Miskolci Egyetem

Gépészmérnöki és Informatikai Kar

Automatizálási és Infokommunikációs Tanszék



Procedurális domborzatmodellezés

Szakdolgozat

Tervezésvezető:

Elek Tibor

ME. ÁIT. mérnöktanár

Általános Informatikai Tanszék

Készítette:

Czinke Zoltán

DV4BF3

Mérnök informatikus BSc

3508 Miskolc, Futó út 15.

Tartalomjegyzék

1. Bevezetés 1

2. Matematikai modell 3

2.1. Vetődés algoritmusa (3) (4) 3

2.1.1. Algoritmus menete 3

2.1.2. Implementáció 3

2.1.2.1. Egyenes egyenlete alapján 4

2.1.2.2. Vektoriális szorzat alapján 5

2.1.2.3. Az eltolás mértéke 5

2.1.3. Variációk 6

2.1.3.1. Körök algoritmusa (5) 6

2.1.3.2. Kiterjesztés gömb alakzatra (6) 7

2.2. Diamond-square algoritmus (7) (8) 8

2.2.1. Az algoritmus menete 8

2.3. Value noise algoritmus (9) 10

2.3.1. Implementáció 10

2.4. Perlin zaj (10) 12

2.4.1. Az algoritmus menete (11) 13

2.4.2. Implementáció (12) 14

2.5. Simplex zaj (13) 18

3. Felhasznált eszközök, technológiák 21

3.1. C++ és Java 21

3.1.1. Teljesítmény 21

3.1.2. OpenGL 22

3.2. OpenGL (16) (17) 23

3.2.1. Shader 23

3.2.1.1. Verzió 24
3.2.1.2. Típusok 24
3.2.1.3. Vertex shader 27
3.2.1.4. Tessellation 27
3.2.1.5. Geometry shader 31
3.2.1.6. Fragment shader 32
3.2.2. OpenGL objektumok 33
3.2.2.1. Objektum létrehozása 33
3.2.2.2. Objektum kapcsolása 33
3.2.2.3. Zérus objektum 34
3.3. GLM (18) 34
4. Implementáció 35
4.1. Inicializáló műveletek 35
4.1.1. Ablak és kontextus 35
4.1.2. OpenGL függvény pointerek beállítása 35
4.2. Magasságtérkép generálása 36
4.2.1. Heightmap osztály 36
4.2.1.1. Csúcspontok létrehozása 36
4.2.1.2. Indexek létrehozása 37
4.2.1.3. Magasság hozzáadása és normálvektorok számítása 38
4.3. Magasságtérkép kirajzolása 40
4.3.1. Dinamikus részletességszint (22) (23) 41
4.4. Felhasználói felület 44
5. Észrevételek 46
6. Fejlesztési lehetőségek 47
7. Összefoglalás 48
8. Summary 49

9. Irodalomjegyzék 50

1. Bevezetés

Szakdolgozatom célja, egy olyan program megírása, amely különböző algoritmusokkal képes domborzatot generálni, illetve a generált domborzatot megjeleníteni. Ehhez természetesen szükséges, hogy ismertessem az elterjedtebb algoritmusok matematikai hátterét is, illetve az általam választott technológiákról is szó esik a későbbiekben.

Egyetemi tanulmányaim során, egész pontosan Számítógépi grafika tárgy keretein belül ismerkedtem meg az OpenGL technológia alapjaival. Az említett tárgy egyrészt remek alapot adott ahhoz, hogy önszorgalomból tudjam folytatni a témakör tanulását, másrészt felkeltette az érdeklődésemet nem csak a számítógépi grafika, hanem a hozzá szorosan kapcsolódó matematika iránt is, mely többek között mátrix/vektor műveletekre, trigonometrikus függvényekre és egyéb grafikai algoritmusokra koncentrálódik.

A procedurális generálás megértéséhez talán a legkézenfekvőbb példa az emberi gondolkodásmód vizsgálata. Például amikor egy szorzást fejben, vagy nagyobb számok esetén írásban végzünk el, ahelyett hogy előre megjegyeztük volna az összes létező szám szorzatának kombinációit (ami egyébként is lehetetlen, mivel végtelen sok szám létezik), ismerünk egy eljárást, amellyel tetszőleges méretű számoknak a szorzatát elő tudjuk állítani. A procedurális domborzatgenerálás is hasonló elven működik, ahelyett hogy egy előre „megjegyzett” fix méretű domborzatot használunk például egy játékban a játéktérhez, helyette egy erre alkalmas algoritmussal a technikai korlátokat figyelmen kívül hagyva, végtelen nagyságú játékteret lehet létrehozni.

A korai számítógépek nagymértékben korlátozottak voltak tárhely szempontjából, ez kényszerítette a programozókat arra, hogy a gazdagép számára olyan programot írjanak, mely előre gyártott tartalom helyett, egy adott algoritmussal hozza létre a domborzatot vagy akár egyéb játékelemeket is. Ezt nevezik procedurális tartalom generálásnak. Előre meghatározott módon inicializált pszeudo véletlen szám generátorok segítségével elérhető az, hogy az algoritmus ugyanazt a tartalmat generálja le, bármelyik lefutás során. Erre egy remek példa a Pitfall nevű platformjáték, ahol is egyrészt szobákra van osztva a játéktér, másrészt procedurálisan generált akadályokkal illetve kinézettel rendelkezik, de mivel a véletlen szám generátor az adott „szobaszámmal” inicializálódott, ezért minden alkalommal, amikor egy adott szobába lépünk, pontosan ugyanazok az akadályok generálódnak le. (1) (2)

Ez a technika a játékokon kívül komolyabb programok működésének leírására is alkalmas. Ilyen például a Pandromeda cég által fejlesztett MojoWorld nevű program, mely egy fraktál alapú modellező program, amivel egy egész bolygó domborzatát legeneráltathatjuk. Sajnos a cég működése 2006-ban megszűnt, így a programot sem fejlesztik tovább.

A dokumentumban először a domborzatgeneráláshoz használt algoritmusokat részletezem, a legegyszerűbbtől a bonyolultabb algoritmusok felé haladva. Itt szeretném megjegyezni, hogy a továbbiakban, mind az algoritmusok tárgyalása, mind az implementáció részletezése során az y koordináta minden esetben a felfelé mutató irányt, tehát a magasságot jelzi. Szó esik továbbá a program megvalósításához felhasználható technológiákról, valamint a konkrét program tervezése és implementációja is részletezve lesz.

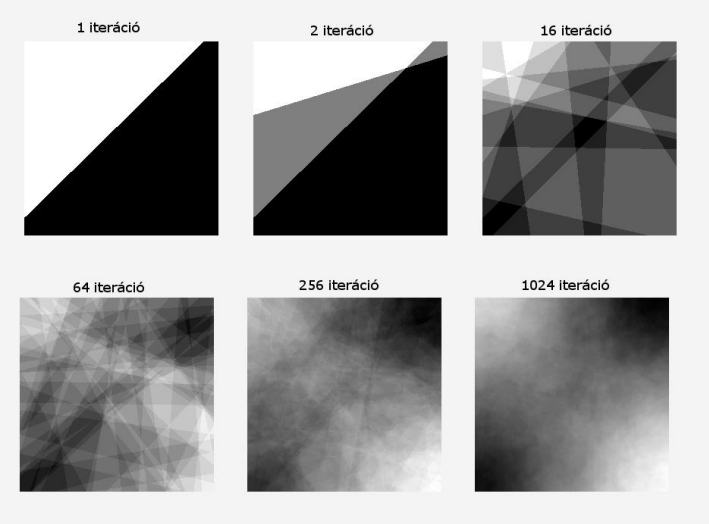
2. Matematikai modell

2.1. Vetődés algoritmusa (3) (4)

Az általam ismert algoritmusok közül ez az egyik legegyszerűbb, tulajdonképp középiskolai szintű matematikai tudás szükséges az algoritmus megértéséhez. Matematikai egyszerűsége miatt gyors a lefutási ideje. További előnye az algoritmusnak, hogy nem csak sík magasságtérképekre lehet alkalmazni, hanem gömbökre is.

2.1.1. Algoritmus menete

A magasságtérkép pontjait nullára állítjuk, majd véletlenszerűen kiválasztunk egy egyenest, amely két külön területre osztja fel azt. Az így kapott területek közül az egyiken minden pont magasságát egy előre meghatározott paraméterrel növeljük, míg a másik terület pontjait ugyanezzel a paraméterrel csökkentjük. Az így kapott domborzaton kell a felosztást ismételten elvégezni. A magasságtérkép méretétől függően 400-1000 iteráció után kapunk valóságosnak tűnő domborzatot.



1. ábra: Fault algoritmus különböző iterációkon.

2.1.2. Implementáció

Az egyikmegoldandó probléma a magasságtérképet kétfelé osztó egyenes meghatározása. Egy lehetséges egyszerű megoldás, hogy az egyenes a magasságtérkép középpontjáról indul és az emelkedése véletlenszerű. Ez azonban nem szerencsés, mivel így túlságosan aszimmetrikus lesz a domborzat, azaz az egyik oldalt egy magas hegy, míg a másik oldalt

egy mély völgy lesz. Ezek alapján olyan megoldást kell keresni, ahol általános esetben nem halad át a magasságtérkép középpontján az egyenes.

2.1.2.1. Egyenes egyenlete alapján

Az egyenes normálvektoros egyenlete:

$$x * \cos(r) + y * \sin(r) = c$$

$r - az\ egyenes\ x\ tengellyel\ bezárt\ szöge, jelen\ esetben\ egy\ véletlen\ szám$

Az egyenes a normálvektoros alakjában két változónak az egyenlete: r-nek és c-nek, tehát ezeket a paramétereket kell úgy meghatározni, hogy a kapott egyenes két részre ossza a magasságtérképet. A távolság egy ponttól $(x1, z1)$, egy egyenesig $(ax + bz - c = 0)$ a következőképp határozható meg:

$$táv = a * x1 + b * z1 - c$$

A távolság az egyenes egyik oldalán negatív lesz, az egyenes mentén nulla, az egyenes másik oldalán pedig pozitív. A középpontot $(0,0)$ kizárólag a c változó határozza meg:

$$táv = a * 0 + b * 0 - c = -c$$

Tehát a c változónak az értéke kisebb kell, hogy legyen, mint a középponttól lévő legtávolabbi pont távolsága. A magasságtérkép hossza (l), és szélessége (w)alapján a c változót a $[-\frac{d}{2}, \frac{d}{2}]$ intervallum határozza meg, ahol d

$$d = (w * w + l * l)^2$$

Amennyiben $w = l$

$$d = w * 2^2$$

Ez után a magasságtérkép összes pontjára el kell végezni az ellenőrzést, hogy az egyenes melyik oldalára esik, és ettől függően növelni vagy csökkenteni a magasságát. A továbbiakban feltételezzük, hogy a magasságtérkép négyzet alakú.

Az algoritmus pszeudo-kódja:

```
for iterációk száma do{
      v=rand();
      a=sin(v);
      b=cos(v);
      d=sqrt(2)*magasságtérkép.size();
      c=(rand() / RAND_MAX) * d - d/2;
            for each pont(tx,tz) in magasságtérkép do{
                  if(a * tx + b * tz - c > 0)
                        magasságtérkép(tx,tz) += eltolás;
                  else
                        magasságtérkép(tx,tz) -= eltolás;
      }
}
```

2.1.2.2. Vektoriális szorzat alapján

A magasságtérképen véletlenszerűen választott két pontból vektort képzünk:

$$v0 = (x2 - x1,\ z2 - z1)$$

Majd a magasságtérkép minden pontjára képezzük a vt vektort. A két vektor keresztszorzatának y irányba mutató komponensének előjele adja meg, hogy az aktuális pontot melyik irányba mozdítjuk el.

$$vt = (t_x - v0_x,\ t_z - v0_z)$$

```
méret=magasságtérkép.size();
for iterációk száma do{
vec3 p1(rand() mod méret,0,rand() mod méret);
vec3 p2(rand() mod méret,0,rand() mod méret);
vec3 v0(p2-p1);

for each pont(tx,tz) in magasságtérkép do{

vec3 vt(tx-v0.x,0,tz-v0.z);

      if (v0.z * vt.x - vt.z * v0.x) > 0
                  magasságtérkép(tx,tz) += eltolás;
            else
                  magasságtérkép(tx,tz) -= eltolás;
      }
}
```

2.1.2.3. Az eltolás mértéke

A következő megoldandó probléma, az iterációnkénti eltolás mértékének a meghatározása. A legegyértelműbb megoldás, hogy az elmozdítás minden iterációnál konstans értékkel rendelkezik. Egy másik lehetséges megközelítés, hogy a változás mértéke lineáris mértékben csökken iterációnként, ügyelve arra, hogy egy nullától nagyobb értéket elérve ne csökkenjen tovább. A lineáris csökkentéshez fel lehet használni az iterációk számát a következőképpen, feltételezve, hogy `i` a pillanatnyi iteráció értéke, `d0`, `di`, `dn` a kezdeti

eltolás mértéke, az adott iteráción kiszámolt eltolás, illetve a végleges eltolás, `n` pedig az utolsó iteráció, ahol csökkentjük az eltolás mértékét.

```
if(i < n)
      di = d0 + (i / n) * (dn – d0)
else
      di=dn;
```

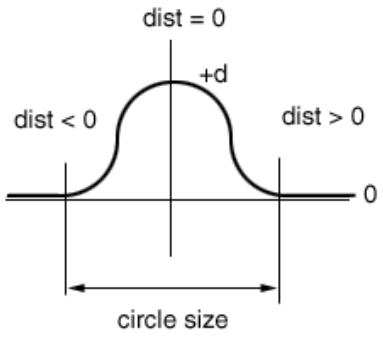
Külön figyelmet kell fordítani az `n`, illetve a `dn` értékének megválasztására. Ha a `dn` értéke sokkal kisebb, mint `d0` értéke, akkor `n` –t megfelelően nagynak kell választani, egyébként az eltolás mértéke túlságosan gyorsan lecsökken, és a kezdeti iterációk túl erős nyomot hagynak a domborzaton. Ebben az esetben, amennyiben `n` nem elég nagy az első iteráció nyomai mindenképp látszódni fognak, függetlenül attól, hogy mennyi iteráció fut le.

Továbbá az eltolás mértékét ki is lehet terjeszteni nagyobb területre; ahelyett hogy az egyenes mentén hirtelen változtatjuk a magasságot, az egyenes távolságának függvényében több pont magasságán is változtathatunk.

2.1.3. Variációk

2.1.3.1. Körök algoritmusa (5)

Hasonlóan az eredeti algoritmushoz, a magasságtérképen valamilyen logika szerint kiválasztott pontokat felfelé tol el. Amiben különbözik, az a kiválasztás módja. A magasságtérképet kettészelő egyenes helyett, egy véletlenszerűen kiválasztott pont környezetét módosítja. Ebbe a körbe eső pontok felfelé tolódnak, míg a körön kívül lévő pontok magassága nem változik.



2. ábra: A kör sugarán belül elhelyezkedő pontok a koszinusz függvény szerint tolódnak el.

Amint látható, ez az algoritmus az eredetitől is egyszerűbb, mivel nem kell sem egyenes egyenletével számolni, sem vektorokat képezni. Első lépésként kiválasztunk véletlenszerűen egy pontot, amely a körnek a közepe lesz, majd a megadott sugár alapján eltoljuk a pontokat.

Az algoritmus pszeudo-kódja:

```
for each point(tx,tz) in magasságtérkép do{
pontTáv  = középpontTáv*2 / átmérő;
      if abs(pontTáv) <= 1.0
      magasságtérkép(tx,tz)+= eltolás/2+cos(pontTáv*PI)*eltolás/2;
}
```

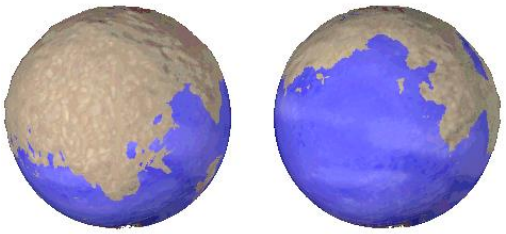
2.1.3.2. Kiterjesztés gömb alakzatra (6)

Mint már említésre került, az algoritmus logikáját kis módosítással, alkalmazni lehet gömbökre is. A megoldandó probléma szinte ugyanaz, mint az eredeti algoritmusnál, annyi különbséggel, hogy egy véletlenszerűen kiválasztott egyenes meghatározása helyett egy sík magasságtérképen, itt egy síkot kell meghatározni, amely kettészel egy háromdimenziós gömböt.

Először is természetesen szükség van egy gömbre, amelyen az algoritmust el tudjuk végezni. Célszerű olyan gömböt választani, amely minél több pontból áll, illetve sokat javít a végeredményen, ha a gömb pontjai egyenletesen oszlanak el a felületén.

Első lépésben meg kell határozni a síkot, amely két részre osztja a gömböt. A sík ebben az esetben definiálható egy, a síkon elhelyezkedő pont, illetve a sík normálvektor segítségével. Az egyszerűség kedvéért a normálvektorok a gömb közepéből indulnak ki. Ez garantálja azt, hogy a sík metszi a gömböt. Mivel az algoritmus működése során a konkrét sík egyenletre nincs szükség, csupán azt kell ellenőrizni, hogy a gömb aktuálisan vizsgált pontja a sík mögött van-e, ezért elegendő csak a sík normálvektorát előállítani, majd skaláris szorzat segítségével ellenőrizni, hogy a vizsgált pont a sík melyik oldalán van.

A *3. ábrán* észrevehető az algoritmus egyetlen hibája, a szemközti oldal és a hátoldal fordított tükörképe egymásnak, illetve ami az egyik oldalon szárazföld, az a másik oldalt tenger. Ennek a szimmetriának az oka, hogy a generált síkok a gömb középpontján áthaladnak. Ezt nagyon könnyen ki lehet javítani, csupán annyit kell tenni, hogy a generált normálvektor nem csak a sík irányvektorát adja meg, hanem a sík távolságát a gömb középpontjától.



3. ábra: 1000 iteráció eredménye, szemből illetve hátulról

```
loop{
n = vec3( rand() );
m = véletlenszerűen -1 vagy 1;
      for each pont {
            p = aktuális pont koordinátája;
            v = p – n;
            d = dot(n, v);
            if ( d > 0 ) then
                  aktuális pont += m;
            else
                  aktuális pont -= m;

      }
}
```

Mivel az általános esetben a sík nem halad át a gömb középpontján, így a gömb középpontja mindenképpen a sík mögött lesz, emiatt a pontok nagy része is minden iteráción a középponttól kifelé fog tolódni, így a végeredmény minden esetben egy csak szárazföldből álló gömb lesz. Ezt ellensúlyozza az `m` változó, amely iterációnként véletlenszerűen veszi fel az eltolás pozitív, vagy negatív értékét, így az iterálás során véletlenszerűen dől el, hogy az adott sík mögött lévő pontok melyik irányba tolódnak el.

2.2. Diamond-square algoritmus (7) (8)

Szintén az egyszerűbb algoritmusok közé tartozik - sebességét tekintve talán az egyik leggyorsabb lefutás idővel rendelkezik, amennyiben a magasságtérkép mérete nem túlságosan nagy -, ennek ellenére viszonylag jó eredményt produkál, ugyanakkor rengeteg hátránya is van, elsősorban mivel az algoritmus memóriaigényes, a lefutás ideje nagymértékben megnövekedhet, amennyiben túlságosan nagyméretű magasságtérképet generálunk.

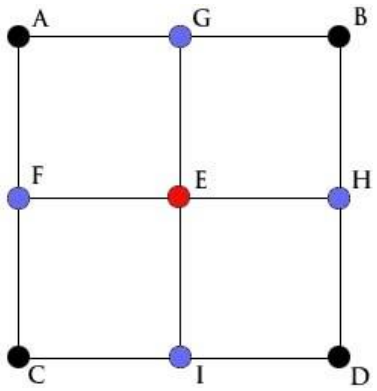
A domborzat iterálások sorozataként jön létre, minden iterálás növeli a végeredmény részletességét. Az algoritmus arra lett tervezve, hogy négyzet alakú, $2^n + 1$ méretű magasságtérképeket generáljon, ahol n az iterációk száma. Ez azt jelenti, hogy 8 iterációval, 257x257 méterű, jól kinéző domborzatot lehet létrehozni. Az algoritmus legfontosabb paramétere az érdesség konstans. Ez az érték határozza meg a végleges domborzat kinézetét, egészen a sima domborzattól, az érdesig.

2.2.1. Az algoritmus menete

Az algoritmus 2x2-es méretű magasságtérképpel indul, a négy pont kezdeti magassága lényegtelen. Az első lépésben ki kell számolni, a magasságtérkép közepén lévő magasságot, amely a 4 sarok magasságának az átlag értéke, plusz egy véletlen érték.

$$E = (A + B + C + D)/\,4 + RAND(d)$$

Ahol $RAND(d)$ egy véletlen érték $[-d, d]$ intervallumon. A d értéke, a maximális eltolás mértéke, az adott iteráción. Ha ezután a lépés után rajzolnánk ki a magasságtérképet, akkor négy darab gyémánt alakzatot látnánk (innen ered az algoritmus neve).



4. ábra Középpontok meghatározása, az első iterálás során.

Következő lépésben az élek középpontjait kell kiszámolni:

$$F = (A + C + E + E)/\,4 + RAND(d)$$
$$G = (A + B + E + E)/\,4 + RAND(d)$$
$$H = (B + D + E + E)/\,4 + RAND(d)$$
$$I = (C + D + E + E)/\,4 + RAND(d)$$

A felsorolt képletek utalnak arra, hogy a magasságtérkép mintázata önismétlődő, tehát több generált magasságtérképet egymásmellé lehet rakni úgy, hogy az illesztések mentén nem lesz semmiféle hézag. Gyakorlatban ez a tulajdonság nem használatos, mivel észrevehető ismétlődést okoz a domborzatban, és ez egy nem kívánt tulajdonság. Így a fenti képlet a következőképp egyszerűsödik:

$$F = (A + C + E)/\,3 + RAND(d)$$

Tehát bármely él középpontját annak 3 szomszédos pontjának átlagából számoljuk ki. Ezzel az első iteráció végéhez ért az algoritmus, a második iteráció előtt kell a d értékét megváltoztatni, következő módon:

$$d = d * 2^{-r}$$

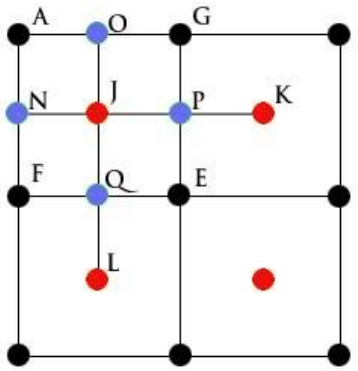
Ahol az r változó az érdesség konstans.

A következő iterációk során szinte ugyanaz az eljárás, csak a kiinduló magasságtérkép egy négyzet helyett, már 4 kisebb négyzetből áll, melyeknek a középpontjait hasonló módon számoljuk, mint az első iterációnál.

$$J = (A + G + F + E)/\,4 + RAND(d)$$
$$K = (G + B + E + H)/\,4 + RAND(d)$$
$$L = (F + E + C + I)/\,4 + RAND(d)$$
$$M = (E + H + I + D)/\,4 + RAND(d)$$

Majd ezek után hasonlóan a kis négyzetek éleinek a középpontjainak meghatározása következik:

$$N = (K + A + J + F)/\,4 + RAND(d)$$
$$O = (L + A + G + J)/\,4 + RAND(d)$$
$$P = (J + G + K + E)/\,4 + RAND(d)$$
$$Q = (F + J + E + L)/\,4 + RAND(d)$$



5. ábra Középpontok meghatározása, a második iterálás során.

Illetve amennyiben nem önismétlődő magasságtérképet szeretnénk generálni, a széleken lévő pontok képlete a következőképp változik:

$$N = (A + J + F)/\,4 + RAND(d)$$
$$O = (A + G + J)/\,4 + RAND(d)$$

Az iterációkat addig ismételjük, amíg a kívánt méretet el nem érjük.

2.3. Value noise algoritmus (9)

Különböző frekvenciájú és amplitúdójú zaj értékek interpolálása, és összegzése alapján jön létre a magasságtérkép. Ha fogunk néhány véletlen magassággal rendelkező pontot, és interpolált vonalakkal összekötjük őket, végeredményként kapunk egy kevésbé részletes domborzatot, amelyre ha ismételten elvégezzük ugyanezt a folyamatot, csak más értékekkel, akkor ez egyre részletesebb lesz.

Az eddig tárgyalt algoritmusokhoz képest az előnye, hogy bármelyik pont kiszámítása független a többi ponttól, így nem kötelező memóriában tárolni az egész magasságtérképet. Ezen kívül, ennél az algoritmusnál a részletességet is jól lehet szabályozni. Ezeknek a jó tulajdonságoknak viszont ára van, hiszen egyrészt sokkal lassabb a lefutási ideje az algoritmusnak, mint az eddigieknek, illetve valamilyen szinten nehezebb az implementálása is.

2.3.1. Implementáció

A konkrét kód magyarázata előtt célszerű tisztában lenni a következő kifejezésekkel: oktáv, frekvencia, amplitúdó. Az oktáv értéke határozza meg, hogy hány réteget képzünk egymásra, illetve hogy a végleges domborzat mennyire lesz részletes. A frekvencia értéke határozza meg, hogy az adott rétegen mennyi pontot fog tartalmazni az adott terület, illetve

a tereptárgyak szélességét. Például magas hegyekhez elegendő néhány pont, amelyet a kis frekvencia reprezentál, az apró részletekhez pedig sok száz pont is szükséges lehet, melyet nagyobb frekvencián érhetünk el. Az amplitúdó határozza meg, hogy a domborzat általánosságban mennyire legyen magas.

A legjobb eredményt úgy érhetjük el, ha minden oktáv (iterálás) után csökkentjük az amplitúdót, és növeljük a frekvenciát. A tartósság (persistence) értéke határozza meg, hogy az amplitúdó milyen mértékben csökken az iterálások során. Fontos, hogy jól válasszuk meg ezt az értéket, ugyanis nagy érték esetén az amplitúdó nem csökken elég gyorsan, így az apró részletek is magasak lesznek, túl kicsi értéknél pedig a részletek nem fognak látszódni.

```
for each point(x,z) in magasságtérkép do{
összeg = 0.0f;
frekvencia = 1.0f/(méret);
amplitúdó = tartosság;
for(int i=0; i < oktávok ; ++i){
      összeg+=interpoláltZaj(x*frekvencia,z*frekvencia)*amplitúdó;
      frekvencia *= 2;
      ampltitúdó *= tartosság;
      }
magasságtérkép[x][z] = összeg
}
```

Az interpoláció különösen fontos kisebb frekvenciákon, mivel csak néhány pont áll rendelkezésre. Lineáris interpoláció esetén két pont között egy egyenes vonal jön létre, amely legfeljebb a magasabb hegyek létrehozásához jó. Általánosságban célszerű koszinuszos interpolációt használni, amely csak egy kicsit lassabb a lineárisnál, viszont sokkal szebb eredményeket produkál. Amennyiben az algoritmus gyors lefutási ideje nem szempont, használható például harmadfokú interpoláció is, amely az éleket szépen lesimítja, viszont cserébe sokkal időigényesebb, mint a koszinuszos interpoláció.

Az algoritmus legfontosabb része a véletlen szám generátor. Egy iteráláson belül többször is meg kell hívni a függvényt, azonban minden hívásnál ugyanazon pontokra ugyanazt a véletlen számot kell visszaadnia.

```
float zaj(int x, int z){
      int r;
      float s;
      srand(z*méret+x+seed);
      r=rand() / RAND_MAX;
      return r;

}
```

Lényegében a véletlen szám generátort minden függvényhívásnál újrainicializáljuk. A `seed` értéke a program futásának elején állítódik be, így minden magasságtérkép egyedi lesz. Továbbá a véletlen szám generátor inicializálásában szerepet játszanak az `x` és `z` koordináták is, így lehet elérni azt, hogy egy iteráláson belül, ugyanazokra a koordinátákra ugyanazokat a véletlen számokat adja vissza a függvény. Az utolsó megoldandó probléma az, hogy a véletlen szám generátor akkor is ugyanazt a számot adja vissza, ha az adott pont koordinátáit felcseréljük. Ez a probléma könnyen megoldható. Annyit kell tenni, hogy az egyik koordinátának az értékét megszorozzuk egy, a magasságtérkép méretével megegyező, vagy attól nagyobb számmal.

2.4. Perlin zaj (10)

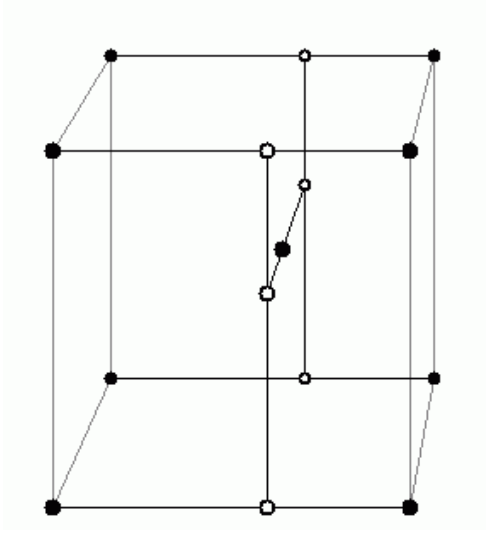
Egy jó véletlen szám generátor olyan számokat ad, melyek között nincs megfigyelhető ismétlődés. Valamilyen szintű véletlenszerűség szükséges, ha természetes viselkedést szeretnénk egy algoritmusban leírni, viszont csak és kizárólag a véletlenszerűségre támaszkodni nem megfelelő, mivel az esetek többségében a pszeudo-véletlen számok nem keltenek természetes hatást. A Perlin zaj algoritmus, mely a nevét a megalkotója, Ken Perlin után kapta, amely ezt az elgondolást veszi figyelembe. Az algoritmust az eredeti *Tron* filmhez fejlesztette ki, az 1980-as években; arra lett tervezve, hogy procedurális textúrákat hozzon létre, számítógép által generált effektekhez. A Perlin zaj használható különféle effektek generálásához, amik természetes tulajdonságokkal rendelkeznek: többek között felhők, domborzatok és mintás textúrákhoz mint például a márvány.

A zaj véletlenszerűnek tűnik, de valójában nem az. Ha így volna, akkor a zajfüggvény bármely meghívására más és más eredményekkel térne vissza. Ehelyett a függvény pszeudo-véletlen, tehát csak a véletlenszerűség látszatát kelti. A zaj tulajdonképp egy $R^n \rightarrow R$ leképezés, ahol n a dimenziók száma, és a bemenet egy n dimenziós, valós vektor. A leggyakrabban használt az $n = 1,\ n = 2,\ n = 3$ dimenziók, ahol az egy dimenziós zajt például animációk leírásához lehet használni, a kétdimenziós zajtextúrák létrehozásához, a háromdimenziós zajt pedig komplexebb textúrák létrehozásához lehet használni. Harmadik

dimenzió fölötti zajokat idő függvényében változó, tömör textúrák generálásához is lehet használni. A zaj sávkorlátozott, tehát az energiájának a nagy része frekvencia spektrum kis tartományára esik.

2.4.1. Az algoritmus menete (11)

Adott egy bemeneti P pont, amely egy rácsban helyezkedik el. Ennek a pontnak a szomszédjait kell megkeresni. Két dimenzió esetén négy szomszédos pontot, három dimenzió esetén 8 szomszédos pontot találunk. N dimenzióban 2^n számú szomszédja lesz a pontnak. Minden szomszédos Q rács ponthoz választunk egy pszeudo-véletlen G gradiens vektort. Fontos, hogy a rácsban minden adott ponthoz ugyanazt a gradiens vektort válasszuk. Következő lépésben a $G * (P - Q)$ skaláris szorzatot kell elvégezni. Ez adja meg a függvény értékét P pontnál, G gradienssel, amely zérus a Q rácspontnál. Ezt minden pontra elvégezve 2^nértéket kapunk, ezeket kell interpolálni a pontra, egy S-alakú görbét (pl.: $3t^2 - 2t^3$) használva. Ehhez a lépéshez n darab, S-alakú görbét kell kiszámolni, illetve $2^n - 1$ lineáris interpoláció elvégzése szükséges. Három dimenzió esetén, mint már említve volt, nyolc szomszédja van egy adott pontnak. Ahhoz hogy a szomszédos pontok hatásait összegezzük, trilineáris interpolációt kell használni, amely nem más, mint a tér három irányába történő lineáris interpoláció. Gyakorlatban ez azt jelenti, hogy miután kiszámoltuk az x, y, z koordinátákhoz tartozó interpolációhoz használt görbéket, hét darab lineáris interpolációt kell elvégezni, ahhoz hogy a végeredményt megkapjuk. A 6. *ábrán* az x, y, z koordináták a *jobb, fel, előre* irányokat jelentik, ebben a sorrendben. A hat darab fehér pont az első hat interpoláció eredménye: négy interpoláció *x* irányban, ezt követve két interpoláció *y* irányban. Végezetül a hetedik interpoláció a *z* irányba adja a végeredményt.



6. ábra

A gradiens kiszámolásánál fontos a sebesség, mivel az algoritmus futása során számos alkalommal kell elvégezni azt. Továbbá az is nagyon fontos, hogy ne legyen látható korreláció a szomszédos gradiens értékek között. Szerencsére a zaj függvény ismétlődése nem gond, de csak amennyiben hosszú táv megtétele után ismétlődik. Néhány száz egység megtétele után, nem számít, ha ugyanazok a mintázatok jelennek meg. A zajnak nincsenek

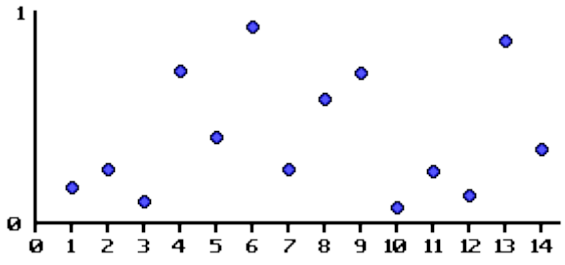
nagyméretű mintázatai, így ha a megfigyelő távolról is nézi a mintát, nem fog látni ismétlődést, mivel a kisebb méretű minták távolról nézve összemosódnak. Ezek alapján a véletlen gradiensek számolása a következőképp gyorsítható: szükséges egy P permutációs tábla, 256 bejegyzéssel, amely összekeveri a számok sorrendjét 0 és 255 között, továbbá szükséges egy G tábla, amely az előre kiszámolt gradienseket tartalmazza. Az adott bementekre (i,j,k) a következő kódsor adja meg az eredményt:

```
P[n] = permutációk
G[n] = gradiensek
g = G[(i+P[(j+P[k])mod n])mod n]
```

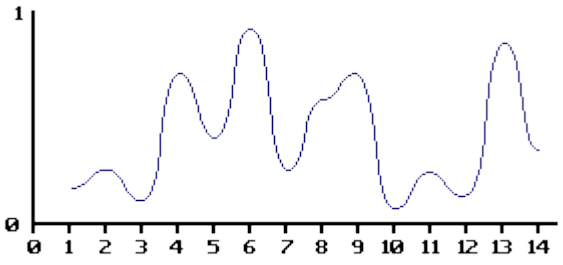
Mivel 256 kettő hatványa, így az egész osztás helyett maszkolást is lehet alkalmazni, amely még tovább gyorsítja az algoritmust. A gradienseket úgy kell meghatározni, hogy egyenletesen legyenek elosztva az egységsugarú gömb felszínén. Ehhez nem kell mást tenni, mint egyenletesen eloszló vektorokat létrehozni egy gömb alakú térfogatban, úgy, hogy a vektorok kiindulópontja a gömb közepe, majd normalizálni a vektorokat.

2.4.2. Implementáció (12)

A zaj függvény tulajdonképp egy fix értékkel inicializált véletlen szám generátor. Bemenetként egy (pontosabban a dimenziószámmal megegyező) egész számot vár, és e paraméterek alapján ad vissza egy véletlen számot. Ha ugyanazokkal a paraméterekkel hívjuk meg a függvényt, ugyanazt a véletlen számot adja vissza. Ez a legfontosabb tulajdonsága a Perlin zajnak.



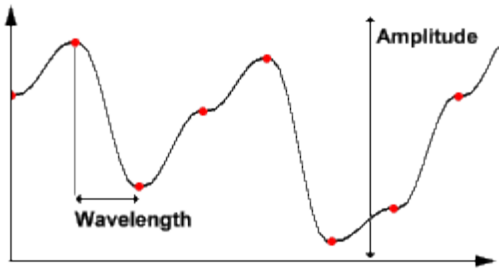
7. ábra



8. ábra

A *7. ábrán* látható egy példa a zajfüggvényre, mely egy véletlen értéket rendel nulla és egy között minden X tengelyen lévő számhoz. A *8. ábrán* látható, hogy amennyiben a véletlen értékek között interpolációt végzünk, kapunk egy folytonos függvényt, amely egy valós számot vár paraméterként. Ha a függvényt hullámként fogjuk fel, akkor a hullámhossz a *9. ábrán* látható piros pontok közötti távolág, az amplitúdó pedig a függvény minimum és

maximum értékeinek a különbsége, a frekvencia pedig a hullámhossz reciproka. Végeredményben a Perlin zaj nem más, mint ilyen interpolált függvények összege.



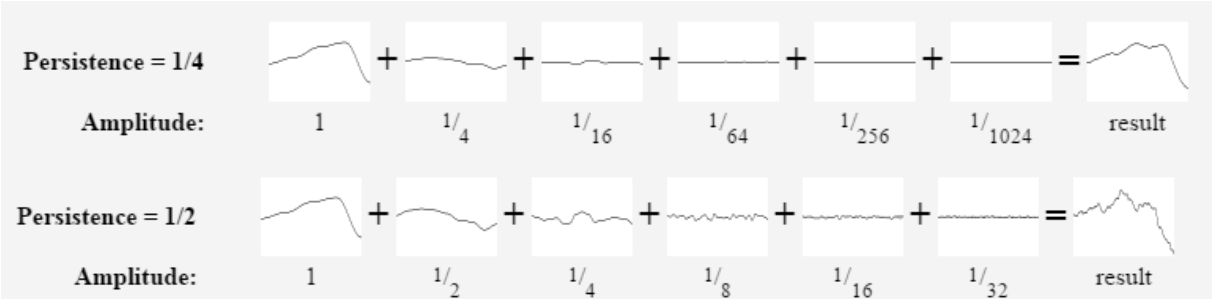
9. ábra

Az összeadás során a zajfüggvények amplitúdóját és frekvenciáját célszerű változtatni, a leggyakoribb megoldás az, hogy minden összeadás után a frekvenciát duplázzuk, az amplitúdót megfelezzük. Tulajdonképp ez a megoldás annyira elterjedt hogy sokan nem is használnak más megoldást. Ahhoz hogy különböző karakterisztikájú perlin zajt hozzunk létre minden lépésnél más és más frekvenciát és amplitúdót kell használni. Például dombok létrehozásához lehet egy olyan Perlin zajt használni, amely alacsony frekvenciákon nagy amplitúdókat használ, magas frekvenciákon pedig nagyon kis amplitúdókat.

Hogy egyszerűbbé tegyük a dolgot, valamint azért is, hogy elkerüljük azt, hogy minden iterációnál definiálni kelljen az amplitúdót és a frekvenciát, egyetlen számmal határozzuk meg azokat. Ez az érték a perzisztencia. A perzisztencia definíciója a következő:

```
frekvencia = 2^i
amplitúdó = perzisztencia^i
```

Ahol `i` az adott soron következő, a végeredményhez hozzáadandó zaj függvény. A 10. *ábrán* különböző perzisztenciákkal generált Perlin zajokat láthatunk.



10. ábra 1 dimenziós Perlin zajok 0.25 illetve 0.5 perzisztenciákkal

Minden egyes zaj függvény hozzáadását egy oktávnak nevezzük. Ennek oka, hogy minden zaj függvény frekvenciája kétszerese az előzőnek. Zenei értelemben vett oktávoknak ugyanez a tulajdonságuk. Hogy mennyi oktávot használunk, az teljes mértékben ránk van bízva, de amennyiben a Perlin zajt textúrák, vagy egyéb vizuális elemek létrehozásához használjuk, egy bizonyos frekvencia fölött nem lesz már látható különbség az újabb

oktávok hozzáadásával, mivel az adott megjelenítő eszköz nem biztos, hogy rendelkezik annyi pixellel, hogy az apró részleteket is megjelenítse. Egy bizonyos amplitúdó szint alatt sem célszerű folytatni az oktávok hozzáadását, ugyanezen okok miatt.

A zajfüggvény létrehozása után a kapott eredményeket interpolálni kell. Az alap interpolációs függvény három paramétert vár, `a` és `b` az interpolálandó értékek, `x` pedig egy 0 és 1 közötti szám. Az interpolációs függvény visszatérési értéke x függvényében egy `a` és `b` közötti szám, ha `x`=0 akkor `a` –val, ha `x`=1 akkor `b` –vel, ha `x` 0 és 1 közötti értékkel rendelkezik, akkor `a` és `b` közötti értékkel tér vissza.

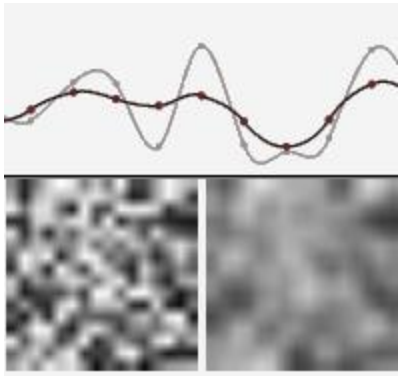
A lineáris interpoláció a leggyorsabb és egyben legrosszabb eredményeket produkáló interpoláció, mert két pontot csak egyenesekkel tud összekötni. Használata csak akkor célszerű, ha valós időben szeretnénk Perlin zajt generálni.

```
function Linear_Interpolate(a, b, x)
    return  a*(1-x) + b*x
end of function
```

A koszinusz interpoláció azért sokkal jobb mert, a pontokat görbékkel köti össze, és csak kicsit erőforrás igényesebb mint a lineáris interpoláció.

```
function Cosine_Interpolate(a, b, x)
      ft = x * 3.1415927
      f = (1 - cos(ft)) * 0.5

      return  a*(1-f) + b*f
  end of function
```



11. ábra

Harmadfokú interpoláció adja a legszebb eredményt, viszont cserébe a leglassabb interpolációk egyike. Ez az interpoláció kettő helyett 4 bemenő pontot vár.

```
function Cubic_Interpolate(v0, v1, v2, v3,x)
      P = (v3 - v2) - (v0 - v1)
      Q = (v0 - v1) - P
      R = v2 - v0
      S = v1

      return Px^3 + Qx^2 + Rx + S
  end of function
```

Interpoláció mellett lehetséges még „kisimítani” a zajt, így kevésbé fog véletlenszerűnek tűnni. Ahelyett hogy a zaj függvény értékét egy pontban vizsgálnánk, a kiválasztott pont környezetében lévő értékeket átlagoljuk. A *11. ábrán* észrevehetjük a különbséget az eredeti illetve a finomított zaj között. Látható, hogy a finomított zaj sosem éri el az eredeti zaj kiugróan magas, vagy alacsony értékeit, illetve a frekvencia körülbelül megfeleződik. A függvény 2 illetve 3 dimenziós változatainál ez a simítás még hasznosabb, mivel az eredmény négyszögletűségét csökkenti, ugyanakkor hátránya, hogy a kontraszt is csökken.

1 dimenziós simított zaj

```
function Noise(x)
...
end function

function SmoothNoise_1D(x)
  return Noise(x)/2  +  Noise(x-1)/4  +  Noise(x+1)/4
end function
```

2 dimenziós simított zaj

```
function N(x, y)//noise
    ...
end function

function SmoothNoise_2D(x>, y)

corners=(N(x-1, y-1) + N(x+1, y-1) + N(x-1, y+1) + N(x+1, y+1))/16
sides=(N(x-1, y) + N(x+1, y) + N(x, y-1) + N(x, y+1))/8
center =N(x, y)/4

 return corners + sides + center
end function
```

Ezek után már csak össze kell rakni az előbb részletezett függvényeket, ahhoz, hogy Perlin zajt kapjunk. Fontos megjegyezni, hogy a Perlin zaj nem több, mint interpolált zaj függvények összege, tehát a Perlin zaj is csak egy függvény. Egy vagy több bemeneti paraméter megadása után visszatér egy számmal. A Perlin zaj függvény fő része a ciklus, amelyben minden iteráció során hozzáadja a következő oktávot kétszeres frekvenciával a végeredményhez.

2 dimenziós Perlin zaj

```
function PerlinNoise_2D(float x, float y)

total = 0
p = persistence
n = Number_Of_Octaves - 1

      loop i from 0 to n

frequency = 2^i
amplitude = p^i

total=total+I_Noise(x*frequency,y*frequency)*amplitude

      end of i loop
      return total
  end function
```

A Perlin zajt rengeteg féle módon lehet hasznosítani, például az 1 dimenziós változatát virtuális élőlények természetes mozgásának modellezésére. Továbbá számítógép által generált egyenes vonalak „zajosítására", így természetesebbnek ható vonalakat lehet létrehozni. Kétdimenziós Perlin zajnak a legegyértelműbb felhasználási lehetősége a domborzatgenerálás. Ellentétben a felosztásos módszerrel, ezzel az algoritmussal nem kötelező a memóriában tartani a magasságtérképet, illetve a magasságtérkép bármely pontjának értéke könnyen kiszámolható. Ezeken kívül hatalmas előnye az algoritmusnak, hogy elméletileg végtelen nagyságú magasságtérkép generálható, tetszőleges részletességgel. A domborzat tulajdonságai is könnyen definiálhatóak. Domborzaton kívül textúrák, illetve felhő textúrák generálására is kifejezetten alkalmas az algoritmus. Háromdimenziós zajjal lehetséges térbeli felhők generálása, vagy animált 2 dimenziós felhők létrehozása.

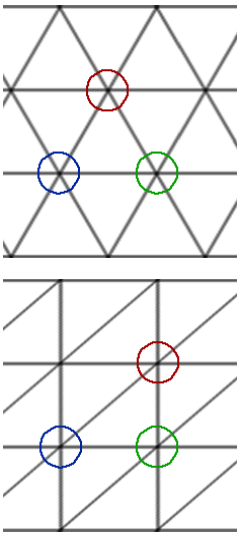
2.5. Simplex zaj (13)

A Simplex zajt szintén Ken Perlin alkotta, tulajdonképp felfogható, mint Perlin zajnak a továbbfejlesztése. Főként a magasabb dimenziókban történő gyors működés elérése volt a cél. Mivel a Perlin zaj egy négyzetes rácson alapul, minden értéket lineárisan kell interpolálni. Ez az interpoláció a dimenziók számának növelésével nagyon gyorsan növekszik. Három dimenzió esetén még csak 7 interpolációt kell végezni, de például 15 dimenziónál, már minden pontra 32767 lineáris interpolációt kell elvégezni iterációnként. A Perlin zaj futási ideje $O(N * 2^n)$ míg a simplex zajnak csak $O(N^2)$, ahol N a dimenziók

számát jelöli. Emiatt a simplex zaj kisebb dimenzióknál csak minimálsan lesz gyorsabb, viszont magas dimenzióknál nagyságrendekkel gyorsabb futási ideje lesz.

A Simplex zaj struktúrája hasonló a Perlin zajéhoz, mindkét algoritmus ugyanazt a permutációs táblát, és gradiens táblát használja. Egy adott ponthoz tartozó értéket ugyanúgy a környező pontok gradienseiből számítja ki. A szimplex zajnál használt rács viszont nem négyzetes, illetve a gradiensekből más módon számítja a végső értéket.

A Perlin zaj gradiensei egy négyzetes rácson helyezkednek el. Ez 3 dimenzióban kockákból épített rács, és így tovább magasabb dimenzióknál. A gond ezzel, hogy így több sarokpont jön létre a kelleténél. N dimenzióban a legkevesebb sarokpontból létrehozható alakzatot szimplexnek nevezik, innen ered az algoritmus elnevezése. Két dimenzióban egyenlő oldalú háromszögekkel le lehet fedni egy felületet, így kevesebb ponttal kell számolni, így elméletileg kevesebb interpoláció is szükséges. Tulajdonképp ennél az algoritmusnál nincs is szükség lineáris interpolációkra, helyette egy bizonyos csillapítás függvénnyel ki lehet számolni az adott pont hozzájárulását, majd ezeket az eredményeket összeadva kapjuk a végső eredményt. Ezzel a módszerrel magasabb dimenziókon sokkal kevesebb erőforrás felhasználásával érjük el ugyanazt az eredményt.



12. ábra Ferdítés előtt, illetve utána

Első lépésben a kiválasztott pont szomszédjait kell megkeresni. Ez Perlin zaj esetén egyértelmű, de Simplex zaj esetén, mivel háromszögekből épül fel a rács, kicsit bonyolultabb. Az ölet az, hogy a rácsot az átló mentén elferdítjük, így látszólag egy háromszögekből felépült négyzetes rácsot kapunk. Ezek után a kékkel illetve pirossal jelzett pontok megtalálása nem okoz gondot. A háromszög harmadik pontja ezek után lehet akár a zölddel jelzett, illetve az azzal szemben lévő pont is. Ha az adott pont X koordinátájának tört része nagyobb mint az Y koordinátájának a tört része, akkor a háromszög harmadik pontja az alsó(zölddel jelzett) pont lesz. Logikusan ha a pont X koordinátájának tört része kisebb mint az Y koordinátájának a tört része, akkor pedig a zölddel jelzett ponttal szemközt lévő pont lesz a háromszög harmadik pontja. Fontos megjegyezni hogy a rács pontok számozása tetszőleges, viszont konzisztens kell hogy legyen. Ahhoz hogy a teret elferdítsük, illetve visszaállítsuk a teret a következő értékekkel kell megszorozni a koordinátákat.

```
n = dimenziók_száma
elferdítés =(sqrt(n+1)-1)/n
visszaferdítés=(n+1-sqrt(n+1))/(n*(n+1))
```

Második lépésben, mivel már ismerjük egy tetszőleges pontnak a szomszédos koordinátáit, tudunk egy gradienst hozzárendelni. Ezt ugyanazzal a permutációs tábla metódussal tesszük.

```
gradb =perm[(cornerbx +perm [cornerby &255])&255]&7;
gradm =perm [(cornermx +perm [cornermy &255])&255]&7;
gradt =perm [(cornertx +perm [cornerty &255])&255]&7;
```

Ahol `gradb, gradm, gradt` az alsó, középső illetve felső pontokhoz tartozó gradienst jelentik.

Harmadik lépésben számítjuk ki a környező pontok hozzájárulását az aktuális ponthoz. Minden sarokpontra a következő képletettel kell számolni:

$$sugár = 0.5 - x_táv^2 - y_táv^2$$

Amennyiben a sugár kisebb vagy egyenlő mint nulla, akkor nullára kell állítani, mivel egy bizonyos távolságon túl már nem szabad hogy hozzájáruljon a végeredményhez az adott pont érétke. Ha a sugár nagyobb mint nulla, akkor a hozzájárulás a következő képlettel számítható ki:

$$zaj = sugár^4 * dot(gradiens * x_táv * y_táv)$$

A gradiens a második lépésben, az adott ponthoz választott gradiens értéke. Utolsó lépésben már csak összegezni kell a három szomszédos pontból számított hozzájárulásokat, nincs szükség interpolációra.

Minőségében nincs túlságosan nagy eltérés, bár a különbséget észre lehet venni, és bár 2 dimenzióban a sebesség növekedés sem érzékelhető, mindössze ~10%, magasabb dimenziókban nagyságrendekkel gyorsabb.

3. Felhasznált eszközök, technológiák

3.1. C++ és Java

A C++ vitathatatlanul a legsokoldalúbb és legelterjedtebb programozási nyelvek egyike. A nyelv ugyanúgy lehetőséged ad nagy teljesítményű kód írására, mint kifejező elvonatkoztatások, és tervezési konstrukciók létrehozására. A nyelv nem tökéletes, de remek kompromisszumot köt a potenciálisan egymásnak ellentmondó programozás nyelvi képességeknek. A C++ egyesíti az „alacsony szintű” programozást, amely egy adott architektúrára lett szabva, a „magas szintű” programozással, amely folytán a kód teljesen elvonatkoztatott lehet bármely architektúrától. (14)

A Java nyelv tervezése során a hordozhatóság, biztonság és a több szálon való programozás megvalósítása volt a cél. A nyelv szintaktikája a sok hasonlóságot mutat a C nyelv szintaktikájával. Továbbá a Java nyelv eredetileg egyszerűbbre lett tervezve, mint a C++, de a fejlesztések során ez az egyszerűség valamilyen szinten megszűnt. Az 1999 és 2009 közötti időszakban, a programozási ipar főleg vállalati megoldásokra specializálódott, a virtuális gépekre támaszkodó nyelvek fejlődése emelkedett ki. Ez főként a produktivitásnak kedvezett, a teljesítmény rovására, amely ez idő tájt tökéletesen érthető volt, amikor a számítógépek növekvő teljesítménye miatt szükség egy könnyű programozási nyelvre, amelyet nem csak könnyű használni, de gyorsan is meg lehet tanulni. (15)

3.1.1. Teljesítmény

A számítási teljesítmény az erőforrás használatának a mérése, amikor a hardver és szoftver együttesen elvégez egy számítást. Nagyobb teljesítmény a definíció szerint az, hogy ha a számítás kevesebb erőforrást használ fel. Erőforrás alatt értünk memóriát, sávszélességet, állandó tárhelyet, és processzor ciklusokat. Mivel az utóbbin kívül a számítógépek bővelkednek az erőforrásokban, a köznyelvben a teljesítmény a minél kevesebb processzor ciklus használatot jelenti, amely általában a gyorsabb lefutással rendelkező számítás. Két különböző szoftveres nyelv összehasonlításához fix hardveres platform, illetve (általában relatív) mérések szükségesek kettő vagy több szoftveres alrendszeren. A korai Java verziók nagyságrendileg elmaradtak a statikusan fordított nyelvek teljesítményétől, mivel a programutasítások kisszámú gépi kóddá fordulnak le C++ esetében. A Java fordító ezzel

szemben jó néhány instrukciót használ ugyanahhoz az utasításhoz, amelyet ezután a Java JVM interpretál. A beágyazott rendszereknél ez a helyzet még mindig fennáll, a JIT (futásidejű fordítás) fejlődése miatt ez a teljesítmény különbség csökkent a két nyelv között, különösen a hosszú ideig futó szerver alkalmazások esetében. (15)

3.1.2. OpenGL

A szakdolgozatomhoz készült programot eredetileg Java nyelven kezdtem írni, amely végeredményben csak részben bizonyult jó döntésnek. A Java nyelvhez a JOGL(Java Binding for the OpenGL API) könyvtárcsomagot használtam az OpenGL utasítások hívására, amely kényelmi, és hibakeresési szempontokból remekül helytállt. A kirajzoláshoz definiálni kellett egy osztályt, amely implementálja a `GLEventListener` interfészt. Ez az interfész az `init`, `display`, `dispose`, `reshape` metódusok definiálását teszi kötelezővé, melyek értelem szerűen hívódnak meg, így viszonylag könnyen lehet jól olvasható kódot írni. Ezen kívül az OpenGL kódban történő hibakeresés is nagyságrendekkel könnyebb, mint C++ nyelven. Mivel míg C++ esetén minden OpenGL hívás után a `glGetError()` metódussal kell ellenőrizni, hogy történt-e hiba, addig Java nyelven lehetőség van rá, hogy az általánosan használt `GL` objektum helyett, a `DebugGL` objektumot használjuk, amely ezt a hibaellenőrzést minden OpenGL hívás után ellenőrzi, és amennyiben hibát talált egy `GLException` kivételt dob, amely alapján könnyen meg lehet találni a hiba okát, és helyét. Ezek a funkciók, illetve az egyetemi tanulmányaim során szerzett tapasztalatok Java nyelven, nagyban megkönnyítették, hogy megértsem az OpenGL működését.

Mindezen előnyök ellenére mégis úgy döntöttem, hogy a már félig elkészült programot újrakezdem C++ nyelven, aminek az egyik fő oka az volt, hogy szerettem volna a Java mellett egy másik programozási nyelven is ismereteket szerezni, továbbá sokan úgy tartják, hogy az OpenGL használatához a C++ a legjobb nyelv. Az eddigi tapasztalataim ezt alátámasztják. Ugyan szinte az összes JOGL-ben megszokott kényelmi funkció hiányzik, miután - már ha csak egy kicsit is -, de tapasztalt voltam az OpenGL működésével kapcsolatban, nem is volt igazán ezekre szükség.

3.2. OpenGL (16) (17)

Az OpenGL (Open Graphics LIbrary), egy API, a grafikus hardver felé. Az API néhány száz eljárásból és függvényből áll. A legtöbb OpenGL funkcióhoz szükséges hogy a grafikus hardver tartalmazzon framebuffert. Rengeteg függvényhívás különböző geometriai alakzatok kirajzolását vezérli, mint például pontok, vonalak, és poligonok, de a kirajzolás folyamata a framebuffer jelenlétét megköveteli.

A programozó szempontjából az OpenGL utasítások halmaza, amely lehetőséged ad shader programok definiálására, illetve a shaderek által használt adatok meghatározására, és különböző OpenGL állapotok változtatására. Az adat általában valamilyen kettő vagy háromdimenziós geometriai alakzat, a shaderek pedig ezeknek a geometriai alakzatoknak a feldolgozását, raszterizálását, illetve egyéb effektek, például fény szimulálását vezérli. Ezen kívül a *fragmentek* kirajzolása a framebufferbe is a shaderek feladata.

Egy tipikus OpenGL program egy ablak megnyitásával kezdődik, amelybe a rajzolás fog majd történni. Ezután létre kell hozni egy OpenGL kontextust, és asszociálni kell a már létező ablakkal. Amint a kontextus létrejött, a shadereket, textúrákat és a geometriát is létre lehet hozni, majd ki lehet adni a kirajzolás parancsot, amely az előre meghatározott geometriai alakzat részeit továbbítja a shaderre. Maga a kirajzolás utasítás egyszerű geometriai objektumokat határoz meg, mint például pont, vonal, poligon, amelyeket tovább lehet manipulálni a shaderekben. Továbbá léteznek olyan utasítások is, amelyek közvetlenül a framebuffert változtatják, pixel írás/olvasással.

3.2.1. Shader

A *shader* tulajdonképp egy utasítás csoportot határoz meg, amelyek akkor érvényesülnek, amikor az adat különböző szakaszain halad át az OpenGL csővezetéknek, amely a geometria csúcspontjainak meghatározásával indul, és a fragmenteknek a framebufferbe való kiírásával végződik.

A shader használatához először egy shader objektumot kell létrehozni, amelybe az adott shader forráskódját kell betölteni, majd fordítani. Egy shader objektum megfelel egy szakasznak az OpenGL csővezetékben, és gyakran úgy említik, mint a shader *típusa*. Egy másik megoldás, hogy előre fordított bináris kódként töltjük be a shader programot a shader objektumba. Egy vagy több shader objektumot lehet hozzá rendelni a program objektumhoz. A program objektum az összekapcsolás után legenerálja a futtatható kódot, a

shader forrásokból. Amikor a program objektum hozzákötődik a shader szakaszhoz, a aktív program objektummá válik.

A shaderek fordítása előtt szükséges van néhány információ átadására a fordítónak. Egy parancssoros fordítónál ezek az információk lennének a parancssori paraméterek. A GLSL fordítás esetén viszont ezek az információk a programozási nyelvnek a részei.

3.2.1.1. Verzió

A GLSL számos felülvizsgálaton ment keresztül, jó néhány közülük elég lényeges. Az OpenGL specifikáció alapján minden OpenGL verzió feltétele hogy egy bizonyos verziószámú GLSL-t támogasson. A GLSL verzió definiálása a `#version 150` direktívával történik, amely az 1.50 verziójú fordítót hívja meg, vagy hibát dob, amennyiben a grafikus kártya nem támogatja azt.

3.2.1.2. Típusok

A GLSL jó néhány adattípust definiál, illetve az új adattípusok létrehozásának menetét is leírja. Bármelyik típus dupla pontosságú változata csak az OpenGL 4.0 vagy afölötti verziókban érhető el.

Skalár típusok

- `bool`: logikai változó, értéke lehet `true` vagy `false`
- `int`: előjeles 32-bites egész szám
- `uint`: előjel nélküli 32-bites egész szám
- `float`: IEEE-754 egyszeres pontosságú lebegőpontos szám
- `dobule`: IEE-754 dupla pontosságú lebegőpontos szám

Vektor típusok

Mindegyik vektor típus, beleértve a boolean vektorokat, rendelkeznek 2, 3 illetve 4 komponensű változattal. Az `n` betű jelzi a komponensek számát.

- `bvecn`: boolean típusokból álló vektor
- `ivecn`: előjeles egész számokból álló vektor

- `uvecn`: előjel nélküli egész számokból álló vektor
- `vecn`: egyszeres pontosságú lebegőpontos számokból álló vektor
- `dvecn`: kétszeres pontosságú lebegőpontos számokból álló vektor

A vektor értékekre ugyanazokat a matematikai operátorokat lehet alkalmazni, mint az alap skalár értékekre. Ezek a műveletek minden vektorkomponensen elvégződnek. Az operátorok működésének feltétele, hogy a vektorok komponenseinek a száma megegyezzen.

A vektorokkal kapcsolatban érdemes említést tenni az úgynevezett „swizzling”-ről is. Ennek a lényege, hogy a vektorok komponenseire az x,y,z,w betűkkel hivatkozhatunk, a hivatkozások sorrendje nincs megkötve.

```
vec4 v0 = vec4(0,1,2,3);
vec3 v1 = v0.zyx;             // v1 értéke (2,1,0)
vec4 v2 = v1.xxxx;            // v2 értéke (0,0,0,0)
v0.wzyx = vec4(1,2,3,4);      // v0 értéke (4,3,2,1)
v1.xx = vec2(4,4);            //hiba!
```

Az x,y,z,w hivatkozásokon kívül használhatóak még az r,g,b,a betűk a színekhez, illetve az s,t,p,q betűk a textúra koordinátákhoz. Ezek kizárólag a szintaktikai szépítést szolgálják. Működés szempontjából bármelyik típusú hivatkozás ugyanúgy működik, viszont fontos megjegyezni, hogy a hivatkozás típusokat nem szabad egymással keverni, tehát a `.xrq` hivatkozás, fordítás idejű hibához vezet!

Mátrixok

A mátrixok komponensei kizárólag lebegőpontos típusok lehetnek. Hasonlóan a vektorokhoz, a mátrixok oszlopainak illetve sorainak méretei is 2, 3 vagy 4 lehet.

- `matnxm`: Egy `n` számú oszloppal, és `m` számú sorral rendelkező mátrix, például: `mat3x4`.
- `matn`: Egy `n` számú oszloppal és sorral rendelkező mátrix

Sampler

A textúra hozzáférés nem csak egy memóriacímről való olvasást jelent. Különböző szűréseket és egyéb műveleteket kell alkalmazni a textúrákra, illetve a textúra koordináták interpretálása is része a textúra hozzáférésnek, emiatt ez egy összetett művelet. Jó néhány

típusú mintavevő létezik, ezek három fő kategóriába vannak sorolva; lebegőpontos, előjeles és előjel nélküli egész mintavevők. A GLSL-ben lévő mintavevő neve a típusára utal. Ezek a nevek hasonló csoportosulást mutatnak, mint a vektor típusoknál, tehát az egyszeres pontosságú mintavevő neve `sampler`–el kezdődik, az egész típusú mintavevő `isampler`, az előjel nélküli mintavevő pedig `usampler`. Ha a textúra kép formátum nem egyezik meg a mintavevő formátumával, akkor bármelyik olvasási próbálkozás a mintavevőről nem definiált eredményeket produkál.

Struktúrák

Egy struktúrát C++-hoz hasonló módon kell definiálni, néhány megkötéssel. A GLSL nem támogatja a névtelen struktúrák létrehozását, és a struktúrának minimum egy tag deklarációval kell rendelkeznie. Továbbá a struktúrák definiálása során nem definiálhatunk egy másik struktúrát, de egy struktúra tagja lehet egy másik struktúra.

```
struct Light
{
  vec3 eyePosOrDir;
  bool isDirectional;
  vec3 intensity;
  float attenuation;
} variableName;
```

A fenti struktúra definíció `Light` típusú struktúrát hozz létre, valamint egy `variableName` nevű változót. A struktúra nem tartalmazhat „átlátszatlan” változót, mint például mintavevő, kép, vagy atomi számláló. A struktúra tagjai a pont operátorral érhetőek el.

Shader szakasz be- és kimenetek

Globális változók amelyeket `in` minősítővel látunk el, az adott shader szakasz bemeneti változói lesznek. Ezeknek az értékeit az előző shader szakasz adja, általában interpolált érték, az előző szakasz többszöri lefutása után. Ezek a változók nem számítanak konstansnak, azonban az értékük nem változtatható meg programkódból.

Hasonlóan, ha egy globális változót `out` minősítővel látunk el, akkor a változó az adott shader szakasz kimeneti változója lesz. Ezek a változók a következő shader szakasz bemenetei lesznek. Az adott szakaszon lévő összes kimeneti változót be kell állítani a végrehajtódás során, kivéve azokat a változókat, amelyeket a következő szakasz nem

olvas. A ki és bemeneti változó típusa bármilyen alap típus lehet, tehát például struktúra nem lehet, viszont tömb igen.

A shader szakaszok közötti kommunikáció ezekkel a változókkal történik. Minden változónév és annak típusa meg kell hogy egyezzen.

Uniform

Az uniform változó egy globálisan deklarált változó `uniform` minősítővel. Ezek a változók paraméterként viselkednek a shader program egészének szempontjából. Értékük a shader program objektumban tárolódik. Az uniform változó értéke nem változik meg a shader program lefutása során, tehát egy kirajzolás alatt az értéke konstansnak tekinthető, a shader kódból való megváltoztatása fordítás idejű hibát eredményez, ebből kivételt képez a konstansokkal végzett standard inicializálás. Továbbá uniform változó nem kaphat sem `in`, sem `out` minősítőt.

3.2.1.3. Vertex shader

A vertex shader az első programozható shader szakasz az OpenGL csővezetéken. Ez a szakasz a különálló poligon csúcsok feldolgozására specializálódott. Az adatokat a vertex attribute adathalmazból kapja, olyan formátumban, ahogy az a vertex array object-ben lett meghatározva. A vertex shader egy lefutás során egy poligon csúcspontot dolgoz fel, és a kimenetén is csak egy csúcspont jöhet létre. Tipikus művelet ebben a szakaszban a csúcspont transzformációk elvégzése, csúcspontonkénti megvilágításhoz való számítások elvégzése, illetve egyéb számítások előkészítése a további shader szakaszokhoz.

3.2.1.4. Tessellation

Ellentétben a vertex shaderrel, ez a szakasz a csúcspontfeldolgozásban, teljes mértékben opcionális. Ezen a szakaszon lehetőség van a vertex shaderből beérkező csúcspont halmazok kisebb darabokra való felosztására. Ez a művelet három szakaszból áll, ebből csak két szakasz programozható. Amennyiben ezek használatban vannak, a kirajzoláshoz használt célpont az egyéb esetekben használt `GL_POINTS`, `GL_LINES`, `GL_TRIANGLES`, stb. helyett kötelezően `GL_PATCHES`. A `GL_PATCHES` egy általános célú primitív típus, amelynél a programozó határozza meg, hogy hány csúcspont után kezdődjön az új primitív. Ez a szám a `glPatchParameteri(GL_PATCH_VERTICES, érték)` paranccsal

határozható meg. Ennek az értéknek a maximuma implementációfüggő, de sosem kevesebb, mint 32.

Tessellation control shader

Ez a szakasz vezérli a felosztásoknak a mértékét, illetve definiálja egy poligonnak a méretét, továbbá különböző szűréseket végez a bejövő adathalmazon. A fő célja, hogy átadja a felosztások szintjét Tessellation primitive generator-nak, valamint a Tessellation Evaluation szakaszt ellássa adatokkal.

A végrehajtás során minden lefutáskor egy csúcspontot generál egy kimeneti patchhez. Minden bemeneti patchhez a kirajzolás során `n` számú shader hívás történik, ahol `n` a csúcspontok száma a kimeneti patchben. Tehát egy kirajzolás utasítás, amely 20 patch-t rajzol ki, amelynek darabonként van 4 csúcspontja, összesen 80 különálló TCS hívást eredményez. Különböző meghívások, amelyek ugyanazon kimeneti patchhez szolgáltatnak adatokat, összekapcsoltak. Ezek a hívások osztoznak ezen szakasz kimeneti értékein, tehát tudják olvasni azokat az adatokat, amelyeket más meghívások írtak ki egyazon patchhez, de ahhoz hogy ez megfelelő módon működjön, szükséges egy szinkronizációs mechanizmus, amely garantálja, hogy mindegyik más meghívás már befejeződött.

A kimeneti patch méret az a szám, amely meghatározza, hogy a patch hány csúcsponttal rendelkezzen. Ez a szám továbbá definiálja, hogy hány TCS meghívást kell majd elvégezni, hogy az adott patchhez tartozó adatokat kiszámolja. A kimeneti méret nem kötelezően ugyanaz, mint a bemeneti méret. A kimeneti csúcspontok számát egy patchben a következőképp kell definiálni.

```
layout(vertices = patch_méret) out;
```

Ahol a `patch_méret` egy egész számú konstans kifejezés, amely nullától nagyobb.

Minden bemeneti változó a vertex shaderből a TCS-be érkezve tömbökbe összesítődik, amelyeknek a mérete megegyezik a bemeneti patch mérettel. Minden TCS meghívás egy adott patchhez ugyanazon adatokhoz fér hozzá, ebből kivétel a `gl_InvocationID`, amely minden hívásnál különbözik.

A bemeneti változókat méret nélküli tömbökként deklarálhatjuk:

```
in vec3 vPos[];
```

Amennyiben a bemeneti patch méretétől nagyobb tömb indexszel próbáljuk olvasni a tömböt, az eredmény minden esetben határozatlan. Nem számít hibának, ha a bemeneti változók rendelkeznek interpolációs minősítővel, viszont nincs rájuk hatással.

A TCS a következő beépített változókkal rendelkezik:

- `in int gl_PatchVerticesIn`: a csúcspontok száma a bemeneti patchben
- `in int gl_PrimitiveID`: a pillanatnyi patch sorszáma az adott kirajzolásban
- `int in gl_InvocationID`: a pillanatnyi TCS meghívás sorszáma

A fentieken kívül a TCS rendelkezik még az alábbi vertex shader kimenetéről érkező beépített struktúrával:

```
in gl_PerVertex
{
vec4gl_Position;
floatgl_PointSize;
float gl_ClipDistance[];
} gl_in[gl_MaxPatchVertices];
```

A TCS kimenetei közvetlenül érkeznek a TE-be, bármiféle interpoláció nélkül. Ezek a kimenetek lehetnek csúcsonkénti kimenetek, vagy patchenkénti kimenetek. A csúcspontonkénti kimenetek tömbökbe összesítődnek, amelyet a következőképp kell deklarálni:

```
out vec3 vPos[];
patch out vec3 vPos;
```

A tömb mérete mindig a kimeneti patch méretével fog megegyezni, tehát ezt nem kell újradefiniálni. A patchenkénti kimenetek nem rendeződnek tömbökbe, kivéve, ha a programozónak nem kifejezetten ez a szándéka. Minden TCS hívás ugyanazzal az értékkel látja ezt a változót.

Talán a legfontosabb része a TCS-nek a kimeneti, pontosabban a következő két változó:

```
patch out float gl_TessLevelOuter[4];
patch out float gl_TessLevelInner[2];
```

Ezek a változók határozzák meg a külső és belső felosztását a poligonnak. A pontos értelmük függ a patch típusától, és más beállításotól amelyeket a TES-ben definiálunk. Továbbá a TCS a következő beépített struktúrával is rendelkezik:

```
out gl_PerVertex
{
vec4gl_Position;
floatgl_PointSize;
float gl_ClipDistance[];
} gl_out[];
```

Ezt a struktúrát szabadon lehet használni, de mivel a TCS-t kötelezően kell hogy kövesse egy TES, ezért gyakorlati felhasználási lehetősége ennek ebben a shader szakaszban nincs.

Tessellation evaluation shader

Ezen a szakaszon a TCS-ből érkező adatok, és egyéb paraméterek alapján kiszámolja a patch csúcspontok interpolált értékeit, majd a következő shader szakaszra továbbítja azokat. Minden egyes TES meghívás egy csúcspontot generál. A TES meghívások száma függ az implementációtól, de minimum egy meghívás történik minden csúcspontra, és az is garantált, hogy ugyanazon csúcspontra nem fog egynél többször meghívódni. Ugyanakkor hasonlóan a vertex shaderhez, egy bemenetre egy kimenetet produkál.

A TES jelenlétében a program, vagy a program csővezeték az, amely eldönti azt, hogy a feldarabolt primitív generálás szakasz megtörténjen-e. Emiatt néhány paraméter definiálása ezen a szakaszon történik meg. Ezeket a paramétereket a következőképp kell definiálni:

```
layout(abstract_patch_type, spacing, primitive_ordering)in;
```

Az `abstract_patch_type` a patch típusát határozza meg. Ezt az opciót kötelező megadni. A lehetséges értékek a következők:

- `isolines`: A patch négyzetes blokkból álló párhuzamos vonalakból áll. a kimenet vonalak sorozata.
- `triangles`: A patch egy háromszög, a kimenet háromszögek sorozata.
- `quads`: A patch egy négyszög, a kimenetek háromszögek sorozata.

A `Spacing` paraméter határozza meg a feldarabolt csúcspontok közötti teret. Ez a paraméter opcionális, az alapértelmezett értéke az `equal_spacing`. A lehetséges értékek a következők:

- `equal_spacing`: Két tetszőleges csúcs között egyenlő távolság lesz.
- `fractional_even_spacing`: Mindig lesz egy páros számú szegmens a szimmetrikus oldalain az élnek. Két csúcsot kivéve a szegmenseknek fix hosszúsága lesz, a maradék csúcspontok két egyenlő hosszúságú szegmensre lesznek osztva.
- `fractional_odd_spacing`: Hasonló, mint az előző paraméter, azzal a kivétellel, hogy páratlan számú szegmensek lesznek.

`Primitive ordering`. A poligonok kibocsátása során a sorrend a selejtezés szempontjából fontos. A darabolás folyamata egy absztrakt patchen végződik el, amely nincs hozzárendelve egy koordináta rendszerhez sem. A TES felelőssége, hogy az absztrakt patch koordinátákból generáljon egy valós koordinátarendszerhez kötött pozíciók. Ez a paraméter opcionális, az alapértelmezett értéke `ccw`, a másik lehetséges értéke `cw`.

3.2.1.5. Geometry shader

Hasonlóan a TCS és TES shaderekhez a GS is opcionális. Ezen a szakaszon egy bemenő csúcspontból több kimenő csúcspontot generálhatunk. Implementációtól függően léteznek korlátok arra, hogy maximálisan hány csúcspontot generálhatunk egy GS meghívás során. A GS programok egy adott számú csúcspontokat várnak a bemeneten, és egy adott számú csúcspontokat bocsátanak ki. Tulajdonképp a TES-hez hasonló darabolásra is lehet használni a GS-t, de általánosságban véve nem erre lett tervezve. A főbb okok a GS használatára a következők:

- Réteges kirajzolás: Egy adott poligon többszöri kirajzolása.
- Transform feedback: Jellemzően számítási feladatok elvégzéséhez.

A GS úgy lett tervezve, hogy egy bizonyos típusú primitív típust fogadjon el a bemenetén, és egy másik primitív típust bocsásson ki a kimenetén. Az bemenő elfogadott primitív típust a következőképp lehet definiálni:

```
layout(input_primitive) in;
```

Az `input_primitive` paraméter típusának kötelezően meg kell egyeznie azzal, amelyet a kirajzolás utasításnál használunk. Az elfogadott értékek, illetve a hozzátartozó OpenGL primitív típusok a következőek.

GS bemenet	OpenGL primitívek
points	GL_POINTS
lines	GL_LINES, GL_LINE_STRIP, GL_LINE_LIST
lines_adjacency	GL_LINES_ADJACENCY, GL_LINE_STRIP_ADJACENCY
triangles	GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN
triangles_adjacency	GL_TRIANGLES_ADJACENCY, GL_TRIANGLE_STRIP_ADJACENCY

A kimeneti primitívet a következőképp kell definiálni:

```
layout(output_primitive, max_vertices = vert_count) out;
```

Ahol az `output_primitive` értéke lehet `points, line_strip, triangle_strip.` Ezek az értékek ugyanúgy működnek, mint az OpenGL megfelelőik.

Fontos még megemlíteni a GS esetén két beépített függvényt is. Az `EmitVertex()` utasítja a rendszert, hogy írja ki a kimeneti értékeket. A függvény hívása után a kimeneti változók értékei nem definiált értékeket tartalmaznak. Az `EndPrimitive()` utasítás befejezi a pillanatnyi primitívet, és újat kezd.

3.2.1.6. Fragment shader

Az FS a primitívek raszterizálása után fut, minden pixel mintára egy „fregmens” generálódik. Minden fregmens rendelkezik egy ablak koordinátarendszerben lévő koordinátával, és többek között tartalmazza az interpolált csúcsonkénti kimeneti értékeket

az előző shader szakaszból. Az FS kimenete egy mélyég érték, egy stencil érték, illetve nulla vagy annál több szín érték, melyek ki lesznek írva a framebufferbe. Az FS egy fregmens bemenetre, egy fregmenst produkál a kimeneten.

Az FS bemenetei lehetnek rendszer által generált, illetve programozó által deklarált bemenetek is. A programozó által deklarált változók interpolálva lesznek, a változóhoz megadott interpolációs minősítő alapján. Fontos megjegyezni, hogy az FS bemeneti változóinak a neve meg kell hogy egyezzen az FS előtti shader szakasz kimeneti változóinak a nevével.

3.2.2. OpenGL objektumok

Egy OpenGL objektum egy olyan struktúra, amely valami féle állapotot tárol. Amikor ez az objektum a kontextushoz kapcsolódik, az állapot, amelyet az objektum tárol leképeződik a kontextus állapotára. Tehát a kontextus állapotának megváltoztatása eltárolódik az adott objektumban, és az utasítások, amelyek a kontextus állapotát változtatják, az objektum állapotára is hatással lesznek. Az OpenGL egy „állapot gépezet”. Különböző API hívások változtatják az állapotot, vagy akár lekérdezést végeznek róla.

3.2.2.1. Objektum létrehozása

Ahhoz hogy létrehozzunk egy objektumot, az objektum nevét kell legenerálni, amely valójában egy nullától nagyobb előjel nélküli integer szám lesz. Ez az érték egy referencia a létrehozott objektumra. Az objektum csak akkor kapja meg az alapértelmezett állapotát, amikor először hozzákapcsoljuk a kontextushoz. A függvény, amely létrehozza az objektumokat, és a referenciáit a következő:

```
glGen*(Glsizei n, Gluint *objects);
```

Ahol * a létrehozandó objektum neve többes számban; `n` számú objektum fog létrejönni, amely az `objects` nevű tömbben tárolódik el.

3.2.2.2. Objektum kapcsolása

Mivel az OpenGL objektumok állapotok halmazaként lett definiálva, ahhoz hogy módosítsunk egy objektumot először hozzá kell kapcsolni a kontextushoz. A kapcsolás hatására a bennük eltárolt állapot kerül át a kontextus állapotába. Ez azt jelenti, hogy

bármilyen utasítás, amely változtatja a kontextus állapotnak azon részét, amelynek az adott objektumhoz valamilyen köze van, nem a kontextus állapotát változtatja, hanem az objektum állapotát, ezáltal megőrzi a kontextus eredeti állapotát. Újonnan generált objektumok kapcsolása esetén egy új állapot jön létre az objektumhoz. Különböző objektumoknak különböző kapcsoló utasításuk van, viszont elnevezésben és paraméterszignatúrában hasonlóak:

```
glBind*(GLenum target, GLuint object);
```

Ahol a * az objektum típusa, és az `object` maga az objektum, amelyet kapcsolni szeretnénk. A `target` paraméter az, ahol a különböző objektumtípusok különböznek.

3.2.2.3. Zérus objektum

Az objektumok esetén a zérus referencia (GLuint értéke 0) külön van kezelve, illetve különböző objektumok is különféle módokon kezelik ezt az értéket. Zérus referenciát sosem ad vissza a `glGen*` függvény. A legtöbb objektum hasonlóan kezeli a zérus referenciát, mint a null pointert, tehát nem veszi objektumnak. Amennyiben egy ilyen objektumot próbálunk bármiféle kirajzolás művelethez használni, hibás működést fog eredményezni.

Textúrák esetében a zérus referencia az „alapértelmezett” objektumot jelzi. Ebben az esetben ezt az objektumot nem is lehet törölni.

3.3. GLM (18)

Az OpenGL Mathematics egy kizárólag C++ headerekből álló matematikai függvénykönyvtár, amely a GLSL specifikációja alapján készült. A függvénykönyvtár osztályokkal és metódusokkal látja el a programozót, melyek tervezése és implementálása során a GLSL konvenciókat és funkcionalitásokat követi. Így ha a programozó jártas a GLSL-ben akkor a GLM-et már könnyen tudja elsajátítani. A projekt nem korlátozott kizárólag a GLSL funkciókra. Tartalmaz például mátrix transzformációkat, kvaterniókat, véletlen szám generátorokat illetve procedurális zaj függvényeket is. Továbbá az OpenGL 3.0-s verzióban elavulttá vált mátrix függvényeket is helyettesíti.

4. Implementáció

Szakdolgozatom célkitűzése, hogy az előző fejezetekben említett domborzatgeneráláshoz használatos algoritmusok közül néhány implementációját megvalósítsam. A kiválasztott algoritmusok a vektoriális szorzat alapján működő Fault algoritmus, a Diamond-sqare algoritmus illetve Perlin zaj segítségével generált magasságtérkép. A választásom azért esett ezekre az algoritmusokra, mert a megvalósításuk viszonylag egyszerű, és látványos eredményt produkálnak. A magasságtérkép generálása és kirajzolása mellett próbálkoztam egy egyszerűbb dinamikus részletesség szint algoritmus implementálásával.

Fontosnak tartom megjegyezni, hogy a program célja nem az, hogy könnyen használható szoftver legyen, sem az hogy szép eredményeket produkáljon. Célom elsősorban az volt, hogy a C++ nyelv alapjait elsajátítsam, illetve a modern OpenGL működését megértsem. Modern OpenGL alatt a 4.0 verziószám fölötti funkcionalitásokat értem.

4.1. Inicializáló műveletek

4.1.1. Ablak és kontextus

Az OpenGL kontextus létrehozás, illetve az ablak létrehozása is megoldható az SFML függvénykönyvtár használatával. Tulajdonképp a kirajzoláshoz használt ablak az `sf::Window` objektum, amely egy ablak objektumtól elvárhatóan, a konstruktorában vár egy ablak méretet, egy ablak nevet, illetve lehetőség van a címsorral kapcsolatos beállítások átadására is. Az ablak konstruktorának lefutása során egy alapértelmezett OpenGL kontextus is létrejön, amely beállításait opcionálisan átadhatjuk a konstruktor paraméterei között.

4.1.2. OpenGL függvény pointerek beállítása

Ezen a ponton még nem hívhatunk OpenGL függvényeket. A grafikus kártya gyártójának kötelessége az OpenGL funkcionalitások pontos implementációja, az alapján, hogy az adott grafikus kártya milyen képességekkel rendelkezik. Nem lenne túl szerencsés, ha egy adott program csak egy adott videó kártyán egy adott driver verzióval működne megfelelően. Tehát a programnak futásidőben ellenőriznie kell, hogy mely függvény

hívások érhetőek el, és dinamikus kapcsolatokat kell létrehoznia azokkal. A GLEW(OpenGL Extension Wrangler) ezt a terhet veszi le a programozó válláról. (19)

A `glew.h` header inkludálása után a `glewExperimental` globális változó értékét `GL_TRUE` –ra kell állítani, ennek hatására lesznek elérhetőek az újabb OpenGL funkcionalitások. A `glewInit()` metódus meghívása során jönnek létre a függvénypointerek. E nélkül bármelyik OpenGL függvény hívása szegmentációs hibához vezet.

4.2. Magasságtérkép generálása

4.2.1. Heightmap osztály

A Heightmap osztály egy magasságtérképet reprezentál. Az egyszerű implementáció kedvéért kizárólag olyan magasságtérképek létrehozása lehetséges ezzel az osztállyal, amely négyzet alakú. Az osztály nem csak a magasságtérképhez tartozó pontokat tartalmazza, hanem a pontok indexeit illetve normálvektorait is.

4.2.1.1. Csúcspontok létrehozása

Algoritmizálás szempontjából talán ez a legegyszerűbb része a programnak. A `createVertices` metódus a `Heightmap` osztály `size` és `rate` változóit felhasználva generálja le a sík magasságtérképet. Az osztály tervezése során fontolóra vettem, hogy célszerű lenne-e a generált pontokat egy kétdimenziós tömbben tárolni. Ezt végül elvetettem, mivel a `glBufferData` OpenGL metódus a grafikus kártyán létező bufferre tölti fel az adatot és azokat egydimenziós tömbben várja.

A csúcspontok a magasság adatot nem tartalmazzák, ezeket külön töltöm fel a grafikus kártyára, és a shader program segítségével egyesítem őket. Emiatt itt az y koordinátákat nem is generálom le. Továbbá a méret alapján eltolom az összes generált pontot negatív irányba, olyan mértékben, hogy az alapértelmezett nézőpont pontosan a magasságtérkép közepén legyen. Ezek után már csak egy egyszerű kétszeresen beágyazott for ciklusban generálom le a csúcspontokat. A `rate` változó a két csúcspont közötti távolságot határozza meg.

4.2.1.2. Indexek létrehozása

Bármilyen objektum kirajzolása során szembesülünk azzal a problémával, hogy ugyanazzal a koordinátával rendelkező csúcspontokat kell feltölteni a grafikus kártyára. Ennek oka az, hogy OpenGL-ben minden poligont háromszögekből kell (célszerű) felépíteni. Példaként vegyünk egy négyzetet. Ennek kirajzolásához két darab háromszöget kell definiálni, amely összesen hat darab pontból áll, viszont könnyen beláthatjuk, hogy hat pontból kettő koordinátái megegyeznek. Ez nem tűnik soknak, viszont komplex objektumoknál rengeteg ilyen pont létezik, illetve egy pont tulajdonságai (attribútumai) nem feltétlen csak a pont koordinátáit jelentik, hanem tartalmazhatnak akár szín értéket, textúra koordinátát vagy bármilyen egyéb információt. Ennek megoldására létezik az indexelt kirajzolás. Ennek keretében a duplikált csúcspontokat ki lehet hagyni. Az indexeket megfelelő kell feltölteni a grafikus kártyára, illetve a `glDrawArrays` parancs helyett a `glDrawElements` utasítást kell használni, a többi teendőt a grafikus kártya elvégzi.

GL_PATCHES

A 4.0 fölötti OpenGL verziókban lehetőség van programozó által definiált primitívet létrehozni. Eredetileg a pont, vonal és háromszög primitív típusok közül lehetett választani. Ez azt jelenti, hogy például ha egy háromszöget akarunk kirajzolni, azt a grafikus kártyának átadott három pont együttese határozza meg egy, és amennyiben nem hárommal osztható számú pontot adunk meg, akkor a maradék pontok eldobásra kerülnek. A `GL_PATCHES` primitív típus ezzel szemben megengedi, hogy kirajzolás előtt definiáljuk, hogy hány átadott pontonként legyen „kész” az aktuális primitív.

A programom esetében ez a szám 16, tehát minden 16.-ik pont jelent egy új primitívet, amely az indexek szempontjából lényeges, mivel olyan sorrendben kell meghatározni azokat, hogy egy 4x4 es méretű hálót reprezentáljanak. Ez a dinamikus részletesség megvalósításához szükséges, melyet egy későbbi fejezetben részletezek.

A konkrét index értékek létrehozásánál az egyetlen változó, amelyet használni kellett, az a magasságtérkép mérete. A csúcspontok oszlop sorrendben lettek létrehozva, tehát például a nulladik csúcsponttól egyel előrébb lévő csúcspont az első csúcspont, és az eggyel mellette lévő csúcspont sorszáma megegyezik az egy sorban lévő csúcspontok számával, a kettővel mellette lévő pont sorszáma megegyezik az egy sorban lévő csúcspontok számának

kétszeresével és így tovább. Jó tulajdonsága a `GL_PATCHES`-nek, hogy az egy patchen belüli pontokat kell csak meghatározni, a tesselation shader szakasz közben visszafejtődnek háromszögekké, tehát elegendő az egy patchen belüli pontokat oszlop sorrendben indexekkel ellátni.

4.2.1.3. Magasság hozzáadása és normálvektorok számítása

Mint már említve volt, a magasságtérkép csúcspontjai nem tartalmazzák a magasságot. Ezeket, mint textúra töltöm fel a grafikus kártyára, és a shader program segítségével egyesítem a kettőt. Ennek megvalósításához először a kiválasztott algoritmussal kell egy textúrát generálni. Ez a textúra minden esetben egy szürkeárnyalatos 8-bites négyzetes textúra lesz. A textúrán egy adott pixel árnyalata határozza meg a domborzat adott pontján lévő magasságot, minél világosabb az árnyalat, annál magasabb az adott pont. A használandó algoritmust a felhasználói felületről lehet kiválasztani.

Perlin zaj

Magát a Perlin zaj algoritmusát nem implementáltam, mivel a GLM függvénykönyvtár része, csak felhasználtam. A felhasználás módja tulajdonképp megegyezik az előzőekben tárgyalt módszerrel, amiben különbözik, hogy a kezdőfrekvencia méretén kívül minden változó értékét a felhasználói felületről lehet állítani. A kezdőfrekvencia értéke a magasságtérkép méretének a reciproka. Az adott pont iterálása nem folytatódik, amint a frekvencia értéke nagyobb lesz, mint a magasságtérkép mérete, hiszen az attól nagyobb frekvenciák már olyan kis részleteket produkálnának, amelyek nem látszódnának.

Diamond-square (20)

Az algoritmushoz egy Java nyelven implementált megoldást találtam a fenti linken, melyet minimális módosítással átírtam C++ nyelvre. Az algoritmus eredetileg 2 dimenziós tömbbe generálja le a magasságadatot, így ezt egy egyszerű konverzióval egy dimenziós tömbbé kell alakítani. Továbbá, az algoritmus $n^2 + 1$ méretű adathalmazt tud csak generálni, ezért az első sornak és oszlopnak az adatait a konverzió során elhagyom.

Fault algoritmus

Ezt az algoritmust még a Java nyelven íródott programomhoz implementáltam, minimális változtatásokat kellett csak alkalmazni, hogy C++ nyelven is működjön. Ahogy már

említve volt, az iterációnkénti magasságváltoztatás mértékét iterációnként célszerű módosítani, de ennél az algoritmusnál a célom kizárólag az egyszerű implementálás volt priorizálva, így az értéket nem változtatom az iterálások során.

Mielőtt a generált adatokat feltölteném a grafikus kártyára, a magasságpontokhoz tartozó normálvektorokat kiszámítom. Erre a fényhatások szimulációjához van szükség. Normálvektort nem lehet egyetlen vektorhoz, hanem csak egy felülethez rendelni. Grafikai programozásban általános megoldás, hogy egy felületponthoz tartozó normálvektort a környező 4 vagy 8 pont segítségével határozzák meg. A programomban 4 ponttal számoltam. A normálvektor számolásának az első lépése, hogy megkeressük a szomszédos pontokat. Minden szomszédos ponttal vektort képzünk az eredetei pontra, majd két, síkban egymással nem párhuzamos vektor vektoriális szorzatát kiszámítjuk. Végül a kapott négy vektornak vesszük az átlagát, majd normalizáljuk azt. Ennek eredménye egy olyan vektor, amelynek komponensei $[-1; 1]$ intervallumban vehetnek fel értékeket. Mivel a normálvektorokat is textúraként kívánom feltölteni, ezt az értéket a $[0; 1]$ intervallumba kell eltolni, majd a shader programban az értékek felhasználása előtt visszatolni az eredeti intervallumba. Ellentétben a magasságtérkép textúrával, a normálvektor textúra három színkomponensből áll össze, ahol a vörös, zöld és kék komponensek a normálvektor x, y és z komponensei ebben a sorrendben.

Mielőtt a textúrákat feltölteném a grafikus kártyára, a kész textúrákat kép formátumban a lemezre írom. Ez főleg hibakeresési funkciókat látott el az implementálási szakaszban, illetve itt próbálkoztam a C++ nyelv adta szálkezelési lehetőségekkel, amelyek a C++11-ben a szabványos könyvtárcsomagba behozott `std::thread` objektummal könnyűnek bizonyultak.

Az utolsó lépés a textúrák feltöltése előtt a domborzat minimum és maximum pontjainak meghatározása, amely szükséges a megfelelő megjelenítéshez. Ahhoz hogy a grafikus kártya által értelmezhető formátumba hozzuk a textúrát néhány paraméter megadása kötelező.

```
glTexImage2D(target,level,internalformat,width,height,border,format
,type,pixels);
```

Ahol a függvény paraméterei a következők:

- `target`: A feltöltendő textúra típusa, a programomban kizárólag a `GL_TEXTURE_2D` paraméterrel használtam, amely szimpla kétdimenziós textúrát jelent.
- `level`: Részletességi szint, esetünkben mindig nulla.
- `internalformat`: A textúrához használt színkomponensek száma. A magasságtérkép textúra esetében csak a vörös komponenst használom `(GL_RED)`, de értelem szerűen azt sem szín reprezentálására, hanem a magasság érték tárolására. A normálvektor textúra esetén egyszerű, tömörítés nélküli háromkomponensű textúrát használtam `(GL_RGB)`.
- `width, height`: Szélesség, magasság. Mindkét érték a magasságtérkép méretével megegyezik.
- `border`: Ez az érték mindig nulla. (21)
- `format`: A pixel formátum típusa. Általában megegyezik az `internalformat`-nál megadott értékkel.
- `type`: A pixel formátum adattípusa. A programban kivétel nélkül GL_FLOAT típusú pixelekkel dolgozok.
- `data`: Egy mutató a textúra adatra.

Ezek után nincs más hátra, csak a textúrák, illetve az egyéb kiszámolt adatok a grafikus kártyára feltölteni.

4.3. Magasságtérkép kirajzolása

Miután beállítottuk az OpenGL kontextust, valamint a shader program is sikeresen lefordult, illetve minden adat a grafikus kártyára lett töltve, elkezdhetjük a tényleges kirajzoláshoz való előkészületeket. A folyamatos kirajzolás érdekében egy ciklust kell létrehozni, amely egészen addig fut, amíg a kirajzoláshoz használt ablak be nem záródik. Az ilyen ciklusok általában a következőképp néznek ki:

```
while(ablak.nyitva){
bufferekTörlése();
frissítés(); //pl.:kirajzolandó objektumok, nézőpont pozicíció stb.
kirajzolás();
bufferekCseréje();
}
```

A bufferek törlése nem feltétlenül egy buffer törlését jelenti, például, mint a programom esetében is, ha háromdimenziós objektumokat rajzolunk ki, a szín buffer mellett célszerű a mélyég buffert is kitörölni, ezzel elkerülhető a térbeli objektumok összemosódása.

A program esetében a két kirajzolás között történő frissítés nagyrészt a nézőpont pozíciójának beállítása, valamint a felhasználói felületen végzett módosítások alkalmazása a domborzatra. Ide tartozik még a billentyűlenyomások feldolgozása is.

Duplabufferes kirajzolás esetén a bufferek cseréjére is szükség van. Ennek a működési elve, hogy amíg az egyik buffert a felhasználó éppen látja, addig a másik bufferre megtörténik a kirajzolás. Ezzel a módszerrel nagymértékben csökkenthető a kirajzolásból eredő hibák száma.

4.3.1. Dinamikus részletességszint (22) (23)

Ez idáig szándékosan nem beszéltem a kirajzolás menetéről, mivel ez a dinamikus részletességszint megvalósításával szoros kapcsolatban áll. Az algoritmus megvalósítása szinte teljes mértékben a shader programban történik meg, az ehhez szükséges előkészületekről már volt szó az előző fejezetekben. A következőkben a shader szakaszok lefutási sorrendjében részletezem, hogy melyik szakaszban éppen mi történik.

Vertex shader szakasz

Ezen a szakaszon mindösszesen egy bejövő attribútum típusú változó van, ezek a magasságtérkép pontjai, amelyek még nem tartalmazzák a magasságot, tulajdonképp egy négyzetekből álló rács. További uniform típusú bemenetek a magasságtérkép textúra, a normálvektor textúra, nézőpont létrehozásához szükséges mátrixok illetve, a nézőpont pozíciója. Normál esetben itt történne meg a csúcspontok véglegesítése, de az algoritmus miatt itt ez elmarad, csak előkészítem az adatokat a tesselation szakaszhoz. Először is a magasságtérkép pontokat egyesítem a magasság textúrával, de ez még nem a konkrét kirajzoláshoz való előkészület, hanem a domborzattal módosított távolság pontos méréséhez szükséges. Továbbá egyéb számításokat is végzek, amely nem tartozik kifejezetten a részletesség szint algoritmushoz, ilyen például az adott pont környezetének a meredekségének a kiszámolása, valamint az adott magasságtérkép ponthoz tartozó normálvektor kinyerése a normálvektor textúrából.

Tesselation Control shader szakasz

Ez a szakasz az első konkrét lépés a dinamikus részletesség szint megvalósítására, mivel itt lehetőség van a külső és belső tesselation szint beállítására. Ez a szint határozza meg, hogy az adott poligont milyen mértékben osztja fel kisebb darabokra. A vertex shaderből érkező csúcspontok 16 elemet tartalmazó patchekbe rendeződnek, ez az egyik fontos lépés az algoritmus menetében. Így, a vertex shaderrel ellentétben, ahol egy lefutás alatt csak egy pontot látunk, itt egy lefutás alatt 16 ponttal tudunk dolgozni, amely lehetőséget ad különböző interpolációs műveletek elvégzésére. Ezek az interpolációk csak a következő szakaszban történnek meg. Itt csak a feldarabolás mértékékének meghatározására van lehetőség. Ezt az adott pontnak a nézőponttól való távolsága határozza meg.

```
float d0=distance(eyePos,tcWorldPos[0]);
float d1=distance(eyePos,tcWorldPos[3]);
float d2=distance(eyePos,tcWorldPos[12]);
float d3=distance(eyePos,tcWorldPos[15]);


gl_TessLevelOuter[0]=level(mix(d0,d1,0.5));
gl_TessLevelOuter[1]=level(mix(d1,d2,0.5));
gl_TessLevelOuter[2]=level(mix(d2,d3,0.5));
gl_TessLevelOuter[3]=level(mix(d3,d0,0.5));
```

Értelem szerűen a `distance` metódus a két megadott vektor távolságát adja vissza, ahol `tcWorldPos[n]` az aktuális pont koordinátája, a kódban megadott tömb indexek pedig az aktuális patch sarokpontjai. A négy beállított `gl_TessLevelOuter[n]` érték a négyzet négy oldalának a felosztási szintje. A `level` metódus a darabolás értéket egy megadott érték fölött levágja, ennek a maximális értéke grafikus kártyától függ, de a tapasztalatom szerint 32-nél magasabb értéket, ebben az esetben legalábbis, felesleges beállítani. Az egyéb beérkező attribútum változókat változtatás nélkül másolom a következő shader szakaszra.

Tesselation evaluation shader

Ezen a szakaszon történik meg az interpoláció elvégzése, illetve a pontok koordinátájának kiírása. Az interpolációhoz Bézier mátrix transzformációt használok, amelyhez az interpolálandó adatokat 4x4-es méretű mátrixokba kell rendezni. A transzformáció koordináta komponensenként működik, tehát csúcspont koordinátánként 3 darab mátrix jön létre. A mátrix létrehozása közben ismét megtörténik a magasság textúra egyesítése a

magasságtérkép pontokkal, itt már az interpoláció, illetve a végleges koordináták kiírása céljából.

```
 vec4 U=vec4(u*u*u,u*u*u,u,1);
 vec4 V=vec4(v*v*v,v*v*v,v,1);

mat4 B=mat4(
         -1,  3, -3, 1,
          3, -6,  3, 0,
         -3,  3,  0, 0,
          1,  0,  0, 0);
 mat4 BT=transpose(B);

   mat4 Px=mat4(
         x[0],x[1],x[2],x[3],
         x[4],x[5],x[6],x[7],
         x[8],x[9],x[10],x[11],
         x[12],x[13],x[14],x[15]);

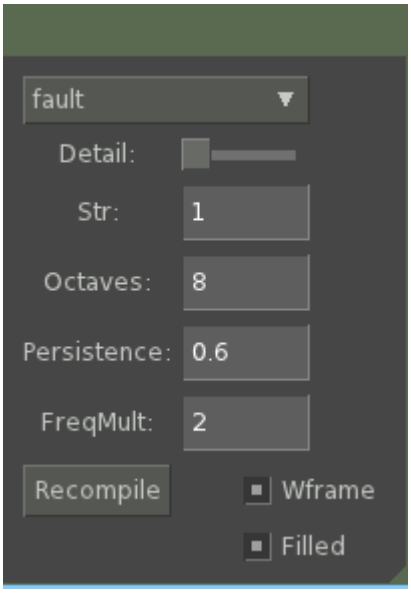
   mat4 cx=B * Px * BT;
   float x=dot(cx * V,U);
```

A fenti kódrészletben az aktuális patch x komponenseit sorrendben egy mátrixba rendezem, majd a megadott módszerrel kiszámítom a `cx` együttható mátrixot. A `B` mátrix a Bézier transzformációs mátrix, melynek a levezetése elég hosszadalmas, ezért nem foglalkozok vele, hasonlóan az `U` és `V` vektorokkal sem. Az `u` és `v` vektorok baricentrikus koordináták, amelyek a shader ezen szakaszán beépített változók. Négyzetek esetén ezek 2 komponensű vektorok, amelyeknek értékei 0 és 1 közötti számok, amelyek egy csúcspontot határoznak meg az adott négyzeten belül. Ezeket a műveleteket hasonló módon kell elvégezni a csúcspont vektorok többi komponensére is, illetve a normálvektorok komponenseire is. Ezek után lehet kiírni az immár végleges koordinátákat.

Fragment shader

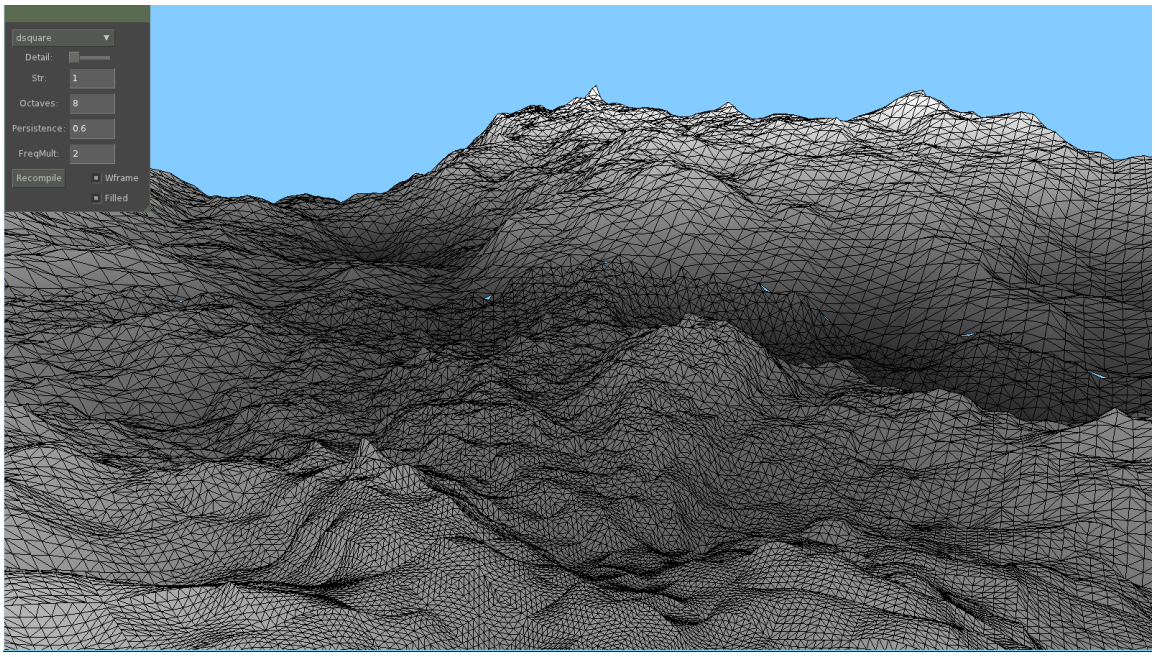
Ezen a szakaszon már csak a pixelek színeinek meghatározása történik. Amennyiben a kirajzolás drótváz módban van, minden kirajzolt pixel színe fekete. Ha telített módban rajzol a program, akkor, a magasságtérkép textúrából, az adott pixel színét kapja az adott pont a kirajzolás során, tehát egy pont minél magasabban van, annál világosabb színű lesz. Továbbá egy egyszerűbb fény szimuláció implementálásával is próbálkoztam.

4.4. Felhasználói felület



14. ábra: A felhasználói felület

A felhasználói felület célja egyrész a domborzatgenerálás paramétereinek beállítása, másrészt a generált domborzat megjelenítése. A legördülő lista tartalmazza az implementált algoritmusokat, a kiválasztása esetén egyből generálódik a domborzat. A detail csúszka a dinamikus részletességnek a távolság paraméterét növeli. Az Str paraméter a generált domborzat magasságának a szorzószáma, az Octaves az iterációk száma, amely csak a Perlin zaj, illetve a Fault algoritmus esetén van figyelembe véve. A Persistence és a FreqMult paraméterek csak a Perlin zaj esetén vannak figyelembe véve, ahol a Persistence az iterációnkénti amplitúdócsökkenés szorzószáma, a FreqMult pedig az iterációnkénti frekvencia szorzószáma. A Wframe és Filled rádiógombok a drótvázas illetve a kitöltött rajzolás mód között váltanak. A Recompile gomb a shader programot újrafordítja, illetve a domborzatot is újragenerálja.



13. ábra A program működés közben

Továbbá a nézőpont mozgatása a W, A, S, D billentyűkkel, a nézőpont forgatása jobb egérgomb lenyomva tartásával és az egér mozgatásával történik. A T és Z billentyűk lenyomásának hatására ugyanazt érjük el, mint a detail csúszka mozgatásával, a G és H

billentyűk pedig a dinamikus részletesség szint részletesség paraméterét növelik. Az escape billentyű megnyomásával lépthetünk ki a programból.

5. Észrevételek

A fejlesztés során a programomat többféle hardveren is futtattam, melyek közül az egyik NVIDIA, a másik ATI grafikus kártyával rendelkezett. A tapasztalatom az volt, hogy az ATI kártyákon a tesselation control shader eléggé kiszámíthatatlanul működik. A hiba akkor jön elő, ha az említett shader szakaszon egynél több attribútum típusú változót próbálok módosítás nélkül továbbítani a következő shader szakaszra. A program normál módon való futtatáskor az esetek nagy részében jól futott a program, néha viszont egész egyszerűen semmit nem rajzolt ki, a kirajzolás utasítás pedig INVALID_OPERATION hibát dobott. Debug módban való futtatáskor viszont bármelyik próbálkozásra az első kirajzolás előtt szegmentációs hibával állt le a program. Ezt a jelenséget NVIDIA kártyákon nem tapasztaltam, megjegyezném viszont, hogy az ATI kártya egy generációval régebbi volt, mint az NVIDIA.

6. Fejlesztési lehetőségek

Mivel a programot eredetileg kizárólag azzal a céllal kezdtem el írni, hogy megtanuljam a C++ illetve a modern OpenGL alapjait, ezért ezt a programot nem fogom továbbfejleszteni, viszont a tervezés és implementáció során úgy érzem, rengeteg tapasztalatot szereztem, melyet a későbbiekben biztos, hogy fel fogok használni egy hasonló, sokkal komplexebb program megírására.

A tervezett program elsősorban nem magasságtérkép, hanem voxel alapú, amivel sokkal komplexebb domborzatokat lehet létrehozni, vagy akár természetes hídképződményeket, illetve barlangokat, amelyeket a magasságtérképes megoldással nem lehetett, mivel a magasságtérkép tulajdonképp nem más, mint egy síkfelület, amelynek a pontjait egy algoritmus szerint eltoljuk az y tengelyen. Voxeleket használva definiálunk egy háromdimenziós rácsszerkezetet, amelyben a rácsok a voxelek. Ebben az esetben, ha egy domborzatot akarunk generálni, akkor az algoritmus, például Perlin zaj, eredményét nem mint magasságértéként fogjuk fel, hanem mint sűrűség érték. Bizonyos sűrűség érték felett az adott voxel domborzat lesz, egyébként pedig egy átlátszó pont. További tervezett funkció a programban, hogy nem egy fix méretű domborzatot generál, hanem mindig ott generál valamennyi domborzatot, ahol épp a megfigyelő tartózkodik, és az egyszer már legenerált domborzatot a lemezre menti, és onnantól kezdve nem generál, hanem a lemezről olvassa a domborzatot hasonló módszerrel, mint a Minecraft nevű sandbox játékban. Ez a módszer ad lehetőséget arra, hogy tulajdonképp végtelen nagyságú területet generáljunk, amelyet ráadásul módosítani is lehet. Bár a végtelen méretű domborzat jól hangzik, az én tervem mégis inkább olyan algoritmus megvalósítása, amely egy gömb alakzatra teszi rá a voxel alapú domborzatot, tehát tulajdonképp egy bolygót, és annak domborzatát szeretném létrehozni. A domborzatgeneráláshoz szorosan hozzátartozik az erózió szimulációja is, illetve a víz jelenléte is, melyek szintén tervezett funkciók a programban.

7. Összefoglalás

Szakdolgozatomban néhány procedurális domborzatgeneráló algoritmus működését ismertetem, melyek közül a Fault algoritmust, a Diamond-square algoritmust illetve a Perlin zajt egy program fejlesztésével szemléltetem. Azért esett ezekre az algoritmusokra a választásom, mivel ezeket viszonylag egyszerű implementálni, viszont szép eredményeket produkálnak. Ezen kívül egy kezdetleges dinamikus részletezettség szintet megvalósító algoritmust is implementálok a programba, amely kihasználja a modern OpenGL adta lehetőségeket.

A programot eredetileg Java nyelven kezdtem írni, de időközben rá kellett jönnöm, hogy az ilyesfajta programozásra a C++ nyelv sokkal kézenfekvőbb, mert gyorsabb, ugyanakkor bonyolultabb, viszont ez több lehetőséget nyújt. Az új nyelv elsajátítása csak a kezdetben volt nehéz, például a C++ pointerek működése, illetve a függvényhívások módjának megértése.

Az algoritmusok működésének leírását az interneten fellelhető tudományos, programozással és számítógépes domborzatgenerálással kapcsolatos weboldalakon találtam meg. Az algoritmusokon kívül a kutatás során rengeteg hasznos információt találtam, amely nagymértékben hozzájárult, hogy az OpenGL-lel kapcsolatos tudásomat fejlesszem.

Az implementáció során rengeteg tapasztalatot szereztem, melyet - amennyiben lesz rá lehetőségem -, szeretnék még tovább gyarapítani. Ezt a programot már nem szeretném tovább fejleszteni, helyette egy teljesen új programot tervezek, amely többek között nem magasságtérképekkel, hanem voxel alapon működik, illetve nem fix méretű, hanem dinamikusan generált domborzattal rendelkezik.

8. Summary

In my thesis I present a few procedural terrain generation algorithms, from which I implement the Fault algorithm, the Diamond-square algorithm and the Perlin noise in a program. I chose these algorithms because they are relatively easy to implement but they procude nice results. Besides I also implement a simple dynamic level of detail algorithm which takes advantage of the capabilities of the modern OpenGL.

I started to write the original program in Java language but in the meantime I had to realize that the C++ language is far more superior for these kinds of purposes, because it is faster however much more complicated but this offers many more opportunities. Learning this language was hard only in the beginning, for example the arithmetics of C++ pointers and understanding the method of function calls.

The documentation of the operation of algorithms can be found on scientific web pages and on sites that are related to programming and terrain generation on the internet. Besides the algorithms I found lots of useful information while I was doing research, which greatly contributed to my improvement of my knowledge related to OpenGL.

During the implementation I earned plenty of experience, which I would like to further improve if I had the opportunity. I would not like to further develop this program instead I plan making a completely new program, which among others will not use heightmaps instead it will be based on voxels, also its terrain will not be static but dynamically generated.

9. Irodalomjegyzék

1. History of procedural generation. (Ellenőrizve: 2014.11.19)
http://www.giantbomb.com/procedural-generation/3015-328/.

2. Procedural generation. (Ellenőrizve: 2014.11.19)
http://en.wikipedia.org/wiki/Procedural_generation.

3. Fault algoritmus. (Ellenőrizve: 2014.11.19)
http://www.lighthouse3d.com/opengl/terrain/index.php3?fault.

4. Fault algoritmus implementáció. (Ellenőrizve: 2014.11.19)
http://www.lighthouse3d.com/opengl/terrain/index.php?impdetails.

5. Körök algoritmusa. (Ellenőrizve: 2014.11.19)
http://www.lighthouse3d.com/opengl/terrain/index.php?circles.

6. Fault algoritmus gömbre alkalmazva. (Ellenőrizve: 2014.11.19)
http://freespace.virgin.net/hugo.elias/models/m_landsp.htm.

7. Mid Point Displacement Algorithm. (Ellenőrizve: 2014.11.19)
http://www.lighthouse3d.com/opengl/terrain/index.php?mpd2.

8. Explanation of Midpoint Displacement. (Ellenőrizve: 2014.11.19)
https://code.google.com/p/fractalterraingeneration/wiki/Midpoint_Displacement.

9. Explanation of Value Noise. (Ellenőrizve: 2014.11.19)
https://code.google.com/p/fractalterraingeneration/wiki/Value_Noise.

10. Daniel Shiffman. The nature of code. (Ellenőrizve: 2014.11.19)
http://natureofcode.com/book/introduction/.

11. Ken Perlin. (Ellenőrizve: 2014.11.19) http://www.noisemachine.com/talk1/index.html.

12. Hugo Elias. Perlin noise. (Ellenőrizve: 2014.11.19)
http://freespace.virgin.net/hugo.elias/models/m_perlin.htm.

13. Explanation of Simplex Noise. (Ellenőrizve: 2014.11.19) https://code.google.com/p/fractalterraingeneration/wiki/Simplex_Noise.

14. C++. (Ellenőrizve: 2014.11.19) http://en.wikiversity.org/wiki/C++.

15. Java comparison with C++. (Ellenőrizve: 2014.11.19) http://en.wikibooks.org/wiki/C%2B%2B_Programming/Programming_Languages/Comparisons/Java.

16. OpenGL specification. (Ellenőrizve: 2014.11.19) https://www.opengl.org/registry/doc/glspec45.core.pdf.

17. GLSL language specification. (Ellenőrizve: 2014.11.19) https://www.opengl.org/registry/doc/GLSLangSpec.4.50.pdf.

18. OpenGL Mathematics. (Ellenőrizve: 2014.11.19) http://glm.g-truc.net/0.9.5/index.html.

19. Alexander Overvoord. Window and OpenGL context. (Ellenőrizve: 2014.11.19) https://open.gl/context.

20. Diamond square algorithm. (Ellenőrizve: 2014.11.19) http://stackoverflow.com/questions/2755750/diamond-square-algorithm.

21.OpenGL Texture (Ellenőrizve: 2014.11.19) https://www.opengl.org/sdk/docs/man/html/glTexImage2D.xhtml.

22. Tessellated Terrain. (Ellenőrizve: 2014.11.19) http://micah-d-lamb.com/blog/opengl-tutorials/tessellated-terrain/.

23. Philip Rideout. Quad Tessellation with OpenGL 4.0. (Ellenőrizve: 2014.11.19) http://prideout.net/blog/?p=49.

24. GLSL Core Language. (Ellenőrizve: 2014.11.19) https://www.opengl.org/wiki/GLSL_Core_Language.

10. CD Melléklet tartalma

- Dolgozat
  - Kiiras.docx
  - Osszefoglalas.docx
  - Osszefoglalas.pdf
  - Summary.docx
  - Summary.pdf
  - Szakdolgozat.docx
  - Szakdolgozat.pdf
- Alkalmazas
  - terr_gen.exe
  - olvassEl.txt
  - vertex.vs
  - tescontrol.tc
  - teseval.te
  - fragment.fs
  - glew32.dll
  - libgcc_s_dw2-1.dll
  - libstdc++-6.dll
  - libwinpthread-1.dll
  - sfgui.dll
  - sfml-graphics-2.dll
  - sfml-system-2.dll
  - sfml-window-2.dll
- Forras
  - Header
    - Ez a könyvtár tartalmazza a C++ header fileokat.
  - Shader
    - Ez a könyvtár tartalmazza a GLSL program fileokat
  - Source
    - Ez a könyvtár tartalmazza a C++ forráskód fileokat.

EREDETISÉGI NYILATKOZAT

Alulírott ……..; Neptun-kód:……
a Miskolci Egyetem Gépészmérnöki és Informatikai Karának végzős ……. szakos hallgatója ezennel büntetőjogi és fegyelmi felelősségem tudatában nyilatkozom és aláírásommal igazolom, hogy

……

című szakdolgozatom/diplomatervem saját, önálló munkám; az abban hivatkozott szakirodalom
felhasználása a forráskezelés szabályai szerint történt.
Tudomásul veszem, hogy szakdolgozat esetén plágiumnak számít:

- szószerinti idézet közlése idézőjel és hivatkozás megjelölése nélkül;
- tartalmi idézet hivatkozás megjelölése nélkül;
- más publikált gondolatainak saját gondolatként való feltüntetése.

Alulírott kijelentem, hogy a plágium fogalmát megismertem, és tudomásul veszem, hogy plágium esetén szakdolgozatom visszautasításra kerül.

Miskolc,.....év ……..hó ……..nap

……..……..…

Hallgató