počítat sesun půdy podobně jako vodu, kdy každá buňka musí spočítat, kolik zeminy se na ni sesune z okolí a kolik se zase odsune z ní. To však vyžaduje buď ukládat rychlosti přesunu v přídavném výpočetním kroku do pomocné datové struktury, nebo pro každou buňku provádět i výpočty okolních buněk.

Dalším aspektem je rozdílná odolnost různých částí terénu k erozi. Nejjednodušší možností je určit tuto tuhost náhodným šumem, například uloženým v textuře. Pokročilejší pak může být reprezentace vrstev v programu přidáním třetího rozměru do reprezentace terénu.

Realistickou erozí půdy s 3D voxelovou reprezentací a vizualizací výsledků se zabývá například Beneš [7], erozi působící vodou reprezentovanou částicemi a akcelerovanou na GPU se zabývá spolu s Kristofem [8]. Mnoho dalších zajímavých publikací obecně o simulací a vizualizaci povětrnostních vlivů je možno najít na Benešových stránkách [9].

2.2 GPGPU

Výkon grafických karet poslední dobou roste daleko rychleji než výkon klasických procesorů. Již delší dobu je možné grafické karty programovat a jejich schopnosti v poslední době dokážou výkon určitých aplikací zvednout více než o řád oproti stejnému úkolu řešenému na CPU. Mezi aplikace s největším potenciálem pro urychlení patří takzvaně datově paralelní úlohy, kdy se nad velkým množstvím dat provádí stejné operace bez nutnosti komunikace jednotlivých výpočetních vláken. Mezi takovéto aplikace patří i celulární automaty a zejména zde použitá simulace vodní hladiny

v pravidelné 2D mřížce přímo ideálně koresponduje se současným grafickým hardware, kdy každá buňka je reprezentována jedním texelem v datové textuře.

2.2.1 Výhody

Simulování vody na GPU má hned několik velmi významných výhod:

- CPU je významně ulehčeno a může se věnovat jiným úkolům, které jsou pro výpočet na GPU méně vhodné.
- Výrazně se omezí komunikace mezi CPU a GPU, protože není potřeba po každém výpočetním kroku zasílat na grafickou kartu nová data vrcholů, což výkon celé aplikace znatelně zvýší.
- Řešený problém velice dobře koresponduje s úkony, pro které jsou současné grafické karty optimalizovány.
- Spočítaná data je možno ihned zobrazit z rychlé paměti GPU.

2.2.2 Limity

Výpočty na GPU mají samozřejmě i své limity, které by se mohly projevit zejména při řešení složitějších problémů. Mnoho z nich však již lze nyní na nejnovějším hardware překonat například pomocí architektury CUDA [10]. Zde používané řešení se navíc téměř všem předem známým problémům vyhýbá využitím méně náročných a přenositelných postupů.

- Jediným opravdovým limitem zde rozebíraných postupů je nutnost vlastnit moderní grafickou kartu. Ta musí být schopná pracovat s texturami v plovoucí řádové čárce a některé doplňující techniky vyžadují i geometrické shadery a plnou podporu celočíselných typů.
- Zásadní limit bývá komunikace mezi CPU a GPU. Pokud na relativně malé množství výpočtů připadne příliš mnoho komunikace, může dokonce „akcelerace“ na GPU výpočty zpomalit. Navrhované postupy nevyžadují komunikaci téměř žádnou, ovšem velký význam má velikost paměti GPU. Pokud se do ní nevejdou veškerá data, je výkon systému znatelně omezen nutností výpočty rozdělit a paměť stránkovat.
- Nemožnost komunikace mezi vlákny. V rámci klasických shaderů neexistuje žádná sdílená paměť. Pomocí architektury CUDA je v současné době možno využít určitý limitovaný paměťový prostor, který je optimalizován pro paralelní výpočty. Jeho kapacita však není příliš velká a i přes optimalizace může práce s touto pamětí mírně zpomalovat (bank conflict). Buňky CA naštěstí potřebují pouze znát stav svého okolí, který je před spuštěním paralelního výpočtu uložen v paměti GPU jako textura.

- Nemožnost využít ukazatele, rekurzi a dynamicky alokované struktury. Opět již neplatí při využití architektury CUDA. Výpočty změny stavů buněk zde využívaných CA však takové funkce nepotřebují.
- Ve svých počátcích CUDA nebyla schopná kooperovat s OpenGL a DirectX. Zde použité algoritmy lze snadno popsat i klasickým grafickým API, takže není důvod přidávat další.
- CUDA je dostupná pouze na hardware od firmy NVIDIA. Alternativou může být OpenCL, které je také schopné s OpenGL kooperovat. Jedná se ovšem o poměrně novou záležitost a implementace GPGPU v klasických shaderech je stále univerzálnější.

2.2.3 Programovací API

Programovat GPU lze samozřejmě pomocí grafických API, jako je zejména OpenGL [11] a DirectX [12]. Kromě grafických API již dnes existují i čistě výpočetní, jako je zejména CUDA [10] od firmy NVIDIA a OpenCL [13] pro heterogenní výpočetní platformy.

- OpenGL: Využívá jazyk GLSL [14] a jeho hlavní výhodou je vysoká přenositelnost, na rozdíl od DirectX, které ve své poslední verzi funguje pouze na Windows Vista. Pomocí systému rozšíření je možno v OpenGL využít i nejnovější funkce grafických karet na kterémkoli systému.
- DirectX: Pro programování GPU slouží zejména Direct3D s jazykem HLSL [15]. V současnosti jsou výpočty na GPU pomocí Direct3D snad ještě limitovanější než v OpenGL, nicméně Microsoft do DirectX 11 chystá takzvané „Compute Shadery“, které by měly GPGPU usnadnit.
- CUDA: Velmi propracované API umožňující paralelní výpočty na GPU. V současné době na novém hardware umožňuje plně využít schopností grafických karet, včetně zápisů na předem nespecifikovanou adresu, paměť sdílenou všemi vlákny, využití ukazatelů a dynamických struktur. Navíc je možné propojit CUDA s OpenGL nebo DirectX a výsledky výpočtů přímo použít při zobrazování. Hlavní nevýhodou je dostupnost pouze na kartách od firmy NVIDIA. Veškerá potřebná dokumentace, tutoriály a další materiály jsou dostupné na webu Cuda Zone [10].
- OpenCL: Slouží pro programování heterogenních paralelních platforem složených z CPU, GPU a jiných procesorů. Jedná se o poměrně novou záležitost, jejíž hlavní výhodou by měla být podpora podstatně většího množství hardware než je tomu u CUDA, které je OpenCL celkem podobné. Jelikož OpenCL spadá pod stejnou organizaci jako OpenGL, je možné mezi nimi také sdílet data.

Jelikož je tento projekt silně zaměřen na vizualizaci a simulační problém je možno velmi dobře popsat pomocí klasických shaderů, rozhodl jsem se plně využít OpenGL a shadery napsat v GLSL. Jak takový výpočetní shader v GLSL vypadá, je ukázáno v příloze A – Ukázky shaderů.

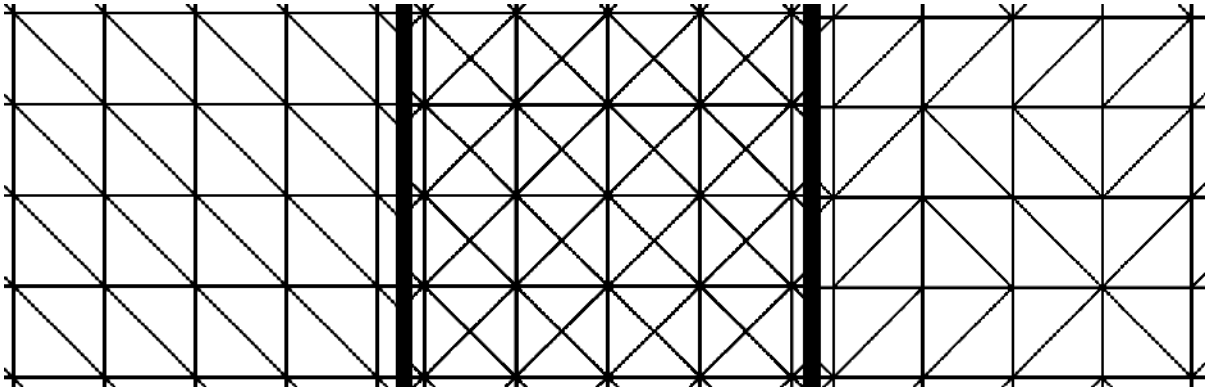
2.3 Vizualizace

Způsob simulace značně ovlivňuje techniku zobrazení. Zároveň lze dobrým zobrazením zakrýt mnohé nedostatky simulace. Zde využitý CA na pravidelné 2D mřížce je ideální pro zobrazení povrchu vody pomocí sítě čtverců. Mnoho užitečných postupů (nejen) pro vizualizaci vody se dá najít například v sérii knih GPU Gems [16][17][18], které jsou mimo jiné dostupné zdarma na webu firmy NVIDIA.

2.3.1 Geometrie

První problém je, že čtverce musí být rozděleny na trojúhelníky, což může být provedeno několika způsoby.

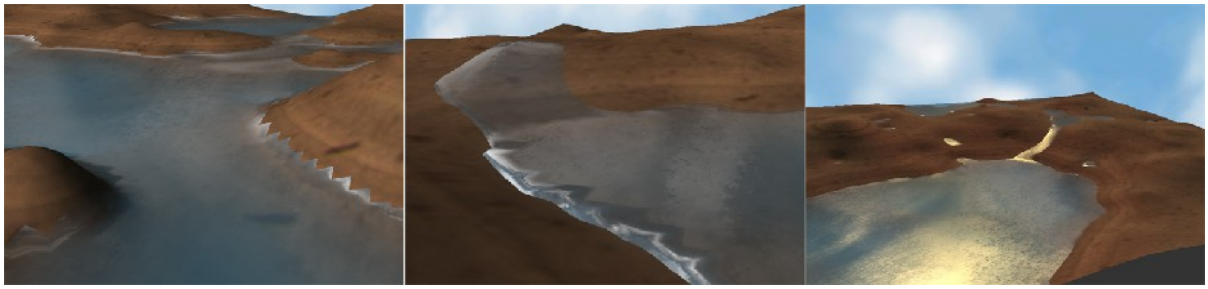
- Rozdělit každý čtverec na dva trojúhelníky pokaždé stejně. Na terénu však může být tato pravidelnost někdy příliš patrná, zejména na prudších zlomech.
- Rozdělit čtverec na čtyři trojúhelníky. Zde zase musíme přidat středový vrchol, pro který se musí správně určit nadmořská výška a normála, což lze provést například interpolací hodnot z rohů. Nevýhodou je zejména dvojnásobné množství trojúhelníků, které má nezanedbatelný vliv na výkon.
- Rozbití pravidelnosti při zachování dvou trojúhelníků na čtverec lze dosáhnout například náhodným střídáním dělení čtverce po hlavní nebo vedlejší diagonále.



Obrázek 5: Dělení čtverců na trojúhelníky. Zleva: na 2 pokaždé stejně, na 4 s přidaným středovým bodem a na 2 s náhodnou diagonálou.

Dalším problémem je zobrazení vodní hladiny na terénu, kde je potřeba zobrazovat vodu až od určité výšky vodního sloupce, protože nízká hladina reprezentuje vodu tekoucí v půdě. Jednoduché posunutí hladiny o určitou konstantu dolů nefunguje hned ze dvou důvodů:

1. Pokud je rozdíl úrovně terénu a vodní hladiny nedostatečný, projevuje se omezená přesnost Z-bufferu prolínáním těchto dvou vrstev.
2. Na hraně rovných vodních ploch, jako jsou zejména jezera s prudšími břehy, se takto posunutá hladina nepřirozeně zvedá a vytváří zubaté okraje.



Obrázek 6: Vlevo pouze posun vodní hladiny o určitou konstantu, uprostřed posun podle výšky vodní hladiny, vpravo zde popsaný algoritmus.

Z těchto důvodů je vhodné upravit zobrazovanou výšku vodní hladiny v místech s nízkou hladinou, tak aby byla znatelně pod terénem, avšak nechat skutečnou výšku jinde. Pokud se pod terénem nachází celý trojúhelník, nemusí být samozřejmě vykreslen vůbec. Na okrajích rovných ploch je také vhodné přiřadit bodu s hladinou pod zemí výšku hladiny sousedních bodů s hladinou nad terénem. Tyto změny se navíc musí provádět plynule, protože rychlé změny v animaci nepůsobí dobře. Výsledkem experimentů s různými postupy je následující algoritmus (1):

Vstup: T – výška terénu v počítaném bodě, W – vodní sloupec, Z – původní zobrazovaná nadmořská výška hladiny, T_i – výšky terénu v okolí , W_i – vodní sloupce v okolí, Z_i – nadmořské výšky hladin v okolí ($Z_i = T_i + W_i$)

Výstup: Vyhlazená hodnota nadmořské výšky, ve které se bude voda vykreslovat v daném bodě Z_{out}.

Metoda:

1. Přiřaď výstupní hodnotě původní výšku hladiny:
   $Z_{out} = Z$
2. Pokud je hladina vyšší než stanovené minimum *MINW*, ukonči algoritmus.
3. Inicializuj proměnné pro vyhlazování:
   $W_{accum} = cnt1 = cnt2 = 0$

4. Spočti pro všechny body Mooreova okolí příznaky a průměry potřebné pro vyhlazení podle následujícího postupu:
   a) Pokud je $Z_i < Z$, proveď následující přičtení:
      $f = W_i * (1 / MINW)$
      $W_{accum} = W_{accum} + Z_i * f$
      $cnt1 = cnt1 + f$
   b) V ostatních případech :
      $cnt2 = cnt2 + 1$
5. Pokud je $cnt1 > 0$, spočítej výstupní hodnotu takto:
   $f = W * (1 / MINW)$
   $Z_{out} = Z * f + (W_{accum} / cnt1) * (1 - f)$
6. Pokud předchozí podmínka neplatí a $cnt2 < 8$, spočítej výsledek takto:
   $Z_{out} = T + W * (1 + DIST / MINW) - DIST$

Krok 2 slouží k eliminaci vyhlazování v místech, kde není potřeba. Ve 4. kroku nestačí počítat pouze s Von Neumannovým okolím, protože zobrazované okolní trojúhelníky mají vrcholy ve všech buňkách Mooreova okolí a vynechání rohových bodů vytváří artefakty. Krok 5 provádí vyhlazování pouze s body, jejichž hladina je níže než hladina počítaného bodu. Vstupní hladina totiž nemůže klesnout pod úroveň terénu, a tak by se mohly do průměru místo hladiny vody počítat výšky terénu, čemuž se právě snažíme vyhnout. Krok 6 slouží pro odsazení pod zemí ležící hladiny od terénu o vzdálenost *DIST*, které je nutné kvůli eliminaci Z-fightingu v případě, že je všude v okolí sucho, což samotný krok 5 nezaručí. Přídavná podmínka s *cnt2* odstraňuje problém s propadáním vodní hladiny ve středu mělkých malých louží, kdy by jinak vyhlazená hladina okraje převyšovala střed.

Shader obsahující tento algoritmus je obsažen v příloze A jako ukázka výpočetního shaderu. Vertex shader může na základě spočítaných dat uložených do pomocné datové textury upravit pozici vrcholů ze statického pole v podobě 2D mřížky uložené v paměti GPU do zakřivené plochy a spočítat normály.

2.3.2 Výpočet normál

Výpočet normál je většinou prováděn pomocí několika vektorových součinů počítajících normály okolních trojúhelníků, které se následně kombinují do výsledné normály bodu. Na pravidelné pravoúhlé mřížce reprezentující výškovou mapu lze normály získat i daleko rychleji pomocí následují jednoduché rovnice popsané v GLSL:

```
vec3 normal = normalize( vec3( zm0 - zp0, z0m - z0p, 2.0 ) );
```
(2)

Ten vychází z výpočtu kolmice k přímce ve 2D. Pokud přímka prochází počátkem kartézské soustavy a nějakým bodem (x, y), její počátkem procházející kolmice musí protnout bod (-y, x). Výšková mapa má 2D souřadnice X, Y a výška je udávaná jako Z. Proměnné *z*, *zm0*, *zp0*, *z0m* a *z0p* reprezentují nadmořské výšky středového bodu a sousedů vlevo, vpravo, dole a nahoře.

Způsob výpočtu bude demonstrován v rovině XZ. Jelikož rozestupy ve 2D mřížce jsou rovny 1, dají se kolmice ke svahu z centrálního bodu do pravého a levého souseda vypočítat následovně:

$$vec_{xzp} = (- (zp0 - z), 1) = (z - zp0, 1) \quad (3)$$

$$vec_{xzm} = (zm0 - z , 1) \quad (4)$$

Souřadnice *x* pro levého souseda se v rovnici 4 neneguje, protože svah ukazuje do opačného směru. Nejlepší výsledky s minimem výpočtů pak dává prosté sečtení souřadnic v ose X a ponechání Z = 1. Tak vznikne rovnice 5, kterou je možno upravit do jednoduchého tvaru 6.

$$vec_{xz} = (z - zp0 + zm0 - z, 0, 1) \quad (5)$$

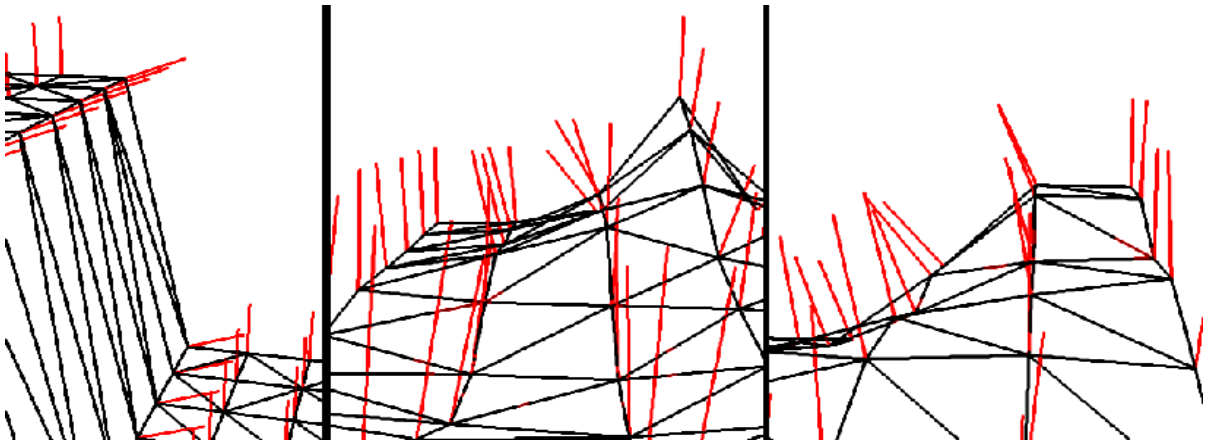
$$vec_{xz} = (zm0 - zp0, 0, 1) \quad (6)$$

Stejně lze spočítat i normálu v ose Y.

$$vec_{yz} = (0, z0m - z0p, 1) \quad (7)$$

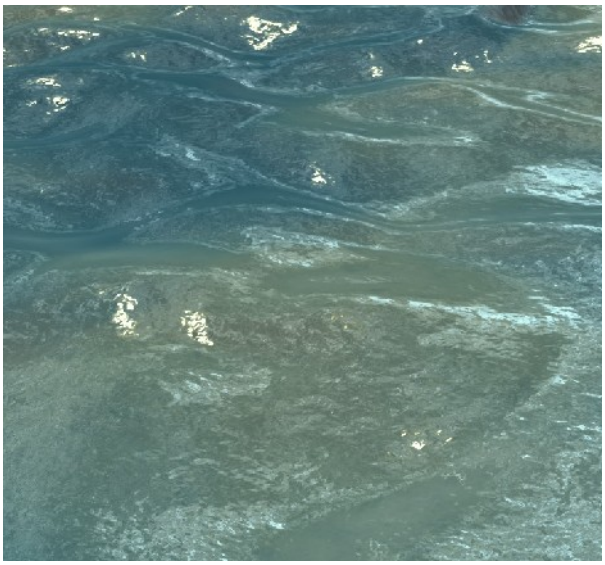
Výsledné vektory vec_{xz} a vec_{yz} již pouze sečteme a výsledek znormalizujeme jako v rovnici 2, tedy použijeme dvě odečtení a jednu normalizaci, což jsou na současném grafickém hardware velmi rychlé operace. Dostaneme tak sice výsledek mírně odlišný od průměru normál spočítaných pro okolní trojúhelníky pomocí vektorových součinů, ale zde uvedený algoritmus je podstatně rychlejší a jeho výsledky subjektivně lépe vystihují reálný tvar terénu, kdy se na prudkém zlomu více projeví dlouhá rovná stěna než krátká sousední rovina, což má za následek korektně osvětlenou větší plochu.

U výpočtů normál vody je navíc vhodné brát v úvahu pozici hladiny pod terénem, kdy se bez dalších úprav projeví počítání normál z ponořených bodů, což vede k nepřirozeným artefaktům na okrajích rovných ploch. Tento jev lze odstranit kontrolováním ponoření u bodů a místo jejich hodnot počítat s hodnotou středu.



Obrázek 7: Normály spočítané uvedeným algoritmem. Vlevo ukázka normál na prudkém zlomu, kde je normála více ovlivněna dlouhou svislou stěnou, která je tak osvětlena korektněji než při výpočtu průměru normál trojúhelníků. Zbylé dva obrázky ukazují normály v méně extrémních situacích.

2.3.3 Texturování



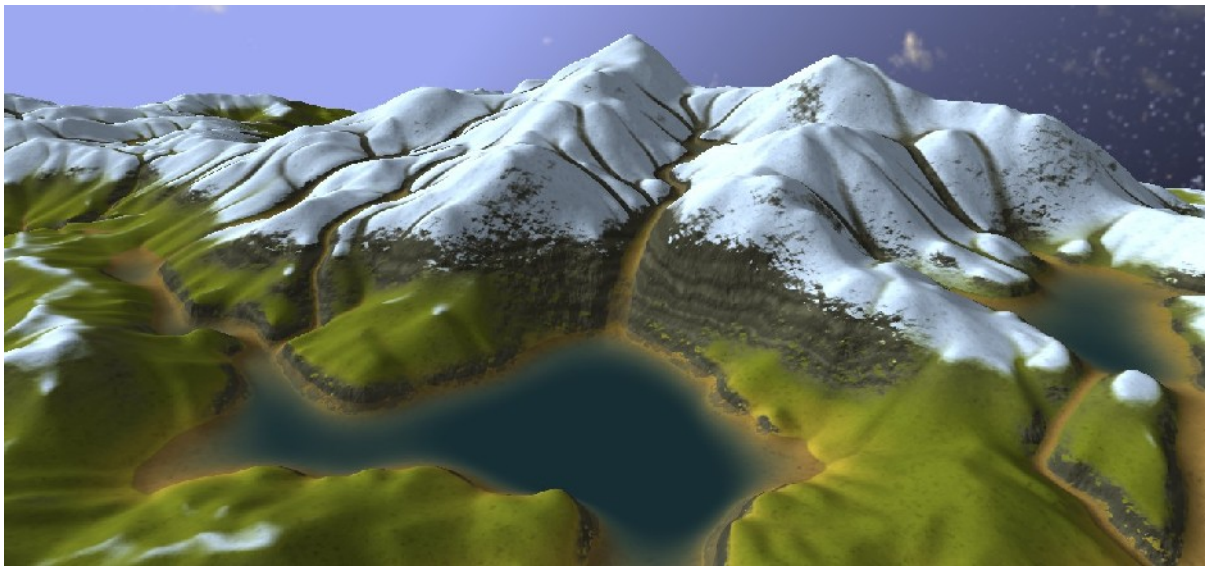
Obrázek 8: Kombinace vln vytvořených geometrií a iluze detailů pomocí texturování.

Texturování vodní hladiny je poměrně jednoduché. Různě pokročilým technikám se věnují například 1. a 2. kapitola GPU Gems [16] a 18. kapitola GPU Gems 2 [17]. I když není problém dnes generovat například Perlinův šum přímo na grafické kartě, jak je popsáno v 5. kapitole GPU Gems a v 26. kapitole GPU Gems 2, nejedná se stále o příliš rychlou záležitost. Je tedy nejdříve potřeba vygenerovat 3D texturu s předem spočítanými normálami, jejíž všechny hrany na sebe dobře navazují. Každá vrstva obsahuje 2D normálovou mapu a třetí rozměr se použije pro animaci. Aby nemusela být textura příliš veliká, použije se ve fragment shaderu několik vzorků, které jsou pak zkombinovány do jedné normály, čímž se omezí opakování a zároveň zajistí zobrazení malých detailů i větších vln. Alternativou může být algoritmus z 12. kapitoly GPU Gems 2, který využívá skládání několika dlaždic do jedné velké textury. Výsledkem je modifikována původní normála vypočtená ve vertex shaderu. Modifikovaná normála slouží k určení odrazu oblohy na vodní hladině a vytvoření iluze menších vlnek než je reálná 3D geometrie.

3 Terén

Zásadní součástí tohoto projektu je terén. Slouží jako podklad pro simulaci tekoucí vody, zároveň je jí sám erodován a tvoří hlavní část zobrazované scény. Kromě eroze jsem vybral následující povětrnostní vlivy a přírodní jevy, které jsou na něm zobrazovány:

- V nejvyšších nadmořských výškách se nachází sníh, který se na prudkých svazích drží méně než na rovinách a navíc je rozpouštěn vodou.
- Pod sněhem se nachází skalnaté svahy. Skály se nacházejí i v nižších polohách, pokud strmost svahu přesáhne určitou hranici.
- V nížinách je tráva, která postupně mění svůj vzhled s nižšími nadmořskými výškami.
- V oblastech s velkou vlhkostí se navíc vyskytuje bahno.
- Jako imitace hloubky vody se k barvě terénu přimíchává také barva vody podle výšky vodního sloupce nad daným bodem a barva mlhy podle dále popsaných pravidel.



Obrázek 9: Ukázka texturování terénu bez zobrazení vodní hladiny a vegetace.

3.1 Reprezentace

Terén je zde reprezentován jednoduchou výškovou mapou. Kromě jednoduchosti zobrazení to má hned několik důvodů, z nichž nejpodstatnější je korespondence každého výškového bodu s jednou buňkou celulárního automatu simulátoru vody. Další důvod je snadná reprezentace dat na GPU v podobě textury s výškovými daty, která zároveň uchovává i výšku vodní hladiny, množství bahna ve vodě a průměrnou vlhkost. Podobně jako u vody se také počítají normály podle rovnice 2.

3.2 Geometrie

Geometrie terénu je prakticky stejná jako geometrie vody popsaná v kapitole 2.3.1, spíše jednodušší. V ideálním případě jsou veškerá zobrazovaná data uložená na grafické kartě, avšak problém může být opět jejich dynamika. Dříve bylo nutno dynamická data nahrávat do GPU po každé úpravě z CPU. To naštěstí již pro většinu aplikací neplatí, a základní statické modely lze přímo na GPU modifikovat pomocí několika proměnných shaderu, nebo přímo výpočty na GPU.

Dynamická výšková mapa spolu se simulací tekoucí vody jsou přímo ideální pro zobrazení čistě pomocí GPU. Na kartu se do statického bufferu nahraje síť vrcholů v 2D mřížce a indexy pro tvorbu trojúhelníků. Stejnou mřížku lze použít pro zobrazení vody i terénu. Třetí souřadnice určující nadmořskou výšku se upraví ve vertex shaderu pomocí čtení z textury.

3.3 Generování a modifikace terénu

Jelikož jsou data uložena na grafické kartě, je vhodné zde terén i generovat a modifikovat. Obě operace lze provést podobně jako simulaci vody pomocí poměrně jednoduchých shaderů. Alternativou generování terénu algoritmicky je samozřejmě načtení výškové mapy ze souboru a její nahrání do paměti GPU.

Pro generování terénu lze využít například klasický Perlinův šum, jehož implementací na GPU se zabývá 26. kapitola GPU Gems 2 [17]. Ten sám o sobě neprodukuje až tak zajímavé terény, ale kombinací několika vzorků s různými parametry lze vytvořit prakticky cokoli. Inspiraci lze nabrat například v 1. kapitole GPU Gems 3 [18], která se zabývá generováním a texturováním komplexního 3D terénu, ovšem principy se dají snadno aplikovat i na klasickou výškovou mapu. Hlavní problém takto generovaného terénu je absence koryt řek a potoků, což však může velmi rychle napravit zde navržený simulátor eroze. Terén s oblastmi rovných nížin a členitých hor lze vytvořit například následujícím jednoduchým algoritmem (2) v GLSL:

```
float n1 = PerlinNoise(pos, 10);
float n2 = PerlinNoise(pos/2.0, 3);
float n3 = PerlinNoise(pos/8.0, 1);

vec3 mul = vec3(uHeight) * vec3(0.1, 0.2, 1.0);
float out = n1*mul.x*pow(n2+1.0, uMountains)
          + n2*mul.y + n3*mul.z;
```

PerlinNoise(vec3 P, int O) je funkce vracející hodnotu Perlinova šumu s *O* oktávami na pozici *P*. Proměnná *uHeight* reprezentuje maximální rozptyl výstupních nadmořských výšek a *uMountains* výraznost členitosti pohoří. Nastavitelných parametrů může být samozřejmě podstatně víc, stejně jako libovolná kombinace dalších technik pro zpestření terénu.

Další modifikace jsou vesměs ještě jednodušší. Jelikož podpora větvení programu na GPU ještě stále není úplně ideální, je vhodné vytvořit shadery zvlášť pro různé operace, nebo je aspoň rozdělit do podobně náročných skupin. Například není vhodné kombinovat shader pro jednoduchou modifikaci nadmořské výšky s přidáváním Perlinova šumu, protože větev se šumem zpomalí i jinak velmi rychlou modifikaci výšky.

3.4 Textury

Ve fragment shaderu terénu využijeme několik různých textur, které je nejdřív potřeba nějak získat. Pořídit kvalitní texturu například z fotografie není úplně jednoduché, data zabírají prostor a navíc budou pokaždé stejná. Proto jsou veškeré textury generovány algoritmicky.

Základní 2D texturu terénu lze opět vytvořit pomocí kombinace několika vzorků Perlinova šumu s různými parametry. Podobně jako u generování terénu může jako dobrá inspirace posloužit 26. kapitola GPU Gems 2 [17]. Dají se ale využít i jiné techniky, jako je například texture bombing z 20. kapitoly GPU Gems [16] a mnoho dalších. Bylo by možné generovat několik druhů textur pro různé povrchy, s použitím níže uvedeného postupu jsou však vizuální výsledky poměrně dobré i s jedinou texturou.

Další důležitou texturou jsou normály. Zde lze s úspěchem využít texturu pro zobrazení vodní hladiny, která nám navíc díky svému třetímu rozměru poskytne dobrou variabilitu a nízké opakování vzorů, pokud pro její mapování na terén využijeme i nadmořskou výšku jako třetí souřadnici.

Pro zobrazení vrstev terénu, zejména na skalách, je možno využít jeden rozměr prakticky jakékoli textury. Pokud chceme dosáhnout nějakého specifického rozložení čar vrstevnic, lze opět s úspěchem využít Perlinův šum nebo jiné podobné techniky pro vygenerování samostatné textury.

3.5 Fragment shader terénu

Výsledný vzhled terénu vytváří fragment shader, který každému bodu přiřadí výslednou barvu. Jelikož je terén dynamický, není možné pouze použít jednu staticky uloženou texturu, respektive výsledek by nebyl příliš realistický. Ve fragment shaderu je tedy nejdříve potřeba určit typ zobrazeného terénu v daném bodě na základě nějakých vstupních parametrů. V našem případě to bude zejména podle nadmořské výšky, sklonu svahů a průměrné vlhkost. Mezi další vstupy by mohlo

patřit třeba rozložení hornin, mapa teploty, osvětlení svahů sluncem, druhy vegetace pro určené oblasti a mnoho dalších.

3.5.1 Výpočet koeficientů

Nejdřív je potřeba stanovit, které typy terénu se jak překrývají. Z námi vybraných typů terénu má nejvyšší prioritu sníh, který překrývá všechny ostatní typy. Dále je to skála, pak následuje bahno, další na řadě je ve vyšších polohách tráva a zbytek zbude na suchou trávu. Na základě zde určeného pořadí je možno začít počítat koeficienty od nejdůležitějších, takže jejich výsledky můžeme využít při výpočtech méně významných koeficientů, jejichž vliv je významnějšími omezen. Je vhodné každý koeficient omezit na interval <0, 1> a zajistit, aby výsledná suma koeficientů byla 1. Pokud tedy hned nejdůležitější koeficient k_1 dosáhne hodnoty 1, ostatní se vůbec neprojeví. V opačném případě je suma méně významných koeficientů rovna $1 - k_1$.

Pro výpočet koeficientů potřebujeme znát normálu zobrazovaného bodu. Základní výpočet proběhne stejně jako u vody podle rovnice 2. Tato hodnota nám pomůže určit prudkost svahu ve zkoumaném místě, a tak i výskyt sněhu, skal a dalších parametrů. Normála však nebude finální, protože podle ní určíme koeficient pro skály, který dále ovlivní míru projevení modifikace normál hodnotami z normálové textury. Na skalnaté části terénu se modifikace projevuje silněji než jinde, čímž skále dodá členitější vzhled, než je popsaný geometrií. Navíc lze tuto upravenou normálu využít i pro rozbití rovných předělů mezi jednotlivými druhy terénu a vytvořit tak členitější vzhled i mimo skály. Nicméně pokud chceme rozbít hranici skal, musíme jejich koeficient přepočítat s novou normálou. Samozřejmostí je použít nové normály i pro osvětlení nebo případný displacemen mapping.

Algoritmus výpočtu jednotlivých koeficientů (3) pak vypadá následovně:

1. Spočítej vliv jednotlivých parametrů na výskyt daného typu terénu, jako je zejména vliv nadmořské výšky, svahu a vlhkosti.
2. Spočítej koeficient pro tento druh terénu tak, že sečteš parametry podporující jeho výskyt a odečteš omezující parametry.
3. Výsledek omez na interval <0, 1>.
4. Omez velikost koeficientu podle dříve vypočítaných významnějších koeficientů.
5. Pokud zbývá více jak jeden typ terénu, vrať se na krok 1 a spočítej další koeficient.
6. Poslední koeficient terénu, tedy ten nejméně významný, je stanoven tak, aby doplnil sumu všech koeficientů na hodnotu 1.

Krok 4 může mít různou podobu pro různé koeficienty. Jelikož například sníh bude lépe zakrývat skály než vegetaci, je dobré koeficient sněhu od koeficientu skal odečíst (a následně zajistit jeho nezápornost), aby zůstal větší prostor pro ostatní typy terénu. Jindy je zase realističtější, když se

velikost právě vypočítaného koeficientu (po omezení na interval <0, 1>) vynásobí hodnotou doplňující sumu významnějších koeficientů do 1.

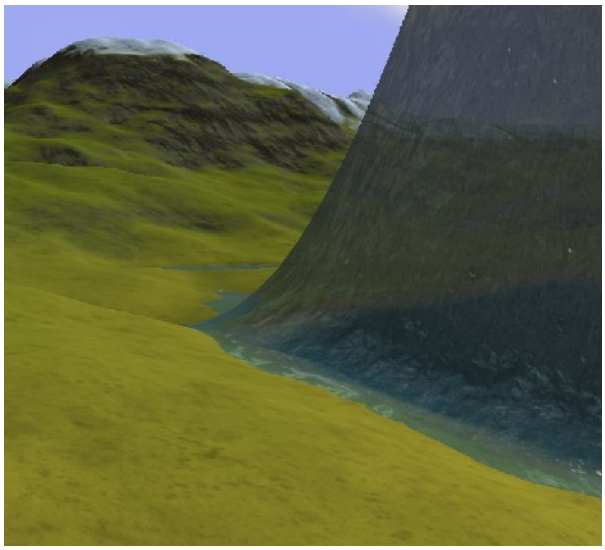
Celý shader obsahující výpočty koeficientů se nachází v příloze A – Ukázky shaderů.

3.5.2 Určení výsledné barvy fragmentu

Výslednou barvu terénu můžeme na základě koeficientů prostě namíchat z barev jednotlivých druhů terénu a pouze tak obarvit univerzální texturu, nebo podle nich můžeme namíchat vzorky několika textur reprezentujících konkrétní typ terénu. Pokud zvolíme dobrou základní texturu, může nám poměrně dobře posloužit i jedna jediná. Ušetříme tak několik čtení z paměti textur a prostor jimi obsazený.

Pro snížení opakování textur je možno opět použít způsob popsaný u texturování vody. Výsledná modifikace normály se spočítá z několika vzorků stejné textury s různým opakováním a s různým významem při míchání. Podobný postup se dá s úspěchem využít i při nanášení základní textury terénu, kterou lze vytvořit tak, aby se dala použít jak pro detaily, tak pro změnu barevnosti větších ploch.

Efekt vrstvení hornin se dá do scény přidat jednoduchým využitím 1D textury mapované podle nadmořské výšky. Aby nebyly linie pouze rovné, je možné opět využít nějaké šumové textury mapované na terén ve 2D. V našem případě jde opět s úspěchem recyklovat texturu normál vody.



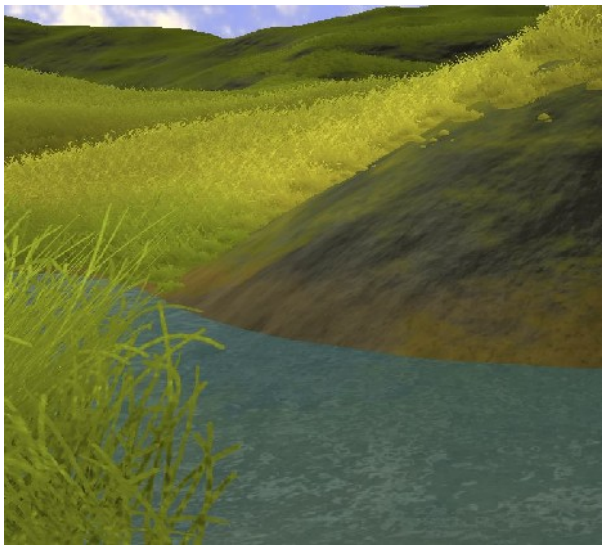
Obrázek 10: Zobrazení hloubky vody pouze pomocí zbarvení terénu nemusí být vždy ideální. Velké vlny jsou příliš průsvitné při pohledu z boku.

Na závěr ještě potřebujeme zohlednit výšku vodního sloupce nad zobrazovaným bodem terénu, protože smícháním barvy vodní hlubiny s barvou terénu je možné imitovat hloubku vody. Metoda to není úplně přesná a má některé nevýhody, je však velmi jednoduchá, rychlá a za normálních okolností i poměrně přesvědčivá. Hlavní nevýhodou je průhlednost velkých vln při pohledu z boku ukázaná na obrázku 10. Dokonalejší alternativou by mohlo být například renderovat obraz terénu i s hloubkou scény do textur a ty pak použít při kreslení vody pro přesnější určení trasy paprsku ve vodě.

4 Další vizuální efekty

Pro podpoření základní vizualizace povětrnostních vlivů a realističnosti celé scény je možno použít velké množství efektů, z nichž některé samy o sobě splňují definici povětrnostních vlivů. Většinou se však jedná pouze o zlepšení dojmu z celé scény, které přímo nesouvisí s tématem diplomové práce, a tak jim není věnováno příliš mnoho pozornosti. Některé efekty byly navíc do projektu přidány v rámci úkolů do různých předmětů. V takových případech je celá dokumentace k příslušnému projektu s více detaily přiložena v elektronické podobě na DVD a zde uvádím jen stručný souhrn. Mnoho inspirace se dá nalézt v sérii knih GPU Gems [16][17][18], které jsou k dispozici i online na vývojářském webu firmy NVIDIA [19], kde je k dispozici také velké množství dalších zdrojů.

4.1 Vegetace



Obrázek 11: Vegetace se řídí podobnými pravidly jako texturování terénu, aby byl přechod co nejplynulejší.

Existuje mnoho metod pro zobrazení vegetace v reálném čase a toto téma samotné by mohlo zabrat celou práci. V tomto projektu slouží zobrazení vegetace hlavně pro podpoření vizualizace povětrnostních vlivů, které jsou jinak zobrazeny texturou terénu. Vegetace také může sloužit ke zviditelnění jinak těžko zachytitelného větru. Základ způsobu renderování trávy je poměrně podobný způsobu popsanému v 7. kapitole GPU Gems.

Dynamická povaha scény vylučuje znát dopředu lokace vhodné pro růst rostlin. Přenášet z tohoto důvodu data na CPU také není příliš vhodné. Jako ideální řešení se tak jeví nechat určení výskytu rostlin na GPU, které pro to disponuje příslušnými daty.

Jedním ze způsobů, jak přimět GPU vykreslit rostlinu, je zaslat jí požadavek na vykreslení jediného bodu. Po vertex shaderu nechceme v takovém případě žádnou práci a vše necháme až na geometry shader. Ten určí typ terénu v požadovaném bodě podobně jako shader texturující terén. Na základě podkladového typu terénu je pak možno generovat konkrétní druh rostliny o určité velikosti. Pohupování ve větru je možno simulovat posunem horních částí rostliny pomocí nějaké funkce nebo dat z textury. Fragment shader již pouze zobrazí trojúhelníky otexturované příslušným typem rostliny.

Bylo by sice možné určit typ rostliny a její velikost již ve vertex shaderu, ale pouze by tak vznikla potřeba předávat data mezi jednotlivými shadery. Význam by to mohlo mít v případě, že by bylo možné příliš malé rostliny vyřadit ze zpracování vertex shaderem, GLSL však obdobu funkce *discard* z fragment shaderů ve vertex shaderech bohužel neobsahuje. Výkon geometry shaderů je navíc dán maximálním možným počtem výstupních primitiv a nezáleží, jestli byly skutečně vytvořeny.

Pro zobrazení dostatečně hustého porostu na celém terénu by bylo potřeba zpracovávat obrovské množství rostlin, které by při minimálním vlivu na vzhled scény výrazně snížily výkon. Z toho důvodu je potřeba rostliny kreslit pouze v okolí kamery, což lze zajistit následujícím postupem:

1. Vygeneruj jeden čtverec vegetace o rozměrech 1 x 1 (reálná velikost může být větší).
2. Nakopíruj tento čtverec do sítě N x N.
3. Tuto síť ulož do statického pole na GPU.
4. Při zobrazení posuň vertex shaderem zobrazovanou síť o celočíselný krok podle pozice kamery.

Tímto způsobem je možno dosáhnout jednoduchého posouvání výřezu zobrazované vegetace po terénu. Vznikají tak ale dva problémy: vegetace se velmi znatelně opakuje a velká část sítě může stále zůstat nezobrazená. První problém lze odstranit velmi snadno tak, že se ve vertex shaderu ke každé pozici bodu přičte hodnota z textury s šumem, k čemuž lze opět recyklovat texturu normál vody. Druhý problém lze při zachování statičnosti dat jedné mřížky do určité míry zredukovat posouváním začátku sítě tak, aby byl na hraně záběru kamery. Protože se kamera může dívat na terén z ptačí perspektivy, je mřížka v základu renderovaná vycentrovaná okolo pozice kamery. Při naklánění do vodorovné polohy lze využít souřadnice X a Y z normalizovaného vektoru směru pohledu k určení posunu mřížky. Hodnoty je třeba pouze vynásobit polovinou rozměru mřížky a přičíst k posunu. V závislosti na šířce záběru kamery se však stále zpracovává zhruba dvojnásobek skutečně zobrazených rostlin. Pokud bychom chtěli dosáhnout lepších výsledků a přitom udržet veškeré paralelizovatelné výpočty na GPU, bylo by pravděpodobně lepší vytvářet geometrii rostlin pomocí architektury CUDA nebo OpenCL.

4.2 Mlha

Obrázek 12: Obloha zamlžená kombinací konstantní a gradientní mlhy, terén zamlžený pomocí funkce hustoty mlhy závislé na nadmořské výšce.

Jedním z povětrnostních vlivů, který spíše dokresluje atmosféru scény je i mlha. Aproximace mlhy může navíc pomoci při vizualizaci dalších jevů, jako je zejména déšť, sníh, prachové bouře a podobně. OpenGL poskytuje ve své statické funkcionalitě jednoduchou možnost přidat do scény konstantně hustou mlhu, která však nemusí pro mnoho aplikací stačit a při využití fragment shaderů ji stejně není možné použít. Navíc se výsledný obraz skládá ze dvou na sobě nezávislých částí, takže je potřeba vyřešit zobrazení mlhy zvlášť na terénu s vodou a vegetací a zvlášť na obloze.

4.2.1 Terén, voda a vegetace

Úroveň zamlžení pixelu je možno aproximovat pomocí integrálu hustoty mlhy na trase paprsku z kamery do cílového bodu. V závislosti na složitosti distribuční funkce mlhy to může být úkol triviální až velmi složitý. OpenGL poskytuje pouze triviální možnost s rovnoměrně rozloženou mlhou, jejíž rovnice je možno nalézt například v OpenGL Programming Guide [20]. Pestřejšího rozložení mlhy lze dosáhnout popsáním složitější funkcí [21], využitím objemové geometrie [22], uložením její hustoty do rastru v textuře [23], zobrazit mlhu pomocí systému otexturovaných částic [18][24] a podobně. Způsobů je mnoho a téma by opět vydalo na samostatnou práci. Při použití textury je potřeba provádět obdobu raytracingu projitím všech texelů na trase paprsku, což při vyšších detailech mlhy může být značně nákladné. Částice se zase spíše hodí pro lokální mlhu a kouř. Jako rozumný kompromis rychlosti a oživení scény se jeví popsat globální mlhu relativně jednoduchou funkcí, pro kterou půjde s rozumnými výpočetními nároky spočítat integrál.

V rozsáhlejším terénu bývá nejvýraznější rovnoměrně rozložená mlha zaručující vzdušnou perspektivu, která může být doplněna mlhou rozvrstvenou podle nadmořské výšky, která může zejména aproximovat mlhu ležící v údolích.

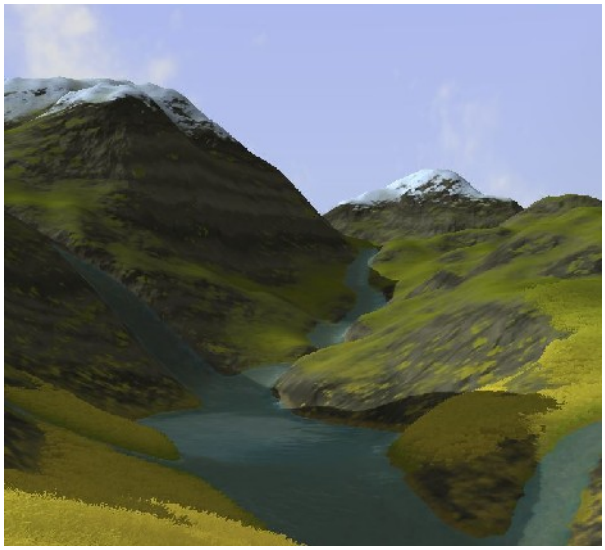
Vypočtená hodnota zamlžení pak slouží při zobrazení terénu, vody i vegetace k míchání původní barvy s barvou mlhy.

4.2.2 Obloha

Mlhu a opar na obloze je potřeba zobrazit jiným způsobem, protože obloha je renderovaná na pozadí pomocí krychlové textury jakoby v nekonečné vzdálenosti. Není tak přímo součástí scény a nedá se určit délka paprsku procházejícího mlhou ani nadmořská výška.

Oblohu je samozřejmě možno zamlžit nějakou konstantní hodnotou, což je triviální úkol. Výsledek však není příliš přesvědčivý, protože obloha bývá více zamlžená u obzoru. To lze simulovat pomocí skalárního součinu normalizovaného vektoru ukazujícího aktuální pozici v krychlové textuře s vektorem ukazujícím kolmo vzhůru. Výsledek může sloužit jako parametr funkce, která takto vzniklý gradient může dále upravovat.

4.3 Stíny



Obrázek 13: Stíny kopců na terénu, vodě a trávě.

Stíny pomáhají dokreslit prostorové rozložení celé scény a výrazné přispívají k její realističnosti. Jsou v zásadě dva způsoby, jak v současnosti ve scéně stíny renderovat v reálném čase: stínová tělesa a stínová textura [25].

Tím starším jsou stínová tělesa (shadow volumes) [20], která vykreslením pomocné geometrie kopírující siluetu tělesa směrem od zdroje světla vytváří masku zastíněných oblastí do stencil bufferu. Dříve bylo potřeba vytvářet masku ve dvou průchodech, jedním ve stencil bufferu přičítat přivrácené plošky stínových těles a druhým průchodem odčítat odvrácené. Podobně bylo potřeba výslednou scénu renderovat dvakrát, jednou jako by byla celá ve stínu a jednou s osvětlením, ale jen na místech určených maskou. Dnes je již možno masku vytvořit jedním průchodem a scénu shadery vykreslit dalším. Mezi hlavní problémy stínových těles patří poměrně náročná konstrukce stínových těles (se kterou dnes mohou výrazně pomoci geometry shadery), problém s obrácením stínů při zastíněné kameře a ostrost stínů, kterou však můžeme někdy považovat i za klad. Navíc není možné touto technikou efektivně renderovat stíny texturovaných průhledných trojúhelníků využívajících alpha test.

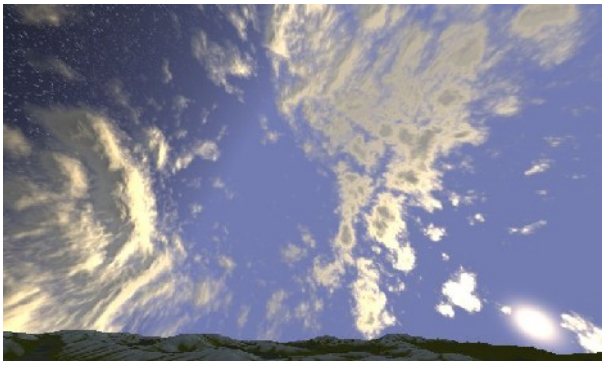
Novější a v základní podobě snadněji implementovatelnou možností jsou stínové textury [26] [27][28]. Vše co má vrhat stíny je potřeba jednoduše vyrenderovat z pohledu zdroje světla do hloubkové textury, není však při tom potřeba provádět většinu nákladných výpočtů finální barvy. Při

renderování výsledné scény je potřeba transformovat renderované souřadnice pomocí stínové matice a pro každý fragment provést porovnání jeho vzdálenosti od hodnoty uložené v textuře. Základní problém shadow mappingu je aliasing textury, která většinou nemá dostatečné rozlišení při velkém přiblížení k zastíněnému objektu. Dále je potřeba řešit omezenou přesnost hloubky framebufferu, stínové textury a výpočtů, které mají za následek objevování stínů i v nežádoucích místech, takže je potřeba přidávat k Z-hodnotám posunutí (offset). Problém může být i ostrost stínů, která je bez použití pokročilých a výpočetně nákladných metod všude konstantní. Přes všechny tyto a další nevýhody je shadow mapping velice rychlý a v základní verzi jednoduchý na implementaci.

Stíny se projevují na terénu, vodě a vegetaci. Základní princip je všude stejný, ovšem v detailech se všechny komponenty liší. Nejjednodušší situace je u terénu, který je pouze v zastíněném místě potřeba vyrenderovat se stejnými parametry jako od slunce plně odvrácené plochy. U vody se navíc přidává nutnost zabránit v zastíněném místě projevům odrazu slunce, který zde nemá co dělat.

Trochu jiný přístup lze zvolit pro vegetaci při použití stínové textury. Každá rostlina je generovaná jedním voláním geometry shaderu, kde je možno zjistit zastínění celé rostliny podle pozice jejího kořenu, a tak značně omezit objem dat čtených z texturovací paměti a odstranit množství operací určujících zastínění jednotlivých bodů. Protože stíny jsou ve scéně rozsáhlé, nepůsobí toto urychlení rušivě.

4.4 Obloha



Obrázek 14: Širokoúhlý záběr procedurálně generované oblohy.

Procedurální generování oblohy a zobrazení osvětlení mraků pomocí normál bylo tématem mého projektu do PGR (Pokročilá počítačová grafika) [3], jehož celá dokumentace je přiložena na DVD.

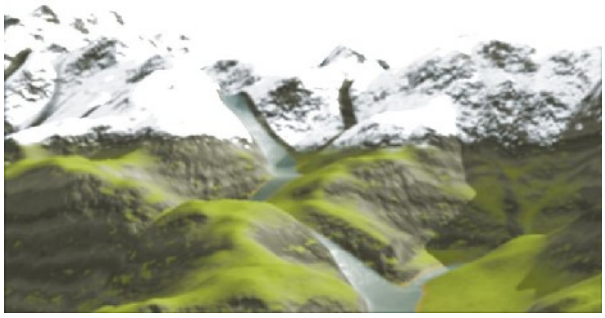
Jelikož program zobrazuje pouze výřez terénu, na který je kamera většinou namířena svrchu, rozhodl jsem se vytvořit mírně nerealistickou aproximaci oblohy vzbuzující dojem, že se terén nachází uprostřed koule z mraků. To samozřejmě není příliš přesné, nicméně zde uvedená metoda se dá použít i pro generování realistické polohy oblohy vůči terénu. Vizualizace je rozdělena do dvou kroků, z nichž první je poměrně výpočetně náročný, zatímco druhý lze provádět bez problémů v reálném čase.

Nejprve je potřeba vygenerovat textury oblohy. Tou první je výšková mapa mraků na kouli okolo terénu uložená do krychlové mapy, kterou lze opět získat například pomocí Perlinova šumu. Z této textury je potřeba vypočítat normály. Výsledkem pak bude RGBA textura s normálou ve složkách RGB a tloušťkou mraku ve složce A.

Pro lepší zobrazení noční oblohy je ještě vhodné vytvořit texturu s hvězdami, k čemuž bohatě poslouží jednoduchý náhodný generátor. Pokud chceme přidat i útvary podobné mlhovinám, lze zase využít Perlinův šum.

Nyní se dostáváme ke druhému kroku, kdy se tyto textury použijí ve fragment shaderu k renderování oblohy, kde slunce bude prosvětlovat mraky podle jejich tloušťky a osvětlovat je z boku díky normálám. Fragment shader také snadno zajistí zobrazení samotného slunce a rozdělí oblohu na denní a noční části.

4.5 Postprocessing



Obrázek 15: Lehké rozmazání a posunutí světlosti a saturace barev.

Postprocessing je převzat z projektů do MUL (Multimédia) [4] a VIN (Výtvarná informatika) [5], jejichž dokumentace je přiložena na DVD.

Do tohoto projektu je postprocessing zařazen zejména proto, že globální změna barev nebo třeba některé konvoluční filtry mohou významně změnit vzhled celé scény a pomoci při vizualizaci některých povětrnostních vlivů, nebo jen dotvořit požadovanou atmosféru virtuální reality.

Postprocessing celé scény je možno provést vyrenderováním nejdříve do obdélníkové textury na GPU o rozměrech zobrazovacího okna. Výsledná scéna je pak vykreslena jako jeden obdélník přes celou obrazovku s touto texturou, který je zpracován podle potřeby dynamicky sestavovaným fragment shaderem.

Podobnou techniku je také možno využít pro imitaci lomu světla na vodní hladině, nebo pro výpočet mlhy na základě paměti hloubky.

5 Implementace

Projekt byl vytvořen v Microsoft Visual Studiu 2008 pro operační systém Windows XP a vyšší. Hlavní program je napsaný v jazyce C++, využívá knihovny OpenGL, GLU, GLEW a ZYLib, která má sama určité níže popsané požadavky. Knihovnu ZYLib jsem vytvořil sám a všechny ostatní použité knihovny jsou pro nekomerční účely zdarma. Jelikož značná část projektu využívá programovatelné grafické karty, je významná část programu napsaná v jazyce GLSL, většinou ve verzi 1.20 nebo 1.30 podle nároků na funkčnost.

Zdrojové texty v jazyce C++ jsou komentované pro generování dokumentace programem Doxygen, pro GLSL byly zvoleny jednodušší komentáře, protože Doxygen neumí pracovat s příslušnými soubory.

5.1 ZYLib

Součásti projektu je i knihovna ZYLib, která je z velké části vytvářena spolu s ním, nicméně již byla použita i pro projekty jiné a může obsahovat i zde nevyužité součásti. Jedná se primárně o jednoduché zapouzdření práce s OpenGL do objektů, ale obsahuje i mnoho dalších užitečných věcí, jako je zejména práce s vektory, maticemi, Perlinovým šumem a různé matematické a pomocné funkce.

Přímo s OpenGL nyní souvisí pouze zapouzdření práce se shadery a s texturami. Některé dříve implementované třídy, jako například podpora materiálů, jsou již ze současné verze vyřazeny, protože zejména díky shaderům ztratily význam.

Matice jsou nyní implementovány pouze v rozměru 4 x 4 s datovým typem *float*, zatímco vektory jsou od 2D do 4D jak *float* tak *int*. Desetinný 4D vektor navíc obsahuje implementaci v SSE2 a je ho možno použít při násobení s maticemi.

Pro práci s obrazovými soubory je využita knihovna GFL SDK [29], která podporuje čtení a zápis velkého množství formátů a je pro nekomerční účely zdarma. Zpracování XML využívá implementaci parseru od Franka Vanden Berghena [30] zveřejněnou pod BSD licencí.

Stejně jako hlavní program je i celá knihovna komentovaná pro generování dokumentace programem Doxygen.

5.2 OpenGL 3.0, rozšíření a GPGPU

Současná implementace vyžaduje OpenGL 3.0, zejména kvůli využití GLSL 1.30. Díky systému rozšíření by se dalo využít i nižší verze OpenGL, ale kód ve verzi 3.0 je čistší a jednodušší, jelikož

mnohá rozšíření jsou zde přímo součástí specifikace a není potřeba se starat o jejich přítomnost a konkrétní název, tedy jestli jsou uvedeny jako ARB, EXT nebo podle výrobce.

Některá užitečná rozšíření se však stále do verze 3.0 nedostala. Zejména se jedná o geometrické shadery (*GL_ARB_geometry_shader4* / *GL_EXT_geometry_shader4*), které využívám pro vizualizaci okraje terénu, vegetace a ladících informací.

Dalším užitečným rozšířením je *GL_NV_primitive_restart*, které je však dostupné pouze na kartách od firmy NVIDIA. Díky němu je možno pomocí speciálního restartovacího indexu vykreslit například několik vějířů trojúhelníků (*GL_TRIANGLE_FAN*) jedním voláním funkce *glDrawElements*.

Díky využití OpenGL 3.0 se program nemusí starat o funkčnost využívanou při GPGPU, jako jsou textury se složkami typu *float* a *int* a všechny s nimi spojené operace. GLSL 1.30, které je dostupné právě až od verze OpenGL 3.0, již plně podporuje standardní operace s celými čísly.

5.2.1 GPGPU

Jelikož je OpenGL grafické API, je potřeba výpočty iniciovat vykreslením nějaké geometrie. Toho lze docílit například následujícím postupem:

1. Vytvoříme datové textury klasickým způsobem s příslušnými parametry a naplníme je inicializačními daty.
2. Pro renderování výsledku do paměti karty vytvoříme *frame buffer object* funkcí *glGenFramebuffers*, aktivujeme ho pomocí *glBindFramebuffer a* nastavíme cílové textury funkcemi *glFramebufferTexture2D* a *glDrawBuffers*.
3. Nastavíme projekční matici pomocí *gluOrtho2D(0, width, 0 height)* na rozměry cílové textury, transformační matici na identitu a použijeme *glViewport(0, 0, widht, height)* pro určení velikosti renderované oblasti.
4. Aktivujeme výpočetní shader a nastavíme jeho vstupní proměnné.
5. Nyní už jen přimějeme OpenGL vykreslit grafická primitiva přes celý viewport. Asi nejvhodnější možností je jeden obdélník s rohy na pozicích (0, 0) a (width, height).

Tento postup umožňuje v GLSL snadno získat celočíselné souřadnice aktuálně počítaného texelu pro funkci *texelFetch* nebo při použití textur s GL_TEXTURE_RECTANGLE_ARB z interpolované pozice vrcholů. Pokud potřebujeme pro čtení interpolovaných dat z textur využít funkci *texture,* která očekává souřadnice v rozsahu <0, 1>, je potřeba pozici buď vydělit rozměry textury, nebo OpenGL nastavit tak, aby se všechny texely vyrenderovaly při vykreslení čtverce s rohy na pozicích (0, 0) a (1, 1). Pro výpočty se většinou lépe hodí celočíselné souřadnice.

5.3 GLSL shadery

Shadery hrají v implementaci obrovskou úlohu. Nejen že program prakticky nevyužívá zastaralou fixní funkcionalitu OpenGL pro zobrazování a všechno renderování se děje pomocí shaderů, ale shadery zastávají i mnoho čistě výpočetních úkonů, jako například simulaci pohybu vody. Práci s nimi je tedy věnována nemalá pozornost.

5.3.1 Načítání .sp souborů

Třída zapouzdřující práci se shadery je implementována v rámci knihovny ZYLib. V terminologii OpenGL jeden program obsahuje různé druhy shaderů a tvoří tak jeden aktivovatelný celek, který pak zpracovává geometrická primitiva. V ZYLib každý objekt třídy *Shader* obsahuje jeden takovýto program. Ten se skládá ze shaderů pro vrcholy, geometrii a fragmenty. Každý shader navíc může být složen z několika zdrojových textů, nicméně OpenGL samo nepodporuje nic ve stylu *#include*, a tak je vhodné něco podobného vytvořit.

Za tímto účelem vznikl jednoduchý formát souboru, který je pro knihovnu snadno čitelný. Základní myšlenka je obsáhnout definici celého programu, tedy všech druhů shaderů, v jednom souboru a zavést jednoduchý mechanizmus ve stylu *#include* jazyka C.

Formát XML není pro uchovávání a manuální psaní zdrojových textů úplně ideální, protože tagy jsou značeny příliš často používanými znaky. Z toho důvodu má soubor základní formát ve tvaru *#PRIKAZ{ obsah příkazu }#*, kde *#PRIKAZ{* plní funkci otevíracího tagu a *}#* uzavírá daný blok, který může sám obsahovat další tagy. Parser rozlišuje velká a malá písmena, text mimo tagy ho nezajímá.

Na nejvyšší úrovni se mohou vyskytovat následující příkazy:

- PROGRAM: Definuje jméno shaderu, momentálně hlavně pro ladící výpisy.
- DEFINE: Může obsahovat další tagy s libovolným jménem, jejichž obsah je následně chápán jako blok textu, které je možno vložit do zdrojového kódu shaderů. Je tak možné například definovat funkci nebo proměnnou, která bude použita ve více shaderech tohoto programu.
- VERTEX, GEOMETRY, FRAGMENT: Obsahují definici daného shaderu pro tento program. Ta je provedena pomocí následujících vnořených tagů, které se mohou opakovat, nikoli však dále vnořovat:
  - CODE: Obsah tagu je přímo kód shaderu.
  - SRC: Obsah tagu je jméno souboru, jehož obsah bude na toto místo vložen.
  - USE: Obsah tagu je jméno zkratky definované v tagu DEFINE, jejíž text bude na toto místo vložen.

- PARAMS: Tento tag je možné použít pouze pro GEOMETRY shader a slouží k definici jeho specifických parametrů. Ty jsou definovány pomocí těchto tagů:
  - IN: Typ vstupních primitiv, jedno z POINTS, LINES, LINES_ADJACENCY, TRIANGLES, TRIANGLES_ADJACENCY
  - OUT: Typ výstupních primitiv, jedno z POINTS, LINE_STRIP, TRIANGLE_STRIP
  - CNT: Maximální počet výstupních vrcholů.

Ukázky tohoto formátu v podobě jednoho výpočetního a jednoho zobrazovacího shaderu jsou obsaženy v příloze A.

5.3.2 GLSL

Některé shadery byly napsány ještě v době, kdy dostupná verze jazyka GLSL byla pouze 1.20. GLSL 1.30 navíc označuje některé dosud používané vestavěné proměnné a funkce za zastaralé a do logu o nich vypisuje varování, ačkoli mnohé z nich jsou i dnes dobře použitelné. Z těchto důvodů jsou v GLSL 1.30 napsané pouze shadery, které jejich funkcionalitu opravdu potřebují.

5.4 Terén a voda

Základem celého projektu je práce s terénem, na něm se pohybující vodou a jejich vzájemná interakce. Program je stavěn na celoplošnou simulaci deště a postupné stékání vody po terénu, která pak tvoří toky a jezera. Voda pak může dále formovat terén erozí a lokální vlhkost má vliv i na vegetaci. Z těchto důvodů musí reprezentace daných jevů vyhovovat všem složkám.

5.4.1 Datové textury

Terén a voda jsou reprezentovány čtyřmi RGBA texturami s 32-bitovými float hodnotami na složku. Pro některé účely lze použít i 16-bitové textury, nicméně pro tento projekt se taková přesnost ukázala jako nedostatečná.

První textura obsahuje v jednotlivých složkách následující informace:

- R: Nadmořská výška terénu.
- G: Výška vodního sloupce
- B: Obsah bahna ve vodě pro simulaci eroze.
- A: Plovoucí průměr vodního sloupce v daném bodě.

Další textura ve svých složkách uchovává aktuální rychlost přelévání vody do 4 sousedních buněk, jak je popsáno v kapitole 2.1 o simulaci vody.

Zbylé dvě textury mají stejné významy jako předchozí dvojice a jsou střídány pro čtení současného stavu a zápis nového.

Kromě těchto základních textur se používá ještě jedna pomocná, která slouží pro uložení normál terénu a zobrazované hladiny vody. Normála se hodí pro pozdější snadnou interpolaci fragment shaderem, vykreslovaná hladina je zase pro lepší vizuální efekt lehce odlišná od prostého součtu výšky terénu s vodním sloupcem. Pro tvorbu této textury se využívá jinak nesouvisející shader počítající nové rychlosti vody, protože se tak ušetří mnoho čtení z paměti textur, a tak se i urychlí běh celého programu.

5.4.2 Zobrazení

Pro zobrazení využívám osy X a Y jako základní 2D pozici v terénu, zatím co osa Z reprezentuje nadmořskou výšku. Data vrcholů pro vykreslení terénu a vody jsou stejná – síť vrcholů reprezentující body v pravidelné 2D mřížce s nulovou hodnotou Z. Pozice vrcholů a indexů pro kreslení primitiv jsou uloženy pomocí *vertex buffer objektů* staticky přímo na grafické kartě, čímž se značně redukuje objem přenášených dat po sběrnici. Zobrazovaná souřadnice Z je vypočtena ve vertex shaderu na základě aktuálně vypočítané datové textury, takže vše může být velmi dynamické.

Na základě datové textury je možno vytvářet i různé úrovně detailů geometrie (LOD) pomocí hrubší mřížky vrcholů. Pokud pro výpočty koeficientů typů terénu použijeme výšková data a normály z textur místo interpolace hodnot z vertex shaderu, zůstává texturování stálé i při změně geometrie. Nastává však problém se správným zobrazením hladiny vody zejména v mělčích a členitějších oblastech a není úplně snadné určit správnou polohu vegetace, aby ležela přímo na zobrazovaném terénu.

5.4.3 Shadery

Na terén a vodu je zaměřeno největší množství shaderů. Dají se rozdělit zhruba do tří skupin: zobrazovací, simulační a manipulační.

- Zobrazovací
  - terrain.sp: Zobrazení otexturovaného terénu. Vertex shader pouze vypočte pozici vrcholu a souřadnice stínové textury. Fragment shader spočítá koeficienty pro různé druhy terénu a určí finální vzhled pro daný bod, tedy zejména barvu a výraznost změny normály. Normála se bere z pomocné textury a dále se modifikuje pomocí několika vzorků textury se šumem. Celý shader je obsažen v příloze A.
  - water.sp: Zobrazení zvlněné vodní hladiny. Vertex shader počítá výslednou polohu vrcholu, normálu pomocí vzorků okolních texelů, souřadnice odrazu oblohy, stínové souřadnice a základy osvětlení. Fragment shader pak zejména kombinuje několik vzorků šumu pro úpravu normály za účelem vytvoření iluze vlnek na hladině.

- terrain_skirt.sp: Zobrazení okrajů terénu. Vertex shader v tomto případě pouze předává svůj vstup do geometry shaderu, který na základě dvou vstupních bodů generuje čtyřstěn. Ten je pak ve fragment shaderu otexturován tak, aby vypadal jako vrstvy skalního podloží terénu.
- terrain_geometry.sp: Slouží pro vykreslení geometrie terénu při renderování stínové textury, takže pouze upravuje pozici vrcholů ve vertex shaderu.

- Simulační
  - speeds.sp: Hlavním úkolem je spočítat nové rychlosti přelévání vody do okolí. Jelikož je k tomu potřeba několik vzorků z okolí, je tento shader zároveň ideální pro výpočet dalších pomocných informací, které by jinak potřebovaly ty samé vzorky a zbytečně by tak zpomalovaly výpočet. Počítá se tu tedy navíc vyhlazení vodní hladiny a normály terénu.
  - transport.sp: Na základě vypočítaných rychlostí provádí přesun vody. Je nutno odečíst od aktuálního bodu veškerou odtékající vodu a přičíst přitékající. Rovněž je potřeba vypočítat změnu rychlostí v důsledku přitékající vody. Tento shader se navíc velmi hodí pro přesun bahna společně s vodou a tedy k simulaci eroze.
- Manipulační
  - terrain_gen.sp: Zajišťuje generování náhodného terénu pomocí kombinace několika vzorků Perlinova šumu s různými parametry.
  - terrain_proc_xxx.sp: Skupina shaderů zajišťujících modifikace terénu. Rozdělením do několika shaderů je dosaženo podstatně lepší rychlosti než využitím podmíněných skoků v jednom shaderu. Kromě změny nadmořské výšky v oblasti kurzoru je možno terén vyhlazovat, přidávat do něj šum, nebo manipulovat s vodou na něm.

5.5 Vegetace

Vegetace opět využívá datovou texturu terénu pro určení nadmořské výšky, ve které má daná rostlina kořeny. Také se podle plovoucího průměru vlhkosti z textury dá určit druh a velikost. Vertex shader lehce posouvá zadané vrcholy v osách X a Y pomocí šumové textury, aby se snížilo opakování vizuálně podobných prvků. Geometry shader počítá zbarvení, velikost a finální pozici rostliny. Pokud splní kritéria pro zobrazení, zejména minimální velikost, tak je generováno několik polygonů, které jsou následně otexturovány fragment shaderem. Normála pro osvětlení se bere pro celou rostlinu na pozici jejího kořenu z pomocné textury stejná jako pro terén, aby se snížil rozdíl mezi terénem s vykreslenou geometrií vegetace a bez ní.

Aby se zbytečně netransformovalo příliš mnoho vrcholů zadávajících pozice rostlin, je nutné je kreslit pouze v blízkosti kamery. To je v programu řešeno pomocí vertex buffer objektu naplněného opakujícím se 2D vzorem náhodně rozmístěných rostlin, který se posouvá s pohybem kamery po daných skocích, takže rostliny jsou v blízkosti kamery renderovány stále na stejných pozicích, zatím co ve skutečnosti se vše skokově posouvá.

Aby nebyl přechod od zobrazení vegetace na holý terén příliš ostrý, snižuje se od určité minimální vzdálenosti práh alfa kanálu, podle kterého jsou jednotlivé fragmenty zahazovány (volání discard ve fragment shaderu).

5.6 Obloha

Implementace generování a zobrazení oblohy je z velké části převzata z projektu do Pokročilé počítačové grafiky [3]. Rozšíření se týká zejména optimalizace a přidání zobrazení mlhy. Přidání celkové mlhy je triviální míchání výsledné barvy oblohy s barvou mlhy v daném poměru, gradient na obzoru je řešen pomocí sklonu vektoru ukazujícího do krychlové textury, který je určen podle skalárního součinu sklonu s vektorem ukazujícím kolmo vzhůru.

5.7 Čtení dat z grafické karty

Přestože se téměř vše odehrává na grafické kartě, je stále potřeba některé výsledky dostat zpět na CPU. Typickým příkladem je zjištění nadmořské výšky cílového bodu kamery. Přenášet vypočtená data celého terénu je pro zjištění několika vzorků naprosto nepřijatelná zátěž, a tak program raději zadá speciálnímu dotazovacímu shaderu příkaz k vyrenderování odpovědi na požadované dotazy na zadané pozice v paměti grafické karty, která je pak pomocí *pixel buffer objektu* přečtena na CPU.

V současné situaci je takto možno snížit objem komunikace o několik řádů a navíc shadery mohou využít vestavěné funkce GPU pro interpolaci textur, sloučit několik vzorků z různých textur do jednoho výsledku a podobně.

Hlavní problém je, že dotazovací shader nesmí obsahovat příliš náročné větve, protože způsob práce s větvením programu může na současném hardware způsobit prodlevu i pokud zrovna vykonáváme pouze krátkou větev shaderu, čímž sníží výkon všech dotazů. Z toho důvodu je vhodné mít pro náročné dotazy speciální shadery.

Další problém je zjišťování maximální a minimální nadmořské výšky v terénu, která je využívána pro výpočet ohraničujícího kvádru scény, který je potřeba při renderování stínové textury. Pomocí minima je také kreslen svislý okraj terénu. Není problém po vygenerování projít jednorázově všechny body a zjisti maximální a minimální hodnoty, ale jelikož je scéna velmi dynamická, můžou

se hodnoty velmi rychle změnit. Skenování je sice možné poměrně snadno na GPU paralelizovat, i tak je to však stále nevítaná zátěž. Z toho důvodu program při renderování každého snímku vzorkuje několik bodů, podle kterých pak případně maximum a minimum upravuje. Nejrychlejší změny se mohou dít v oblasti kurzoru při editování terénu, takže první vzorek je vždy odtud. Dále se bere několik náhodně vybraných vzorků. Tímto způsobem se sice nedá zjistit snížení rozsahu a některé změny se nemusí projevit ihned, ovšem vliv na vizuální kvalitu je ve srovnání s urychlením přijatelný.

5.8 GUI

Program je napsaný pro Microsoft Windows XP a novější pomocí Windows API (Win32). Novější technologie, jako například .NET, zde využity nejsou zejména kvůli jistým problémům s kompatibilitou a rychlostí OpenGL v počátcích tvorby programu.

Program obsahuje klasické menu, klávesové akcelerátory a volitelně zobrazitelný nemodální dialog s parametry. K plnému využití všech funkcí je potřeba využít myši i klávesnice. Téměř veškerý vstup je zpracováván klasickou obslužnou funkcí hlavního okna. Pouze postranní dialog má funkci vlastní a ovládání chůze pomocí kurzorových kláves je řešeno asynchronním čtením klávesnice.

Popis ovládání programu je v podobě manuálu obsaženo jako příloha B.

5.9 Problémy

Současná implementace bohužel trpí i několika problémy, které však nejsou příliš vážné:

- Alpha test u vegetace: Vzhledem k využití mipmap pro texturu vegetace je poměrně obtížné správně stanovit alpha hodnotu pro vyřazení fragmentu ze zpracování. Vzdálenější textury jsou totiž interpolovány i s alpha kanálem a se vzrůstající vzdáleností postupně mění svou charakteristiku až do chvíle, kdy je celý polygon zobrazen jako jediný bod s jedinou alpha hodnotou, takže je možno ho buď celý zahodit nebo celý zobrazit, což bohužel vylučuje plynulý přechod z geometricky zobrazené vegetace na holý terén ve větších vzdálenostech.
- Texturování: Na množství zobrazených detailů textur má poměrně značný vliv mipmapping, takže se může vzhled textur na terénu lehce měnit podle úhlu nebo se vzrůstající vzdáleností od kamery. Lineární i anizotropické filtrování mají bohužel výsledky ještě horší.
- V nížinách ležící mlha se momentálně bohužel projevuje i na zaplaveném terénu, kde přebíjí imitaci hloubky vody. K odstranění tohoto problému by bylo potřeba znát trasu paprsku procházející vodou, což není úplně triviální, a tedy ani rychlé.

6 Výsledky

Jelikož celý projekt stojí zejména na výkonné grafické kartě, mohou se výsledky pro různé stroje zásadně lišit. Ty zde uváděné platí pro následující sestavu:

- Grafická karta: Zotac NVIDIA GeForce 9800GTX+ s 1 GB paměti
- Procesor: Intel Core 2 Quad, Q9650 na frekvenci 3 GHz
- RAM: 4 GB
- Operační systém Windows Vista Business (32-bitový)

6.1 Simulace vody

Prvním důležitým výsledkem bylo obrovské urychlení simulace a zobrazení vody oproti bakalářskému projektu. Zatímco původní verze byla schopna na terénu 1000x1000 bodů dosáhnout pouze něco málo přes 0,6 obrázku za vteřinu s jedním výpočetním krokem, je současná implementace schopna provést na stejném počítači při zachování stejného framerate až asi 500 výpočetních kroků a výsledek jednou zobrazit. Při zobrazení každého výpočetního kroku dosahuje současná implementace asi 30 fps (bez vegetace).

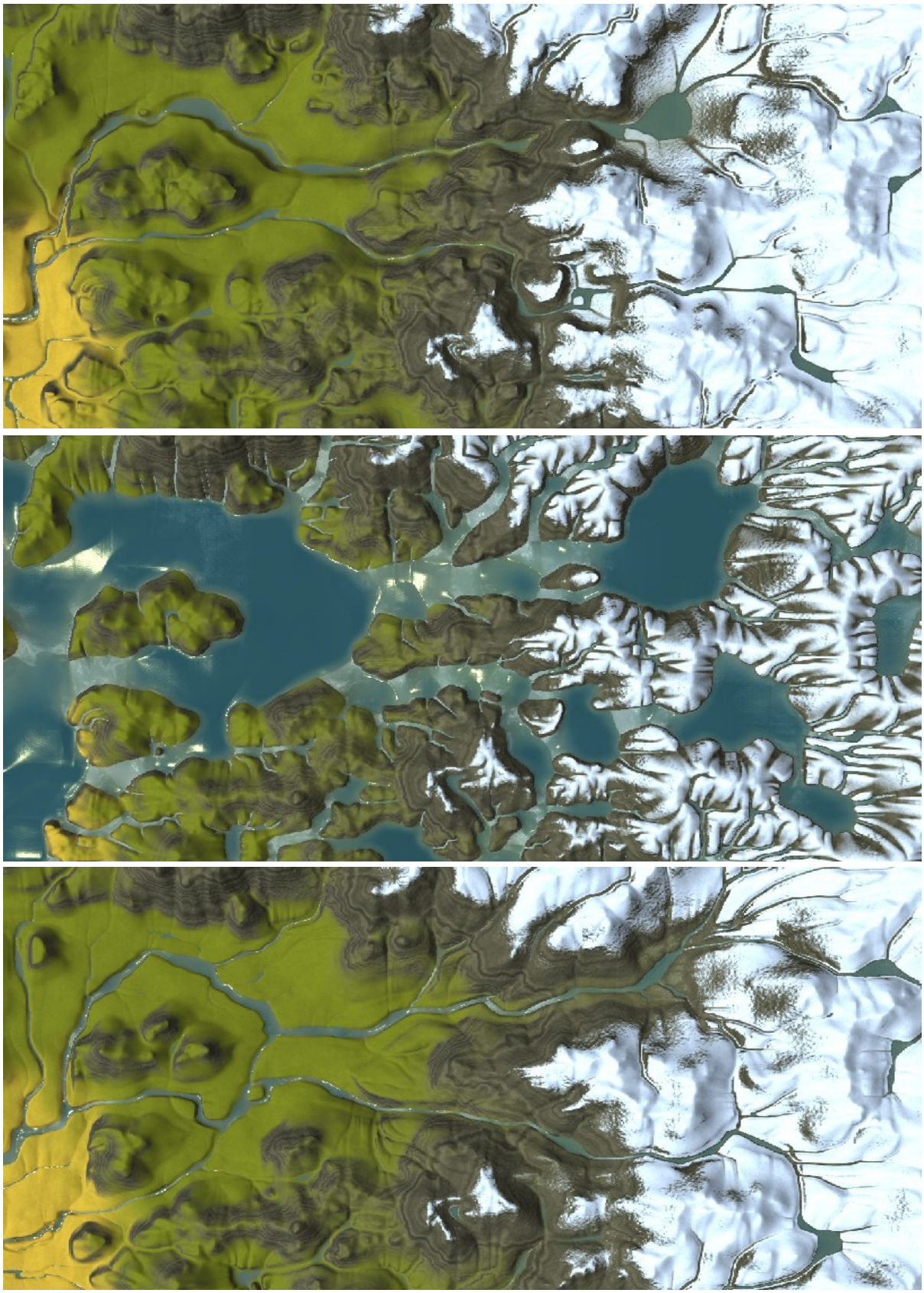
Obrovské urychlení simulace o více než dva řády je dáno několika faktory. Oproti původní implementaci je počítána rychlost jen ve čtyřech směrech místo původních osmi, to má ale význam maximálně polovičního urychlení, spíš méně. Daleko důležitější je přesunutí všech dat trvale na GPU a téměř nulová komunikace mezi CPU a GPU. Data pro zobrazení vrcholů jsou uložena staticky v paměti GPU oproti dřívějšímu posílání aktualizací pro každý obrázek z CPU, kdy bylo pro každý bod potřeba poslat souřadnice terénu, vody a dvě normály, tedy 48 bytů. Pro terén 1000x1000 bodů to dělá asi 45 MB na každý obrázek, při 30 obrázcích za vteřinu je to přes 1,3 GB/s. Odstranění komunikace je umožněno počítáním simulace na GPU, které výsledky renderuje přímo do své paměti, takže můžou být okamžitě opět použity pro úpravu pozice vrcholů ve vertex shaderech.

Dalším faktorem umožňujícím toto obrovské urychlení je extrémně dobrá korespondence řešeného problému s původním účelem GPU a využitím jeho optimalizací. To je zejména obrovská propustnost vnitřní datové sběrnice, která však stále spíše limituje výpočetní výkon shaderových jader, která i tak musí často čekat na data. V době čekání je možno provádět poměrně značné množství výpočtů, pokud k nim postačí menší množství dříve načtených vstupních dat. Navíc texturovací hardware umožňuje levnou interpolaci a vzorky z blízkého okolí se rovněž čtou velmi rychle. K tomu všemu je zde velice dobrá podpora práce s vektory, maticemi a mnoha optimalizovanými vestavěnými funkcemi.

Oproti původní implementaci se tak hlavní limity velikosti terénu zcela mění. Nejvýznamnější z nich je velikost paměti grafické karty, kam se momentálně musí vejít několik velkých textur pro simulaci společně s daty pro zobrazení. Velikost terénu je samozřejmě možno zvětšit rozdělením dat do několika oblastí, které se budou pro výpočty na kartu vždy postupně nahrávat a zase číst. Tím ale dojde k výraznému zpomalení kvůli zahlcení sběrnice. Spíše než simulace je nyní náročná vizualizace, zejména výpočty spojené s transformacemi vrcholů.

Na závěr ještě uvedu několik důležitých poznatků z implementace:

- Datové textury mohou být uloženy v plovoucí řádové čárce buď na 16 nebo na 32 bitech. 16 bitů poskytuje téměř dvojnásobný výkon výpočetním shaderům, což je dáno menším objemem přenášených dat, ale na rychlost zobrazovacích shaderů nemá vliv téměř žádný. Bohužel 16 bitů se ukázalo jako nedostatečná přesnost, kdy i pouhým okem bylo vidět například poskakování výšky hladiny po příliš viditelných krocích.
- Kromě klasického příkazu pro čtení interpolovaných hodnot z textury *texture(samplerND s, vecN p)* existuje v GLSL 1.30 také příkaz *texelFetch(samplerND s, ivecN p, int lod)*, který slouží ke čtení neinterpolované hodnoty ze souřadnic zadaných celým číslem. Oproti původnímu předpokladu, že čtení touto funkcí bude jednodušší a rychlejší, se ukázalo, že její použití zpomalí výpočetní shadery asi na poloviční výkon.
- Při čtení z datových textur je potřeba dávat dobrý pozor, jak jsou filtrovány. Pro výpočetní shadery se lépe hodí brát neinterpolovanou hodnotu pomocí *GL_NEAREST*, kdežto při zobrazování se s výhodou využije lineární interpolace *GL_LINEAR*. Při použití *GL_LINEAR* je ovšem pro správnou funkci potřeba texturovací souřadnice o půl texelu posunout, zatímco u *GL_NEAREST* podobný posun způsobí nežádoucí artefakty.



Obrázek 16: Ukázka simulace vody a eroze. Je zde zachycen původní terén, výrazná povodeň a změna koryt po povodni.

6.2 Texturování terénu a vody

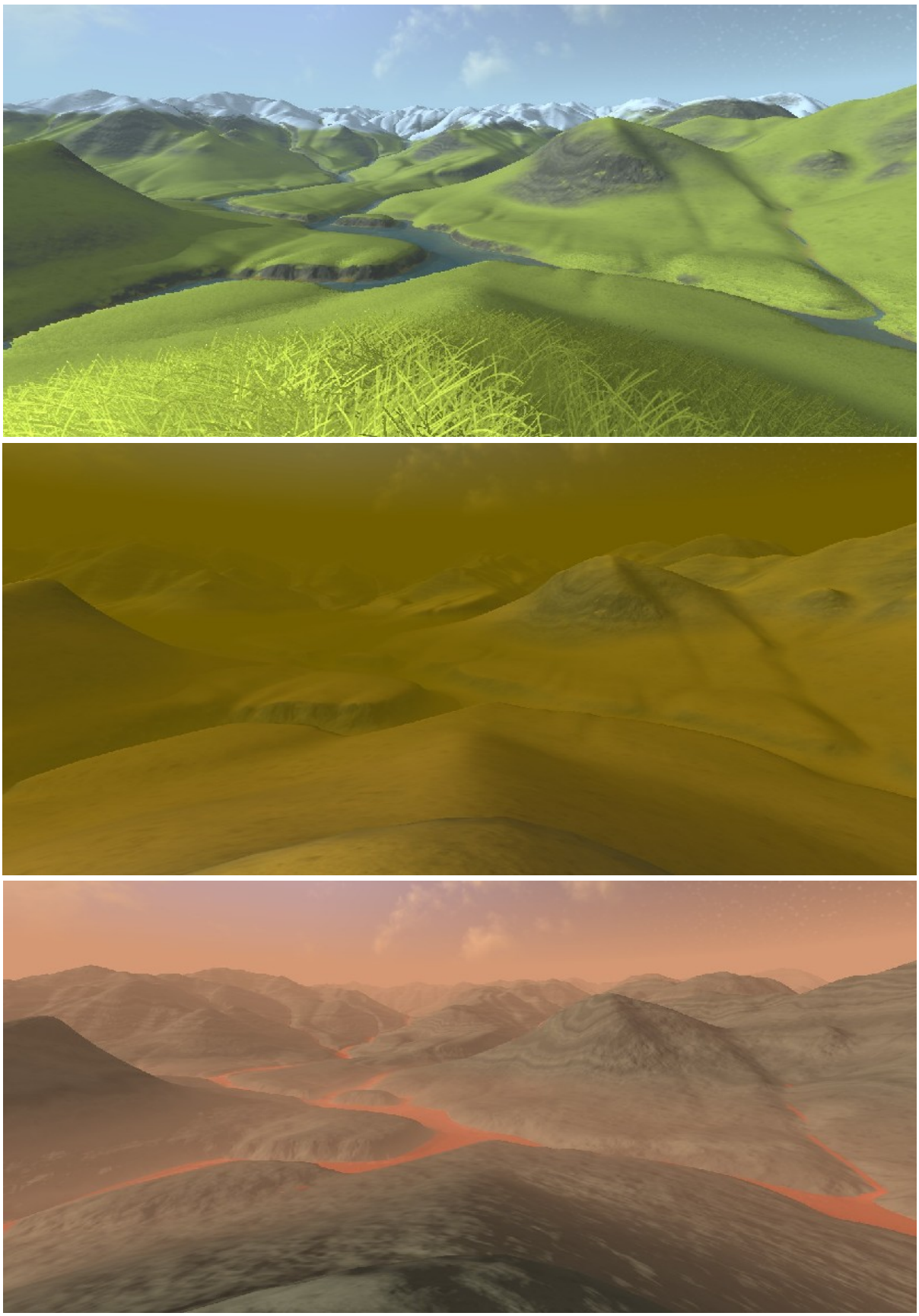
Texturování terénu je nyní řešeno komplexním fragment shaderem na rozdíl od původní fixní funkcionality v bakalářské práci. Je využito několik vzorků šumové textury pro úpravu normál fragmentu, jejichž základ je čten z pomocné textury renderované společně s rychlostmi vody, aby se snížilo celkové množství čtených vzorků textur.

Na základě nadmořské výšky, normál a lokální vlhkosti je pak určen typ terénu výpočtem několika koeficientů s různými prioritami. Podle těchto koeficientů je pak namíchána výsledná barva, určí se, nakolik se budou projevovat určité znaky z textur a změny normály při osvětlení.

Jednou v velkých výhodou tohoto přístupu kombinovaného s normálami v pomocné textuře je stálost zobrazení i při redukci počtu trojúhelníků, protože veškerá podkladová data se berou z textur vypočítaných při simulaci. Je tak možno poměrně snadno pracovat s různým LOD terénu, což má nemalý vliv na rychlost, protože nejvýznamnější zpomalení v celé současné implementaci působí právě transformace vrcholů.

Výsledkem je velmi pestrý terén s vysokou úrovní detailů, který navíc může být značně dynamický bez dalšího vlivu na výkon a ušetří se paměť jinak potřebná na velké textury terénu. Implementaci je samozřejmě možno rozšířit o další druhy terénu, více textur a jiných vstupních parametrů.

V texturování vody oproti bakalářské práci nedošlo k výraznějšímu posunu. Stále je využíváno několik vzorků šumové textury pro úpravu normál fragmentu při zobrazení odlesků slunce a oblohy, pouze bylo provedeno několik drobných optimalizací.

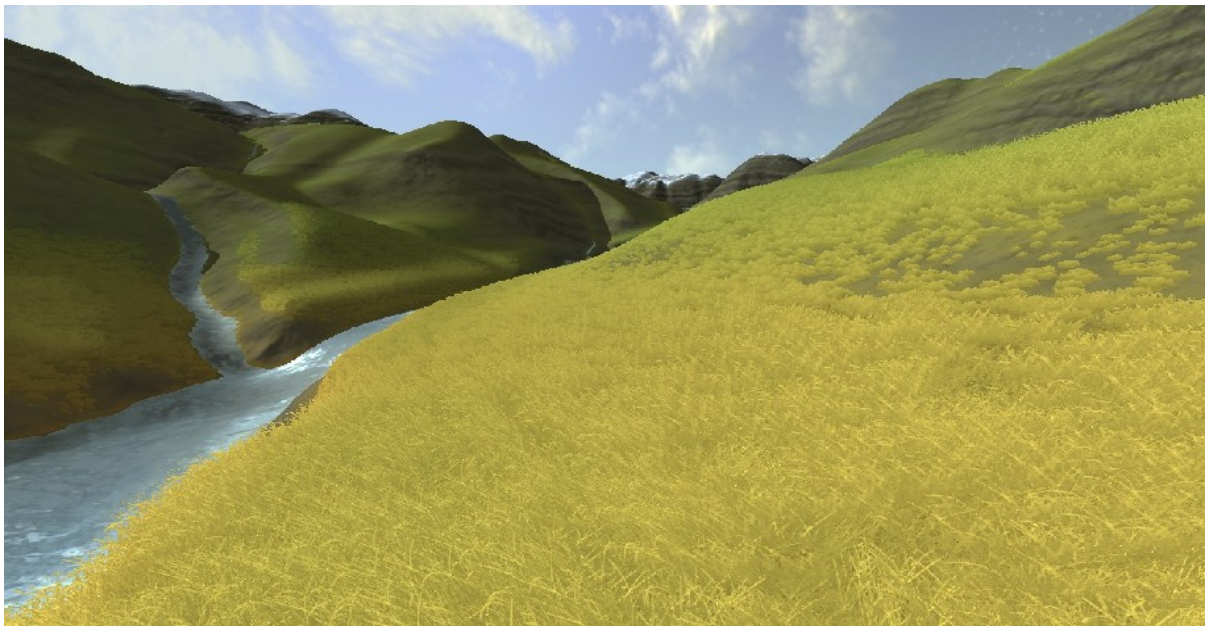


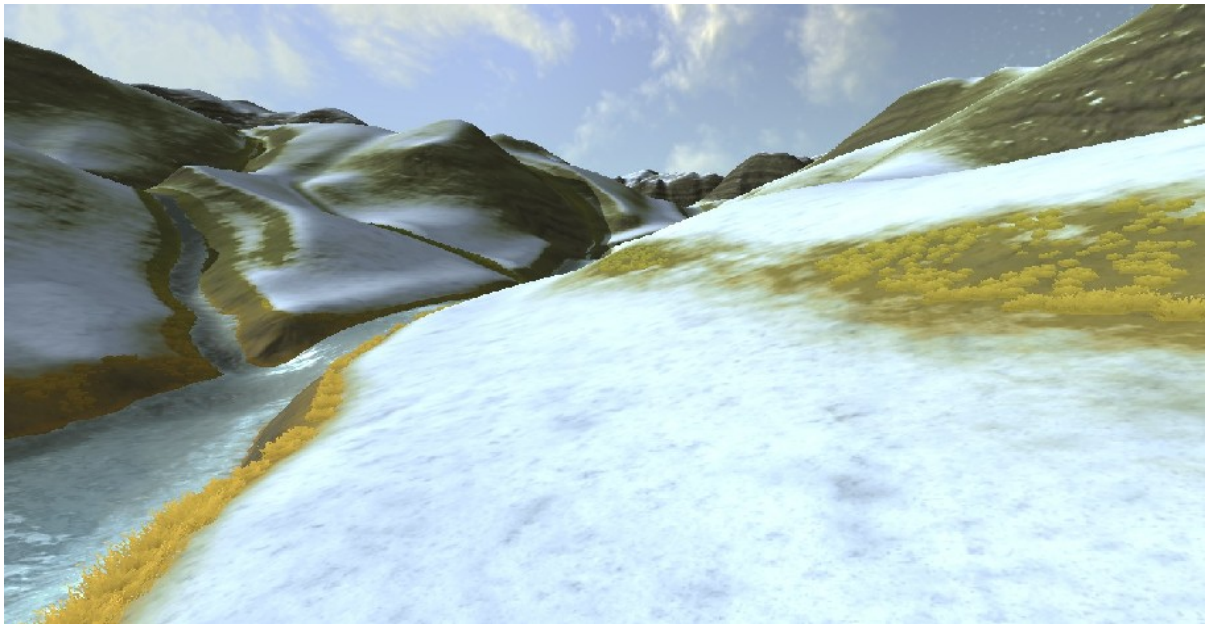
Obrázek 17: Ukázky texturování terénu, které je možno modifikovat bez jakéhokoli zpomalení v reálném čase pouze změnou několika parametrů.

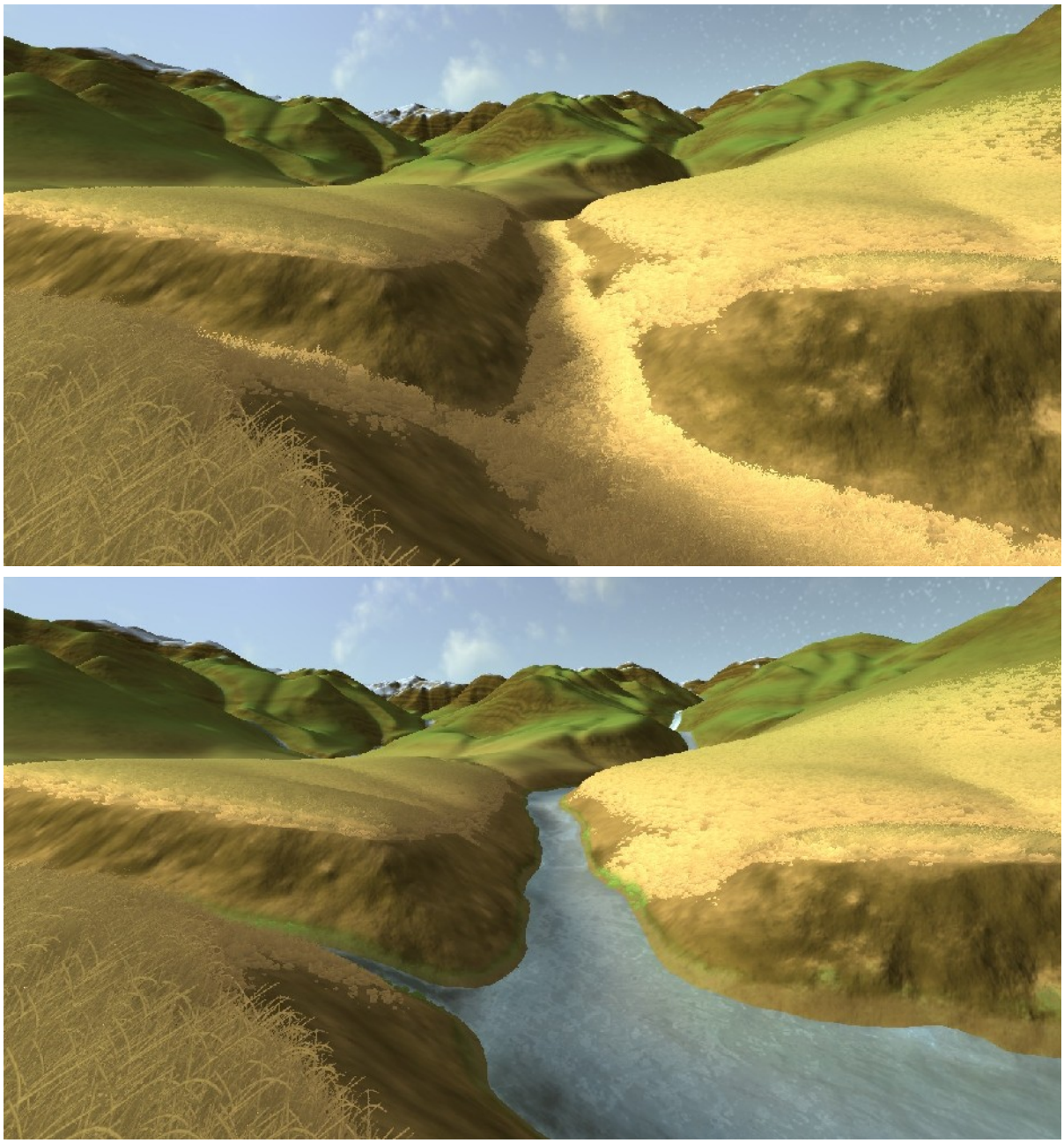
6.3 Vegetace

Momentálně je implementována pouze jednoduchá travní vegetace s jedinou texturou. Program je možno velice snadno rozšířit o více druhů podobných travin, o něco složitější by bylo přidání druhů s jinou geometrií a nejsložitější pak vizualizace keřů a stromů. Vizualizace vegetace však nebyla tématem této práce, a tak je zde pouze demonstrováno několik základních principů, jak určit výskyt rostlin pouze podle výškové mapy terénu a vlhkosti v daném bodě, což je vizualizováno pomocí změny velikosti a zbarvení rostliny.

Navíc byla implementována jednoduchá simulace pohupování ve větru pomocí šumové textury, což opět přispívá k lepší uvěřitelnosti scény.





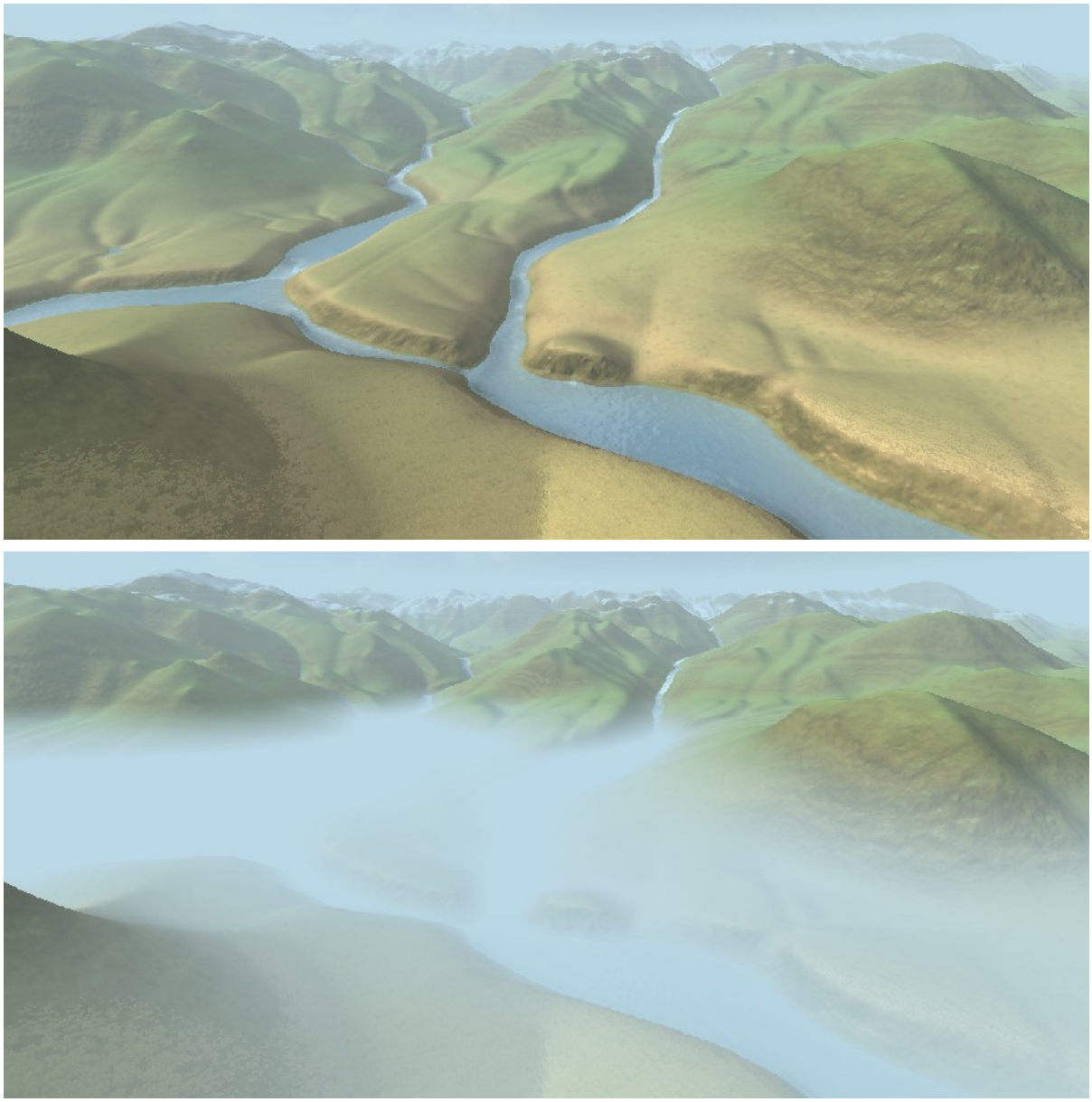


Obrázek 18: Série obrázků ukazuje, jak tráva neroste v oblastech skal, sněhu a pod vodou. Dále je zde vidět sušší tráva v nížinách, která je však v blízkosti vody zelená.

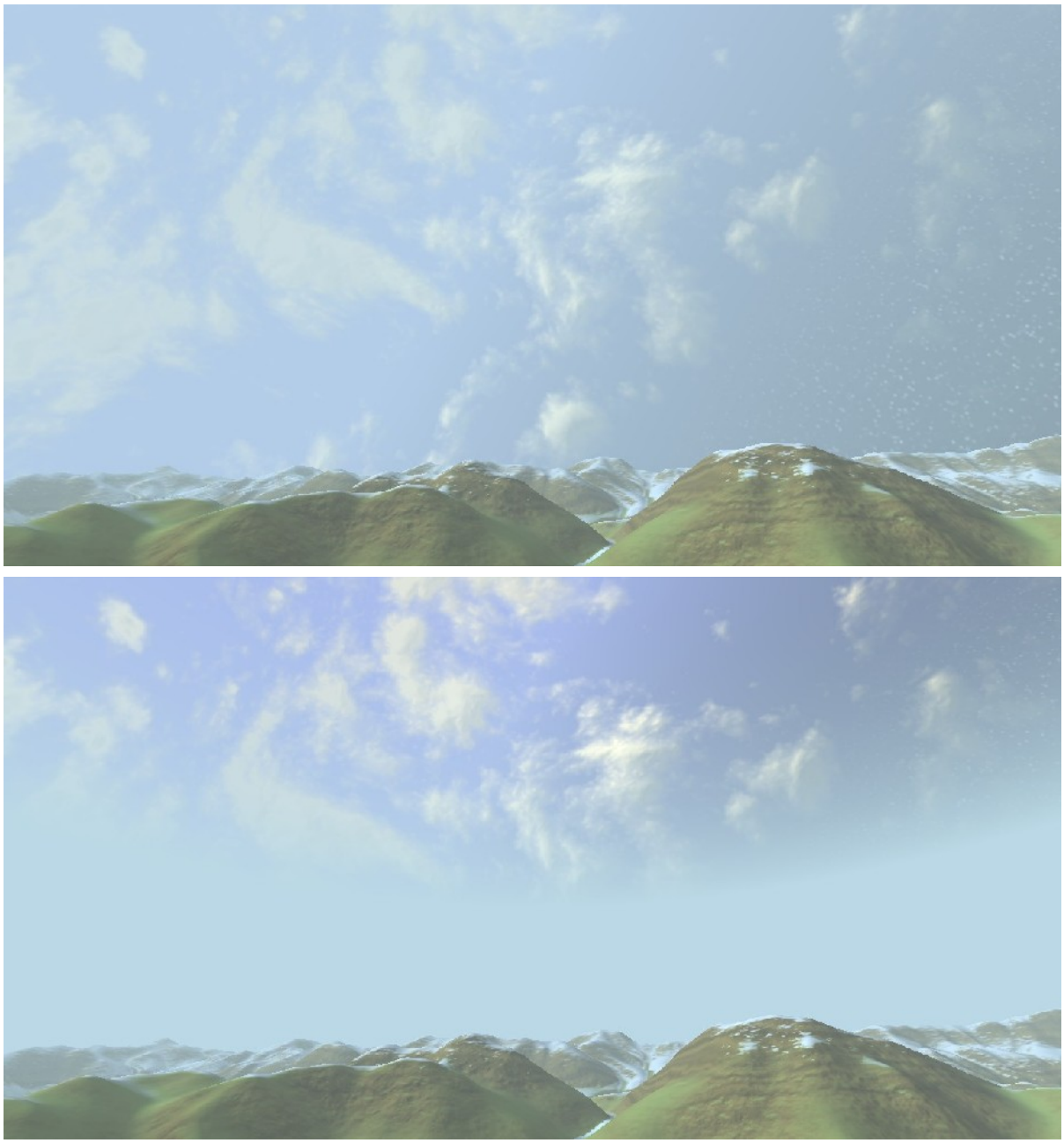
6.4 Mlha

Součástí vizualizace terénu, vody, vegetace i oblohy je zobrazení mlhy. Oproti klasické jednoduché mlze z fixní funkcionality OpenGL je pro každý bod terénu, vody a vegetace spočten integrál z rozložení hustoty mlhy na dráze paprsku. Pro demonstrování tohoto přístupu je využita jednoduchá funkce rozložení mlhy na základě nadmořské výšky, kdy výše je řídká mlha s konstantní hustotou, ve spodní části konstantně hustá mlha a mezi nimi je plynulý přechod.

Ačkoli je implementace v shaderu provedena s pomocí několika podmíněných příkazů kvůli různým variantám průchodu paprsku mlhou, je vliv na rychlost zobrazení scény pod deset procent.



Obrázek 19: Nahoře klasická mlha pouze na základě vzdálenosti, dole mlha založená na funkci hustoty.



Obrázek 20: Nahoře rovnoměrné zamlžení oblohy, dole obloha s gradientem mlhy.

Současná implementace má dva hlavní nedostatky:

1. Mlha nijak neinteraguje se stíny, pouze je překrývá. V ideálním případě by byly stíny vidět i na samotné mlze, to však vyžaduje daleko složitější postup.
2. Mlha v nížinách se projevuje i pod hladinou vody, protože je poměrně nákladné zjistit jaká část dráhy paprsku prochází vodou. Pokud nechceme procházet výšku hladiny všech bodů terénu v dráze paprsku, je možné podvodní dráhu odhadnout z výšky hladiny v cílovém bodě. Vliv tohoto jevu je však poměrně malý a momentálně nestojí za další zpomalení zobrazení.

7 Další práce

Téma povětrnostních vlivů je velice obsáhlé a i přes poměrně široký záběr projektu se jim nedá věnovat všem. Existuje mnoho prací zabývajících se pouze určitými vlivy nebo jejich specializovanými případy. Tato práce sama staví na mé vlastní bakalářské práci [2] a několika menších pracích do různých předmětů [3][4][5]. Na projektu tohoto typu je možné pracovat prakticky neomezeně dlouho a stále bude co vylepšovat. Zde tedy uvádím jen několik námětů:

- Mlha: Odstranění problému s projevem mlhy na terénu pod vodou.
- Zobrazení vody: Renderování podkladu do textury a následné imitovaní lomu světla a vlnění podkladu. Navíc by se díky paměti hloubky dal lépe zobrazit průchod paprsku masou vody a její vliv na barvu texelu.
- Víc textur pro terén: Vylepšení vizuální kvality terénu je možno dosáhnout více texturami pro různé typy terénu uloženými v *texture array* [31], které se pak budou míchat místo současných barev. Určitou nevýhodou tohoto přístupu oproti současnému řešení je nutnost načíst několik dalších vzorků textur.
- Parametry shaderu terénu: Další zvýšení pestrosti scény lze dosáhnout také uložením texturovacích parametrů shaderu terénu do dalších datových textur, které pak po něm budou s nižším rozlišením roztaženy, takže například minimální svah pro výskyt skal nebude všude stejný. Toto řešení má hlavně význam, pokud nebude zavedena 3D reprezentace terénu.
- Simulace vody a eroze: Současná implementace jistě nabízí mnoho prostoru pro lepší přiblížení reálnému pohybu vody terénem. Po dosažení maximální přesnosti s využitím 2D mřížky je možno přejít na 3D reprezentaci [7], případně na simulaci částic [8].
- Kapky: Zlepšení kvality vizualizace deště je možno dosáhnout také vizualizací kapek na vodní hladině, na terénu a při letu vzduchem [32]. Podobně lze vizualizovat také sníh nebo jiné poletující malé předměty.
- LOD: Současná implementace prakticky nepodporuje LOD, pouze demonstruje stálost texturování při změně LOD celého terénu. Důvodem jsou problémy s prolínáním zobrazovaných vrstev. Je potřeba uspokojivě vyřešit překrývání terénu vrstvou vody a hlavně zobrazovat vegetaci ve správné výšce na hrubém LOD.
- Vegetace: Více druhů trav, například pomocí *texture array* [31], další byliny, keře a stromy. Také je možné přidat mapy s výskytem různých druhů, jejichž šíření se dá rovněž simulovat [9], podobně jako šíření požárů.

- Skybox: Pro realistické zobrazení by bylo nutné generovat mraky lépe aproximující kouli nad povrchem země. Také je možno generovat několik vrstev mraků s různými parametry a animovat ji. Generátor je navíc možno přepsat na GPU a tvorbu textur tak výrazně urychlit.
- Simulace mraků, pohybu mlhy, kouře: Podobně jako je zde simulována voda ve 2D mřížce, může být simulován i plyn ve 3D, jako například v demu od firmy NVIDIA [19]. Náročnost 3D mřížky je však již dost značná a pravděpodobně by se pro tyto účely spíše hodilo simulovat částice, což lze rovněž akcelerovat grafickou kartou [8].
- Zobrazení rozsáhlejších scén: Rozsáhlost scén je při současném přístupu limitována převážně velikostí paměti grafické karty provádějící simulaci. Zvětšení scény je možno dosáhnout několika způsoby:
  1. Předpočítat větší oblast a následně zobrazovat statická data.
  2. Stránkovat paměť GPU. Tím ovšem může dojít k přetížení sběrnice a zpomalení celého systému.
  3. Zavést LOD pro simulaci, která by se ve vzdálenějších oblastech prováděla na hrubší mřížce. Mezi hlavní problémy patří navazování mřížek a konzistence výsledků simulací s různými parametry.

8 Závěr

Cílem tohoto projektu bylo vytvořit co nejkomplexnější scénu vizualizující povětrnostní vlivy v terénu v reálném čase. Základní analýzou a výběrem vhodných vlivů jsem se zabýval ve svém semestrálním projektu [1], jehož cílem bylo také najít společné rysy vybraných vlivů, pro které pak mohou být vytvořeny sdílené zdroje. Jakožto ideální základ pro generování mnoha druhů přírodně vypadajících textur se dnes jeví Perlinův šum [17]. S ním souvisí také normálová textura vody, kterou je možno s úspěchem recyklovat i pro mnohé zdánlivě nesouvisející účely od texturování terénu po rozmisťování a animaci rostlin. Simulaci pohybu vody z mé bakalářské práce [2] je zase možné poměrně snadno rozšířit o erozivní působení na podklad.

Za účelem dosažení co nejlepších výsledků byly využity nejnovější technologie, jako je zejména OpenGL 3.0, GLSL 1.30, geometrické shadery a techniky GPGPU. Důležitým výsledkem je obrovské urychlení simulace vody oproti původní implementaci na CPU, které díky omezení komunikace z CPU na GPU a velmi dobrému mapování problému na hardware GPU dosahuje i přes větší objem prováděné práce zrychlení více než o dva řády.

S tekoucí vodou úzce souvisí i eroze, která byla oproti zadání nejen diskutována jako možné rozšíření, ale stala se rovnou součástí celého projektu a jedním z vizualizovaných povětrnostních vlivů. Bylo ukázáno, že i s jednoduchou imitací eroze lze dosáhnout zajímavých výsledků a generování uvěřitelně tvarovaných terénů. Pokud by bylo potřeba simulovat reálné záplavy a erozi, bylo by nutné program ještě mírně upravit, nicméně jednalo by se převážně o napsání komplexnějších výpočetních shaderů, které by sice nemusely být nejvhodnější pro výpočty v reálném čase, stále však mohou zpřístupnit obrovský výpočetní výkon GPU třeba i pro vědecké účely. Pro komplikovanější výpočetní úkoly by ovšem mohlo být vhodnější využít spíše architekturu CUDA nebo OpenCL.

Několik dalších povětrnostních vlivů je vizualizováno pomocí fragment shaderu terénu s parametry nastavitelnými v reálném čase. Jedná se zejména o zasněžení, skalnatost, výskyt dvou druhů trav a bahna, nicméně na základě zde popsaných principů je samozřejmě možné vytvořit i daleko komplexnější scenérie. Koeficienty jednotlivých typů terénu značně závisí na lokálním sklonu svahu, který je určen z normál. Ty jsou na pravidelné 2D mřížce počítány velice jednoduchým a rychlým způsobem, popsaným v kapitole 2.3.2, pomocí dvou odečtení a jedné normalizace vektoru, která je na GPU velmi rychlá. Výsledky navíc dobře vystihují tvar výškové mapy.

Pro dotvoření atmosféry a realistického vzhledu scény pak slouží zobrazení travní vegetace, mlhy, stínů a oblohy. Podstatná je zejména geometrie trávy, jejíž výskyt je řízen stejnými principy jako texturování terénu a je generována přímo na GPU pomocí geometry shaderů. Hustota mlhy je oproti klasickému jednoduchému přístupu s konstantní hustotou popsaná funkcí, a dovoluje tak zobrazit pestřejší efekty při zachování poměrně malé výpočetní náročnosti.

Výsledkem práce je simulátor schopný zvládat netypicky rozsáhlou dynamickou scénu, jehož potenciál dalšího rozvoje je obrovský. V budoucnu může být snadno rozvíjen jak směrem k seriózní vědecké aplikaci, tak jako součást herního engine nebo editoru terénu.

Literatura

[1] VLČEK, A. *Real-time vizualizace povětrnostních vlivů v terénu.* Brno, 2009. FIT VUT v Brně. Semestrání projekt, vedoucí Ing. Michal Seeman.

[2] VLČEK, A. *Vizualizace tekoucí vody v krajině.* Brno, 2007. FIT VUT v Brně. Bakalářská práce, vedoucí Ing. Michal Seeman.

[3] VLČEK, A. *Procedurálně generovaný dynamický SkyBox zobrazovaný shadery.* Brno, 2007. FIT VUT v Brně. Projekt do předmětu Počítačová grafika.

[4] VLČEK, A. *Post-processing OpenGL scény.* Brno, 2008. FIT VUT v Brně. Projekt do předmětu Multimédia.

[5] VLČEK, A. *Generování obrazů pomocí náhodných generátorů a post-processingu mezivýsledku.* Brno, 2008. FIT VUT v Brně. Projekt do předmětu Výtvarná informatika.

[6] PERINGER, P. *SNT - Celulární automaty* [online]. Brno, 2009. FIT VUT v Brně. Materiály k předmětu Simulační nástroje a techniky. [cit. 24-5-2009]. Dostupné z WWW: <https://www.fit.vutbr.cz/study/courses/SNT/public/Prednasky/SNT-cellular.pdf>

[7] BENEŠ, B. - TĚŠÍNSKÝ, V. - HORNYS, J. *Hydraulic Erosion* [online]. Computer Animation and Virtual Worlds 17(2), pp: 99-108, 2006. [cit. 24-5-2009]. Dostupné z WWW: <http://www2.tech.purdue.edu/cgt/Facstaff/bbenes/private/papers/Benes06JVVW.zip>

[8] KRIŠTOF, P. - BENEŠ, B. - KŘIVÁNEK, J. *Hydraulic Erosion Using Smoothed Particle Hydrodynamics in Computer Graphics Forum* [online]. Eurographics 2009, vol. 28 No.2, 2009. [cit. 24-5-2009]. Dostupné z WWW: <http://www2.tech.purdue.edu/cgt/Facstaff/bbenes/private/papers/Kristof09EG.zip>

[9] BENEŠ, B. *Osobní stránky na Purdue University* [online]. 2009. [cit. 24-5-2009]. Dostupné z WWW: <http://www2.tech.purdue.edu/cgt/Facstaff/bbenes/private/benes.htm>

[10] Nvidia. *Cuda Zone* [online]. 2009. [cit. 24-5-2009]. Dostupné z WWW: <http://www.nvidia.com/object/cuda_home.html>

[11] SEGAL, M. - Akeley, K. *The OpenGL Graphics System: A specification, Version 3.0* [online]. August 11. 2008. [cit. 24-5-2009]. Dostupné z WWW: <http://www.opengl.org/documentation/specs/>

[12] Microsoft. *DirectX MSDN* [online]. 2009. [cit. 24-5-2009]. Dostupné z WWW: <http://msdn.microsoft.com/en-us/directx/default.aspx>

[13] Khronos Group. *OpenCL* [online]. 2009. [cit. 24-5-2009]. Dostupné z WWW: <http://www.khronos.org/opencl/>

[14] KESSENICH, J. *The OpenGL Shading Language, Language Version 1.30, Document revision 08* [online]. 2008. [cit. 24-5-2009]. Dostupné z WWW: <http://www.opengl.org/documentation/specs/>

[15] Microsoft. *HLSL MSDN* [online]. 2009. [cit. 24-5-2009]. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/bb509561(VS.85).aspx>

[16] FERNANDO, R. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics* [online]. Addison-Wesley Professional, 2004. ISBN 978-0321228321. [cit. 24-5-2009]. Dostupné z WWW: <http://developer.nvidia.com/object/gpu_gems_home.html>

[17] PHARR, M. - FERNANDO, R. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation* [online]. Addison-Wesley Professional, 2005. ISBN 978-0321335593. [cit. 24-5-2009]. Dostupné z WWW: <http://developer.nvidia.com/object/gpu_gems_2_home.html>

[18] NGUYEN, H. *GPU Gems 3* [online]. Addison-Wesley Professional, 2007. ISBN 978-0321515261. [cit. 24-5-2009]. Dostupné z WWW: <http://developer.nvidia.com/object/gpu-gems-3.html>

[19] Nvidia. *Developer Zone* [online]. 2009. [cit. 24-5-2009]. Dostupné z WWW: <http://developer.nvidia.com/page/home.html>

[20] WOO, M. - NEIDER, J. - DAVIS, T. *OpenGL Programming Guide.* Addison-Wesley Developer Press, 1997. ISBN 0-201-46138-2
[21] *Real-Time Fog using Post-processing in OpenGL*. Anonimní výzkumná zpráva. [cit. 24-5-2009]. Dostupné z WWW: <http://www.cs.gmu.edu/~jchen/cs662/fog.pdf>
[22] Nvidia. *Fog Polygon Volumes.* Nvidia SDK White Paper, 2004. [cit. 24-5-2009]. Dostupné z WWW: <http://developer.download.nvidia.com/SDK/9.5/Samples/DEMOS/Direct3D9/src/FogPolygonVolumes3/docs/FogPolygonVolumes3.pdf>
[23] TARIQ, S. - LLAMAS, I. *Real-Time Volumetric Smoke using D3D10* [online]. 2007. [cit. 24-5-2009]. Dostupné z WWW: <http://developer.download.nvidia.com/presentations/2007/gdc/RealTimeFluids.pdf>
[24] LORACH, T. *Soft particles* [online], Nvidia SDK White Paper, 2007. [cit. 24-5-2009]. Dostupné z WWW: <http://developer.download.nvidia.com/SDK/10.5/direct3d/Source/SoftParticles/doc/SoftParticles_hi.pdf>
[25] ŽÁRA, J. - BENEŠ, B. - FELKER, P. *Moderní počítačová grafika: kompletní průvodce metodami 2D a 3D grafiky.* 2. přepracované a rozšířené vydání. Brno, Computer Press, 2004. 609 s. ISBN 80-251-0454-0.
[26] URALSKY, Y. - AHMAD, A. *Soft Shadows* [online], Nvidia SDK White Paper, 2004. [cit. 24-5-2009]. Dostupné z WWW: <http://developer.download.nvidia.com/SDK/9.5/Samples/DEMOS/Direct3D9/src/HLSL_SoftShadows/docs/HLSL_SoftShadows.pdf>
[27] MEYERS, K. *Variance Shadow Mapping* [online]. Nvidia SDK White Paper, 2007. [cit. 24-5-2009]. Dostupné z WWW: <http://developer.download.nvidia.com/SDK/10.5/direct3d/Source/VarianceShadowMapping/Doc/VarianceShadowMapping.pdf>
[28] BAVOIL, L. *Percentage Closer Soft Shadows* [online]. Nvidia SDK White Paper, 2008. [cit. 24-5-2009]. Dostupné z WWW: <http://developer.download.nvidia.com/SDK/10.5/direct3d/Source/PercentageCloserSoftShadows/doc/PercentageCloserSoftShadows.pdf>
[29] XnView. *GFL SDK*. 2009. [cit. 24-5-2009]. Dostupné z WWW: <http://www.xnview.com/en/gfl.html>
[30] BERGHEN, V. F. *XML Parser*. 2008. [cit. 24-5-2009]. Dostupné z WWW: <http://www.applied-mathematics.net/tools/xmlParser.html>
[31] DUDASH, B. *Texture Arrays* [online]. Nvidia SDK White Paper, 2007.[cit. 24-5-2009]. Dostupné z WWW: <http://developer.download.nvidia.com/SDK/10.5/direct3d/Source/TextureArrayTerrain/doc/TextureArrayTerrain.pdf>
[32] TARIQ, S. *Rain* [online]. Nvidia SDK White Paper, 2007. [cit. 24-5-2009]. Dostupné z WWW: <http://developer.download.nvidia.com/SDK/10.5/direct3d/Source/rain/doc/RainSDKWhitePaper.pdf>

Seznam zkratek

1D, 2D, 3D, 4D	Označení počtu rozměrů
CA	Celulární automat
CPU	Central Processing Unit, procesor
CUDA	Compute Unified Device Architecture, původní význam se již nepoužívá
GLSL	OpenGL Shading Language
GPGPU	General-Purpose computing on Graphics Processing Units
GPU	Graphics Processing Unit
HLSL	High Level Shader Language
LOD	Level of Detail
OpenGL	Open Graphics Libray
OpenCL	Open Computing Language
RGBA	Barevný model obsahující složky: červená, zelená, modrá, alpha

Seznam příloh

Příloha A: Ukázky shaderů

Příloha B: Manuál

Příloha C: DVD

- Text práce v elektronické podobě, bakalářská práce, semestrální projekt, dokumentace k projektům do PGR, MUL a VIN.
- Zdrojové texty a dokumentace z Doxygenu
- Spustitelná verze demonstračního programu
- Screenshoty a videa z programu

A. Ukázky shaderů

Ukázky shaderů jsou uvedeny ve formátu popsaném v rámci implementace, který umožňuje specifikovat všechny shadery pro jeden OpenGL shader program v jednom souboru.

Výpočetní shader – počítání změny rychlostí

```
#PROGRAM{ Speeds }#
/* Compute the water transport speeds based on the gradients and some water
   surface auxiliary smoothing values. */

// Just transform the corners of the quad for the view
// and interpolate position in vPos.
#VERTEX{
  #SRC{ compute_vertex.glsl }# // use the specified file
}#

// Real work - water speed and surface smooth
#FRAGMENT{
  #CODE{
    #version 130

    uniform ivec2 uSize;          // data texture size
    uniform sampler2D uTerrain;   // terrain data texture
    uniform sampler2D uWater;     // water speeds texture
    uniform float uStep;          // simulation step

    in vec4 vPos;                 // input vertex position form vertex shader

    out vec4 outSpeeds;           // output speeds
    out vec4 outAux;              // output aux data

    // water smoothing variables and constants
    const float MINW = 0.1;       // water surface minimal depth before
                                  // smoothing is applied
    const float MINW_MUL = 1.0f / MINW; // speedup constant

    // variables are global for simple function calls
    float new_w = 0;    // accumulator for new water z value
    float cnt = 0;      // dividing factor for new_w
    float center = 0;   // center water z value
    int mid_spec = 0;   // counts surrounding higher terrain values

    // water smoothning function called for 8 surrounding points
    void WAC(vec4 tx)
    {
      float f = tx.y * MINW_MUL;

      if(tx.x+tx.y < center){ // water height is less than center water
        new_w += ( tx.x + tx.y ) * f;
        cnt += f;
      }
      else           // surrounding terrain is higher, for shallow water
        mid_spec++;  //   holes reduction
    }
    // simple wrap for texture fetches
    vec4 GetTexel(sampler2D sampler, float x, float y)
    {
      return texture(sampler, (vPos.xy + vec2(x, y)) / uSize);
    }
```

```
    // compute the water speeds and surface info
    void main(void)
    {
      vec4 terr   = GetTexel(uTerrain, 0.0, 0.0);   // terrain texel
      vec4 speeds = GetTexel(uWater, 0.0, 0.0);     // water speed for this texel
      float w = terr.x + terr.y;                    // water surface initial Z

      vec4 tx[8];
      tx[0] = GetTexel(uTerrain, 1.0, 0.0); // 8 surrounding texels for:
      tx[1] = GetTexel(uTerrain, 0.0, 1.0); // 1. 4 for speed computation
      tx[2] = GetTexel(uTerrain, -1.0, 0.0);// 2. all 8 for water surface smoothing
      tx[3] = GetTexel(uTerrain, 0.0, -1.0);
      tx[4] = GetTexel(uTerrain, 1.0, 1.0);
      tx[5] = GetTexel(uTerrain, 1.0, -1.0);
      tx[6] = GetTexel(uTerrain, -1.0, 1.0);
      tx[7] = GetTexel(uTerrain, -1.0, -1.0);

      // compute display water Z for this point
      float display_w = w;
      if(terr.y < MINW){
        center = terr.x+terr.y;    // initialize variable used in WAC

        for(int i = 0; i < 8; i++) // compute water average
          WAC(tx[i]);

        if(cnt > 0.0){             // compute average
          float cf = terr.y * MINW_MUL;
          display_w = w * cf + (new_w / cnt) * (1.0 - cf);
        }
        else if(mid_spec < 8){     // offset the water layer from terrain
          display_w = terr.x + terr.y * (1.0 + 2.0 / MINW) - 2.0;
        }
      }

      // compute normals from surrounding texels,
      // using simple "2D" normals and addition,
      // storing to auxiliary texture for use in display frag shaders
      outAux = vec4(normalize(vec3(tx[2].x - tx[0].x, tx[3].x - tx[1].x, 2.0)),
                    display_w);

      // water surface Z for neighbours
      vec4 ws = vec4(tx[0].x + tx[0].y, tx[1].x + tx[1].y,
                     tx[2].x + tx[2].y, tx[3].x + tx[3].y);

      // compute/update speeds of water transfer
      speeds += (vec4(w) - ws) * uStep; // modify the spped by the gravity
      speeds *= 0.97;                   // damping
      speeds = max(speeds, 0.0);        // resolve negative speeds

      // resolve the situations when the speeds would transfer more water
      // than is available
      float total = (speeds.x + speeds.y + speeds.z + speeds.w) * uStep;
      if(total > 0 && total > terr.y){
        speeds *= terr.y / total;
      }

      // write the result
      outSpeeds = speeds;
    }
  }#
}#
```

Zobrazovací shader – zobrazení terénu

```
#PROGRAM{ Terrain }#
/* Shape and shade terrain for display. */

#DEFINE{
  // common pragmas and variables
  #COMMON{
    #version 120

    uniform sampler2D uTerrain;     // terrain data texture
    uniform ivec2 uSize;            // terrain size
  }#
}#

/*
  Vertex shader shapes the flat vertices by the values from the terrain data
  texture computed by shaders on GPU. It also computes shadow texture
  coordinates. All other operations are done per fragment.
*/
#VERTEX{
  #USE{ COMMON }#                 // header and varying

  #CODE{
    uniform mat4 uMatMVP;         // gl_ModelViewProjectionMatrix
    uniform mat4 uMatShadow;

    varying vec4 vPos;            // position of the fragment
    varying vec4 vCoordShadow;    // coordinates of the shadow texture

    // all the work...
    void main(void)
    {
      // modify terrain altitude by data from the texture
      vPos = gl_Vertex;
      vPos.z += texture2D(uTerrain, (gl_Vertex.xy + vec2(0.5)) / uSize).x;

      vCoordShadow = uMatShadow * vPos; // compute shadow texture coordinates
      gl_Position = uMatMVP * vPos;     // compute vertex transformation
    }
  }#
}#

/*
  Computes the terrain type and mixes color by the results.
  Mix a few samples of 3D noise to modify normal based on terrain type, find
  texture samples for detailed 2D, large scale 2D and 1D layers (modified by
  noise). Then mix it all together, and compute shadow map influence. Finally add
  the water depth fog and draw cursor where necessary.
*/
#FRAGMENT{
  #USE{ COMMON }#         // header and varying, uTerrain & uSize
  #SRC{ cursor.glsl }#    // cursor drawing, includes uniform vec4 uCursor
  #SRC{ pcf.glsl }#       // shadow drawing function
  #CODE{
    uniform sampler2D       uAux;         // terrain data texture - normals
    uniform sampler2D       uBase;        // terrain visual texture
    uniform sampler1D       uHeight;      // terrain layers texture
    uniform sampler3D       uNormal;      // noise texture for normal modification
    uniform sampler2DShadow uShadow;      // shadow texture
    uniform float           uShadowSize;  // shadow testure size
    uniform int             uShowShadows; // whether shadows should be rendered
    uniform int             uCursorType;  // cursor type, currently just on/off
```

```
  uniform vec4            uEye;         // required by the fog computation

  varying vec4 vPos;                    // position of the fragment
  varying vec4 vCoordShadow;            // coordinates of the shadow texture

  // terrain type colors
  uniform vec4 uMudColor;
  uniform vec4 uGrassLColor;
  uniform vec4 uGrassHColor;
  uniform vec4 uRockColor;
  uniform vec4 uSnowColor;
  uniform vec4 uWaterDepthColor;

  // terrain texturing coeficients
  uniform float uWaterDepth;
  uniform float uMudMul;
  uniform float uSnowMin;
  uniform float uSnowMax;
  uniform float uSnowSlope;
  uniform float uSnowSlopeSmooth;
  uniform float uSnowSlopeLow;
  uniform float uSnowWater;
  uniform float uRockMin;
  uniform float uRockMax;
  uniform float uRockSlopeMin;
  uniform float uRockSlopeMax;
  uniform float uGrassMin;
  uniform float uGrassMax;
}#

#SRC{ fog.glsl }#       // fog computation - requires vPos and uEye to be defined

#CODE{
  // just a simple shortcut for interpolated data textur fetching
  vec4 GetTexel(sampler2D s, vec2 pos, float x, float y)
  {
    return texture2D(s, (pos + vec2(x, y)) / uSize);
  }

  // the main texturing work
  void main(void)
  {
    // basic texture reading
    // ---------------------

    // get the basic geometry variables
    vec4 terrain_texel = GetTexel(uTerrain, vPos.xy, 0.5, 0.5);
    vec3 normal = GetTexel(uAux, vPos.xy, 0.5, 0.5).xyz;
    vec3 pos = vec3(vPos.xy, terrain_texel.x); // use texture detail z values

    // several noise samples for repetition reduction
    // the W component holds heightfield height for horizontal lines displacement
    // noise1 = very large scale, noise2 = large, noise3 = detail,
    // noise4 = high detail, noise5 is used for heightfield displacement
    vec4 noise1 = texture3D(uNormal, pos.xyz * 0.053) * 2.0 - 1.0;
    vec3 noise2 = texture3D(uNormal, pos.xyz * 0.31).rgb * 2.0 - 1.0;
    vec3 noise3 = texture3D(uNormal, pos.xyz * 1.31).rgb * 2.0 - 1.0;
    vec3 noise4 = texture3D(uNormal, pos.xyz * 10.31).rgb * 2.0 - 1.0;
    vec4 noise5 = texture3D(uNormal, pos.xyz * 0.0052) * 2.0 - 1.0;

    // neaded to too eliminate sharp transitions caused by big differences in
    // water_average
    vec2 plus = pos.xy + vec2(noise1.xy * 2 + noise2.xy +
                              noise3.xy * 0.5 + noise4.xy * 0.25);
```

```
// averaged water average
float water_average = (terrain_texel * 4 +
  (GetTexel(uTerrain, plus, 1.5, 1.5) +
  GetTexel(uTerrain, plus, 1.5, -0.5) +
  GetTexel(uTerrain, plus, -0.5, 1.5) +
  GetTexel(uTerrain, plus, -0.5, -0.5)) * 2 +
  GetTexel(uTerrain, plus, 1.5, 0.5) +
  GetTexel(uTerrain, plus, -0.5, 0.5) +
  GetTexel(uTerrain, plus, 0.5, 1.5) +
  GetTexel(uTerrain, plus, 0.5, -0.5)).w / 12.0;

// terrain type determination
// --------------------------

// rock is in high attitudes or on slopes
float rock_smooth = uRockMax - uRockMin; // speedup variable
float f_rock = clamp(smoothstep(uRockMin, uRockMax,
                                pos.z + (1 - normal.z) * rock_smooth) +
                     smoothstep(uRockSlopeMax, uRockSlopeMin, normal.z),
                     0.0, 1.0);

// mix the noises and modify the final normal, get two different mixes
vec3 noise_rock = normalize(noise1.rgb) * (0.2 + f_rock * 0.5); // speedup
vec3 noise    = noise_rock + normalize(noise2) * 0.3
              + normalize(noise3) * 0.4 + normalize(noise4) * 0.3;
vec3 noise_sm = noise_rock + normalize(noise2) * 0.2
              + normalize(noise3) * 0.1 + normalize(noise4) * 0.05;
noise.z = noise_sm.z = 0.0;                   // the Z value is not necessary
float noise_mul = 0.5 + pow(f_rock * 1.5, 2.0);  // noise impact on normal
vec3 n1 = normalize(normal + noise * noise_mul); // basic nontransf. normal
vec3 n_sm = normalize(normal + noise_sm * noise_mul); // smoother nontrans. n
vec3 n = normalize(gl_NormalMatrix * n1);     // basic transformed light n

// snow can be only high enough and on decent slopes, water melts it
float f_snow = clamp(smoothstep(uSnowMin, uSnowMax,
                                pos.z + (n_sm.z * uSnowSlopeLow))
                     - (1.0 - smoothstep(uSnowSlope,
                                         uSnowSlope + uSnowSlopeSmooth,
                                         n_sm.z))
                     - max(water_average * uSnowWater - (1.0 - n1.z), 0.0),
                     0, 1);

// recomputation of rock factor with modified normals
f_rock = clamp(smoothstep(uRockMin, uRockMax,
                          pos.z + (1 - n_sm.z) * rock_smooth)
                          + smoothstep(uRockSlopeMax, uRockSlopeMin, n_sm.z),
               0.0, 1.0);

// mud is where water is, but has less impact on rocks
float f_mud = max(clamp(water_average * 0.7 * uMudMul - (1.0 - n1.z),
                  0.0, 1.0) - f_rock * 0.5, 0.0);

// update rock factor once more, it's covered by mud and snow
f_rock = clamp(f_rock - (f_snow + f_mud),  0.0, 1.0);

// grass is everyvhere else
float f_grass_l = clamp(1.0 - (pow(clamp(water_average * 10 - f_rock,
                                   0, 1.0), 2)
                               + f_rock + f_snow + f_mud
                               + smoothstep(uGrassMin, uGrassMax,
                                 pos.z + n_sm.z * (uGrassMax - uGrassMin))),
                        0.0, 1.0);
```

```
      // higher altitude grass have different color, could also simulate sand
      float f_grass_h = clamp(1.0 - (f_rock + f_snow + f_mud + f_grass_l),
                              0.0, 1.0);

      // water depth attenuation
      float f_depth = smoothstep(0.0, uWaterDepth, terrain_texel.y);

      // final color assignment
      vec4 terrain_type_color = uSnowColor * f_snow + uRockColor * f_rock
        + uGrassHColor * f_grass_h + uGrassLColor * f_grass_l + uMudColor * f_mud;

      // texturing and ligting
      // ---------------------

      // get visual texture samples
      vec4 tex0 = texture2D(uBase, pos.xy / 3.0);     // detail
      vec4 tex1 = texture2D(uBase, pos.xy / 20.0);    // large scale
      vec4 tex2 = texture2D(uBase, pos.xy / 100.0);   // very large scale
      vec4 tex3 = mix(vec4(1.0, 1.0, 1.0, 1.0),        // horizontal rock layers
                  texture1D(uHeight, pos.z / 200.0 + noise1.w / 100
                                                   + noise5.w / 20.0),
                  f_rock * 2.0);                       // display mainly on rocks

      // recompute normal modification impact for lighting
      n = normalize(gl_NormalMatrix
        * normalize(normal + noise * (0.5 + pow(f_rock * 1.5, 2.0))));

      // compute lighting diffuse lighting, no specular neaded
      vec4 color = terrain_type_color * gl_LightSource[0].ambient;
      float NdotL = max(dot(n, normalize(gl_LightSource[0].position.xyz)), 0.0);
      if(NdotL > 0.0){
        float shadow = ShadowPCF(uShowShadows, vCoordShadow, uShadow, uShadowSize);
        color += terrain_type_color * gl_LightSource[0].diffuse * NdotL * shadow;
      }

      // write the resilting color
      gl_FragColor = tex0 * tex1 * tex2 * tex3; // color from textures
      gl_FragColor *= color;                    // apply the light and material

      // water depth and fog
      gl_FragColor = mix(gl_FragColor,
                         mix(uWaterDepthColor, uWaterDepthColor * 0.5, f_depth),
                         f_depth);
      gl_FragColor = Fog(gl_FragColor);
      gl_FragColor.a = 1.0;

      // draw cursor?
      float cf = CursorFade(pos.xy);
      if(uCursorType == 1 && ((cf > 0.01 && cf < 0.05) || cf > 0.99))
        gl_FragColor = mix(gl_FragColor, vec4(1.0, 0.0, 0.0, 1.0), 0.5);
    }
  }#
}#
```

B. Manuál

Požadavky na spuštění

Program je napsaný pro operační systém Microsoft Windows XP a výše, testovaný zejména na 32-bitových Windows Vista. Vyžaduje grafickou kartu schopnou zvládnout OpenGL verze 3.0 s rozšířením pro geometrické shadery a aktuální drivery. Mezi karty schopné zvládnout nároky programu patří například:

- NVIDIA: GeForce od řady 8000 výš nebo řada Quadro FX. Bližší informace k OpenGL 3.0 na kartách NVIDIA je možno nalézt na stránce
  http://developer.nvidia.com/object/opengl_3_driver.html
- ATI: Minimálně Radeon HD 2400 s drivery Catalyst 9.1

Spuštění programu

Program není potřeba nijak instalovat. Je možné ho spustit přímo z přiloženého DVD, ovšem po nakopírování programu na disk nebo jiné zapisovatelné médium se budou některé ladící informace, jako například textová konzola, přepisovat do souborů, což může být užitečné při hledání případného problému.

Programu je možné zadat jako parametr konfigurační soubor, který je schopen nastavit velké množství proměnných přímo při startu programu, i když většina se dá následně za běhu změnit. Konfigurační soubor je ve formátu XML a má následující formát:

Soubor by měl začínat standardní XML hlavičkou, následovanou hlavním tagem *ini*. Uvnitř tagu *ini* se mohou vyskytovat následující vnořené tagy:

- *program*: Atributy se vztahují k funkci programu.
  - *swap_interval*: Číslo od nuly výš, které určuje počet plně vykreslených obrazovek před vykreslením nového stavu scény – vertikální synchronizace.
  - *camera_control*: Jakým způsobem se bude kamera držet nad povrchem. 0 = volný pohyb včetně procházení terénem, 1 = držení kamery nad terénem, 2 = zajištění viditelnosti cílového bodu.
  - *tex_shadow_size*: Velikost stínové textury.
  - *veg_density*: Počet rostlin na jednom čtverci terénu.
  - *mode*: V jakém režimu se bude program nacházet po startu. 0 = editování a ovládání programu, 1 = nastavení texturování, 2 = nastavení post-processingu.

- *walk_height*: Desetinná hodnota určující zvednutí cíle kamery nad terénem v metrech.
- *walk_speed*: Desetinná hodnota určující rychlost pohybu pomocí kurzorových kláves.
- *auto_rotate*: Zadáním jakékoli hodnoty jiné než 0 se zapne automatické otáčení scény.
- *aquarium*: Jakákoli jiná hodnota než 0 způsobí, že voda na okrajích neodtéká z terénu pryč a hromadí se na něm.
- *stop_rain, stop_erosion, stop_water, stop_water_shader*: Hodnoty rozdílné od 0 znamenají zastavení deště, eroze, simulace vody nebo animace vodního shaderu.
- *show_toolbox, show_terrain, show_water, show_vegetation, show_shadows, show_sky, show_box*: Hodnoty 0 určují, že se nebude vykreslovat toolbox, terén, voda, vegetace, stíny, obloha nebo ohraničují kvádr.

- *terrain*: Vlastnosti terénu.
  - *tex_base_size. tex_height_size*: Velikosti základní textury terénu a textury zobrazující vrstevnice.
  - *x, y*: Rozměry generovaného terénu.
  - *height*: Určuje rozptyl hodnot nadmořské výšky v terénu.
  - *type*: Určuje tvar terénu. 0 = rovnoměrné rozložení, 1 = větší kopec uprostřed, 2 = jezero uprostřed, 3 = údolí, 4 = hřeben
  - *smooth*: Určuje kolik vyhlazovacích kroků je provedeno po vygenerování základního terénu.
- *water*: Vlastnosti vody:
  - *tex_size:* Rozměr normálové textury vody ve 2D.
  - *frames*: Počet snímků v animaci textury vody.
- *skybox*: Vlastnosti oblohy:
  - *tex_size*: Velikost hrany krychlové textury oblohy.
  - *clouds, turbulence, stars*: Desetinné hodnoty v rozsahu 0 až 1 určující pokrytí oblohy mraky, nakolik se v nich projevují turbulence a kolik je na noční obloze hvězd.
- *postproc*: Vlastnosti postprocessingu:
  - *enable*: Pokud má být postprocessing povolen, je nutno zadat hodnotu různou od 0.

Je připraveno několik konfiguračních souborů a k nim příslušných dávkových souborů pro snadné spuštění programu s určitou sadou parametrů. Pokud je program spuštěn bez zadaného souboru, použijí se vestavěné počáteční hodnoty, které se použijí i pro všechny proměnné nespecifikované v souboru.

Ovládání

Program se ovládá převážně myší, nicméně mnoho akcí je možno pohodlněji provést pomocí mnoha klávesových zkratek a pro některé akce je nutno použít kombinaci klávesnice s myší. Většinu akcí lze vyvolat pomocí menu, kde jsou uvedeny i příslušné zkratky. Další ovládací prvek tvoří nemodální dialog pevně ukotvený v levé části hlavního okna, ve kterém je několik záložek sdružujících ovládání určitého okruhu funkcí.

Pohyb ve scéně

Ve scéně je možno se pohybovat pomocí myši nebo s využitím kurzorových kláves. Posouvat terénem je možno podržením levého tlačítka a pohybem myši. Pohyb myši s pravým zmáčknutým tlačítkem scénou otáčí a při velkém přiblížení kamery k cíli tak i určuje směr pohledu kamery při pohledu první osoby. Vzdálenost cíle od kamery je možné ovlivnit tažením myši nahoru a dolů se zmáčknutými oběma tlačítky nebo rolováním kolečka. Pokud je zmáčknutá klávesa *Ctrl* a roluje se kolečkem, rotuje kamera okolo osy pohledu, s klávesou *Shift* kolečko nastavuje zorný úhel.

Kurzorové klávesy slouží pouze k pohyby dopředu, dozadu a k úkrokům do stran. Rychlost pohybu závisí na vzdálenosti kamery od svého cíle a se vzrůstající vzdáleností se zvyšuje. Dále lze rychlost pohybu ovlivnit posuvníkem *Walk speed* v záložce *Control* v postranním dialogu. Podobně jde také posuvníkem *Walk height* nastavit zvednutí kamery nad terénem.

Editování terénu

Aby bylo možno terén editovat, je nutné být v režimu *Control*, což je poznat jednak ze zobrazení stejnojmenné záložky v postranním dialogu a jednak ze zobrazení červeného kruhu symbolizujícího kurzor. Ten se vždy nachází ve středu zobrazení a jeho velikost lze ovlivnit potáhnutím myši se zmáčknutou klávesou *Shift* a oběma tlačítky myši.

Samotné editování lze provést několika způsoby:

- Při držení klávesy *Shift* a levého tlačítka myši lze pohybem myši nahoru a dolů terén přidávat a ubírat.
- Stisknutím klávesy *M* je provedena v oblasti kurzoru modifikace vybraná z nabídky v postranním dialogu s tamtéž určenou intenzitou.
- Podobně lze přímo klávesou *S* provést vyhlazení oblasti kurzoru, *N* způsobí přidání šumu, *A* přidá výšku, *Alt* + *A* výšku ubere.

Menu a klávesnice

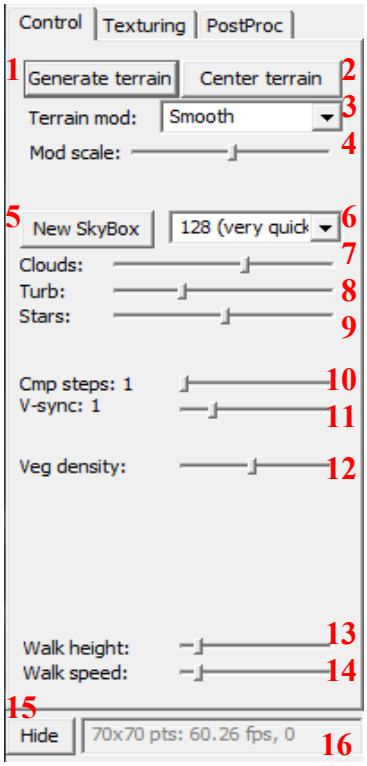
Většinu jednorázových operací lze vyvolat pomocí menu programu nebo klávesových zkratek, které jsou u dané položky uvedeny. Menu má následující složky:

- File: Možnosti uložení screenshotu a videa, znovunačtení shaderů a ukončení programu. Video se ukládá jako série RAW obrázků v RGB do adresáře *vid*. Velikost obrázků je uložena do textového souboru *vid-size.txt*. Ke složení obrázků do videa je potřeba použít další programy.
- Edit: Pouze umožňuje přepínat režim programu.
- View: Obsahuje položky, určující co všechno a jakým způsobem bude zobrazeno.
- Terrain: Zde je možno zadat příkazy pro celkovou modifikaci terénu.
- Water: Položky ovládající pohyb vody.
- Help: Jednoduchá nápověda.

Dialog

Postranní dialog lze zobrazit nebo schovat pomocí klávesy *F9*. Obsahuje několik záložek, které zároveň reprezentují několik režimů práce programu.

- Control: Záložka sdružující ovládání funkce programu a editování terénu.



1: Generování nového terénu se stejnými parametry.

2: Umístění kurzoru na střed terénu.

3: Výběr druhu modifikace terénu.

4: Nastavení intenzity modifikací.

5, 6: Generování nové oblohy v zadané kvalitě. Generování bohužel může chvíli trvat.

7, 8, 9: Nastavení oblačnosti, turbulencí a množství hvězd na nové obloze.

10: Počet výpočetních kroků na jedno zobrazení.

11: Vertikální synchronizace obrazovky.

12: Hustota vegetace. Úplně se vypne klávesou *V*.

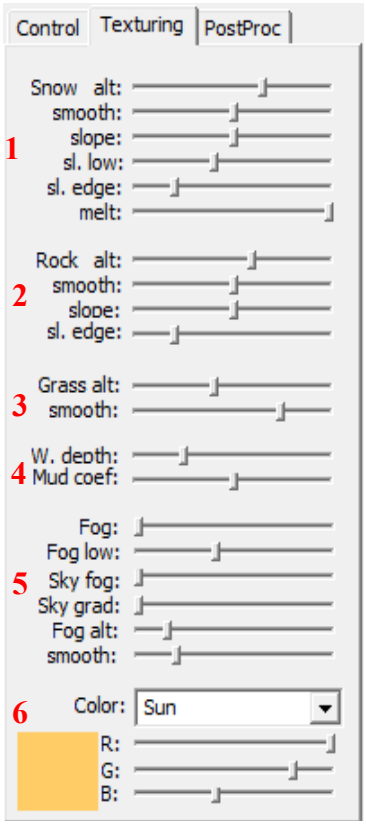
13: Zvednutí cíle/pozice kamery nad terénem.

14: Rychlost chůze pomocí kurzorových kláves.

15: Tlačítko pro schování dialogu (*F9*).

16: Lišta zobrazující velikost terénu a aktuální FPS.

- Texturing: Nastavení texturování terénu.



1: Nastavení texturování sněhu:

- *Snow alt*: Nadmořská výška začátku sněhu.
- *smooth*: Ostrost linie začátku sněhu.
- *slope*: Jak moc se sníh drží na vysokých svazích.
- *sl. low*: Jak svah ovlivní linii začátku sněhu.
- *sl. edge*: Ostrost hrany sněhu na svazích.
- *melt*: Jak moc voda sníh rozpouští.

2: Nastavení texturování skály:

- *Rock alt*: Nadmořská výška začátku skal.
- *smooth*: Ostrost linie začátku skal.
- *slope*: Výskyt skal i níž na prudších svazích.
- *sl. edge*: Ostrost hrany nízko položených skal.

3: Nastavení texturování trávy:

- *Grass alt*: Nadmořská výška začátku trávy.
- *smooth*: Ostrost přechodu s druhým typem trávy.

4: Nastavení texturování spojeného s vodou:

- *W. depth*: Jak moc se projevuje zbarvení v důsledku výšky vodního sloupce.
- *Mud coef*: Míra rozbahnění terénu v místech s nižší hladinou vody.

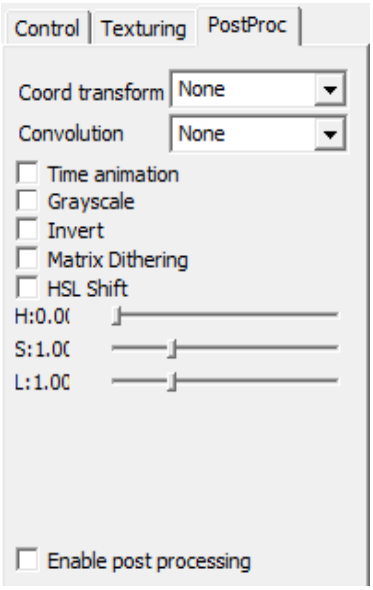
5: Nastavení zobrazení mlhy:

- *Fog*: Základní hustota mlhy ve vyšších výškách.
- *Fog low*: Hustota mlhy v nížinách.
- *Sky fog*: Celkové zamlžení oblohy.
- *Sky grad*: Gradient zamlžení oblohy, kdy obzor je zamlženější.
- *Fog alt*: Nadmořská výška začátku přechodu z řidší do hustší mlhy.
- *smooth*: Ostrost přechodu z řidší do hustší mlhy.

6: Nastavení barevnosti, kliknutím na barevný čtverec se zobrazí dialog výběru barvy:

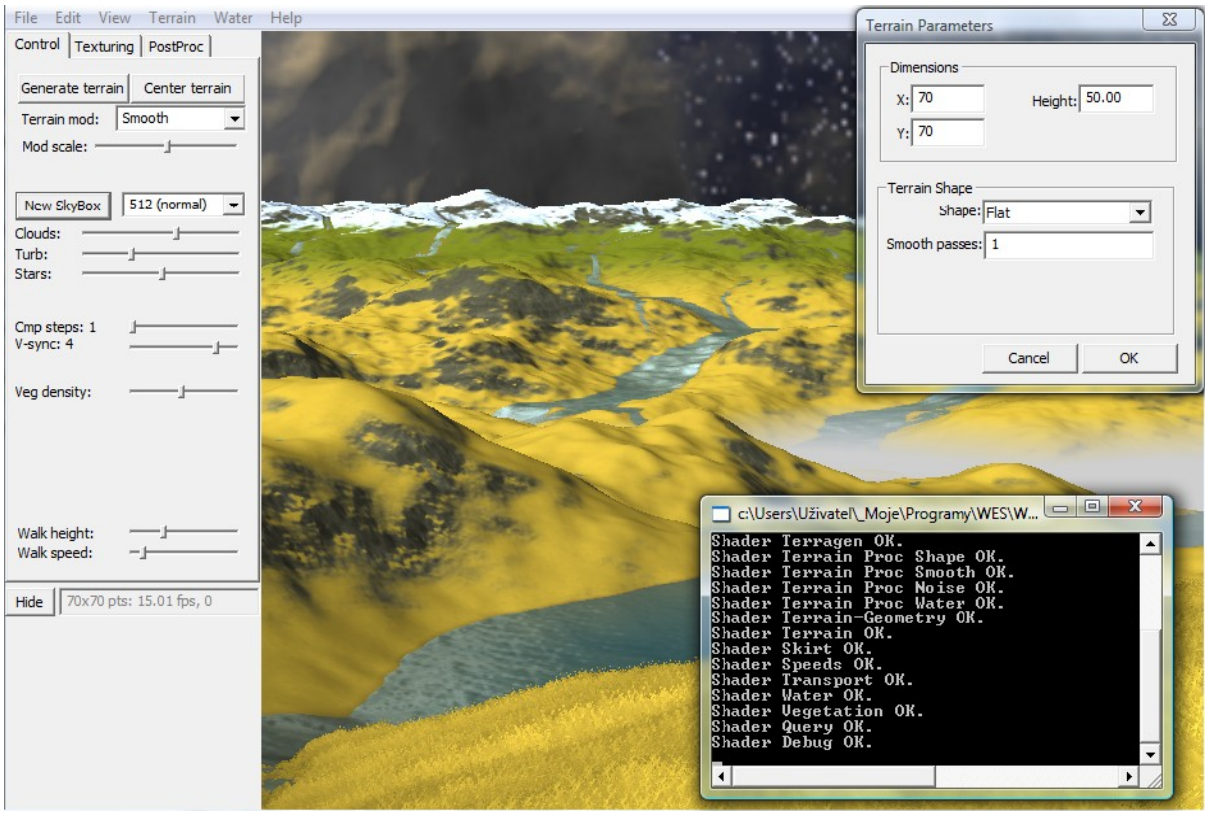
- *Color*: Výběr editované barvy ze seznamu.
- *R, G, B*: Složky editované barvy v pořadí: červená, zelená, modrá.

- PostProc: Nastavení postprocessingu celé scény.



  - Coord transform: Zde lze nastavit transformaci celého obrazu.
  - Convolution: Výběr konvolučních filtrů.
  - Time animation: Některé efekty se mohou měnit v čase.
  - Grayscale: Převede celý obraz do stupňů šedi.
  - Invert: Převrátí barevné složky.
  - Matrix dithering: Maticové rozptýlení se stupni šedi.
  - HSL Shift, H, S, L: Posun barev pomoci modelu hue-saturation-lightness.
  - Enable post processing: Povolení postprocessingu.

Pomocí současně zmáčknutých kláves *Shift* a *Ctrl* a tažení myší s pravým tlačítkem lze navíc v režimu postprocessingu měnit některé jeho parametry, zejména výraznost transformací obrazu.



Obrázek 21: Téměř celé GUI programu - hlavní okno s postranním dialogem, dialog na generování nového terénu a textová konzola.