Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Luka Maške

SIMULACIJA IN VIZUALIZACIJA EROZIJE TERENA Z UPORABO GPE

diplomsko delo
na univerzitetnem študiju

Mentor: doc. dr. Matija Marolt

Ljubljana, 2013

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01966 / 2013
Datum: 15.10.2013

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **LUKA MAŠKE**

Naslov: **SIMULACIJA IN VIZUALIZACIJA EROZIJE TERENA Z UPORABO GPE**
SIMULATION AND VISUALISATION OF TERRAIN EROSION ON GPU

Vrsta naloge: DIPLOMSKO DELO UNIVERZITETNEGA ŠTUDIJA

Tematika naloge:

V diplomskem delu preučite postopke za generiranje geometrije realističnega terena v realnem času. Preučite načine za generiranje višinskih slik in simulacijo erozije, tako hidravlične kot termalne. Preučite tudi kako lahko v teren proceduralno dodate ceste s hevrističnim preiskovalnim algoritmom. Izbrane algoritme implementirajte na GPE in jih evaluirajte.

Mentor:

doc. dr. Matija Marolt

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:

prof. dr. Nikolaj Zimic

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Luka Maške, z vpisno številko **63080108**, sem avtor diplomskega dela z naslovom:

Simulacija in vizualizacija erozije terena z uporabo GPE

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Matije Marolta,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne Podpis avtorja:

Zahvaljujem se mentorju doc. dr. Matiji Maroltu za dosegljivost, nasvete in usmerjanje pri izdelavi diplomske naloge. Prav tako gre zahvala moji družini in dekletu, za vso podporo v času študija.

Kazalo

Seznam uporabljenih kratic

CPE - (*angl. Central Processing Unit*) Centralna Procesna Enota
GPE - (*angl. Graphics Processing Unit*) Grafična Procesna Enota
HLSL - (*angl. High-Level Shader Language*) Programski jezik za pisanje senčilnikov

Povzetek

Problem oblikovanja geometrije realističnega terena je v računalniški grafiki prisoten že od vsega začetka. Največkrat zasledimo uporabo realističnega terena v filmih, igrah ter raznih simulacijah. V diplomskem delu predstavimo postopek generiranja geometrije realističnega terena v realnem času, kar je mogoče le z vzporednim računanjem na GPE. Postopek začnemo z generiranjem naključnega terena s pomočjo fraktalov, ki kasneje služi kot bazna geometrija za simulacijo erozije. Na terenu nato v realnem času izvajamo simulacijo pretakanja tekočin in spremljamo, kako hidravlična erozija na terenu ustvari rečne struge. Termalna erozija zaradi sprememb temperature povzroči pokanje in posledično krušenje kamnin in tvorbo melišča na vznožju hribov in gora. Na koncu predlagamo še model proceduralnega generiranja geometrije cest. Najprej izračunamo energetsko najučinkovitejšo pot od točke A do točke B z ustrezno izbiro hevristične funkcije preiskovalnega algoritma A*. Dobljeno pot nato zgladimo ter zgradimo cesto in bankino poljubne širine. Končni rezultat diplomskega dela je realistična geometrija terena in cesta, ki realistično vijuga po terenu in se izogiba naravnim oviram.

Ključne besede: Proceduralno generiranje, teren, hidravlična erozija, termalna erozija, simulacija, GPE, tekočina, energetsko najučinkovitejša pot, A*.

Abstract

The problem of realistic terrain geometry generation has been present in computer graphics since the very begining. The most frequent use of realistic terrain can be seen in movies, games and various simulations. In our thesis, we present a procedure for real time generation of realistic terrain, which can only be achieved by parallel computing on the GPU. We start the process by generating a random terrain using fractals, which later serves as a base geometry for erosion simulation. Then we start the real time fluid simulation and observe how hydraulic errosion creates river beds on the terrain. Due to the changes in temperature, thermal erosion causes rocks to crack and therefore to crumble and cause the formation of scree at the bottom of hills and mountains. Finally we propose a model for procedural generation of the roads. First we calculate the most energy-efficient path from point A to point B by choosing the right heuristic function of the A* search algorithm. Then we smooth the resulting path and create the road and verge of arbitrary width. The final result of the thesis is a realistic geometry of the terrain and road that winds over the terrain and avoids natural obstacles.

Key words: Terrain generation, hydraulic erosion, thermal erosion, simulation, GPU, fluids, energy-efficient path, A*.

Poglavje 1

Uvod

Realistično oblikovanje terena po mnogih letih razvoja še vedno predstavlja velik izziv, saj se z razvojem stojne opreme vztrajno večajo tudi zahteve po realizmu. Kadar je potrebno narisati večje površine naravnih terenov, je skoraj nemogoče vse površine obdelati ročno, zato se je kmalu pojavila potreba po proceduralnem generiranju terena. Tako programer ali grafični ustvarjalec le nastavi potrebne parametre, program pa sam generira vso potrebno geometrijo. Generiranje realističnega terena je procesorsko zelo zahtevno delo, saj je treba v veliki večini primerov simulirati različne naravne efekte kot so na primer hidravlična erozija, ki s časoma vdolbe v teren rečne struge in odnese prst ter kamnine v nižje predele. Takšne simulacije zahtevajo ogromno procesorskega časa, vendar se jih da z nekaj truda paralelizirati, kar drastično pohitri celoten postopek. Zahvaljujoč proceduralnemu generiranju se razvijalci lahko osredotočijo na ostale naloge, ki jim procesorji ali algoritmi za enkrat še niso kos.

Celoten postopek generiranja terena ločimo na tri večje dele. Najprej izdelamo bazno geometrijo, ki približno posnema naravno razgibanost terena. Podatki so predstavljeni z višinsko karto, ki sicer ne omogoča predstavitve previsov ali jam, saj je na eni koordinati lahko le ena višinska točka, vendar pa poenostavi problem do te mere, da je celotno simulacijo možno izvajati v realnem času. V drugem delu sledi simulacija vode s pomočjo enačb plitvih voda (angl. shallow water equations) in simulacija erozije, ki glede na vodni tok spodje teren in ga nanese na spodnje dele vodotoka, kjer se voda umiri, razširi in začne izhlapevati. V tretjem delu predstavimo algoritem za proceduralno generiranje cest, ki skušajo čim manj poseči v teren ter istočasno vodijo po energetsko najcenejši poti od točke A do točke B.

1.1 Cilj diplomskega dela

V diplomskem delu želimo napisati program, ki bo naključno generiral realistično geometrijo terena, istočasno pa bi želeli spremljati simulacijo erozije v realnem času. Ker proces erozije v naravi potrebuje več tisoč let, da spremeni izgled terena, bomo tok vode simulirali z realnimi parametri, proces erozije pa bomo močno pohitrili. Lahko si tudi predstavljamo, da je teren zgrajen iz zelo mehke prsti in je zato proces erozije toliko hitrejši. Da bo simulacija lahko tekla v realnem času, bo potrebno simulacijo izvajati na GPE. Simulacijo vode bomo izvajali z modelom, ki le približno opisuje pretakanje tekočin, saj lahko le tako izvajamo simulacijo v realnem času. Na koncu simulacije želimo zgraditi na terenu eno ali več poti, ki bodo realistično vijugale med naravnimi ovirami, tako da bi do cilja porabili čim manj energije.

Izris simulacije bo zelo preprost, saj bomo uporabili le nekaj usmerjenih luči in Phongov model senčenja. Tekstur in ostalih podrobnosti ne bomo uporabili, saj je glavni namen generiranje geometrije.

Ključno je, da celoten postopek poteka povsem proceduralno, tako da uporabnik le nastavi parametre, program pa izračuna naključno geometrijo s pomočjo teh parametrov. Na ta način bo geometrija lahko služila tudi kot podlaga za računalniško igro, saj bo možno naključno geometrijo izračunati v nalagalnem času. Prav tako ne bo potrebno shranjevanje modela na disku, saj se bo izračunal ob zagonu programa in shranil v glavni pomnilnik.

Poglavje 2

Razvojno okolje

2.1 Programski jeziki in orodja

Za generiranje izhodiščnega terena smo uporabili programski jezik *C#* in ogrodje *.NET* v okolju *Visual Studio C# Express 2010*. Logiko za simulacijo vode ter hidravlične in termalne erozije smo napisali v jeziku HLSL, da je lahko simulacija tekla na GPE. Za vizualizacijo smo uporabili v jeziku *C#* napisano ogrodje *XNA 4.0*, ki omogoča komunikacijo med CPE in GPE ter vsebuje metode za upravljanje z GPE. Celoten projekt smo izdelali v operacijskem sistemu Microsoft Windows 8.

Če bralca zanima ogrodje XNA, naj povemo, da je Microsoft prenehal z razvojem in podporo tega ogrodja. Bralcu zato priporočamo ogled njegove odprtokodne izvedenke Monogame, ki je definitivno boljša izbira saj lahko teče tudi na ostalih platformah ter za razliko od XNA ponuja tudi novejše modele senčilnikov.

2.2 Strojna oprema

Za izvedbo in simulacijo smo uporabljali osebni računalnik s procesorjem Intel® Core™ i7 920 @ 2.67GHz, grafično kartico Nvidia GTX 680 ter 4GB DDR3 pomnilnika.

2.3 Sorodna dela

Modeliranje terena je bila pomembna tema že od samega začetka računalniške grafike. Na začetku so za modeliranje večinoma uporabljali fraktale [1, 2].

Glavna pomanjkljivost fraktalov je bila odsotnost nadzora nad generiranjem terena, zato so kmalu začeli razvijati modele, s katerimi bi imeli vsaj delni nadzor nad generiranjem. Tako so začeli uporabljati *L-sisteme* in razne metode za poljubno spreminjanje obstoječega terena. Šele z dejansko simulacijo pojavov kot je erozija terena [3, 4, 5], je teren zares dobil naravnejši izgled. Z današnjo tehnologijo lahko simuliramo večino naravnih pojavov na GPE in tako pohitrimo simulacijo.

Že nekaj let je možno generirati fotorealistični teren s pomočjo orodij kot je Terragen [6], ki je trenutno po našem mnenju najboljše orodje za generiranje realističnih terenov. Orodje združuje različne modele generiranja, ki so žal poslovna skrivnost podjetja. Njegova največja pomanjkljivost je, da potrebuje za izdelavo terena ogromno časa ter tako podaljša razvojne iteracije.

Za učinkovito simulacijo erozije je potreben tudi dober model vodnega toka. Simulacijo vode lahko dosežemo na več načinov, najbolj pogosto srečamo diferencialne valovne enačbe ali pa simulacijo s pomočjo več milijon delcev. Diferencialne valovne enačbe delujejo pravilno le v primerih, kjer je podlaga ravna in se ne spreminja, zato se največkrat uporabljajo za simulacijo oceanov ali jezer. Simulacija delcev je trenutno najboljši način za simulacijo tekočin, vendar je časovna zahtevnost tako velika, da simulacija ne bi tekla v realnem času. Na terenu bi lahko uporabili enačbe *Navier-Stokes* [7], ki odlično opisujejo gibanje tekočin po poljubnih površinah, vendar je njihova zahtevnost v treh dimenzijah prevelika, zato se v simulacijah erozije ne uporabljajo. Največkrat lahko v modelih erozije zasledimo enačbe plitvih voda (angl. Shallow water equations [8]), ki so izpeljane iz enačb *Navier-Stokes*, vendar so dosti bolj enostavne. Njihova slabost je, da lahko z njimi modeliramo le plitve tekočine, drugače pride do numeričnih nestabilnosti.

Slika 2.1: Primer terena, generiranega z orodjem Terragen.

Poglavje 3

Proceduralno generiranje naključnega terena

Pri generiranju bazne geometrije terena, okrog katere bomo kasneje gradili podrobno geometrijo, smo se omejili na predstavitev podatkov z 2D mrežo, kjer so vse razdalje med sosednjimi točkami v mreži enake, zato bo naš rezultat kar višinska karta, realizirana z 2D seznamom. Prva in druga dimenzija predstavljata koordinatni osi x in y, vrednost pa predstavlja višino na tej koordinati.

V praksi obstaja veliko različnih algoritmov za generiranje naključnih terenov. Razlikujejo se predvsem po časovni zahtevnosti in značilnih oblikah terena, ki ga generirajo. Večina algoritmov pa v končni fazi generira šum, ki pa je močno odvisen od svoje okolice. Zadnja leta je postal zelo popularen šum *Simplex* [9], ki je izboljšana različica znamenitega Perlinovega šuma [9]. Zanimive rezultate dajejo tudi algoritmi, ki za generiranje šuma uporabljajo fraktale. Med te algoritme sodita tudi algoritem premaknjene središčne točke (*angl. midpoint displacement*) in njegova izboljšava, *diamond-square* [10, 11].

Ker je algoritem diamond-square najbolj intuitiven in zelo hiter, smo se odločili, da bomo z njim generirali našo bazno geometrijo.

3.1 Algoritem Diamond-square

Algoritem začnemo s kreiranjem 2D tabele velikosti $2^n + 1$, kamor bomo shranjevali višine. Kot lahko ugotovimo že iz imena algoritma sta najpomembnejša koraka algoritma vzorčenje kvadratov in vzorčenje diamantov, kjer je diamant predstavljen kot kvadrat, rotiran za 45 stopinj. Algoritem sprejme dva parametra in sicer obseg naključnih števil, določen s spodnjo in zgornjo mejo, ter

faktor k, ki zmanjšuje ta obseg po končanem obhodu neke globine.

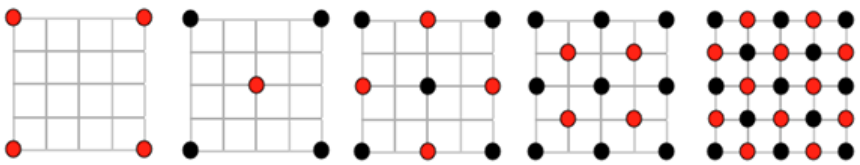
Najprej določimo naključne vrednosti za vse štiri robove in oba parametra, nato začnemo z izvajanjem algoritma:

1. *Faza diamantov:* za vsak kvadrat na trenutni globini izračunamo središčno točko tako, da izračunamo povprečje oglišč kvadrata in prištejemo naključno vrednost. S to fazo generiramo diamante.

2. *Faza kvadratov:* za vsak diamant na trenutni globini izračunamo središčno točko tako, da izračunamo povprečje oglišč diamanta in prištejemo naključno vrednost. S to fazo generiramo kvadrate.

3. Zmanjšamo obseg naključne vrednosti za faktor k, tako poskrbimo, da točke na nižjem nivoju konvergirajo k povprečju vzorčenega lika. Če je na primer začetni obseg velik 150 enot in k enak 0.5, bo v naslednjem obhodu obseg naključne vrednosti velik le 75 enot. Velikost likov, ki jih bomo vzorčili v naslednjem obhodu, se zmanjša za polovico, saj se globina poveča za 1.

4. Algoritem ponavljamo, dokler nismo izračunali vseh vrednosti, oziroma dokler je dolžina trenutno obravnavanega kvadrata večja od 2.



Slika 3.1: Prikaz poteka algoritma diamond-square. Rdeče točke predstavljajo novo razvite točke.

3.1.1 Prednosti

Prednost algoritma *diamond-square* je njegova hitrost in preprosta implementacija ter možnost, da lahko sami vstavimo začetne podatke do poljubne globine, algoritem pa nato nadaljuje do konca. Na ta način lahko "posejemo" hribe ali doline na poljubna mesta. Celotno višinsko karto shranimo tako, da shranimo le parametre, s katerimi smo zgradili karto. Edini pogoj je, da uporabljamo vedno isti generator naključnih števil. Najpomembnejši parameter je 32 bitna številka, ki predstavlja tako imenovano seme naključnega generatorja števil, ki z ostalimi

parametri sestavlja enolični identifikator višinske karte. Višinske razlike lahko nadziramo z velikostjo začetnega obsega naključnih vrednosti, gladkost terena pa s faktorjem k, ki po koncu vsakega obhoda zmanjša naključno vrednost tako, da jo pomnoži s tem faktorjem.

3.1.2 Slabosti

Zaradi narave algoritma je rezultat vedno kvadratno $2D$ polje dolžine $2^n + 1$, tako moramo v najslabšem primeru izračunati štirikrat več točk in jih nato odrezati. Podobno je treba obrezati kvadrat do poljubnih oblik, na primer do kroga, elipse ali poljubnih poligonov. Ker se algoritem vedno začne v štirih ogliščih kvadrata, lahko velikokrat opazimo podobnosti, saj bodo ekstremne razlike velikokrat najbolj vidne v teh štirih ogliščih.

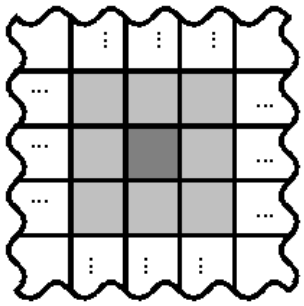
3.1.3 Primerjava z algoritmom midpoint displacement

Razlika med algoritmom *diamond-square* in *midpoint displacement* je ravno v dodani fazi, kjer generiramo diamante. Algoritem *midpoint displacement* začne s kvadratom in spet generira le kvadrate, pri tem pa se pojavijo artefakti v obliki kvadratov, kar pa diamond-square odpravi tako, da doda še eno fazo, kjer algoritem izmenično generira kvadrate in diamante. S to rešitvijo povsem odpravimo artefakte.

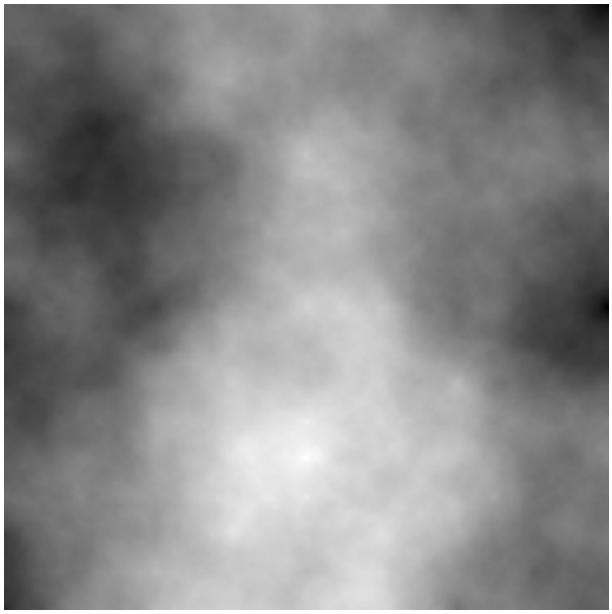
3.1.4 Realizacija

V našem primeru smo izbrali dimenzijo polja 513×513, kar pomeni, da bo teren predstavljen z 263.169 višinskimi točkami. Časovna zahtevnost algoritma je $O(n^2)$, kjer je n velikost dimenzije $2D$ polja, kar pomeni, da zahtevnost narašča linearno s številom točk. V algoritmu sta najzahtevnejši operaciji izračun naključnega števila in deljenje. Testi so pokazali, da je algoritem v povprečju potekal le 25ms pri velikosti 513×513. Čeprav se na prvi pogled zdi, da bo rekurzivna implementacija algoritma primernejša, smo dokaj hitro ugotovili, da je iterativna implementacija v tem primeru dosti enostavnejša, zato smo algoritem implementirali iterativno. Faktorju k smo določili vrednost 0.5, začetni obseg naključnih števil pa sega od 0 do 150.

Ker je algoritem kljub zmanjševanju naključne vrednosti včasih še vedno ustvaril kakšno špico na terenu, smo napisali še filter, ki za vsako točko pregleda še Moore-ovo okolico, ter v primeru, da je točka višja od vseh ostalih, izračuna povprečje vseh devetih točk ter ga zamenja z vrednostjo točke. Enako naredimo, če je točka nižja od vseh ostalih.



Slika 3.2: Moore-ova okolica.



Slika 3.3: Višinska karta naključno generiranega terena. Višinske karte najlažje predstavimo s črno belo sliko, kjer bela barva označuje višja področja, črna pa nižja. Potrebno je le normalizirati višine. Hitro lahko opazimo, zakaj algoritem *diamond–square* včasih opisujejo tudi kot plazemski ali pa oblačni fraktal.

Slika 3.4: Ista višinska karta, tokrat izrisana v treh dimenzijah. Ker je resolucija višinske karte le 513×513, so tla dokaj groba. Uporabili bi lahko bikubično interpolacijo, vendar bomo kasneje pokazali, da to sploh ni potrebno.

Poglavje 4

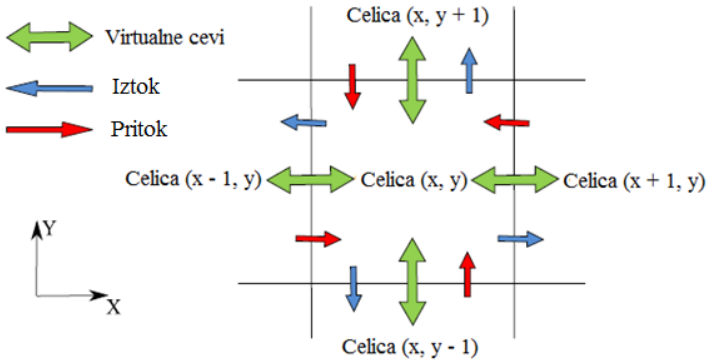
Model plitvih voda ter hidravlična in termalna erozija

Za simulacijo pretakanja tekočin po terenu potrebujemo poleg enostavnega, hitro izračunljivega modela tudi možnost paralelnega izvajanja na GPE, saj lahko le tako dosežemo, da bo simulacija tekla dovolj hitro. Omenimo naj še, da vseh algoritmov ni mogoče izvajati paralelno, zato je treba skrbno izbrati ustrezne modele in algoritme.

V tem poglavju bomo predstavili metode, s katerimi lahko v realnem času simuliramo obnašanje plitvih tekočin. S pomočjo podatkov, ki jih bomo pridobili med simulacijo tekočin, bomo nato simulirali še proces erozije. Če bi želeli simulirati pretakanje vode in proces erozije natančno in z vsemi znanimi parametri, bi morali uporabiti sistem delcev, kjer bi volumen vode opisali z milijoni majhnih delcev, ki bi natančno upoštevali fizikalne enačbe. Ta način zahteva dosti več procesiranja, zato simulacija v realnem času ne bi bila mogoča. Zato smo se odločili, da bomo za simulacijo vode uporabili model plitvih voda, ki zelo dobro in učinkovito opisuje dinamiko tekočin, dokler le te niso preveč globoke. Model je istočasno dovolj natančen za simulacijo erozije, saj za simulacijo ne potrebujemo globoke vode. Potrebno je bilo najti le še model, ki numerično reši diferencialne enačbe in je primeren za paralelno izvajanje.

4.1 Model virtualne cevi

Leta 1995 je James F. O'Brien predlagal model virtualne cevi za simulacijo plitvih tekočin [12], ki je primeren tudi za paralelno izvajanje. Voda se pretaka skozi cevi zaradi razlike v hidrostatičnem pritisku med sosednjima celicama. Isti model smo kasneje uporabili pri modeliranju erozije, kjer se po cevi pretaka suspenzija sedimenta in vode. Ker je pri tem modelu možno rezultat vedno shraniti v centralno celico, je kot nalašč primeren za implementacijo na GPE. Zakaj je možnost shranjevanja le v centralno celico bistvenega pomena, bomo razložili na strani 21.



Slika 4.1: Model virtualne cevi. V našem primeru bomo uporabljali Von Neumannovo okolico, saj je za primer simulacije vode dovolj natančna. Če bi želeli, bi lahko okolico razširili tudi na Moore-ovo, vendar bi simulacija tekla počasneje.

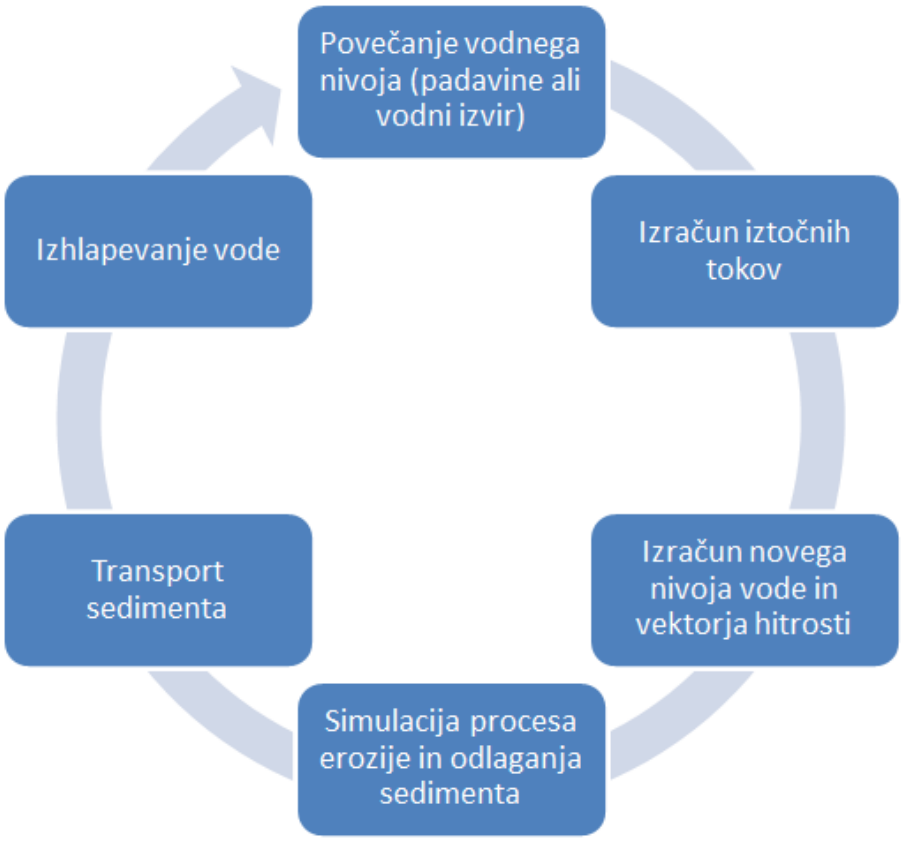
Model simulacije vode, hidravlične in termalne erozije ločimo na pet korakov. Najprej opišimo potrebne količine in notacije za proces simulacije:

- višina terena h
- višina vode w
- količina sedimenta s v suspenziji
- iztočni tok $f = (f_L, f_R, f_T, f_B)$
- vektor hitrosti $\vec{v} = (u, v)$

Za vsako celico moramo hraniti zgoraj opisane vrednosti, in jih posredovati simulaciji, ta pa nam vrne novo stanje celic. Pet korakov simulacije je:

1. Povečanje vodnega nivoja.
2. Simulacija vodnega toka s pomočjo prirejenega modela plitvih voda.
3. Proces erozije in odlaganja.
4. Transport suspenzije sedimenta.
5. Izhlapevanje vode.

Postopek ponavljamo, dokler ne dosežemo željenega učinka.



Slika 4.2: Shema poteka simulacije tekočin in erozije.

4.2 Simulacija vodnega toka

Na podlagi razlik v hidrostatičnih pritiskih lahko izračunamo količino in smer vodnih tokov. Ker mora GPE zagotavljati integriteto podatkov, ne smemo spreminjati vrednosti v sosednjih celicah, zato je potreben drugačen pristop. Rešitev je, da razbijemo potek na dva koraka. Najprej izračunamo le pozitivne iztočne tokove, v naslednjem koraku pa celici odštejemo njene pozitivne iztočne tokove, prištejemo pa iztočne tokove, ki pridejo v to celico iz sosednjih celic. Tako vedno zapišemo rezultat le v centralno celico.

4.2.1 Povečanje višine vodnega stolpca

Zaradi dežja ali izvirov rek povečamo nivo vode na terenu. Za simulacijo dežja smo uporabili enakomerno porazdelitev vode, saj bi z naključnimi padavinami v končni fazi dosegli enak rezultat. Naj bo $r_t(x, y)$ količina vode, ki prispe na celico (x, y) v vsaki časovni enoti. Nivo vode tako posodobimo z naslednjo enačbo:

$$w_1(x, y) = w_t(x, y) + \Delta t * r_t(x, y) \tag{4.1}$$

kjer oznaka w_1 pomeni vmesno višino vode, ki jo bomo še spreminjali tekom simulacije, Δt pa je časovni korak iteracije.

4.2.2 Izračun iztočnih tokov

Iztočni tok f_L proti levi celici iz celice $c(x, y)$ izračunamo takole:

$$f^L_{t+\Delta t(x,y)} = max(0, f^L_t \cdot A \cdot \frac{g \cdot \Delta h^L(x, y)}{l}) \tag{4.2}$$

kjer je A površina preseka cevi, g gravitacijski pospešek, l dolžina virtualne cevi in $\Delta h^L(x, y)$ višinska razlika med celico (x, y) in njenim levim sosedom $(x–1, y)$:

$$\Delta h^L_t(x, y) = h_t(x, y) + w_1(x, y)–h_t(x–1, y)–w_1(x–1, y) \tag{4.3}$$

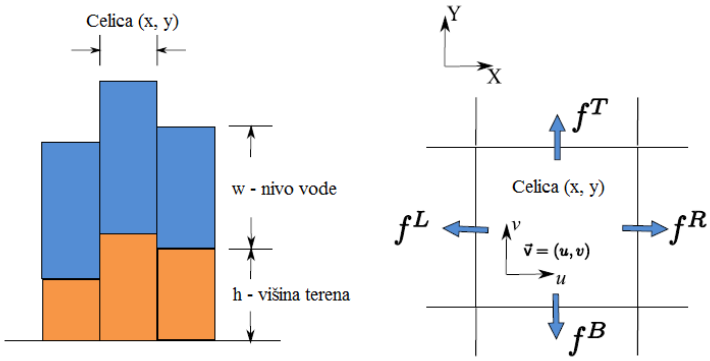
Za podrobno izpeljavo enačbe 4.2 naj bralec prebere O'Brienov origialni članek [15]. Iztočni tokovi F^R, F^T, F^B so izračunani na podoben način. Če količina iztočnega toka preseže količino vode v celici, je potrebno tok pomnožiti s faktorjem K, da se izognemo potencialno negativnemu posodabljanju višin vodnih stolpcev:

$$K = min(1, \frac{w_1 \cdot lx \cdot ly}{(f^L + f^R + f^T + f^B) \cdot \Delta t}) \tag{4.4}$$

Iztočni tok $f = (f^L, f^R, f^T, f^B)$ nato pomnožimo s faktorjem K:

$$f^i_{t+\Delta t}(x, y) = K \cdot f^i_{t+\Delta t}(x, y), i = L, R, T, B \tag{4.5}$$

Pri vsaki simulaciji tekočin je potrebno določiti robne pogoje. V našem primeru smo se odločili, da voda na robovih ne sme iztekati iz sistema. To lahko zagotovimo tako, da je iztočni tok robnih celiv v smeri pravokotni na rob vedno enak nič. Za celice na spodnjem robu velja: $f^B(x,0) = 0$. Podobno velja za ostale tri robne pogoje.



Slika 4.3: Iztočni tokovi.

4.2.3 Izračun novega nivoja vode in vektorja hitrosti

Sedaj lahko s pomočjo izračunanih iztočnih tokov za vsako celico posodobimo nivo vode tako, da od centralne celice odštejemo vse iztočne tokove, prištejemo pa iztočne tokove njenih sosedov, katere smeri kažejo proti centralni celici. Za vsako celico (x, y) je naslednja sprememba volumna vode enaka:

$$\begin{aligned} \Delta V(x,y) =& \Delta t \cdot (\sum f_{in} - \sum f_{out}) \\ =& \Delta t \cdot (f^R_{t+\Delta t}(x-1,y) + f^T_{t+\Delta t}(x,y-1) \\ &+ f^L_{t+\Delta t}(x+1,y) + f^B_{t+\Delta t}(x,y+1) \\ &- \sum_{i=L;R;T;B} f^i_{t+\Delta t}(x,y)) \end{aligned} \tag{4.6}$$

Nivo vode nato posodobimo na naslednji način:

$$w_2(x,y) = w_1(x,y) + \frac{\Delta V(x,y)}{lx \cdot ly} \tag{4.7}$$

V istem koraku lahko izračunamo še vektor hitrosti $\vec{v} = (u, v)$, ki je potreben za modeliranje erozije in odlaganja, izračunamo pa ga lahko iz iztočnih tokov. Najprej izračunamo povprečno količino vode, ki teče skozi celico (x, y) v enem

časovnem koraku. Količino v smeri x lahko izrazimo kot:

$$\Delta W_X = \frac{f^R(x-1,y) - f^L(x,y) + f^R(x,y) - f^L(x+1,y)}{2} \tag{4.8}$$

ΔW_X lahko izračunamo tudi s hitrostno komponento u v smeri X:

$$l_y \cdot \overline{w} \cdot u = \Delta W_X \tag{4.9}$$

kjer je $\overline{w} = \frac{(w_1+w_2)}{2}$. S pomočjo teh dveh enačb lahko izračunamo komponento u , na podoben način lahko izračunamo tudi komponento v.

4.3 Erozija in odlaganje sedimenta

Erozijo terena lahko opazimo na mestih, kjer tok vode odnaša prst in kamenje ter jih odlaga v nižjih, položnejših predelih. Proces erozije in odlaganja je v večji meri odvisen od transportne kapacitete sedimenta. V preteklosti je bilo predlaganih veliko modelov, ki opisujejo pojav erozije in z njo povezane kapacitete sedimenta. Največkrat je kapaciteta odvisna od hitrosti toka ter naklona lokalnega terena. Z eksperimenti smo prišli do formule, ki ponuja boljše rezultate kot pa predlagane metode.

Kapaciteto transporta sedimenta C za vodni tok v celici (x, y) izračunamo kot:

$$C(x,y) = K_c * sin(\alpha(x,y)) \cdot |\vec{v}(x,y)| \cdot w_2(x,y) \tag{4.10}$$

Kjer je K_c konstanta kapacitete sedimenta, $\alpha(x, y)$ lokalni naklon in v vektor hitrosti. C nato primerjamo z suspendiranim sedimentom s in če je $C > s_t$, se delež zemlje raztopi in doda k s.

$$\begin{aligned} h_{t+\Delta t} &= h_t - K_s \cdot (C - s_t) \\ s_1 &= s_t + K_s \cdot (C - s_t) \\ w_2 &= w_2 + K_s \cdot (C - s_t) \end{aligned} \tag{4.11}$$

kjer je K_s konstanta raztapljanja. Kadar pa je $C \leq s_t$, odložimo del suspendiranega sedimenta nazaj na teren:

$$\begin{aligned} h_{t+\Delta t} &= h_t - K_d \cdot (s_t - C) \\ s_1 &= s_t - K_d \cdot (s_t - C) \\ w_2 &= w_2 - K_d \cdot (s_t - C) \end{aligned} \tag{4.12}$$

V enačbah 4.11 in 4.12 na koncu še ustrezno popravimo višino vodnih stolpcev, saj drugače pride do neželenih napak na površini vode zaradi sprememb višine terena.

4.4 Transport sedimenta

Suspendiran sediment je potrebno transportirati z vodnim tokom, ki ga opisuje vektor hitrosti $\vec{v}$, zato večina modelov predlaga za prenos sedimenta uporabo enačbe advekcije:

$$\frac{\partial s}{\partial t} + (\vec{v} \cdot \nabla s) = 0 \tag{4.13}$$

Da rešimo advekcijsko enačbo, je potreben Eulerjev korak nazaj v času:

$$s_{t+\Delta t}(x, y) = s_1(x - u \cdot \Delta t, y - v \cdot \Delta t) \tag{4.14}$$

S to rešitvijo nismo bili zadovoljni, saj je pogosto prihajalo do napak. V nekaterih primerih se količina sedimenta ni ohranjala, ker smo prenesli enako količino sedimenta iz iste točke v več različnih točk. Zato smo se odločili, da bomo za transport sedimenta predlagali drugačen model.

V našem modelu transporta sedimenta predpostavljamo, da je suspenzija sedimenta in vode zaradi vrtinčenja vode v povprečju homogena. Ta model se je kasneje v praksi dobro izkazal, čeprav gre le za približek. V realnosti je hitrost sedimentacije odvisna od več različnih parametrov, simulacija pa bi se z uporabo le teh upočasnila.

Transport sedimenta z uporabo naše metode simuliramo tako, da najprej izračunamo količino vode, ki v nekem koraku odteče iz celice, v enakem razmerju pa istočasno odštejemo tudi količino suspendiranega sedimenta. Podobno naredimo tudi s tokovi, ki pridejo v celico, vendar moramo razmerje izračunati za vsakega posebej, saj je koncentracija sedimenta v vsaki celici drugačna. Tako lahko zelo preprosto realiziramo stabilnejšo simulacijo transporta sedimenta.

4.5 Termalna erozija

Zaradi visoke frekvence temperaturnih sprememb, začne kamnina pokati in tako nastane drobir, ki sčasoma tvori melišče, za katerega je značilen konstanten 30° naklon. Termalno erozijo simuliramo tako, da najprej definiramo kot melišča, ki določa ali bomo v določeni celici simulirali termalno erozijo. Ker Moore-ova okolica zagotavlja dosti boljše rezultate, moramo najprej izračunati kote med centralno celico in njenih osem sosedov. Za vsak kot med centralno in sosednjo celico, ki je večji od kota melišča, uporabimo naslednjo formulo:

$$H_{diff}(x, y) = T_e \cdot (|\Delta h| – tan(\alpha)) \tag{4.15}$$

kjer je T_e konstanta, ki določa hitrost erozije, Δh višinska razlika med sosedoma, α kot melišča v radianih, H_{diff} pa je sprememba višine centralne celice.

Potrebno je še dodati, da če je $\Delta h > 0$ je potrebno H_{diff} pomnožiti z -1.

4.6 Izhlapevanje

V zadnjem koraku zmanjšamo nivo vode, da simuliramo izhlapevanje. Med simulacijo predpostavljamo, da je temperatura vode in okolice konstantna, tako lahko zapišemo enačbo za izhlapevanje kot:

$$w_{t+\Delta t}(x, y) = w_2(x, y) \cdot (1 - K_e \cdot \Delta t) \tag{4.16}$$

kjer je K_e konstanta izhlapevanja. S tem korakom smo zaključili celotno iteracijo simulacije.

Poglavje 5

Implementacija in simulacija

5.1 Računanje s pomočjo GPE

Da bi celotna simulacija lahko tekla v realnem času, bi potrebovali dosti večjo računsko moč, kot nam jo lahko ponudijo današnji procesorji. Rešitev je v računanju z uporabo GPE, saj te vsebujejo tudi do več tisoč procesorjev, ki podatke vzporedno preračunavajo le s približno dva do trikrat nižjo frekvenco kot CPE. GPE ima svoj glavni pomnilnik, vanj pa lahko prenaša le določene strukture kot so na primer objekti tipa *VertexBuffer* in *IndexBuffer*, teksture ter program imenovan *senčilnik* (*angl. shader*). Za potrebe numeričnega računanja pride v poštev le prenašanje podatkov preko tekstur, tako da v barvne kanale zapišemo vrednosti spremenljivk namesto vrednosti barv. Ker uporabljamo grafično knjižnico Dx9, smo omejeni na *senčilni model 3* (*angl. shader model 3*), ta pa dovoljuje največjo resolucijo teksture 4096×4096. Pri paralelnem procesiranju je potrebno zagotoviti integriteto podatkov, v našem primeru tekstur, saj do njih dostopa več procesorjev istočasno. Vsak procesor lahko zato piše le v koordinato teksture, ki mu je bila dodeljena, bere pa lahko iz poljubne koordinate. Če bi lahko pisal tudi na ostale koordinate teksture, bi s tem pokvaril rezultate ostalih procesorjev, zato pisanje v ostale koordinate ni mogoče. Za celotno logiko poskrbi senčilnik, ki se izvaja v vseh procesorjih GPE naenkrat. Senčilnik je program, ki teče na GPE in je v osnovi namenjen senčenju fragmentov oziroma pikslov v računalniški grafiki. Za programiranje senčilnikov smo uporabili jezik HLSL, ki je kompatibilen s knjižnico *DirectX*. Senčilniki običajno podpirajo le okrog 100 vgrajenih funkcij [12], ki jih lahko uporabljamo

v senčilnikih. Ker morajo senčilniki teči zelo hitro in ker je v senčilnem modelu 3 število ukazov omejeno, je treba skrbno izbirati operacije.

5.2 Implementacija

Najprej na CPE z algoritmom *diamond–square* generiramo višinsko karto, ki jo shranimo v 2D polje spremenljivk tipa *float*. Za simulacijo moramo najprej pripraviti 5 tekstur, saj ne moremo pisati v tekstruo, ki jo trenutno beremo, in pet senčilnikv. Vsaka tekstura je v našem primeru resolucije 513×513, format pa je tipa *Vector4*, kar pomeni, da imamo na vsaki koordinati štiri 32 bitne kanale za shranjevanje podatkov. Prvo teksturo smo poimenovali *heightWaterSediment-EmptyRenderTarget*. V njej hranimo višino terena, višino vodnega stolpca in količino suspendiranega sedimenta. Zadnji kanal je prazen, ker format *Vector3* ne obstaja. Drugo teksturo smo poimenovali *outflowFluxRenderTarget*, kjer hranimo iztočne tokove proti levi, desni, zgornji in spodnji celici, vsakega v svojem kanalu. Zadnjo teksturo smo poimenovali *velocityVectorRender* target, v kateri hranimo hitrost vodnega toka v smeri x in y, tako sta dva kanala ostala prazna. Čeprav obstaja tudi format *Vector2*, smo uporabili *Vector4*, kamor smo lahko shranili še dodatne podatke za vizualizacijo. Drugi razlog pa je, da če želimo pisati na več različnih tekstur naenkrat, moramo upoštevati pogoj, da so vse teksture istega formata. Četrta in peta tekstura so istega formata kot prva in druga tekstura, namenjeni pa sta pisanju podatkov.

5.3 Simulacija

Za vsak cikel simulacije je potrebno poklicati pet različnih senčilnikov na grafični kartici, ki smo jih napisali za namen simulacije. Senčilnike bomo zaradi lažjega razumevanja opisovali kot *senčilnik1*, *senčilnik2*, ..., *senčilnik5*, teksture pa kot *tekstura1*, *tekstura2*, ..., *tekstura5*.

Pred začetkom simulacije moramo najprej prepisati podatke iz višinske karte v prvi kanal prve teksture.

V prvem koraku na izhod prvega *senčilnika1* nastavimo *teksturo4*, na vhod pa *teksturo1*. Nato poženemo *senčilnik1*, ki prebere količino vode iz *teksture1*, jo glede na tip vodnega izvira (dež ali izvir reke) ustrezno poveča, ter zapiše v *teksturo4*.

V drugem koraku na izhod *senčilnika2* nastavimo *teksturo5*, na vhod pa *teksturo2* in *teksturo4*. Nato poženemo *senčilnik2*, ki za vsako celico na podlagi razlik v višinah vodnih stolpcev izračuna iztočni tok in ga zapiše v *teksturo5*.

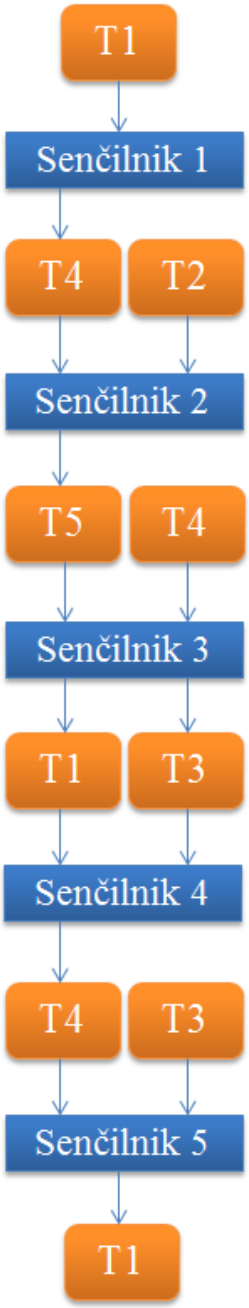
V tretjem koraku na izhod *senčilnika3* nastavimo *teksturo1* in *teksturo3*, na vhod pa *teksturo4* in *teksturo5*. Nato poženemo *senčilnik3*, ki na podlagi prej izračunanih iztočnih tokov izračuna novo višino vodnega stolpca, hitrost vodnega toka ter transportira sediment. Rezultate zapiše v izhodne teksture.

V četrtem koraku na izhod *senčilnika4* nastavimo *teksturo4*, na vhod pa *teksturo1* in *teksturo3*. Nato poženemo *senčilnik4*, ki poskrbi za simulacijo erozije ter odlaganja. Rezultat zapiše v *teksturo4*.

V petem koraku na izhod *senčilnika5* nastavimo *teksturo1*, na vhod pa *teksturo4* in *teksturo3*, nato poženemo *senčilnik5*, ki simulira izhlapevanje vode ter termalno erozijo terena. Rezultate zapiše v *teksturo1*.

Po petem koraku je simulacijski cikel zaključen. Potrebno je le še zamenjati teksturi, ki vsebujeta iztočni tok, tako da *tekstura5* postane *tekstura2* in obratno. To je potrebno zaradi cikličnega menjavanja vhodnih in izhodnih tekstur, pri ostalih menjava ni potrebna, ker se cikel slučajno pravilno izide.

Parametre simulacije smo nastavili kar v senčilnikih in se tako izognili nepotrebnemu prenašanju parametrov na GPE. Pomembno je poudariti, da višine oglišč v objektu *VertexBuffer* med simulacijo ne spreminjamo. Če bi želeli spreminjati parametre objekta *VertexBuffer*, bi bilo potrebno po vsakem koraku objekt prenesti na CPE, ga prebrati v seznam, ga popraviti, seznam shraniti nazaj v objekt in nato objekt shraniti nazaj v glavni pomnilnik GPE. Ker je operacija prenašanja tako velikih objektov iz GPE na CPE časovno zelo zahtevna, v našem primeru uporabimo objekt *VertexBuffer* le za določanje koordinat x in z. Višinsko koordinato y v senčilniku nadomestimo z vrednostjo, ki jo preberemo iz teksture. Ker GPE bere in piše v teksture, ni potrebe, da bi se teksture prenašale iz GPE v CPE.

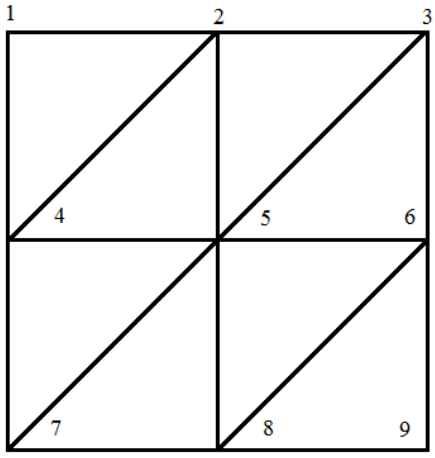


Slika 5.1: Shema prikazuje relacije med teksturami in senčilniki. Med simulacijo ostanejo teksture ves čas na GPE. CPE med simulacijo nastavi za vsak senčilnik le vhodne in izhodne teksture ter ga požene.

Poglavje 6

Vizualizacija

Ker je cilj diplomskega dela vizualizacija simulacije erozije, je potrebno izrisati stanje simulacije po vsakem zaključenem simulacijskem ciklu, ki vsebuje pet korakov, ki smo jih opisali na strani 22. Za vizualizacijo bomo potrebovali le *teksturo1*, kjer imamo shranjeno višinsko karto, vodne nivoje in količino suspendiranega sedimenta. Za ta namen smo napisali dodatni senčilnik, ki na vhod poleg teksture sprejme še dva pomembna objekta imenovana *VertexBuffer* in *IndexBuffer*. V Objektu *VertexBuffer* hranimo za vsako višinsko točko tako imenovano oglišče (*angl. vertex*), ki vsebuje $3D$ koordinate točke v prostoru, ter njihove normale. V objektu *IndexBuffer* pa hranimo zaporedje indeksov, ki dolča katera oglišča v objektu *VertexBuffer* bodo tvorila trikotnik.



Slika 6.1: Prikaz povezav med oglišči.

Številke na sliki predstavljajo oglišča, ki skupaj tvorijo ortogonalno mrežo, s katero bomo predstavili višinsko karto. Objektu *IndexBuffer* posredujemo indekse oglišč, kjer trije zaporedni indeksi kasneje tvorijo trikotnik. V tem primeru bo *IndexBuffer* vseboval naslednje indekse: 1, 4, 2, 4, 2, 5, 2, 3, 5, 5, 3, 6, 4, 5, 7, 7, 5, 8, 5, 6, 8, 8, 6, 9. Indekse bi lahko posredovali tudi na dosti bolj učinkovit način, saj se ogromno oglišč v tem načinu ponavlja. Kljub temu smo se odločili, da bomo za ta projekt obdržali to strukturo, saj z našo strojno opremo ne bomo imeli težav z grafičnim pomnilnikom.

Sedaj lahko zaženemo senčilnik za izris tako, da mu na vhod podamo teskstruo1, *VertexBuffer* in *IndexBuffer*. Senčilnik nato v senčilniku oglišč za vsako oglišče izračuna normalo tako, da najprej izračuna normale vseh 6 prileţnih trikotnikov in nato izračuna njihovo povprečje. V senčilniku pikslov nato s Phongovim modelom senčenja le še osenčimo piksle glede na normalo piksla, ki jo GPE izračuna sama z interpolacijo oglišč trikotnika v katerem se obravnavani piksel nahaja, ter glede na pozicijo treh usmerjenih luči, ki so podane kot $3D$ vektorji. Na tak način prikažemo simulacijo erozije površine terena. Če želimo narisati še vodo, je potrebno v senčilniku oglišč le prišteti višini terena še višino vodnega stolpca. Spremeniti je potrebno še barvo oglišč in sicer je intenziteta modre barve odvisna od korena višine vodnega stolpca pomnoženega z neko konstanto. Zanimiv je tudi prikaz hitrosti vodnega toka, kjer barvo oglišča zamenjamo z dolžino vektorja hitrosti ali pa kar z dejanskim vektorjem, kar nam omogoča vpogled v hitrost vodnega toka tudi po komponentah. Prav tako je zanimiv prikaz transporta sedimenta, kjer barvo oglišča zamenjamo z vrednostjo suspendiranega sedimenta.

Med simulacijo lahko kot kamere premikamo z miško, položaj kamere pa spreminjamo s tipkami W, A, S, D, Q, E. Simulacija teče le takrat, ko držimo tipko *Space*, padavine pa simuliramo s pritiskom na tipko *Shift*. S pritiskom na tipko R prikažemo mrežo terena, s tipko F pa prikažemo pomanjšano $2D$ višinsko karto terena.

Poglavje 7

Proceduralno generiranje poti

Terenu smo želeli dodati še pot, ki bi vodila iz točke A v točko B po čim krajši poti, ki bi bila energetsko čim bolj učinkovita, hkrati pa ne prestrma. Pot smo želeli izoblikovati iz iste mreže, ki predstavlja tudi sam teren. Kmalu smo ugotovili, da je potrebno zaradi te zahteve zgraditi geometrijo ceste z lastnim algoritmom, saj nikjer nismo našli ustreznega algoritma za generiranje ustrezne geometrije. Generiranje geometrije poti ločimo na tri dele. V prvem delu je potrebno najti ustrezno pot iz točke A do točke B za poljubno višinsko karto. V drugem delu je potrebno pot zgladiti, saj iskalni algoritmi najdejo le grobo, nazobčano pot, ki zaradi prehitrih sprememb smeri ni primerna. V zadnjem delu generiramo geometrijo poti na terenu.

7.1 Iskanje poti

Da lahko sploh začnemo z generiranjem poti, je najprej potrebno poiskati pot, ki vodi po terenu tako, da se višina poti čim manj spreminja. Za namen iskanja poti s temi zahtevami, smo uporabili algoritem A^* [13], ki zagotavlja optimalno rešitev, kadar uporabljamo optimistično hevristiko. Hevristika je optimistična, kadar je dejanska vrednost velika vsaj toliko, kot njena hevristična ocena. Število vozlišč, ki jih bo algoritem pregledal, je odvisno od natančnosti optimistične funkcije. V našem primeru je natančnost hevristične ocene večja, kadar smo blizu cilja in obratno, saj smo za hevristično funkcijo izbrali spremembo višin dveh celic. Natančneje, spremembo višin vstavimo v funkcijo M [16], ki opisuje porabo energije za premik enega metra po klancu, kjer je s

tangens naklonskega kota terena v smeri gibanja:

$$M(s) = 0.2635 + 1.737s + 4.237s^2 - 2.143s^3 + 1.493s^4 \tag{7.1}$$

Da iskanje še pohitrimo, definiramo skalirano Moore-ovo okolico, kjer smerne vektorje okolice še pomnožimo s faktorjem k, ter tako pregledujemo dosti manjši prostor. Na primer namesto severnega vozlišča $v(x+1, y)$ uporabimo za iskanje vozlišče $v(x + 1 * k, y)$. Podobno naredimo z ostalimi sedmimi vozlišči. Poleg pohitritve iskanja nam ta način omogoča tudi poljubno aproksimacijo vmesnih točk, saj bi bila drugače pot preveč nazobčana.

7.2 Razdelitev poti

Ko končamo z iskanjem poti, je rezultat le seznam točk, ki so poljubno oddaljene ena od druge, odvisno od velikosti faktorja k, ki definira oddaljenost sosednjih točk v fazi iskanja. Če bi točke med seboj povezali in vmesne točke določili z linearno interpolacijo, bi dobili zelo nerealistično obliko poti, saj pot ne bi imela nobenih krivulj in bi bila sestavljena le iz povezanih daljic. Da bi obliko poti naredili bolj realistično, je bilo potrebno pot ukriviti. Prišli smo do dokaj preproste rešitve s tako imenovano razdelitvijo točk (*angl. subdivision*). Algoritem poteka na naslednji način:

1. Med vsakim parom zaporednih točk v seznamu dodamo novo točko tako, da točki seštejemo in delimo z dve, tako dobimo na daljici med dvema točkama novo točko ravno na polovici daljice. Ta postopek bi lahko imenovali tudi računanje povprečja dveh točk.

2. Položaj vsake točke t, ki je sedaj v seznamu med dvema novo dodanima točkama, spremenimo tako, da najprej izračunamo točko m, nato pa točka t postane povprečje točk t in m. Točko m dobimo tako, da izračunamo povprečje med novo dodanima sosednima točkama.

3. Postopek ponavljamo do poljubne natančnosti.

Po končanem postopku dobimo seznam točk, ki skupaj tvorijo pot. Pot sedaj predstavlja krivulja, ki se delno prilagaja naši poti, predvsem pa zgladi ostre zavoje. Podoben postopek naredimo tudi z višinami teh točk, ki jih hranimo v posebnem seznamu, tako da zgladimo tudi spremembo višin poti.

7.3 Generiranje geometrije poti

Ko imamo izračunane vse točke in njihove višine, lahko začnemo graditi geometrijo poti. Ker pot pogosto zavija, žal ni mogoče generirati geometrije tako, da bi izračunali kar pravokotnico na pot, jo premikali naprej po točkah in z njo generirali teren, ker bi pri ostrih zavojih na zunanji strani izpustili veliko točk, še posebno, če bi bila pot zelo široka. Pot je zato treba najprej razdeliti na zelo majhne segmente, ki so večinoma štirikotniki, v nekaterih primerih pa celo trikotniki. Ko imamo izračunan nek segment, lahko nastavimo vse točke terena, ki padejo v ta segment na višino končne točke segmenta. V primerih, ko več segmentov pokrije isto višinsko točko, se izračuna povprečje vseh višin. Vsak segment izračunamo na naslednji način (postopek velja le za vmesne segmente, saj je za začetek in konec potreben drugačen postopek) :

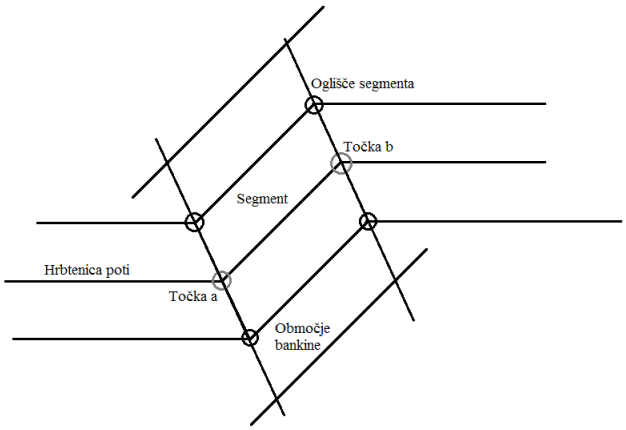
Najprej vzamemo dve zaporedni točki v seznamu poti in ju povežemo, tako da tvorita daljico, ki jo imenujemo hrbtenica segmenta. Sedaj je potrebno zgraditi levo in desno stranico štirikotnika tako, da bo kot med hrbtenico in stranico enak kotu med prejšnjo hrbtenico in to stranico, podobno velja za stranico na desni strani. V primeru, da se stranici sekata, jih je potrebno ločiti, tako konec ene daljice postane konec druge in obratno. Podobno se naredi v primeru, kadar je potrebno zamenjati začetka daljic. Dolžine stranic so odvisne od kota v sklepu dveh hrbtenic ter od širine poti in jih izračunamo na naslednji način:

```
1 Vector3 v1 = nextPoint - currentPoint;
2 v1.Normalize();
3 Vector3 v2 = previousPoint - currentPoint;
4 v2.Normalize();
5 f = (v1-v1).Length() / 2;
6 f = pathWidth / f;
```

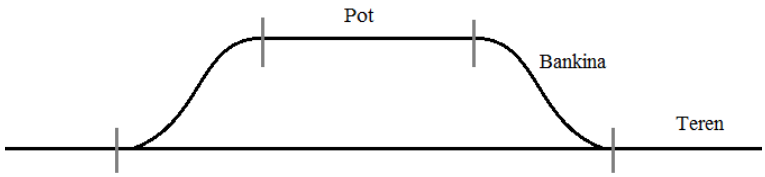
Dolžino normaliziranega vektorja stranice sedaj pomnožimo z faktorjem f. Povezati moramo še zgornja in spodnja dela leve in desne stranice, da dobimo štirikotnik, ki sedaj predstavlja segment. V praksi potrebujemo le štiri točke štirikotnika. Potrebno je še nastaviti višine segmentov tako, da najprej štirikotniku očrtamo pravokotnik, in za vsako točko v štirikotniku z uporabo štirih skalarnih produktov izračunamo ali je točka znotraj štirikotnika. Če to velja, višinsko karto na tej točki nastavimo na višino druge točke hrbtenice tega segmenta.

7.4 Generiranje geometrije bankine

Da se pot na robu lepo zlije s terenom, je potrebno generirati še geometrijo bankine. Območje bankine definiramo podobno kot območje segmenta, uporabimo



Slika 7.1: Skica segmenta poti, ki ga napnemo skozi točki a in b in poljubno širino poti.

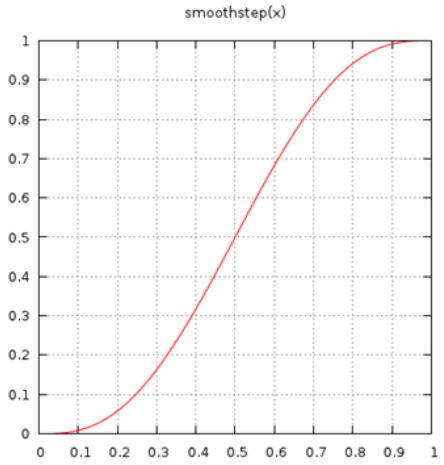


Slika 7.2: Naris ceste. Na skici je prikazan primer, kjer je cesta dvignjena nad ravnino terena.

le večjo širino poti, tako dobimo večji segment, od njega pa odštejemo segment poti. Sedaj je treba višino točk v območju segmenta bankine pravilno prilagoditi, da dobimo čim bolj realistično predstavo. Višino bankine je potrebno prilagoditi tako, da je blizu poti čim bolj podobna višini poti, ko pa se oddaljuje, se čim bolj približuje višinski točki terena, ki je od roba med potjo in bankino oddaljena ravno za širino bankine. Funkcija, ki interpolira vrednost med dvema točkama na ravnokar opisan način se imenuje *smoothstep* [14] :

$$smoothstep(x) = 3x^2 – 2x^3 \tag{7.2}$$

Višino vsake točke v segmentu bankine nastavimo na vrednost, ki jo interpoliramo s pomočjo funkcije *smoothstep*. Za začetno vrednost vstavimo višino roba poti, za končno vrednost pa vstavimo višino točke na terenu, ki je od roba oddaljena za širino bankine. Za interpolacijo je potrebno še vstaviti odstotek



Slika 7.3: Graf funkcije *smoothstep*.

poti, na kateri se nahaja točka, katere višino želimo interpolirati. To preprosto izračunamo tako, da oddaljenost točke od roba poti delimo z širino poti.

7.5 Glajenje poti in bankine

Po končanem generiranju geometrije poti in bankine lahko najdemo nekaj nepravilnosti v geometriji zaradi zaokroževanja točk pri glajenju poti. Te nepravilnosti lahko preprosto odpravimo tako, da višino točke nastavimo na povprečje sosednjih n točk. V povprečje se štejejo le točke, ki so na poti. Podobno naredimo z bankino, le da tokrat upoštevamo točke, ki se nahajajo na področju bankine. Končni rezultat je gladko oblikovana pot. V nekaterih primerih še vedno pride do raznih nepravilnosti, ki pa jih za enkrat dopuščamo, saj gre le za eksperimentalno delo.

Poglavje 8

Rezultati simulacije

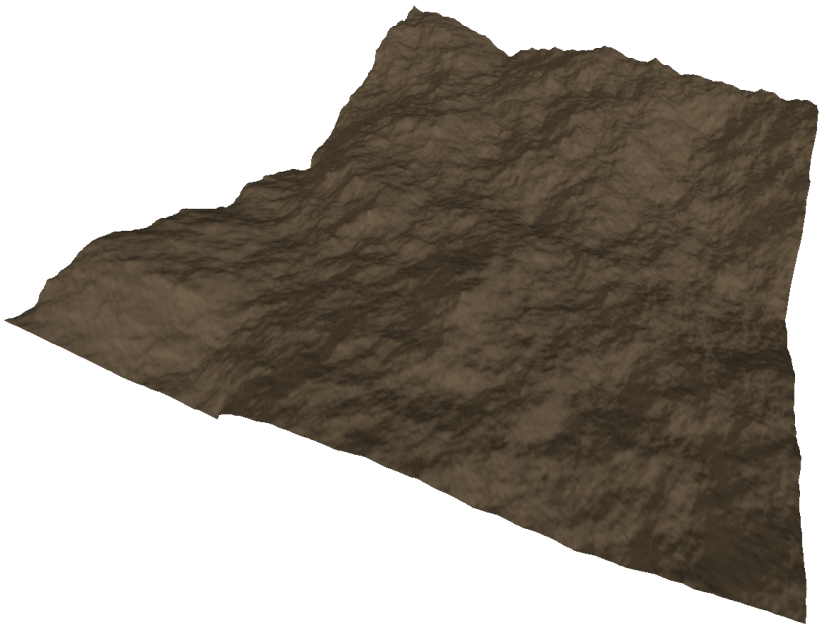
Končni izdelek smo simulirali v realnem času in sicer z 250 sličicami na sekundo, tako da smo poleg hitre simulacije erozije uspešno simulirali še pretakanje plitvih tekočin po terenu, saj je bila hitrost pretakanja vode enaka hitrosti v realnem svetu. Celotna gradnja poti, od iskanja do konstrukcije, je trajala v povprečju približno $100ms$, generiranje terena z algoritmom *diamond-square* pa vedno okoli 25ms. Da smo dobili dokaj realističen izgled terena, je bilo potrebno približno 1500 iteracij simulacije erozije, v našem primeru je to trajalo okrog 6 sekund, za res dober izgled pa je bilo potrebno simulacijo poganjati približno 30 sekund. Če med simulacijo ustavimo pritok vodnih virov in pustimo da voda odteče in izhlapi, dobimo še dodatne podrobnosti na terenu in malenkost drugačen izgled.

Interaktivni nadzor nad padavinami in izviri rek se je obnesel odlično, saj smo tako lahko nadzirali količino padavin, kar je drastično vplivalo na oblikovanost terena. S popravljenim modelom transporta sedimenta smo bili v splošnem zadovoljni. V primerih, ko smo transportirali ogromne količine sedimenta, se je sediment prehitro posedel, zato so včasih vidni polkrožni vzorci dvignjenega terena pri izlivu reke v stoječo vodo. To težavo smo poizkušali rešiti tako, da smo povečali dovoljeno koncentracijo suspendiranega sedimenta, vendar smo problem le preložili, hkrati pa smo izgubili podrobnosti, kot so vijugaste črte v smeri toka na dnu rečne struge.

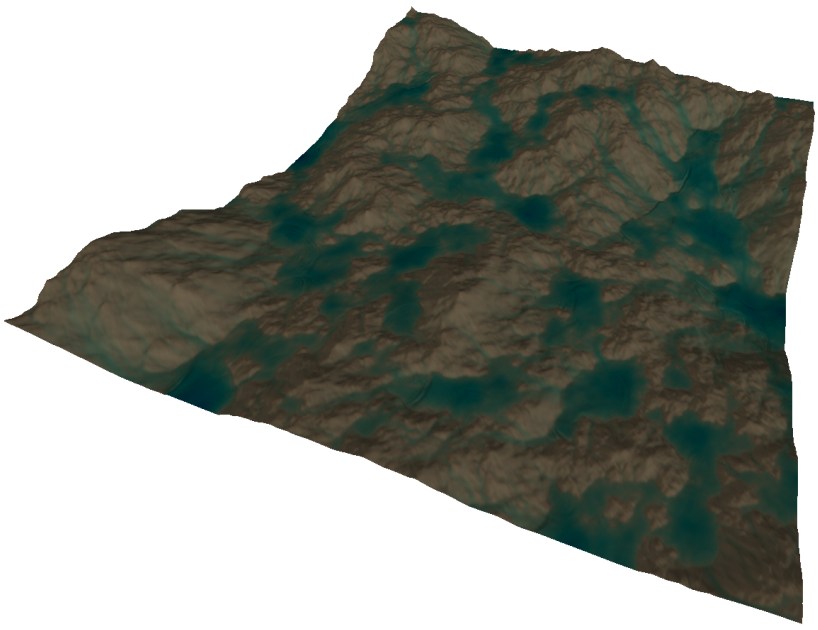
Algoritem ima po našem mnenju dve večji slabosti. Prva slabost je, da algoritem slabo simulira dinamiko vode, ko ta postane malo višja, saj postanejo valovi skoraj navpični in zelo ozki. Druga, slabost pa je, da je izredno težko uravnotežiti vse parametre, da ne pride do nestabilnosti v sistemu. Zaradi tega žal ne moremo poljubno spreminjati parametrov, saj je v matematičnem modelu ogromno nenapisanih omejitev, ki jih lahko najdemo le s poskušanjem.

V končni fazi nam je s skrbno izbranimi parametri uspelo simulirati pretakanje vode z realistično hitrostjo in hkrati simulirati pospešen proces erozije.

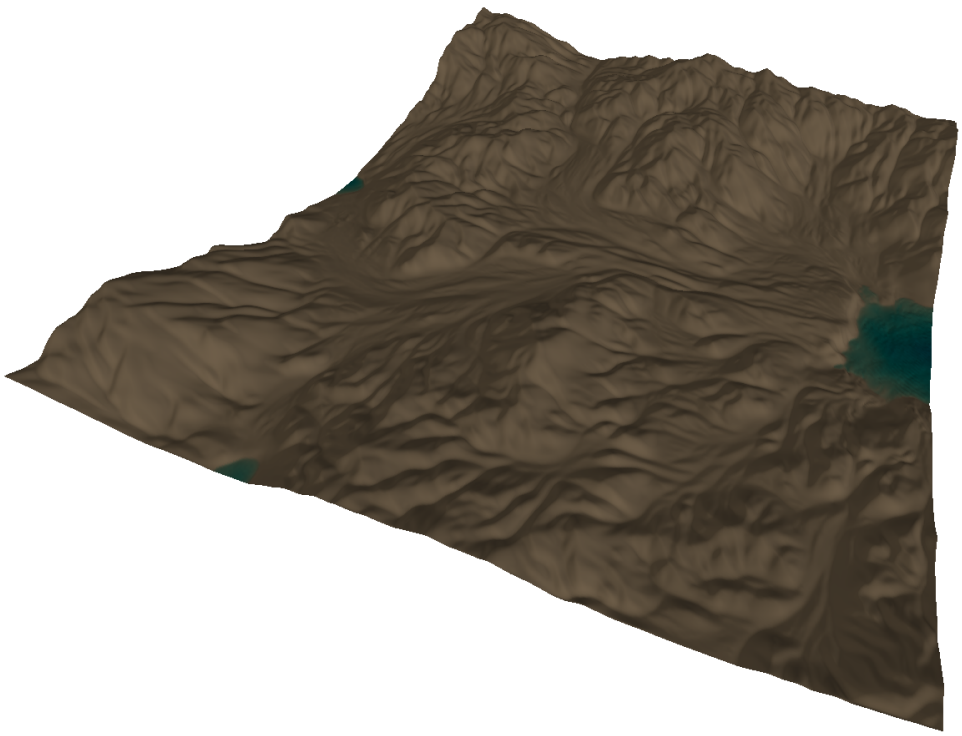
Zasedenost CPE je med izvajanjem simulacije le 4%, saj CPE skrbi le za prenos enostavnih parametrov in za podajanje naslovov tekstur, ki pa so med simulacijo ves čas na GPE. Generiranje bazne geometrije z algoritmom *diamond-square* je edina faza, kjer je CPE polno zasedena, pa še ta traja le okoli $25ms$. Dobra lastnost simulacije je torej zalo mahjna poraba CPE, tako lahko med simulacijo še vedno izkoristimo CPE in izvajamo še druge simulacije ali operacije.



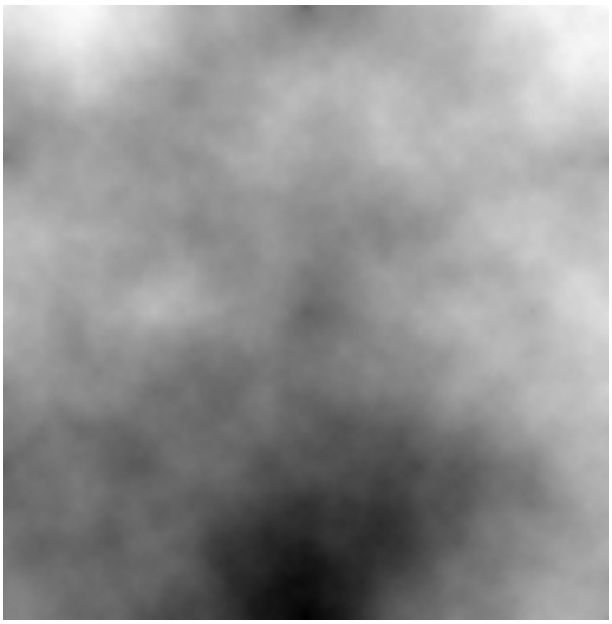
Slika 8.1: Začetna geometrija terena, ki je rezultat algoritma *diamond-suare.*



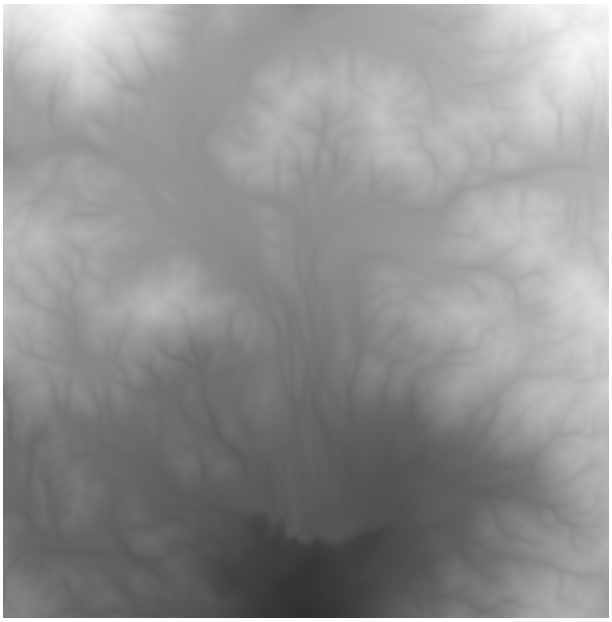
Slika 8.2: Zaradi simulacije deževja začne voda teči po terenu, začne se tudi proces erozije.



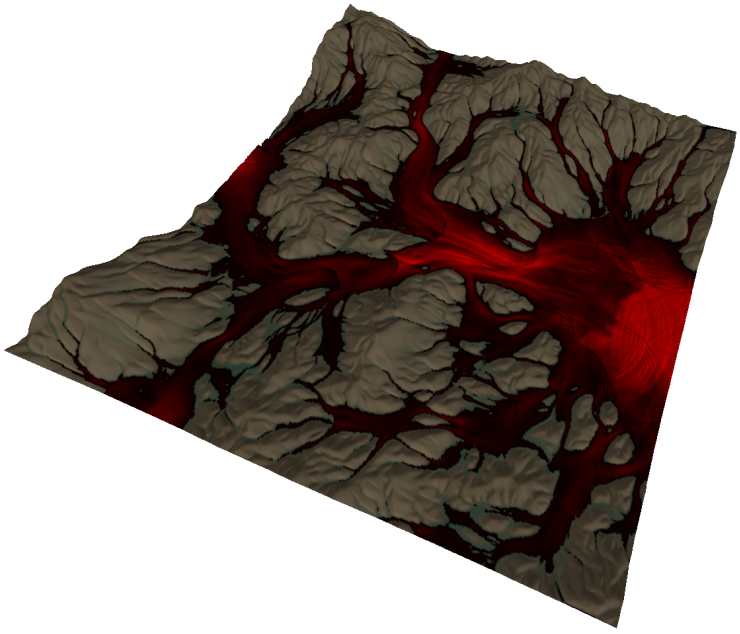
Slika 8.3: Teren po končani simulaciji, voda počasi izhlapeva.



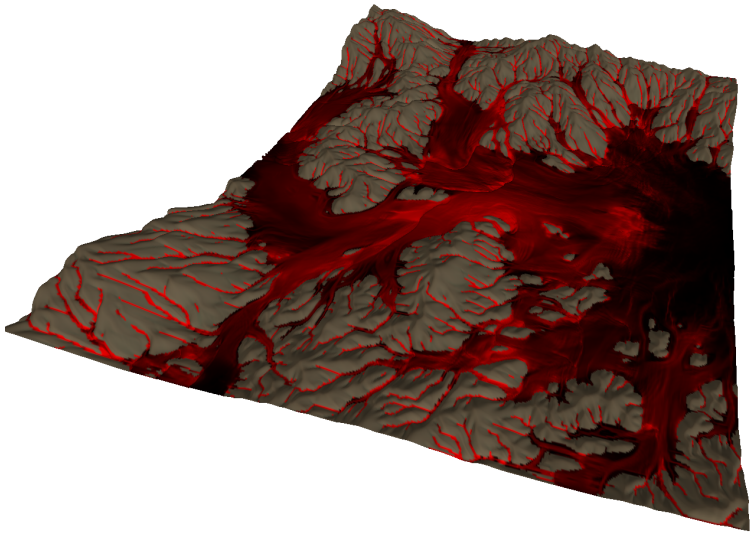
Slika 8.4: Višinska karta terena pred simulacijo.



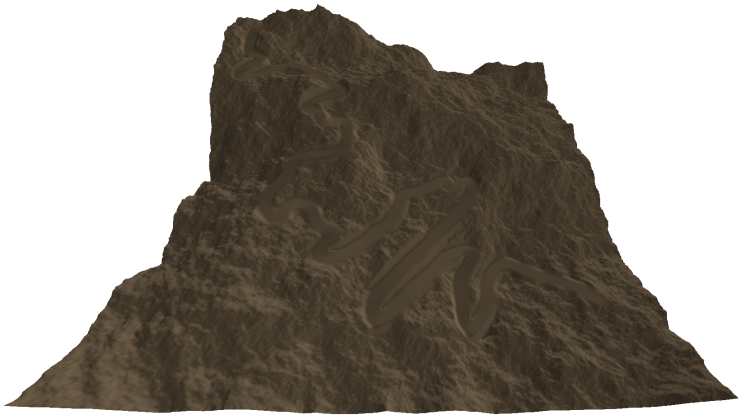
Slika 8.5: Višinska karta terena po simulaciji.



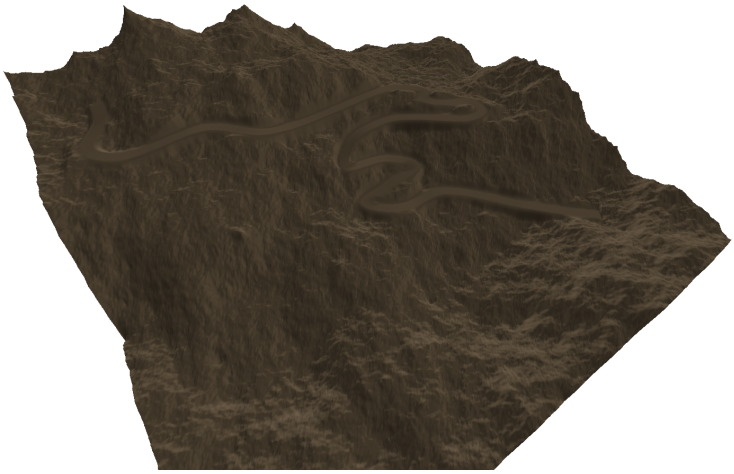
Slika 8.6: Prikaz koncentracije sedimenta, kjer je intenziteta rdeče barve sorazmerna s koncentracijo sedimenta.



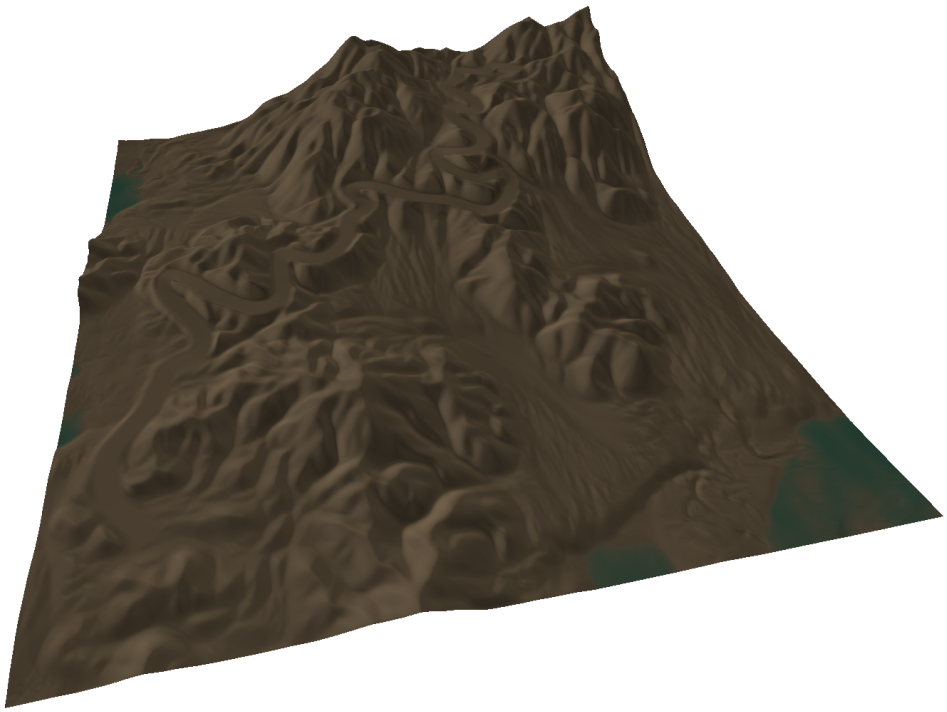
Slika 8.7: Prikaz hitrosti vodnega toka, kjer je intenziteta rdeče barve sorazmerna s hitrostjo vodnega toka.



Slika 8.8: Primer poti, ki vodi iz desnega spodnjega kota v levi zgornji kot. Zaradi prevelike strmine začne pot vijugati, saj je postopen vzpon ključen pri prihranku energije.



Slika 8.9: Še en primer vzpona, kjer algoritem raje izbere daljšo in položnejšo pot, kot pa krajšo in strmejšo.



Slika 8.10: Končni rezultat diplomskega dela, ki prikazuje končan proces erozije na terenu ter pot, ki vodi iz levega spodnjega kota v desni zgornji kot, vmes pa se izogiba večjim višinskim spremembam poti.

Poglavje 9

Sklepne ugotovitve

V diplomskem delu smo generirali naključni teren, na njem izvajali simulacijo pretakanja tekočin ter hidravlično in termalno erozijo v realnem času, na koncu pa smo na teren vstavili še cesto, ki vodi od točke A do točke B po energetsko najučinkovitejši poti. Proceduralno generiranje terena z uporabo algoritma *diamond-square* se je izkazalo kot pravilna odločitev, saj kljub svoji preprosti in intuitivni strukturi v končni fazi generira zelo dobre rezultate. Proceduralno generiranje baze terena bi lahko še izboljšali, če bi uporabili kompleksnejše algoritme, ki bi že v začetku generirali bolj realistično geometrijo, vendar bi generiranje trajalo dlje časa.

Pri simulaciji tekočin in erozije nam je največji problem predstavljala numerična nestabilnost modela, saj je bilo zelo težko izbrati parametre, pri katerih smo minimizirali numerične napake in hkrati dobili realistične rezultate. Ugotovili smo, da bi bilo potrebno postaviti veliko omejitev glede parametrov, če bi želeli spreminjati parametre simulacije, vendar bi za to potrebovali natančno analizo numeričnih podatkov ter njihove odvisnosti. Sam model erozije bi lahko izboljšali tako, da bi dodali več plasti kamnin z različnimi lastnostmi. Glavna pomanjkljivost simulacije erozije s tem modelom se opazi pri večjih strminah, saj je strmo pobočje lahko zelo strmo, simulacija pa ga vidi le kot kratko sosednje območje, prav tako je lahko pri vizualizaciji en trikotnik raztegnjen čez celotno pobočje, kar se hitro opazi kot izguba podrobnosti na strmem terenu. Podoben problem je s hitrostjo vode, saj teče na strmih pobočjih prehitro. Zaradi narave modela žal ne moremo poljubno dodati oglišč na strmih pobočjih, ker mora biti mreža za pravilno simulacijo ortogonalna ter homogena. Edina možna rešitev bi bila, da bi bazni geometriji takoj po njenem generiranju z bikubično interpolacijo povečali resolucijo in nato pognali simulacijo. V tem primeru pa bi simulacija tekla dosti počasneje, saj bi se število točk drastično povečalo. Spre-

menili smo tudi model erozije in odlaganja sedimenta, ker z osnovnim modelom nismo dosegli želenih rezultatov. Odlaganje sedimenta bi želeli kljub popravljenemu modelu še izboljšati, ker se v nekaterih primerih sediment prehitro nabere na bregovih.

Pri implementaciji senčilnikov je bilo treba paziti na količino uporabljenih registrov spremenljivk, ukazov in število vzorčenj, saj smo s številom le teh precej omejeni s strani senčilnega modela 3. Pri simulaciji vode je potrebno izračunati približno 250 korakov simulacije na sekundo, da voda teče z realno hitrostjo. Če bi želeli ohraniti enako hitrost vode pri nižjih frekvencah posodabljanja podatkov, bi morali spremeniti parametre, ki bi lahko poslabšali natančnost simulacije ali pa povzročili numerične nestabilnosti. Z izidom generiranja poti smo zelo zadovoljni, saj v veliki večini primerov deluje odlično, nepravilnosti se pri normalnih parametrih pojavijo le redko, vendar moramo upoštevati določene relacije med parametri. Za nadaljnjo delo je odprtih veliko možnosti, saj smo generirali le geometrijo terena. Zanimivo bi bilo proceduralno generiranje tekstur za skale, zemljo, mah in ostale podrobnosti do poljubne natančnosti. Dodali bi lahko še geometrijo trave, dreves, kamnov in jih simulirali v skladu s fizikalnimi zakoni. Na primer trava bi lahko plapolala v vetru, enako velja za drevesa in ostalo podrastje.

Končna geometrija bi lahko služila kot podlaga za računalniško igro, potrebno bi bilo le dodati teksture in dodatno geometrijo za drevesa, skale in ostale podrobnosti. Generiranje poti bi lahko uporabili v realnih problemih načrtovanja cest, saj bi za izračun potrebovali le višinske podatke določenega terena, dobili pa bi optimalen načrt za gradnjo ceste. Z nekaj dela bi lahko dodali tudi predore in mostove, ki bi še izboljšali načrtovanje cest.

Literatura

[1] Lewis J. P.: "Generalized stochastic subdivision", *ACM Trans. Graph.* 6, 3 , str. 167–190, 1987.

[2] Mandelbrot B. B.: "The Fractal Geometry of Nature", *W.H. Freeman and Company*, 1983.

[3] B. Beneš. "Real-time erosion using shallow water simulation", *VRIPHYS'07: 4th Workshop in Virtual Reality Interactions and Physical Simulation*, 2007.

[4] Mei X., Decaudin P., Hu B.-G.: "Fast hydraulic erosion simulation and visualization on GPU", *Proc. of Pacific Graphics*, st.r 47–56, 2007.

[5] Stava O., Beneš B., Brisbin M., Krivanek J.: "Interactive terrain modeling using hydraulic erosion", *Gross M., James D., (Eds.), Eurographics Association*, str. 201–210, 2008.

[6] Orodje Terragen, november 2013, URL: *http://planetside.co.uk/*

[7] Enačbe Navier-Stokes, november 2013, URL: *http://en.wikipedia.org/wiki/Navier-Stokes_equations*

[8] Enačbe plitvih voda (*angl. Shallow water equations*), november 2013, URL: http://en.wikipedia.org/wiki/Shallow_water_equations

[9] Ken Perlin, "Noise hardware", *Real-Time Shading SIGGRAPH Course Notes*, 2001.

[10] Miller, Gavin S. P., "The Definition and Rendering of Terrain Maps", *SIGGRAPH 1986 Conference Proceedings, Computer Graphics*, 20, 4, avgust 1986.

[11] Algoritem diamond-square, november 2013, URL: *http://www.game programmer.com/fractal.html#diamond*

[12] Intrinzične funkcije jezika HLSL, november 2013, URL: *http://msdn.microsoft.com/en-us/library/windows/desktop/ff471376(v=vs.85).aspx*

[13] Algoritem A*, november 2013, URL: *http://en.wikipedia.org/wiki/A*_search_algorithm*

[14] Funkcija smoothstep, november 2013, URL: *http://en.wikipedia.org/ wiki-/Smoothstep*

[15] O'Brien J. F., Hodgins J. K., "Dynamic simulation of splashing fluids", *Proc. of Computer Animation*, 1995.

[16] M. Llobera in T. Sluckin, "Zigzagging: Theoretical insights on climbing strategies", *J. Theo. Bio*, 249, str. 206-217, 2007