Proceduralna generacija 3D terena u OpenGL-u

Grozdanić, Marko

Sveučilište J.J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni preddiplomski studij matematike i računarstva

Marko Grozdanić

Proceduralna generacija 3D terena u OpenGL-u

Završni rad

Osijek, 2021.

Sveučilište J.J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni preddiplomski studij matematike i računarstva

Marko Grozdanić

Proceduralna generacija 3D terena u OpenGL-u

Završni rad

Mentor: doc. dr. sc. Domagoj Ševerdija

Osijek, 2021.

Sažetak

U sljedećem završnom radu istražuje se proceduralna generacija, točnije Diamond-Square Midpoint Displacement algoritam implementiran u svrhu generacije 3D modela terena. Ujedno se i opisuje izrada aplikacije koristeći moderni OpenGL 4.6 kojom ćemo vizualizirati terene, odnosno proceduralnu generaciju terena. Implementacija renderera sadrži podršku za teksture, materijale, kameru, osvjetljenje i sjenčare na osnovu Phongova modela osvjetljenja kao i vizualizaciju modela terena i njegovu generaciju.

Ključne riječi

Proceduralna generacija, OpenGL 4.6, Diamond-Square, Midpoint Displacement, 3D, teren, sjenčar, teksture, renderer, kamera, osvjetljenje, Phongov model

Procedural 3D Terrain Generation in OpenGL

Summary

Procedural generation is one of the most popular concepts in 3D computer graphics due to its endless possibilities and practical use. The following Bachelor's thesis provides an analysis of the Diamond-Square Midpoint Displacement algorithm for procedural generation. It also showcases and describes the creation of a renderer application in modern OpenGL 4.6, for the purpose of visualising the procedurally generated terrain. The renderer implementation contains features that enable textures, materials, a camera, lighting and Phong shading for enhanced visualisation of the procedurally generated terrain.

Key words

Procedural generation, OpenGL 4.6, Diamond-Square, Midpoint Displacement, 3D Terrain, sjenčar, Texture, Renderer, Camera, Lighting, Phong model

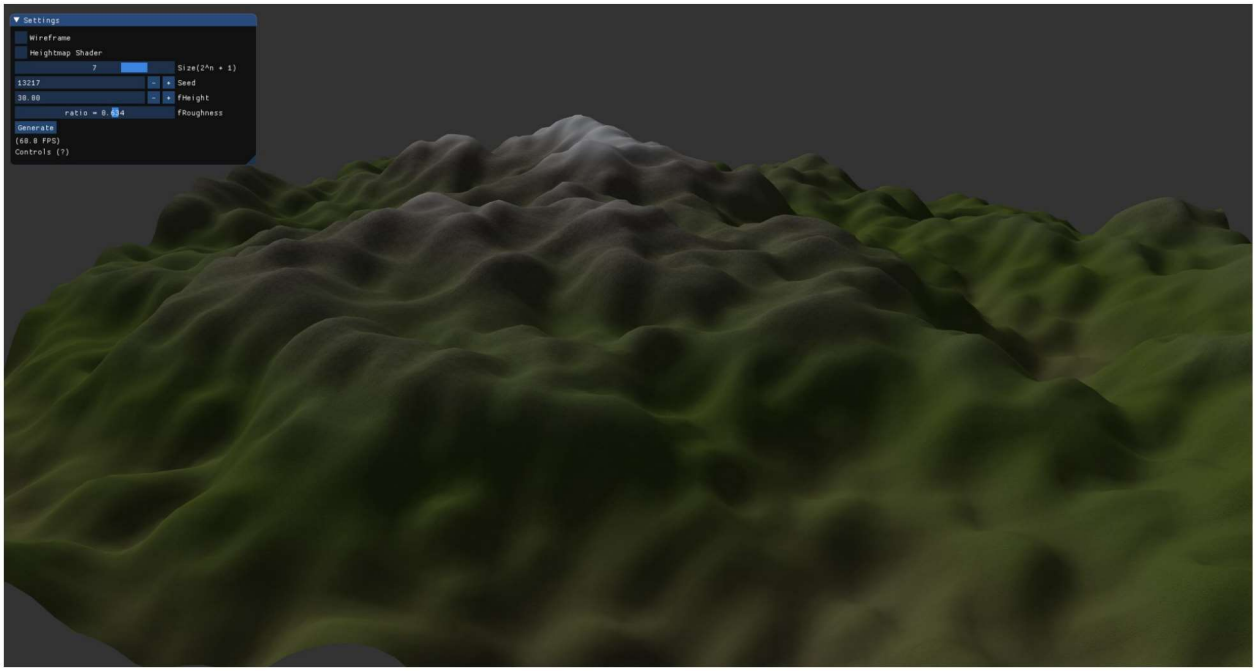
Sadržaj

Uvod

Proceduralna generacija jedan je od najvažnijih pojmova u modernoj 3D računalnoj grafici, tj. metoda koja je omogućila razvoj projekata nekoć nezamislivih razmjera. Algoritamsko generiranje podataka umjesto isključivo ručne izrade uvelike je ubrzalo kreiranje 3D modela i tekstura primijenjenih u industriji video igara i filmskoj industriji.

U sljedećem završnom radu opisuje se izgradnja OpenGL aplikacije za proceduralnu generaciju 3D modela terena koristeći *Diamond-Square* algoritam. Osim algoritamskog dijela, aplikacija sadrži sve potrebne značajke za vizualiziranje generiranog terena - osvjetljenje, nekoliko sjenčara(eng. *shader*) i tekstura. Također, aplikacija sadrži i upravljivu kameru za pregled dobivenog terena kao i korisničko sučelje s postavkama generacije i postavkama prikaza. Krajnji rezultat aplikacije je prikazan na slici 1, a izvorni kod cijele aplikacije dostupan je na [1].

Slika 1: Konačna aplikacija



1 OpenGL i struktura OpenGL renderera

Osim što želimo proceduralno generirati 3D terene, potrebno nam je i moći vizualizirati dobivene terene. Izgradit ćemo vlastiti prikazivač(eng. *renderer*) koristeći OpenGL 4.6. OpenGL(*Open Graphics Library*) je programsko sučelje koje omogućuje korištenje grafičke kartice za računanje i prikazivanje 2D i 3D vektorske grafike, a verzija 4.6 je najmodernija verzija OpenGL-a. Dakle, prikazivač ćemo graditi koristeći:

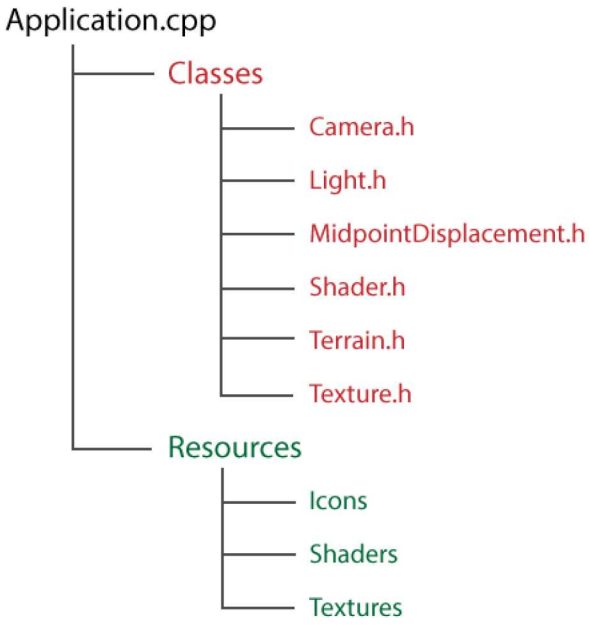
- OpenGL 4.6 programsko sučelje
- C++ programski jezik
- GLSL(*OpenGL Shading Language*) programski jezik za pisanje sjenčara
- Visual Studio Community 2019 za konfiguraciju i kompilaciju projekta

Prikazivač čine tri logičke cjeline:

- Application.cpp - ulazna točka aplikacije u kojoj se kreira prozor i vrti logika prikazivanja u svakoj sličici(eng. *frame*).
- Klase - C++ kod, klase sadrže različite logičke komponente i OpenGL implementaciju tih komponenata(npr. učitavanje teksture i njena svojstva kao objekta).
- Resursi - Sve grafičke(binarne) datoteke aplikacije kao što su teksture, ikona prozora i sjenčari .

Na slici 2 prikazana je struktura aplikacije po gore navedenim cjelinama.

Slika 2: Struktura aplikacije



1.1 Prozor, kamera i kontrole

Prije nego što počnemo algoritamski generirati teren, prvo je potrebno kreirati OpenGL prozor i kontekst. U aplikaciji za to koristimo GLFW[4], GLAD[5] - biblioteke otvorenog izvora(eng. *open source*) koje nam pružaju programsko sučelje(eng. API - *Application Programming Interface*) za naše potrebe. Dakle, u ulaznoj datoteci Application.cpp stvorimo prozor i kontekst proizvoljno definirane rezolucije. Nakon toga, velik dio logike aplikacije izvodi se *per-frame* odnosno na razini svake sličice koju prikažemo. Kod 1 prikazuje kreiranje prozora i konteksta.

Kod 1: Kreiranje prozora i konteksta

```
GLFWwindow* window = glfwCreateWindow(SCR_WIDTH, SCR_HEIGHT, "Application"
    , NULL, NULL);
if (window == NULL)
{
    std::cout << "Failed to create GLFW window" << std::endl;
    glfwTerminate();
    return -1;
}
glfwMakeContextCurrent(window);
while (!glfwWindowShouldClose(window))
    {
      //per-frame logic
    }
```

Također želimo i neke jednostavne kontrole prozora, npr. zatvaranje prozora pritiskom na tipku Escape. GLFW nam također omogućuje provjeru stisnutih tipki u prozoru čiji status provjeravamo svaku sličicu pa time i možemo implementirati ovu funkcionalnost. Kod 2 prikazuje primjer funkcije koja zatvara prozor nakon pritiska tipke Escape.

Kod 2: Zatvaranje prozora tipkom Escape

```
while (!glfwWindowShouldClose(window))
    {
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);
    }
```

Koristeći spomenutu detekciju tipki i registriranjem *callback* funkcija za druge ulaze poput korištenja kursora implementiramo i kameru koja se kreće u prostoru pritiskom tipki W, A, S, D, koja zumira prikaz korištenjem kotačića i koja se rotira držanjem desnog klika. Time imamo sve što je potrebno za kontrolu nad programom i pomicanje u prostoru. Registrirane *callback* funkcije proslijeđuju argumente prozora objektu klase Camera, koji onda računa nove matrice prikaza. Proces je ilustriran u kodu 3 na primjeru zumiranja.

Kod 3: Primjer GLFW registracije callback funkcije za zumiranje prikaza

```
glfwSetScrollCallback(window, scrollCallback);
...
void scrollCallback(GLFWwindow* window, double xoffset, double yoffset)
{
    camera.ProcessMouseScroll(yoffset);
}
```

1.2 Sjenčari

Nakon što imamo dobro definiran prozor i kameru možemo se početi baviti grafikom. Osnovne značajke koje će nam biti potrebne su **sjenčari**. Sjenčari su programi koji se izvršavaju na grafičkoj kartici, a njihova glavna zadaća je sjenčanje fragmenata(piksela) na zaslonu računanjem pozicija, boja i ostalih fizikalnih svojstava koji utječu na konačnu boju svakog piksela na zaslonu. Sjenčari u ovoj aplikaciji pisani su programskim jezikom GLSL koji je sličan C-u. Bitno svojstvo sjenčara koje se ovdje koristi je da osim hardkodiranih i očekivanih ulaza i izlaza u svaki sjenčar, kao što je npr. ulaz lokacije svakog vrha u sjenčar vrha(*vertex shader*), mogu se koristiti i **uniforme sjenčara**(eng. *shader uniform*). Pomoću uniformi svakom sjenčaru možemo dodatno proslijediti proizvoljne podatke u bilo kojem trenutku izvođenja aplikacije. U aplikaciji se koriste dva tipa sjenčara:

- Sjenčar vrha - 3D sjenčar kojemu se proslijeđuje svaki vrh(*vertex*) definiran u 3D prostoru, tj. njegovi atributi poput koordinata, normala i koordinata tekstura. U sjenčaru vrha najčešće se manipulira koordinatama svakog vrha i vrši se transformacija u 2D koordinate u kojima će se vrh, odnosno sada rasterizirani fragment pojaviti u 2D koordinatnom sustavu. Koordinate fragmenta se zatim proslijeđuju sjenčaru fragmenta gdje vršimo naknadnu obradu(eng. *postprocessing*).

- Sjenčar fragmenta(piksela) - 2D sjenčar kojemu je glavna zadaća određivanje konačne boje proslijeđenog fragmenta ili piksela. U ovoj aplikaciji jedan sjenčar fragmenta vrši fizikalne kalkulacije za vrijednosti osvjetljenja svakog fragmenta i dodaje vrijednosti tekstura za konačnu boju svakog piksela koji se prikazuje na zaslonu. Drugi sjenčar fragmenta koji se koristi konačnu boju svakog piksela određuje jednostavnim bojanjem piksela ovisno o njegovoj visini u terenu.

U aplikaciji su implementirana dva glavna sjenčara, od kojih svaki ima svoj sjenčar vrha i sjenčar fragmenta a to su *heightmapShader* i *diffuseSpecularAmbientShader*.

Više o sjenčarima možete pronaći u [3].

1.3 Sjenčar visinske mape

U aplikaciji pod nazivom heightmapShader, sjenčar visinske mape je vrlo jednostavan sjenčar koji služi za gradijentni prikaz visina u terenu. Svaka točka terena obojana je crvenom bojom intenziteta ovisnog o visini, tj. više točke imaju jači intenzitet crvene boje. Ovaj sjenčar nema nikakvih fizikalnih kalkulacija za svjetlost i sjene već mu je svrha dati jasan pregled rezultata generacije terena i njegovih visina. Ovaj sjenčar se sastoji od sjenčara vrha *heightmapShader.vert*(prikazan u kodu 4) i sjenčara fragmenta *heightmapShader.frag*(prikazan u kodu 5).

heightmapShader.vert na ulazu prima samo lokaciju vrha u 3D prostoru $aPos$, a na izlazu vraća njegovu lokaciju na 2D zaslonu $gl\_Position$ i varijablu s pomičnim zarezom $heightPos$ koja je y koordinata vrha odnosno njegova visina.

Kod 4: heightmapShader.vert

```
#version 460 core
layout (location = 0) in vec3 aPos;
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
out float heightPos;
void main()
{
  vec3 FragPos = vec3(model * vec4(aPos, 1.0));
  gl_Position = projection * view * model * vec4(aPos, 1.0f);
  heightPos = FragPos.y;
}
```

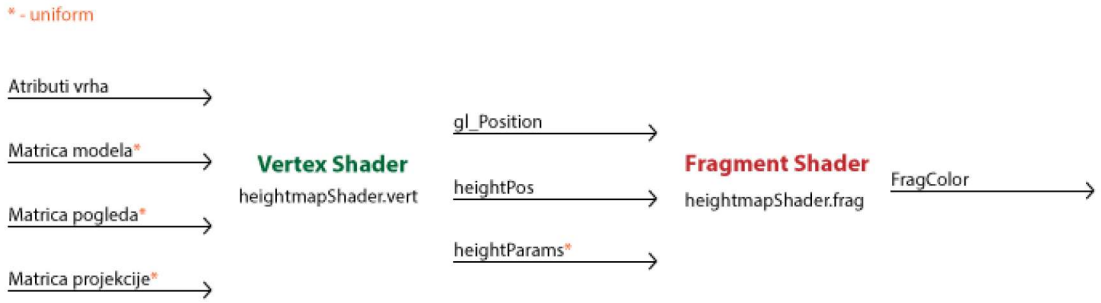
heightmapShader.frag na ulazu prima $heightPos$ iz sjenčara vrha, i uniformnu varijablu $heightParams$ koja je vektor s dva elementa. $heightParams$ se proslijeđuje kao uniformna varijabla prilikom svake generacije novog terena, a njeni elementi su minimalna i maksimalna visina u terenu. U ovom sjenčaru se onda svakom vrhu dodjeljuje crvena boja intenziteta od 0.0 do 1.0. Sve pozicije vrhova, koje mogu biti bilo koji realni broj, se preslikaju u interval $[0.0, 1.0]$ odnosno $normalisedHeight = \frac{visinaVrha - minimalnaVisina}{maksimalnaVisina - minimalnaVisina}$.

Kod 5: heightmapShader.frag

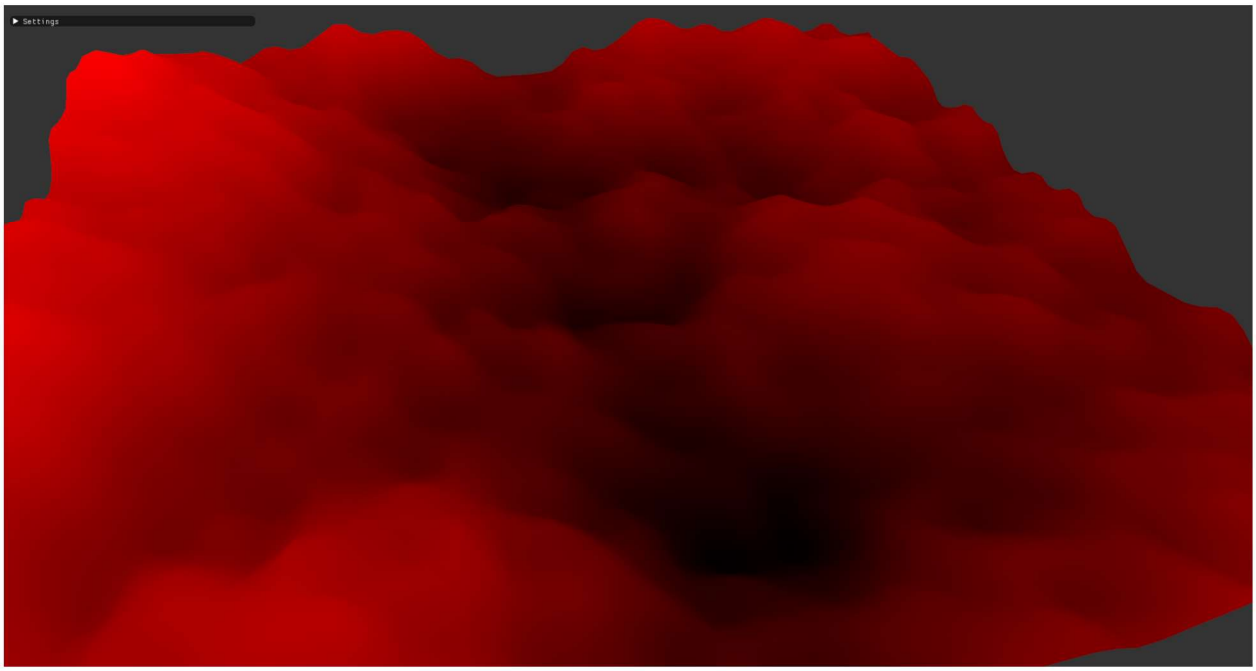
```
#version 460 core
in float heightPos;
uniform vec2 heightParams; //x - min height, y - max height
out vec4 FragColor;
void main()
{
    float normalisedHeight = ((heightPos - heightParams.x) / (heightParams
        .y - heightParams.x));
    FragColor = vec4(normalisedHeight, 0.0, 0.0, 1.0);
}
```

Cijeli tok s ulazima i izlazima iz sjenčara vrha i sjenčara fragmenta koji čine ukupni sjenčar visinske mape prikazan je u slici 3, a slika 4 prikazuje rezultat iscrtavanja tim sjenčarom.

Slika 3: Tok heightmapShader sjenčara



Slika 4: Rezultat heightmapShader sjenčara



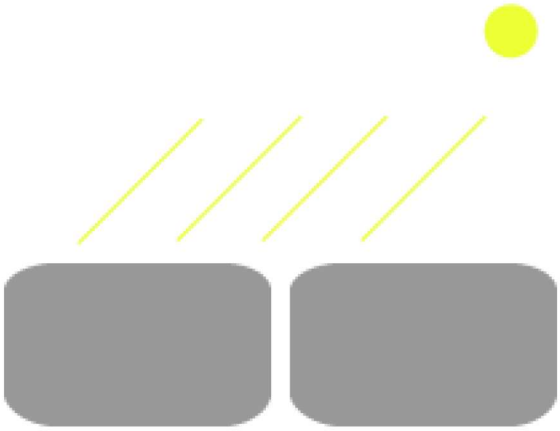
1.4 Sjenčar fizikalnog osvjetljenja

Za razliku od prethodnog, jednostavnog sjenčara ovaj sjenčar koristi Phongov model sjenčanja i teksture za realističan prikaz terena. U aplikaciji pod nazivom *diffuseSpecularAmbientShader*, naziv sjenčara je ujedno i opis svjetlosnih komponenti koje se koriste u Phongovom modelu a to su:

- Difuzna komponenta - najsnažniji doprinos, vrijednost difuzne svjetlosti je veća što je objekt bliži izvoru svjetlosti, a računa se kao skalarni produkt vektora smjera osvjetljenja i normale fragmenta.
- Zrcalna(eng. *specular*) komponenta - doprinos sjajnih površina. Slično kao i kod difuznog sjenčanja, ovaj doprinos ovisi o smjeru osvjetljenja i normale fragmenta ali i vektora smjera kamere odnosno kuta pogleda. Sjajnije površine imaju manju površinu odsjaja ali intenzivniji efekt.
- Ambijentna komponenta - aproksimacija globalnog osvjetljenja. Svakom fragmentu se zbraja mala vrijednost osvijetljenja neovisno o izvorima svjetla, tj. nijedan fragment neće biti potpuno crn iako nije direktno osvijetljen već postoji neko 'ambijentno' osvjetljenje.

Zbroj prethodna tri doprinosa čini boju svakog fragmenta. U aplikaciji se koristi samo jedan izvor svjetlosti, a to je direktno svjetlo koje baca paralelne zrake svjetlosti po cijeloj sceni kao što je prikazano u slici 5 i čija je klasa prikazana u kodu 6.

Slika 5: Direktno svjetlo



Kod 6: klasa Light.h - direktan izvor svjetlosti

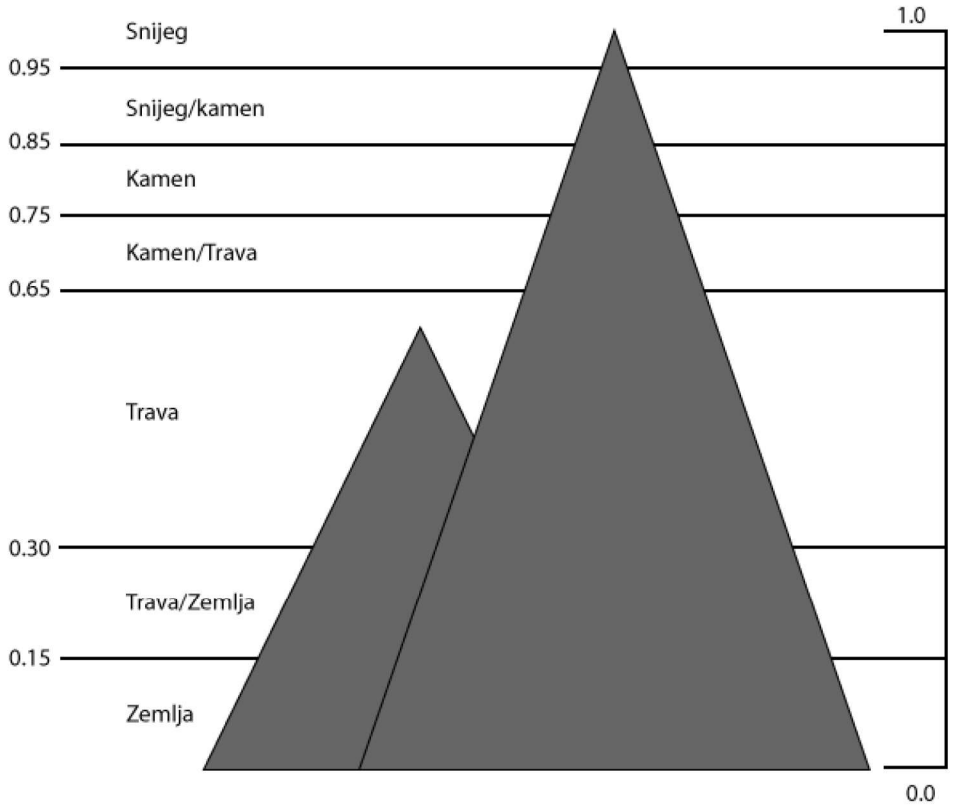
```
class Light {
public:
  LightType type;
  Shader* shader;

  Light(LightType type, Shader* shader);

  void setDirection(float PosX, float PosY, float PosZ);
  void setAmbient(float PosX, float PosY, float PosZ);
  void setDiffuse(float PosX, float PosY, float PosZ);
  void setSpecular(float PosX, float PosY, float PosZ);
};
```

Pogledajmo sada sjenčar vrha i fragmenta:

diffuseSpecularAmbientShader.vert radi slično kao i sjenčar vrha u sjenčaru visinske mape, međutim sada se na ulazu uz poziciju svakog vrha proslijeđuju i vrijednosti normale vrha, kao i koordinate tekstura vrha potrebne za teksturiranje terena. Ovaj sjenčar onda samo izvrši fazu rasterizacije 3D koordinata vrha u koordinate 2D fragmenta i zatim sjenčaru fragmenta dalje proslijedi globalnu koordinatu vrha, njegovu normalu i koordinate teksture za računanje osvjetljenja.

diffuseSpecularAmbientShader.frag je puno kompleksniji. On na ulazu prima sve što sjenčar vrha proslijeđuje i još monogo toga. Ovom sjenčaru fragmenta uniformno proslijeđujemo poziciju kamere(zbog zrcalne komponente) $viewPos$, atribute izvora svjetlosti $dirLight$, četiri materijala $grass$, $earth$, $rock$, $snow$, i $heightParams$ koji sadrži minimalnu i maksimalnu visinu u generiranom terenu.
Proces sjenčanja svakog fragmenta u ovom sjenčaru je nešto kompliciraniji. Ulogu imaju doprinos svjetlosti prema Phongovom modelu opisanome gore a konačna boja se formira iz materijala kojemu pojedini fragment pripada. Pripadnost materijalu ovisna je o visini fragmenta, tj. najviši vrhovi terena pokriveni su snijegom, zatim kamenom, zatim travom, a najniži vrhovi terena pokriveni su zemljom. Postoji i dodatan tranzicijski sloj između svaka dva sloja koji je mješavina tekstura za efekt prirodnog prijelaza. Slojevi terena prikazani su na slici 6.

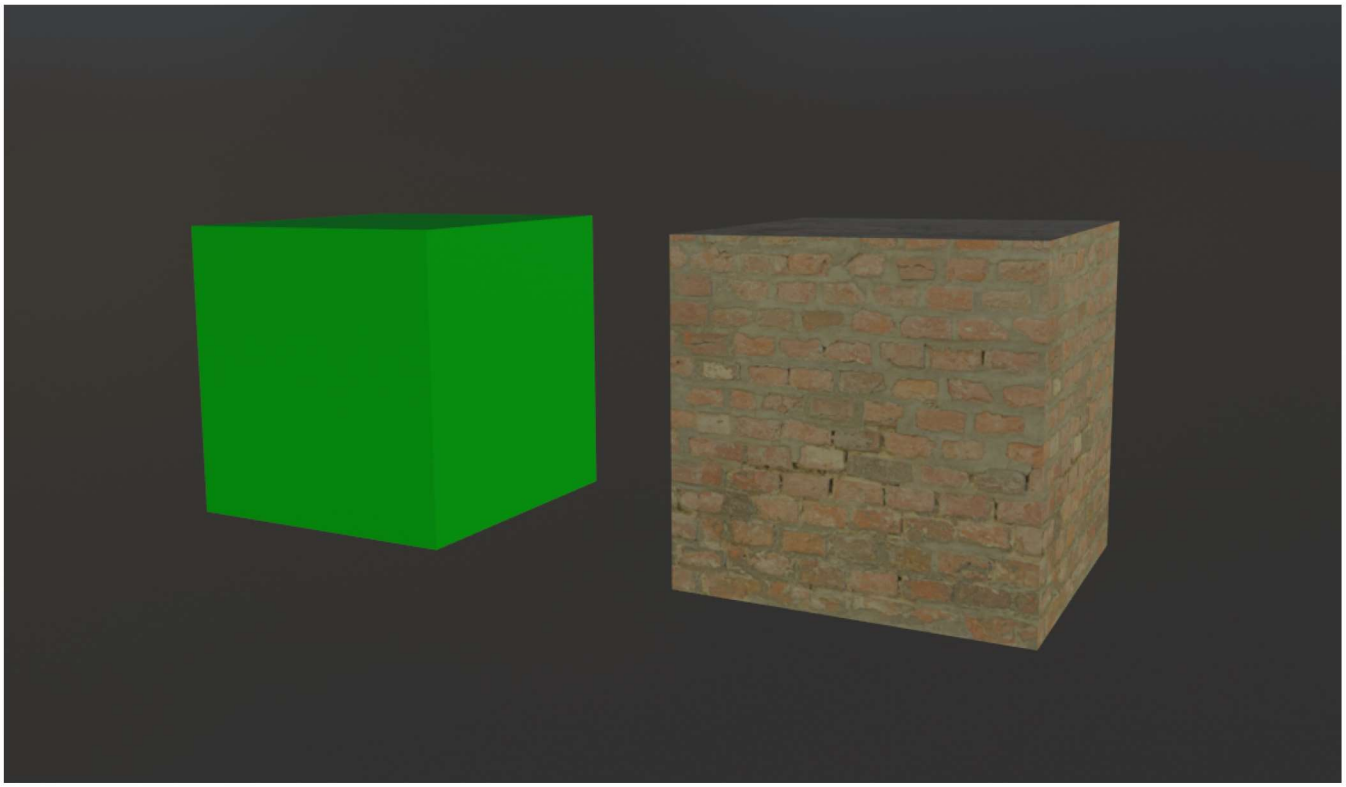
Slika 6: Slojevi terena



Teksture i materijali

Tekstura(ponekad se koristi izraz mapa) je obično 2D površina(slika), a pikseli s teksture se mapiraju na 3D površinu modela. Teksture u ovoj aplikaciji koristimo za mapiranje podataka o difuznoj i spekularnoj mapi sjenčarima. Difuzna tekstura, odnosno mapa, daje informaciju o temeljnoj boji pojedinog piksela dok spekularna mapa daje informaciju o tome koliki je odsjaj pojedinog piksela prilikom interakcije sa svjetlom. Ti podatci se zatim koriste u prethodno opisanim izračunima u sjenčaru fragmenta. Ilustrativno, slika 7 prikazuje razliku između neteksturiranog 3D modela i 3D modela teksturiranog difuznom mapom.

Slika 7: Neteksturiran i teksturiran model



Materijali su kolekcija tekstura i drugih proizvoljnih svojstava koja uz sjenčar pridružujemo modelu kako bi ga iscrtali. Svaki materijal u ovoj aplikaciji sastoji se od dvije teksture i komponente *shininess* koja regulira raspršenost svjetlosti zrcalne komponente. Općenito, materijali se mogu sastojati od proizvoljnog broja različitih mapa s različitim funkcionalnostima ovisno o načinu na koji isrctavamo.

Struktura materijala prikazana je u kodu 7, a materijali kamena i zemlje u slikama 8 i 9.

Kod 7: struct Material

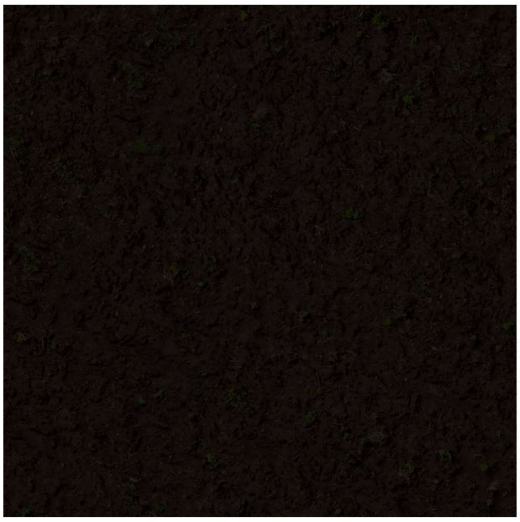
```
1 struct Material {
2     sampler2D diffuse;
3     sampler2D specular;
4     float shininess;
5 };
```

(a) Difuzna mapa

(b) Spekularna mapa

Slika 8: Materijal *rock*

(a) Difuzna mapa



(b) Spekularna mapa

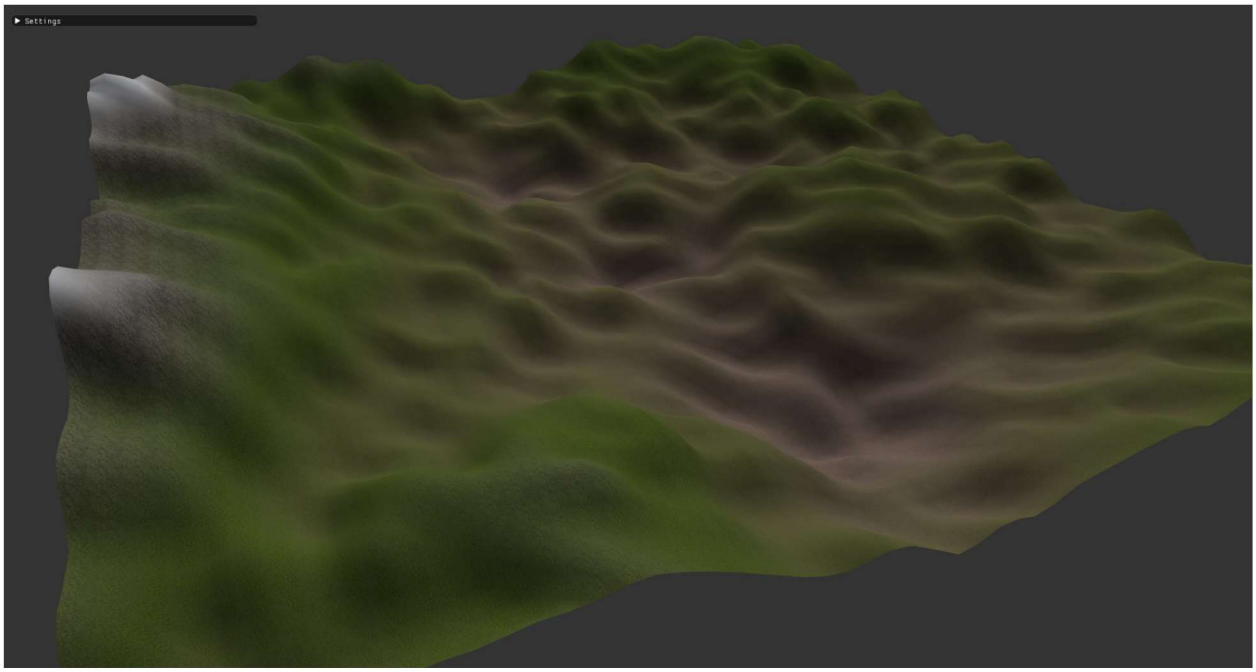
Slika 9: Materijal *earth*

Doprinos svjetlosti, materijala i visine kako je opisano se onda konačno kombinira u fragment sjenčaru opisanome u kodu 8, a koji daje rezultat iscrtavanja prikazan u slici 10:

Kod 8: diffuseSpecularAmbientShader.frag

```
void main()
{
    // properties
    vec3 norm = normalize(Normal);
    vec3 viewDir = normalize(viewPos - FragPos);

    // phase 1: directional lighting
    vec3 result = CalcDirLight(dirLight, norm, viewDir);
    result *= 1.1f; //Amplify directional light influence

    FragColor = vec4(result, 1.0);
}


vec3 CalcDirLight(DirLight light, vec3 normal, vec3 viewDir)
{
    float normalisedHeight = ((FragPos.y - heightParams.x) / (heightParams
       .y - heightParams.x));

    vec3 lightDir = normalize(-light.direction);

    float diff = max(dot(normal, lightDir), 0.0); // diffuse shading

    vec3 reflectDir = reflect(-lightDir, normal); // specular shading
    float mixRatio;
    float spec;
    vec3 ambient, diffuse, specular;
    if(normalisedHeight >= 0.95){ //SNOW
        spec = pow(max(dot(viewDir, reflectDir), 0.0), snow.shininess);
        ambient = light.ambient * vec3(texture(snow.diffuse, TexCoords));
        diffuse = light.diffuse * diff * vec3(texture(snow.diffuse,
           TexCoords));
        specular = light.specular * spec * vec3(texture(snow.specular,
           TexCoords));
    }
    .
    .
    .
    return (ambient + diffuse + specular); // combine results
}
```

Slika 10: Rezultat novog sjenčara na istom terenu iz slike 4



2 Proceduralna generacija terena

2.1 Teren

Teren je standardan 3D model koji se sastoji od vrhova, bridova i trokuta. Teren je u početku generiran kao kvadratna mreža sačinjena od $(2^n+1)^2$ vrhova, gdje je n prirodan broj, a 2^n+1 broj redaka/stupaca mreže. Svaki vrh ima početnu visinu 0.0. Proceduralno generirati teren u ovom smislu znači svakom vrhu pridružiti vrijednost visine. Klasa teren ima metode za generaciju novog(praznog) terena, pripremu terena za prikazivanje i prikazivanje terena. Za proceduralnu generaciju terenu se pridružuje objekt klase *MidpointDisplacement* koji proceduralno generira visine vrhova. Kod 9 prikazuje klasu *Terrain*.

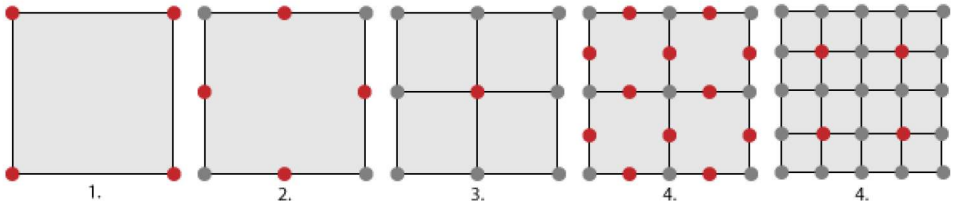
Kod 9: Terrain.h

```
1  struct Vertex {
2    glm::vec3 Position;
3    glm::vec3 Normal;
4    glm::vec2 TexCoords;
5    int adjacentFaceCount;
6  };
7
8  class Terrain {
9  public:
10   unsigned int sizeX;
11   unsigned int sizeN;
12   std::vector<Vertex> vertices;
13   std::vector<unsigned int> indices;
14   std::vector<float> terrainHeightmap;
15
16   float minHeight;
17   float maxHeight;
18
19   Terrain(unsigned int sizeX);
20   void generateTerrain();
21   void resetTerrain();
22   void setupTerrain();
23   void drawTerrain(bool wireframe);
24   void setSize(int n);
25   glm::vec3 calculateVertexNormals(Vertex vertex);
26   glm::vec3 calculateFaceNormals(glm::vec3 v0, glm::vec3 v1, glm::vec3 v2)
       ;
27
28   void genMidpointDisplacement(unsigned int seed, float spread, float
       spreadReductionRate);
29
30 private:
31   //  render data
32   unsigned int VAO;
33 };
```

2.2 Diamond-Square algoritam

U pozadini ove aplikacije proceduralna generacija terena izvodi se tzv. Diamond square algoritmom.

Slika 11: Diamond-Square algoritam



Model terena koji je kvadratna 2D mreža subdivizijom dijelimo na sve manje i manje kvadrate u kojima pojedinim točkama pridodajemo visinu. Algoritam radi s četiri glavne vrijednosti:

- size - veličina kvadratne mreže.
- seed - cijeli broj kojim inicijaliziramo generator pseudonasumičnih brojeva za računanje dodane vrijednosti visine vrha.
- fHeight - realan broj, amplituda visina koju želimo. Visina svakog vrha je u intervalu $[-\frac{fHeight}{2}, \frac{fHeight}{2}]$.
- fRoughness - realan broj koji kontrolira smanjenje nasumičnog faktora visine prilikom svake subdivizije. Veća vrijednost fRoughness varijable rezultira kaotičnim terenom.

Rad algoritma prikazanog na slici 11 možemo zatim opisati kao:

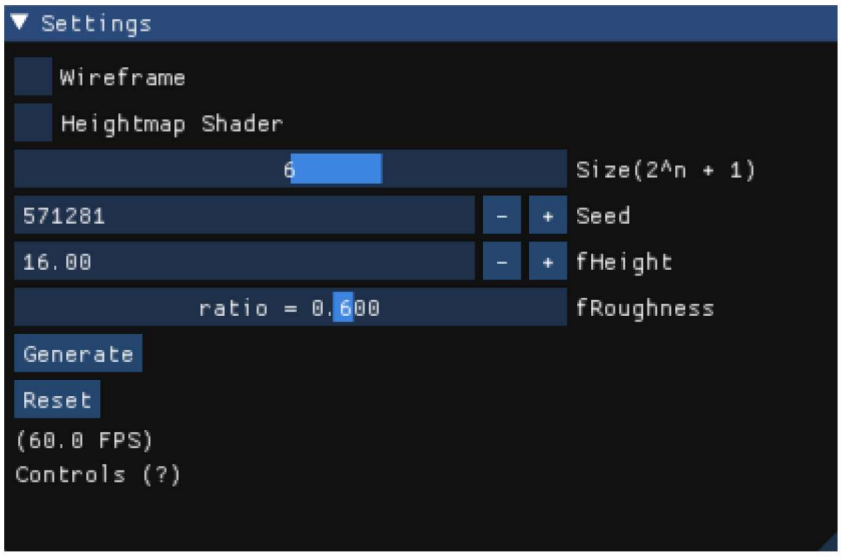
1. Postavi visinu četiri rubne točke kvadrata na nasumičnu vrijednost u intervalu fHeight.
2. Izračunaj visinu točaka na sredini bridova(midpoint) između rubnih točaka tako da je ta visina aritmetička sredina visina točaka brida plus nasumična vrijednost.
3. Izračunaj visinu točke u sredini kvadrata tako da je ta visina aritmetička sredina visina četiri točke izračunate u prošlom koraku plus nasumična vrijednost.
4. Izvrši subdiviziju trenutnog kvadrata i ponavljaj korake 2, 3, 4 sve dok ne postoje podkvadrati. Također smanji faktor fHeight tako da je novi fHeight *= fRoughness.

Više o ovome algoritmu možete pronaći u [2].

3 Korisničko sučelje

Naposlijetku, aplikacija dolazi i sa korisničkim sučeljem izrađenim pomoću biblioteke *Dear ImGUI*[6]. Korisničko sučelje ove aplikacije služi za konfiguriranje vizualnih značajki iscrtavanja terena kao i za precizno manipuliranje postavkama proceduralne generacije terena.

Slika 12: Korisničko sučelje



Korisničko sučelje aplikacije prikazano na slici 12 sadrži sljedeće elemente:

- Wireframe - uključuje wireframe način prikaza terena, odnosno mrežasti prikaz samo vrhova, bridova i trokuta bez popunjavanja trokuta.
- Heightmap Shader - uključuje *heightmapShader* sjenčar. Obično je uključen *diffuseSpecularAmbientShader* sjenčar.
- Size - Pomični element kojime korisnik odabire veličinu mreže.
- Seed - Element kojime korisnik odabire vrijednost varijable *seed*.
- fHeight - Element kojime korisnik odabire vrijednost varijable *fHeight*.
- fRoughness - Pomični element kojime korisnik odabire vrijednost varijable *fRoughness*.
- Generate - Gumb koji generira teren prema trenutnim postavkama.
- Reset - Gumb koji teren postavlja na početno 2D stanje.
- FPS counter - Broj sličica u sekundi.
- Controls(?) - Držanjem miša nad ovim tekstom korisnik dobija informacije o kontrolama programa.

Reference

[1] Grozdanić, M., 2021. GitHub - Mundus Novus - An OpenGL 3D Terrain generator. [online] GitHub. Dostupno na: https://github.com/marko-grozdanic/mundus-novus

[2] Polack, T., 2003. Focus on 3D terrain programming. Cincinnati, OH: Premier Press.

[3] de Vries, J., 2020. Learn OpenGL: Learn modern OpenGL graphics programming in a step-by-step fashion. Kendall, Welling.

[4] GLFW. 2021. GLFW - An OpenGL library. [online] Dostupno na: www.glfw.org [Pristupljeno 1.7.2021].

[5] Herberth, D., 2021. GitHub - Dav1dde/glad: Multi-Language Vulkan/GL/GLES/EGL/-GLX/WGL Loader-Generator based on the official specs.. [online] GitHub. Dostupno na: github.com/Dav1dde/glad [Pristupljeno 1.7.2021].

[6] Cornut, O., 2021. GitHub - ocornut/imgui: Dear ImGui: Bloat-free Graphical User interface for C++ with minimal dependencies. [online] GitHub. Dostupno na: github.com/ocornut/imgui [Pristupljeno 5.7.2021].