Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tjaž Hrovat

Interaktivna simulacija in deformacija terena v računalniških igrah

diplomsko delo

visokošolski strokovni študijski program prve stopnje računalništvo in informatika

Mentor: doc. dr. Peter Peer
Asistent: as. Bojan Klemenc

Ljubljana 2013

Rezultati diplomskega dela so intelektualna lastnina avtorja in Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavljanje ali izkoriščanje rezultatov diplomskega dela je potrebno pisno soglasje avtorja, Fakultete za računalništvo in informatiko ter mentorja.

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: 00403/2013
Datum: 02.04.2013

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

Kandidat: **TJAŽ HROVAT**

Naslov: **INTERAKTIVNA SIMULACIJA IN DEFORMACIJA TERENA V RAČUNALNIŠKIH IGRAH**

INTERACTIVE SIMULATION AND DEFORMATION OF TERRAIN IN COMPUTER GAMES

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija prve stopnje

Tematika naloge:

Opišite razvoj prikaza terena v računalniških igrah, predstavite tehnike za simulacijo terena, zgradbo pogona, izbiro ustrezne programske in strojne opreme. Nato implementirajte simulator terena, ki temelji na uporabi višinske slike. Z uporabo grafične procesne enote prikažite dinamično osvetlitev, večteksturni preliv ter deformacijo terena v senčilnem programu. Na koncu pojasnite prednosti in slabosti takšne simulacije terena, možne rešitve slednjih in navedite predloge za nadaljnji razvoj.

Mentor:

doc. dr. Peter Peer

Dekan:

prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Tjaž Hrovat, z vpisno številko **63080413**, sem avtor diplomskega dela z naslovom:

Interaktivna simulacija in deformacija terena v računalniških igrah.

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Petra Peera in asistenta Bojana Klemenca,
- 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 28. maja 2013 Podpis avtorja:

Rad bi se zahvalil staršem, ki so mi omogočili študij in me tekom študija spodbujali z veliko mero potrpljenja.

Hvala mentorju doc. dr. Petru Peeru za mentorstvo, prijaznost in korekten odnos med nastajanjem diplomskega dela. Hkrati se zahvaljujem asistentu Bojanu Klemencu za vse nasvete in velikodušno pomoč pri izdelavi diplomske naloge.

Zahvaljujem se tudi vsem asistentom in profesorjem pri odstiranju novih pogledov v računalništvu.

Kazalo

Kazalo slik

Kazalo izsekov kode

Seznam uporabljenih kratic in simbolov

2D – dvodimenzionalno.

3D – trodimenzionalno.

AI – angl. Artificial Intelligance – umetna inteligenca.

API – angl. Application Programming Interface – programski vmesnik.

DDHM – angl. Dynamically Displaced Hight Map – predstavlja mapo oziroma dvodimenzionalen niz podatkov, v katerem so shranjeni podatki o odmikih višine.

FBO – angl. Frame Buffer Object – omogoča izris scene v teksturo namesto na zaslon.

FPS – angl. Frames per Second – število slik na sekundo.

GLSL – angl. OpenGL Shading Language – visokonivojski senčilni jezik, ki temelji na sintaksi, podobni programskemu jeziku C, in je ustvarjen z namenom kontrole cevovoda na grafični kartici.

GPU – angl. Graphic Processing Unit – grafična procesna enota.

LOD – angl. Level of Detail – omogoča reduciranje kompleksnosti 3D-objektov na sceni v odvisnosti od razdalje od opazovalca.

MIP – angl. Much in Little – kratica, ki pomeni veliko v malem. Uporablja se v računalniški grafiki skupaj s preslikavo (mip-preslikava), s čimer je poimenovana optimizirana kolekcija slik različnih velikosti.

PBO – angl. Pixel Buffer Object – omogoča hiter prenos pikslov iz pomnilnika grafične kartice v glavni pomnilnik in manipulacijo z njimi.

Piksel – angl. Picture element – slikovni element.

Teksel – angl. Texture element – temeljna enota v prostoru teksture, ki se uporablja v računalniški grafiki.

VBO – angl. Vertex Buffer Object – omogoča prenos podatkov iz glavnega pomnilnika, hranjenje in delo s podatki v pomnilniku grafične kartice.

VRAM – angl. Video RAM – termin, s katerim je poimenovan specializiran pomnilnik na grafični kartici.

VTF – angl. Vertex Texture Fetch – omogoča branje podatkov v ogliščnem delu senčilnika, na podoben način kakor v fragmentnem delu. Uporablja se za različne učinke, kot so odmiki na terenu, simulacija tekočin in vode, eksplozije itd.

Povzetek

Večina današnjih 3D-iger temelji na statičnem virtualnem okolju, ki ga ni moč poljubno spreminjati. Zato so okolja, v katerih ima igralec več svobode pri interakciji s terenom, vedno bolj zaželena v 3D-igrah. Takšno okolje igralcu omogoči, da je lahko bolj ustvarjalen pri igranju, ker ima praktično neomejene možnosti, in ima od tega tudi bolj bogato izkušnjo.

V sklopu diplomske naloge je najprej opisan razvoj prikaza terena v računalniških igrah, zatem so predstavljene tehnike za simulacijo terena, zgradba pogona, izbira ustrezne programske in strojne opreme. Glavni cilj v projektu je simulacija terena, ki temelji na uporabi višinske slike, v kateri so shranjeni podatki o odmikih višine. Z uporabo grafične procesne enote smo prikazali dinamično osvetlitev, večteksturni preliv s kombinacijo podrobnih in osnovno barvnih tekstur (angl. diffuse texture) ter deformacijo terena v senčilnem programu. Na koncu so pojasnjene še prednosti in slabosti takšne simulacije terena, možne rešitve in predlogi za nadaljnji razvoj.

Ključne besede:

C/C++, izris terena, OpenGL, interaktivna simulacija terena, LOD, GLSL, večteksturni preliv, deformacija terena, GPU.

Abstract

Most of today's 3D games are based on static virtual environment, which does not allow player to make arbitrary changes on terrain. Therefore, the environment in which the player has more freedom to interact with the terrain are becoming increasingly more desirable in 3D games. Dynamic environment allows the player to be more creative, with practically unlimited possibilities and gives a richer gameplay experience.

In the first part of the thesis we briefly describe the history of terrain development in computer games and present some basic techniques for simulating terrain (presentation, creation and modification) and look into basic hardware and software requirements. The main objective of the project is to make a program for dynamic simulation of terrain deformation (displacement) based on hightmap image. Dynamic lighting, multiple texture splat (combination of detail and diffuse textures) and dynamic terrain deformation are implemented on the GPU using shader programs. In the conclusions we give the pros and cons of such terrain simulation, possible solutions for problems of this type of simulation and suggestions for further development.

Keywords:

C/C++, terrain rendering, OpenGL, interactive terrain simulation, LOD, GLSL, texture splatting, mesh deformation, GPU.

Poglavje 1

Uvod

Svet v današnjih 3D-igrah je v celoti zgrajen iz množice poligonov, ki predstavljajo skelet objektom v igri. V večini primerov so to statični svetovi, ki bolj kot ne odražajo omejeno igralno izkušnjo, ki pomeni, da teren in večina objektov v igri ostajajo med igranjem nespremenjeni. Takšen pristop igralcu ne omogoča, da bi imel neposreden vpliv na okolico, in je uporaben predvsem za filmske scenarije v igri, s katerimi želijo ponudniki iger v največji meri nadzorovati potek dogodkov v igri. Hkrati tudi ne omogoča svobode igralcu, da bi po lastni želji spreminjal svet v igri, kar bi lahko spremenilo potek dogodkov v igri ali ga spodbujalo k ustvarjalnosti in želji po ponovnemu igranju. Zahteve po igrah, ki nudijo drugačno in bolj dinamično igralno izkušnjo, so danes veliko večje kot kadar koli prej, ob tem pa je tudi samoumeven kvaliteten izris grafike v igri. Računska moč v grafičnih karticah se je v zadnjem desetletju močno povečala. Velik napredek je dosežen tudi na področju programibilnih grafičnih cevovodov, ki je sprožil nove inovacije v igrah in pristope za deformacijo ter manipulacijo s terenom v 3D-igrah. Igre, ki omogočajo deformacijo terena, so še vedno relativno novo področje. Igralcu želijo zagotoviti, kolikor se le da, dinamično okolje in izris visoko kvalitetne scene. Izris kvalitetne scene je lahko precej zahteven, ker je treba izdelati visokoločljive teksture, točkovno osvetlitev (angl. per-pixel lighting), sence in vse skupaj je treba prilagoditi z dinamiko terena.

Cilj v diplomski nalogi je prikazati simulacijo in dinamično deformacijo terena v realnem času z uporabo grafične procesne enote (GPU) na podoben način, kot je izvedena v računalniških igrah. Simulacija terena temelji na uporabi sivinske slike, katere vrednosti predstavljajo odmike y-komponente na terenu. Odmike v teksturi lahko dinamično spreminjamo z uporabo programirljivega cevovoda, s katerim lahko izvedemo simulacijo različnih oblik na terenu. Skupaj z uporabo večstopenjskega izrisa v teksturo pa lahko izvedemo tudi dinamično deformacijo v realnem času.

Na začetku je predstavljena kratka zgodovina razvoja in deformacije terena v računalniških igrah. Nato sledi predstavitev različnih tehnik (Poglavje 2), ki so pogosto uporabljene pri implementaciji terena.

V poglavju 3 sta predstavljena prototip pogona ter izbira ciljne strojne in programske opreme, s katero bo razvita simulacija. Pogledali smo si in izbrali različne programske jezike ter grafične programske vmesnike, ki so uporabljeni pri razvoju simulacije, in jih primerjali med seboj.

Zelo pomemben del je implementacija terena, za katero je podrobno predstavljena zgradba mreže terena (Poglavje 5.1). Zgradba terena temelji na kombinaciji višinske slike, v kateri so shranjeni podatki o y-koordinatah, in enotske mreže, ki hrani koordinati x in z. Mreža je sestavljena s tehniko povezanih vozlišč trikotnikov (angl. Tringle strip).

Osvetlitev je zelo pomembna pri vsaki simulaciji virtualnega sveta, ker poudari objekte na sceni, in je nek približek k realni osvetlitvi. Pogledali si bomo tri osnovne tipe osvetlitve (Poglavje 5.2) in opisali način, s katerim lahko izračunamo normale v senčilnem programu na podlagi odmikov, ki jih preberemo iz sivinske slike (Poglavje 5.6). Normale so zelo pomembne pri računanju osvetlitve in jih je mogoče uporabiti na različne načine. Osvetlitev se prilagaja skupaj z dinamiko terena in je v celoti izvedena s pomočjo grafične procesne enote.

Pogledali si bomo tudi način, kako lahko obarvamo teren (Poglavje 5.7) z večteksturnim prelivom in dinamičnimi teksturnimi utežmi po stopnjah, ki so razporejene glede na višino terena, od najnižje pa do najvišje točke, s kom-

binacijo detajlnih in osnovnih barvnih tekstur (angl. diffuse texture). Predstavljen je tudi način za izboljšanje vizualne predstavitve terena na območjih s strmejšim naklonom.

Deformacija terena (Poglavje 5.8) je lahko danes neodvisno izvedena z minimalno uporabo centralne procesne enote, s preusmeritvijo na implementacijo v senčilnem programu in z uporabo grafične procesne enote. V projektu je deformacije terena prikazana v dveh različnih pristopih, ki omogočata izvedbo deformacije v realnem času. Prvi pristop temelji na uporabi tekstur, zaradi česar se porabi tudi precej več pomnilnika na grafični kartici, medtem ko se drugi pristop bolj zanaša na računsko moč z izvedbo izračunov deformacije v senčilnem programu. Oba pristopa sta v celoti implementirana na grafični procesni enoti in se zanašata na večstopenjski izris v teksturo s spreminjanjem odmikov višine.

Kot je pogosto v večini urejevalnikov terena v 3D-igrah, so predstavljena tudi osnovnega orodja, s katerimi lahko oblikujemo teren (Poglavje 5.9), ki so prav tako implementirana v senčilnem programu. Za testiranje različnih oblik deformacije na terenu je predstavljena tehnika usmerjenega žarka (angl. ray casting), ki nam pomaga pri interakciji s terenom. Pri delu z orodji lahko s klikanjem miške po zaslonu v območju perspektivne projekcije izberemo lokacijo na terenu. Z uporabo omenjene tehnike lahko preverimo presek daljice z mrežo terena. Da nam ni treba preverjati trkov z vsakim poligonom na mreži, je na koncu predstavljeno še štiriško drevo [40], ki nam izdatno pohitri iskanje preseka daljice s poligoni na mreži terena. V zaključku so predstavljene še prednosti in slabosti takšne implementacije terena, možne rešitve in predlogi za nadaljnje delo.

Poglavje 2

Kratka zgodovina

Generiranje terena v igrah izhaja še iz časov, ko so bile stopnje v igrah zgrajene samo iz 2D-ploščic (angl. tiles). Med prvimi igrami, ki so omogočale izometrično projekcijo, je bila Segina arkadna igra Zaxxon, ki je izšla leta 1982, in je za izris posamičnih stopenj v igri uporabljala mapo (podlaga oziroma teren v igri) z različnimi barvnimi ploščicami. Kmalu je šel razvoj še korak naprej in je izšla prva 3D-igra Elite, izdana pri Arconsoftu leta 1984. Glavna novost v igri je bil izris objektov z mrežo, ki je omogočila izris preprostih poligonov s pomočjo perspektivne projekcije. Pogon v igri je omogočal tudi potovanje med planeti, ki so bili sproti zgenerirani med potovanjem po virtualnem prostoru. Danes imajo že vse moderne igre polno podporo perspektivni projekciji, vendar osnovna ideja ostaja enaka in ne odstopa veliko od izvirnih v predhodnih igrah [1, 2].

Realno-časovna deformacija terena z isometrično projekcijo je bila prvič predstavljena že v igri Populus leta 1989, kjer je lahko igralec med igranjem uporabljal orodja, ki so omogočala odmike na terenu v obliki dviganja ali spuščanjem terena. Ko je igralec spreminjal obliko terena, so se temu primerno prilagajali tudi elementi v igri [3].

Le nekaj iger je eksperimentiralo z različnimi možnostmi pri implementaciji terena, ki je še vedno v večini primerov 2D (za posamezen x, z imamo samo en y) tudi v 3D-pogonih. Prva igra z novo obliko deformacije terena

je bila Earth 2150, ki je bila razvita pri TopWare Interactive in je izšla leta 2000. Igra je imela teren, sestavljen iz dveh plasti, pri čemer so na prvi (vrhnji) plasti enote v igri lahko kopale po površju terena, s tem pa je bilo mogoče spreminajati topologijo površja, kot na primer gorata območja razdeliti na dva dela. Sovražnikovo bazo je bilo mogoče doseči tudi s kopanjem rovov pod zemljo v obliki labirintov, kar je bilo implementirano na naslednji plasti. Deformacija terena je bila vgrajena tudi v urejevalniku stopenj v obliki orodja za dvigovanje ali spuščanje terena, medtem ko nekaj podobnega znotraj igre ni bilo moč uporabiti. V urejevalniku so bila izdelana tudi orodja, ki so omogočala lepljenje tekstur na površju in vstavljanje 3D-objektov v igro. Trislojne mape (orbita, površje in podzemlje) so bile predstavljene v igri Metal Fatigue (2001). Kot dodatek znotraj igre so bile enote lahko še transportirane na popolnoma ločene mape, kjer je imela vsaka svoje okno v uporabniškem vmesniku. Danes ima že vsaka igra za svoj 3D-pogon vgrajen lasten urejevalnik z vsemi potrebnimi orodji za oblikovanje terena in lepljenje tekstur [4].

Deformacija terena danes že nekaj časa ni nič presenetljivo novega. V zadnjih letih je zelo napredovala grafična strojna oprema, kar je kmalu omogočilo uporabo deformacije tudi v igrah, ki so narejene za konzole, kot na primer trenutno aktualni Xbox 360, Playstation 3 in Nintendo Wii, kar pa je bilo v preteklosti bolj težavno za implementacijo zaradi pomanjkanja računske moči. Razvijalci iger neprestano iščejo nove načine, kako najbolje izkoristiti dano računsko moč pri aktualni strojni opremi. V praksi je vedno nova tehnologija najprej razvita na osebnem računalniku, šele nato pa je prenešena na konzole.

Trenutno je na trgu na voljo že kar nekaj visokoproračunskih iger, ki omogočajo realno-časovno deformacijo terena. To so, na primer, v zadnjem času najbolj odmevni naslovi, v prvi vrsti z realno-časovno vojno strategijo Company Of Heroes, From Dust, serija Battlefield: Bad Company in danes zelo priljubljen Battlefield 3. V Company Of Heroes dinamično spreminjanje terena ni imelo le estetskega pomena, pač pa je igralo pomembno vlago tudi

v strategiji igre, kjer so nastali kraterji lahko omogočili zaklonišče pehoti ali upočasnjevali vozila, kar je doprineslo k večji dinamiki znotraj igre, in sicer kot novosti, ki so jih lahko nato igralci izkoristili za potencialno prednost. V igri je dinamična deformacija terena omogočila, da teren ni nikoli ostal enak kot na začetku, ampak je razplet dogodkov v kombinaciji z igralčevimi odločitvami oblikoval teren v igri na unikaten način. Nekaj dinamike smo dobili tudi v seriji Battlefield: Bad Company in Battlefield 3, v kateri je bila omogočena deformacija terena v obliki kraterjev, vendar le na določenih mestih. Še najbolje je z odmiki predstavljena deformacija terena v igri From Dust (slika 2.1), kjer je glavni cilj igre igranje z danimi naravnimi silami. V igri prevzamemo vlogo višje sile, ki mora rešiti majhno pleme pred popolno katastrofo. Pri tem pa je treba tekmovati z nasprotnimi naravnimi silami, ki obsegajo vpliv gravitacije na teren, katastrofalni izbruh vulkanov, erozijo, potrese itd. Destruktivne sile narave je treba izničiti z različnimi orodji oziroma danimi zmožnostmi, ki omogočajo premikanje zemlje, vode, nastajanje točkov itd. [5]

Izris terena še vedno temelji na mreži, ki je sestavljena iz množice poligonov, in je še vedno najbolj priljubljen način za simulacijo terena v 3D-igrah. Glavna razlika je v vključitvi nove komponente, ki nam omogoči novo dimenzijo v 3D-prostoru, s katero lahko ponazorimo odmike oglišč v višino na mreži terena. Danes je to izvedeno z uporabo sivinske slike, ki vsebuje podatke o odmikih, na kateri lahko preprosto ustvarimo odstopanje oglišč na terenu za neko poljubno vrednost. To pomeni, da lahko odmike v višino na mreži terena ustvarimo tudi z uporabo teksture, v kateri piksle lahko uporabimo tudi kot vrednosti za prikaz odstopanja oglišč na določenem delu terena. S tem pristopom lahko na enostaven ustvarimo gore, pobočja, kraterje, rečne struge ali kakršno koli drugo obliko na terenu. To je mogoče že precej enostavno doseči z naprednimi tehnologijami in knjižnicami, kot sta OpenGL in DirectX, ki lahko s pomočjo preprostih klicev ustvarijo velika, razgibana 3D-okolja. Z razvojem novih tehnologij in z vse večjimi potrebami igračarjev po bolj dinamičnem okolju je deformacija terena v zadnjem času postala spet

Slika 2.1: Deformacija terena v igri From Dust [5]

nekoliko bolj priljubljena. Zato jo proizvajalci iger poskušajo na različne načine vgraditi na kar se da najboljši način na obstoječih platformah. V preteklosti je bilo to področje namenjeno predvsem urejevalnikom stopenj, s katerimi se je lahko ustvarilo nove mape, a še vedno izključno le za statičen svet v igri.

Moderni cevovodi v današnjih grafičnih karticah prinašajo mnogo novosti, med katerimi sta pri generiranju zelo pomembna dostop do večteksturnih enot hkrati in branje podatkov z visko natančnostjo v ogliščnem modelu senčilnika iz teksturnega objekta [6].

Poglavje 3

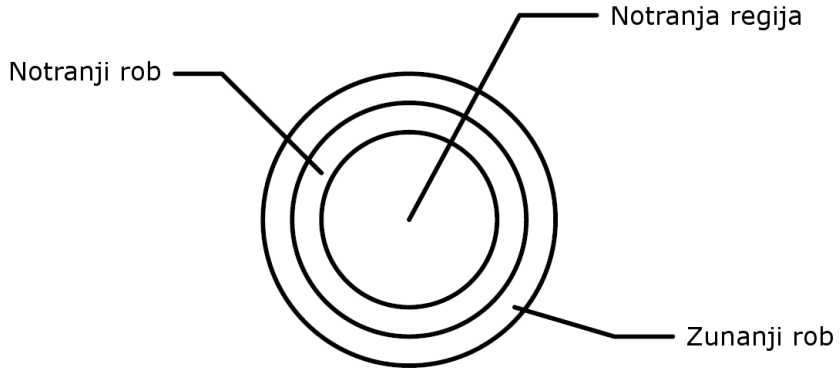
Izris in deformacija terena

Obstaja veliko načinov simulacije terena, ki vključujejo uporabo širokega nabora različnih tehnik za generacijo, optimizacijo in algoritmov LOD (angl. Level of detail). Raziskali bomo nekaj tehnik v povezavi s simulacijo terena in na kratko predstavili vsako posebej.

Ker bomo kot produkt v študiji razvili teren, ki ga bo mogoče deformirati z uporabo grafične procesne enote, si bomo ogledali primer deformacije terena s kraterji. Zelo pomembno področje je tudi vizualna predstavitev terena, za katero si bomo pogledali, kako lahko teren s teksturnim prelivom obarvamo, na koncu pa bomo predstavili še algoritme za optimizacijo terena z LOD.

3.1 Deformacija terena s kraterji

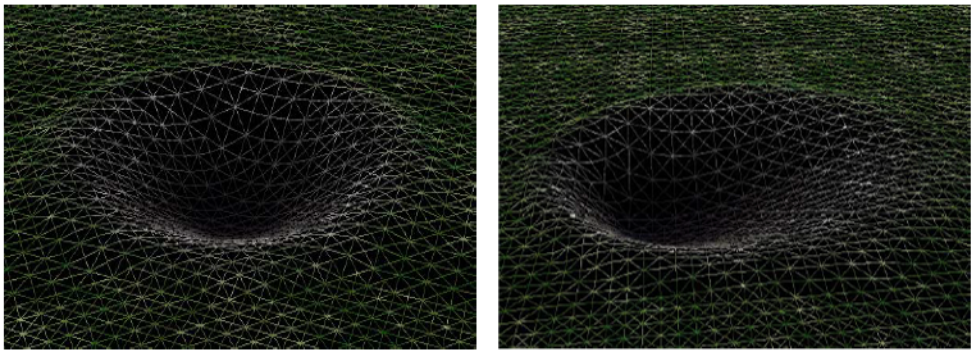
Simulacija dinamičnega terena je v zadnjem času postala vedno bolj pomembno področje v računalniških igrah. Aplikacije, ki lahko uporabljajo dinamičen teren, obsegajo različna področja, kot so 3D-igre, vojni simulatorji, avtomobilska industrija itd. Ko pride do virtualnega bojnega polja, so tehnike, ki nam ustvarijo kraterje ob eksploziji nekega projektila, ključne pri predstavitvi v simulaciji. Večina simulacij temelji predvsem na vizualni podobi in ob eksploziji le izrišejo krater na površino slike. V virtualnem bojnem polju bi eksplozije granat, raket in ostalih eksplozivnih teles spremenile



Slika 3.1: Predstavitev regij v kraterju [10]

topologijo in ključne značilnosti na površini terena z ustvarjanjem kraterjev. Da bi lahko simulirali deformacijo na terenu, moramo imeti prisoten tudi fizikalni model kraterja. Pogledali si bomo tehniko, ki omogoča realno-časovno deformacijo terena s kraterji na virtualnem bojnem polju. Pri tem je zelo pomembna zgradba kraterja, katerega usmeritev je odvisna od smeri eksplozije. Za predstavitev kraterjev na mreži terena se uporablja tehnika DDHM skupaj s teksturo, ki hrani višinske odmike, s katero je mogoče simulirati kraterje na površini terena. Za pohitritev simulacije in za izris obsežnih površin terena je uporabljen algoritem ROAM (angl. Real-time optimally adapting mesh).

Celotno podobo kraterja sestavljajo tri regije: notranja regija, notranji rob in zunanji rob kraterja (slika 3.1). Notranja regija predstavlja konkavni del kraterja, notranji rob pa rob kraterja, ki je podaljšek notranje regije kraterja ter izstopa nad površino terena, zunanji rob pa prikazuje zaključek notranjega roba in predstavlja zunanjo regijo kraterja. V teoriji in tudi v praksi je oblika kraterja definirana glede na material, na katerem je nastala eksplozija. Globina kraterja je, na primer, veliko večja na pesku, kot je recimo na materialu, ki je podoben glini. Za računski opis kraterja se uporabljajo različni parametri. Najpomembnejša sta koeficient radija in globine, poleg tega še sila eksplozije, položaj regij, koeficient razmerja med globino



Slika 3.2: Primerjava med osnovnim in nadgrajenim modelom kraterja [10]

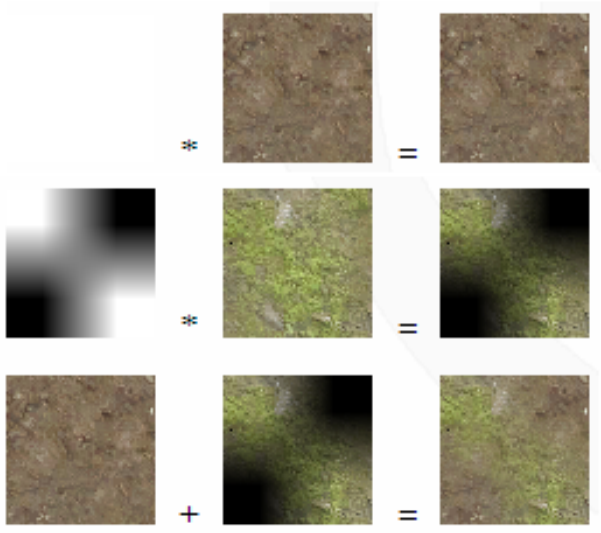
in količino odstopanja roba od površine terena, faktor ukrivljenosti notranje regije itd. S parametri lahko ustvarimo približek dejanski deformaciji, ki jo ustvari krater na realnem bojnem polju ob eksploziji nekega telesa, ki prileti na površino terena. Poleg tega je treba upoštevati še usmerjenost kraterja, ki se lahko spreminja glede na smer, iz katere je priletelo eksplozivno telo. V primeru eksplozije rakete, ki prileti na teren pod različnimi koti, lahko ustvari kraterje z različnimi usmeritvami. Za simulacijo tega je treba nadgraditi osnovni model kraterja. Če pogledamo na krater, ki je nastal pod različnim kotom, z vrha, dobimo obris, ki spominja na obliko elipse (slika 3.2).

Zelo pomembna je tudi vizualna predstavitev območja z oglišči, na katerega je priletel projektil in povzročil nastanek kraterja. Za vizualno predstavitev in izdelavo proceduralne teksture so lahko uporabljene tri različne teksture: oglje, kamenje in trava. Na osnovi teh tekstur se nato določi piksle na kraterju, ki so izračunani kot razdalja med ogliščem in centrom kraterja. Na podlagi te razdalje se potem izračuna vsak individualen piksel na kraterju. Vsi ti deleži se potem skupaj seštejejo in dobimo končno barvo na površini kraterja [10].

3.2 Večplastno teksturiranje

Za izris terena s teksturami obstaja mnogo možnosti, že dolga leta pa sta dva izmed najbolj priljubljenih načinov generacija terena s pomočjo slike, ki vsebuje podatke o odmikih višine, in uporaba maske alfa za mešanje ter kompenzacijo z osnovnimi barvnimi teksturami. Osnovna barvna tekstura predstavlja tip teksture, ki da barvo poligonu, ki se nahaja na površini nekega objekta, in je zelo pomemben faktor pri večplastnem teksturiranju. Zelo znan pristop, ki ga je predstavil Charles Bloom [11], opisuje prehod med teksturami (angl. texture splating) kot mešanje osnovnih barvnih tekstur na površini z uporabo mask alfa. Maske alfa so v bistvu sivinske slike, ki so naložene v enem od kanalov teksture. Običajno so v rdečem, če jih je več, pa tudi v preostalih. Pri mešanju tekstur je maska alfa uporabljena kot označevalec, kolikšen del neke teksture bo uporabljen na neki lokaciji terena. Delež neke teksture se preprosto izračuna kot zmnožek med masko alfa in osnovno barvno teksturo. Če je vrednost teksla v maski alfa 1, se bo ta tekstura prikazala v polni vrednosti, v nasprotnem primeru, ko je vrednost 0, se tekstura sploh ne bo pojavila oziroma ne bomo videli detajlov iz te teksture. Za osnovno barvno teksturo terena lahko izberemo enega izmed najbolj pogostih tipov terena: trava, kamenje, sneg ali kateri koli drug tip. Bloom je opisal osnovno barvno teksturo in njeno masko alfa kot kombiniranje tekstur s pomočjo prosojnosti (angl. texture splat). To pomeni, da je treba površino, na kateri bomo izvedli kombiniranje tekstur, razdeliti na več delov (angl. chunks). Vsak del se ponovi tolikokrat, da je prekrito vse območje na terenu. Za mehak prehod med posameznimi tipi terena poskrbi linearna interpolacija.

V prvotni implementaciji se posamezne teksture ne mešajo med seboj, ampak vsaka tekstura z osnovno barvo predstavlja ločen prehod pri izrisu. To je lahko zelo draga operacija, ker je treba izris scene razbiti na posamezne dele, ki se jih nato pri končnem izrisu na zaslon združi. Takšen pristop je bil včasih obvezen, ker grafične kartice niso imele na voljo dovolj teksturnih enot. Ker pa je tehnologija v zadnjih letih že precej napredovala in imajo



Slika 3.3: Združitev osnovnih barvnih tekstur na podlagi maske alfa [11]

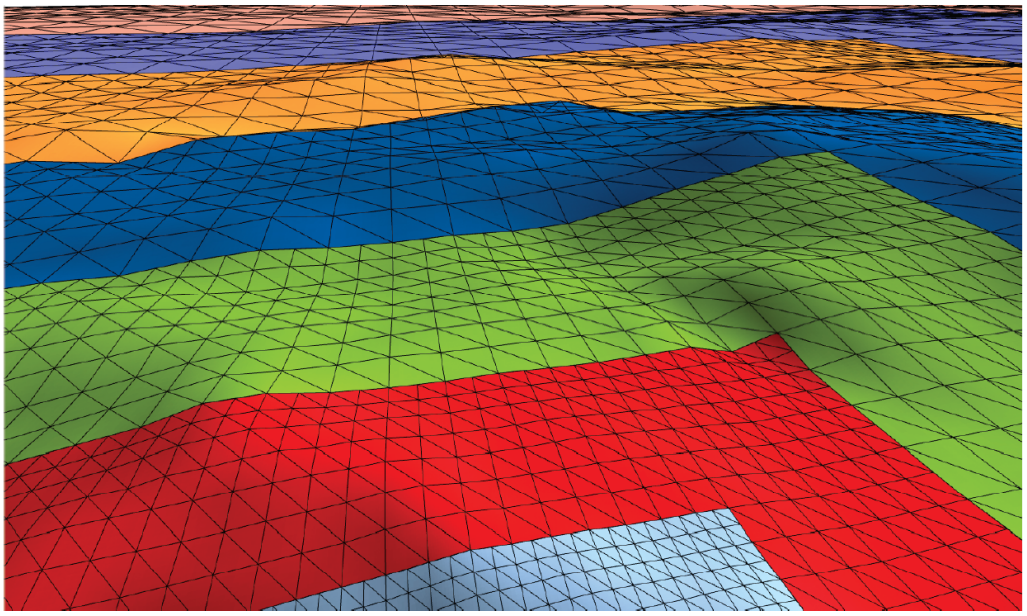
grafične kartice na voljo precej več teksturnih enot kot v preteklosti, nam zato ni več treba delati ločenih obhodov, ampak lahko izris tekstur opravimo v samo enem obhodu [6, 11].

3.3 LOD-algoritmi

LOD-algoritmi imajo zelo pomembno vlogo pri vsaki implementaciji terena, tako v simulaciji kot v igrah, ker omogočajo prikaz širnih območij terena in optimizirajo izris. Pogledali si bomo dve najbolj priljubljeni tehniki, ki sta danes pogosto implementirani tudi v igrah.

3.3.1 Izris terena z uporabo gnezdenih mrež (angl. Geometry clipmap)

Izris terena z uporabo gnezdenih mrež nam omogoči izris mreže terena po stopnjah, na katerih se kvaliteta podatkov prilagaja z oddaljenostjo od gledalca. Tehnika je bila razvita že leta 2004 in sta jo predstavila Lossaso in

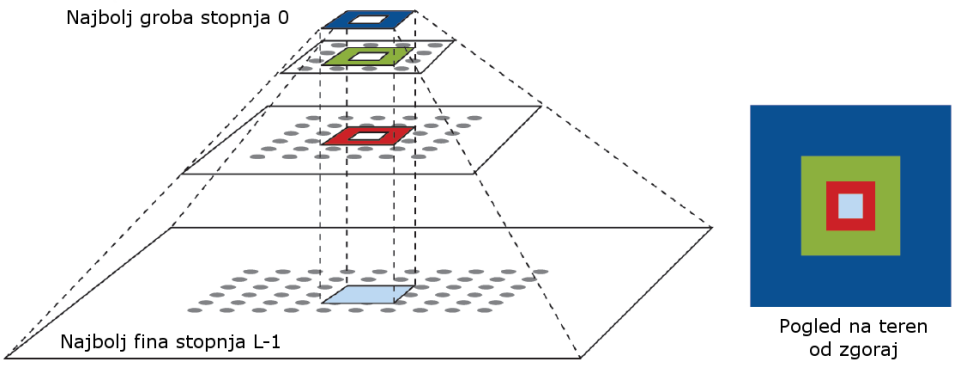


Slika 3.4: Pogled na mrežo terena iz smeri opazovalca [7]

Hoppe [7]. Prvotno je bilo zamišljeno, da se jo implementira na centralnem procesorju (CPU), vendar imamo danes na voljo že bolj zmogljive in napredne grafične kartice in se jo lahko brez težav implementirana tudi na grafični procesni enoti (GPU), kar je tudi veliko bolj učinkovito.

Na začetku je treba v predpomnilnik naložiti seznam gnezdenih mrež, katerih kvaliteta (velikost kvadratov na posamezni stopnji) je pogojena s prirastkom od razdalje in se jih da prilagoditi v odvisnosti od oddaljenosti od opazovalca. Pri obdelavi terena na grafični procesni enoti imamo lahko te mreže shranjene v seznamu tekstur, kar nam omogoči, da lahko tudi izvedemo vse potrebne izračune kar na grafični procesni enoti. Izris terena z uporabo gnezdenih mrež je preprosta tehnika za izdelavo in omogoča prikaz velike količine podatkov na mreži terena pri velikem številu sličic na sekundo.

Tehnika obravnava teren kot dvodimenzionalno višinsko sliko, ki je na začetku filtrirana v obliki pomanjšane izvedbe tekstur (angl. mipmap) na piramidi, ki ima število stopenj L. Pri zelo velikih mrežah terena je lahko izris celotne piramide enostavno prevelik, da bi ga spravili v pomnilnik. Zato



Slika 3.5: Predstavitev piramide po stopnjah glede na kvaliteto podatkov [7]

se na vsaki stopnji piramide naloži v pomnilnik samo vzorce iz mreže terena v obliki oken in $n \times n$ velikosti. Okna so gnezdena eno vrh druge po posameznih stopnjah v piramidi in so centrirana glede na pozicijo gledalca (slika 3.4). Pri tem pa je treba upoštevati, da okna z najbolj grobo stopnjo (spodnji del slike 3.5) zasedejo manj prostora kot tista z najbolj fino (zgornji del slike 3.5). Le najbolj fina so izrisana kot popoln kvadrat mreže. Vsa ostala pa so izrisana kot votel obroč, znotraj katerih so gnezdena le okna z najbolj grobo stopnjo. Ko se gledalec premika, se z njim premaknejo in prilagodijo tudi stopnje na piramidi glede na nove podatke, prejete z mreže terena. Da bi zagotovili čim hitrejše in čim učinkovitejše osveževanje stopenj, je treba na vsaki stopnji piramide do pomnilnika dostopati v obliki naslovnega prostora 2D. Ena izmed prednosti izrisovanja po stopnjah je tudi možnost prilagajanja stopenj v piramidi z dodajanjem ali zmanjšanjem števila preslikav oken z mreže terena.

Originalna implementacija iz leta 2004 shranjuje oglišča po posameznih stopnjah piramide v slikovni pomnilnik, ki je enodimenzionalen. Danes je zelo priljubljena implementacija z uporabo ogliščnih tekstur in tehnike VTF (angl. Vertex Texture Fetch), ki nam omogoča branje podatkov s teksture v ogliščnem modelu senčilnika. Tako je lahko vsaka stopnja v piramidi, shranjena v VRAM-u, predstavljena kot dvodimenzionalna slika z odmiki, kar je

bolj naraven način in nam zelo poenostavi predstavitev terena. V teksturo shranjujemo samo podatke, ki hranijo odmike o višini terena. To pomeni, da je treba pri izdelavi mreže terena oglišča (x, y, z) razbiti na dva dela:

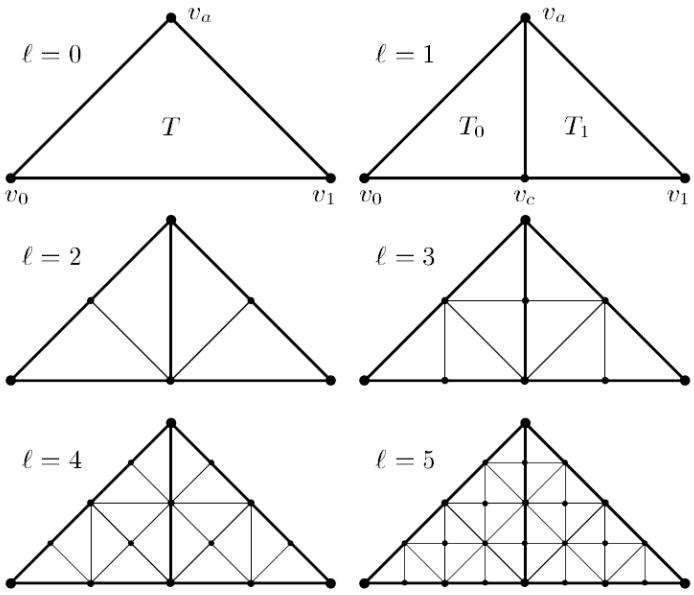
- Koordinati x in z bosta shranjeni v medpomnilniku kot konstanti in bosta ob inicializaciji terena preneseni v ogliščni del senčilnika, kjer nam bosta služili za branje dvodimenzionalne lokacije na mreži terena.
- Koordinata y je shranjena v 2D-teksturi v enem kanalu kot slika, ki nam bo služila za pridobitev podatkov o višini (angl. elevation map). Pred tem pa lahko poljubno določimo dimenzije na $n \times n$ mreži oziroma na teksturi, in sicer za posamezno stopnjo na piramidi. Ti podatki v teksturah se potem med izvajanjem dopolnjujejo, ko se gledalec premika.

Shranjevanje podatkov v obliki odmikov višine v teksturo in zgradba terena kot seznam tekstur nam omogočata neposredno procesiranje s pomočjo grafične procesne enote v delu cevovoda, kjer se izvaja rasterizacija. Poleg tega so lahko vsi potrebni izračuni pri izrisu terena izvedeni s pomočjo grafične procesne enote v realnem času [7].

3.3.2 Real-time Optimally Adapting Mesh (ROAM)

Algoritem ROAM definira mrežo terena kot hierarhično binarno iskalno drevo za izris trikotnikov. V tem drevesu vsako vozlišče predstavlja verzijo trikotnika, ki vsebuje manj podrobnosti kot pa verziji njegovih podvozlišč (podtrikotnikov). Vozlišča z listi predstavljajo najvišje trikotnike v LOD-algoritmu, medtem ko izhodiščna vozlišča predstavljajo najnižje. Celotna procedura postane rekurzivna v trenutku, ko začnemo s prečnim razpolavljanjem vozlišč (trikotnikov) (slika 3.6) in se odločimo, katera vozlišča bodo šla v trenutni izris na zaslon. Ko začnemo testirati posamezna vozlišča, lahko izberemo in označimo trikotnike, ki naj bi šli na izris, ali pa vstopimo še globlje v drevo in naredimo enako testiranje na podvozliščih. Ker je vsako vozlišče

predstavljeno s tremi oglišči, ki sestavljajo trikotnik, lahko 3D-lokacijo v virtualnem svetu najdemo na podlagi razdalje, ki je definirana med trenutnim vozliščem in opazovalcem. Ko imamo izračunananano razdaljo, lahko izvedemo še mnogo bolj kompleksne algoritme pri izbiri vozlišč (trikotnikov), ki bodo šla na izris. Čeprav algoritem ROAM proizvede niz detajlnih stopenj na mreži terena, ki so prilagojene glede na specifično stopnjo trikotnika, se algoritem precej slabše obnese na grafični procesni enoti. Ker lahko grafična procesna enota procesira samo podatke v lokalnem pomnilniku grafične kartice, je treba vsako spremembo iz nabora izrisljivih podatkov prenesti v pomnilnik. Takšnen prenos je vse prej kot poceni in prekomerna uporaba lahko zelo hitro povzroči zastoj v cevovodu, ker mora grafični procesor počakati, da se najprej prenesejo vsi podatki. Za hitrejši izris se raje odločimo za prenos podatkov v pomnilnik grafične kartice (angl. buffer) v času inicializacije terena, v bodoče pa raje poskusimo čim bolj zmanjšati število prenosov med izvajanjem aplikacije. Čeprav ROAM omogoča odličen pristop k vizualizaciji terena, vsebuje tudi elemente, ki so težko izvedljivi znotraj igre. Zato se razvijalci iger raje odločajo za bolj lokalne implementacije algoritma, ki temeljijo na prilagoditvi glede na njihove potrebe pri razvoju igre [8, 9].



Slika 3.6: Rekurzivno razpolavlanje vozlišč po stopnjah (0-5) [9]

Poglavje 4

Načrtovanje in uporaba tehnologij za simulacijo terena

Obstaja mnogo načinov, ki omogočajo izgradnjo 3D-pogona za simulacijo. Glavni namen je razviti pogon, ki ga bo mogoče uporabiti za dinamično simulacijo terena in testiranje različnih oblik deformacije na mreži terena. Simulacija bo namenjena kot demonstracija za vse, ki se ukvarjajo z izdelavo iger, ali za vse zanesenjake, ki jih zanima to področje.

4.1 Prototip pogona

Igralni pogon (angl. Game engine) je sistem, ki olajša izgradnjo računalniških iger. Pogon skrbi za osnovne funkcije igre, kot so upodabljanje, animiranje, podpora okolju, fizika, zvočna podpora, omrežna podpora, podpora posebnim efektom, uporabniški vmesnik in podpora za vhodne naprave.

Za upodabljanje se v osnovi sistemi ločijo na 2D- in 3D-sisteme. Upodabljanje je proces generiranja slike na zaslonu iz računalniško ustvarjenih modelov. Modeli so pogosto povezani v scenske strukture. Upodabljevalnik mora skrbeti za prikaz geometrije, teksture, osvetlitve in senčenja scene. V osnovi pa poskrbi, da razvijalcem iger ni treba skrbeti, kako se bodo elementi igre prikazovali na zaslonu. Za pogone, ki podpirajo 3D-upodabljanje,

je pogosto treba modele uvoziti iz programov za 3D-modeliranje.

Animacija v igrah pomeni premikanje elementov v virtualnem svetu, kar daje občutek, da svet živi. V osnovi gre za cikle vnaprej pripravljenih animacij, ki se ob določenih pogojih predvajajo.

Pogoni skrbijo tudi za emuliranje okolja. K okolju spada kreiranje tal, kar je pogosto doseženo s pomočjo različnih map, preko katerih pogon generira podatke o višinah in teksturah za določen del tal.

Za detekcijo trkov skrbi fizika v igrah, ki izračunava premikanje predmetov po virtualnem svetu preko fizikalnih zakonov. Običajno pa gre v pogonih zgolj za simulacijo fizike do takšne mere, da deluje navidezno realno. V igralnih pogonih pa je pomemben tudi mehanizem trka, ki preračunava razdalje med elementi v virtualnem svetu in razvijalcem ponuja programerski prostor, da se odzovejo na določene trke elementov.

Podsistem za zvok v pogonu ja zbirka funkcij, ki programerjem olajšajo delo z zvočnimi datotekami. Podsistem olajša programerju predvajanje zvokov v igrah. Naprednejši sistemi ponujajo tudi funkcije, ki povezujejo zvoke z objekti v virtualnem svetu in tako predvajajo zvoke na različnih razdaljah.

Podpora za omrežje v igralnem pogonu omogoča programerjem enostavno prilagoditev svojih iger za igranje v omrežnih skupnostih.

Sestavni del vseh igralnih pogonov so tudi posebni učinki (angl. special effects). Posebni učinki so lahko izvedeni izključno v senčilnem programu s postprocesiranjem slike ali s "sistemom delcev" (angl. particle system). To je sistem, ki vsebuje neviden element v virtualnem svetu, iz katerega izhajajo delci in tako kreirajo animacijo oz. poseben efekt. Delci, ki izhajajo iz začetnega elementa, so večinoma sličice stopenj animacije, ki se izmenjujejo in tako tvorijo iluzijo efekta.

Za osnovno navigacijo v igrah so izdelani uporabniški vmesniki, ki predstavljajo vizualni vmesnik (torej meniji), in tudi deli, ki prikazujejo statuse igralcu (npr. števec hitrosti). Ti deli iger se prikazujejo ločeno od virtualnega sveta in so že predpripravljeni v modernih pogonih.

Pri programiranju si programerji pomagajo z razvojnimi paketi, ki so

zbirke programja, s pomočjo katerih se olajša razvoj iger. Mnogi moderni igralni pogoni ponujajo svoje razvojne pakete, ki pa se razlikujejo po številu funkcionalnosti. V osnovi pa vsi paketi vsebujejo razvojno okolje za pisanje programske kode in nadzor nad viri (slikami, zvoki, 3D-modeli ...). Pogosto pa ponujajo tudi urejevalnike virtualnih svetov in urejevalnike posebnih efektov. V programskih paketih so tudi programske knjižnice, ki jih moramo povezati s programom.

Knjižnica (angl. library) oziroma programska knjižnica je v računalništvu zbirka podprogramov (oziroma funkcij) za pomoč pri izdelavi oziroma razvoju programske opreme [12]. Izbira knjižnice je odvisna od potrebe programerja in operacijskega sistema, za katerega izdelujemo aplikacijo. Pri programiranju se največkrat srečamo z dvema tipoma knjižnic:

- Statično povezane knjižnice (poimenovane tudi arhivske); med povezovanjem (angl. linking) jih povežemo v naš program in postanejo del našega programa. Ko prevedemo program, ki vsebuje statično povezane knjižnice, postanejo vse funkcionalnosti, ki jo vsebuje ta knjižnica, del naše zagonske datoteke aplikacije [13].

- Dinamično povezane knjižnice (ali deljene knjižnice); njihova začilnost je, da so naložene v našo aplikacijo med samim izvajanjem programa. Ko se program prevede skupaj z dinamično knjižnico, za razliko od statičnih, ta knjižnica ne postane del naše zagonske datoteke, ampak ostane ločena oziroma neodvisno shranjena od kode preostalega programa [13].

Ena najbolj pomembnih stvari v našem programu je gotovo izgradnja glavne zanke. Kot je bilo že rečeno na začetku, vsak pogon za igre temelji na osveževanju stanja v igri, izrisa, efektov glasbe, umetna inteligenca (angl. artificial intelligence) itd. Vse to je pod nadzorom glavne zanke programa [14, 6]. Ker nas zanimata le dve funkciji od zgoraj navedenih, in sicer osveževanje stanja in izrisa, se bomo osredotočili predvsem na ti dve.

Obstaja ogromno igralnih pogonov in vsak pogon ima svoje funkcional-

nosti, a vendar se večina funkcionalnosti ponavlja med vsemi pogoni. Ker bomo razvili prototip pogona za simulacijo, bo vključeval le glavne funkcionalnosti s pogoni za igre, kot so inicializacija programa, izgradnja glavne zanke, izgradnja tal ter zaključitev in izhod iz programa.

4.2 Strojna oprema

Podpora za strojno opremo, na kateri bomo poganjali simulacijo, naj bi spadala nekje med manj ter srednje zmogljivo in bi nudila podporo vsaj za senčilni model, verzije #130 v GLSL. Strojna oprema bo podprta na naslednji specifikaciji ali celo boljša:

- CPU: dvo- ali večjedrni; pri frekvenci 2000 Mhz; arhitektura x86
- GPU: AMD Radeon HD 3000 / Nvidia Geforce 8000
- Glavni pomnilnik: 1 Gb

Vse ostale specifikacije strojne opreme, ki nimajo podpore za senčilni model s podobno funkcionalnostjo, kot jo ima GLSL v verziji #130 ali več, bodo izključene iz te simulacije. Ker je cilj raziskati tudi tehnologijo, ki jo ponuja današnja grafična strojna oprema, si bomo pogledali tudi trenutno najbolj priljubljene metode dela z grafičnimi pospeševalniki. Ker hočemo velik del aplikacije implementirati v senčilnem programu, bo tudi simulacija usmerjena na grafično procesno enoto.

4.3 Operacijski sistem

Danes je na voljo že veliko operacijskih sistemov, ki podpirajo 3D-grafiko, vključno z Microsoft Windows, na katerem sta podprta kar dva grafična API-ja (angl. Application programming interface), DirectX in OpenGL, Mac OS, Linux, platformami Java in operacijskimi sistemi na mobilnih platformah, ki podpirajo OpenGL. Pri tem je zelo pomembno imeti na voljo razvojno okolje,

ki ima dobro podporo in je hkrati široko dostopno. Windows izgleda kot očitna izbira, ker ima podporo za več kot le eno grafično knjižnico, poleg pa ima tudi odlično razvojno okolje Visual Studio in je od vseh še najbolj široko uporaben sistem. Velika večina ciljnih uporabnikov uporablja Windows, ne pa kateri koli drug operacijski sistem, zato je tudi razvoj za to platformo precej bolj smiselen. V končni meri bo razvoj za izbrani operacijski sistem temeljil predvsem na programski opremi, izbranem orodju in široki izbiri virov.

4.4 Programski jezik

Na izbrani platformi bomo potrebovali visokonivojski programski jezik. Visokonivojski programski jezik je v primerjavi z ostalimi nizkonivojskimi jeziki veliko bolj prenosljiv in lahko deluje na različnih platformah z nekaj prilagoditvami v kodi. V tem primeru se bomo odločali med jeziki, ki so najbolj razširjeni na platformi Windows:

- C# je visokonivojski programski jezik, ki so ga razvili pri Microsoftu z namenom združitve računske moči in enostavnosti uporabe. C# temelji na jeziku C++ in je po sintaksi zelo podoben Javi. Prilagojen je za delo na Microsoftovi platformi .Net in ima izdelan sistem za avtomatično upravljanje s pomnilnikom (angl. garbage collector). Poleg tega pa omogoča razvijalcem aplikacij, da lahko razvijejo visoko prenosljive aplikacije. Ker programerji lahko še naprej razvijajo na obstoječi kodi brez dopolnjevanja, je pričakovano, da je programiranje s C# hitrejše in bolj ekonomično. Za grafične programerje je na Microsoftovi spletni strani na voljo tudi posebna dokumentacija za razvoj iger skupaj z uporabo tega programskega jezika, v kombinaciji z njihovo grafično knjižnico DirectX in knjižnico za razvoj iger XNA, kar nam da dobro orodje za izdelavo iger. Slabost tega jezika je v tem, da je uporaben le na platformi Windows skupaj z ogrodjem .Net in je v primerjavi s C++ precej počasnejši [15].

- C/C++ je najbolj pogosto uporabljen programski jezik pri razvoju iger in je podprt na vseh večjih platform, vključno s konzolami in PC-ji, in ima na voljo zelo veliko količino različnih virov z vseh področij programiranja. Nudi zelo dober nivo dostopa do pomnilnika, kar mu omogoča učinkovito upravljanje s podatki, in se prevede naravnost v strojno kodo, kar mu da na podlagi testov status enega izmed najhitrejših jezikov na svetu. Zaradi svoje starosti in priljubljenosti ima tudi napisano zelo veliko število različnih knjižnic s področja računalniške grafike. Ker je C++ nadgradnja programskega jezika C, pomeni, da je z njim tudi navzgor združljiv, kar nam omogoča, da lahko pišemo in prevajamo njegovo kodo, posledično pa tudi vse njegove knjižnice. Združljivost s C-jem je po drugi strani tudi njegova slabost, ker je C++ do neke mere potem tudi nizkonivojski programski jezik in zna biti zelo kompleksen pri razvoju aplikacij, ker, na primer, zahteva, da se na vsakem operacijskem sistemu na svoj način implementira postavitev oken. Težavno pa je lahko tudi upravljanje z viri in pomnilnikom, ki hitro dopustita prostor za napake. Težja prenosljivost kode pomeni slabost v primerjavi z drugimi programskimi jeziki, kot je Java [16, 17].

- Java je objektno usmerjen programski jezik, podoben C++, ampak veliko bolj poenostavljen, da odpravi jezikovne pomanjkljivosti, ki so povzročale najbolj pogoste programske napake in pomanjkljivosti pri nizkonivojskih programskih jezikih. Programska koda v Javi se ne prevaja v strojno kodo, temveč v vmesno kodo (bytecode). Ta koda se na običajnih procesorjih ne more izvajati, ker potrebuje tolmača (interpreter). Javino izvajalno okolje, poimenovano tudi javanski navidezni stroj (JVM, Java Virtual Machine), je postalo mnogo več kot tolmač vmesne kode, ki omogoča programom, da se izvajajo na različnih strojnih podlagah. JVM ponuja nadzorovano izvajalno okolje in hkrati veliko stopnjo varnosti v primeru prekoračitve pomnilnika. Java ima prav tako podporo za OpenGL preko ovojnih knjižnic JOGL ali LWJGL, ki omogočata uporabo funkcij OpenGL v naši aplikaciji. Številne predno-

sti nadzorovanega okolja imajo seveda svojo ceno [18].

Ker nas zanimajo predvsem prenosljivost kode, fleksibilnost ravnanja s pomnilnikom in hitrost, je C++ še najbolj primerna izbira. C++ ima tudi odlično podporo na vseh ostalih platformah, poleg tega pa ima že na voljo ogromno bazo virov na temo grafičnega programiranja.

4.5 Grafični API in tehnologije

V tem delu si bomo izbrali knjižnice, ki jih želimo uporabiti pri izdelavi 3D-simulacije terena za naš projekt.

Na platformi Windows imamo na voljo dva programska vmesnika za grafiko na izbiro: Direct3D in OpenGL. Oba vmesnika imata skoraj povsem identično funkcionalnost, z nekaj razlikami pri obliki sintakse. DirectX je nabor orodja s strojno podporo grafičnemu pospeševanju, obdelavi vhodnih podatkov, omrežju, upravljanju z glasbo, slikami, videi itd. OpenGL pa je le nizkonivojen API za izris grafičnih elementov na zaslon. Ena izmed poglavitnih slabosti knjižnice DirectX je v tem, da je omejena le na Microsoftove produkte. V nasprotju s tem pa je OpenGL na voljo na vseh pomembnejših operacijskih sistemih, kot sta Linux in Mac Os X, ter konzolah PS3 in Wii ter na danes vse večjem trgu operacijskih sistemov na pametnih mobilnih napravah, kot sta Android in iOS. Na operacijskem sistemu Windows imamo na voljo obe knjžnici in prav tako dobro podporo z dokumentacijo, kar se tiče programiranja. Vendar pa se bomo zaradi širše podprtosti še na ostalih platformah odločili za razvoj s knjižnico OpenGL [19, 21].

OpenGL (Open Graphics Library) je nizkonivojski proceduralni programski vmesnik, ki programerju omogoča, da do potankosti opredeli vsak korak, potreben za izris scene. Nizkonivojska zasnova programerju omogoča tudi večjo svobodo pri implementaciji novih algoritmov za izris na sceno, a zahteva dobro poznavanje grafičnega cevovoda. V osnovi OpenGL deluje tako, da prejme osnovne elemente, kot so točke, daljice ali trikotniki, in jih nato pretvori v slikovne pike (piksle), ki so potem izrisane na zaslonu. To delo

opravlja grafični cevovod, ki je poznan tudi kot avtomat OpenGL (OpenGL state machine). Večina ukazov, ki jih OpenGL pošlje naprej v grafični cevovod, je v obliki osnovnih elementov ali ukazov, ki povedo, kako naj cevovod obdeluje te elemente. Pred izidom OpenGL 2.0 je vsaka stopnja cevovoda opravljala fiksno določeno delo in je bila nastavljiva le v manjši meri. Z OpenGL 2.0 in vsemi kasnejšimi verzijami dalje pa smo dobili na voljo več stopenj, ki so s pomočjo jezika GLSL popolnoma programabilne [19, 20].

OpenGL omogoča odličen realno-časovni prikaz grafičnih elementov na sceni, vendar moramo, če jih hočemo prikazati na zaslonu, najprej omogočiti izris vsebine na zaslonu z OpenGL. Ker je OpenGL nizkonivojski grafični API, pomeni tudi, da ne omogoča oziroma nima potrebne funkcionalnosti, ki bi nam zgradila okno z vsebino OpenGL za našo aplikacijo. Zato si moramo sami ustvariti okno z vsebino OpenGL. Da ni treba pisati odvečne kode in izgubljati časa po nepotrebnem, obstajajo že napisane knjižnice, ki bodo namesto nas zgradile okno z vsebino OpenGL na izbrani platformi:

- GLUT (OpenGL Utility Toolkit) je že dolga leta najpreprostejši način za prikaz vsebine OpenGL na zaslonu. GLUT je bil izdelan z namenom ustvarjanja majhnih do srednje velikih programom z vsebino OpenGL, ki služi kot tanka plast med obstoječim operacijskim sistemom in OpenGL. Njegova uporabnost temelji na izgradnji in upravljanju oken z vsebino OpenGL, sestavi glavne zanke, osveževanju vsebine, sprejemu vhodnih podatkov itd. Danes je GLUT že precej zastarel in nezanimiv, vendar je preprost API, ki omogoča spoznavanje in delo s knjižnico OpenGL. Počasi ga že izpodrivajo bolj napredne knjižnice, kot sta na primer GLFW in SDL [22, 23].

- SDL (Simple Direct Media Layer) je odprtokodna, multimedijska knjižnica, spisana v programskem jeziku C in nudi podporo za vsebino OpenGL na vseh večjih platformah. Programerji jo uporabljajo pri razvoju računalniških iger in aplikacij, ker nudi še veliko dodatnih funkcionalnosti in omogoča poganjanje aplikacije OpenGL na različnih platformah. Kakor GLUT je SDL le tanka plast med obstoječim operacijskim sis-

temom in OpenGL-om. Knjižnica v primerjavi z GLUT omogoča še veliko več kot le izgradnjo okna in je razdeljena na več podsistemov, ki vključujejo obdelavo videa, glasbe, nalaganje slik, podporo večnitenju, branje vhodnih podatkov itd. [25, 23].

- GLFW je novejša knjižnica v primerjavi z ostalimi, napisana v programskem jeziku C in nam omogoča, da lahko na preprost način ustvarimo vsebino OpenGL in z njo tudi upravljamo. Zanimiva je predvsem zato, ker omogoča enostavno upravljanje z okni, branje vhodnih podatkov s tipkovnice, miške ali krmilne palice in omogoča odlično upravljanje s časom. Kot vse podobne knjižnice je GLFW tanka, med-platformna, abstraktna plast za aplikacije, kjer grafični izhod temelji na OpenGL-u. Zelo je uporabna pri razvoju aplikacij, ki lahko bivajo na več platformah hkrati, prav tako pa je primerna tudi za vse razvijalce, ki se odločijo za razvoj na eni platformi, temelječi na specifičnih funkcijah, in je bolj namenjena širšemu naboru različnih aplikacij. V osnovi GLFW ni knjižnica z uporabniškim vmesnikom, ki omogoča le to, da odpre prazno okno, brez menijev, gumbov in ostalih gradnikov, značilnih za uporabniški vmesnik [24].

Tako SDL kot GLFW sta odlični knjižnici, vendar se bomo pri razvoju aplikacije odločili za GLFW, ki je po svoji velikosti precej manjša v primerjavi s SDL in je zato še najmanj odvisna od platforme. To, da je manjša, pomeni, da ima tudi manj funkcij na voljo v primerjavi s SDL, kar pa niti ni ravno slabost, ker so skrbno izbrane le najbolj pomembne. GLFW, na primer, nima podpore za zvok, za kar imamo na voljo že knjižnico OpenAL (Open Audio Library). Zaradi svoje velikosti, prenosljivosti in enostavnosti je GLFW še najbolj primerna za izdelavo naše simulacije.

4.5.1 OpenGL razširitve in GLEW

Ena izmed prednosti, ki je omogočena v OpenGL API, je tudi to, da je lahko takoj razširljiva za uporabo novih tehnologij, ki so razvite na grafični strojni

opremi. Z uporabo razširitvenega mehanizma OpenGL lahko proizvajalci strojne opreme razlikujejo nove razširitve od obstoječih, da lahko razvijalci programske opreme hitro dostopajo do dodatnih zmogljivosti in najnovejših tehnologij na grafični strojni opremi. Nove razširitve omogočajo razvijalcem aplikacij nove možnosti pri izrisu, ki so nad standardnimi zmogljivostmi, ki so že vgrajene v OpenGL API. Poleg tega razširitve OpenGL omogočajo tekoče sledenje trenutnim trendom, da je lahko OpenGL API na tekočem z zadnjimi inovacijami, ki jih uvede posamezen ponudnik na področju grafične strojne opreme, in z algoritmi za izris. Razširitev, narejena iz smeri enega ponudnika strojne opreme, se imenuje vendor-spefic, kar pomeni, da je razširitev specifična le za določenega ponudnika, razširitev, narejena izmed več ponudnikov, pa se imenuje multivendor, kar pomeni, da se ponudniki med seboj dogovorijo, katere razširitve bodo vključili. Če se izkaže, da je razširitev dovolj dobra, potem jo OpenGL ARB (Architecture Review Board) predlaga kot odobreno razširitev ARB. Če pa se razširitev izkaže kot zelo uporabna, pa se mogoče pri ARB odločijo, da jo integrirajo kot standardno razširitev v OpenGL API. Podobno se je zgodilo že pri senčilnem jeziku GLSL, ki je postal del standardne razširitve z verzijo OpenGL 2.0 in z vsemi kasnejšimi. Zmožnost, da lahko upravljamo z različnimi razširitvami, naredi OpenGL zelo zmogljiv API, ker ostane jedro kode tudi za nazaj združljivo. To pomeni, da so vsi programi OpenGL, ki so bili spisani pred desetimi leti nazaj, še vedno delujoči tudi danes. Pregled vseh možnih razširitev, ki so nam trenutno na voljo, se da pogledati z ukazom glGetString(GL_EXTENSIONS), ki nam vrne s presledkom ločen seznam vseh obstoječih razširitev [27, 28].

GLEW (OpenGL Extension Wrangler Library) je knjižnica C/C++, ki nam pomaga pri iskanju in nalaganju primernih razširitev OpenGL. GLEW nudi v času izvajanja učinkovite mehanizme, pri določanju katere razširitve OpenGL so podprte na ciljni platformi. Vse razširitve OpenGL so prikazane v zaglavni datoteki, ki predstavlja avtomatično generiran seznam vseh uradno podprtih razširitev. GLEW je na voljo v širokem naboru operacijskih sistemov, ki vključuje Windows, Linux, Mac OS X, Solaris itd [26].

4.5.2 Programski jezik za senčilnik

Na področju računalniške grafike senčilnik (angl. shader) pomeni program, ki se izvede na grafični procesni enoti in je uporabljen pri senčenju. Senčenje je proces izdelave pravih stopenj osvetlitve in zatemnitve znotraj slike, da se doseže želeni efekt, danes pa se uporablja tudi za druge stvari, kot je post-procesiranje podatkov. Senčilniki računajo izrisljive efekte na grafični kartici z visoko stopnjo prilagodljivosti. Senčilni jeziki so uporabljeni pri programiranju grafične procesne enote (GPU) v programirljivem cevovodu, ki je danes že v večji meri izpodrinil cevovod s fiksno funkcionalnostjo, ki je omogočal le osnovne transformacije na geometriji in senčenje točk (pikslov). Z uporabo senčilnikov smo dobili možnost, da lahko izdelamo želene efekte po lastni potrebi. S tem smo dobili možnost, da lahko drastično izboljšamo realizem v virtualnem svetu s pisanjem kode, ki se izvede na grafični procesni enoti. Običajno senčilni program razdelimo na dva dela, kjer začetni del obravnava oglišča, zaključni del pa točke oziroma piksle. To pomeni, da je treba za vsak model senčilnika posebej napisati kodo, kaj naj dela. V prvi del se naložijo samo oglišča, na katerih se bo koda, spisana za ogliščni model senčilnika, izvedla za vsako oglišče posebej. V tem delu lahko še dodatno spremenimo pozicijo ogliščem, izračunamo osvetlitev, normale, teksturne koordinate itd. ali posredujemo podatke naprej v fragmentni model senčilnika. V drugem delu pa se točkovni model senčilnika sprehodi čez vsako točko posebej in izvede kodo, napisano za točkovni model senčilnika. V tem delu senčilnika se lahko odločimo, katero barvo bomo dali poligonu, vzorčimo podatke iz teksturnega objekta ali pa izračunamo osvetlitev itd. Poleg teh dveh obstajajo še drugi modeli sečilnikov, kot je senčilnik geometrije, ki nam ustvari nove ali pa odstrani obstoječe poligone, ki gredo v izris, in nadzorni ocenjevalec teselacije za povečanje ali zmnjšanje števila poligonov. V zadnje dele se ne bomo spuščali, ker jih ne potrebujemo ali pa nimajo podpore pri podani specifikaciji naše strojne opreme [29, 30, 31].

Skupaj z OpenGL bomo uporabili tudi Graphics Library Shader Language (GLSL ali GLslang), ki je privzeti senčilni jezik za OpenGL. GLSL

je visokonivojski senčilni jezik, ki temelji na podobni sintaksi, kot jo ima C. Izdelan je bil pri organizaciji OpenGL ARB review board z namenom, da omogoči razvijalcem pri izrisu računalniške grafike večji nadzor nad grafičnim cevovodom. Znotraj kode GLSL tudi omogoča uporabo struktur, kot so matrike in vektorji, in transformacije z matrikami. Poleg tega ima podporo tudi za osnovne matematične funkcije, ki so obvezne pri vsakem programskem jeziku. Za shranjevanje podatkov iz centralnega procesorja uporablja tipe spremenljivk *uniform* in *attribute* ter dostop do tekstur preko teksturnih objektov *texture* [29].

4.6 Pogon simulacije

Pogon za simulacijo terena je bil v celoti zgrajen s knjižnico OpenGL in ne vsebuje knjižnic, ki bi že imele lasten sistem za upravljale, urejanje ali izrisovanje objektov na sceni. V pogon je bilo treba vključiti tudi nekaj odprte kode, predvsem na določenih mestih, ki bi jih bilo bolj težko ali časovno preveč potratno posebej implementirati. Uporabili smo knjižnice, ki nam omogočijo branje datotek .obj pri vnosu podatkov iz 3D-modela, uvažanje slik .png in prikaz besedila.

Kot je že bilo že omenjeno, smo se odločili za slikovni format PNG, ki bo uporabljen za uvoz slik v aplikacijo. Za ta namen smo izbrali odprtokodno knjižnico, ki se imenuje LodePNG in je napisana za programski jezik C/C++. Knjižnica nam omogoča dekodiranje in zakodiranje širokega nabora PNG-formatov.

V aplikacijo je vključena tudi možnost za izpis besedila na zaslon, ki je lahko zelo uporaben pri prikazu informacij trenutnega stanja v programu, pri izbiri različnih funkcionalnosti, ki so vgrajene v simulacijo, ali pa za prikaz števila sličic na sekundo. Za prikaz oziroma izris besedila na zaslonu je uporabljena knjižnica Freetyepe. Knjižnjica omogoča nalaganje fontov v teksturo, ki se nato v sečilnem programu uporabi za izris črk, na podlagi katerih lahko potem izpišemo na zaslon poljubno besedilo. Različne pisave

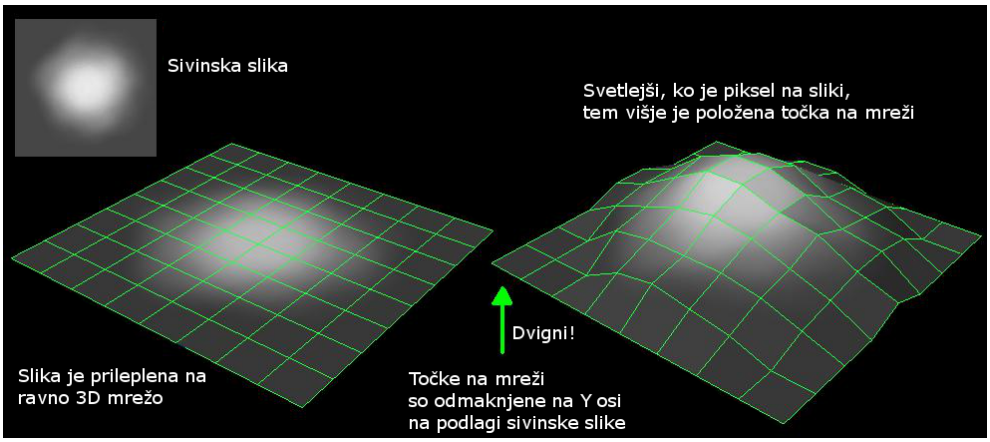
(angl. fonts) so lahko shranjene v datoteki s končnico .ttf, ki se ob zagonu programa naloži v teksturo.

Za večkratni izris na zaslon z več stopnjami hkrati smo uporabili `Frame buffer objects` (FBO). Za izpis podatkov iz pomnilnika grafične kartice smo uporabili funkcijo, ki omogoča branje podatkov iz pomnilnika grafične kartice z uporabo `Pixel buffer objects` (PBO). Več o tem pa v naslednjih poglavjih. V pogonu so bile implementirane še različne funkcije, ki omogočajo izris primitivov, črt, poligonov in besedila, tako v 3D-svetu kot na zaslonu. Za simulacijo so bili izdelani še posebni objekti za lažje upravljanje z ogliščnimi ter teksturnimi podatki, ki olajšajo sprotno generiranje različnih objektov v medpomnilniku (angl. bufferjev), bolj temeljito delo s podatki znotraj in večji nadzor nad hranjenem podatkov, kot so normale, teksturne koordinate, teksture ali pozicija oglišč. Poleg tega nam tudi omogočijo, da imamo podatke ločeno shranjene tako na glavnem pomnilniku kot na pomnilniku grafične kartice.

Za bolj učinkovito delo s programskim jezikom C++ je uporabljena tudi knjižnica Boost. Boost pri razvoju programske opreme predstavlja zbirko knjižnic, ki so napisane izključno za programski jezik C++ in nudijo široko podporo za različna opravila ter strukture, kot so linearna algebra, večnitnost, procesiranje slik, pseudoslučajno generacijo naključnih števil, skupni kazalci (shared pointers) itd. Trenutna verzija vsebuje kar 18 individualnih knjižnic. Knjižnice Boost so namenjene za uporabo širši množici razvijalcem v jeziku C++. Boost je produkt obsežnega dela in raziskav na področju generičnega programiranja in metaprogramiranja v programskem jeziku C++. Večino knjižnic Boost sestavljajo zglavne datoteke, ki so napisane kot vrstične funkcije [32].

4.7 Teren

Po pregledu različnih možnosti za implementacijo terena se bomo v tem primeru odločili za izdelavo mreže terena, ki bo podatke o odmikih višine



Slika 4.1: Izdelava terena s pomočjo sivinske slike [35]

pridobila iz sivinske slike. Sivinska slika se bo v obliki teksture pri izrisu terena vnesla v senčilni program. Podatki, ki bodo naloženi v sivinsko sliko, naj bi predstavljali odmike oglišč na Y-osi in bodo dali končno obliko na terenu. Poleg tega bomo na mreži terena izdelali tudi način za dinamično deformacijo terena v realnem času med izvajanjem aplikacije, kar bomo dosegli z dodajanjem ali odštevanjem vrednosti k Y-komponenti v obliki odmikov višine na mreži terena [35].

Višinska slika, ki vsebuje odmike (angl. hightmap), ima podatke shranjene samo v enem kanalu, v katerem je shranjena razdalja neke točke na mreži terena od najnižje možne vrednosti. Kot je bilo že omenjeno, je višinska slika v bistvu sivinska slika, na kateri črna barva predstavlja najnižjo točko na terenu, bela barva pa predstavlja najvišjo točko. Kakšne vrednosti bodo imeli piksli na sliki, pa je odvisno od predstavitve terena (slika 4.1).

Pri izrisu terena si bomo pomagali z dostopanjem do teksture v ogliščnem delu senčilniškega programa (podprto od glsl-verzije #130 naprej). Tehnika nam bo omogočila, da bodo podatki o višini terena dostopni znotraj senčilnega programa in nam bodo na voljo nato še za nadaljnje procesiranje na grafični procesni enoti.

Za dostop do podatkov v ogliščnem modelu senčilnika moramo imeti

podatke v teksturnem objektu naložene v obliki dvodimenzionalnega seznama. Podatke imamo lahko v teksturi shranjene le s plavajočo vejico, kar je tudi edini način, ki nam dopušča branje iz teksturnega objekta v ogliščnem modelu senčilnika. To pa zato, ker sta v obstoječem naboru grafičnih kartic, ki nudijo podporo za OpenGL 3.0 in več, uradno podprta le dva formata za branje podatkov iz teksture v ogliščnem modelu senčilnika: GL_LUMINANCE32F_ARB in GL_RGBA32F_ARB. Čeprav je bilo lepljenje tekstur v ogliščnem modelu senčilnika mogoče že v OpenGL 2.1 s pomočjo dodatnih razširitev, je standardna razširitev postala šele z verzijo 3.0. Obstajajo pa tudi drugi formati, ki nimajo širše podpore med proizvajalci grafičnih kartic oziroma imajo podporo le pri določenem proizvajalcu. V tem primeru se bomo odločili za format GL_LUMINANCE32F_ARB, ki ga bomo uporabili pri shranjevanju podatkov o odmikih višine v teksturo [33, 34].

Z branjem iz teksture v ogliščnem delu senčilnika lahko dosežemo precej enostavno in učinkovito rešitev za deformacijo terena. To pomeni, da bomo izvedli vse izračune normal, osvetlitve, večteksturni preliv in deformacijo terena pa moramo izvesti na grafični procesni enoti. Velikost mape bo odvisna od podanih dimenzij slike s podatki o višini terena. Ker OpenGL lahko sprejme samo teksture dimenzij 2^n, bo temu prilagojena tudi velikost mape.

Poglavje 5

Implementacija

V tem delu se bomo osredotočili na implementacijo simulacije terena. Pri tem pa bodo po posameznih poglavjih predstavljeni potrebi koraki, ki so nujni za izgradnjo naše simulacije. Pri zgradbi terena je treba upoštevati tudi določeno število stopenj, s katerimi se srečamo pri vsaki implementaciji terena, od sestave mreže in izvedbe osvetlitve do teksturnega preliva.

5.1 Zgradba terena

Najprej je treba izdelati mrežo terena, na kateri se bo kasneje lahko izvedlo osvetlitev z vsemi vizualnimi efekti, in spreminjanje topologije terena z orodji, ki omogočajo izvedbo deformacije na terenu. Obstaja več pristopov, ki omogočajo mreže terena. V tem primeru se bomo odločili za kombinacijo enotske mreže, izrisa poligonov s tehniko povezanih oglišč trikotnikov (angl. Triangle strip) in uporabo sivinske slike, ki nosi podatke o odmikih višine.

Podatke s terena bomo shranili v vrsto objektov, ki se imenuje `Vertex Buffer Objects` (VBO).

Vertex buffer object ustvari prostor v zelo zmogljivem pomnilniku VRAM na grafični kartici. Podatke, shranjene v medpomnilniku (angl. buffer), lahko kopiramo ali pa jih kar prilepimo v pomnilnik in jih nato poljubno spreminjamo. OpenGL nam omogoča uporabo funkcij, s katerimi lahko hitro in

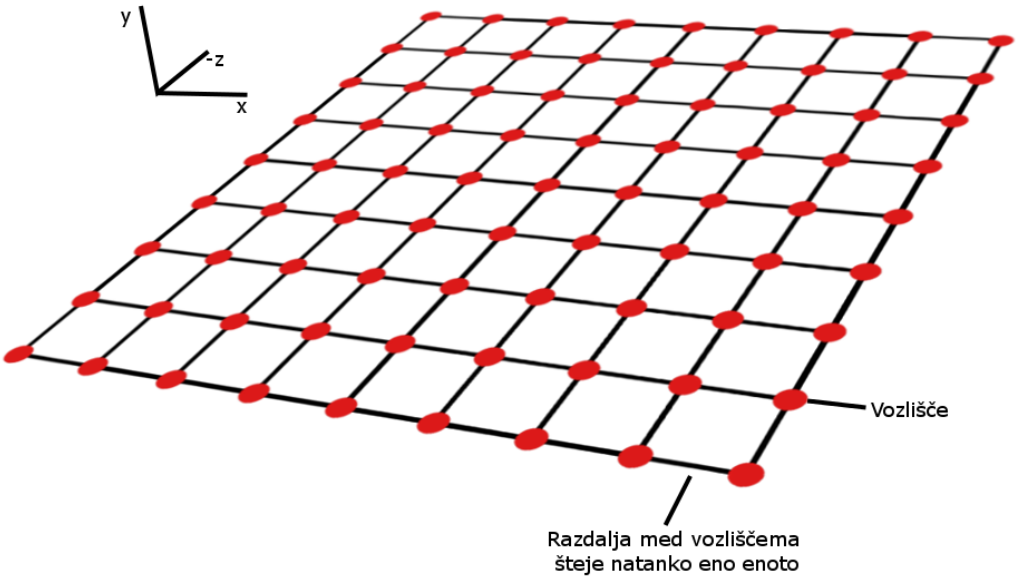
enostavno dostopamo do podatkov znotraj medpomnilnika. Tak je na primer klic funkcije *glBufferData*, ki nam omogoči prenos podatkov iz glavnega pomnilnika v pomnilnik, *glBufferSubData* omogoči spreminjanje vsebine na že prednaloženih podatkih, *glDeleteBuffers* pa poskrbi za brisanje naslovnega prostora v pomnilniku skupaj s shranjenimi podatki. VBO je nastal z namenom, da nam ni treba ob vsakem ponovnem izrisu na zaslon prenašati ogromne količine podatkov o geometriji iz glavnega pomnilnika. Zato je veliko hitreje, če so ti podatki že shranjeni v pomnilniku grafične kartice. V 3D-igrah in v simulacijah je namen VBO-jev shranjevanje pozicij oglišč, normal in teksturnih koordinat za različne objekte na sceni v pomnilnik grafične kartice, preden se jih pošlje na izris [36].

V našem primeru ga bomo uporabili za hranjene podatkov X- in Z-koordinat oglišč z enotske mreže, ki bo razložena v naslednjem poglavju. Normale za osvetlitev, kot osvetlitev samo, in teksturne koordinate bodo izračunane iz teh koordinat in teksture z odmiki.

5.1.1 Enotska mreža

Za izdelavo mreže terena potrebujemo enotsko mrežo, kvadratne oblike, na kateri bomo lahko enostavno prebrali odmike v višini oglišč. Enotska mreža potrebuje za hranjenje samo dve koordinati, X in Z, pri tem pa je vsako vozlišče od sosednjega oddaljeno natanko eno enoto (slika 5.1), kar pomeni, da enako velja tudi za vrstice in stolpce na mreži.

Takšna mreža je tudi najenostavnejši način za prikaz terena, ker nam omogoča, da so izračuni za osvetlitev, algoritmi za preverjanje trkov in izračuni teksturnih koordinat mnogo hitrejši ter enostavnejši za implementacijo. Tudi mreže, ki niso enotske, imajo svoje prednosti, kot na primer, da omogočajo previs na nekem delu terena, česar pa ni mogoče doseči na enotski mreži [6].



Slika 5.1: Prikaz enotske mreže z enakomerno porazdeljenimi oglišči

5.1.2 Pas trikotnikov (angl. Triangle strip)

Pas trikotnikov je serija med seboj povezanih trikotnikov, ki si delijo vozlišča in omogočajo racionalno uporabo prostora v pomnilniku. Glavni cilj je zmanjšanje uvoza nepotrebnih podatkov pri izdelavi trikotnikov in predvsem zmanjšanje porabe VRAM-a na grafični strojni opremi. Ker nam ni treba hraniti oglišč za vsak trikotnik posebej, je tudi manj porabljenih podatkov za hrambo oglišč na disku. S tem se nam je število oglišč, ki jih potrebujemo za izris trikotnikov, zmanjšalo s 3N na N+2, kjer N pomeni število potrebnih trikotnikov za izris.

Ko izrisujemo mrežo terena z načinom GL_TRIANGLE_STRIP, bo OpenGL pri uporabi vsakega izmed treh vozlišč iz seznama vozlišč in s postopnim pomikanjem naprej, vozlišče za vozliščem, ustvarjal nov trikotnik. To pomeni, da ima vsak naslednji trikotnik skupno po dve vozlišči iz predhodnega trikotnika. Za primer so podane postavitve vozlišč, ki bi lahko bile povezane v skupino trikotnikov:

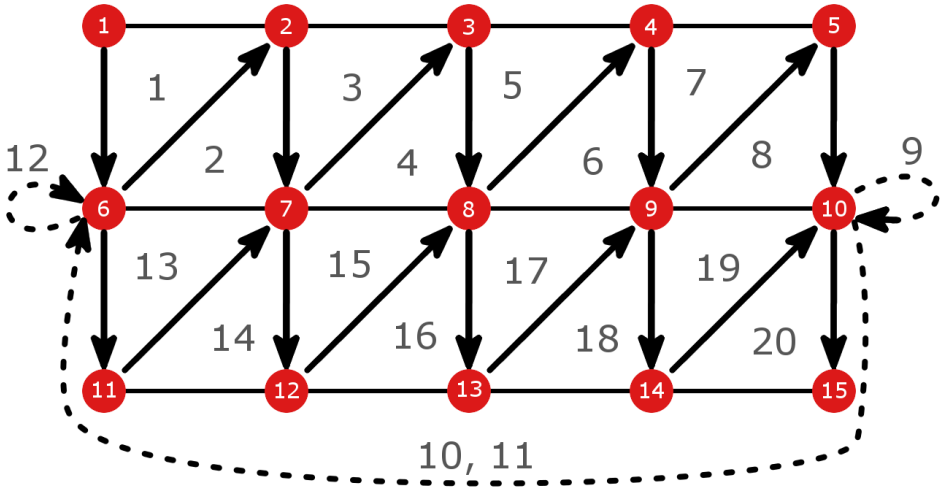
- Trikotnik 1 = 1, 6, 2

- Trikotnik 2 = 6, 2, 7
- Trikotnik 3 = 2, 7, 3
- Trikotnik 4 = 7, 3, 8
- Trikotnik 5 = 3, 8, 4
- Trikotnik 6 = 8, 4, 9
- ...

OpenGL mora pri sestavljanju trikotnikov prav tako upoštevati ureditev vozlišč, ki se imenuje tudi trikotniški red. To pomeni, da ureditev prvih treh vozlišč določa ureditev tudi vseh ostalih vozlišč. Če ima prvi trikotnik ureditev vozlišč v nasprotni smeri urinega kazalca, bodo imeli tudi vsi ostali trikotniki ureditev v nasprotni smeri urinega kazalca (zamenjana vozlišča so odebeljena)(slika 5.2):

- Trikotnik 1 = 1, 6, 2
- Trikotnik 2 = **2**, **6**, 7
- Trikotnik 3 = 2, 7, 3
- Trikotnik 4 = **3**, **7**, 8
- Trikotnik 5 = 3, 8, 4
- Trikotnik 6 = **4**, **8**, 9
- ...

Takšna ureditev je povsem na mestu v primeru, kadar želimo imeti samo eno vrstico trikotnikov. Če pa želimo izdelati kvadratno mrežo terena, ki bo znotraj sebe imela vse kvadrate istih dimenzij, moramo najti boljši način, ki bo omogočal prehod iz trenutne vrste v naslednjo vrsto trikotnikov. To lahko dosežemo s ponavljanjem zadnjega vozlišča trenutne vrstice in prvega vozlišča v naslednji vrstici (slika 5.2). Kako bi to izgledalo v kodi s seznamom šestih stolpcev na posamezno vrstico, je prikazano v spodnji kodi:



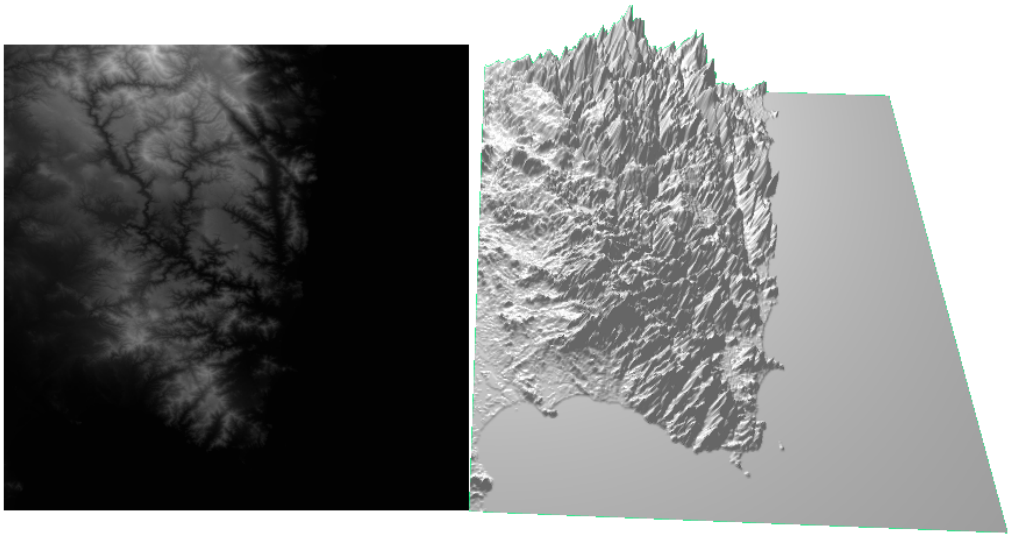
Slika 5.2: Zgradba mreže terena s povezanimi trikotniki

```
seznamIndeksov = {
   1, 6, 2, 7, 3, 8, 4, 9, 5, 10, 10, 6, 6, 11, 7, 12, 8,
   13, 9, 14, 10, 15
}
```

S ponavljanjem zadnjega vozlišča v trenutni vrstici in prvega vozlišča v novi vrstici smo dobili izrojene trikotnike, ki bodo izpuščeni skozi proces rasterizacije v cevovodu in nam bodo povezali prvo vrstico trikotnikov z drugo ter tako dalje [37].

5.1.3 Višinska slika (angl. Hightmap)

Uporaba tekstur, ki hranijo različne podatke, ne samo o barvi pikslov na sliki, je veliko bolj široka, kot se zdi na prvi pogled. S podatki v teksturi si lahko pomagamo na različne načine. Lahko jo uporabimo kot teksturo za lepljenje izboklin (angl. bump mapping), ki nam služi za prikaz drobnih detajlov na objektu, lepljenje tekstur na sceni ali pa jo uporabimo pri odmikih višine, s



Slika 5.3: Prikaz dela otoka Korzike v 3D iz sivinske slike

čimer lahko nato izdelamo realno-časovno deformacijo terena [38].

Kot je bilo že rečeno, smo v našem primeru sivinsko sliko uporabili za izdelavo terena z odmiki višine. V višinsko sliko lahko naložimo podatke o različnih odmikih, ki oblikujejo teren in lahko temeljijo na resničnih virih, na primer izris Korzike (slika 5.3), ali pa na umetno izdelanih virih. Za pridobitev odmikov iz sivinske slike imamo na disku v slikovnem formatu PNG shranjeno sliko, ki vsebuje le en kanal, na katerem je mogoče hraniti 8-bitne podatke. Ob inicializaciji terena se ti podatki dekodirajo in naložijo v teksturni objekt, od koder jih je mogoče kasneje vzorčiti v ogliščnem modelu senčilnika.

Ker lahko ogliščni del senčilnika naložimo samo v teksturo, ki ima podatke shranjene s plavajočo vejico (angl. floating point), je treba najprej te podatke pretvoriti v teksturo, v kateri bodo ti podatki shranjeni v 32-bitnem formatu s plavajočo vejico. Kot je bilo že omenjeno, je ta format obvezen zaradi podprtih formatov (Poglavje 4.7), če želimo teksturo brati v ogliščnem delu senčilnika.

5.2 Osvetlitev

Izris interaktivne 3D-grafike z osvetlitvijo v realnem času temelji na približni simulaciji zakonitosti, kot so te predstavljene v fiziki ali optiki. Izračuni za osvetlitev so v večini primerov le približek k realni osvetlitvi in izhajajo iz preprostih modelov, ki bolj kot ne slonijo na plitvih empiričnih opazovanjih [39]. V naši simulaciji bomo uporabili tri primere takšne osvetlitve:

- ambientno osvetlitev,
- zrcalno osvetlitev in
- difuzno (razpršeno) osvetlitev.

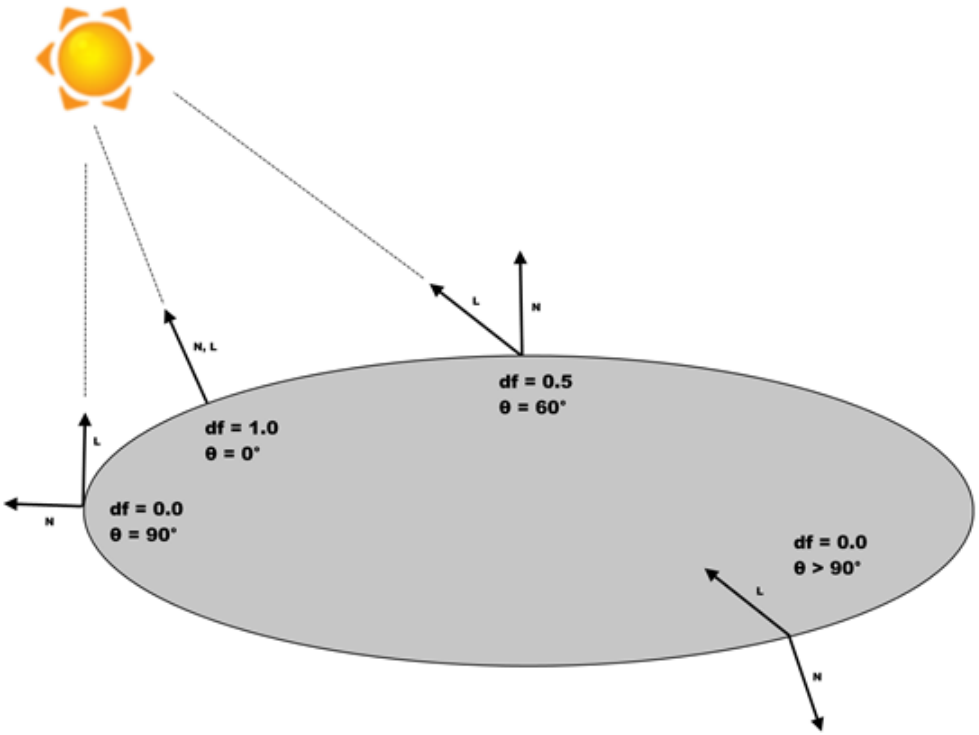
5.3 Ambientna osvetlitev

Prepričljiva ambientna osvetlitev z mehkimi, nežnimi sencami je za izris lahko zelo zapletena. Ambientna osvetlitev, uporabljena v povezavi z OpenGL, pa velikokrat pomeni nekaj povsem samoumevnega: vseskozi monotono, skupno barvo. Imenovati to osvetlitev je bolj kot ne vprašljivo, saj intenziteta ni pogojena s smeri vira osvetlitve ali orientacije površine, ampak je pogosto kombinacija, ki je sestavljena iz drugih modelov osvetlitve, da se lahko dobi svetlejšo površino poligonov na nekem objektu [39].

5.4 Difuzna osvetlitev

Je najbolj pogosto uporabljena oblika realno-časovnega osvetlitve, pri kateri se prehod med intenziteto osvetlitve na posameznem delu površine spreminja glede na kot med površino in virom svetlobe. Takšna oblika osvetlitve je najbolj pogosta, ker je enostavna za izračun in da zadovoljiv občutek globine za človeško oko.

V zgornjem diagramu 5.4 je vektor L prikazan kot enotski vektor, ki je usmerjen proti viru svetlobe. N je normala na površini in je prav tako



Slika 5.4: Difuzna osvetlitev [39]

enotski vektor, ki je pravokoten na površino poligona. Faktor razpršenosti svetlobe oziroma razpršilni faktor df leži med vrednostmi 0 in 1 in se ga pomnoži skupaj z intenziteto svetlobe ter barvo materiala, iz česar lahko potem dobimo končno razpršeno barvo. Izračun razpršene barve:

$$DifuznaBarva = IntenzitetaOsvetlitve * BarvaMateriala * df$$

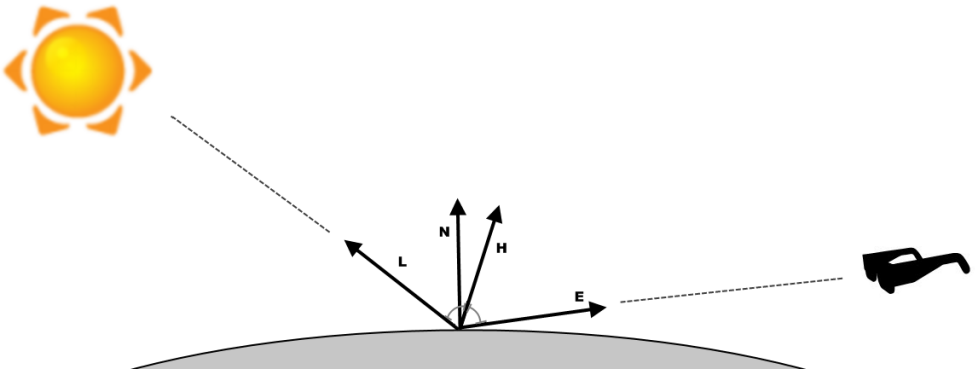
Df je vrednost, ki jo dobimo pri izračunu skalarnega produkta normale na površini poligona in vektorja, ki je usmerjen proti viru svetlobe. Nato pa vse skupaj še primerjamo, ali je dobljeno število med vrednostmi 0 in 1. V nasprotnem primeru, ko dobimo negativno število, ga nastavimo na vrednost 0, kar pomeni, da tam ni osvetlitve [39]. Izračun razpršilnega koeficienta:

$$df = max(0, N * L)$$

5.5 Zrcalna osvetlitev

Za razpršeno osvetlitev je značilno, da ni pogojena s pozicijo kamere, ampak osvetlitev fiksne točke ostaja enaka ne glede na to, iz katere strani gledamo (slika 5.5). To je v nasprotju z zrcalno osvetlitvijo, ki omogoča, da se pozicija osvetlitve na terenu spreminja v odvisnosti od lokacije opazovalca. Zrcalna osvetlitev simulira poudarjen odsev svetlobe na gladki površini objekta le pod določenim kotom gledanja na površino. Izračun zrcalne osvetlitve je veliko bolj zahtevna operacija v primerjavi z razpršilno osvetlitvijo, ker potrebujemo eksponentno formulo, ki nam izračuna moč odseva na površini. Z eksponentom določimo delež odboja svetlobe na površini nekega objekta [39]. Večji ko imamo eksponent, tem bolj bleščeč odsev dobimo na površini objekta.

H je vektor, ki razpolavlja kot med virom svetlobe in opazovalcem na polovico. Podobno kot pri razpršeni osvetlitvi je cilj izračunati koeficient med 0 in 1. Naslednja funkcija prikazuje ta izračun:



Slika 5.5: Zrcalna osvetlitev [39]

$$H = normaliziraj(L + E) \tag{5.1}$$
$$sf = [max(0, N * H)]^{odsev}$$

V praksi se da vedno pretvarjati, da je opazovalec neskončno oddaljen od točke osvetlitve. V tem primeru potem vektor E odštejemo od vektorja (0, 0, 1). Ta tehnika se imenuje globalni opazovalec. Kadar pa je E uporabljen, se imenuje lokalni opazovalec. Osvetlitev je za naš primer izvedena dinamično, kar nam omogoča, da so lahko nove spremembe pri izrisu terena pravilno videne oziroma prikazane. Ker simuliramo okolje na prostem, smo se odločili za globalno smer osvetlitve, ki nam približno simulira sončno svetlobo. Ker se smer sončne svetlobe ne spreminja med izvajanjem programa, pomeni, da so vsi nadaljnji računi v tej smeri že izključeni iz projekta. Osvetlitev je pri dinamičnem osvetljevanju zelo draga operacija, ker zahteva veliko računov plavajoče vejice. Zato je priporočljivo, da se izvajanje teh opravlja z uporabo grafičnih procesorjev, ki so namenjeni računanju plavajoče vejice. Če hočemo to doseči, moramo uporabiti senčilni program, vendar pa lahko le najnovejša strojna oprema na grafičnih pospeševalnih učinkovito izvaja takšne račune [39].

5.6 Izračun normal

Za izris terena skupaj z osvetlitvijo so nujno potrebne normale. To so tri komponente vektorja, ki označujejo površino na mreži terena za vsak poligon posebej in nam omogočajo simulacijo odboja svetlobe od površine. Obstajata dva različna načina za izračun normal na terenu. Prvi način je izračun ploskih normal (angl. face normals), drugi pa je izračun mehkih normal (angl. smooth normals). Ploske normale so hitrejše in zahtevajo le nekaj izračunov za vsako oglišče. Pri mehkih normalah je pomembno, da se najprej izračuna ploske normale, iz katerih je nato mogoče izračunati skupno povprečje, ki ga lahko uporabimo pri mehkih normalah. Ploske normale so najbolj primerne za gladke površine z velikim številom poligonov in so najbolj uporabne pri vseh vrstah simulacije terena.

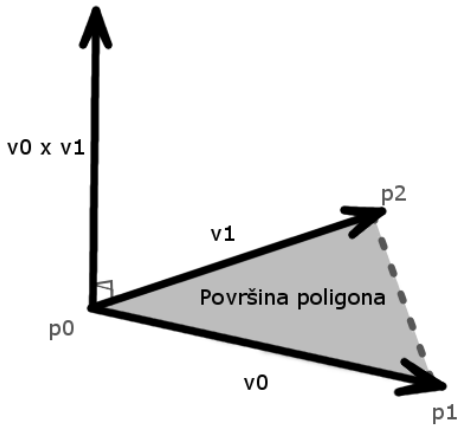
Da dobimo plosko normalo na površini nekega poligona, moramo najprej izračunati razdaljo vektorjev, ki predstavljajo stranice poligona, usmerjene iz izhodiščne točke, ter jih pomnožiti s križnim produktom:

$$N = (p1 - p0) \times (p2 - p0)$$

Na sliki 5.6 vrednost p0 predstavlja položaj izhodiščne točke, p1 in p2 pa sta sosednji točki, ki skupaj določajo ravnino. Na sliki imamo tudi normalo, ki je pravokotna na vektorja v0 in v1. Po izračunu vektorskega produkta smo dobili normalo za določen poligon na mreži terena, ki jo lahko že uporabimo pri osvetlitvi s ploskimi normalami. Križni produkt se lahko izračuna na naslednji način:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} y0 \cdot z1 - z0 \cdot y1 \\ z0 \cdot x1 - x0 \cdot z1 \\ x0 \cdot y1 - y0 \cdot x1 \end{pmatrix}$$

S križnim produktom dobimo vektor, ki je pravokoten na oba sosednja vektorja in je vzporeden normali ravnine. Normalo je treba še normalizirati, da dobimo enotski vektor. Enotski vektor je vektor, ki je dolžine 1. Da lahko



Slika 5.6: Izračun ploske normale na trikotniku

to dobimo, je treba pred tem še normalizirati rezultat prejšnjih izračunov. Normalizacija je dobljena z izračunom dolžine vektorja $||v||$, ki se ga lahko izračuna na sledeči način:

$$\sqrt{x * x + y * y + z * z}$$

To poznano kot koren skalarnega produkta vektorja, ki da magnitudo vektorja ne glede na smer. Ko imamo izračunano dolžino vektorja, vektor samo še delimo z dolžino in dobimo enotski vektor:

$$\hat{V} = \frac{v}{||v||}$$

Izračun enotskega vektorja je precej potratna operacija, ker zahteva kvadratno operacijo in tri deljenja, zato je priporočljivo, da se jo uporablja čim manj, če je le mogoče. Teoretično je tako za mehko senčenje kot plosko senčenje priporočljvo imeti le eno normalizacijo za posamezno oglišče.

Vsaka mehka normala je izračunana iz povprečja sosednjih normal, ki jih dobimo iz osmih sosednjih poligonov. Prav tako pa moramo vzeti v račun

tudi stranice mape, v primeru, da mogoče ob strani nimamo več sosednjega poligona, ker smo najverjetneje dosegli rob mape. Ugotavljanje, ali smo na robu mape in ali imamo ob strani našega poligona še kakšen sosednji poligon, je relativno preprosto. Če je trenutna vrednost indeksa oglišča več kot nič, vemo, da so poleg nas še oglišča na levi. V primeru, da imamo x-vrednost manjšo od dolžine mape, vemo, da so poleg nas na desni še sosednja oglišča. Na podoben način lahko preverimo, ali imamo na vrhu in spodaj na mapi sosednja oglišča, in sicer tako, da preverimo še z-komponento.

Največji problem pri vsaki simulaciji terena je vzdrževanje in osveževanje VBO-jev. Večje ko so spremembe na mreži terena, več je potrebnih izračunov za izris posamezne sličice, kar nam hitro poveča časovno potratnost. Da bi izračunali mehke normale, morajo biti podatki shranjeni na obeh pomnilnikih, tako na tradicionalnem RAM-u kot na VRAM-u za nedoločen čas, ker so lahko vse hitre spremembe opravljene na procesorju, če je prostor že rezerviran zanje. Navadno pri vsaki implementaciji 3D-sveta je, da so mehke normale za celotno mapo izračunane takoj na začetku in so potem prenesene v VBO. Na srednje zmogljivem računalniku bi lahko ta proces trajal tudi sekundo, kar pa nikakor ne pride v poštev pri izvajanju. Ko pride na vrsto deformacija, so samo normale potencialno spremenljivih poligonov ponovno izračunane, brez povprečenja, z namenom, da bi porabili čim manj ciklov na centralnem procesorju [9,15].

Da bi se izognili omenjenim težavam, se lahko obrnemo na grafično procesno enoto in izvedemo potrebne izračune s pomočjo programabilnega cevovoda. Normale, izračunane v ogliščnem modelu senčilnika, temeljijo na rdeči komponenti teksture, v kateri so shranjeni podatki o odmikih višinine na terenu. To je tudi posebnost te implementacije, ki nudi sprotni izračun normal iz slike z odmiki v ogliščnem modelu senčilnika.

Ko imamo normale enkrat izračunane, lahko opravimo še nekaj dodatnih izboljšav na kvaliteti osvetlitve, kot na primer, da uporabimo v senčilnem programu točkovni model osvetljevanja namesto ogliščnega za ambientno, spekularno in razpršeno osvetlitev. Za izboljšanje kvalitete osvetlitve lahko

uporabimo fragmentni model senčilnika.

Izračun mehkih normal v senčilnem programu je zelo podoben tistemu pri klasični implementaciji na centralnem procesorju. Namesto da naredimo obhod zanke skozi vsako oglišče posebej na vseh poligonih terena na centralnem procesorju, je lahko to že sproti narejeno na grafični procesni enoti v ogliščnem modelu senčilnika. V izseku kode 5.1 je prikazan izračun mehkih normal v ogliščnem delu senčilnika.

```
// izračun površinskih normal za oba
// trikotnika, ki sestavljata štirikotnik
vec3 v0 = p2 - p0;
vec3 v1 = p1 - p0;
vec3 v2 = p1 - p3;
vec3 v3 = p2 - p3;

// izračun normale za prvi trikotnik
vec3 fn1 = cross(v0,v1);
// izračun normale za drugi trikotnik
vec3 fn2 = cross(v2,v3);

// normalizacija seštevka obeh normal
vec3 fn = normalize(fn2 + fn1);

// prištevek k povprečnim in normalizacija
average_normal = normalize(fn + average_normal);
```

Izsek kode 5.1: Izračun mehkih normal v ogliščnem modelu senčilnika

V senčilni program moramo poslati tudi dimenzije mape v obliki uniformne vrednosti. Le na podlagi podanih dimenzij mape se da v ogliščnem modelu senčilnika določiti teksturne koordinate iz atributa, ki hrani X- in Z-pozicijo na mreži terena. Teksturo smo dobili kot vir v objekt tipa `sampler2D`, ki predstavlja uniformno vrednost za branje podatkov o odmikih višine. Ko izračunamo teksturno koordinato glede na trenutno pozicijo oglišča na mreži,

jo lahko posredujemo naprej v funkcijo `texture2D`, iz katere bomo prebrali Y-komponento oziroma odmik v višini. Funkcija `texture2D` sprejme samo dva parametra: objekt uniformnega tipa `sampler2D`, v katerem je shranjena tekstura z odmiki višine v ogliščnem modelu senčilnika, in uv-koordinato 2D, s pomočjo katere lahko izberemo želeni teksel na teksturi. Prebrani odmik višine oziroma koordinato Y bomo skupaj s trenutnim atributom, ki hrani 2D-lokacijo na mreži, uporabili pri sestavi oglišč za vsak poligon na mreži terena. Na podoben način se lahko spravimo iskati tudi sosednje koordinate, ampak z odmikom največ za en teksel v eno izmed osmih možnih smeri od trenutne pozicije. Na primer: sosednjo desno koordinato najdemo z dodajanjem enega teksla k X-komponenti, sosednjo zgornjo koordinato pa najdemo z dodajanjem enega teksla k Y-komponenti (Izsek kode: 5.2). Na podlagi uniformne lestvice odmikov lahko hitro pridobimo višino iz sosednjih oglišč.

```
// teksturni koordinati u in v sta pridobljeni na podlagi koordinat
//  x in z, ki ju delimo z dimenzijami višinske slike
float u,v;

// ko imamo za vsako vozlišče teksturno koordinato, moramo le še
// prebrati y-vrednost iz rdeče komponente v teksturi

// levo zgornje oglišče štirikotnika
u = x/MapWidth;
v = z/MapHeight;
// sestavimo vektor za vsako točko v kvadratu (x, y, z)
vec3 p0=vec3(x,texture(YHightmap, vec2(u,v)).r*ScaleFactorY,z);

// desno zgornje oglišče
u = (x+1)/MapWidth;
v = z/MapHeight;
vec3 p1=vec3(x+1,texture(YHightmap, vec2(u,v)).r*ScaleFactorY,z);
```

```
// levo spodnje oglišče
u = x/MapWidth;
v = (z+1)/MapHeight;
vec3 p2=vec3(x,texture(YHightmap, vec2(u,v)).r*ScaleFactorY,z+1);

// desno spodnje oglišče
u = (x+1)/MapWidth;
v = (z+1)/MapHeight;
vec3 p3=vec3(x+1,texture2D(YHightmap, vec2(u,v)).r*ScaleFactorY,z+1);
```

Izsek kode 5.2: Izračun oglišč štirikotnika s pomočjo višinske slike

V izseku kode 5.2 parametra `MapWidth` in `MapHight` predstavljata dimenziji mape terena. Dimenziji delimo z X- in Z-koordinatami sosednjih oglišč, iz katerih lahko potem izračunamo teksturni koordinati u in v. `YHightmap` je tekstura, ki vsebuje podatke o višini terena in `ScaleFactorY` predstavlja lestvico med posameznimi enotami višine. Y-koordinato poiščemo tako, da pošljemo teksturo, ki vsebuje podatke o višini, in teksturni koordinati u in v v funkcijo `texture2D`. Iz tega pa na podlagi rdeče komponente samo še preberemo vrednost.

Za oglišča, ki so na desni in zgornji strani glede na trenutno izbrano, lahko uporabimo podobno metodo za izračun ploskih normal za vsak trikotnik. Če hočemo izračunati mehke normale v senčilnem programu, moramo dobiti vsega 12 lokacij oglišč, k temu pa dodamo še osem sosednjih normal ter skupaj normaliziramo rezultat. Te operacije niso pretirano drage, ker je tekstura z višinskimi podatki shranjena v grafičnem pomnilniku in je enostavno dostopna znotraj senčilnega programa. Križni produkt in druge operacije s plavajočo vejico so na ta način veliko hitreje izračunljive za celotno mapo, kar se hitro pozna pri izrisu vsake sličice [6, 39].

5.7 Teksture

Obstaja veliko možnosti pri izbiri tekstur za teren. Najbolj idealna rešitev bi bila izdelava visoko kvalitetnih, različnih tekstur, ki ne porabijo velike količine pomnilnika. Ena izmed možnosti za dosego tega cilja je do neke meje uporaba proceduralno generirane vsebine.

5.7.1 Večteksturna metoda z dinamičnimi teksturnimi utežmi

Večteksturna metoda je implementirana z uporabo senčilnega programa GLSL v ogliščnem in fragmentnem modelu senčilnika, kar da učinkovito uporabo tekstur in se izogne kompleksnosti OpenGL-ovim teksturnim okoljskim stanjem, ki zahtevajo uporabo CPU-ja in fiksnega cevovoda. Prav tako omogoča enostaven dostop do več teksturnih enot hkrati. Teksturna enota deluje kot vsebovalnik za dostop do specifične teksture. Na grafični strojni opremi potrebujemo več kot le eno teksturno enoto, da imamo lahko podporo za več tekstur v enem izrisovalnem obhodu. Da bi učinkovito zmanjšali porabo pomnilnika, smo uporabili pet osnovnih tipov tekstur: travo, kamenje, pesek, sneg in vodo (slika 5.7). To so obenem tudi najbolj pogosto uporabljeni tipi terena v simulacijah in računalniških igrah, ki lahko skupaj s primernimi tehnikami omogočajo prikaz kvalitetne scene. Vsi ti tipi tekstur so vključeni v obliki individualnih osnovnih barvnih tekstur [6].

Podrobna tekstura (angl. detail texture) je po navadi sivinska slika (slika 5.8), ki omogoča gladko prileganje in je uporabljena za dodajanje kompleksnosti na preproste nizko kvalitetne osnovne barvne teksture. Vseh tekstur skupaj imamo le osem, pri čemer morajo biti vse poslane v sečilni program terena. V poglavju *Deformacija* GPU-implementacija uporablja teksturo za leplenje odmikov, ki jo je prav tako treba poslati v senčilni program. Čeprav že veliko grafičnih pospeševalnikov trenutno podpira tudi do 118 teksturnih enot, je vedno dobra praksa, da se uporabi čim manj teksturnih enot. Ena izmed dobrih rešitev je tudi uporaba podrobnih tekstur.

Slika 5.7: Pet različnih tekstur za večteksturni preliv terena

V naši implementaciji so podrobne teksture shranjene v enem kanalu 32-bitne teksture. 32-bitne teksture imajo 4 kanale, vključno z alfa kanalom, zato lahko naložimo seznam podrobnih tekstur za vse tipe terena v eno teksturo z uporabo vsakega od štirih kanalov, ki bo predstavljal podrobno sliko. To pomeni, da lahko teren izrišemo le s 6 teksturnimi enotami in še vedno zagotovimo dovolj podrobnosti za vsak tip terena. Vsaka osnovna barvna tekstura je prilepljena na teren le enkrat. Osnovne barvne teksture dajo zelo malo oblike na površini terena in v glavnem služijo za dodajanje podrobnosti k osnovni barvi ter celotnemu teksturnemu učinku. Podrobne teksture ne vsebujejo nobene informacije o barvi, ampak so prilepljene na teren v višji resoluciji, da dobimo občutek unikatnih detajlov na površini, ki se obnašajo kot maska za fragmentno barvo za osnovne barvne teksture.

V senčilni program je treba poslati tudi določeno količino podatkov v obliki spremenljivk tipa `uniform`, na podlagi katerih lahko potem vzorčimo piksle iz teksture in lažje kontroliramo izris. Spremenljivke tipa `uniform` predstavljajo vrsto parametrov, ki se med izvajanjem senčilnega programa ne spreminjajo. Pridobitev potrebnih podatkov za izris terena se izvrši s preprostim klicem v senčilni program, ki je bil izdelan posebej za ta projekt.

Slika 5.8: Primer podrobne teksture za travo

Podatki, ki jih pošljemo v senčilni program, nosijo informacije o dimenzijah mape, informacije o lestvici odmikov, informacije o resoluciji uporabljenih tekstur na terenu, stikalo za preklop med izrisom tekstur ali barv in sedem teksturnih enot (Izsek kode: 5.3).

```
// dimenzije mape
glUniform1i(shader->mapWidthUniform,
                        elevTexY->Width());
glUniform1i(shader->mapHeightUniform,
                        elevTexY->Height());

// lestvica z višino odmikov
glUniform1i(shader->solidUniform, solid);
// lestvica za višino odmikov
glUniform1i(shader->scaleTexFacUniform, scaleTexFac);

//voda
glActiveTexture(GL_TEXTURE0);
water->GLBind();
glUniform1i(shader->waterSamplerUniform,0);
```

```
//pesek
glActiveTexture(GL_TEXTURE1);
sand->GLBind();
glUniform1i(shader->sandSamplerUniform,1);
//trava
glActiveTexture(GL_TEXTURE2);
water->GLBind();
glUniform1i(shader->grassSamplerUniform,2);
//kamenje
glActiveTexture(GL_TEXTURE3);
rock->GLBind();
glUniform1i(shader->rockSamplerUniform,3);
//sneg
glActiveTexture(GL_TEXTURE4);
snow->GLBind();
glUniform1i(shader->snowSamplerUniform,4);

// slika, ki vsebuje odmike višine
glActiveTexture(GL_TEXTURE5)
elevTexY->GLBind();
glUniform1i(shader->yHightmapSampler,5);

// oglišča (x, z)
verticesXZ->GLBind();
glVertexAttribPointer(shader->xzPositionAttribute
        ,2,GL_FLOAT,GL_FALSE,2 * sizeof(GLfloat),0);
```

Izsek kode 5.3: Prenos podatkov v senčilni program

Imena parametrov (Izsek kode: 5.4), ki so poslana, se morajo ujemati s tistimi v senčilnemu programu, da se jih lahko potem uporablja podobno kot konstante v C-ju.

```
// parametri v ogliščnem modelu senčilnika
```

```
#version 130
attribute vec2 XZPosition;
uniform sampler2D YHightmap;
uniform int MapWidth;
unoform int MapHeight;


// parametri v fragmentnem modelu senčilnika
#version 130
varying vec4 OutputColor;
varying vec2 TexCoordsOut;
varying vec3 NormalsOut;
varying float DisplacementOut;

uniform bool Solid = false;
uniform float ScaleTexFac = 100.0f;
uniform sampler2D Detail;
uniform sampler2D Sand;
uniform sampler2D Grass;
uniform sampler2D Rock;
uniform sampler2D Snow;
uniform sampler2D Water;
```

Izsek kode 5.4: Parametri v senčilnem programu

Kot je lahko razvidno iz kode 5.4, so vse podrobne slike, razen vode, shranjene v eno teksturo `Detail` in jih lahko pridobimo z vzorčenjem rdeče, zelene, modre ali alfa barve.

Preden izrišemo teksture na terenu, jih moramo najprej pomnožiti z osnovnimi barvnimi teksturami in z barvo, ki smo jo izračunali pri osvetlitvi. Pri izboljšanju kvalitete pri upodobitvi terena si lahko pomagamo tudi s teksturnimi utežmi, ki določajo delež vsakega izmed štirih vrst tipov terena na določeni lokaciji mape. Poleg tega nam omogočijo teksturni preliv

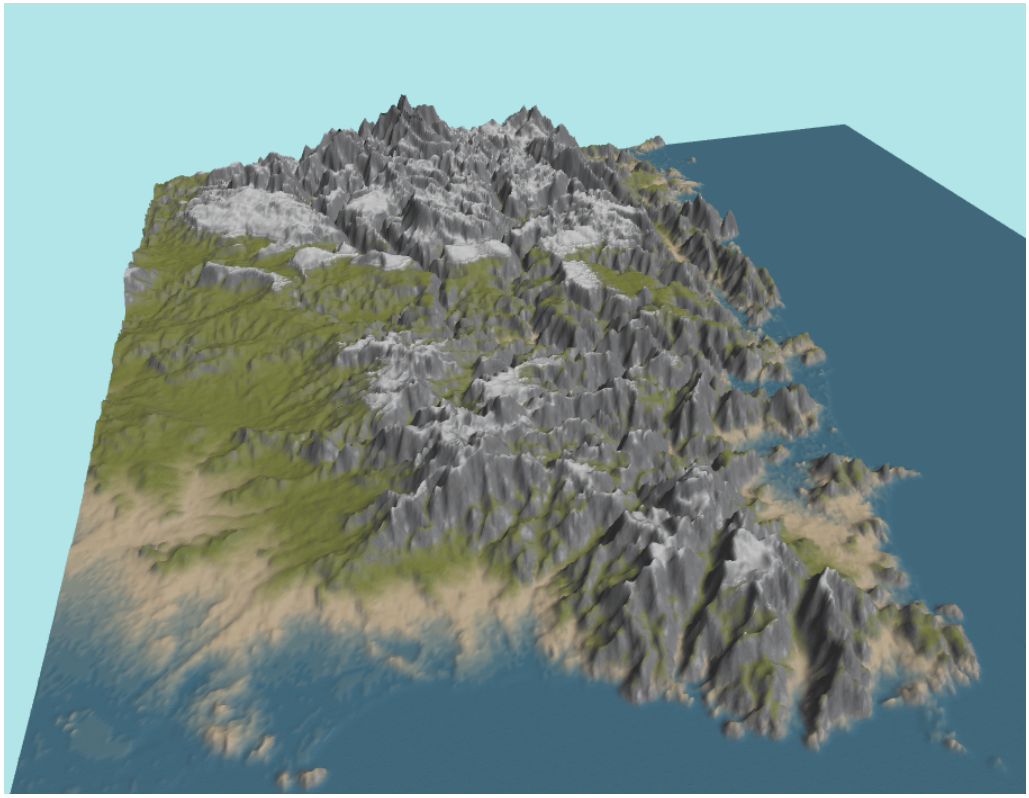
med dvema različnima tipoma terena na mapi. Teksturne uteži smo uporabili v fragmentnem modelu senčilnika v obliki linearne interpolacije med posameznimi tipi terena, in sicer v obliki tranzicije med višinskimi pasovi na mapi in strmih pobočij.

Za branje pikslov iz teksture v fragmentnem modelu senčilnika moramo uporabiti funkcijo `texture`. Funkcija `texture` prejme teksturni koordinati u in v na vhodu ter eno teksturno enoto tipa `sampler2D`, v zameno pa vrne barvo oziroma piksel iz izbranega dela teksture. Na terenu so teksturne koordinate uporabljene v skupaj z osnovnimi barvnimi in podrobnimi teksturami pri vzorčenju pikslov iz posameznega tipa terena. V naši implementciji smo izdelali tudi način za dinamično lepljenje tekstur na terenu, v povezavi z deformacijo terena, da so lahko teksture tudi primerno osvežene. Tudi teksturne uteži so izračunane glede na dinamiko terena.

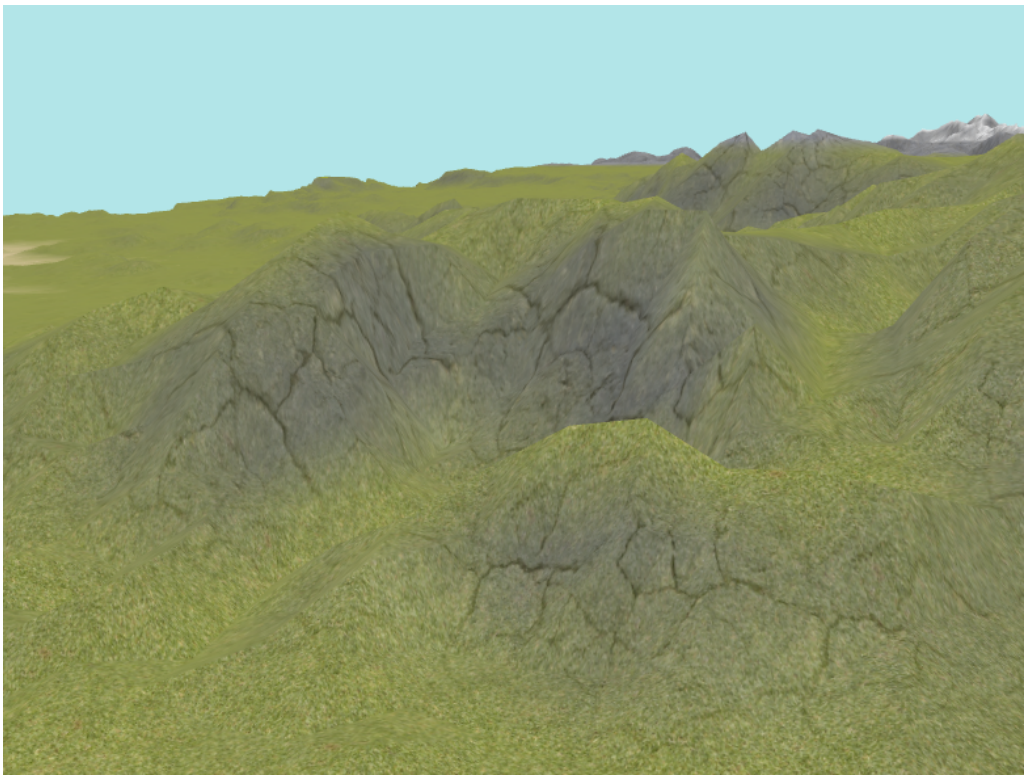
5.7.2 Višina in naklon terena

Višina terena je zelo pomemben dejavnik pri izračunu teksturnih uteži. Posamezen tip terena zavzame četrtino višine za vsako oglišče na terenu. Na terenu so od najnižje do najvišje točke razvrščene teksture za vodo, pesek, travo, kamenje in sneg (slika 5.9).

Takšna razvrstitev omogoča, da se lažje po pasovih razloči nižine od goratih območij. V ogliščnem senčilniku so opravljeni vsi potrebni izračuni, pri čemer moramo najprej normalizirati odmik za višino vsakega oglišča na terenu, na podlagi česar se nato določi, kateri tip terena bo prikazan. Ker je to enostaven pristop za večteksturni preliv terena, se lahko pokaže tudi veliko nepredvidljivih napak (angl. artefacts). Da bi zmanjšali uniformnost terena, je treba upoštevati tudi naklon. Naklon vsakega poligona je določen z oglišči, ki ga sestavljajo, in se ga izračuna (slika 5.10) v ogliščnem senčilniku, v katerem so izračunane tudi normale. Ker smo normale že izračunali za osvetlitev, jih lahko tudi tokrat uporabimo za prikaz klančin na terenu. Y-komponenta bo bližje 1, kadar bo površina bolj ravna, kadar pa bo bližje pravokotnemu kotu, bo vrednost bližje 0. Pregib terena naj bi imel negativno vrednost,



Slika 5.9: Večteksturni preliv terena po posameznih plasteh



Slika 5.10: Naklon terena in teksturni preliv

čeprav to ni izvedljivo na enotski mreži terena [6].

5.8 Deformacija

Glavni cilj v našem projektu je tudi izdelava deformacije terena v realnem času z uporabo grafične procesne enote. V projektu lahko različne oblike deformacije razdelimo na dva dela.

V prvem imamo različne oblike deformacije, izvedene na podlagi izračunov, ki se jih na podani lokaciji terena izvede s pomočjo grafične procesne enote. Drugi primer pa temelji na kopiranju podatkov o odmikih s teksture, na kateri imamo lahko poljuben vzorec, kot je na primer oblika kraterja, in se jih nato v senčilnem programu uporabi pri vzorčenju na določeni lokaciji terena.

Največja razlika med omenjenima pristopoma je v porabi pomnilnika in računske moči. Ker moramo v prvem primeru učinek deformacije izračunati v senčilnem programu, to pomeni tudi, da potrebujemo veliko več dodatne računske moči. Drugi način pa je precej enostavnejši od prvega, ker se zahteva, da se le prekopira vzorec z odmiki na določenem mestu terena. Posledično je potem tudi poraba pomnilnika veliko večja, ker je treba za vsak nov vzorec izdelati novo sliko, ki se jo nato naloži v program. Poleg tega pa zahteva tudi čas, ki je namenjen grafičnemu oblikovanju. Skupno pri obeh pristopih je to, da je treba podati tudi lokacijo na mreži terena, na kateri se bo zgodila deformacija, oba primera pa temeljita zgolj na procesiranju na grafični kartici. Prednost procesiranja na grafični kartici je v tem, da lahko deformacijo na terenu izvedemo povsem neodvisno od centralnega procesorja s pomočjo senčilnega programa na grafični procesni enoti, ki nam vrne hitrejše rezulatate pri večjem številu oglišč.

Za učinkovito izvedbo deformacije na grafični procesni enoti je treba uporabiti več stopenj izrisa, ki se izvršijo v teksturo s pomočjo objektov OpenGL tipa FBO. Pristop, ki smo ga uporabili v tem primeru, je sestavljen iz kombinacije FBO-jev, teksturnih objektov in senčilnega programa. Vse nove spremembe pri odmikih oglišč, normal in osvetlitve so izvedene v realnem času na grafični procesni enoti. To nam precej razbremeni centralni procesor, zmanjša prenos podatkov iz glavnega pomnilnika in omogoči, da se lahko na enostaven način izvede deformacija na mreži terena brez uporabe kompleksnih algoritmov na centralnem procesorju.

Osnovna ideja je, da lahko podatke o odmikih višine na terenu enostavno in hitro spreminjamo s pomočjo ogliščnega modela senčilnika, pri tem pa si lahko pomagamo s tehniko, ki omogoča branje teksture v ogliščnem modelu senčilnika (angl. Vertex Texture Fetch). Kot je že bilo rečeno, je vsa deformacija izvedena na teksturi z več stopnjami izrisa in uporabo objektov OpenGL tipa FBO. Ti nam bodo omogočili, da bomo izris iz okna aplikacije lahko preusmerili v teksturo. Preden gremo na izris v teksturo, moramo pred tem v FBO-objekt pripeti teksturni objekt, v katerega bomo kasneje

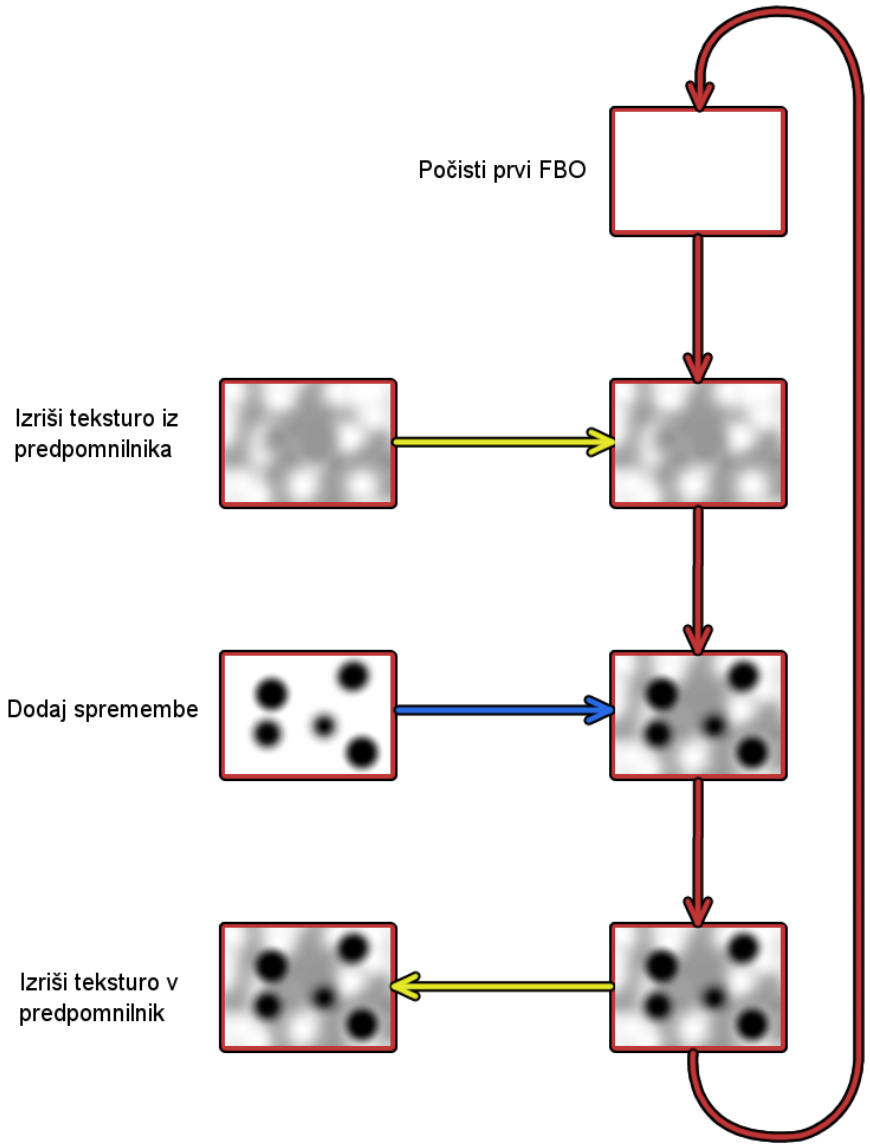
izrisovali podatke o odmikih višine. Pri tem pa nas nič ne ovira, da imamo v teksturnem objektu že prednaložene podatke z odmiki, ki jih bo mogoče preko novih stopenj izrisa še dodatno spreminjati. V vsaki stopnji se bodo ti podatki menjavali oziroma dopolnjevali.

Objekt terena vsebuje dva FBO-objekta, ki imata ločeno pripeto vsak svojo teksturo za shranjevanje podatkov o odmikih in nam bosta služila kot prostor za izris novih odmikov v teksturo. Na teksturi, ki je pripeta na prvi FBO-objekt, so shranjeni vsi novi podatki s sveže naloženimi podatki o odmikih na trenutni stopnji izrisa. Medtem pa drugi FBO služi kot predpomnilnik oziroma prostor, kjer bodo shranjeni vsi odmiki po končanem izrisu iz prvega FBO-ja. Teksturo v drugem FBO-ju lahko uporabimo kot medpomnilnik, ki vsebuje najnovejše oziroma zadnje podatke o odmikih v višino, in bo poslana na vzorčenje v končnem izrisu terena.

Takšna zasnova nam omogoča simuliranje dinamične deformacije terena z več plastmi v enem enem izrisovalnem obhodu. Ko želimo simulirati deformacijo terena z eksplozijami kraterjev, se tega, na primer, lahko lotimo tako, da v izris večkrat zaporedoma pošljemo poljubno število tekstur, ki se jih pošlje na izris v obliki plasti. Med izrisom se bodo te plasti tekstur izrisale ena vrh druge, odvisno od podanega zaporedja, in kot končni produkt bomo dobili v prvem FBO-objektu teksturo, ki je kombinacija večteksturnih plasti z odmiki oziroma, v tem primeru, kraterjev.

Ko smo zaključili z izrisom v prvi FBO ter z vsemi novimi odmiki v teksturi, je treba vse skupaj iz prvega FBO-ja izrisati še v drugi FBO. Drugi FBO služi kot predpomnilnik, v katerem so naloženi vsi najnovejši podatki o odmikih, in kot zadnja verzija teksture s podatki o odmikih višine, ki se bo kot končni produkt poslala v izris skupaj s terenom na sceno. Po končanem izrisu v drugem FBO-ju moramo počistiti prvega, da lahko nato ob ponovnem izrisu iz predpomnilnika v prvi FBO ponovno izrišemo zadnje podatke o odmikih. Nato pa izrišemo vse nove odmike preko prejšnih in celoten proces samo še ponovimo (slika 5.11).

S tem smo si zagotovili, da so odmiki shranjeni in so kasneje v novem



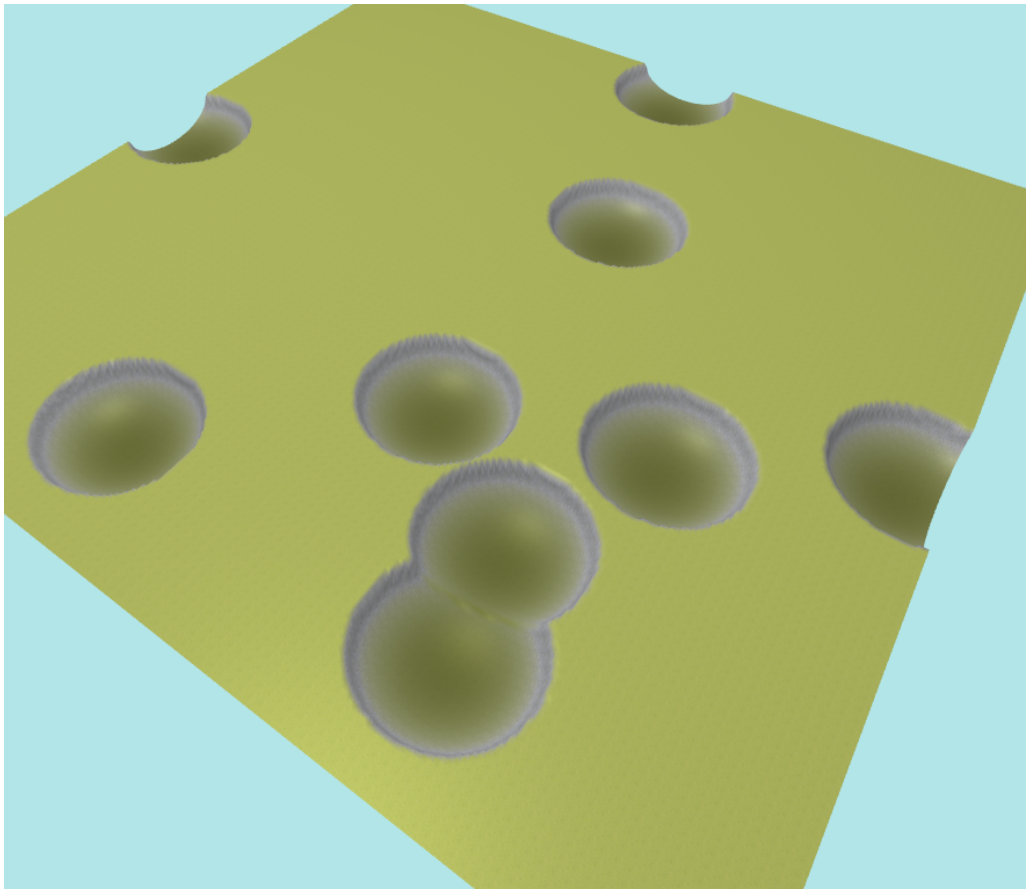
Slika 5.11: Proces izrisovanja odmikov v FBO

izrisovalnem obhodu lahko brez težav spet ponovno uporabljeni. Na sliki 5.11 je na desni strani prikazan postopek izrisa v prvi FBO. Ko je proces izrisa zaključen, je treba teksturo samo še shraniti v predpomnilnik (drugi FBO), iz katerega bomo podatke v naslednjem koraku spet uporabili za izris.

Ob zaključku je tekstura poslana iz predpomnilnika na vzorčenje v glavni program senčilnika kot tip `sampler2D`, in sicer v ogliščni model senčilnika. Glavni program senčilnika je tako tudi zadnja stopnja pri izrisu terena. V programu so implementirane funkcije za izris terena, kot so izračun osvetlitve, normal, teksturnih koordinat in večteksturni preliv.

Za izris novih odmikov v prvi FBO smo uporabili poseben senčilni program, katerega naloga bo odvisna od uporabe pristopa, s katerim bomo izrisovali odmike v prvi FBO. To pomeni, da se bo senčilni program, ki nam bo omogočil branje odmikov iz teksture in jih nato izrisal v prvi FBO, razlikoval od tistega, ki nam bo izračunal deformacijo. V ogliščni model senčilnika so poslane tudi štiri koordinate, ki predstavljajo obliko štirikotnega poligona, sestavljenega iz dveh trikotnikov, na podlagi katerih bo mogoče prilepiti teksturo ali izračunati odmike v fragmentnem modelu senčilnika. V izseku 5.5 je prikazana koda za izračun odmikov v fragmentnem modelu senčilnika, ki nam izriše v teksturo z odmiki višine preprosto obliko kraterja (slika 5.12):

```
vec3 V;
V.x = VertexOut.x;
// preberemo y-lokacijo iz teksture, ki vsebuje odmike
V.y = texture(TextureSampler,TexCoordOut).r;
V.z = VertexOut.y;
// izračunamo razdaljo v prostoru med trenutno točko
// in lokacijo centra kraterja
V -= Location;
float dist3 = dot(V,V);
// preverimo, ali se točka nahaja v obsegu radija
if (dist3 <= Radius * Radius) {
    // normaliziramo 2D-razdaljo med trenutno pozicijo (točko) in
```



Slika 5.12: Izračun kraterjev na mreži terena

```
    // lokacijo kraterja
    float dist2 = sqrt(V.x * V.x + V.z * V.z);
    // razmaknemo vse točke, ki so znotraj kraterja, na rob
    float deg = acos( dist2 / Radius );
    float Y = -sin(deg) * Radius + Location.y;
    // osvežimo trenutni piksel oziroma odmik na terenu
    gl_FragColor = vec4(Y,0,0,1);
}
else
    // če ni v obsegu kraterja, opustimo
    discard;
```

Izsek kode 5.5: GLSL-koda za izračun kraterja

Kot je lahko razvidno v kodi 5.5, se odmike pridobi iz teksture na podlagi rdeče komponente, iz katere nato sestavimo oglišča skupaj s pomočjo X- in Z-lokacije, ki sta podani kot atributa oglišč teksture. Ko smo zaključili z izrisovanjem v teksturo, jo je treba poslati še v končni izris terena na sceni. Ko hočemo izris preusmetiti nazaj na zaslon, to lahko izvedemo z ukazom `glBindFramebuffer`, ki seprejme vrednost 0 , da nam preusmeri izris nazaj v okno aplikacije.

Kadar želimo izvesti deformacijo z uporabo tekstur, moramo najprej izbrano teksturo naložiti v senčilni program, ki smo ga v tem primeru posebej izdelali. Senčilni program v fragmentnem modelu senčilnika lahko sprejme naenkrat le eno teksturo z odmiki. Teksturo se polšlje v uniformno spremenljivko `sampler2DRect`, ki lahko za razliko od klasičnega `sampler2D`, ki sprejme koordinate med 0 in 1, sprejme dejanske koordinate v pikslih. Na podlagi teh koordinat je potem mogoče prebrati odmike s teksture in jih nato prilepiti na določeno mesto na terenu. Poleg tega so v senčilnik poslane še dimenzije, ki nam povedo širino in višino teksture, na katero bomo prilepili novo. V izseku kode 5.6 je prikazano lepljenje tekstur v fragmentnem modelu senčilnika (slika 5.13):

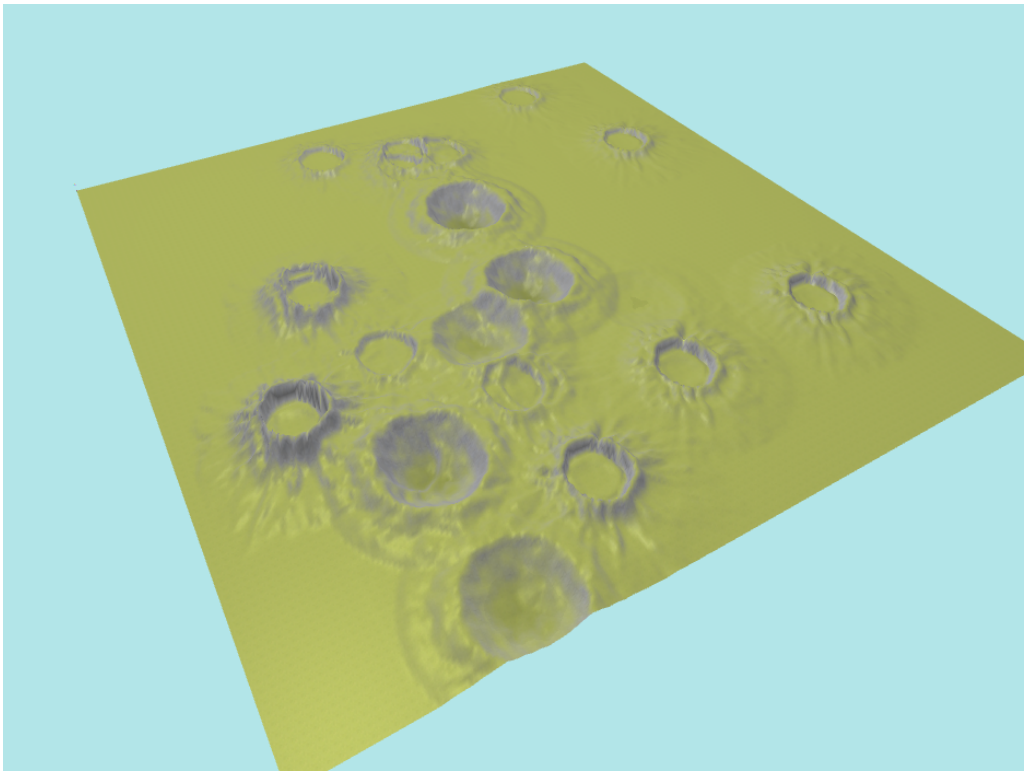
```
// pridobimo X- in Z-koordinati
```

```
float X = CoordsOut.x * Dimensions.x;
float Z = CoordsOut.y * Dimensions.y;

// centriramo sliko
flaot halfDimX = TextureInDimensions.x/2.0f;
flaot halfDimZ = TextureInDimensions.y/2.0f;

// preverimo, ali se koordinati nahajata v območju teksture,
// kjer naj bi se prilepilo sliko
if (Location.x - halfDimX <= X && X <= Location.x + halfDimX &&
        Location.z - halfDimZ <= Z && Z <= Location.z + halfDimZ) {
        // preberemo odmik s teksture
        float Y = texture(TextureIn, vec2(X-Location.x+halfDimX,
        Z-Location.z+halfDimZ)).r

        // če je vrednost večja od 0, izrišemo piksel,
        // drugače pa pustimo
        if (Y > 0.0f) {
            Y = Y * Scale + Location.y
            glFragColor = vec4(Y,0,0,1)
            return;
        }
}
discard;
```

Izsek kode 5.6: GLSL-koda za lepljenje odmikov s teksturo

5.9 Interakcija

Da bi lahko testirali različne oblike deformacije na mreži terena, je bilo treba razviti način, ki nam bo omogočil testiranje različnih oblik deformacije na terenu. Poleg tega je bilo treba izdelati tudi osnovno premikanje kamere v



Slika 5.13: Izris kraterjev na mreži terena na podlagi vzorcev

3D-svetu simulacije. Uporabnik lahko kontrolira premikanje kamere z vhodnimi podatki s tipkovnice in lahko povzroči deformacijo s klikom miške na določenem mestu terena.

5.9.1 Interakcija s terenom

Interkacija s terenom je v simulaciji izvedena z uporabo orodij, ki omogočajo, da lahko uporabnik s klikom miške na zaslonu izbere mesto deformacije na terenu, ki se nahaja v območju perspektivne projekcije. S premico, ki je usmerjena iz smeri gledanja kamere pa vse do konca vidnega polja, se preveri mesto preseka na mreži terena. Tam, kjer se presekata daljica in poligon, se izvrši deformacijo na terenu. Da lahko določimo daljico v prostoru, potrebujemo dve točki iz bližnje in daljne projekcijske ravnine. Kako se to izračuna, je prikazano v izseku kode 5.8.

```
// za izračun točk bomo potrebovali parametre
// iz bližnje in daljne projekcijske ravnine
float npw, fpw, nph, fph, nz, fz;
render->GetFrustumParameters(
        npw,    // dolžina bližnje proj. rav.
        fpw,    // dolžina daljne proj. rav.
        nph,    // višina bližnje proj. rav.
        fph,    // višina daljne proj. rav.
        nz,             // globina bližnje proj. rav.
        fz              // globina daljne proj. rav.
);
// potrebujemo tudi višino in širino okna
float sw,sh;
render->GetDisplaySize( sw, sh );

// izračun koordinat na bližnji in daljni proj. ravnine
// iz miškinih koordinat na zaslonu (mouseX, mouseY)
float nearX = 0, nearY = 0;
```

```
if ( now_controling == CMOUSE ) {
        nearX = 2*fpw * ((float)mouseX) /
                                (sw-1) + npw;
        nearY = -(2*fph * ((float)mouseY) /
                                (sh-1) + nph);
}
float farX = nearX * fz;
float farY = nearY * fz;
float vert = -camera.getVertRotDegrees();
float horiz = -camera.getHorzRotDegrees();

// izračun točk na bližnji in daljni proj. rav.
// dobimo z množenjem vektorja z obema vektorjema
// in matriko, ki predstavlja orienatacijo kamere
a_npoint = ( camera.GetOrientation()
                        * vec3( nearX, nearY, 0 ) )
                        + camera.GetPosition();
a_fpoint = ( camera.GetOrientation()
                        * vec3( farX, farY, nz-fz ) )
                        + camera.GetPosition();
```

Izsek kode 5.7: Izračun daljice iz bližnje in daljne projekcijske ravnine

Če sta obe točki, ki določata daljico, nad ali pod mrežo terena, ni treba dodatno preverjati preseka z mrežo, v nasprotnem primeru pa moramo preveriti, s katerim poligonom na mreži je presekana daljica, kar lahko učinkovito storimo z uporabo štiriškega drevesa (angl. Quadtree), kar je opisano v naslednjem poglavju.

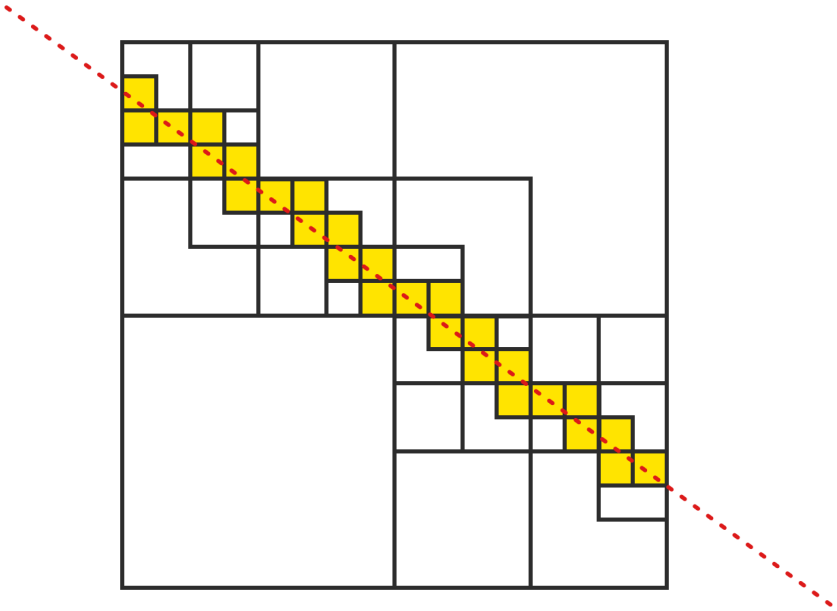
5.9.2 Štiriško drevo in presek daljice z mrežo terena

Cilj v štiriškem drevesu (slika 5.14) je izgradnja strukture v obliki drevesa, ki nam bo omogočila prostor rekuzivno razdeliti na štiri enake dele. Vsak

štirikotnik pa se bo dalo še naprej rekurzivno razdeliti na štiri enake dele do določene globine. Pri izgradnji algoritma je treba upoštevati pogoj, da so meje začetnega štirikotnika pogojene z dimenzijo območja, na katerem smo uporabili algoritem. V tem primeru bodo te meje pridobljene iz enotske mreže, ki je bila izdelana v ta namen. Na enotski mreži sta dimenziji višine in dolžine v trodimenzionalnem koordinatnem sistemu predstavljeni z X- in -koordinatama, ki bosta uporabljeni pri izdelavi štiriškega drevesa.

Ko vstavimo nek element v drevo algoritma, ga v bistvu vstavimo v kvadrat, ki obsega dimenzije tega elementa. Na vsaki stopnji posebej je v drevesu določena tudi maksimalna velikost vsebine v kvadratu. Kadar je ta vsebina presežena, je treba kvadrat še naprej razdeliti na štiri enake dele (podkvadrate), ki so v bistvu podvozlišča (child nodes) pri izhodiščnemu kvadratu (parent node). Če bomo ta postopek še naprej nadaljevali, bomo prišli do novih kvadratov, ki bi lahko že predstavljali nove liste v štiriškem drevesu. [14] Da bi lahko prišli do listov v drevesu, moramo preiskovanje drevesa na določeni globini tudi zaključiti. To pomeni, da je treba sestaviti pogoj, pri katerem bomo algoritem na določeni stopnji v drevesu tudi zaključili pri kvadratu, ki bi ustrezal podanim pogojem, in se nato vrnil na preiskovanje še preostalih poti. V našem primeru bomo s preiskovanjem poti zaključili takrat, ko dobljeni podštirikotnik ne bo pokrival več dela vstavljenega elementa ali ko bosta obe dimenziji štirikotnika, ki sta sestavljeni iz X- in Z-koordinat, na enotski mreži dolžine 1, kar bo pomenilo, da smo prišli do najgloblje točke v drevesu [40].

Glavni namen algoritma je pohitritev iskanja indeksov v pomnilniku s pomočjo koordinat na enotski mreži. Ti indeksi so pomembni, saj se na njihovi podlagi lahko hitro in predvsem v konstantnem času prebere koordinate najbližjih oglišč na določeni lokaciji terena iz pomnilnika. Indekse pridobimo na podlagi vozlišč kvadratov, katerih pozicijo določata X- in Z-komponenti na enotski mreži. Pri tem nam Z pove indeks vrstice, X pa nam pove indeks stolpca. Ker imamo podatke o indeksih v pomnilniku shranjene kot enodimenzionalni niz, moramo lokacije vozlišč kvadrata, ki smo ga dobili s



Slika 5.14: Uporaba štiriškega drevesa pri iskanju preseka premice

preverjanjem preseka s premico, ki je usmerjena iz izhodiščne točke projekcijske ravnine do konca vidnega polja, pretvoriti v enodimenzionalne podatke, da se jih potem lažje locira v pomnilniku. Tako lahko zelo hitro pridobimo podatke o ogliščih na mreži terena iz pomnilnika. Ker se ta premica nahaja v 3D-prostoru, pomeni, da jo moramo najprej preslikati še iz 3D-sveta v premico v 2D-svetu. To pa zato, da jo lahko potem projiciramo na enotsko mrežo terena in poišče vse nove kvadrate v drevesu, ki jih presaka. Preslikava premice iz 3D-sveta v 2D je precej enostavna, ker potrebujemo samo X- in Z-koordinati, Y, ki predstavlja višino, pa lahko zanemarimo [40].

5.9.3 Izdelava orodij za spreminjanje topologije terena

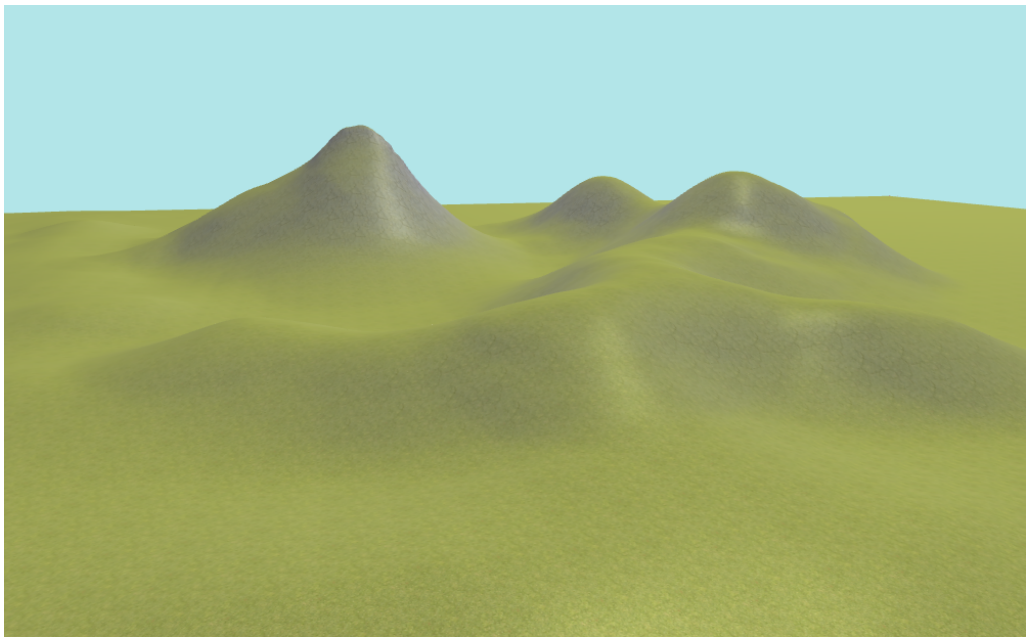
Za oblikovanje terena so bila izdelana tudi različna orodja, s katerimi si lahko pomagamo pri spreminjanju topologije z vdolbinami, izboklinami ali izravnavanjem terena. Orodja omogočajo spremembe v obliki odmikov oglišč na Y-osi z dodajanjem ali odštevanjem od začetne vrednosti. Na ta način lahko spreminjamo celotno podobno ali posamezne dele na površini terena, s čimer lahko dobimo želene oblike na terenu, kot so gore, kanjoni, hribovita področja, pobočja, kraterji itd. Da bi dobili želene oblike, moramo najprej izdelati osnovne tipe orodij. V ta namen so bili izdelani trije osnovni tipi orodij, ki so značilni za vsak urejevalnik terena v igri: uniformni dvig, stožčasti dvig in glajenje. V navedenih orodjih je obseg deformacije odvisen od radija, ki določa območje deformacije oziroma območje, kjer bodo nastali novi odmiki na mreži terena.

```
// preveri, če smo v obsegu radija
if (dist2 <= Radius * Radius) {
    switch(Tool) {

    // uniformni dvig
    case 0:
        gl_FragColor = vec4( texture( Hightmap, TexCoordOut ).r
        + Strength, 0,0,1 );
```

```
            return;

    // stožčasti dvig
    case 1:
        // izračunaj razdaljo od centra def. (x,z)
        float real_dist = sqrt( dist2 );

        // v odvisnosti od razdalje odmakni y
        float normal_inv_dist = real_dist == 0.0f ?
        1.0f : 1.0f - ( real_dist / Radius );

        gl_FragColor = vec4( texture( Hightmap, TexCoordOut ).r
        + normal_inv_dist * Strength ,0,0,1 );

        return;

        ...

    }
}
discard;
```

Izsek kode 5.8: Uniformni dvig in stožčasti dvig

Pri uniformnem dvigu se odmakne vsa vozlišča, ki se nahajajo v obsegu radija, enakomerno glede na poljubno vrednost. Stožčasti dvig (slika 5.15) pa nam odmakne najbolj vsa tista oglišča, ki so najbližje centru deformacije, tista, ki so oddaljena bolj stran, pa vedno manj. Odmike pri stožčastem dvigu se izračuna na podlagi razdalje med centrom deformacije in oddaljenostjo od točke. Bolj ko je točka oddaljena od centra, tem manjši je odmik in obratno. Pri glajenju pa je treba najprej izračunati povprečje okoliških točk in nato primerjati dobljeno povprečje z višino trenutno izbrane točke. Za izračun povprečja je uporabljenih 12 okoliških točk, iz katerih se nato dobi



Slika 5.15: Stožčasti dvig

skupno povprečje za vsako točko, ki se nahaja v dosegu radija. Če je višina točke manjša od povprečja, je treba prišteti določeno vrednost k višini, v nasprotnem primeru se jo odšteje.

Vsa orodja potrebujejo tudi različne parametre, s katerimi lahko nastavimo količino odmika v odvisnosti od časa, pozicijo na terenu in stikalo za preklapljanje med posameznimi tipi orodij. V osnovnih orodjih se upošteva samo radij v dveh dimenzijah, ki upošteva X- in Z-koordinati, Y pa ni uporabljena. To je smiselno v situaciji, ko imamo neko točko na terenu, ki je v višno veliko bolj oddaljena od preostalih točk, kar bi pomenilo, da bi ob upoštevanju še Y-komponente lahko deformirali le to točko, vse ostale točke, ki ležijo nižje, pa bi bile preprosto izpuščene, ker ne bi bile v dosegu radija. V ostalih oblikah deformacije, kot je deformacija terena s kraterji, pa je upoštevana Y-komponenta, da dobimo bolj enotno obliko kraterja na terenu.

5.9.4 Kamera

Kamera v simulaciji je prostorska in predstavlja usmerjen objekt, ki ga je mogoče premikati v 3D-prostoru z vhodnimi podatki s tipkovnice. Predstavlja značilno kamero, ki je lahko implementirana tudi v igrah.

Poglavje 6

Sklepne ugotovitve

Izris terena z uporabo višinske slike je enostavna in praktična rešitev za predstavitev terena v igri. V projektu smo pokazali, kako lahko na preprost način oblikujemo teren na podlagi slike, ki hrani odmike v višino oziroma podatke o višini posameznih oglišč na mreži terena. Ena izmed zelo preprostih metod za oblikovanje površine terena v ogliščnem modelu senčilnika, ki je bila prikazana, je na podlagi uvoza sivinske slike s prednaloženimi podatki višine z diska, ki lahko predstavljajo poljubno obliko na terenu. Podatke imamo lahko shranjene v enem izmed slikovnih formatov, ki ga ob inicializaciji terena dekodiramo in prenesemo vsebino v teksturo. Ker je slika shranjena v enem kanalu, ima zato na voljo le 8 bitov, kar omogoča prikaz največ 256 različnih slojev na terenu. To pa ni dovolj za kvaliteten prikaz terena, saj lahko hitro pride do izraza pri bolj strmem naklonu, ker ne zagotovi dovolj visoke ločljivosti, s katero bi dobili mehkejši prehod med posameznimi sloji terena. To težavo bi se dalo rešiti z uporabo vseh kanalov v slikovnem formatu, vključno z alfa kanalom, ki bi skupaj zagotovili dovolj prostora za shranjevanje podatkov v celoštevilskem podatkovnem formatu.

Še pred nekaj leti je bilo dostopanje do teksturnih podatkov v ogliščnem modelu senčilnika nemogoče, ker so morali biti podatki o odmikih shranjeni v obliki niza v medpomnilniku, ki se ga je pred izrisom poslalo v senčilni program. Težava pri takšnem pristopu je v tem, da ga ni bilo mogoče ne-

posredno spreminjati na grafični procesni enoti, prav tako pa ni bilo mogoče dostopati do podatkov v sosednjih ogliščih. Danes je dostop do teksturnih podatkov v ogliščnem modelu senčilnika že obvezen pristop v večini iger. Tako so se odprle nove možnosti pri uporabi senčilnega programa, kot sta izračun normal na grafični procesni enoti in možnost boljšega zevedanja o sosednjih ogliščih. Z uporabo programabilnega cevovoda na grafični kartici si lahko pohitrimo izvajanje igre, s čimer se izognemo porabi dodatnih ciklov na centralnem procesorju, hkrati pa lahko v tem času izkoristimo procesor za ostale stvari, kot sta fizika ali umetna inteligenca v igri. V našem projektu smo to izkoristili pri sprotnem izračunu mehkih normal za dinamično osvetlitev terena in za teksturni preliv pri naklonu terena na strmih pobočjih. Poleg tega smo si tudi pri realno-časovni deformaciji terena pomagali z orodji, ki so bila razvita izključno za ta projekt v senčilnem programu. Pokazali smo tudi simulacijo deformacije na terenu s kraterji in večstopenjski izris v teksturo, ki ga je bilo mogoče izračunati ali prilepiti na podlagi odmikov, pridobljenih iz teksture, ki jo je bilo mogoče v enem obhodu izrisa spremeniti v celoti na vsej površini terena brez večje izgube v hitrosti.

Izbrani pristop je lahko precej omejujoč v primeru, da želimo na terenu prikazati pregibe ali zelo strma pobočja. Če hočemo dostopati do podatkov v sosednjih ogliščih v ogliščnem modelu senčilnika, jih je najprej treba organizirati v obliki enotske mreže, na kateri se lahko potem s pomočjo koordinat določi sosednja vozlišča. Organizacija v obliki enotske mreže pomeni, da morajo biti vsa vozlišča enakomerno porazdeljena med seboj, kar pomeni, da jih ni mogoče spreminjati v X- in Z-smeri, da se lahko pri branju sosednjih oglišč iz teksturnih koordinat določi Y-koordinato na podlagi X- in Z-koordinat. Kot je bilo že rečeno, zaradi tega ne moramo uporabiti pregiba na terenu in predorov ali zelo strmih območij. Praktične rešitve, ki bi nam enostavno rešila to težavo, ni. Programerji računalniških iger morajo v izogib podobnim težavam poiskati druge rešitve, ki na mestih, kjer naj bi bile luknje na terenu, le odstranijo odvečne poligone ali pa dodajo še dodatno plast geometrije, s katero lahko ustvarijo preklon.

Pogledali smo si tudi način za izvedbo teksturnega preliva na terenu na podlagi odmikov z uporabo petih različnih tekstur, z večteksturnim prelivom med stopnjami, hkrati pa smo pogledali, kako lahko ponazorimo teksture na poligon v odvisnosti od naklona terena, ki ga definirajo normale. Da bi prikazali več podrobnosti na terenu in zmanjšali občutek ponavljajočih vzorcev, smo uporabili detajlne teksture. Tukaj je še veliko možnosti za izboljšanje celotne podobe terena. Naslednja stopnja pri izboljšanju vizualne podobe terena bi lahko bila uporaba tehnike lepljenja izboklin (angl. bump mapping), s katero bi lahko dodali občutek senc za detajle k teksturi terena. Za izboljšanje podrobnosti bi lahko pri vsakem tipu teksture z osnovno barvo terena vsaj dvakrat, z namenom združitve osnovnih barvnih tekstur skupaj kot nalaganje ene vrh druge pri različnih lestvicah uv-koordinat, nato popravili njeno začetno svetlost. To bi nam še dodatno zmanjšalo občutek ponavljajočih se tekstur na terenu in dobili bi občutek, da je vsak del terena unikaten.

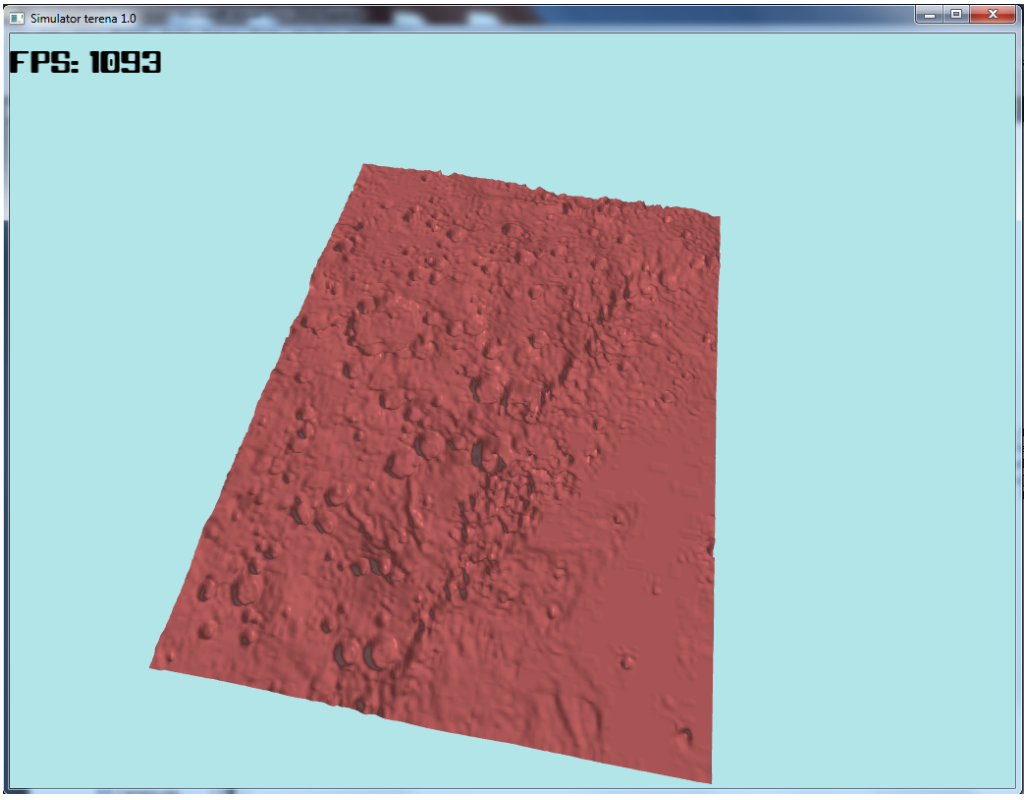
Ker smo v tej implementaciji omejeni le na določeno velikost površja, ki ga lahko simuliramo, ne moremo simulirati zelo velikih map s terenom. Največja mapa, ki smo jo lahko izrisali pri zadovoljivem številu sličic na sekundo (30-40 FPS), je bila dimenzije 1024×2048 slikovnih pik. V nadaljevanju bi lahko nadgradili simulacijo terena do naslednje stopnje, ki bi lahko izrisala praktično neomejeno površino terena, ki bi bila sestavljena iz ogromnega števila poligonov. Za izris velikega števila poligonov moramo poiskati primeren LOD-algoritem, ki nam bo odstranil določeno število nepotrebnih poligonov, ki so toliko oddaljeni, da so izven našega vidnega polja, in bi ohranil le tiste, ki so nam najbližji. Težava pri LOD-algoritmih je v tem, da zahtevajo dodatne CPU-cikle, ker morajo s trenutne točke na terenu določiti, kateri poligoni bodo šli na izris in pri kakšni kvaliteti. Nato pa morajo vse skupaj prenesti v pomnilnik na grafični kartici, kar zna biti zelo potratno. Zato je izbira ustreznega algoritma nadvse pomembna in je odvisna od predstavitve terena ter zahtev aplikacije.

S hitrim razvojem grafičnih pospeševalnikov se odpirajo nove možnosti

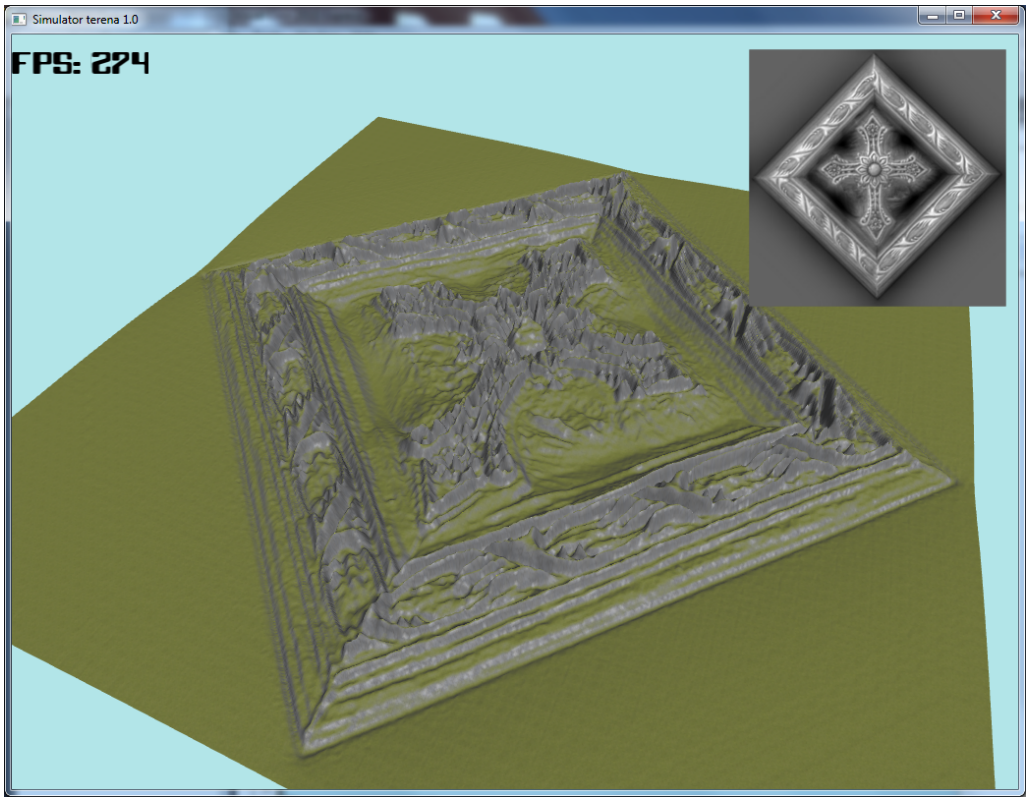
pri simulaciji in deformaciji terena, ki omogočajo, da lahko veliko stvari, ki so povezane z izrisom terena, opravimo že v senčilnem programu.

Priloga – zaslonski posnetki simulacije

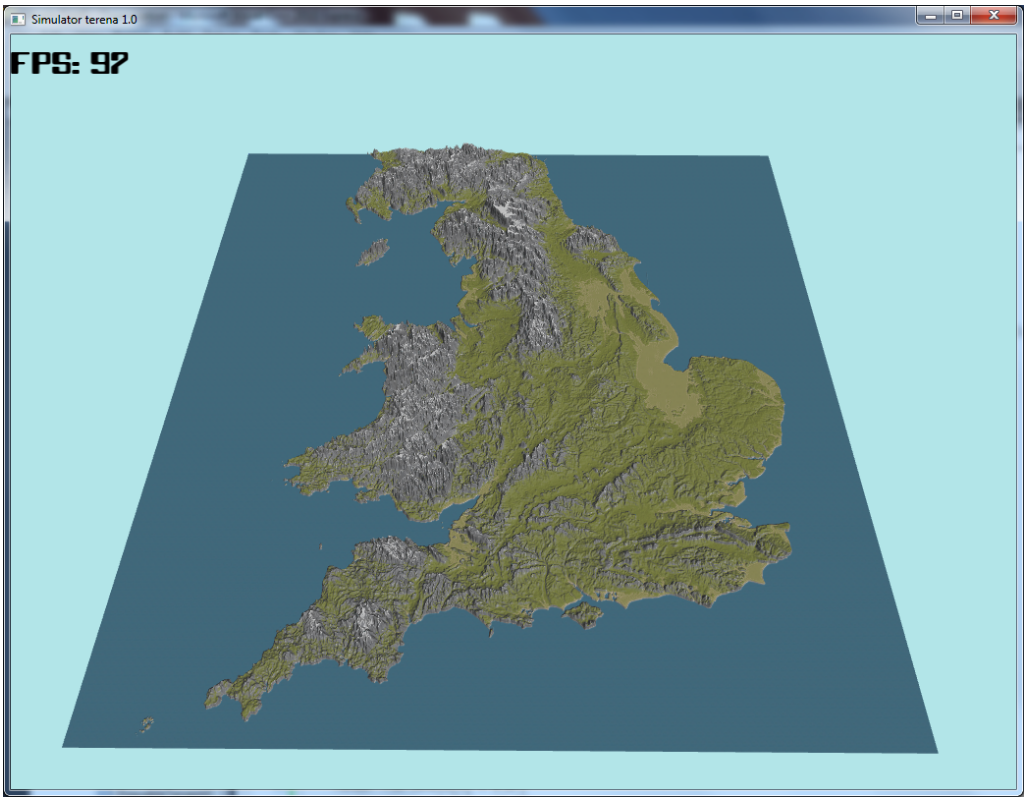
Na slikah 6.1 – 6.3 je prikazana ena izmed prvih implementacij terena z odmiki in senčenjem, uporaba teksturnega preliva na strmejših pobočjih (slika 6.2), izvedba večteksturnega preliva za vizualni prikaz posameznih plasti (slika 6.3) in deformacija terena (slika 6.4).



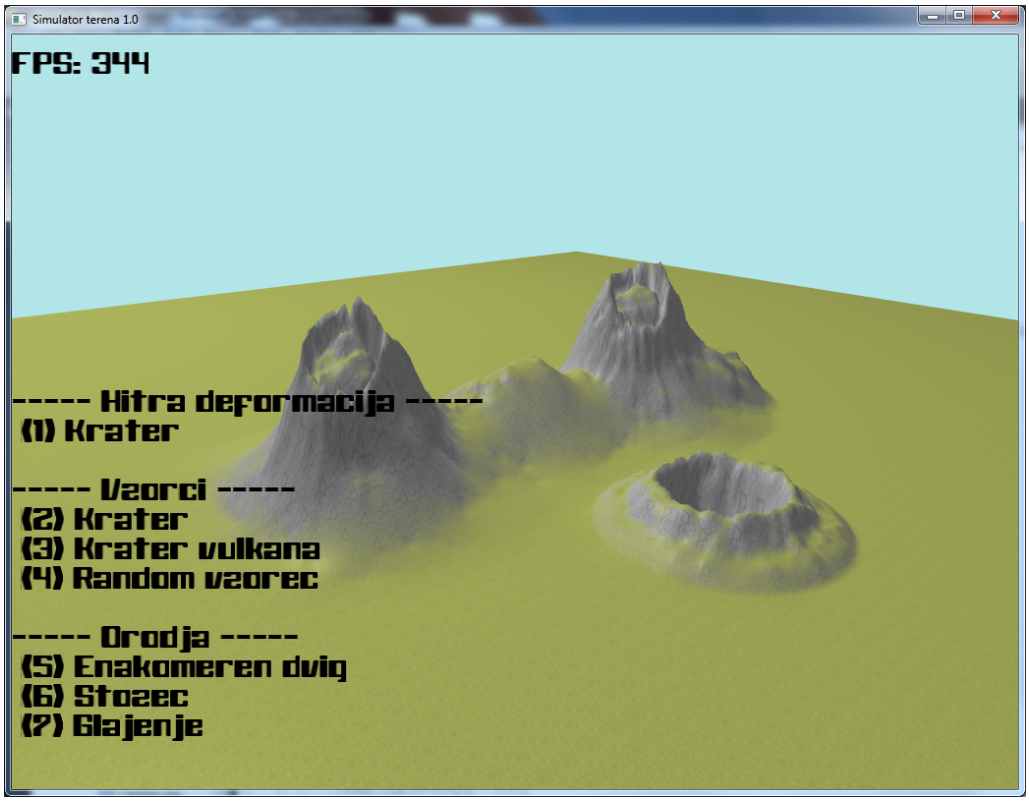
Slika 6.1: Površina Marsa



Slika 6.2: Testiranje odmikov iz sivinske slike



Slika 6.3: Izris britanskega otočja, dimenzije 1024 × 1024



Slika 6.4: Eno izmed prvih testiranj orodja in deformacije na terenu

Literatura

[1] (2012) Zaxxon. Dostopno na:
http://en.wikipedia.org/wiki/Zaxxon

[2] (2012) Elite (video game). Dostopno na:
http://en.wikipedia.org/wiki/Elite

[3] (2012) Popolous. Dostopno na:
http://en.wikipedia.org/wiki/Populous

[4] (2013) Real-time strategy. Dostopno na:
http://en.wikipedia.org/wiki/Real-time_strategy

[5] (2011) From Dust Review. Dostopno na:
http://www.gamespot.com/from-dust/reviews/from-dust-review-6325105/

[6] A. Bleasdale, *Interactive terrain simulation and deformation for computer games*, Liverpool John Moores University, 2009.

[7] (2004) Terrain Rendering Using GPU-Based Geometry Clipmaps. Dostopno na:
http://research.microsoft.com/en-us/um/people/hoppe/gpugcm.pdf

[8] M. White (2007) Real-Time Optimally Adapting Meshes: Terrain Visualization in Games. Dostopno na:
http://www.classes.cs.uchicago.edu/archive/2003/fall/23700/docs/roam.pdf

[9] M. Duchaineau, M. Wolinsky, D. Sigeti, M. Miller, C. Alrich, M. Mineev-Weinstein. *ROAMing terrain: real-time optimally adapting meshes*. Lawrence Livermore National Laboratory, Livermore, Kalifornija, ZDA, Julij 1997.

[10] D. Wang, C. Wang. *Real-time GPU-based Simulation of Dynamic Terrain in Virtual Battlefield*. Digital Engineer&Simulation Research Center, Huazhong University of Science and Technology, Wuhan 430000, Kitajska, 2011.

[11] N. Glasser (2008) Texture Splatting in Direct3D. Dostopno na: http://archive.gamedev.net/archive/reference/articles/article2238.html

[12] (2012) Knjižnica. Dostopno na: http://sl.wikipedia.org/wiki/Knji%C5%BEnica_%28ra%C4%8Dunalni%C5%A1tvo%29

[13] (2007) A.1 — Static and dynamic libraries. Dostopno na: http://www.learncpp.com/cpp-tutorial/a1-static-and-dynamic-libraries/

[14] (2013) Igralni pogon. Dostopno na: http://sl.wikipedia.org/wiki/Igralni_pogon

[15] (2007) C#. Dostopno na: http://searchwindevelopment.techtarget.com/definition/C

[16] (2012) An Overview of Programs and Programming Languages. Dostopno na: http://www.cplusplus.com/info/description/

[17] (2012) C++. Dostopno na: http://sl.wikipedia.org/wiki/C%2B%2B

[18] U. Mesojedec, B. Fabjan, *Java2, Temelji proramiranja*, Založba Pasadena, Slovenija, Ljubljana, 2004, 1. del, Uvod v Javo, str. 25.

[19] (2013) OpenGL: Dostopno na:
http://en.wikipedia.org/wiki/OpenGL

[20] (2012) OpenGL: Dostopno na:
http://sl.wikipedia.org/wiki/OpenGL

[21] (2013) DirectX: Dostopno na:
http://en.wikipedia.org/wiki/DirectX

[22] (2012) GLUT - The OpenGL Utility Toolkit. Dostopno na:
http://www.opengl.org/resources/libraries/glut/

[23] (2011) Picking a GUI library to use with OpenGL. Dostopno na:
http://blog.codersbase.com/2011/03/picking-gui-library-to-use-with-opengl.html

[24] (2013) GLFW. Dostopno na:
http://www.glfw.org/

[25] (2013) Simple DirectMedia Layer: Dostopno na:
http://en.wikipedia.org/wiki/Simple_DirectMedia_Layer

[26] (2012) The OpenGL Extension Wrangler Library. Dostopno na:
http://en.wikipedia.org/wiki/Simple_DirectMedia_Layer

[27] (2012) OpenGL Extensions. Dostopno na:
http://www.opengl.org/documentation/extensions/

[28] (2007) OpenGL Extensions Tutorial. Dostopno na:
http://www.opengl.org/sdk/docs/tutorials/ClockworkCoders/extensions.php

[29] (2012) OpenGL Shading Language. Dostopno na:
http://www.opengl.org/documentation/glsl/

[30] (2011) Tutorial2: VAOs, VBOs, Vertex and Fragment Shaders (C / SDL). Dostopno na:

http://www.opengl.org/wiki/Tutorial2:_VAOs,_VBOs,_Vertex_and_Fragment_Shaders_%28C_/_SDL%29

[31] (2013) GLSL. Dostopno na: http://en.wikipedia.org/wiki/GLSL

[32] (2012) Boost (C++ libraries). Dostopno na: http://en.wikipedia.org/wiki/Boost_%28C%2B%2B_libraries%29

[33] (2011) Vertex Texture Fetch. Dostopno na: http://www.opengl.org/wiki/Vertex_Texture_Fetch

[34] (2006) Vertex Displacement Mapping using GLSL. Dostopno na: http://www.ozone3d.net/tutorials/vertex_displacement_mapping_p03.php

[35] (2012) 3D survival guide for starters #4, Light/AO/HeightMaps. Dostopno na: http://tower22.blogspot.com/2012/12/3d-survival-guide-for-starters-4.html

[36] (2006) VBOs, PBOs and FBOs. Dostopno na: http://hacksoflife.blogspot.com/2006/10/vbos-pbos-and-fbos.html

[37] (2012) Android Lesson Eight: An Introduction to Index Buffer Objects (IBOs). Dostopno na: http://www.learnopengles.com/android-lesson-eight-an-introduction-to-index-buffer- objects-ibos/

[38] (2013) Heightmap: Dostopno na: http://en.wikipedia.org/wiki/Heightmap

[39] P. Rideout, *iPhone 3D Programming, 1st Edition*, O'Reilly Media, 1 edition 2010, pogl. 4.

[40] A Simple QuadTree Implementation in C#. Dostopno na: http://www.codeproject.com/Articles/30535/A-Simple-QuadTree-Implementation-in-C.