Debreceni Egyetem

Informatika Kar

Terrain modellezés GLSL segítségével

Témavezető:

Dr. Tornai Róbert

Egyetemi adjunktus

Készítette:

Girhiny Róbert

Programtervező Matematikus

Debrecen

2010

Tartalomjegyzék

I BEVEZETÉS

Bevezetés

A mai világban egyre nagyobb szerepet kap a grafika. Ha csak az operációs rendszerek fejlődését kísérjük figyelemmel, az elmúlt 20-25 évben már nagy változásokat tapasztalhatunk nem csak funkcionalitási szinten, hanem grafikai szinten is. Míg régen (1982) a DOS operációs rendszer szöveges üzemmóddal rendelkezett, addigra ma már egy Tablet PC érintő kijelzőkkel rendelkezik és grafikus ikonokra kattintgatunk az ujjainkkal ahelyett, hogy parancssorban gépelnénk be az utasításokat. Nem beszélve arról, hogy a programok eladhatóságát is jócskán megnöveli, ha egy program külsője megnyerő. Gyakran előfordul, hogy a vásárló inkább megveszi a dizájnosabb, szebb külsejű programot, még ha az funkcionalitásban kevesebbet is tud, mint egy grafikailag visszamaradottabb programtársa, ami esetleg „okosabb” nála. A grafikai fejlődés a játékfejlesztésben és a filmiparban is megmutatkozik. Éppen ezért ez egy olyan területe az informatikának, a programozásnak amit nem szabad elhanyagolni, nem mondhatjuk azt hogy majd a grafikus megrajzolja a dizájnt és kész, valakinek azt le is kell programoznia méghozzá úgy, hogy az adott hardveren a lehető legjobb képi hatást elérve ne terheljük le a gépeket. Ezt általában trükkökkel érhetjük el, a felhasználót „becsapjuk” a képi hatással. Ez nem kis kreativitást, probléma megoldó készséget igényel a programozótól. Teljes mértékben tisztában kell lennie az igényekkel, hogy a felhasználó mit szeretne látni és azzal is tisztában kell lennie, hogy a megjelenítés során hol, mivel és mennyire „csalhat”, hogy a kép még élethű maradjon és kielégítse a vele szemben felállított igényeket.

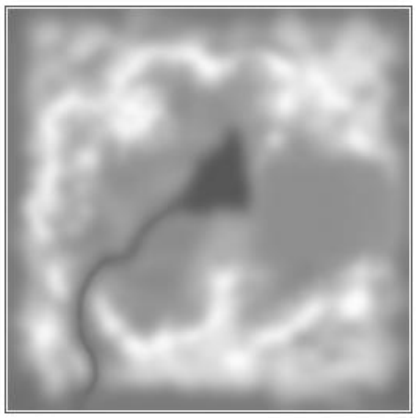
A mai világban nagy mértékben elterjedt shader technológia jó eszköztára a grafikai trükközéseknek. Segítségükkel az eddigi merev automata folyamatokat programozhatóvá tehetjük, ezáltal még látványosabb effekteket produkálhatunk.

A dolgozat célja, hogy az olvasót bevezesse ebbe a grafikai világba és konkrét példákon végigvezetve bemutassa a shader technológiát és egy pár grafikai trükköt. Mindezt Delphi programozási nyelven az OpenGL grafikai könyvtárat felhasználva. Ritka és fura párosítás lehet

az olvasó számára a Delphi-OpenGL páros, hiszen a mai világot egy „micro” szoftvervállalat uralja és irányítja. A cégek egytől-egyig az ő programnyelvüket preferálják, így a programozóktól is az az elvárás, hogy ahhoz értsenek, így a Delphi és egyéb programozási nyelvek háttérbe szorulnak, pedig nem feltétlenül rosszabbak. A dolgozat rejtett célja az is, hogy esetleg ezt is megmutassa.

A dolgozatban szereplő képek vagy saját készítésűek, ez esetben nem szerepel a forrás helye, minden más esetben az adott kép alatt olvasható a forrás.

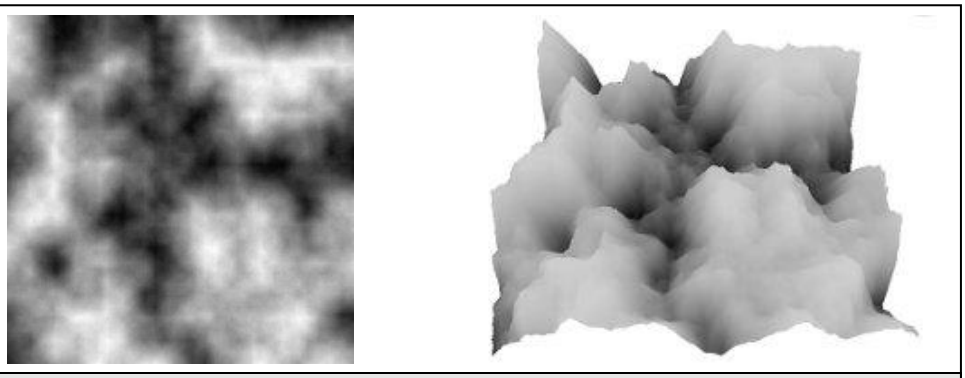
II HEIGHTMAP



Heightmap

A számítógépes grafikában a magasságtérkép (heightmap vagy heightfield) egy 2D-s raszter kép. Olyan implicit felületek, ahol az f(x,y,z)=0 implicit egyenlet a z=h(x,y) alakra hozható. A magasságtérképeket gyakran alkalmazzák terepmodellezésre, hiszen elképzelhetjük úgy is, hogy a tengerszinthez képest megadjuk a terep magasságát. Mivel az értelmezési tartomány egy téglalap, ezért a magasságmezőket kétdimenziós tömbökben szokás tárolni. Így adott egy rács, és benne a magasságérték. A rácspontok között interpolálunk. A magasságtérkép általában egy szürkeárnyalatos kép, (II/1. ábra bal oldalt) amely csak az egyik csatornájában hordoz számunkra értékes információt, illetve a kép mindegyik csatornájából ugyanaz az érték olvasható ki, hiszen szürkeárnyalatos. Ritka az, hogy több csatornán is tárolnánk az értékeket, hiszen ez helypazarlás lenne. Ugyanakkor magában hordozza azt a lehetőséget, hogy a többi csatornán is tároljunk további hasznos információkat a magasságtérképről. Ilyen lehet például a normálvektorok tárolása: ha a raszterképünk négy csatornás, tehát tartalmaz vörös, zöld, kék és alfa csatornát (röviden: RGBA), akkor lehetőségünk van a fennmaradó 3 csatornában letárolni az adott ponthoz tartozó normálvektor koordinátáit. Így nem szükséges azt minden alkalommal kiszámolnunk a magasságtérképhez.

Egy 32 bites képet tekintve a négy csatornára 8-8 bit jut, azaz minden egyes csatornának az értékkészlete 0-255 közötti érték. Minél nagyobb ez az érték, annál világosabb lesz az adott képpont. 0 érték esetén fekete, 255 esetén fehér. Ez a terepünk esetén azt jelenti hogy az adott ponton felvett érték minél világosabb a terep annál magasabb lesz és fordítva, minél sötétebb az adott pont a terep annál alacsonyabb lesz, de ezt a II/3-as rész bővebben taglalja, hogy hogyan is készítsünk egy heightmapból egy terepet (terraint). A magasságtérképből előállított terep a II/1-es ábra jobb oldalán látható.



II/1. ábra: a, Bal oldalt egy szürkeárnyalatos magasságtérkép,
b, jobb oldalt a magasságtérképből készített terrain.

II/1 Heightmapok készítése

Számos lehetőség, program és módszer áll a rendelkezésünkre, hogy magasságtérképeket készítsünk. Az egyik kézen fekvő lehetőség, hogy egy képszerkesztő programot felhasználva rajzolunk egyet, vagy valamilyen plugin segítségével elkészítjük. Az ingyenes Gimp[1] képszerkesztő programmal például nagyon egyszerűen generálhatunk magasságtérképeket:

1. lépés: Hozzunk létre egy üres képet, mondjuk 256x256-os felbontással.
2. lépés: Generáljunk zajt a <szűrők> → <megjelenítés> → <felhők> → <plazma> menüpont alatt.
3. lépés: tegyük szürkeárnyalatossá a képet a <színek> → <telítetlenné tevés> menüpont alatt.

A folyamat és az előállított kép, valamint a magasságtérképből készített terrain a képtár 1-es képén látható. (IX/1. Ábra)

Egy másik lehetőség, hogy direkt erre a célra fejlesztett szoftvereket használunk. Ilyen szoftver például a Terragen[2] melynek az első verziója ingyenesen használható. A magasságtérkép legyártásához néhány paramétert meg kell adnunk majd a generál gombra kattintanunk. A program jóval többet nyújt mint a magasságtérképek generálása. Az elkészített

1 The GNU Image Manipulation Program hivatalos weboldala: http://www.gimp.org/
2 A Planetside Software által fejlesztett Terragen hivatalos weboldala: http://www.planetside.co.uk/

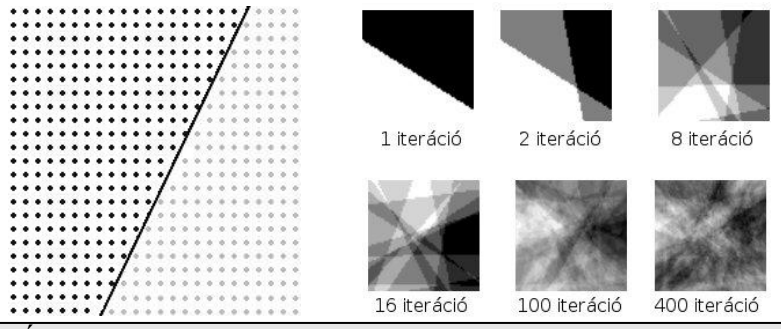
magasságtérképből realisztikus tájképet tud renderelni. A kettes verzió már shader alapú. Növényzetet, égboltot és számtalan effektet adhatunk hozzá, amivel még életszerűbb lesz a renderelt kép. Továbbá kulcsképkockák megadásával képes mozgófelvételt is renderelni.

Azonban ha nem szeretnénk külső programot használni, ha a programon belül szeretnénk generálni magasságtérképeket, akkor számos jól bevált algoritmust használhatunk. Ilyen például a Fault algoritmus, vagy a Midpoint displacement algoritmus, de egyéb algoritmusok is eredményre vezethetnek (különféle zaj generátorok, fraktálok és trigonometrikus függvények). Továbbá különböző szűrőket alkalmazhatunk a magasságtérképeken, amivel tovább finomíthatjuk azokat. A következőkben röviden átnézzük a Fault algoritmust, majd a Midpoint displacement algoritmust és a Circles algoritmust kicsit bővebben. A Körök algoritmusra adunk egy implementációt is Delphi nyelven.

A Fault algoritmus (Vetődés algoritmusa)

Ez az algoritmus nagyon egyszerű, és nem csak sík magasságmezőkre működik, hanem gömbre is, ezzel alkalmazható a bolygók modellezésére is [1].

Első lépésben a magasságmező minden pontjának magassága 0. Aztán választunk egy tetszőleges egyenest, amely kettéosztja a területet. Az egyenes egyik oldalán lévő pontok magasságát növeljük, míg a másik oldalon csökkentjük. Az iterációk számának növelésével kialakul a maga terep, a hegyek a völgyek. II/2-es ábra.

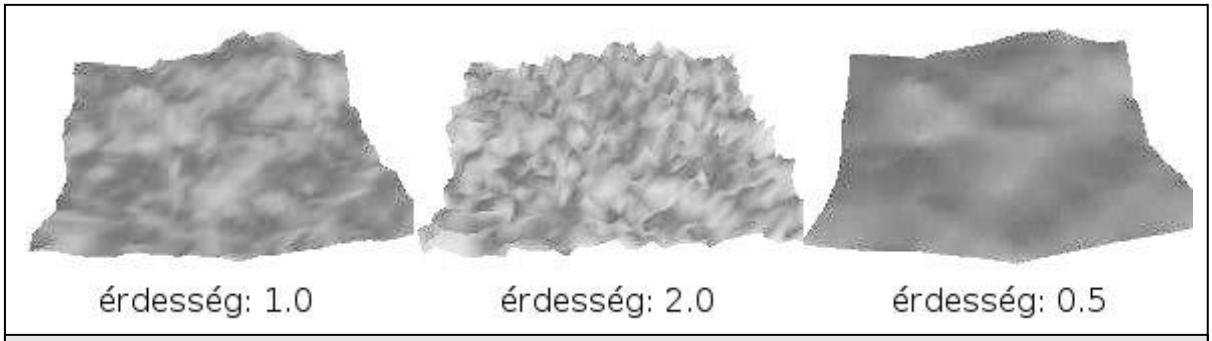


II/2. Ábra a, Bal oldalt a vetődés algoritmus első iterációs lépése.
b, Jobb oldalt a vetődés algoritmus többi iterációs lépései láthatóak.
Forrás: http://www.lighthouse3d.com/opengl/terrain/index.php?fault

A vetődés algoritmus által generált magasságtérképből előállított terület a képtár IX/2-es ábráján látható. A vetődés algoritmusnak további típusai is léteznek, ezekről a lighthouse3d.com weboldalon olvashatunk.

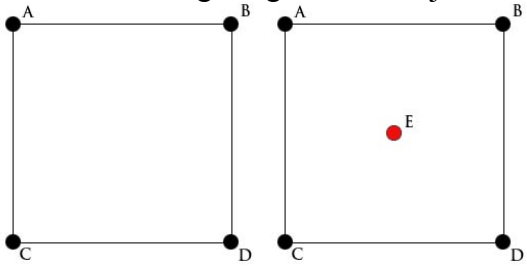
A Midpoint displacement algoritmus MDP (Középpont elmozdító algoritmus)

A középpont áthelyező algoritmus egy felosztó algoritmus. A terep iterációval épül fel, és minden lépésben a részletezettségi szintje nő. A vetődés és a körök algoritmushoz képest a számításigénye alacsony, és nagyon látványos eredményt produkál. Az algoritmus négyzet alakú terepet generál, amelynek a mérete (2n+1)x(2n+1), ahol n a lépések számát jelenti. (8 iterációval 257x257; 10 iterációval 1025x1025 méretű terepet nyerhetünk.) Az algoritmus legfontosabb paramétere az érdesség-konstans (roughness), amely leggyakoribb értéke az 1. Az érdesség paraméter hatása a II/3-as ábrán látható.



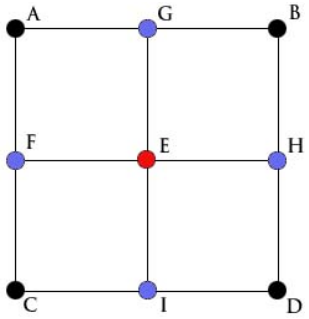
II/3. ábra: a Midpoint algoritmus eredménye különböző érdesség paramétr esetén.
Forrás: http://www.lighthouse3d.com/opengl/terrain/index.php?mpd

Az algoritmus egy 2x2-es négyzetből indul ki. A csúcspontok kezdőmagasságát beállíthatjuk 0-ra, vagy random értékre is. Első lépésben kiszámoljuk a terep középpontjának magasságát a csúcspontok számtani közepének és egy random eltolási értéknek az összegeként: **`E = (A+B+C+D) / 4 + RAND(d),`**



ahol d a maximum eltolást jelenti ebben az iterációs lépésben.

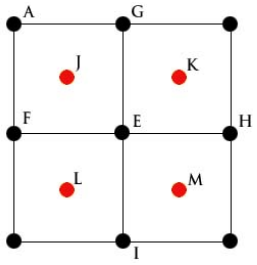
A következő lépésben a négyzet oldalfelező középpontjait számítjuk ki.




```
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)
```

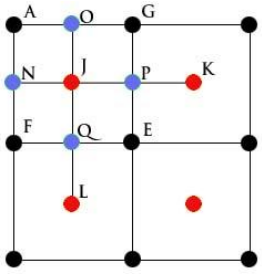
Az iteráció következő lépésénél új d értéket választunk: d'=d*2-r, ahol r az érdességi konstans.

Az iterációt most már 4 négyzeten folytathatjuk.



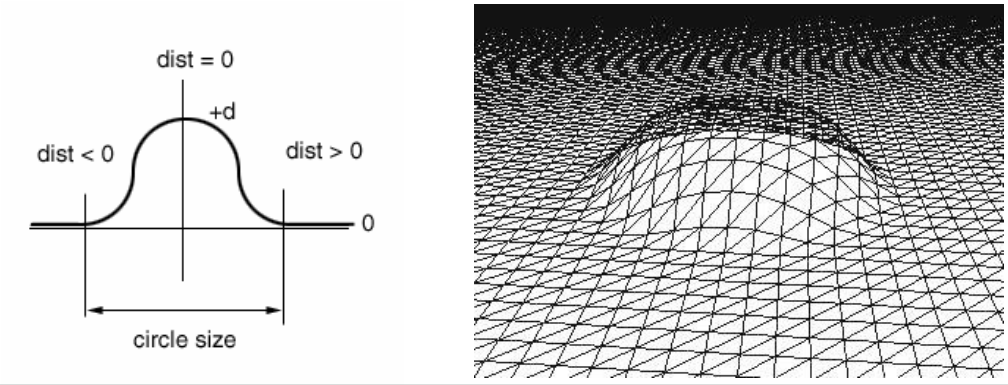

```
J = (A + G + F + E) / 4 + RAND(d')
K = (G + B + E + H) / 4 + RAND(d')
L = (F + E + C + I) / 4 + RAND(d')
M = (E + H + I + D) / 4 + RAND(d')
```

Majd a négyzetközéppontok után az oldalakon (N,O,P,Q), és így tovább...



A Circles algoritmus (Körök algoritmusa)

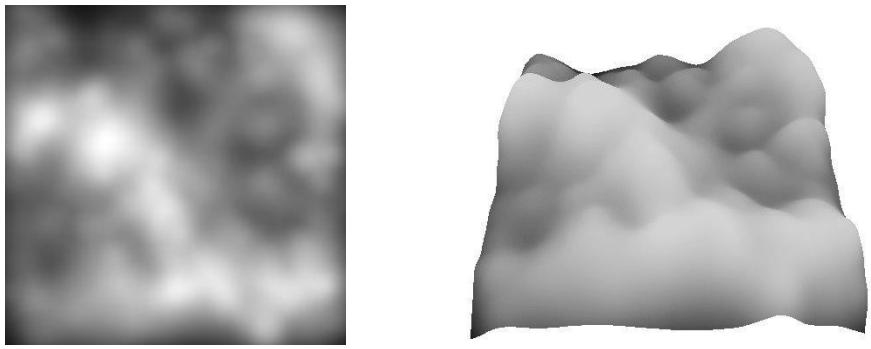
Ez az algoritmus hasonlít a vetődés algoritmusra, itt is pontok magasságát változtatjuk. Fő különbség a pontok kiválasztásában, és azok áthelyezésében van. Véletlenszerűen helyezünk el a síkon köröket, és a körön belüli pontokat mozdítjuk el felfelé a koszinusz függvénynek megfelelően. Szintén egyszerű algoritmus és nagyon szép eredményt ad.



II/4. ábra
a, Bal oldalt a koszinusz függvénynek megfelelően kiemelt bucka.
b, Jobb oldalt a körök algoritmus egy iteráció esetén egy hegyet generál.
Forrás: http://www.lighthouse3d.com/opengl/terrain/index.php?circles

Az algoritmus lényege:

0. lépés: A magasságtérkép képünket töröljük. Nulla értékkel töltjük fel.
1. lépés: Kiválasztunk egy véletlen pontot a magasságtérképen (rx,ry)
2. lépés: Választunk egy sugarat (radius)
3. lépés: Majd a magasságtérkép minden pixelén (x,y) végigmegyünk és közben mérjük a kiválasztott pontunktól való távolságot (pd)
4. lépés: A magasságtérkép minden színintenzitásához hozzáadjuk az alábbi értéket, ha pd értéke kisebb vagy egyenlő mint 1:
   heightmap[x,y] += cos(pd * PI)+1, (II/4-es ábra bal oldala alapján)
   ahol PI értéke a matematikai π értéke.
5. lépés: Minél többször ismételjük meg az 1-4 pontokat, annál több hegy fog keletkezni. Ha csak egy iterációt hajtunk végre, akkor egy db hegy fog keletkezni. II/4-es ábra jobb oldala. (iterációk száma: i)



II/5. ábra: a, bal oldalt a körök algoritmussal generált magasságtérkép, b, jobb oldalt a magasságtérképből készített terrain

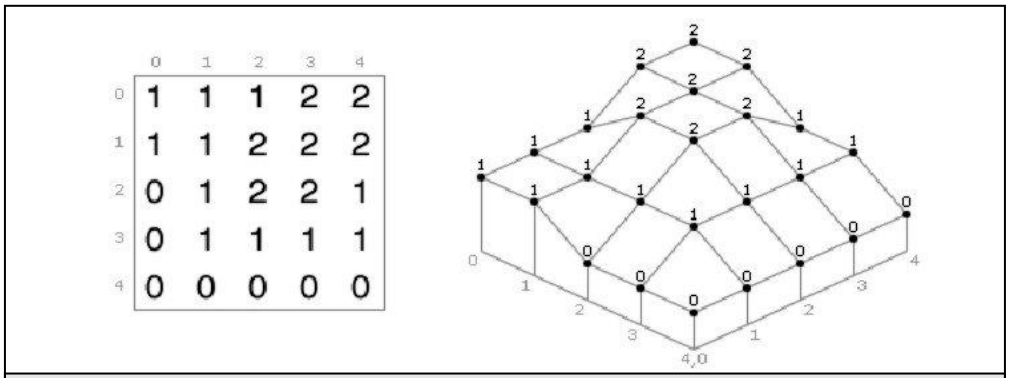
A kód eredménye a II/5-ös ábrán látható. Az algoritmus delphi implementációja a II/6-os ábrán látható. A kód lefordításához szükséges továbbá egy TButton és egy TImage komponenst is elhelyezni a formon.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 randomize;
 image1.Picture.Bitmap.PixelFormat := pf24bit;
 image1.Picture.Bitmap.width:=256;
 image1.Picture.Bitmap.height:=256;
End;

procedure TForm1.Button1Click(Sender: TObject);
 type
  PRGBArray = ^TRGBArray;
  TRGBArray = Array[0..1000000] of TRGBTriple;
 var rx,ry,
     x,y,i      : integer;
     P          : PRGBArray;
     max,radius : extended;
     h          : array[0..255] of array[0..255] of real;
     pd         : real;

begin
 for y:=0 to 255 do //a h tömb kinullázása
  for x:=0 to 255 do
   h[x,y] := 0;
 Max := 0; //a h tömb feltöltése
 for i:=1 to 400 do begin //400 iteráció
  rx:=random(256);
  ry:=random(256);
  radius:= random(100)+50;
  for y:=0 to 255 do begin
   for x:=0 to 255 do begin
    pd := (sqrt(sqr(x-rx)+sqr(y-ry))*2) / radius;
    if abs(pd) <= 1 then
     h[x,y] := h[x,y] + cos(pd*PI)+1;
    if h[x,y]>max then max := h[x,y];
   end;
  end;
 end;
 //a h tömb értékeinek átmásolása a képre (image1-re)
 for y:=0 to 255 do begin
  P := image1.Picture.Bitmap.ScanLine[y];
  for x:=0 to 255 do begin
   p[x].rgbtRed := round( (h[x,y]/max)*255 );
   p[x].rgbtGreen := round( (h[x,y]/max)*255 );
   p[x].rgbtBlue := round( (h[x,y]/max)*255 );
  end;
 end;
 image1.Repaint;
end;
```

II/6 ábra: A The Circles algoritmus delphi implementációja.

II/2 Heightmapból terrain

Ahhoz hogy a magasságtérképünkből elkészítsünk egy terraint, értelmeznünk kell a benne lévő információt. Korábban már beszéltünk arról hogy a kép egyes csatornái 8 bitesek. A 0 jelöli a sötét a 255 pedig a fehért. A terraint úgy fogjuk felépíteni, hogy ahol a magasságtérképen 255-ös érték szerepel, ott a terrain a legmagasabb csúcspontja lesz, ahol 0 ott a legalacsonyabb. A köztes értékeket pedig e két magasság között kell elhelyeznünk. A II/7-es ábra szemlélteti a módszert, egy 5x5-ös magasságtérképen.



II/7. ábra: a, Bal oldalt egy 5x5-ös tömb reprezentálja a magasságtérképet, b, jobb oldalt a mátrixból készített terrain.
A nagyobb értékek magasabban helyezkednek el.

Az elkészítés során meg kell oldanunk, hogy betöltsük a magasságtérkép képünket a memóriába egy tömbbe. Ennek a megvalósítására azonban nem térnék ki bőven, hiszen rengeteg kép formátum létezik és talán annál is több betöltési módszer. A téma terjedelme miatt csak egy fájl formátumot írok le bővebben. Én a program elkészítése során a bmp fájlformátumot használtam, mivel veszteségmentesen tárolja a kép információkat, könnyű kezelni és népszerű formátum. Delphiben egy bmp képet nagyon egyszerűen betölthetünk. A graphics unit tartalmaz egy Tbitmap nevű osztályt amelynek a LoadFromFile metódusával betölthetünk egy bmp képet.

De előtte ne felejtsük el példányosítani:

```
var bt:TBitmap;
    y,x:integer;
begin
 bt := Tbitmap.Create;
 bt.LoadFromFile('heightmap.bmp');
```

Miután sikeresen betöltöttük a képet, az adatokat az alábbi módon tudjuk elérni:

```
 for y:=0 to bt.Height do
  for x:=0 to bt.Width do
   intensity := getRValue(bt.Canvas.Pixels[x,y]);
end;
```

Két for ciklus segítségével bejárjuk a bittérképet. A bittérkép bt.canvas.pixels[x,y] tulajdonsága fogja megadni nekünk a kép x,y-adik koordinátájú pixel színét. Ezt a színkódot át kell konvertálnunk 0-255-re. Ezt megtehetjük a getRValue(color) függvénnyel (ezt a metódust a windows unitban találjuk). Ez a függvény az adott szín vörös intenzitását fogja visszaadni 0-255 érték között. Ha a képünk szürke árnyalatos, akkor ez a módszer jól működik, ha viszont színes kép tartalmazza a magasságtérképet, akkor az egyes pontokhoz tartozó színintenzitást megkaphatjuk ha az adott pixelhez tartozó vörös, zöld és kék intenzitásokat átlagoljuk:

```
intensity := (getRValue(bt.Canvas.Pixels[x,y])+
              getGValue(bt.Canvas.Pixels[x,y])+
              getBValue(bt.Canvas.Pixels[x,y])) div 3;
```

II/3 Vertex, normál és szín koordináták

Először tisztázzuk a vertex szó fogalmát. A vertex általában egy három dimenziós pontot jelöl egy csúcspontot. Egy 3D-s objektum/test csúcspontját jelöli.

Miután betöltöttük a képet és hozzá is tudunk férni a pixel adatokhoz, felépíthetjük a terrainünket. Első lépésben kiszámoljuk a vertex koordinátákat, méghozzá úgy, hogy a

magasságtérkép mx és my koordinátája lesz a vertex vx és vz koordinátája, továbbá a vertex vy koordinátája a magasságtérkép (mx,my) helyen felvett intenzitása lesz.

Tehát:

```
vx := mx;
vy := getRValue(bt.Canvas.Pixels[mx,my]);
vz := my;
```

Úgy szeretnénk elkészíteni a területünket, hogy be tudjuk állítani a méretét, magasságát és a minőségét. Ezeket paraméterként fogjuk átadni a vertex koordinátákat számoló metódusunknak. A kód Delphi implementációja a II/8-as ábrán látható. A delphi kód és a futtatható bináris fájl a heightmap_terrain.zip-ben található.

Paraméterként meg kell adnunk a már előzőleg beolvasott magasságtérkép tömbjét. Ez egy két dimenziós byte tömb: **type THeightMap = array of array of byte;**

A size paraméter a terület méretét adja meg, a quality paraméter pedig a terület részletességét adja meg. Minél kisebb a quality értéke, annál sűrűbben veszünk mintát a magasságtérképből, így az annál részletesebb lesz. A height paraméter pedig egy szorzó érték a terület magasságát, laposságát lehet vele állítani.

Az első sorban meghatározzuk a méret és minőség alapján, hogy mennyi vertexünk lesz, egy sorban ezt tartalmazza n. Az összes vertex szám n*n. A TerrainVertices tömb fogja tartalmazni a vertex koordinátákat, ez egy Tvertex3f típusú rekord tömb. A második, harmadik és negyedik sorban ezen tömb méretét állítjuk be. A következő pár sor változó inicalizálás. A kettős while ciklus pedig végig megy a magasságtérkép pontjain (quality sűrűséggel halad végig rajta) és kiolvassa a magasságtérképből az adott vertexhez tartozó magasságot (y) és beszorozza a height értékkel. Az aktuális vertexhez tartozó x értket a **round((size)/n *(xi+1));** összefüggés adja. A z értéket pedig a **round((size)/n *(yi+1));** összefüggés.

```
type TVertex3f     = record
 x,y,z : GLFloat;
End;
type THeightMap    = array of array of byte;

procedure calcTerrainCoords(heightMap:THeightMap; size,quality:integer; height
:real);
var
  x,z,i,n,xi,yi   : integer;
  dx,dy,xx,yy     : double;
Begin
 n := size div quality;                 // meghatározzuk mennyi vertex lesz

 setLength(TerrainVertices,  n);        // helyfoglalás a vertexek számára
 for i:=0 to n-1 do
  setLength(TerrainVertices[i],  n);

 (**)

 dx:= high(heightMap) / n;               // egy lépés a heightmapon x irányba
 dy:= high(heightMap[0]) / n;      // egy lépés a heightmapon y irányba
 x:=0;                              // x,z az aktuális vertex
                                     világkoordinátái
 xx:=0;                             // xx,yy a magasságtérkép koordinátái
 xi:=0;                              // xi,yi az aktualis vertex tömb
koordinataja
 while xi < n do begin
  z:=0;
  yy:=0;
  yi:=0;
  while yi < n do begin
   x := round((size)/n *(xi+1));
   z := round((size)/n *(yi+1));

   calcNormals(heightMap:THeightMap; xx,yy: double; xi,yi:integer);

   //VERTEX
   TerrainVertices[xi][yi].x:=x;
   TerrainVertices[xi][yi].y:=heightMap[round(xx),round(yy)]*height;
   TerrainVertices[xi][yi].z:=z;

   yy:=yy+dy;
   inc(yi);
  end;
  xx:=xx+dx;
  inc(xi);
 end;
End;
```

II/8 ábra: A Vertex koordináták kiszámítása a magasságtérképből.

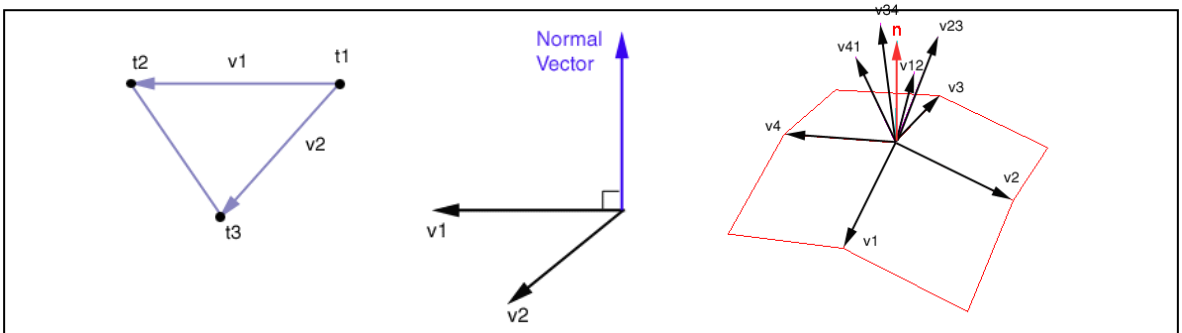
Most térjünk rá a *normálvektor*okra:

A normálvektorokra az árnyalási, megvilágítási számításoknál van szükség. Egy felület valamely pontjában vett normálisán azt a vektort értjük, amely az adott pontban merőleges a felületre, azaz a felület érintősíkjára. Az s(u,v) paraméteres formában adott felület (u0,v0) pontjában vett normálisa a

$$\frac{\partial}{\partial u}s(u0,v0)\times\frac{\partial}{\partial v}s(u0,v0)$$

vektor. A felületek legelterjedtebb ábrázolási módja, hogy a felületet poligonokkal, többnyire háromszögekkel közelítjük és a közelítő poligonokat jelenítjük meg valamilyen megvilágítási rendszer figyelembe vételével. Annak érdekében, hogy a képen a poliéder élei ne látszanak, a csúcsponthoz a pontban találkozó lapok normálisának az átlagát rendeljük normálisként. Ezzel finomabb lesz a megvilágítás.

Tehát a normálkoordinátákat a vertex koordinátákhoz fogjuk kiszámolni úgy, hogy az adott vertexhez tartozó háromszög felületének normálisát határozzuk meg, majd a szomszédos háromszögek normálisával átlagoljuk. Ezt úgy tesszük meg hogy a kérdéses vertexből és a hozzá tartozó háromszog másik két pontjából készítünk egy-egy vektort (v1,v2). II/9-es ábra bal oldalán és középen a t1 vertexhez készítjük el a normálist.



II/9 ábra: a, Bal oldalt a v1,v2 vektorok előállítása a t vertex pontokból.
b, Középen a v1 és v2 vektorok normálisa látható.
c, Jobb oldalt a szomszédos poligonok normálisai és az átlagolt n normálvektor.
Forrás: http://www.lighthouse3d.com/opengl/terrain/index.php3?normals

A két vektornak a normálisát a vektoriális szorzat (cross product) fogja adni, ezt a következő képpen számíthatjuk ki:

```
nx = v1y * v2z - v1z * v2y
ny = v1z * v2x - v1x * v2z
nz = v1x * v2y - v1y * v2x
```

Az n(nx,ny,nz) vektor lesz a v1 és v2 vektorhoz tartozó normálvektor. Szükséges továbbá ezt a vektort még normalizálnunk (egységnyi hosszúra skálázni), minden további számítások megkönnyítésére. Ezt könnyen megtehetjük, egyszerűen le kell osztanunk az egyes koordinátákat a vektor hosszával.

Az így előállt egységhosszúságú normálvektor:

```
nnx = nx / l
nny = ny / l
nnz = nz / l
```

ahol

```
l = sqrt(vx*vx+vy*vy+vz*vz).
```

A normálvektorok kiszámításának delphi implementációja a II/10-es ábrán látható. A makeVector metódus a két bemenő pontból készít egy vektort. A VectorCrossProduct a két vektorból előállítja a normálist. A VectorMultiply megfordítja a normálvektorokat, hogy kifelé mutassanak (-1 -el

```
calcNormals(heightMap:THeightMap; xx,yy: double; xi,yi:integer);
begin
 //NORMAL
 v1 := makeVector( vertex3f(x,heightMap[round(xx),round(yy)],z),

vertex3f(x+1,heightMap[round(xx+dx),round(yy)],z));
 v2 := makeVector( vertex3f(x,heightMap[round(xx),round(yy)],z),

vertex3f(x,heightMap[round(xx),round(yy+dy)],z+1));
 TerrainNormals[xi][yi] := VectorCrossProduct(v1,v2);
 TerrainNormals[xi][yi] := VectorMultiply(TerrainNormals[xi,yi],-1);
 normalizeVector(TerrainNormals[xi][yi]);
end;
```

II/10 ábra: A normál vektorok kiszámítása. (A II/8-as ábra calcNormals eljárása.)

szorozza). A normalizeVector metódus pedig normalizálja a vektort (egységnyi hosszúra állítja be). A TerrainNormals tömb fogja tartalmazni a normálkoordinátákat. V1, V2 pedig Tvertex3f típusú rekordok.

Szín koordináták:

A vertex koordináták mellé nem csak normál koordinátákat adhatunk meg, hanem szín koordinátákat is. Így a területet lehetőségünk van kiszínezni. Az OpenGL kétféle színmegadási módot használ. Az első módszerrel közvetlenül meg tudjuk adni a kívánt szín RGB komponenseit. A másik mód a színindex mód. Ezzel csak egy indexszel hivatkozunk a színtáblázat valamely elemére. Mi az előbbit fogjuk használni a megjelenítés során.

Tehát most az a feladatunk hogy az egyes vertex koordinátákhoz megadjunk valamiféle RGB komponenseket, ami jól reprezentálja a magasságtérképet. Ezt úgy érjük el, hogy a magas csúcsokhoz vagy „hegyekhez" világos színt (fehéret), az alacsony „völgyekhez" pedig sötét színt állítunk be (feketét). Az OpenGL a szín koordinátákat 0 és 1 között értelmezi, az ennél nagyobb kisebb értékeket levágja a 0-1 intervallumra. Adott az egyes vertexekhez a magasság érték, ami 0 és 255 közötti érték, ezt elosztva 255-el megkapjuk a kívánt mennyiséget. A delphi implementációt a II/11-es ábrán láthatjuk. A TerrainColors tömbben fogjuk tárolni az adott vertexhez tartozó színt. (A TerrainColors egy TVertex3f tömb, ezért szerepel (x,y,z) mező, de ezeket felfoghatjuk RGB mezőknek is).

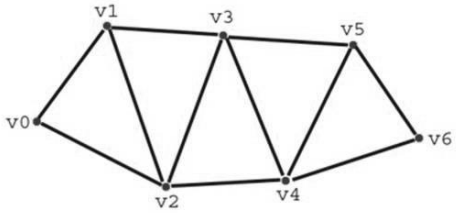
A rendszer a vertexekhez tartozó színeket fogja felhasználni a háromszög belső színeinek meghatározására. A belső színeket a vertexek színéből fogja interpolálni a glShadeModel beállításnak megfelelően.

```
//COLOR
TerrainColors[xi][yi].x:=heightMap[round(xx),round(yy)]/255;
TerrainColors[xi][yi].y:=heightMap[round(xx),round(yy)]/255;
TerrainColors[xi][yi].z:=heightMap[round(xx),round(yy)]/255;
```

II/11 ábra: A színek kiszámítása a területhez, úgy hogy a magasabb részek világosabbak legyenek.

II/4 Terrain megjelenítése

Miután meghatároztuk a vertex, szín és normál koordinátákat, kirajzolhatjuk a területünket. Át kell adnunk ezeket az információkat az OpenGL-nek. A területet háromszög-hálókból (Triangle Strip-ekből) fogjuk felépíteni (II/12-es ábra).



II/12. ábra: Egy háromszög háló. GL_TRIANGLE_STRIP
Forrás: http://www.codesampler.com/d3dbook/chapter_05/chapter_05.htm

A kód Delphi implementációja a II/13-as ábrán látható. Első lépésben meghatározzuk hogy mennyi vertexet fogunk kirajzolni egy sorba, n fogja tárolni ezt az értéket. Majd engedélyezzük a fények használatát, ha a felhasználó ezt kéri. Ezek után a kettős for ciklussal végigmegyünk a pontokon. Minden egyes új sor egy-egy GL_TRIANGLE_STRIP-lesz [2]. n ilyen sor van és n oszlop. A vertex koordinátákat a glVertex3fv OpenGL paranccsal adjuk meg, ahol a 3 arra utal, hogy egy koordinátahármast (x,y,z) adunk meg, az f jelöli, hogy lebegőpontos értékeket adunk meg, a v pedig, hogy nem magát az értéket adjuk át paraméterként, hanem az értékre mutató pointert. Delphiben a @ operátor adja meg az adott változóra mutató pointert. Az így leírt vertex a következő módon adható meg: **glVertex3fv(@TerrainVertices[x,y]);**

A normálvektort hasonlóan adjuk meg mint a vertexeket. A normálvektor megadására OpenGL-ben a glNormal3fv metódus szolgál, a 3fv itt is ugyanaz mint a vertex esetében. A színeket pedig a glColor3fv metódussal állíthatjuk be.

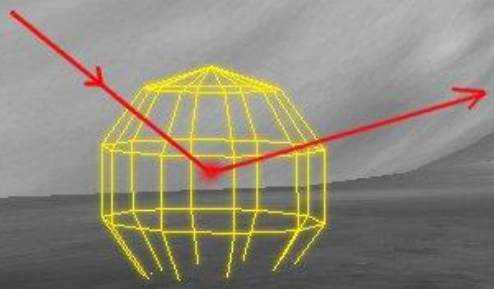
```
procedure drawTerrain;
 var n,y,x:integer;
begin
 n := size div quality;

 if form1.CheckBox5.Checked then
  glEnable (GL_LIGHTING)
 else
  glDisable (GL_LIGHTING);

 for y:=0 to n-2 do begin
  glBegin(GL_TRIANGLE_STRIP);
  for x:=0 to n-2 do begin

   glColor3fv(@TerrainColors[x,y]);
   glNormal3fv(@TerrainNormals[x,y]);
   glVertex3fv(@TerrainVertices[x,y]);

   glColor3fv(@TerrainColors[x+1,y]);
   glNormal3fv(@TerrainNormals[x+1,y]);
   glVertex3fv(@TerrainVertices[x+1,y]);

   glColor3fv(@TerrainColors[x,y+1]);
   glNormal3fv(@TerrainNormals[x,y+1]);
   glVertex3fv(@TerrainVertices[x,y+1]);

   glColor3fv(@TerrainColors[x+1,y+1]);
   glNormal3fv(@TerrainNormals[x+1,y+1]);
   glVertex3fv(@TerrainVertices[x+1,y+1]);

  end;
  glEnd;
 end;
end;
```

II/13 Ábra: A terület megjelenítését végző metódus.

A glShadeModel(param) parancs segítségével beállíthatjuk hogy az OpenGL a vertexek között interpolálja-e a színeket vagy sem. param=GL_SMOOTH esetén a színek interpolálva kerülnek megjelenítésre. param=GL_FLAT esetén az utolsó végpont színével rajzol a rendszer [3].

III ÜTKÖZÉSDETEKTÁLÁS

III/1 Ütközés vizsgálat
III/2 Kamera osztály definiálása
III/3 Kamera – terrain ütközés kezelése



Ütközésdetektálás

Az ütközés vizsgálat többnyire a fizikai szimuláció során jön elő. Arra vagyunk kíváncsiak, hogy kettő (vagy több) „objektum” a mozgása során ütközik-e. Általában ha ütközés következik be, akkor valahogyan reagálni is szeretnénk rá. Tehát valamilyen állapotváltozást szeretnénk leírni. Például az ütköző objektumok az érintkező felületeken elcsúsznak. Ahhoz, hogy egy ilyen ütközést és a hozzá tartozó válasz eseményt le tudjuk reagálni, szükséges lokalizálni az ütközést, vagyis meg kell határozni az ütköző objektumok azon felületét, amik érintkeznek vagy összeérnek. Ezen felületekből kiindulva további számításokkal meghatározható, hogy az egyes objektumok az ütközés után merre fognak elmozdulni [4].

A számítógépes világban az objektumok általában poligonhálók, amiket többnyire háromszögekkel írunk le. Ütközés lokalizálásakor ezen háromszögek közül keressük meg a megfelelőeket, majd ezek csúcspontjait, normálisait stb. figyelembe véve tudjuk meghatározni, hogy melyik objektumot merre mozgassuk, ha azt szeretnénk elérni, hogy elcsússzanak egymáson.

III/1 Ütközés vizsgálat

Ütközés vizsgálatot végezhetünk 2D és 3D esetben is. Mindkét esetben elég komoly matematikai háttér szükséges ezen számításokhoz. Főként síkokkal és vektorokkal számolunk [5]. Egy tipikus ütközés vizsgálat és reagálás a következő képen zajlik:

- A vizsgálandó objektum minden poligonján végighaladunk és vizsgáljuk, hogy a másik objektumból van e olyan poligon, amely metszi az aktuális poligonunkat.
- Ha nincs metsző poligon, akkor nincs ütközés.
- Ha van metsző poligon, akkor ütközést észleltünk, és a megfelelő állapotváltozást le kell írnunk. Ha az elcsúszást szeretnénk szimulálni, akkor ebben a lépésben meg kell határoznunk az elcsúszás vektor irányát és hosszát, amivel el kell tolnunk az objektumot.

- További vizsgálat szükséges, hogy az eltolt pozícióban szintén fellép-e ütközés.
- Mindezeket a lépéseket még azelőtt kell megtennünk, mielőtt kirajzolnánk az objektumokat a képernyőre, hiszen akkor látható lenne, hogy a két objektum egymásba ér, amit nem szeretnénk.

A bonyolult számítások nemcsak az olvasó kedvét vehetik el, hanem a hardvert is megterhelik, főleg ha magas poligonszámmal dolgozunk [6]. Ezért általában a teret fel szokták particionálni részekre és csak azokra a térrészekre vizsgálják az ütközést, ami szóba jöhet, ezzel lecsökkentve a vizsgálandó poligonok számát. További gyorsítási lehetőség, hogy a távoli objektumokat eleve nem vesszük bele az ütközés vizsgálatba. Továbbá az objektumot befoglaló gömbre, téglatestre (bounding-box) vizsgáljuk meg először az ütközést, ha ezekre nincs ütközés, akkor biztos, hogy a vizsgált objektum egyetlen poligonja sem ütközik a másikkal [7].

Mielőtt azonban belemennénk az ütközés vizsgálat bonyolult matematikai hátterébe, vizsgáljuk meg, mit is szeretnénk elérni és rá fogunk jönni, hogy nem is lesz az olyan bonyolult a mi esetünkben. Van egy magasságtérképünk, ezen szeretnénk valamiféle ütközés vizsgálatot végezni. Mondjuk egy gömb ütközését vizsgálni a magasságtérképpel. A gömb pedig lehetne a kamera, amit a felhasználó kedve szerint mozgathat. Tehát a cél, hogy a felhasználó a kamerát ne tudja a magasságtérkép alá mozgatni, ne essen át rajta, hanem finoman csússzon el a felületén. Ahhoz, hogy mindezt megvalósítsuk, szükségünk lesz egy kamera irányító osztályra, majd ha ezt elkészítettük, megvizsgálhatjuk a kamera – magasságtérkép ütközését.

III/2 Kamera osztály definiálása

A kamera osztály célja, hogy be tudjuk állítani, azt hogy honnan, merre nézünk és további metódusokra is szükség lesz ahhoz, hogy szimulálhassuk a kamera mozgását. Az FPS játékokban lévő kameramozgáshoz hasonlót fogunk elkészíteni: az egérrel körbe lehessen nézni, az előre és a hátra nyilakkal pedig a nézési irány felé közelítünk illetve távolodunk, a balra és jobbra iránygombokkal pedig az oldalazást valósítjuk meg.

Nem akarunk a terrainhez „ragadni”, nem lesz gravitáció, tehát repülni fogunk.

A kamerát a következő objektum fogja leírni:

```
type TCamera = Object
      Position : TVertex3f;      // The camera's position
      View     : TVertex3f;      // The camera's View
      UpVector : TVertex3f;      // The camera's UpVector

      procedure GoToCamera(positionX, positionY, positionZ : glFloat;
                           viewX, viewY, viewZ : glFloat);
      procedure MoveCameraByMouse;
      procedure RotateView(const X, Y, Z: glFloat);
      procedure StrafeCamera(speed : glFloat);
      procedure MoveCamera(speed: glFloat);
      procedure SetViewport;
end;
```

ahol a Position a kamera középpontját fogja meghatározni, a View vektor a nézési irányt írja le, az UpVector pedig a felfelé mutató irányt, ez a mi esetünkben mindig (0,1,0) irány lesz.

A GoToCamera metódus beállítja a Position, View, UpVector rekordok mezőit a megadott értékekre. Tehát a kamerát a megadott pozícióba mozgathatjuk vele és a nézési irányt is beállíthatjuk.

A MoveCameraByMouse metódus fogja úgy változtatni a position és view értékeket hogy az megfeleljen az egér mozgásához (a körbenézést valósítja meg). Meghatározza, hogy az egér az adott időben mennyit mozdult el vízszintesen és függőlegesen, ezeket az értékeket felhasználva forgatja a nézetet.

A RotateView metódus a nézési irány vektort forgatja el a kamera középpontja (Position) körül a megadott mértékkel, ez a metódus a kamera egér általi mozgatásához szükséges. Egyszerű trigonometrikus függvényekkel van benne implementálva a forgatás.

A StrafeCamera az oldalazást, a MoveCamera az előre-hátramenetet valósítja meg, adott sebességgel mozgatja el a kamerát. A MoveCamera egyszerűen a pozícióhoz adja a nézési vektor

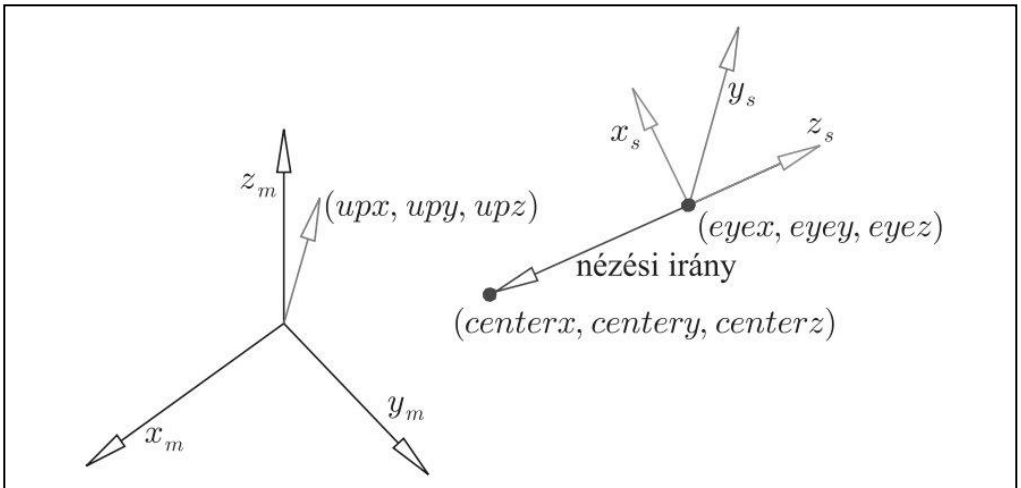
speed-szeresét. Ezzel megkapjuk az új pozíciót. A StarfeCamera metódus kicsit bonyolultabb, ez a felfelé vektor és a nézési irányvektor vektoriális szorzatának (cross productjának) a speed-szeresét adja a position értékhez. Ennek a vektoriális szorzatnak az eredménye egy olyan vektor, ami merőleges az UpVector és a View vector által meghatározott síkra. Az oldalazáshoz pontosan erre van szükségünk.

A SetViewport metódus pedig mindezen kiszámolt értékeket „átadja” az OpenGL-nek és a modellkoordináta-rendszerben beállítja azokat. Ezt a beállítást a GLU függvénykönyvtár gluLookAt() függvénnyével tehetjük meg. A gluLookAt paraméterei:

```
gluLookAt(eyex, eyey, eyez : Gldouble;
          centerx, centery, centerz : Gldouble;
          upx, upy, upz : Gldouble;)
```

ahol (eyex, eyey, eyez) az új nézőpont, (centerx, centery, centerz) az új nézési irányt és az (upx, upy, upz) az új y tengely irányát írja elő. A gluLookAt metódus paraméterei a III/1-es ábrán látható.

A fejezethez tartozó program kamera könyvtárában megtalálható egy példa program ami demonstrálja a fentieket. Tartalmazza a magasságtérképből felépített terraint és a kamera mozgatását. A camera osztály implementációja a camera.pas unitban található.



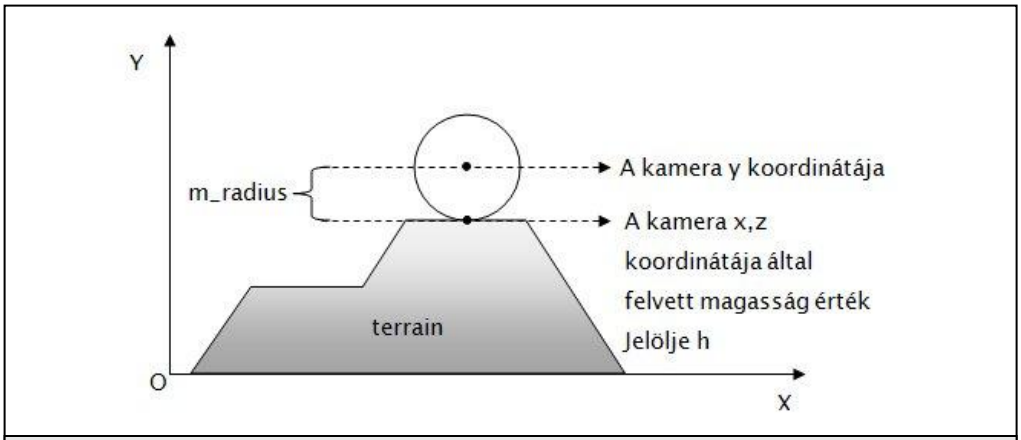
III/1. ábra: Az új nézőpont és nézési irány megadása a gluLookAt függvénnyel. Forrás: Juhász Imre – OpenGL jegyzet

III/3 Kamera – terrain ütközés kezelése

Tehát szeretnénk a kamerát befoglaló gömb és a magasságtérkép ütközését kezelni olyan módon, hogyha ütközne a kamera a terrainnel, akkor elcsússzon a felületén.

Nézzük meg először is, hogy mi szükséges ahhoz, hogy detektáljuk az ütközést. Fentebb már volt szó róla, hogy a vizsgálandó objektumok poligonjait kell néznünk, vajon metszésben vannak-e. A mi esetünkben a kamera egy pont (vagyis egy gömb), aminek van egy m_radius sugara. Vonjuk ki a kamera y koordinátájából az m_radius értéket, ezzel megkapjuk a gömb legalsó pontját. Ezt a pontot kellene megvizsgálnunk a terrain minden egyes poligonjára, hogy valamelyik alatt van-e. Mivel adva van a magasságtérképünk, ami a terrain magasságértékeit tárolja, ezért inkább a következőt tesszük: a kamera világkoordinátáit (x,z koordinátáit) áttranszformáljuk a magasságtérkép értelmezési tartományába és leolvassuk az ebben a pontban levő magasság értéket. Ha ez az érték nagyobb mint a kamera y koordinátája, akkor a kamera a terrain alatt van, tehát ütközést észleltünk. Ez a módszer sokkal egyszerűbb és gyorsabb, mintha minden egyes poligonra megvizsgálnánk hogy alatta van-e.

Ha detektáltuk az ütközést, akkor szeretnénk azt lereagálni, hogy a kamera ne nézhessen be a terrain alá. Ezt úgy érjük el, hogy a kamera y koordinátája legyen egyenlő a magasságtérkép-beli érték (III/2-es ábra h értéke) és az m_radius érték különbségével. Ezzel azt érjük el, hogy a kamerát úgymond visszadobjuk a terrain tetejére.



III/2. ábra: A kamera ütközése a terrainnel.

A III/3-as ábra kódja valósítja meg az előbbi lépéseket.

```
procedure CheckCameraTerrainCollision(m_radius:real);
 var h,v:real;
begin
  h:=getHeight(cam.Position.x,cam.Position.z,heightMult);
  if cam.Position.y-m_radius<h then begin
   v:= cam.Position.y-cam.View.y;
   cam.Position.y := h+m_radius;
   cam.View.y     := h+m_radius-v;
  end;
end;
```

III/3 ábra: A terrain – kamera ütközés vizsgálata és az ütközés lereagálása.

Most nézzük meg a getHeight metódust egy kicsit közelebbről. Ez a metódus fogja a kamera világkoordinátáit áttranszformálni a magasságtérkép értelmezési tartományába és leolvasva az ott lévő értéket visszaadja a kurrens magasságértéket. A III/4-es ábrán lévő kód valósítja meg ezt. Az ix és iy változó értékei lesznek az áttranszformált koordináták. Ezeket az értékét úgy kapjuk meg, hogy a kamera megfelelő koordinátáját megszorozzuk a magasságtérkép szélességének-magasságának és a terrain szélességének-magasságának a hányadosával. A validHeightmapCoords metódus értéke igaz lesz ha a benne szereplő paraméterek a magasságtérkép egy valós pontját alkotják (a magasság térképen belül van).

```
function  getHeight(x,z: real; height:real) : real;
 var ix,iz : integer;
begin
 ix := round(x*high(heightmap)/size);
 iz := round(z*high(heightmap[0])/size);
 if validHeightmapCoords(ix,iz) then
  result := heightMap[ix,iz]*height;
end;
```

III/4 ábra: A kamera x,z koordinátája által meghatározott magasságérték meghatározása.

Ez a módszer nagyon egyszerű, viszont ha gyorsan változik a magasságérték, akkor a kamera mozgása, visszalökése a terrain felé túlságosan darabos lesz.

Finomabb mozgást is megvalósíthatunk úgy, hogy nem csak a kamera alatti magasságértéket használjuk fel az új pozíció kiszámítására, hanem a szomszédos pontok értékét átlagolva. De még finomabb eredményt kapunk, ha egy bilineáris interpolációt használunk a magasságérték meghatározására. A bilineáris interpoláció implementációja a III/5-ös ábrán látható [8]

```
function  getHeight(x,z: real; height:real) : real;
 var ix,iz : integer;
     fx,fz:double;
     topLeft,topRight,
     bottomLeft,bottomRight,percentX,percentZ,omdx,omdz:real;
begin
 fx := x*high(heightmap)/size;
 fz := z*high(heightmap[0])/size;
 ix := round(fx);
 iz := round(fz);
 if not (validHeightmapCoords(ix,iz)and
         validHeightmapCoords(ix+1,iz)and
         validHeightmapCoords(ix,iz+1)and
         validHeightmapCoords(ix+1,iz+1)) then exit;
 topLeft     := heightMap[ix,  iz]  *height;
 topRight    := heightMap[ix+1,iz]  *height;
 bottomLeft  := heightMap[ix,  iz+1]*height;
 bottomRight := heightMap[ix+1,iz+1]*height;
 percentX    := fx - (ix);
 percentZ    := fz - (iz);
 omdx := 1 - percentX;
 omdz := 1 - percentZ;
 result :=  omdx*omdz         *topLeft +
            omdx*percentZ     *bottomLeft +
            percentX*omdz     *topRight +
            percentX*percentZ *bottomRight;
end;
```

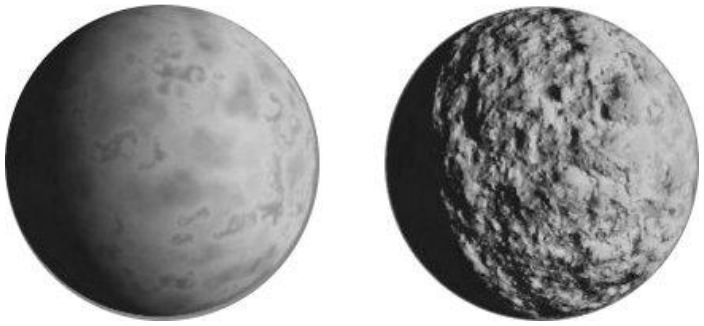
III/5 Ábra: A kamera x,z koordinátája által meghatározott magasságérték meghatározása bilineáris interpolációval.

IV SHADEREK

Shaderek

A shaderek kicsi önálló programok, amiket a videókártya megfelelő proceszora futtat. Feladatuk, hogy a képalkotás során a grafikai csővezeték egyes fix automata folyamatait leváltsák, programozhatóvá tegyék. Ezáltal lehetőség nyílik a programozó számára, hogy sokkal élethűbb, realisztikusabb képet készíthessen, már amennyire ez a célja. Segítségükkel számos effektet megvalósíthatunk. Mindezen műveleteket a GPU (Graphics Processing Unit) megfelelő egysége számolja és a szükséges adatokat a videó memóriából tudja elérni, ami közvetlen közel van hozzá, így nagyon nagy teljesítményt tudunk elérni ezekkel a programokkal.

A shaderek nagyon sokoldalúan felhasználhatók. Manapság többnyire a játékfejlesztés és a filmgyártás területén használatosak. Számos shader program található az interneten is, amelyek különböző effekteket valósítanak meg mint például a Bump-mappping (bucka leképezés), aminek segítségével az adott objektum egy egyenletlenebb képet kap az egyenletes helyett, így sokkal valósághűbb látszatot keltve, hiszen a természetben sohasem fordul elő tökéletesen egyenletes felület.



IV/1. Ábra: Bal oldalt egy gömb felülete bump-mapping nélkül, jobb oldalt bump-mappingel.
Forrás: http://www.chromesphere.com/Tutorials/Vue6/Optics_Basics_Print.html

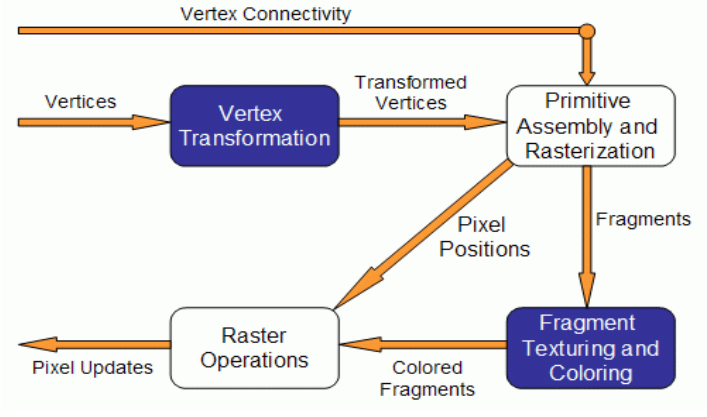
Egy másik példa lehet a realisztikus tükröződő vízfelület, amelynek a felszínén a víz fodrozódik. Egy ilyen shader programot sem túl nehéz megírni, az eredmény és teljesítmény mégis bámulatos lesz. Ezenkívül számtalan példát lehetne még felhozni a shaderek erősségére.

A hardvergyártó cégek is felismerték a shaderekben rejlő lehetőségeket, és arra törekszenek, hogy minél több fix csővezetékes folyamatot leváltsanak programozható részekre, minél több lehetőséget adva a programozók kezébe.

Mint ahogy programozási nyelvekből is többféle létezik, a shader nyelvekből is több létezik, mint például a HLSL (High Level Shader Language) vagy a GLSL (Opengl Shader Language) [9]. A GLSL nyelvvel fogunk foglalkozni a következő részekben [10]. Először megnézzük a standard grafikus képalkotás leegyszerűsített csővezetékét, majd az egyes programozható részeket vizsgáljuk meg, legvégül a fejezet végén egy egyszerű shader példaprogrammal mutatjuk be a működését.

IV/1 3D csővezeték

A Grafikus csővzeték egy nagyon lebutított absztrakt változata látható a IV/2-es ábrán.



IV/2. ábra: Standard grafikus képalkotás csővezetéke.
Forrás: http://www.lighthouse3d.com/opengl/glsl/index.php?pipeline

Az ábrán láthatóak a csővezeték egyes szakaszai és a közöttük haladó adat útvonala. Az egyes részek [11]:

Vertex transformation:

A bemenetet ebben a részben a vertexek, alkotják amelyek rendelkeznek egy bizonyos attribútum halmazzal, mint például a pozíció, szín, normál, textúra koordináták és további más tulajdonságokkal.

Néhány művelet, ami végrehajtódik ebben a szakaszban a vertexeken:

1, vertex pozíció transzformáció,

2, fények számítása a vertex pontokban,

3, a textúra koordináták generálása és transzformálása.

Primitive Assembly and Rasterization:

Ennek a résznek a bemenetét a transzformált vertexek alkotják, valamint a vertexek közötti kapcsolatok információja. Ez utóbbi információ azt mondja meg, hogyan kell a vertexeket összekapcsolni, hogy a megfelelő primitívet kapjuk. Ez a rész itt tulajdonképpen a primitívek úgymond összeszerelése.

Továbbá ez a rész is felelős a vágási műveletekért, mint például a frustum vágás és a hátsó lapok eltávolítása.

A raszterizáció határozza meg a primitívek fragmentjeit és pixel pozícióit. A fragmentum ebben az állapotban még csak egy adatrész amit arra használhatunk, hogy a frame bufferben egy pixelt frissítsünk egy adott helyen. A fragmentum nem csak színt tartalmaz, hanem normálisokat, textúra koordinátákat és az összes olyan attribútumot ami az új pixel színének kiszámításához szükséges.

Tehát ebben a szakaszban a vertex adatokat és a kapcsolódasi információkat felhasználva, interpolálva elő tudjuk állítani a fragmentumok színét.

Fragment texturing and Coloring:

Ennek a résznek a bemenetét az interpolált fragmentum információk alkotják. A

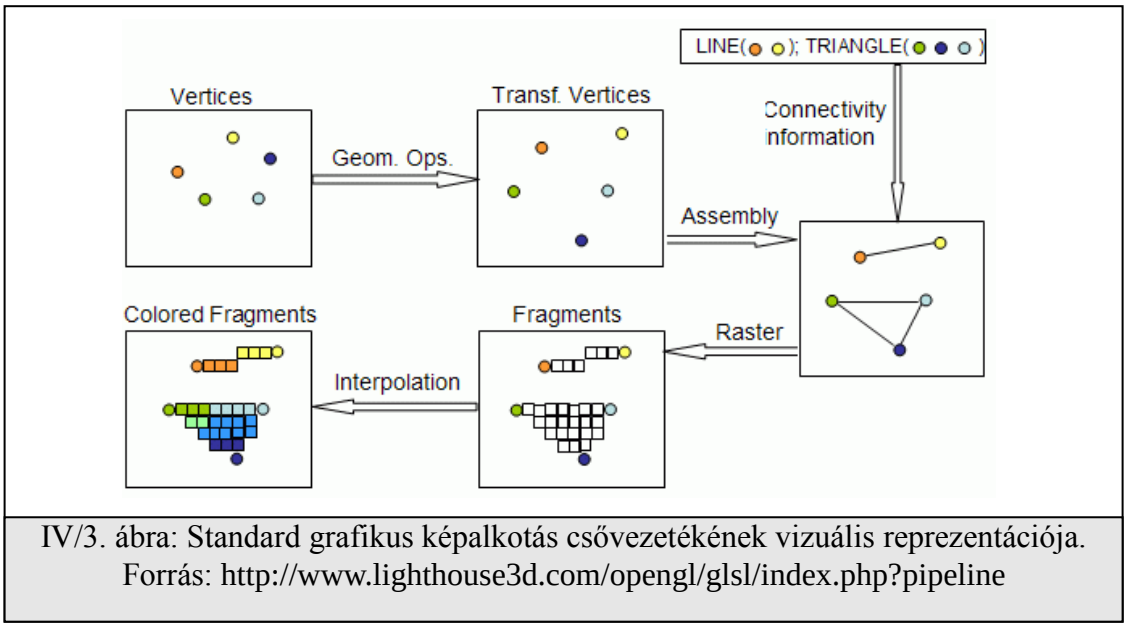
fragmentum színe már ki lett számolva az előző részben. Ebben a részben ezt a színt – ha szükséges – kombináljuk egy texel-el (texture elementel). A textúra koordinátákat már az előző részekben meghatároztuk és interpoláltuk. Ha szükség van atmoszférikus köd effektre, akkor azt itt tudjuk alkalmazni. Ennek a szakasznak a végeredménye egy fragmentum, ami rendelkezik színnel és mélység értékkel.

Raster Operations:

Bemenet: a pixel pozíciója, a fragmentum színe és mélység értéke. A csővezeték utolsó szakaszában néhány tesztet végzünk el a fragmentumokon, nevezetesen: kivágási vizsgálat, alfa vizsgálat, stencil vizsgálat és mélységvizsgálat.

Ha sikeresen átment a fragmentum a vizsgálaton, akkor az információkat felhasználva frissíti a pixel értékét a megadott keverési módnak megfelelően. A frame buffer kizárólag ebben a szakaszban érhető csak el.

Vizuális reprezentációja az egyes szakaszoknak:



IV/3. ábra: Standard grafikus képalkotás csővezetékének vizuális reprezentációja.
Forrás: http://www.lighthouse3d.com/opengl/glsl/index.php?pipeline

A shaderekkel lehetőségünk nyílik a V*ertex transformation* és a *Fragment texturing and coloring* szakaszok funkcionalitásának programozására. A vertex shaderek felülírják a *Vertex transformation* szakaszt, míg a Fragment shaderek (pixel shaderek) felülírják a *Fragment texturing and coloring* szakaszt. A továbbiakban megismerkedünk ennek a két programozható egységnek a működésével.

IV/2 Vertex processzor

A vertex processzor felelős a vertex shader futtatásáért. A vertex shader bemenete a vertex adatok (pozíció, szín, normál, stb...), attól függ, mit küld neki az OpenGL alkalmazás. Egy vertex shaderben a következő feladatokat végezhetjük el:

- Vertex pozíciójának transzformációja a modellview és a projekciós mátrixokat felhasználva.
- A normálkoordináták transzformációja és ha szükséges akkor normalizálásuk.
- Textúra koordináta generálása és transzformálása.
- Megvilágítás számolása vertexre.
- Szín kiszámítása.

Ahogy láthattuk a csővezeték lépéseiből, a vertex processzor nem rendelkezik információval a vertexek kapcsolódásáról. Tehát például egy back face cullingot nem tudunk vele megvalósítani, mivel nem tudjuk mely vertexek alkotnak egy háromszöget, sőt a a vertex shader mindig csak egy adott vertexen operál, nem vertexek halmazán.

A Vertex Shader minden egyes vertexre lefut, nem számít hogy hol van, milyen messze, milyen irányban. A IV/4-es ábra bal oldalán látható, hogy minden vertexre lefut, azokra is amelyek kívül esnek a látómezőn. Egy alap Vertex Shader annyit csinál, hogy az átadott három mártix-szal (Világ, Nézeti és Projekciós) megszorozza a vertex pozícióját. A vertex shader a gl_position változóba írja a kimeneti vertexeket.

IV/3 Fragment processzor

A fragement processzor felelős a fragment shader futtatásáért. A fragment shader bemeneteit az előző csővezeték szakasz kimenetének az interpolált értékei alkotják: vertex, pozíció, szín, normálvektorok stb... Ezek az értékek a vertex shaderben kerültek kiszámolásra, de csak a vertex pozíciókban, mi most magán a primitíven belül fogunk dolgozni, ezért van szükség az interpolált értékekre. A fragment shaderben a következő müveleteket végezhetjük el:

- Kiszámíthatjuk a színeket, és a textúra koordinátákat minden pixelre.
- A textúrát itt alkalmazhatjuk.
- Köd effektet számíthatunk.
- Kiszámolhatjuk a normálkoordinátákat is minden pixelre, ha pixelenkénti megvilágítást akarunk.

Mint ahogy a vertex processzor felülírja a rögzített funkcionalitást, úgy a fragment shader is egy az egyben felülírja a *Fragment texturing and coloring* egységet, így azt nekünk kell implementálni. Nem lehet olyan fragment shadert írni, ami textúrázza a fragmentet, majd a statikus funkcionalitáshoz visszatérve alkalmazza rá a köd effektet például. A programozónak itt kell lekódolnia az összes effektet amire szüksége van az alkalmazásban.

A fragment shader egy fragmenten dolgozik, azaz semmilyen információnk nincs a szomszédos fragmentekről. A fragment shader hozzáférhet az OpenGL állapotokhoz, hasonlóan a vertex shaderekhez, így tudjuk például a köd színét OpenGL-ben meghatározni, majd itt a shaderben azzal a színnel számolni.

Fontos tehát, hogy a fragment shader nem tudja megváltoztatni a pixel koordinátáját, az az előző csővezeték szakaszban már kiszámításra került. Hozzáfér a pixelek helyéhez a képernyőn, de megváltoztatni a helyüket nem tudja.

A fragment shader két kimeneti lehetőséggel rendelkezik:

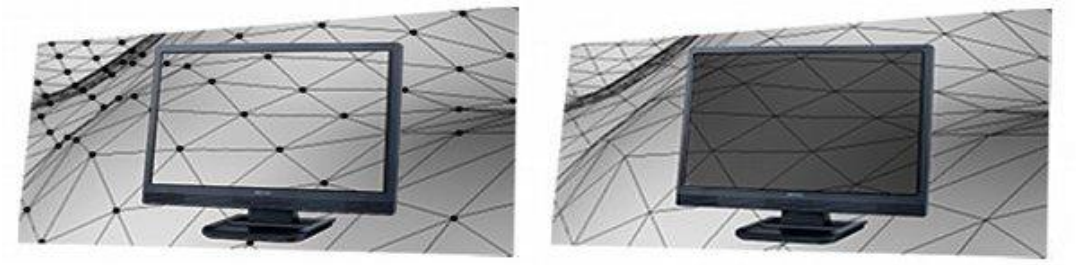
- Eldobhatjuk a fragmentumot, így semmi sem kerül a kimenetre.

- vagy átadhatjuk a kiszámolt fragmentumot a gl_FragColor változónak és ez lesz a fragmentum végleges színe.

A fragment shader nem fér hozzá a frame bufferhez, ez azt jelenti hogy a blending művelet a fragment shader lefutása után lesz végrehajtva.

Összefoglalva:

Ha például a vertex shader 3 db vertexen hajtott végre műveleteket és mondjuk kiszámolta a vertexekhez a színt is, majd a csővezeték következő egysége a vertexekből és a vertexek kapcsolódási információiból öszerakott egy háromszöget, amit ki kell színezni, akkor a háromszög csúcspontjai (a három vertex) színéből interpolációval megkapjuk a háromszög belső pontjainak színét minden egyes pixelre. A fragment shader tehát a képernyőre kerülő pixelekre fut le és nem a textúrákon megy végig, nem összekeverendő a texel (texture element) és a pixel (picture element) fogalma.



IV/4. ábra: Bal oldalt a sötét pontok a vertexek, a vertex shader ezeken operál, jobb oldalt a sötét pixelek a a fragmentumok, a fragment shader ezeken operál.

IV/4 GLSL program használata.

Most pedig következzen egy példaprogram. A nagyon egyszerű kis „hello world” shader alkalmazásunk nem fog mást csinálni, mint egy háromszöget megjelenít, azt vörösre színezve félig átlátszó módon.

A fejezethez kapcsolódó példaprogram forráskódjában látható, hogy a dpr fájlunk mérete megnőtt. Ennek oka az, hogy az eddigiekben használt TopenglPanel komponenst, ami az OpenGL-es felület létrehozásáról gondoskodott, elhagyjuk és magunk hozzuk létre azt a form-ot amiben az OpenGL es környezet fut. Egy OpenGL-es ablak létrehozásáról bővebben a NeHe[3] weboldalán olvashatunk.

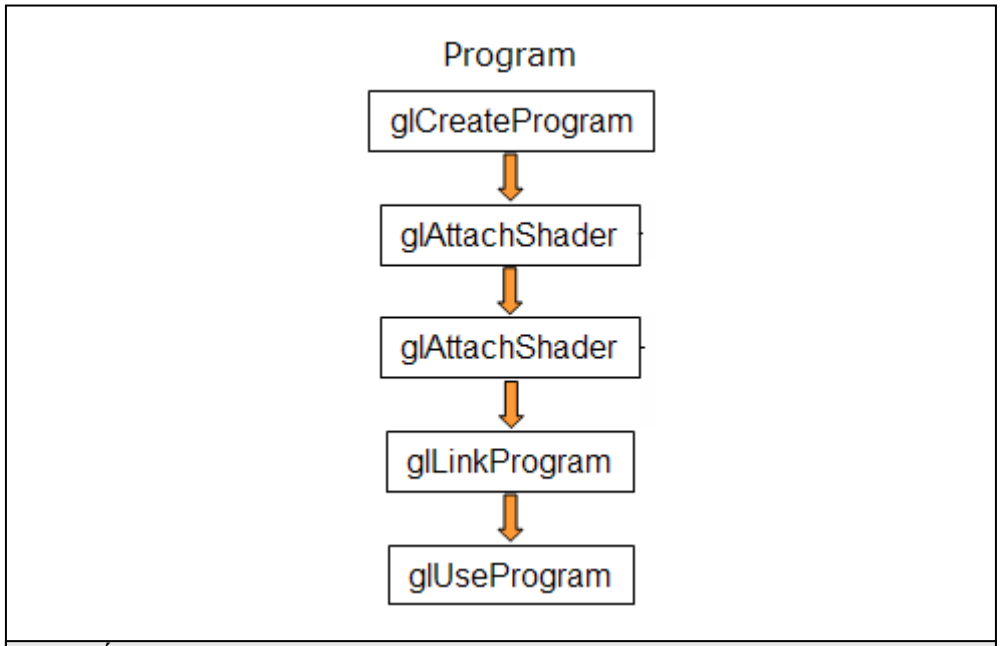
A main.pas fájlba is került némi ablakkezelési változó és típus, de a lényeg ebben a fájlban a BeforeStart, BeforeExit, UpdateInput és a Render eljárások. Ezek funkcionalitásilag rendre a következők:

- BeforeStart: lefut mielőtt az ablak megjelenik. Ide érdemes inicalizációs lépéseket írni, mint például textúrák beolvasása, változók inicalizálása, stb...
- BeforeExit: lefut mielőtt az alkalmazás bezáródna. Memória felszabadítás, esetleg merevlemezre írt átmeneti adatok törlése, stb...
- UpdateInput: Amíg az alkalmazás fut, ebbe az eljárásba mindig belép (végtelen ciklusban van). Ide érdemes írni a billentyűzet és egérkezelési lépéseket. A leütött és felengedett billentyűket a g_keys globális változón keresztül tudjuk elérni.
- Render: Ez az eljárás is végtelen ciklusban van amíg a program fut. Ide írjuk a renderelni kívánt OpenGL kontextust.

További két .pas fájlt is felfedezhetünk a forráskódban. Ezek közül a dglOpengl.pas[4] egy OpenGL header fordítás Delphire és Freepascalra. A Shader.pas osztály pedig a GLSL shaderek betöltéséről és a programhoz való csatolásáról gondoskodik.

3 NeHe tutorial OpenGL ablak létrehozására: http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=01
4 A dglOpengl hivatalos weboldala: http://www.delphigl.com/

Nézzük akkor a forráskódot lépésről lépésre. A BeforeStart metódus (IV/6 ábra) nem sok mindent tartalmaz, egy pár függvényhívást, ami a shader.vert és a shader.frag fájlokból beolvassa a vertex és fragment shader programokat majd, linkeli őket. Itt most álljunk meg és nézzük át röviden, hogy mi szükséges ahhoz, hogy shader programokat futtassunk, mert ahhoz hogy a shader programokat használhassuk, fel kell készítenünk a programunkat rá. A IV/5-ös ábra mutatja a



IV/5. Ábra: a szükséges lépések hogy shader programokat használhassunk. Forrás: http://www.lighthouse3d.com/opengl/glsl/index.php?oglprogram

szükséges lépéseket.

- Az első lépés, hogy létrehozzunk egy objektumot, ami majd a programunkat tartalmazza (shader container). OpenGL parancs: *GLuint glCreateProgram(void);*

A parancs visszatérési értéke egy az objektumra mutató kezelő (handler).

- Következő lépésben csatoljuk az előző szakaszban létrehozott shadert a programhoz. A shadert itt még nem kell fordítani, még forráskóddal sem kell rendelkeznie. Minden ami a csatoláshoz szükséges, benne van a container-ben. Ha vertex és fragment shader programunk

is van, akkor mindkettőt csatolni kell. OpenGL parancs:

```
void glAttachShader(GLuint program, GLuint shader);
```

Ahol a program paraméter a program kezelője (handler), a shader paraméter pedig a csatolni kívánt shader kezelője.

- A következő lépés a linkelés. Ahhoz, hogy ezt elvégezzük, a shader programokat le kell fordítani (compile). OpenGL parancs: *void glLinkProgram(GLuint program);*

  A paraméter ugyanaz mint az előzőkben.
- A végső lépésben pedig a *void glUseProgram(GLuint prog);* paranccsal használhatjuk a paraméterként megadott programunkat. Ha a paraméter értéke 0 akkor a statikus OpenGL csővezeték lesz az aktív és a shader programunk nem lesz használva.

Tehát ezek a lépések mennek végbe a BeforeStart metódusban az utolsó lépés kivételével, hiszen a shader programot itt még nem szeretnénk használni, majd csak a renderelésnél.

A következő metódus az UpdateInput (IV/6 ábra), itt csak annyit ellenőrzünk le hogy az ESC billentyű le van e nyomva. Ha igen akkor a program befejezi működését.

A Render metódus (IV/6 ábra) már érdekesebb. Ebben némi képernyőtörlés van és egy nézeti mód átállás merőleges vetítésre (glOrtho). Majd a shader program szempontjából fontos sor a *glUseProgramObjectARB(Shader);*. Itt kezdjük el használni a shader programunkat. Ezután kirajzolunk egy háromszöget fehér színnel. A képernyőn mégsem fehér színű háromszög fog megjelenni, mivel ezt a színt magán a shader programon belül felülírjuk vörösre félig átlátszóan. A háromszögrajzolás után pedig kikapcsoljuk a shader program használatát a *glUseProgramObjectARB(0);* paranccsal.

Mindezek után vizsgáljuk meg magukat a shader programokat. Mint már fentebb említettük, ezek a shader.vert és a shader.frag állományokban találhatóak. A IV/7 -es ábrán láthatóak a programok.

```
procedure BeforeStart;
begin
 Shader := LinkPrograms( [
      LoadShaderFromFile('shader.vert', GL_VERTEX_SHADER_ARB),
      LoadShaderFromFile('shader.frag', GL_FRAGMENT_SHADER_ARB)
    ] );
End;

procedure UpdateInput;
begin
  if g_keys.keyDown[VK_ESCAPE] then  begin
   PostMessage(g_window^.h_Wnd, $0012, 0, 0);
   g_isProgramLooping := FALSE;
  end;
End;

procedure Render;
begin
  glClearColor(0.8, 1, 1, 1);
  glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity;
  glOrtho(0,APPLICATION_WIDTH,APPLICATION_HEIGHT,0,-1,1);
  glMatrixMode(GL_MODELVIEW);

  glUseProgramObjectARB( Shader );
   glcolor4f(1,1,1,1);
   glBegin(GL_TRIANGLES);
    glVertex2f(30,30);
    glVertex2f(230,30);
    glVertex2f(230,230);
   glEnd;
  glUseProgramObjectARB( 0 );
  glFlush();
end;
```

IV/6 ábra: A fejezethez tartozó program forráskódjából egy részlet.
Az inicalizáló a billentyűzetkezelő és a render eljárások.

A vertex shaderben a gl_Color változóban tudjuk elérni az OpenGL-ben megadott színt a vertexhez, ami most a mi esetünkben fehér lesz, hiszen ezt a színt állítottuk be a render eljárásban korábban. Ezt a szín értéket rendeljük hozzá a front face-hez. Így a fragment shaderben a gl_Color változóban a háromszög három vertex színéből interpolált értéket kapnánk, ha használnánk azt (jelen esetben az interpolált szín értékek egytől egyig sima fehérek lennének, hiszen minden vertex színe ugyan az: fehér). A fragment shaderben nem használjuk ezt a beépített változót, helyette külön kézzel adjuk meg a fragmentum színét egy négy elemű vektorral leírva: *gl_FragColor = vec4(1,0,0,0.5);*.

A vektor értelmezése: (vörös, zöld, kék, alfa) intenzitás értékek. Így a (1,0,0,0.5) vektor pontosan a vörös félig áttetsző fragmentum színt fogja adni. A vertex shaderben a gl_Position változóba írjuk a kimenetet, ami nem más mint a kurrens vertex megszorozva a kurrens modell nézeti mátrixszal.

```
void main()
{
        gl_FrontColor = gl_Color;
        gl_Position   = gl_ModelViewProjectionMatrix * gl_Vertex;
}



// Fragment shader
void main()
{
        gl_FragColor = vec4(1,0,0,0.5);
}
```

IV/7 ábra: A fejezethez tartozó program shader programjai. A vertex és fragment shader.

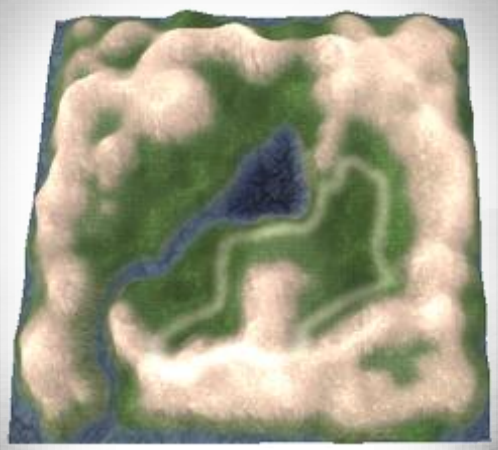
Megjegyzés: Az áttetszőséget, mint ahogy azt már az előzőkben leírtuk, a shader program lefutása után fogja végrehajtani a rendszer, ezért azt OpenGL-ben nekünk még engedélyeznünk kell. A blending beállítása a GLWindow.dpr fájlban található meg az initalize metódusnál:

glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

glEnable(GL_BLEND);

V SHADERES SZÍNEZÉS

V/1 Textúrázás,textúra koordináták.
V/2 Textúra a terrainen.
V/3 Két textúra vegyítése a terrainen.
V/4 Négy textúra vegyítése a terrainen.



Shaderes színezés

Szeretnénk elérni, hogy a megjelenített terrain még élethűbb, még realisztikusabb legyen, ne csak egy valamilyen színnel árnyalt terrain legyen. Azt viszont nem engedhetjük meg magunknak, hogy a modell geometriájával alakítsunk ki minden kis részletet (gödröket, repedéseket, stb...) a terrainen, hiszen azt még a legjobb gépeken se tudnánk valós időben megjeleníteni (ráadásul ember legyen a talpán, aki ilyen részletességgel képes lenne lemodellezni és beszínezni a terraint), éppen ezért textúrákat fogunk használni.

V/1 Textúrázás, textúra koordináták

2D-s textúrákkal fogunk dolgozni, a textúra egy téglalap alakú terület, ami sorokba és oszlopokba rendezett textúraelemekből, texelekből (texture element) áll. Az egyes texelekhez pedig 3 vagy 4 adat fog tartozni, attól függően hogy alfa csatornája van-e a textúrának. A vertexek mellett meg kell adnunk majd a textúra koordinátákat is, melyek egy egységnégyzeten belül változnak, tehát a textúra koordináták 0 és 1 közötti értékek. Ezen a területen kívül eső textúra koordinátákat is megadhatunk, de akkor elő kell írnunk, hogy a rendszer hogyan értelmezze ezeket a koordinátákat, például ismételje a textúrát vagy vágja le azt. A rendszer a vertexekhez rendelt textúra koordinátákat interpolálja közöttük, így tudja meghatározni, hogy melyik fragmentumhoz melyik texel tartozik, ha egyáltalán tartozik hozzá. Az V/1-es ábra szemlélteti ezt a folyamatot.

Egy standard OpenGL-es textúrázás a következő lépésekből áll: engedélyezzük a textúrakezelést (glEnable(GL_TEXTURE_2D)), létrehozzuk a textúrát (glTexImage2D), beállítjuk a textúra szűrőket (glTexParameter), a mipmappingot (gluBuild2DMipmaps), a textúra objektumot létrehozzuk (glGenTextures), használjuk a textúra objektumot (glBindTexture), megadjuk a vertex adatoknál a textúrakoordinátákat, majd végezetül töröljük a textúrát ha már nincs rá szükségünk (glDeleteTextures).

Ha használunk fragment shadert, akkor a textúrázást abban kell megvalósítanunk. Tehát a fragment shaderben fogjuk eldönteni, hogy melyik fragmentumhoz melyik texelt fogjuk hozzárendelni. A következő részekben ezt fogom bemutatni hogyan is valósítható meg a textúrázás egy shader programmal, majd az eredményeket tovább fogom javítani több textúra használatával, hogy minél realisztikusabb képet kapjunk [13].

V/1. ábra: Bal fent egy két dimenziós textúra, jobb oldalt egy nyers és egy fel textúrázott model látható.
Bal lent pedig az a folyamat ahogy a modell vertexeihez a textúra koordináták segítségével megfeleltetjük a texeleket.

V/2 Textúra a terrainen

Ahhoz hogy a textúrázást a fragment shaderen belül meg tudjuk valósítani, át kell adnunk neki textúrát és a textúrakoordinátákat, hiszen ezekből az adatokból tudjuk meghatározni, hogy melyik fragmentumra melyik texelt kell ráfestenünk. Az előző részekben már volt szó arról, hogy a shader programunknak át tudunk adatokat adni, és a fragment illetve vertex shaderek között is tudunk adatokat továbbítani, de a mikéntről még nem volt szó. A shaderekben használt változókat minősíthetjük az *Uniform* és a *Varying* kulcsszóval. Az *Uniform*mal minősített változók értékét a programban adhatjuk meg, tehát a textúrát uniform változóként fogjuk kezelni a shader programban. A textúra koordinátákat a vertex shaderben a *gl_MultiTexCoord0* beépített változóban fogjuk megkapni az OpenGL-ben megadott vertexekhez tartozó textúra koordinátákat. Majd ezt a vertex shaderben továbbadjuk a gl_TexCoord beépített varying minősítésű változónak, amit a fragment shaderből el tudunk érni és a fragment shaderben már az interpolált értékeket fogjuk megkapni. Összefoglalva: az *Uniform*mal minősített változók arra szolgálnak, hogy az OpenGL és a shader program kommunikáljanak, a *varying*al minősített változók pedig arra szolgálnak hogy a vertex shader és a fragment shader kommunikálhasson.

Továbbá szükséges egy osztály, amely a képfájlt beolvassa és létrehozza a megfelelő textúrákat. Az interneten rengeteg segédanyag vagy akár kész osztály áll a rendelkezésünkre e téren. Én a sulaco[5] textúra betöltőjét használtam fel, mert nem csak egyszerű bmp képeket akarunk betölteni, hanem a későbbiek során szükségünk lesz átlátszó tga képekre is [14]. A textúrakezelő osztályunk a Textures.pas fájlban található. Ez a textúrakezelő képes betölteni jpg, bmp és átlátszó tga fájlokat, majd létre is hozza belőle az OpenGL-es textúra objektumokat.

Ezek után nézzük a fejezet példaprogramját, ami egy nagyon kezdetleges program, mindössze egy textúrát fogunk a terrainünkre ráfesteni shaderből. Első lépésként az előző fejezetbeli programunkat módosítani kell, hogy ne csak a vertex, normál és szín koordinátákat határozzuk meg, hanem a megfelelő textúra koordinátákat is. A *calcTerrainCoords* metódust kell

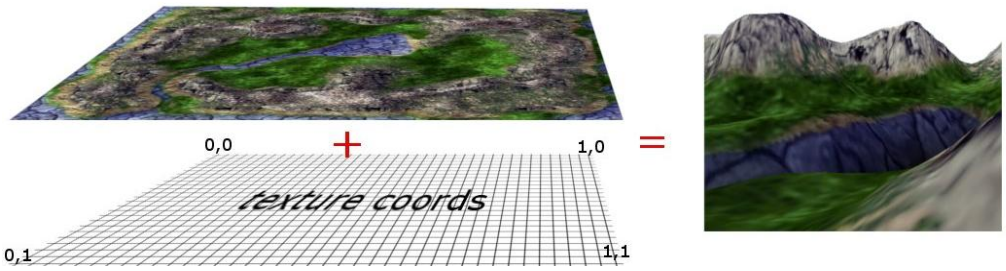
5 Sulaco delphi opengl weboldal: http://www.sulaco.co.za/opengl.htm

kiegészítenünk az V/2-es ábrán látható aláhúzott kódrészlettel. A *TerrainTexCoord* egy *TtexCoordArray* tömb típusú változó, ahol a *TtexCoordArray* egy két elemű rekord tömb. Ebben fogjuk tárolni a vertexekhez tartozó textúra koordinátákat. Az V/2-es ábrán látható, hogy ezt a tömböt milyen módon töltjük fel. Ha .s és .t értéke 0 lenne, akkor a textúra koordináták a textúra bal alsó sarkára mutatnának, ha .s és .t értéke 1 lenne, akkor a textúra jobb felső sarkára mutatnának. A köztes értékek a textúra köztes értékeire mutatnak. Nekünk most az a célunk, hogy a textúrát a terrainre egy az egyben ráfeszítsük, tehát a legelső vertexhez tartozó textúra koordináta 0 lesz, a legutolsó pedig 1, a köztes értékek, pedig annak arányában változnak, hogy melyik vertexről van szó. Tulajdonképpen egy interpolációként is felfoghatjuk 0-1 között, első vertex és utolsó vertex között. Ezért szerepel x/size érték; x a kurrens vertex x világkoordinátája, ami 0 és size érték között változik. Ezt elosztva size-al megkapjuk a kívánt interpolációt. A t értékénél a mínusz előtag egy 90 fokos forgatást ír le, mert így illeszkedik a textúra a terrainre.

```
...
while yi < n do begin
   x := round((size)/n *(xi+1));
   z := round((size)/n *(yi+1));

   //TEXTURE COORD'S
   TerrainTexCoord[xi][yi].s := x / size;
   TerrainTexCoord[xi][yi].t := -z / size;

   //COLOR
   TerrainColors[xi][yi].x:=heightMap[round(xx),round(yy)]/255;
```

V/2 ábra: Textúra koordináták megadása.

A teljes folyamat az V/3-as ábrán látható: a vertexekhez tartozó textúra koordinátákból és a textúra objektumból előállított terrain.



V/3. ábra: Lent a textúra koordináták 0 és 1 közé eső értékek a vertex (x,z) helyzetének függvényében. Fent maga a textúra. Jobbra az előállított terrain.

További módosítások is szükségesek a kódban, hogy a kívánt eredményt megkapjuk. Nem elég a koordinátákat meghatározni, hozzá is kell azt rendelni a vertexekhez rajzoláskor. Tehát a *drawTerrain* metódusban nem csak a vertex, normál koordinátákat adjuk meg, hanem a kiszámolt textúra koordinátákat is:

```
glMultiTexCoord2fvARB(GL_TEXTURE0_ARB, @TerrainTexCoord[x,y]);
```

A textúrakoordináták megadására az OpenGL *glMultiTexCoord2fvARB* metódusát fogjuk használni, ezzel a metódussal több textúra koordinátát is hozzá tudunk majd egy vertexhez rendelni, ami a későbbiekben szükséges lesz. E metódus első paramétere azt adja meg, hogy hányadik textúra koordinátát kívánjuk megadni. Mi csak egy textúra koordinátát használunk most egy vertexre, ezért az értéke *GL_TEXTURE0_ARB* lesz. A második paramétere pedig maga a koordinátapáros.

Szükségünk lesz még továbbá két változóra:

```
texture   : GLUint; // TEXTURES
stexture  : GLUint; // SHADER TEXTURE
```

Ezekből az első változóba fogjuk betölteni a textúrát. Ez lesz a textúra objektum. Az *stexture* pedig a shader-beli megfelelője lesz. A beforeStart metódusban betöltjük és létrehozzuk a textúrát:

```
LoadTexture('texture.bmp',texture,false);
```

Majd a shaderbeli textúra változónak helyet foglalunk:

```
stexture := glGetUniformLocationARB( Shader, 'texture' );
```

Ezek után már csak a render metódusban kell beállítani, hogy a shader programot és a textúrát használni is szeretnénk V/4-es ábra.

```
  glUseProgramObjectARB( Shader );
  glUniform1iARB( stexture, 0);
  glActiveTexture(GL_TEXTURE0);
  glBindTexture(GL_TEXTURE_2D, texture);
  drawTerrain;
  glUseProgramObjectARB( 0 );
```

V/4 ábra: Shader program és a textúra aktiválása.

Most nézzük meg a programhoz tartozó shader programokat V/5-ös ábra:

```
// Vertex shader
void main()
{
    gl_TexCoord[0] = gl_MultiTexCoord0;

    gl_FrontColor = gl_Color;
    gl_Position   = gl_ModelViewProjectionMatrix * gl_Vertex;
}

// Fragment shader
uniform sampler2D texture;
void main()
{
    vec4 textureColor = texture2D(texture, gl_TexCoord[0].st);
    gl_FragColor = textureColor*gl_Color;
}
```

V/5 ábra: Shader program és a textúra aktiválása.

A vertex shaderben a

```
gl_TexCoord[0] = gl_MultiTexCoord0;
```

sor fogja továbbadni a *gl_MultiTexCoord0* változóban lévő vertexekhez tartozó textúra koordinátákat a *gl_TexCoord* változónak, aminek az értékét a fragment shaderben már interpolálva fogjuk megkapni.

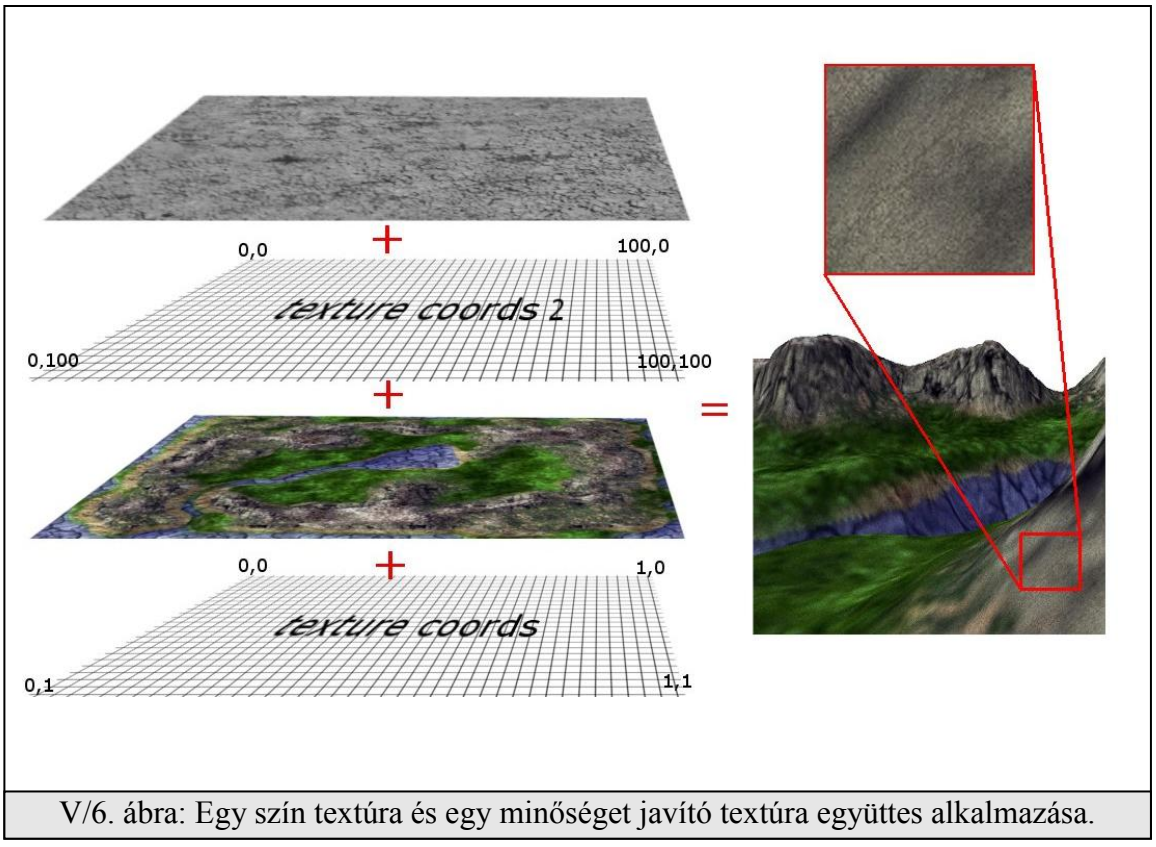
A fragment shaderben az *Uniform* minősítésű *sampler2D* típusú *texture* nevű változóval tudjuk elérni a már OpenGL-ben betöltött textúránkat. Felveszünk egy új változót a textureColor-t, ez egy négy elemű vektor, ebben fogjuk tárolni a fragmentumhoz tartozó texel színintenzitásokat. Ezt a számítást egy beépített metódus a *texture2D(sampler2D, vec2)* fogja elvégezni nekünk. Paraméterként meg kell adnunk a textúrát és a textúra koordinátát, visszatérési értékként pedig megkapjuk a texel szín értékeit. A gl_FragColor-nak ezt az értéket fogjuk átadni, de közben beszorozzuk a gl_Color értékkel is, hogy vertexekhez rendelt színeket is belevegyük a számításba (a terrain alsóbb részei sötétebbek lesznek). A IX/3-as ábra felső képén látható az eredmény.

Ennek a módszernek az előnye, hogy viszonylag könnyen implementálható, könnyen érthető, de igazából a mai világban ezek az eredmények nagyon kevesek ahhoz, hogy használhassuk tényleges rendszerekben őket. Látszik a képen, hogy ha közel megyünk egy terrain részhez, akkor a ráfeszített textúra mennyire homályos. A textúra 512*512-es, lehet javítani az eredményen ha nagyobb textúrát használunk, de rá fogunk jönni, hogy még az óriási textúrák méretei sem lesznek elégségesek, ráadásul rengeteg memóriát igényelnek. A módszeren tehát javítanunk kell, hogy jobb legyen a minőség, azonban a memória használat mégse legyen túl nagy. A következő részben erről lesz szó.

V/3 Két textúra vegyítése a terrainen

A részletesebb képi hatásért a következőt fogjuk tenni. Használni fogunk még egy textúrát (detail), amivel a minőséget feljavítjuk. Ez egy szürkeárnyalatos textúra lesz, repedéseket hasadásokat ábrázol. Ezt a textúrát pluszban fogjuk ráfesteni a terrainre, de nem ugyanazokkal a

textúra koordinátákkal, hiszen akkor ezt a detail textúrát is úgy nyújtaná el a rendszer mint az eredeti textúrát, és akkor egy repedés majdnem akkora lenne mint egy domb. A textúra koordinátákat úgy fogjuk megadni, hogy a rendszer a detail textúrát csempézve fesse rá a terrainre. Tehát mintha többször egymás mellé rakva festenénk ki. Ezáltal a detail textúra hiába kicsi, mégis többszöri alkalmazásával nagyobb minőséget fog eredményezni. A folyamat és az eredménye az V/6-os ábrán látható.



V/6. ábra: Egy szín textúra és egy minőséget javító textúra együttes alkalmazása.

Tehát szükségünk lesz újabb textúra koordinátákra. Azt szeretnénk elérni, hogy a háromszögháló minden egyes négyszögére kirajzoljuk a detail textúrát. Ehez nem szükséges újabb tömböt használnunk ahová előre kiszámoljuk ezeket az értékeket, egyszerűen a kirajzoláskor a drawTerrain metódusba beírjuk a koordinátákat:

```
glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 0, 1);
glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 1, 1);
glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 0, 0);
glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 1, 0);
```

Minden egyes négyszög bal alsó sarkához a (0,0) textúra koordinátát a jobb felsőhöz pedig az (1,1) koordinátákat rendeljük. Ezzel a kívánt eredményt kapjuk. Természetesen a beforeStart metódusban beolvassuk az új textúrát és létrehozzuk belőle a textúra objektumot, majd helyfoglalás után átadjuk a shader-beli változónak. Ezek után tekintsük a shaderek kódjait (V/7-es ábra). Ez a shader nem fog mást tenni mint a két textúrát összevegyíti. A vertex shaderben az

```
// Vertex shader
void main()
{
   gl_TexCoord[0] = gl_MultiTexCoord0;
   gl_TexCoord[1] = gl_MultiTexCoord1;

   gl_FrontColor = gl_Color;
   gl_Position   = gl_ModelViewProjectionMatrix * gl_Vertex;
}


// Fragment shader
uniform sampler2D texture;
uniform sampler2D detail;

void main()
{
    vec4 textureColor = texture2D(texture, gl_TexCoord[0].st);
    vec4 detailColor = texture2D(detail, gl_TexCoord[1].st);

    gl_FragColor = detailColor*textureColor*gl_Color*2;
}
```

V/7 ábra: Két textúra vegyítésének fragment és vertex shadere.

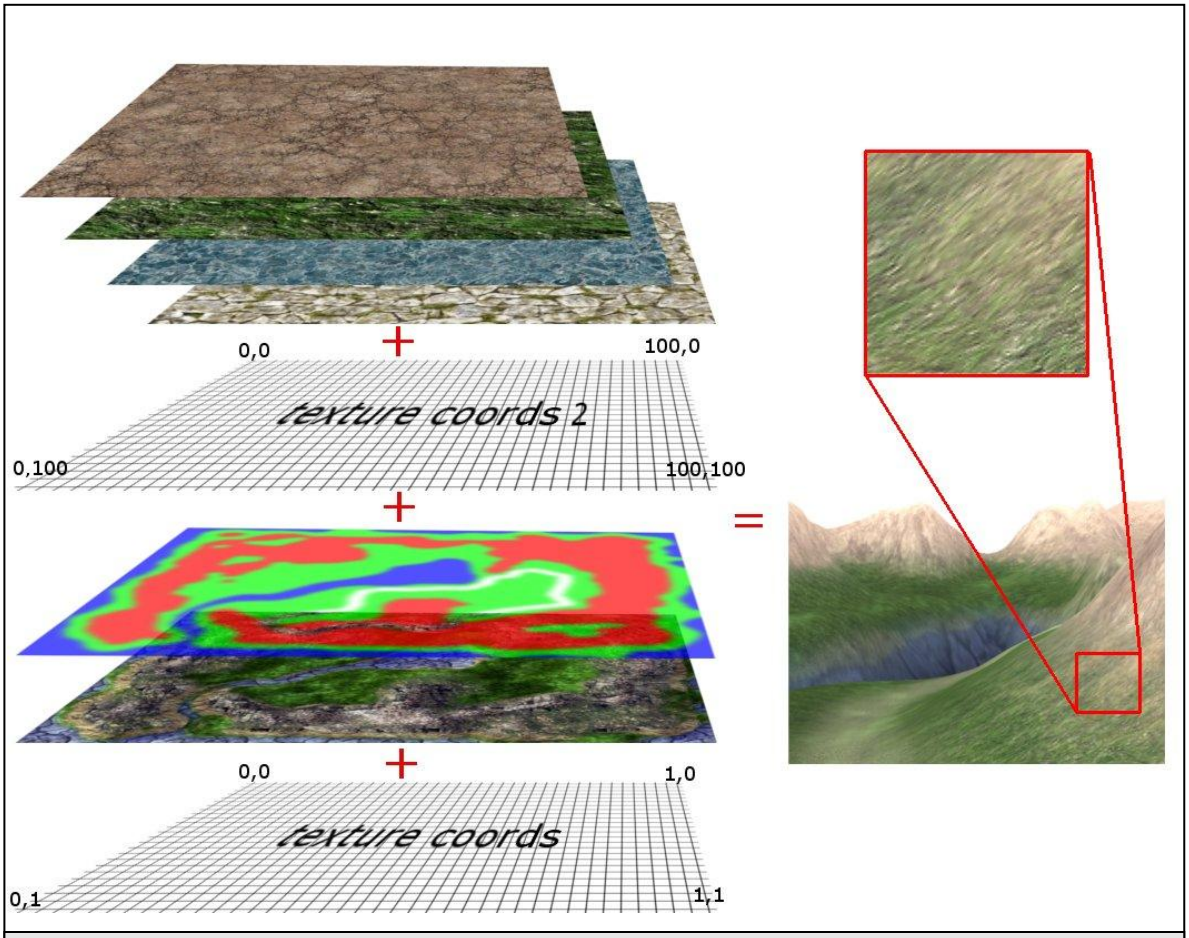
új textúra-koordinátákat (gl_MultiTexCoord1) továbbadjuk a fragment shadernek. Más változtatás itt nem szükséges az előző példához képest. A fragment shaderben az új textúra változót definiáljuk (uniform sampler2D detail), majd a kód törzsében meghatározzuk a

detailColor változó értékét (mintavételezés a detail textúrából a második textúra koordináták segítségével). Majd a végeredmény a kettő mintavételezés és a gl_Color szorzata lesz. Én még beszoroztam kettővel az eredményt, hogy picit világosabbak legyenek a fragmentumok. Megjegyzem, hogy van egy beépített metódus (vec4 mix(color1,color2,arány)), amely két szín összemixelését végzi el a megadott arányban, de ezzal a módszerrel a kép szürkés lett volna a detail textúra miatt, így ezt nem használtam.

Az elért eredmény a IX/3-as ábra közepén látható. A terraint közelebbről megvizsgálva láthatjuk a detail textúrát rajta, ami sokkal jobb képi hatást eredményez az előző módszerhez képest. Előnye, hogy ez sem egy bonyolult megoldás és mégis elég jó eredményt ad, továbbá a hardvert sem terheli nagyon meg. Sok játékban alkalmazták és alkalmazzák ezt a technikát, ahol nem szükséges a nagyon nagy részletezettség, de manapság már ez is kevés a HD monitorok és az új grafikai kártyák sokkal többre képesek ennél. A következő részben újabb javítást nézünk meg.

V/4 Négy textúra vegyítése a terrainen

Az újabb javítás az lesz, hogy nem egy minőségi textúrát hanem négyet fogunk alkalmazni és ezek nem szürkeárnyalatos képek lesznek, hanem színes képek és nem csak repedéseket fognak tartalmazni, hanem mindegyik egy külön anyagtípust (fű, száraz talaj, kavics, stb...). Lesz továbbá egy textúrát meghatározó minta (egy textúraleíró), ami azt fogja meghatározni, hogy melyik minőségi textúra hová kerüljön a terrainen és hány százalékban kell ráfestenünk őket. Ezt a textúraleírót az eredeti textúrakoordinátákkal fogjuk felhasználni, tehát ha megjelenítenénk a terrainen, akkor azon elnyújtva lenne, nem pedig csempézve. Ezt a textúrát nem fogjuk megjeleníteni, csak felhasználjuk a vegyítés számolásához a fragment shaderben. Az V/8-as ábrán grafikus szemléltetéssel látható ez a folyamat.



V/8. ábra: Négy minőségi textúra egy textúra leíróval és az eredeti textúra vegyítése.

Nem sok változtatás szükséges ahhoz, hogy ezt megvalósítsuk a fő programba. Az új textúrákat betöltjük és átadjuk a shader programnak, úgy ahogy az előző részekben csináltuk. Új textúra koordinátákra sincs szükség, hiszen most is kettőt fogunk csak használni. Egyet a nyújtott textúrázásra, egyet pedig a csempézett textúrázásra.

Érdemi változás a fragment shaderben van. V/9-es ábra mutatja azt a fragment shader kódot, ami ezt a vegyítést megvalósítja.

Hat textúrát kap meg a shader, ebből a *tex1-tex4* a négy darab minőségi textúrát

tartalmazza, a *texture* lesz az eredeti, a *mask* pedig a vegyítésért felelős textúra. Az utóbbi kettő szín értékeit a gl_TexCoord[0], a minőségi textúrákat pedig gl_TexCoord[1] textúra koordinátákkal számoljuk (nyújtott, csempézett). Felveszünk három új változót, amivel a kívánt módon állíthatjuk a végleges színintenzitást, fehérséget, sötétséget stb... Akár ezeket a változókat az eredeti programban felhasználva, módosítva és átadva a shadernek, elérhetjük a napszakok színvilágainak szimulálását.

```
uniform sampler2D texture;
uniform sampler2D mask;

uniform sampler2D tex1;
uniform sampler2D tex2;
uniform sampler2D tex3;
uniform sampler2D tex4;

void main()
{
 vec4 textureColor = texture2D(texture, gl_TexCoord[0].st);
 vec4 maskColor = texture2D(mask, gl_TexCoord[0].st);

 vec4 tex1Color = texture2D(tex1, gl_TexCoord[1].st);
 vec4 tex2Color = texture2D(tex2, gl_TexCoord[1].st);
 vec4 tex3Color = texture2D(tex3, gl_TexCoord[1].st);
 vec4 tex4Color = texture2D(tex4, gl_TexCoord[1].st);

 float gamma = 1.5;
 float ambient = 0.1;
 float texIntensity = 0.225;

 gl_FragColor = ((
  (tex1Color*maskColor.r + tex2Color*maskColor.g + tex3Color*maskColor.b)*
  maskColor.a + tex4Color * (1-maskColor.a + vec4(0,0,0,1) + ambient)*
  (gl_Color+0.08) + textureColor * texIntensity) * gamma;
}
```

V/9 Ábra: Négy minőségi és egy textúra vegyítésének fragment shader kódja.

A fragmentum színkeverését a következő módon hajtjuk végre: a textúra leíró textúra (mask) egy alfa csatornás textúra lesz, aminek minden egyes összetevője (red, green, blue, alfa) egy-egy

intenzitásértéket fog jelölni, hogy az adott részen milyen intenzitással vegyük a tex1-tex4 textúrákból a színeket.

Tehát ha például a *tex1Color*-t megszorzom a *maskColor.r*-val, akkor a tex1 színnek a maskColor.r által leírt intenzitású tex1 színt fogjuk kapni. A tex2 * maskColor.g a második textúra megfelelő intenzitású színe lesz és így tovább.

Ezeket a színértékeket összeadjuk és beszorozzuk a textúraleíró alfa csatornájával. Ez azért kell, hogy ahol az alfa csatorna értéke 0, ott ezeket a színeket ne használjuk a keverésben.

Majd ehhez az értékhez hozzáadjuk a tex4Color*(1-alfa csatorna) szorzatát. Ez azért kell, hogy ahol az alfa csatorna 0, ott a negyedik textúrát fessük ki (annak az intenzitását). A vec4(0,0,0,1) azért kell a számításba, hogy látszódjon is valami, az alfa csatorna értékét 1-re állítja azon a helyen, ahol a 4-edik textúrát kell kirajzolni. Az ambient csak egy színkorrekciós változó.

Ezután hozzávegyítjük a vertex színeket (gl_Color), továbbá az eredeti textúrát is vegyítjük a texIntensity mértékben.

Folytatásként az egészet beszorozzuk egy gamma értékkel, ami az egész fényességét fogja szolgáltatni nekünk.

Az eredményt a IX/3-as ábra alsó képe mutatja. Ez a módszer nagyon szép eredményt ad és még ez sem túl bonyolult vagy megterhelő a mai hardverek számára.

Megjegyzés: Itt is használható a beépített mix metódus. Az itt leírtakat nem törvényszerűségként kell kezelni. Pont az a lényeg, hogy itt bárki a saját ízlése szerint eljátszathat az értékekkel, annak érdekében, hogy a kívánt eredményt elérje.

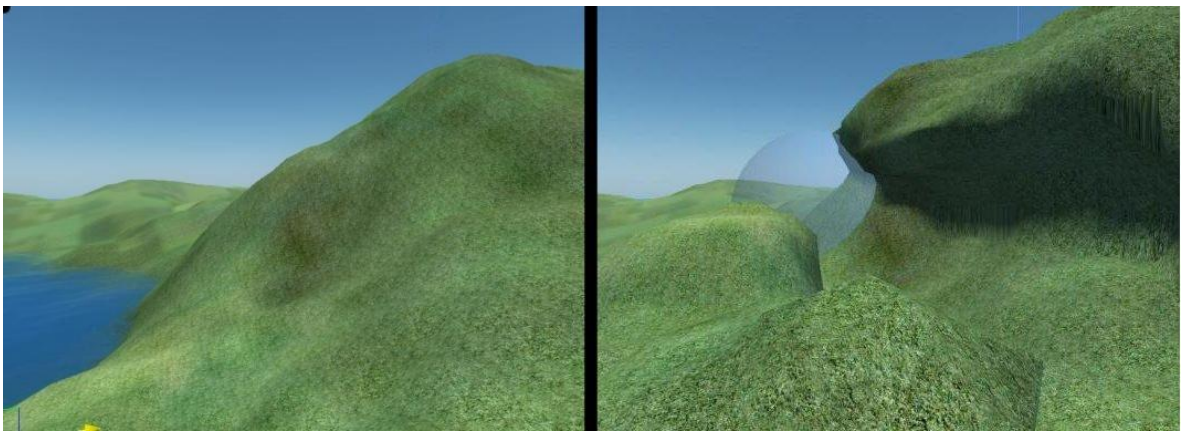
Konklúzió

A dolgozat célja a shader technológia valamint egy pár grafikai trükk bemutatása volt példákon keresztül. A dolgozatban szereplő algoritmusok és shader kódok számítógépes játékprogramok fejlesztésénél adhat segítséget, de bevezető lehet mindazon emberek számára akik még csak most ismerkednek e technológiákkal. A dokumentum egy terrain elkészítésén keresztül vezeti végig az olvasót. A kezdetektől (a magasságtérkép elkészítésétől) az ütközésdetektáláson át a végső feltextúrázási folyamaton mind végighalad. Közben olyan tanácsokat fogalmazva meg amelyek még látványosabb megjelenítést tesznek lehetővé és processzoridőt spórolnak meg mindeközben. Mivel minden fejezethez tartozik legalább egy példaprogram, az olvasó könnyen kipróbálhatja a leírtakat és a kód módosításával még jobban megértheti azok működését.

A dolgozat célja – grafikai betekintés a shaderek nyelvébe – úgy érzem sikeres volt. Valamint a példák és a dolgozatban szereplő tippek hasznos információkkal, egy jó kiinduló ponttal láthatják el a téma iránt érdeklődőt. Összefoglalva az eredményeket a célkitűzéseimet maradéktalanul megvalósítottam. A IX/3-as ábra alján megtekinthető a végeredménye az ismertetett módszernek.

Meg kell azonban említenem, hogy a mai nagy játékfejlesztők egy újra felfedezett terrain típust (voxel terraint) használnak, ami lehetőséget ad olyan terrainek elkészítésére, amikben lehetnek hidak, alagutak. A mi esetünkben a magasságtérképekkel ez nem valósítható meg, természetesen továbbfejleszthető úgy, hogy tudja ezeket is kezelni. A következő oldalon látható egy kép a két terrain típusról (VI/1-es ábra).

A színezési technikákon is lehet még javítani. Kitalálhatunk új módszereket amivel akár még több textúrát vagyunk képesek megjeleníteni egyetlen terrainen. Ha pedig még gyorsabb programot szeretnénk, akkor implementálhatunk egy LOD (Level Of Detail) vagy ROAM (Real-time Optimally Adapting Meshes) terraineket. Segítségükkel a távolban lévő terrain részek nem olyan részletesek, kevesebb háromszögből állnak, így még nagyobb sebességet érhetünk el. Ezeket a trükköket azért nem vettem bele a dolgozatba, mivel nagyon sok módszer van és mindegyik módszerről egy hasonló terjedelmű dokumentumot lehetne írni, viszont akit érdekel a téma, rengeteg leírást találhat az interneten vagy e dolgozat irodalomjegyzékéből is csemegézhet.



VI/1. ábra: Bal oldalt normál terrain, jobb oldalt az új voxel terrain.
Forrás: http://doc.crymod.com/SandboxManual/TerrainVoxelPainter.html

A shader technológia nagy előrelépés a számítógépes grafikában, szinte bármit megvalósíthatunk velük, csak a képzeletünk és a maximális utasításszám szab határt.

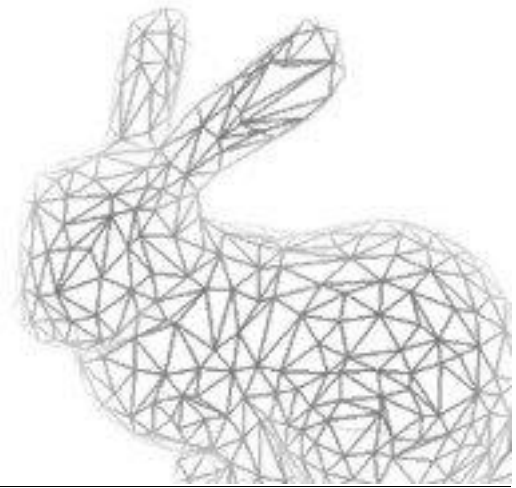
Köszönetnyilvánítás

Ezúton szeretném megköszönni Dr. Tornai Róbertnek, a témavezetőmnek, hogy segítségemre volt a dolgozat elkészítésében, rugalmas hozzáállását a munkámmal kapcsolatban; valamint, hogy felnyitotta a szemem, érdeklődésem a grafika és a programozás terén.

Köszönöm továbbá a családomnak, szüleimnek és nagyszüleimnek a támogatást, és a megértő hozzáállást az egyetemen töltött éveim alatt.

Hálával tartozom Giovanni Brambillának, hogy a program egyes részeit beintegrálva az ő oktató programjába mindig új kihívások elé állítva nem hagyta a lelkesedésemet alábbhagyni.

Végezetül szeretnék köszönetet mondani mindazon programozónak, akik a Gimp, OpenOffice és a Notepad++ ingyenes programok elkészítésében részt vettek, hiszen ezek a programok nagyban megkönnyítették a dolgozat elkészítését.



Irodalomjegyzék

II. fejezet

[1] http://www.lighthouse3d.com/opengl/terrain

[2] Juhász Imre
OpenGL
Debrecen, 2003
http://iam035.inf.unideb.hu/mobidiak/filedownload.mobi?fid=416

[3] Dave Shreiner
The OpenGL Programming Guide, Seventh Edition – The Redbook
2009

III. fejezet

[4] http://www.peroxide.dk/download/tutorials/tut10/pxdtut10.html

[5] http://softsurfer.com/Archive/algorithm_0105/algorithm_0105.htm

[6] http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=30

[7] http://superminimum.atw.hu/colldet/colldet.htm

[8] http://www.geocomputation.org/1999/082/gc_082.htm

IV fejezet:

[9] Wolfgang Engel

ShaderX3: Advanced Rendering with DirectX and OpenGL (Shaderx Series)

[10] Randi J. Rost

OpenGL Shading Language – The Orangebook

[11] http://www.lighthouse3d.com/opengl/glsl/index.php?pipeline

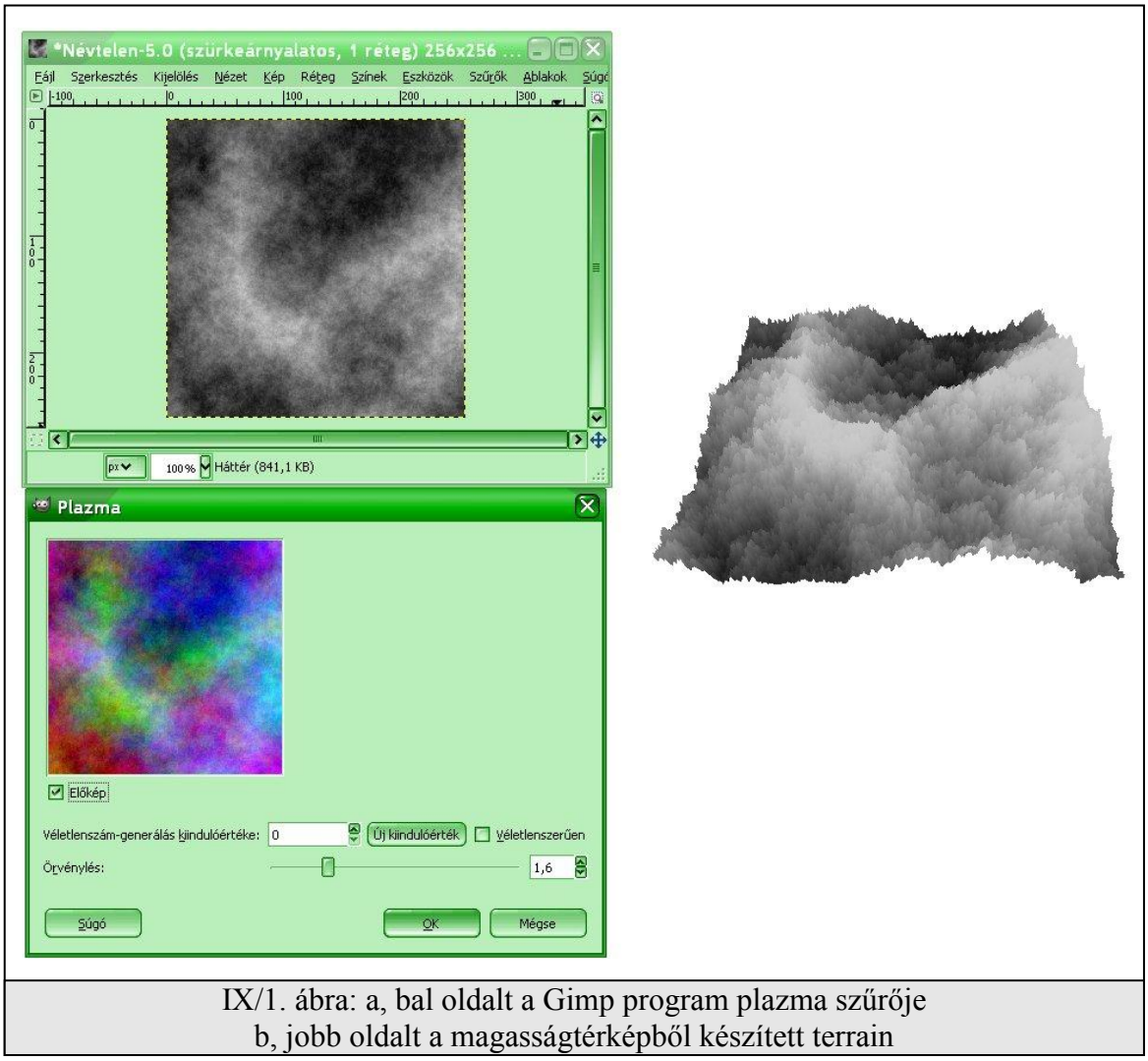
[12] http://nehe.gamedev.net/lesson.asp?index=01

V. fejezet:

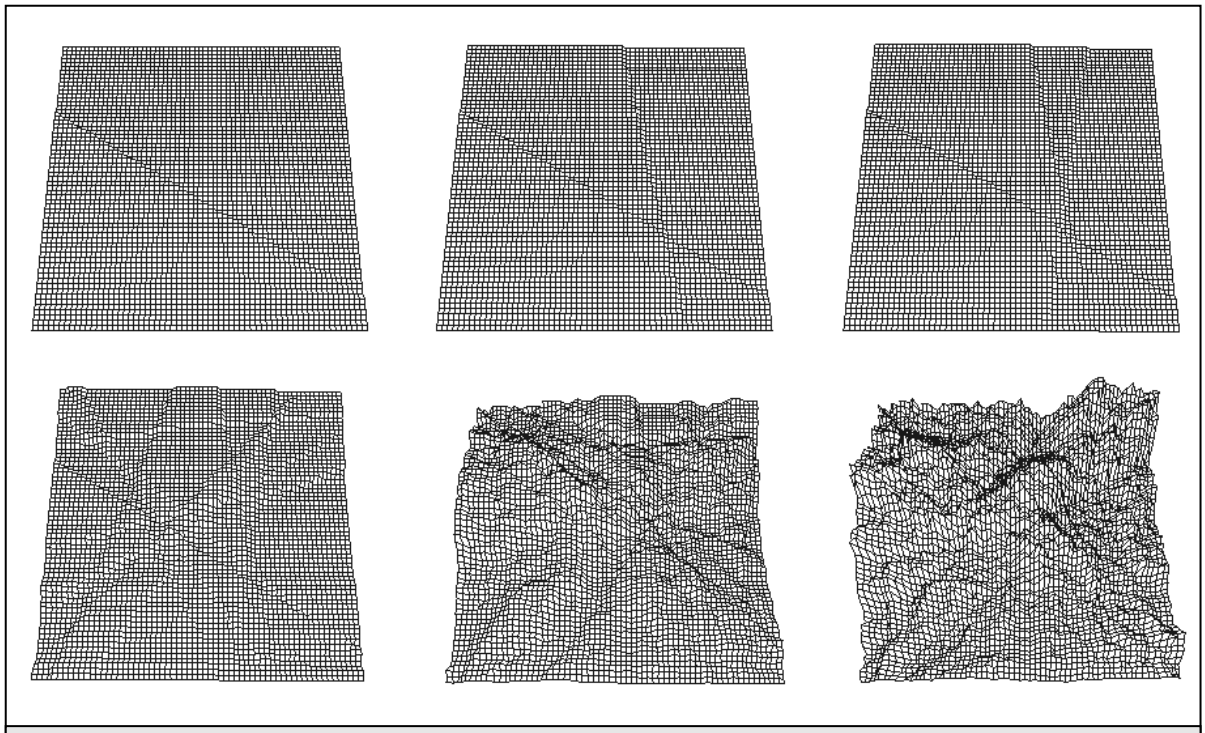
[13] http://www.lighthouse3d.com/opengl/glsl/index.php?texture

[14] http://www.sulaco.co.za/opengl_project_transparent_TGA_files_and_fog.htm

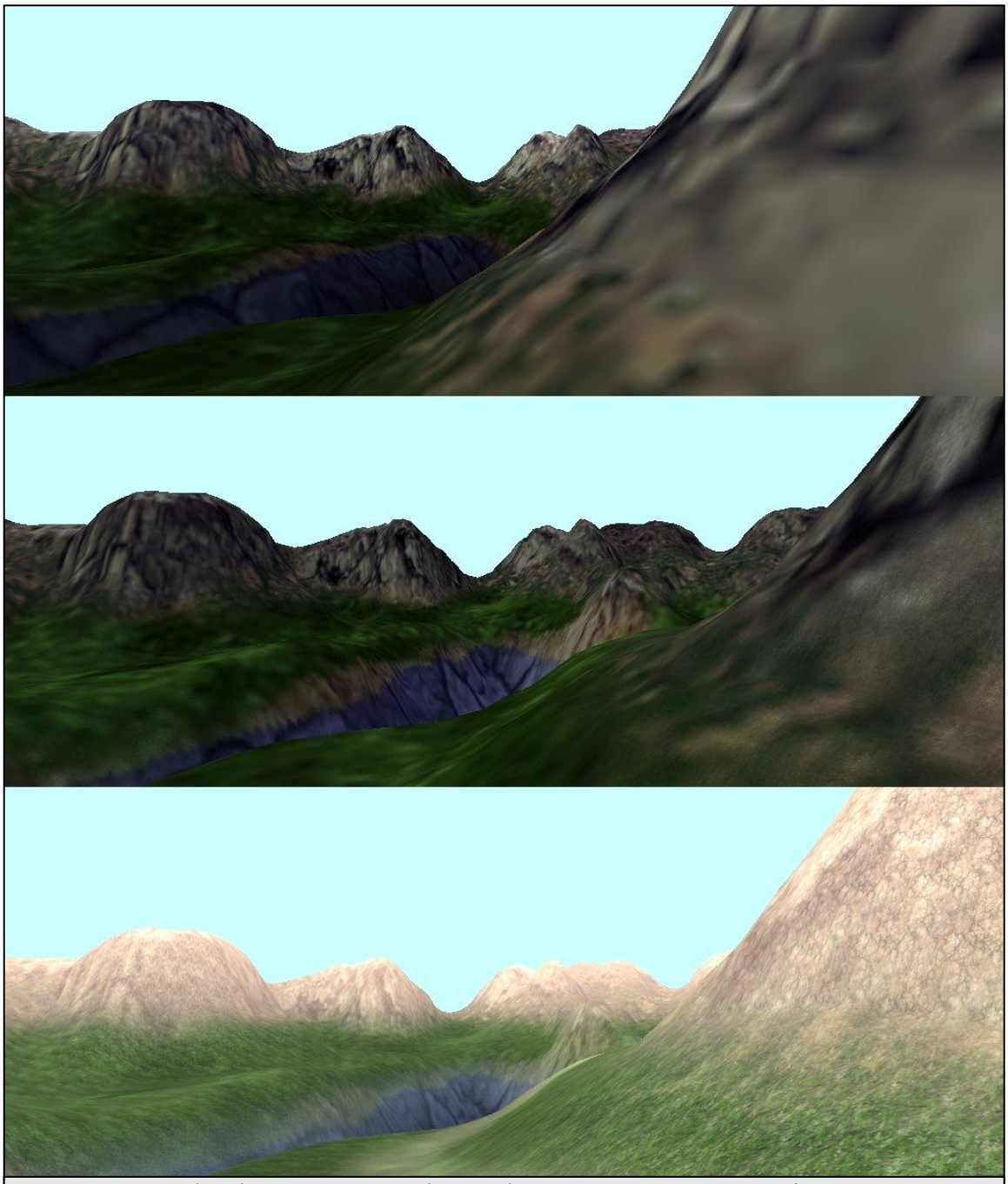
Képtár



IX/1. ábra: a, bal oldalt a Gimp program plazma szűrője
b, jobb oldalt a magasságtérképből készített terrain



IX/2. ábra: A vetődés algoritmus által generált felület, különböző iterációs lépésben.



IX/3. ábra: A végső eredmények. Fent egy textúra a terrainen.
Középen egy textúra és egy minőséget javító (detail) textúra.
Lent 4 minőségi textúra +1 textúraleíró +1 színkorrekciós textúra.