Naturmiljöer med realtidsgrafik

Sammanfattning

Vårt examensarbete handlar om datorgrafik, närmare bestämt tekniker för visualisering av utomhusmiljöer. Vi har inriktat oss på att studera, jämföra och testa olika metoder för visualisering av växtlighet med ”bildskivor”, s.k. billboards, och polygonmodeller. Verktyget för detta har varit Crystal Space, en opensource-programvara för att utveckla spel, försett med ett insticksprogram vi skapat. I detta har vi implementerat tekniker för utplacering av träd och andra växter, representerade av en billboard eller en polygonmodell, inom ett delområde av en landskapsyta, liksom för övergång mellan dessa representationssätt. I våra tester visade sig den bästa billboardtekniken vara en enkel vridbar bildskiva. Men dessutom har vi visat att en övergång mellan en sådan och en polygonmodell, som sker på längre avstånd från betraktaren, kan göras osynlig med den bländningsteknik vi implementerat.

Nature environments with real time graphics

Abstract

Our master thesis deals with computer graphics, more exactly techniques for creation of outdoor environments. We have concentrated upon studying, comparing and testing of various methods for visualizing of vegetation with pictorial boards, so-called billboards, and polygon models. The tool for this has been Crystal Space, an open source program for development of games, supplied with a plugin created by us. Into that plugin we have implemented techniques for setting out trees and other types of plants, represented by billboards or polygon models, inside an area on a landscape surface. In addition we have implemented a technique for transition between a polygon model and a billboard in form of a turning picture board. Our tests have demonstrated a turning picture board to be the best among the billboard types and also that transition far away from an observer, between a polygon model and a turning picture board, can be made invisible with the blending transition technique we have implemented.

Innehållsförteckning

Bilagor

1. Introduktion

Denna rapport omfattar ett examensarbete om 20 poäng, vid institutionen för datavetetenskap vid Lunds Universitet. Vår handledare har varit Lennart Ohlsson, universitetslektor vid LTH i Lund.

Området datorgrafik har under en längre tid, spelat en stor roll inom datavetenskapen. Att det har kommit att spela en större roll nu än för några år sedan kan sammankopplas med det faktum att en processor för hantering av grafisk representation (GPU) numera är lika kraftfull som den vanliga CPU:n. Med kraftfull avses här att den klarar göra en stor mängd beräkningar per tidsenhet. En modern persondator består därför i stort sett av ett två-processorsystem, där den ena (GPU) tar hand om all grafikhantering, medan den andra (CPU) tar hand om övrig hantering. Detta gör att en modern dator klarar visa mycket samtidigt med större detaljrikedom än tidigare. Detta ser man exempel på inom många olika områden, inte minst inom spelindustrin.

Detta har gjort det möjligt att visa fotorealistiska bilder. Man kan numera använda det inbyggda stöd, som finns på grafikkorten, för att beräkna skuggor och avancerad ljussättning. Man brukar som ett mått ange hur många trianglar ett grafikkort klarar att rendera i sekunden. Här rör det sig på ett modernt grafikkort om flera miljoner trianglar i sekunden, vilket gör att i en upplösning av 1024 x 768 pixlar (bildelement) kan bildfrekvensen bli bortemot åttio bilder per sekund i ett modernt spel. Värdet på bildfrekvensen skall här ses väldigt relativt i förhållande till detaljnivån på scenen som visas, grafikkortets prestanda och inte minst datorns prestanda, som spelar en viktig roll i själva helheten.

Men den ökade möjligheten till tredimensionell fotorealism används inte bara i spel utan även bl.a. inom sjukvården, i form av 3d-simuleringar av kroppens organ. Man kan också se tillämpningar som simulering av hur ett nytt hus kan se ut. Här kan en tänkt köpare låta en kamera röra sig runt i huset och därmed bilda sig en uppfattning om hur huset ser ut. Ett annat exempel är en landskapsarkitekt, som vill bilda sig en uppfattning om hur ett landskap skall se ut. Han kan med hjälp av en terrängsimulering placera ut sina objekt på det sätt han önskar och se resultatet direkt.

Som en sammanfattning kan man säga att om området fortsätter att växa i den takt det gjort de senaste åren kommer vi snart att se de ovan nämnda fotorealistiska miljöerna vid daglig användning av en dator.

2. Förutsättningar och syfte

Ämnet för vårt examensarbete är datorgrafik, närmare bestämt skapandet av utomhusmiljöer. Vi har därvid tagit som vår uppgift att studera, jämföra och testa olika metoder för visualisering av vegetation. Som konkret mål för arbetet har vi satt upp att göra ett insticksprogram, s.k. plugin, till spelmotorn Crystal Space. I programmet implementeras tekniker för visualisering och utplacering av vegetation i naturmiljöer.

Vår rapport angående detta arbete inleds med en teoretisk bakgrund, som syftar till att ge en överblick över de tekniker som används för visualisering av utomhusmiljöer. Därefter följer en redovisning av en undersökning vi gjort av förekommande terrängmotorer (programvara för grafisk visualisering av naturmiljöer) samt en därpå baserad presentation av ambitionsnivån för funktionerna hos vårt plugin. Sedan kommer en beskrivning av dessa och hur de implementerats samt testning av dem och resultaten därav, vilka även diskuteras. Avslutningsvis presenteras en demoapplikation, som vi sammanställt för att visa funktionaliteten hos vårt plugin.

2.1 Presentation av Crystal Space

Crystal Space är ett stort opensource projekt, med runt sexhundra personer som har registrerat sig som användare och utvecklare, världen över. Crystal Space är en fri och portabel spelmotor, skriven i C++. Att den är portabel innebär att man kan kompilera den under många olika plattformar t.ex. Unix, Linux, Macintosh eller Windows. En spelmotor kan förklaras som ett operativsystem för spel. Med detta menas att den består av många olika delar, var och en med sin egen uppgift. Exempel på dessa delar är hantering av skymda ytor, kollisionsdetektering, effekter och rendering. Hantering av skymda ytor sker i Crystal Space med hjälp av en kombination av portaler, octträd, BSP-träd och en c-buffert. För rendering har Crystal Space stöd för hårdvarurenderarsystem som OpenGL och Direct3D, men den har även en egen mjukvarurenderare. I kapitel 4.1.4 redogörs för de möjligheter Crystal Space tillhandahåller för visualisering av utomhusmiljöer.

Crystal Space har även stöd för att modellera och skapa en tredimensionell värld med hjälp av skriptfiler. En skriptfil kan beskrivas som ett recept på hur världen skall se ut. I skriptfilen finns anvisningar om vilka texturer som skall användas, vilka plugin som skall laddas och olika beskrivningar av de objekt som skall visas. Som ett exempel på hur detta ser ut kan följande utdrag från en sådan skriptfil studeras:

```
WORLD(
  TEXTURES (
      TEXTURE 'sky_up' (FILE (skybox_u.jpg))
      ...
  )
  ...
```

```
PLUGINS (
      PLUGIN ’terr’ (’crystalspace.mesh.loader.factory.terrfunc’)
      PLUGIN ’spr3dFact’ (’crystalspace.mesh.loader.factory.sprite
                           .3d’)
      ...
  )
  ...
  MESHFACT ’treeFact’ (
      PLUGIN (’spr3dFact’)
      PARAMS(
        MATERIAL (’tree’)
        FRAME ’f1’(
            V (0,0,0:0,1)     ;0
            V (0,10,0:0,0)    ;1
            V (10,10,0:1,0)   ;2
            V (10,0,0:1,1)    ;3

            V (5,0,-5:0,1)    ;4
            V (5,10,-5:0,0)   ;5
            V (5,10,5:1,0)    ;6
            V (5,0,5:1,1)     ;7
        )
        ACTION 'default' (F (f1,1000))
        TRIANGLE (0,1,2)
        TRIANGLE (0,2,3)
        TRIANGLE (0,2,1)
        TRIANGLE (0,3,2)

        TRIANGLE (4,5,6)
        TRIANGLE (4,6,7)
        TRIANGLE (4,6,5)
        TRIANGLE (4,7,6)
      )
  )
  ...
  MESHOBJ 'treeObj_1' (
      PLUGIN ('spr3d')
      PARAMS (
        FACTORY ('treeFact')
        ACTION('default')
        MIXMODE (KEYCOLOR(0,0,0))
        BASECOLOR (1,1,1)
      )
      MOVE (V (0,74,50))
      ZTEST ()
      PRIORITY ('tree')
  )
  MESHOBJ 'treeObj_2' (
      PLUGIN ('spr3d')
      PARAMS (
        FACTORY ('treeFact')
        ACTION('default')
        MIXMODE (KEYCOLOR(0,0,0))
        BASECOLOR (1,1,1)
      )
      MOVE (V (10,75,50))
      ZTEST ()
      PRIORITY ('tree')
  )
  ...
)
```

I skriptfilen på föregående sida ser man att ett grafiskt objekt i Crystal Space består av två delar. Det första man anger är en meshfactory (`MESHFACT`). I denna ingår en punktlista med tillhörande triangellista. För att sedan placera ut ett objekt skapar man ett meshobject (`MESHOBJ`). I beskrivningen av detta ingår en referens till den factory man vill använda. Man kan även i beskrivningen ange vilka egenskaper i form av grundfärg, transparens, renderingsprioritet m.m. som objektet skall ha i världen. Instruktionen `MOVE` (se skriptfil), talar om var objektet skall vara placerat i världen, angivit i x,y,z-koordinater.

För att Crystal Space sedan skall kunna visa den i skriptfilen beskrivna världen, används ett program som kan tolka och visa denna. Ett exempel på ett sådant program följer med Crystal Space och heter walktest, vilket är ett program där man kan ange en skriptfil som en parameter till programmet. Walktest är från början implementerat som ett demonstrations-program för att visa de möjligheter som finns tillgängliga med Crystal Space, men kan även användas för visualisering av en godtycklig skriptfil.

Ett annat sätt att skapa en värld med Crystal Space är att man programmerar denna i ren C++ kod. Man kodar in allting i sitt program istället för att ange det i en skriptfil. Detta görs med de i Crystal Space tillgängliga klasserna, för exempelvis en kamera eller ett meshobject. Ett exempel på hur ett meshobject kan kodas följer nedan:

```
iMeshWrapper* sprite = view->GetEngine()->CreateMeshWrapper(
                       sprite_tmpl, name, room, csVector3 (x,y,z));

iSprite3DState* spstate = SCF_QUERY_INTERFACE (sprite->GetMeshObject
                          (),iSprite3DState);

spstate->SetBaseColor(csColor(1,1,1));
spstate->SetAction ("default");
spstate->DecRef ();

sprite->SetZBufMode (CS_ZBUF_TEST);
sprite->SetRenderPriority (view->GetEngine()->GetRenderPriority
                          ("tree"));
sprite->GetMovable()->SetTransform(csYRotMatrix3((r)));
sprite->GetMovable()->UpdateMove();
```

”sprite_tmpl” ovan är en pekarvariabel till en meshfactory. ”iMeshWrapper” är motsvarigheten till uttrycket MESHOBJ i skriptfilen. Exemplet motsvarar den i skriptfilen angivna beskrivningen av ett meshobjekt.

3. Teoriavsnitt: En översikt av aktuella tekniker för terräng-generering

Terräng, i den betydelse termen användes i detta arbete, omfattar:

- En markyta utomhus med alla dess nivåskillnader, tänkt som en rent geometrisk tredimensionell yta.
- Markytans karaktär. Den kan vara ex. sten, grus, jord, sand eller vatten.
- Markytans eventuella vegetation.
- Himlen över markytan med sol och moln.

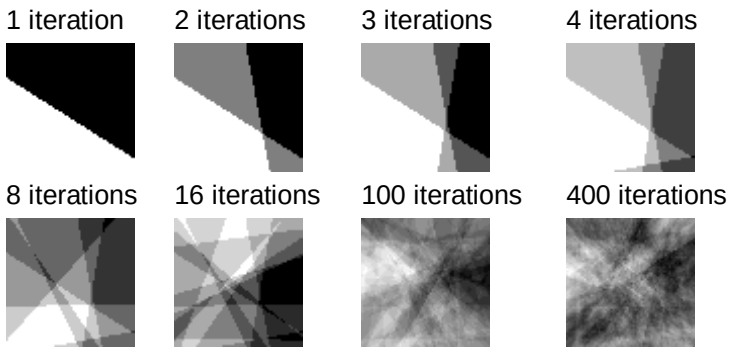
3.1 Modellering av terrängens yta

Modellen av markytans form är oftast förberäknade indata till terrängmotorer i form av en tvådimensionell höjdkarta, dvs. ett rutnät där varje hörnpunkt har ett höjdvärde. Men det förekommer även att terrängmotorer modellerar markytan helt eller delvis i realtid, dvs. när den behöver visualiseras [6].

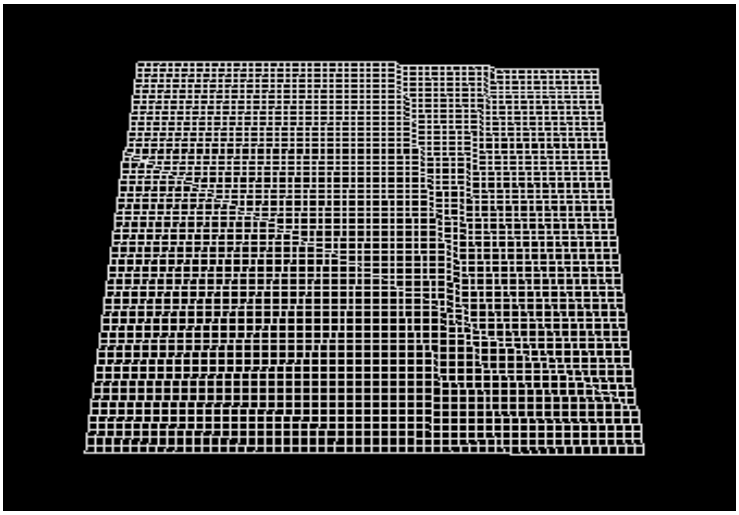
Höjdkartan är nästan alltid en gråskalebitmap, där nivåskillnader svarar mot skillnader i nyans så att ex. ljusare nyanser representerar högre nivåer än mörkare. I litteraturen föreslås flera olika metoder att beräkna höjdvärdena med hjälp av fraktaltekniker, varav några beskrives nedan.

3.1.1 Förkastningsmetoden

Rubricerade metoden benämnes i engelskspråkig litteratur ”fault formation”. Den startar med ett område där alla punkter har höjden noll. Genom detta dras slumpmassigt en linje, som delar det i två delar. Varje punkt på en av delarna, som väljs på ett bestämt sätt, ges höjden dH(0). Därefter minskas dH(0) till dH(1). Ånyo dras en linje slumpmässigt och proceduren upprepas för de områden linjen nu tudelar. Detta upprepas så länge man önskar [1], [9]. Ett exempel på algoritmens förlopp visas i figur nr. 1.



Figur nr. 1a: Exempel på algoritmens förlopp

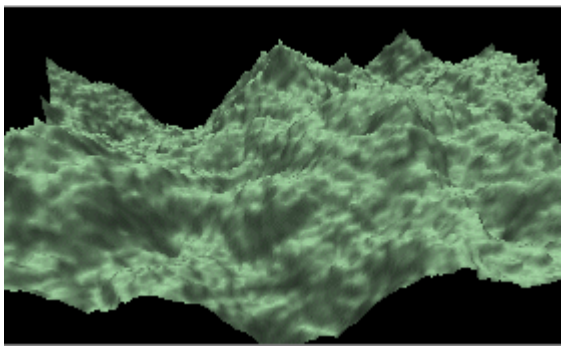


Figur nr. 1b: Efter 3 iterationer [16].

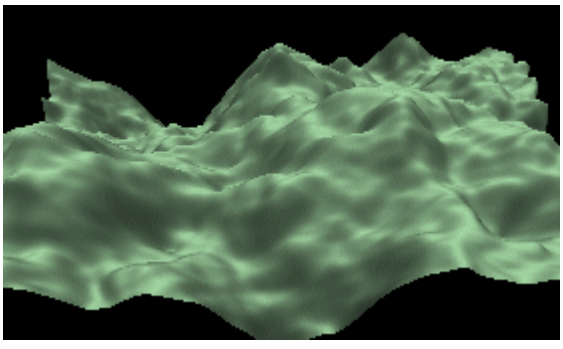
Utan vidare bearbetning skapar förkastningsmetoden, enligt ovan, onaturligt dramatiska skillnader mellan angränsande områden i höjdkartan. Även efter ett större antal iterationer blir resultatet likt en bit papper, som snittats härs och tvärs från kant till kant ett flertal gånger med ett rakblad. Detta kan dock filtreras till en diffusare och därmed mera verklighetstrogen topografi. En enkel sådan filtrering kan göras enligt formeln:

```
höjd(x,z) = höjd(x-1,z) * (1-k) + höjd(x,z) * k
```

där höjd(x,z) är höjden i punkten med koordinaterna (x,z) i xz-planet och k är en filtreringkonstant, 0 <= k <= 1. Värden på k nära 1 ger liten förändring och värden nära noll leder till att varje punkt får ett höjdvärde som är nästan lika med sina grannars. Värdet k = 0.5 rekomenderas. Formeln bör appliceras både fram och tillbaka på resultatets rad- och kolumnhöjdvärden, se figur nr. 2 [1].



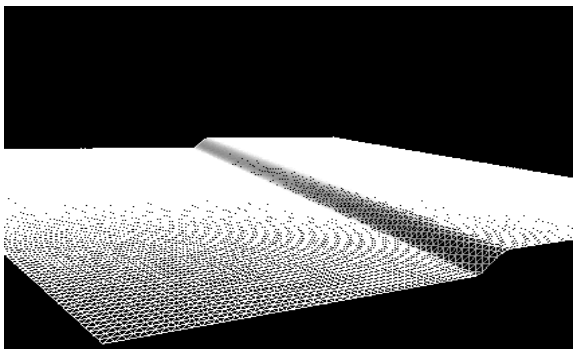
Efter 1 filtreringspass med k = 0.75



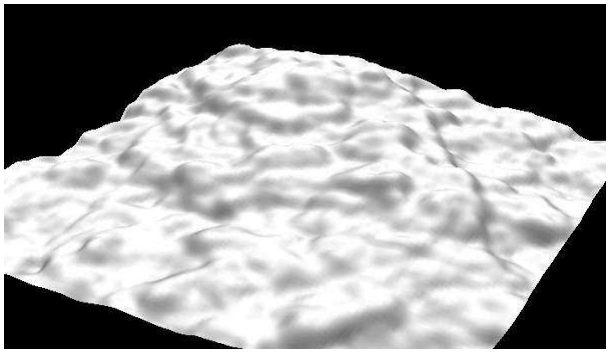
Efter 5 filtreringspass med k = 0.75

Figur nr. 2 [16]

Två intressanta variationer på denna teknik är att snittprofilen längs den räta linjen ändras från en rak vertikal linje till en graf av typen sinus- eller cosinuskurva. Detta åskådliggöres i figurerna nr. 3a och 3b [9].

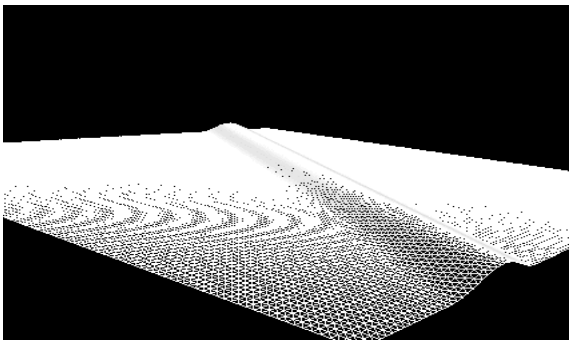


Sinusprofil

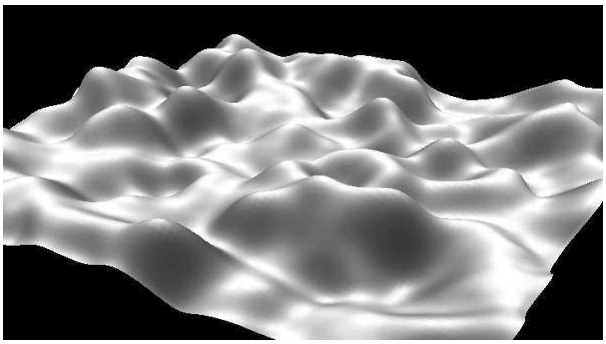


Resultat från 1000 iterationer med sinusprofil

Figur nr. 3a [16]



Cosinuskurva

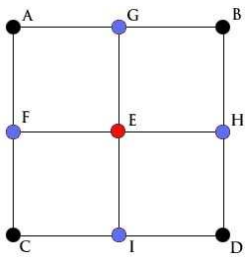


Resultat från 1000 iterationer med cosinusprofil

Figur nr. 3b [16]

3.1.2 Mittpunktsförskjutningsmetoden

En ytterligare välkänd metod är mittpunktsförskjutning. Denna startar med en fyrhörning, vars hörn har givna höjdvärden. Dess sidlängd rekommenderas vara = 2^n för något heltal n. Varje iteration i algoritmen består av två steg. Första steget, diamantsteget, är att beräkna en höjd för polygonens centrum som genomsnittet av de fyra ursprungshörnens, A, B, C, D, höjder plus en slumpfaktor rand(d) , -d<= rand(d) <= d, där |d| representerar den maximala möjliga förskjutningen i den aktuella iterationen:



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

I andra steget, fyrkantsteget, beräknas höjdvärden för mittpunkterna F, G, H, I, mellan ursprungspunkterna enligt följande :

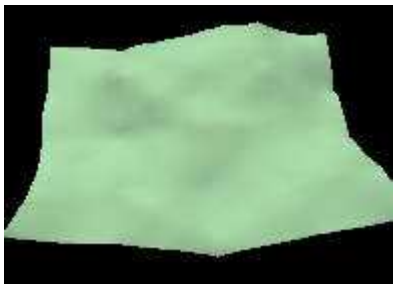
$$F = (A + C + E + E) / 4 + RAND(d), \qquad G = (A + B + E + E) / 4 + RAND(d)$$

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

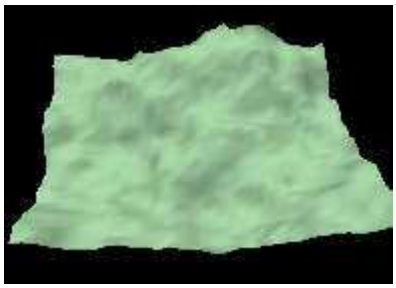
Därefter upprepas de två stegen för var och en av de fyra polygonerna AGEF, GBHE, EHDI och FEIC. För varje iteration i beräknas d(i) enligt :

$d(i) = d(i-1) * 2^{-r}$, där r inverkar på resultatets skrovlighet.

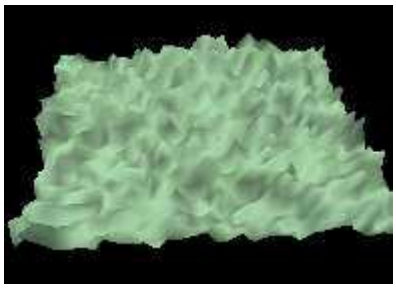
Ju högre värde på r desto större blir denna, se figur nr. 4 [1], [9].



Figur nr. 4a: r = 0.5



Figur nr. 4b: r = 1.0



Figur nr. 4b: r = 2.0

Figur nr. 4 [16]

3.1.3 Nätpunktsförskjutningsmetoden

Denna enkla metod startar i en av höjdkartans hörnpunkter (x, z), som ges ett höjdvärde i form av ett tal d(0). Därefter stegas till en annan hörnpunkt, som ges samma höjd eller kanske d(1) < d(0). Genom upprepning av detta förfarande, som innefattar möjligheten att man återkommer till en tidigare besökt punkt och ändrar dess höjd igen, fås en principiellt användbar metod att generera landskapstopografier. Stegningen kan dock utföras på många olika sätt.

Ett enkelt förfarande är följande [9]:

```
n = rand( seed ) ;  n väljes slumpmässigt bland heltalen 1 - 4.
switch( n ) {
  case 1 : x++; break;
  case 2 : x--; break;
  case 3 : z++; break;
  case 4 : z--; break;
}
```

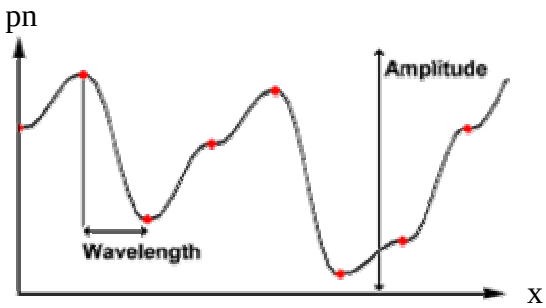
Höjdsättningen kan också beräknas på olika sätt. Ett är att välja formeln i näst föregående avsnitt:

$$d(i) = d(i-1) * 2^{-r}.$$

För att jämna av höjdkonturerna kan man låta de av punktens (x, z) åtta närmaste grannar, vilka har lägre höjd än denna, få dennas åsatta höjdvärde.

3.1.4 Perlinbrus

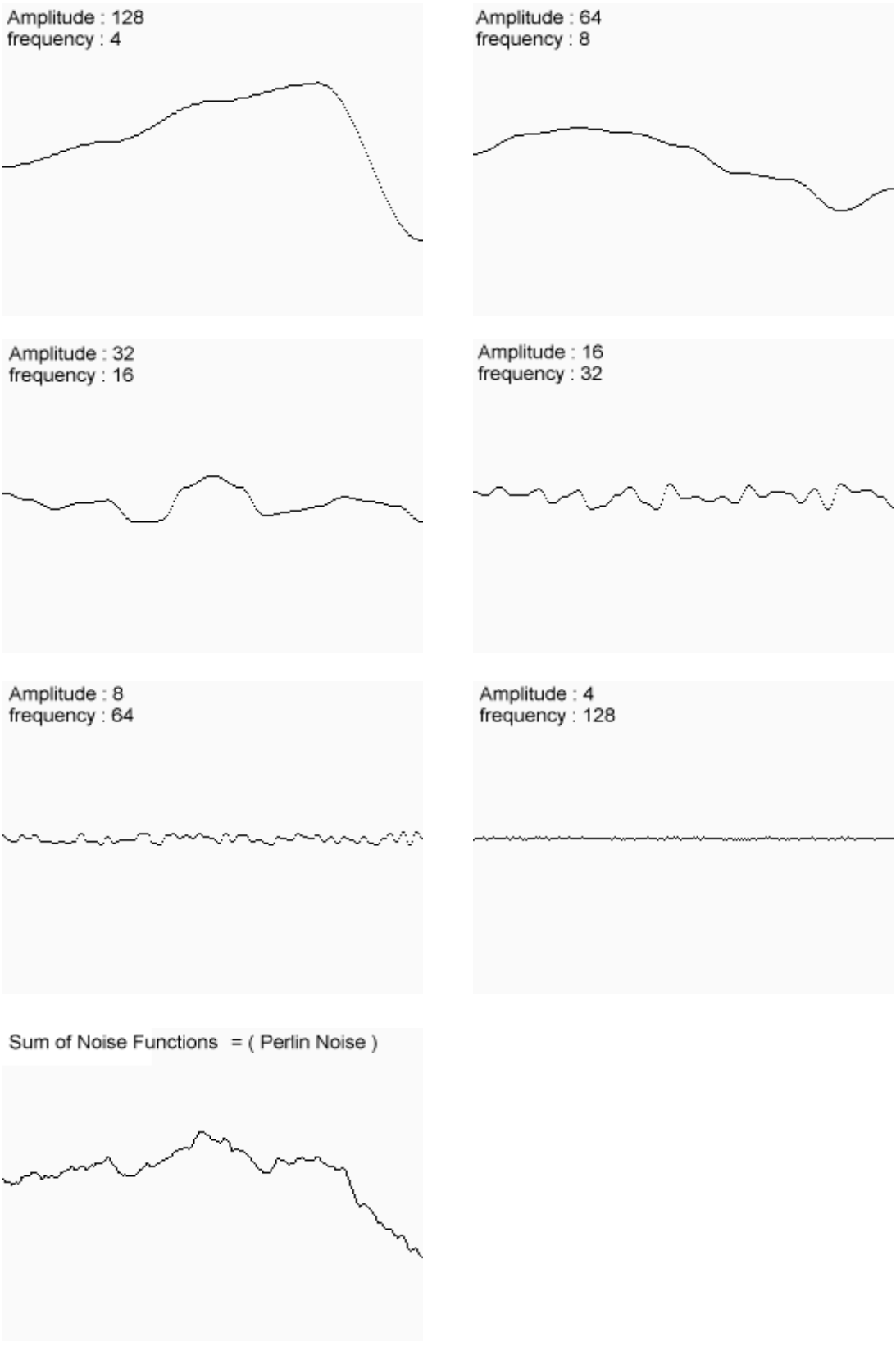
Användandet av perlinbrus är en välkänd metod för att generera texturer men även höjdkartor och naturfenomen som t.ex. molnformationer. Perlinbrus, pn, är värden som, enligt nedan, beräknas för punkter i n - dimensionella rummet $\mathbf{R}^n$, n = 1,2..., med hjälp av slumptal.



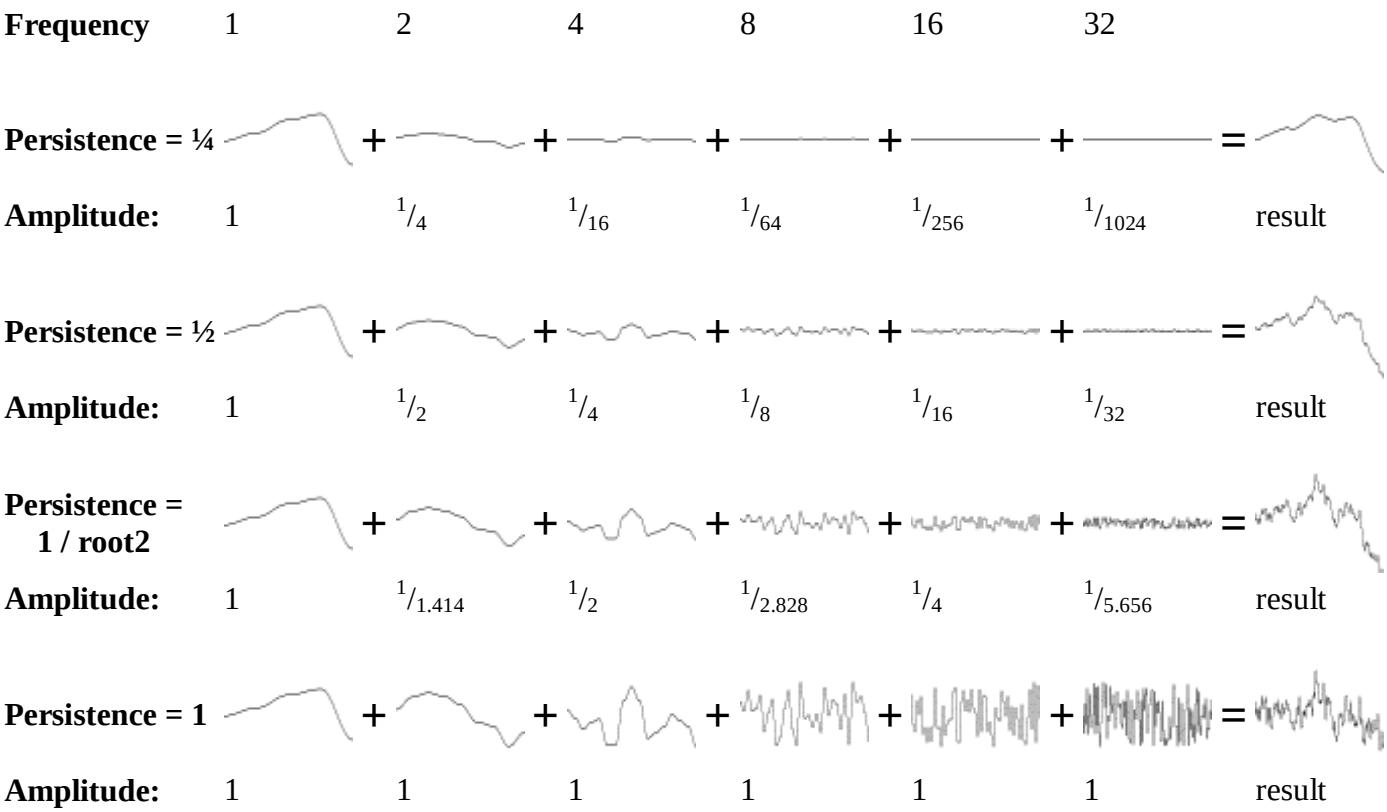
Figur nr. 5: Diagram

Betrakta diagrammet i figur nr. 5. Punkterna däri representerar slumptal för heltalspunkter i ett intervall på x-axeln. Grafen mellan punkterna är resultatet av någon form av interpolation över intervallet. Grafens amplitud är skillnaden mellan största möjliga och minsta möjliga pn-värden i intervallet. Grafens våglängd är avståndet mellan punkterna i intervallet för vilka bruset beräknats. Grafens frekvens över intervallet är 1 / våglängden [8] .

Genom addition av ett antal sådana grafer, beräknade över ett intervall, bildas en perlinbrus - graf för punkterna i intervallet:



Figur nr. 6



Figur nr. 7

Varje sådan brusgraf, som adderas (frekvensfördubbling) för att bilda perlinbrus, brukar kallas en oktav. Detta kan generaliseras till punkter i rum av godtycklig dimension. För punkter i planet åstadkommes på detta sätt landskapsytor i form av perlinbrus [8].

Av pseudokoden nedan för endimensionellt brus framgår, i funktionen **PerlinNoise**(float **x**), hur nämnda addition utförs och hur värdet persistence användes vid brusberäkningen :

/* 15731, 789221, 1376312589 är en unik uppsättning primtal för just slumptalsgeneratorn nr. i */

```
function Noise_i(integer x) )          // slumptalsgenerator nr. i

x = (x<<13) ^ x;
return ( 1.0 - ( (x * (x * x * 15731 + 789221) +
                 1376312589) & 7fffffff ) / 1073741824.0);
end function
```

```
function SmoothedNoise_i(float x)         // slumptalsutjämnare
      return Noise_i(x)/2  +  Noise_i(x-1)/4  +  Noise_i(x+1)/4
end function

function InterpolatedNoise_i(float x)    // interpolator
   integer_X    = int(x)
   fractional_X = x - integer_X
   v1 = SmoothedNoise_i(integer_X)
   v2 = SmoothedNoise_i(integer_X + 1)
   return Interpolate(v1 , v2 , fractional_X)
end function


/* p = persistence , n = Number_Of_Octaves - 1 */

function PerlinNoise(float x)
      total = 0
      loop i from 0 to n
      frequency = 2^i
      amplitude = p^i
      total = total + InterpolatedNoise_i(x * frequency) * am
              plitude
      end of i loop
      return total
end function
```

3.1.5 Kommentarer

Nedanstående beskrivning av begreppet fraktal dimension är hämtad från boken Texturing and Modeling [2]. Den fraktala dimensionen har ett heltalsvärde > 1 och ett decimalt >= .0, <= .999. Heltalsvärdet anger dimensionen hos det geometriska objekt med vilket den fraktala formförändringen startar. T. ex. de ovan beskrivna metoderna startar med ett plan och heltalsvärdet i deras fraktala dimension blir därför 2.

Hade startobjektet varit en linje eller en kub hade således heltalsdelen blivit 1 resp. 3. Den decimala delen kan sägas vara ett mått på tätheten hos utfyllnaden av dimensionen, näst över heltalsdelens, som skapas genom formförändringens upprepningar. Ju mer veckad en utfyllnad blir desto större täthet har den. Beträffande de ovan beskrivna metoderna innebär detta att ju högre fraktal dimension deras resp. formförändringar har desto skrovligare blir slutresultatet.

Gemensamt för de ovan beskrivna metoderna är att höjderna genereras över hela kartans yta genom formförändringar med en konstant fraktal dimension, medförande samma grad av skrovlighet överallt. Verkliga landskap, åtminstone sett över ytor i storleksordningen kvadratkilometer, är emellertid långt ifrån så homogena.

Bergsmassiv reser sig ur flacka, större omgivande ytor, vid sin fot kantade av låga kullar skapade av inlandsisen. Lägre liggande områden har vanligen än jämnare topografisk struktur än högre som, bl.a. pga. erosion, även tenderar att öka i skrovlighet mot sin topp. Vidare har dalar, oavsett på vilken höjd de ligger, jämna bottnar [2].

För att uppnå mer naturtrogna topografier, enligt ovan, har Ebert et. al. [2], skapat perlinbrusbaserade fraktala formförändringar, vars dimension kan variera lokalt för varje punkt de beräknas. Metoden för detta finns som pseudokod med kommentarer, att studera i bilaga A för den intresserade, men användes ej i någon av våra implementationer.

Användes perlinbrusbaserade metoder behöver inte landskapskartan sparas i minnet, ty höjden i varje punkt (x, z) kan beräknas när helst den behövs. Detta ger möjligheten att modellera landskapsytor i realtid (som i Blueberry3D), vilka inte tar slut vid någon kant och med önskad detaljrikedom. Man kan sucessivt zooma in på allt mindre delar av ett landskap genom att minska avståndet mellan mellan de punkter för vilka perlinbrus beräknas.

3.2 LOD - tekniker

En terräng visualiserar ofta ett vidsträckt landskap. Delar av ett sådant, som befinner sig långt bort från betraktaren uppfattas inte med samma detaljrikedom som de i förgrunden. Dessa förstnämnda kan alltså visualiseras med ett mindre antal polygoner än de sistnämnda utan synbarlig kvalitetsförsämring av landskapsbildens naturtrogenhet med större snabbhet. Motsvarande gäller naturligtvis även flata ytor kontra mera skrovliga. Tekniker som med detta syftet reducerar antalet polygoner, som skickas genom renderingspipelinen brukar kallas LOD-tekniker (LOD = level of detail = detaljrikedom).
De består i allmänhet av tre delar:

- Generatorn, som modellerar instanser med olika detaljrikedom av objektet
- Väljaren, som väljer instans med en tillräcklig detaljrikedom för rendering, baserat på något kriterium som ex. dess uppskattade utrymme på skärmen
- Växlaren, som hanterar den nödvändiga växlingen mellan olika nivåer av detaljrikedom.

Visualisering av terräng innebär svårigheter i så motto att den inte består av delar, vars detaljrikedom kan justeras oberoende av varandra samt att denna är beroende av kameravyn.

LOD-tekniker inför terrängrendering bör uppfylla följande kriterier [15], [7]:

1. Kunna åstadkomma olika nivåer av detaljrikedom för skilda delar av terrängen
2. Undvika att terrängen rämnar (cracking) eller på annat sätt blir konstig i gränsområdet mellan områden med olika detaljrikedom.
3. Undvika diskontinuitet vid övergång mellan olika nivåer av detaljrikedom
4. Stödja upprätthållande av en minimum frame-rate under renderingen

Med tanke på den snabba utvecklingen av grafikhårdvaran, som bl. a. inneburit avlastning av CPU: n, kan som ett ytterligare kriterium tillfogas att teknikerna ska stödja och inte motverka en optimal användning av till buds stående hårdvaruresurser. Nedan följer nu en typöversikt av förekommande LOD-tekniker, vilka avslutningsvis kommenteras med hänsyn till kriterierna ovan.

3.2.1 Nätpunktsurval

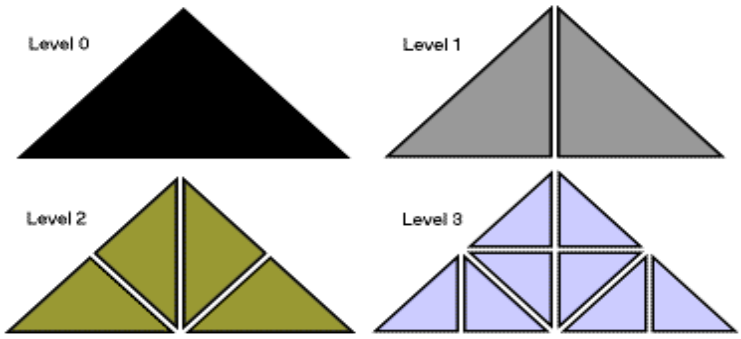
I sin enklaste form väljes punkterna i var k:te rad och kolumn på höjdkartan varefter urvalet trianguleras inför varje frame. Inga andra punkter beaktas därvid. Detta kan utvecklas genom att man delar in höjdkartan i block. I samband med att höjdkartan lästs in beräknas sedan ett quad- eller octträd för höjdkartans begränsningsbox (bounding box), vars löv utgör blockens begränsningsboxar. Inför varje frame traverseras trädet och de block märkes därvid synliga, vars begränsningsboxar åtminstone delvis ligger inom siktvolymen. Dessa block trianguleras och skickas sedan in i renderingspipelinen.

För att åstadkomma en varierad detaljrikedom bland blocken, enligt kriterium 1 ovan, kan exv. användas en metod, som benämnes GeoMipMapping, och påminner om mip-mapping [6]. Enligt metoden beräknas och lagras i minnet, efter höjdkartans inläsning, för varje block en sekvens, kallad GeoMipMap, av approximationer med avtagande triangelupplösning. Med beaktande av ett synligt blocks avstånd till kameran hanterar en väljare vilken av dess GeoMipMaps upplösningsnivåer, som ska renderas, och en växlare övergången mellan dessa val. P.g.a att växlare och väljare är ganska invecklade och då metoden ifråga ej implementeras av oss beskrives de ej närmare.

3.2.2 Hierarkisk uppdelning

Här delas terrängen rekursivt upp i en trädstruktur. Varje nod i denna är associerad med en region av terrängen. I varje steg av uppdelningen utförs siktvolyms-klippning. Om en nod-region ligger helt utanför stoppas förgreningen till denna. Annars avgörs på basis av något kriterium, exempelvis avståndet till kameran, om regionens detaljrikedom är tillräcklig för rendering. Om inte går rekursionen vidare med uppdelning av regionen.

Ett flertal uppdelningsmetoder finnes varav ROAM-algoritmen är den mest kända ROAM betyder Realtime Optimally Adapting Meshes, och presenterades av Duchaineau et al. [7], som en vidareutveckling av metoden sådan den tidigare utarbetats av Lindström et al. [11]. Indata till algoritmens generator är rätvinkliga trianglar där kateterna har (2^n +1) punkter. Den kan i princip tillämpas direkt på var och en av de två trianglar, som uppkommer om en höjdkarta med (2^n +1) * (2^n +1) punkter klyvs längs sin diagonal. Då algoritmen ökar sitt minnesutnyttjande exponentiellt bör höjdkartan delas in i ett antal mindre trianglar på vilka den sedan körs. Indelningen görs lämpligen så att höjdkartan delas upp i ett antal kvadrater med (2^n +1)* (2^n +1) punkter, som var och en klyvs längs diagonalen till två trianglar. Dessa utgör indata till algoritmen och får lagras i någon passande datastruktur. Två trianglar som härrör från samma kvadrat bildar en diamant, vars innebörd förklaras senare nedan. Generatorn delar rekursivt upp indatatriangeln i ett binärt träd av trianglar enligt figur nr. 8.

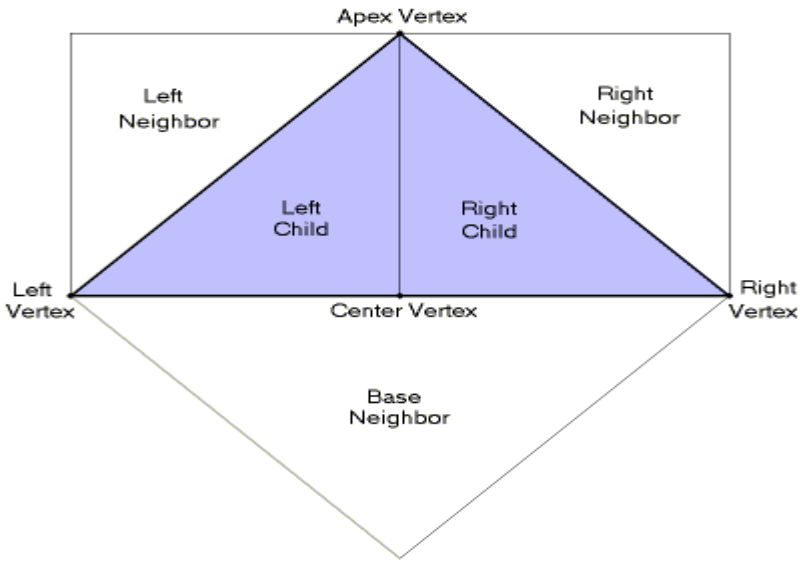


Figur nr. 8: Uppdelning till ett binärt träd

Triangeln på nivå 0 är trädets rot. Trianglarna på nivå ett är, från höger räknat, rotens höger- resp. vänsterbarn. De två trianglarna, på nivå 2, längst till höger, är från höger till vänster räknat, höger- resp. vänsterbarn till rotens högerbarn. De två till vänster blir, räknat på samma sätt, höger- resp. vänsterbarn till rotens vänsterbarn. På så sätt kan fortsättas till nivå 3 osv. rekursivt.

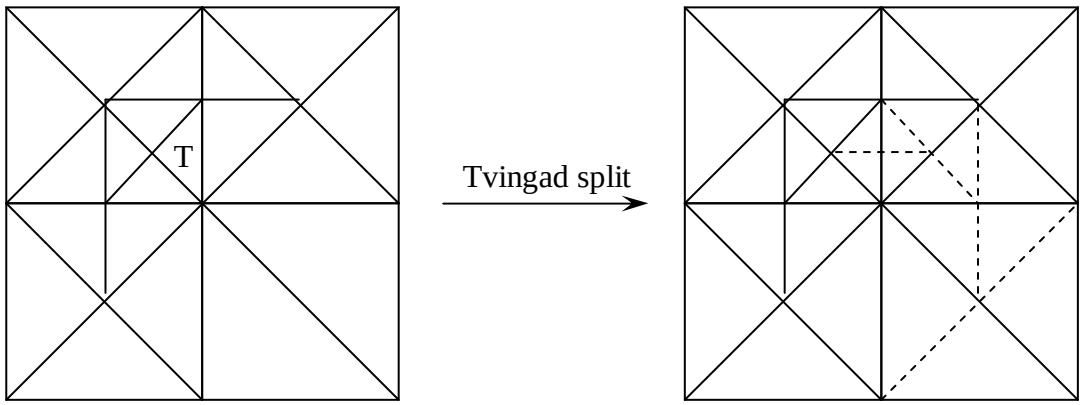
För approximering av höjdkartan fortskrider algoritmen i två pass för varje frame. I det första bildas träd genom att rottrianglarna upplöses till önskad detaljrikedom eller traverseras i tidigare förstapass bildade träd varvid upplösningen kan ökas eller minskas. I det andra traverseras träden varvid deras lövtrianglar renderas. Upplösningen ökas genom delning enligt ovan, som kallas split, och minskas genom sin motsats som kallas merge och innebär sammanslagning av trianglar, som uppkommit genom tidigare split.

Varje inre triangel T i ett träd har de 3 möjliga grannarna till vänster och höger samt mot basen, enligt figur nr. 9 :



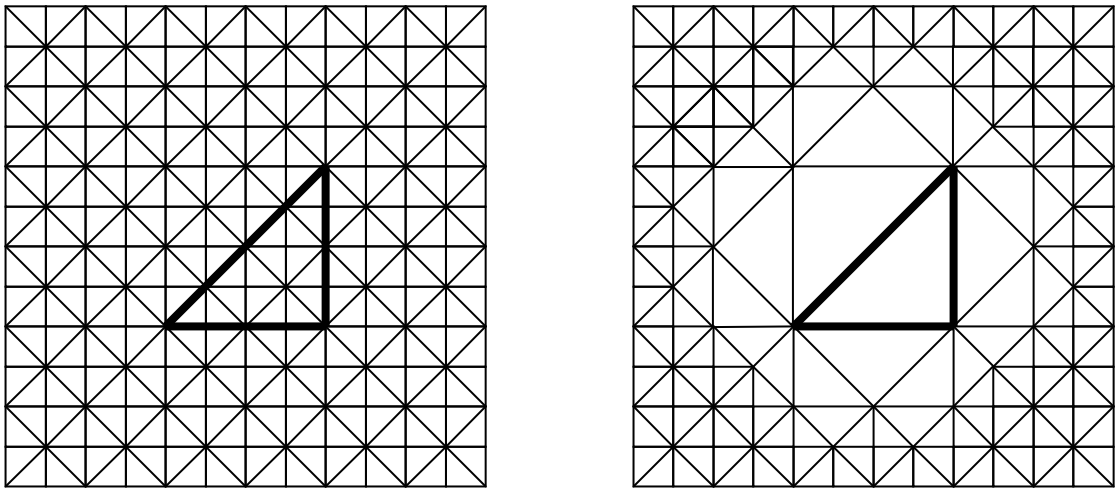
Figur nr. 9

Vänster- resp. högergrannen till T tillhör samma trädnivå l som T, alternativt nivån (l + 1) under. T`s basgranne Tb ligger på samma nivå l som T, alternativt på nivån (l - 1) över. Om T och Tb tillhör samma nivå bildar de en diamant och en sådan basgranne betecknas Tbd. Om en av dem delas (split) så måste också den andra delas, för att undvika cracking och andra konstigheter (shading dicontinuities). Detta medför, om Tb tillhör nivån över T och T ska delas, att Tb också måste delas för att bilda Tbd. Detta kan leda till en rekursion av framtvingade split som slutar i och med diamantsplit enligt figur nr. 10:



Figur nr. 10: Splitrekursionen sker enligt de streckade linjerna

Om T och Tbd har delats endast en gång kan och måste båda återbildas (merge) genom att deras barn slås ihop, enligt algoritmens ursprungliga utformning. Ögren har visat att detta är en onödig begränsning. Varje tidigare splittad triangel T kan återbildas genom att dess barn tas bort. För att undvika cracking och andra konstigheter måste då alla T`s grannar också återbildas, vilket kan leda till en rekursion av framtvingade återbildningar, vars ände blir grannar som inte är delade, enligt exempel i figur nr. 11:



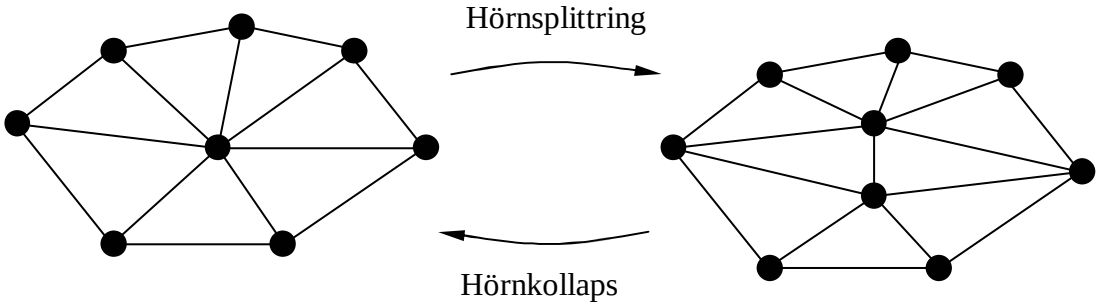
Figur nr. 11: Framtvingad återbildningsrekursion

För att undvika diskontiunitet användes morphing, innebärande att split och återbildning animeras genom en följd av interpolerade frames. Väljaren baseras på en metrik, som mäter felet, vilket uppkommer vid approximation med en viss grad av detaljrikedom jämfört med maximal sådan. Algoritmens resultat är adaptivt optimala i den meningen de minimerar marginalen för nämnda fel givet approximation med ett visst antal trianglar och en viss bildfrekvens.

3.2.3 Progressiva nätverksmetoder

Ett känt exempel är Hoppes progressiva metod, presenterad av Ögren [15]. Den skapar en sekvens av triangelnätverk, approximerande en terräng med varierande detalj-rikedom, kalkylerad med hänsyn till terrängens struktur och kamerans position. Här beskrives metoden översiktligt med utelämnande av detaljerna beträffande väljare och växlare.

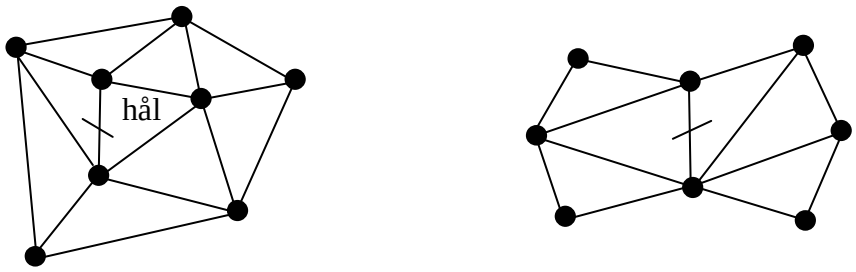
Sekvensen beräknas med utgångspunkt från en minsta detaljrik triangelapproximation M0 av höjdkartan och representeras av en skog, vars träds rötter är hörnpunkterna i M0. Träden är binära och kan växa genom att rötterna splittras i två nya punkter, som blir dessas höger- och vänsterbarn, vilka i sin tur kan splittras o.s.v. Ett barnpunktspar sammanbindes med varandra samt de punkter föräldern var förenade med så att nätverket utökas med två nya trianglar, se figur nr. 12 :



Figur nr. 12: Exempel på hörnsplittring resp. hörnkollaps

Skogens löv, som hålls i en lista, blir alltid hörnpunkterna i det sist beräknade av sekvensens nätverk, vars trianglar också lagras i en lista. Ett hörn i ett nätverk kan kollapsa in i ett annat, som ses i figur 12, varvid två trianglar försvinner. Inför varje frame traverseras nämnda hörnlista, varvid för varje element bestämmes om det ska splittras eller kollapsa in i något annat. Efter varje sådan operation uppdateras hörn- och triangellistan. Efter traverseringen renderas så det nybildade nätverket.

En kollaps får inte leda till att nätverkets topografiska struktur ändras hur som helst. t. ex. får den inte medföra att ett hål försvinner eller ske längs en inre triangelsida som förenar två kantpunkter, se figur nr. 13.



Figur nr. 13: Streckade bågar markerar otillåten kollaps

3.2.4 Kommentarer

Ovan beskrivna metoder synes stödja uppfyllandet av kriterierna 1 - 4 ovan. Men med tanke på att de, undantaget GeoMipMappingen, skapades innan eller omkring mitten av 90-talet kan det ifrågasättas huruvida de verkligen stödjer ett optimalt användande av dagens hårdvaruresurser. De Boer menar att metoderna ovan, undantaget hans egen GeoMipMap version, skapades innan 3d-hårdvarurendering slog igenom och med inriktning på att beräkna i någon mening optimala renderingsdata för sin tids teknik [6]. Eftersom dagens 3d-hårdvara kan rendera mycket fler polygoner än denna per tidsenhet, kan man nu använda annorlunda metoder inriktade på att skicka så många trianglar genom renderingspipelinen som denna kan hantera med minsta möjliga CPU-belastning. Hans GeoMipMapping beträffande landskapsytor skulle vara ett exempel på en sådan metod.

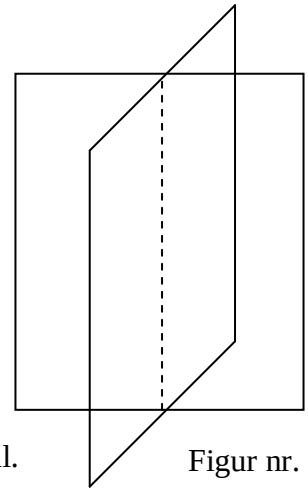
Han har nog rätt beträffande de ovan beskrivna metodernas inriktning men att de, med hänsyn till kapaciteten hos dagens hårdvara, generellt skulle innebära en onödig belastning av CPU: n är mera osäkert. Fortfarande finns ju gränser för hårdvarans kapacitet och utökningar av denna exploateras snabbt genom skapande av nya applikationer, inte minst spel, med allt större detaljrikedom i sina scenerier. Det kan således fortfarande finnas anledning att spara in på antalet polygoner som skickas till rendering, genom användande av metoderna ovan. Men man bör noga överväga deras användning så att de inte innebär en onödig CPU-belastning, som i värsta fall kan leda till både försämrad bildfrekvens och bildkvalitet.

3.3 Modellering av vegetation

Då man skall modellera träd och växter för att i realtid sedan visa dem, finns det några saker att tänka på. Till en början kan det vara klokt att fundera på i vilken typ av applikation man skall använda modellen. Utvecklar man ett så kallat *”first-person-shooter”* spel, kanske det finns andra detaljer som man skall lägga datorkraften på och istället välja en ganska enkel modell, t.ex. *”Billboards”*. Redan nu kan man flika in kommentaren att dagens datorer med dess grafikkort, klarar visa så mycket mer än bara för något år sedan. Detta gör att man allt som oftast ser en kombination av tekniker, som fungerar mer eller mindre bra. Det vi under denna punkt skall visa på är ett antal exempel på olika tekniker som finns att tillgå för visualisering av vegetation. Tekniker för att med hjälp av en bild av en växt, träd, el.dyl., skapa ett intryck av ett tredimensionellt föremål, kallas för bildbaserade renderingstekniker.

3.3.1 Billboardtekniken

Så kallade *”Billboards texture maps”*, är en enkel metod för att ge betraktaren en känsla av att han/hon tittar på ett tredimensionellt objekt. Det man gör är att man på en flat rektangulär polygon, mappar en bild av ett träd, buske el.dyl.. Vanligtvis använder man sig av två sådana bildskivor och placerar dem i kors, se figur nr. 14.



Figur nr. 14.

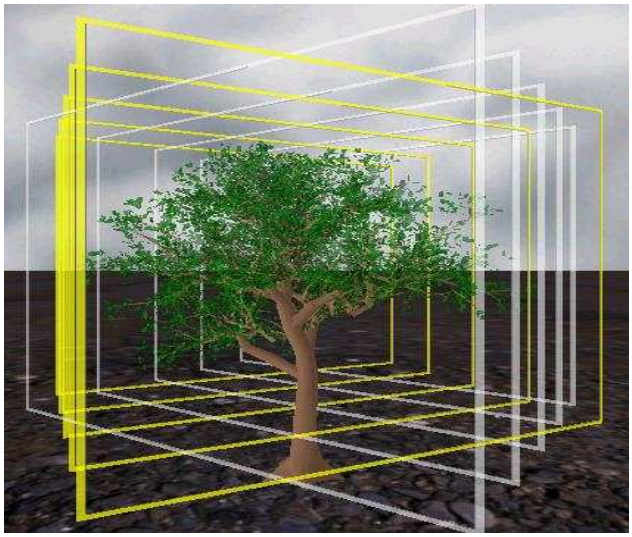
Som ett alternativ kan man istället använda sig av endast en bildskiva och låta denna vrida sig mot betraktaren. För att få trädet att se ytterligare realistisk ut, kan en svart polygon placeras under trädet, symboliserande en skugga. Man kan även tänka sig att man lägger in fler skivor och på det sättet öka intrycket av ett tredimensionellt föremål.

En annan teknik, som bygger vidare på billboardtekniken, är riktade billboards. Idén här är att man fortfarande använder sig av den enkla skivgeometrin, men att man skiftar mellan en till flera olika möjliga texturmappningsbilder, beroende på i vilken riktning betraktaren tittar.

3.3.2 Skivtekniken

Med skivtekniken renderar man exempelvis ett träd i form av skivor, vilka i sin tur är billboards, se figur nr. 15. Skivornas texturer genereras från en detaljerad polygonmodell av trädet. En skiva renderas sedan som en texturerad rektangel. Då det gäller att betrakta trädet från olika håll, räcker det inte med endast en uppsättning skivor, utan den naturliga lösningen blir att skapa en uppsättning olika bildskivor, för tänkbara riktningar.

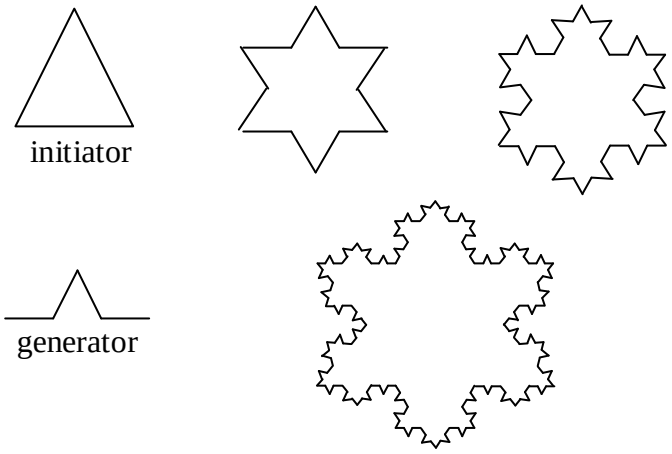
För att förhindra att man flippar mellan bildskivorna och minska dess antal, så kan man rendera två skivor på en gång, och använda *bländning* för att mjuka upp övergången. Detta fungerar bra så länge trädet är tillräckligt långt borta, men mindre bra om betraktaren kommer nära trädet.



Figur nr. 15: Exempel på skivteknik [17]

3.3.3 L-system och procedurell modellering

Det centrala i L-system är så kallad omskrivning. Med utgångspunkt från ett enkelt objekt och användandet av ett antal regler, kan man med hjälp av omskrivning definiera ett komplext objekt. Ett klassiskt exempel på ett grafiskt objekt som är definierad med hjälp av omskrivning är *snöflingskurvan*, framtagen 1905 av von Koch. I figur nr. 16 visas exempel på hur en sådan kurva konstrueras och på nästa sida ges en förklaring därtill.



Figur nr. 16: Konstruktion av snöflingskurvan

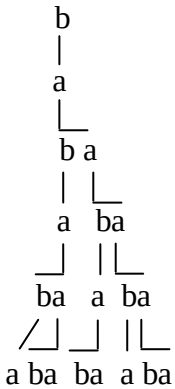
Mandelbrot beskriver framtagandet av denna kurva, med följande ord:

”Man börjar med två figurer, en initiator och en generator. Den senare är en orienterad bruten linje, bestående av N lika stora sidor med längden r. Varje steg av konstruktionen börjar sedan med en bruten linje och består i att, ersätta varje rakt intervall med en kopia av generatorn, reducerad och omplacerad, så att den har samma ändpunkter som det intervall som den ersätter.”

Det förkommer flera exempel på olika omskrivningssystem i litteraturen, och ett klassiskt exempel är ett system som skriver om teckensträngar. Här förekommer framförallt Chomsky’s arbete med en formell grammatik, från 1950. Men 1968 introducerade biologen Aristid Lindenmayer en ny typ av sträng omskrivningsmekanism, som döptes till L-system. Den viktigaste skillnaden mellan Chomskygrammatik och L-system, är på det sätt man tillämpar produktioner. I Chomskygrammatik, tillämpas produktioner sekventiellt, medan man i L-system tillämpar dem parallellt, och kan ersätta bokstäver i ett ord simultant. Den här skillnaden säges reflektera den biologiska motivationen för L-system. Produktionen är tänkt att motsvara en celldelning i multicellsorganismer, där många delningar kan ske samtidigt.

För att förklara hur L-system är uppbyggda, börjar vi här med att beskriva den enklaste klassen av L-system, DOL-system. Dessa system är deterministiska och kontextfria. Ett enkelt exempel är:

”Betrakta strängar (ord) som endast byggs upp av tecknen a och b. Tecknen kan förekomma flera gånger i samma sträng. Varje tecken är vidare associerad med en omskrivningsregel. Regeln a -> ba, betyder att tecknet a skall skrivas om till strängen ab, och regeln b -> a, betyder att b skall skrivas om till ett a. Omskrivningsprocessen börjar sedan med en särskild sträng som kallas axiom. Antag att axiomet endast innehåller tecknet b. I det första omskrivningssteget ersätts då b med a, genom användandet av produktionen b -> a. I det andra steget ersätts a med ba, enligt produktionen a -> ba. Strängen ba består av de två tecknen b och a, som ersätts samtidigt, simultant, i det tredje steget. Då får vi att b ersätts med a, och a ersätts med ba, vilket ger strängen aba. Efter det fjärde steget får vi strängen baaba, se figur nr. 17, och så fortsätter det tills man är nöjd.”



Figur nr. 17: Exempel på härledning i ett DOL-system

För att åskådliggöra detta med hjälp av en grafisk representation, brukar man ta en sköldpadda till sin hjälp. Dock inte en levande, utan ett virtuellt djur, som man kan ge olika instruktioner. Man fäster dessutom en penna eller något annan typ av markeringsdon på sköldpaddan, så att den kan lämna ett spår efter sig då den vandrar fram över en yta. De fyra första grundläggande instruktionerna man brukar ge sköldpaddan är att:

F Förflytta dig framåt ett steg d. Sköldpaddan ändrar läge till (x',y', α), där x'=x+d*cosα och y'=y+d*sinα. Sköldpaddan ritar på så sätt ett linjesegment mellan punkterna (x,y) och (x',y').

F Förflytta dig framåt ett steg d, men rita ingen linje.

\+ Vrid dig vänster med vinkeln δ. Sköldpaddans nästa läge är (x,y, $\alpha+\delta$). Sköldpaddans positiva orientering av vinklarna är medurs.

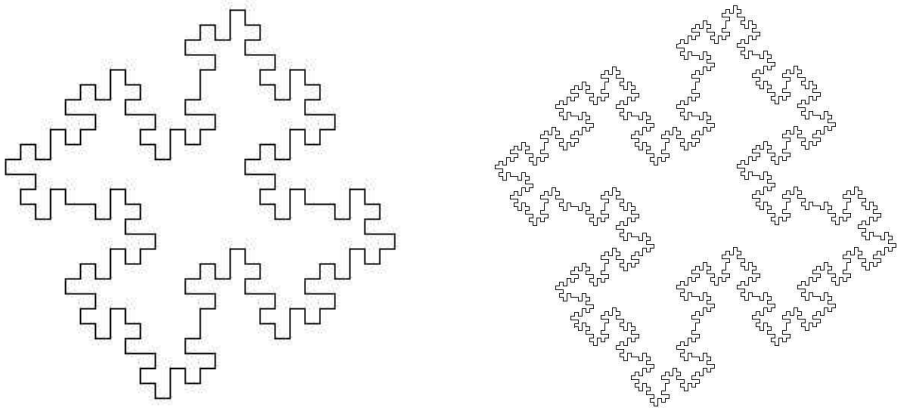
\- Vrid dig höger med vinkeln δ. Sköldpaddans nästa läge är (x,y, $\alpha-\delta$).

Givet en sträng v, det initiala läget på sköldpaddan (x0,y0, α0) och parametrarna d och δ, motsvarar sköldpaddeinterpretationen av v, den figur sköldpaddan ritar upp, med hänsyn tagen till strängen v.

Figur nr. 18, visar ett exempel på en fyra stegs approximation av den *kvadratiska Koch ön*, som är tagen från boken The Algoritmic Beauty Of Plants [3]. Figurerna är framställda genom en sköldpaddeinterpretation av strängar som genererats av följande L-system:

w: F-F-F-F
p: F->F-F+F+FF-F-F+F

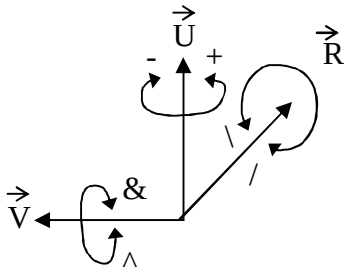
Bilderna motsvarar de strängar som härleds under stegen 0 till 3. Vinkeln δ är nittio grader. Steglängden d minskas för varje steg, så att avståndet mellan ändpunkterna på den efterföljande polygonen är lika långt, som den föregående polygonens.



Figur nr. 18: Generering av en kvadratisk Koch ö.

Exemplet visar på ett nära släktskap mellan Koch- konstruktioner och L-system. Initiatorn korresponderar mot axiomet och generatorn korresponderar mot produktionens efterföljare.

Genom att representera sköldpaddans position i rummet, med hjälp av tre vektorer $\vec{R},\vec{V},\vec{U}$ kan en sköldpaddeinterpretation utökas till tre dimensioner. Vektorernas beteckningar motsvaras av r för riktning framåt, v för riktning till vänster och u för upp. De tre vektorerna är enhetsvektorer och är vinkelräta i förhållande till varandra.



Figur nr. 19: Kontroll av sköldpaddan i tre dimensioner

Följande symboler kontrollerar sköldpaddans orientering i rummet, figur nr. 19 :

+	Rotera åt vänster med vinkeln δ genom tillämpning av rotationsmatrisen $\mathbf{R_U}(\delta)$
–	Rotera åt höger med vinkeln δ, genom tillämpning av rotationsmatrisen $\mathbf{R_U}(\delta)$
&	Vänd neråt med vinkeln δ, genom tillämpning av rotationsmatrisen $\mathbf{R_V}(\delta)$
^	Vänd uppåt med vinkeln δ, genom tillämpning av rotationsmatrisen $\mathbf{R_V(}\delta)$
\	Rulla åt vänster med vinkeln δ, genom tillämpning av rotationsmatrisen $\mathbf{R_R}(\delta)$
/	Rulla åt höger med vinkeln δ, genom tillämpning av rotationsmatrisen $\mathbf{R_R}(\delta)$
|	Vänd runt, genom tillämpning av rotations matrisen $\mathbf{R_U}(180°)$

Det finns sedan ytterligare symboler för att kontrollera sköldpaddans rörelse vid exempelvis uppbyggnaden av ett komplext föremål. De viktigaste är kanske [och] som används för att lagra sköldpaddans läge på en stack.

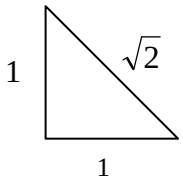
[lägger (push) sköldpaddans läge på stacken. I den information som lagras ingår sköldpaddans position, orientering, och andra attribut som linjefärg och linjevidd.

] hämtar (pop) ett läge från sköldpaddan. Det ritas inte ut någonting men sköldpaddan ändrar läge, till det som hämtats från stacken.

Dessa kan användas för att åstadkomma en trädliknande form. Sköldpaddan börjar nere vid roten och fortsätter sedan uppåt enligt den beskrivning som givits. När den sedan kommer till en gren eller annan utväxt (ex. blad, blomma, knopp), lagrar den sitt tillstånd på stacken för att, efter utritandet av utväxten, kunna fortsätta sin vandring. Genomgången av beskrivningen görs, som vid ovanstående exempel på utritandet av en fraktal figur, rekursivt.

För att modellera ett fullständigt träd med rot, stam, grenar och blad, ges bladen en fullständig beskrivning i form av en yta. När sedan sköldpaddan kommer fram till tecknet ~ i beskrivningen av trädet, sparar den sitt tillstånd på stacken, och övergår till att rita ut den beskrivning av bladet som inkluderats. Tecknet ~ är endast taget som ett exempel på en symbol, vilken används framförallt i boken The Algorithmic Beuaty of Plants [3]. Modellerar man en växt, som har olika typer av utväxter, kan man på samma sätt ge dessa olika symboler i beskrivningen.

Till slut skall vi nämna något om parametriska L-system, detta då de ovanstående beskrivna sköldpaddetekniker har en viss begränsning i sitt användningsområde. Som ett enkelt exempel på en sådan begränsning, kan man ta uppritandet av en rätvinklig triangel med sidan ett, figur nr. 20. Denna triangel blir omöjlig att rita korrekt med tanke på heltalsbegränsningen. Triangelns båda kateter kommer få längden ett medan hypotenusan får längden roten ur två. För att komma ifrån dessa heltalsbegränsningar, föreslog Lindenmayer att man skulle kunna ange numeriska parametrar tillsammans med L-systemets symboler. Detta gör att man får en mer kraftfull och flexibel teknik för att modellera en växt, på ett naturtroget sätt. Till de ovanstående genomgångna instruktionerna kan man således associera en parameter. Som exempel på detta kan man ta instruktionen F, fortfarande med syftning att ge sköldpaddan en instruktion:

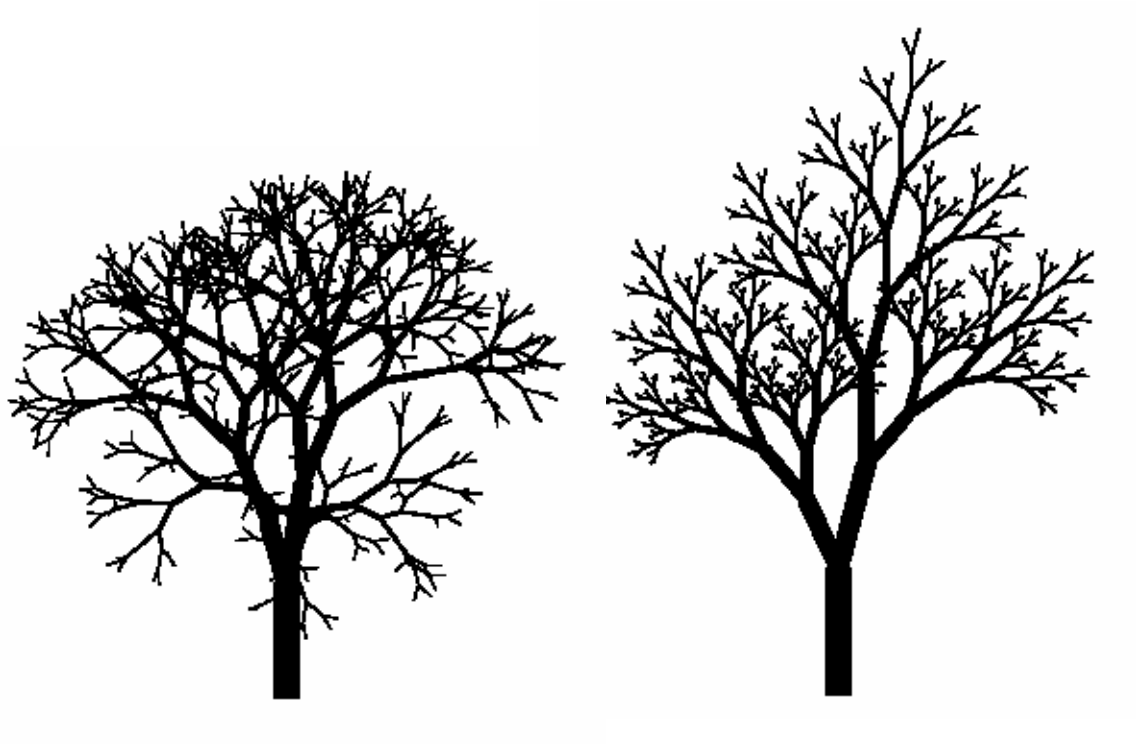


Figur nr. 20: Rätvinklig triangel med sidan ett.

F(a) Förflytta dig framåt ett steg av längden a > 0. Sköldpaddans position ändras till läget (x',y',z'), där $x' = x + a*R_x$, $y' = y + a*R_y$, $z' = z + a*R_z$ där R är sköldpaddans riktningsvektor. Sköldpaddan ritar på så sätt ett linjesegment mellan punkterna (x,y,z) och (x',y',z').

Som en sammanfattning av L-system, kan man säga att de utgör ett system, som kan användas för att framgångsrikt modellera växter. Ovanstående genomgång är endast menad som en kort introduktion till L-system. Det finns många fler instruktioner som man kan ge sköldpaddan, mycket beroende på hur och från vilken teknik man utgår. Detta då det finns många som med utgång från Lindenmayers beskrivningar, utökat dessa, för att passa just den växtgrupp de önskar modellera.

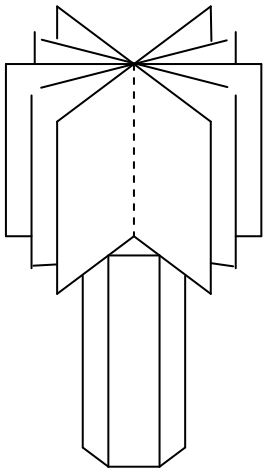
Vidare kan man säga att L-system används framförallt till att rendera tredimensionella bilder av en växt. Det vill säga, de används för att skapa en bild eller modell, som nödvändigtvis inte är tänkt för ett realtidsändamål. Detta mycket beroende på att en detaljerad modell av en växt kan ta lång tid att rendera, samt att den innehåller så mycket information, att det blir beräkningsmässigt svårt att tillämpa modellen i realtidssammanhang. Till slut ges två exempel på trädliknande bilder i figur nr. 21, skapade med parametriska L-system.



Figur nr. 21: Exempel på två trädliknande bilder, skapade med parametriska L-system.

3.3.4 Tekniker för handmodellering av vegetation

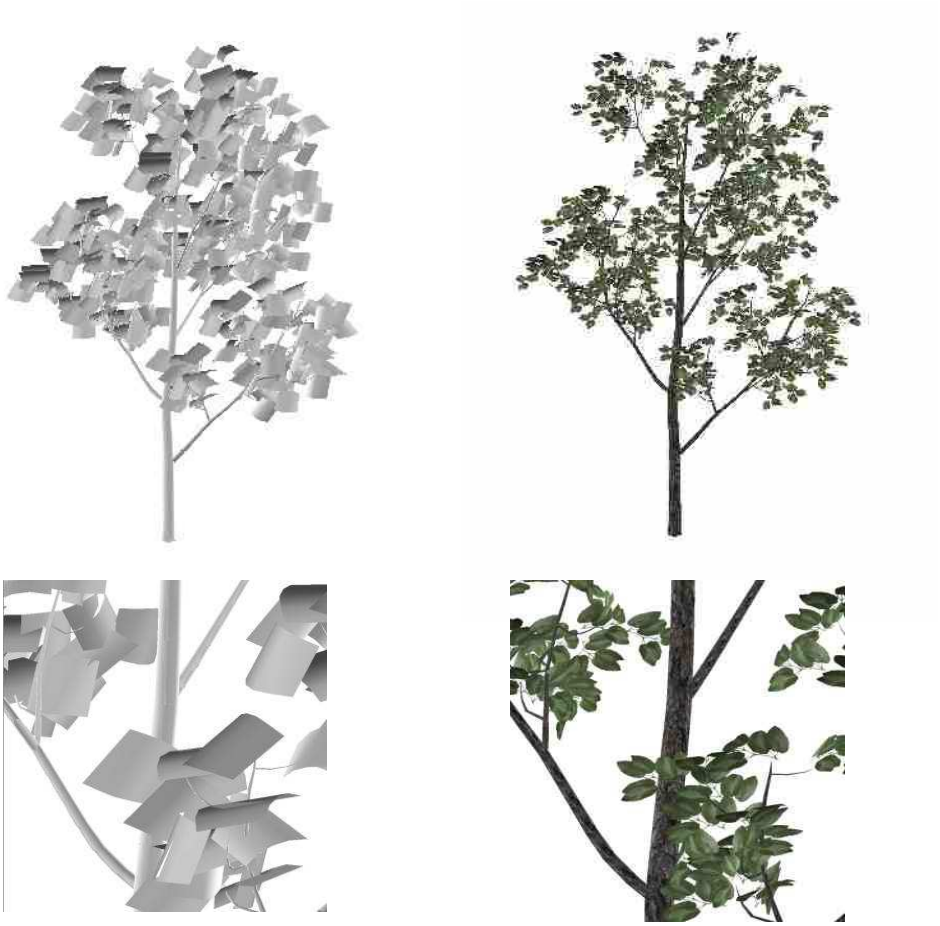
En teknik som används ofta i framförallt datorspel, är handmodellerad vegetation. Med detta menas en teknik som är ett mellanting av billboards och en hel geometri av en växt. Växten byggs vanligtvis upp av en rörliknande stam, bestående av fem till åtta sidor. Varje gren konstrueras av väldigt få polygoner eller en billboard som en hel gren. Beroende på hur man konstruerar och designar växten samt kvaliteten på billboarden kan man få en växt att se mycket naturtrogen ut både på avstånd och på ett relativt nära avstånd. Figur nr. 22 visar exempel på hur man kan konstruera ett träd på ovannämnda sätt.



Figur nr. 22: Exempel på ett träd.

Här kan man även tänka sig att lägga skivor på höjdledden också, allt är upp till den som designar trädet. I spelet Jedi Knight II: Jedi Outcast förekommer det träd som är skapade på just det sätt som visas i figur nr. 22.

Med hjälp av programmet XFrog, från företaget Greenworks, kan man på ett intuitivt och förhållandevis enkelt sätt modellera sin växt. Man bygger upp sin växt med hjälp av olika komponenter, där stammen är en komponent, bladen är en annan o.s.v. För varje komponent kan man sedan ställa in en mängd parametrar, t.ex. tjockleken på grenen (stammen), grenens form, vilken typ av utskott som skall finnas, vilken textur som skall användas m.m. Vidare finns det animeringsmöjligheter för att t.ex. animera hur en växt rör sig i vinden. För att sedan kunna använda sin modell i en spelmiljö, finns det ett komplementprogram från Greenworks, som kan skala ner antalet polygoner i modellen.



Figur nr. 23: En tredimensionell modell av en körsbärsbjörk (Betula lenta).

I figur nr. 23 visas hur en körsbärsbjörk kan se ut i programmet före och efter texturering. Här ser man att själva bladverket är bilder, som man har projicerat på den angivna ytan, medan stammen och grenarna består av en rörliknande polygonstomme, som i sin tur täcks med en textur för att få en barkliknande struktur.

3.3.5 Sammanfattning

Som en sammanfattning av ovanstående genomgång av de olika tekniker som finns för att modellera vegetation kan man säga att valet av teknik avgörs med tanke på användningsområde. Då det gäller användning av L-system för modellering, ställer dessa vissa krav på användaren. I Interactive Modeling of Plants [12], menar man att L-system kan vara intuitiva för biologer, men med utgångspunkt från modellering, vill man kunna förändra hela växten på en gång. Detta gör att idag används framförallt program som XFrog, som är komponentbaserade snarare än syntaktiska, för att modellera växter för användning i 3D-appliktioner.

En annan synpunkt då det gäller val av modelleringsteknik är i vilken miljö vegetationen skall förekomma. Är det en miljö till ett actionspel man skall modellera, kanske det räcker med att visa billboards, för att spelaren skall få uppfattningen att han/hon springer förbi ett träd. Vill man däremot hitta en balans mellan dessa tekniker, kanske man väljer att handmodellera vegetationen med en stomme som bygger på L-systemen eller är komponentbaserad, medan vegetationens detaljer (blad, löv, knoppar) visas i form av billboards.

Vidare finns det möjligheten att man på långt avstånd visar en bild av trädet, men då betraktaren kommer närmare trädet bländar man över till den tredimensionella modellen av trädet. Fördelen här är att man sparar in på datorkraft, som kan användas till andra beräkningar, t.ex. A.I., animationer o.s.v.. Nackdelen är att det kan uppstå diskontinuitet då man skiftar mellan de olika modellerna.

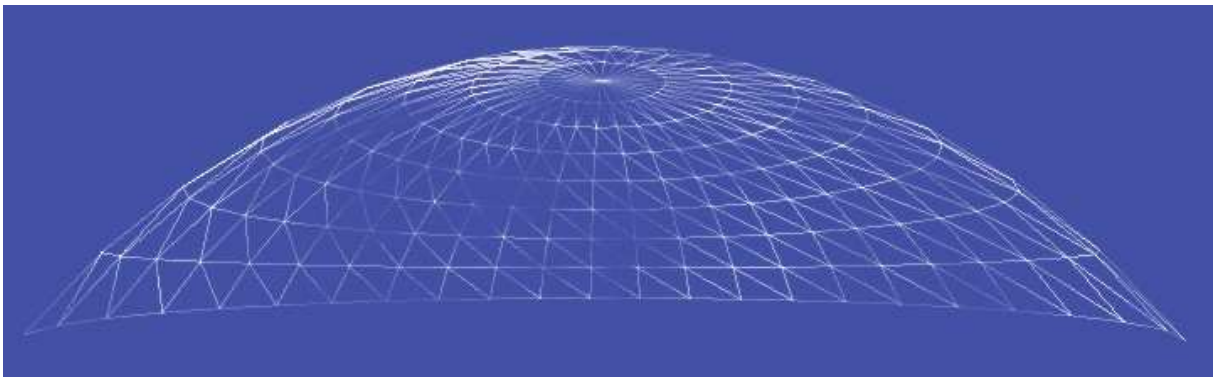
3.4 Himlavalvet, solblänk, dimma och vatten

I följande stycken presenteras olika tekniker för att visualisera atmosfäriska effekter samt även vatten. Till dessa effekter räknas även tekniker för att åskådliggöra himlavalvet på ett naturtroget sätt. Andra effekter är dimma och solblänk.

3.4.1 Himlavalvet

Då man skall visualisera en himmel i en utomhusmiljö, kan detta göras på lite olika sätt. För att ge betraktaren en tredimensionell upplevelse, brukar man använda sig av begreppet *”skydome”*. Begreppet kan översättas med himlavalv, men i det fortsatta resonemanget har vi valt att behålla termen skydome.

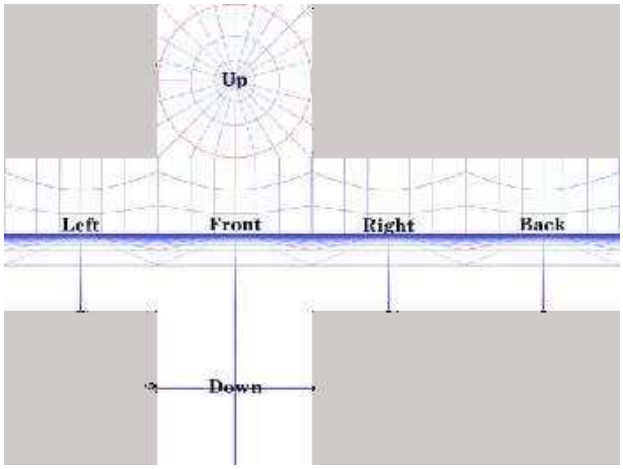
Skydomen modelleras som en halvsfär runt den värld, där betraktaren befinner sig. Betraktaren kommer således befinna sig inuti halvsfären och se i riktning mot dess innerväggar. Figur nr. 24 visar ett exempel på hur en skydome kan se ut. Figuren visar också hur sfären är uppdelad i form av trianglar, som approximerar halvsfären på ett bra sätt.



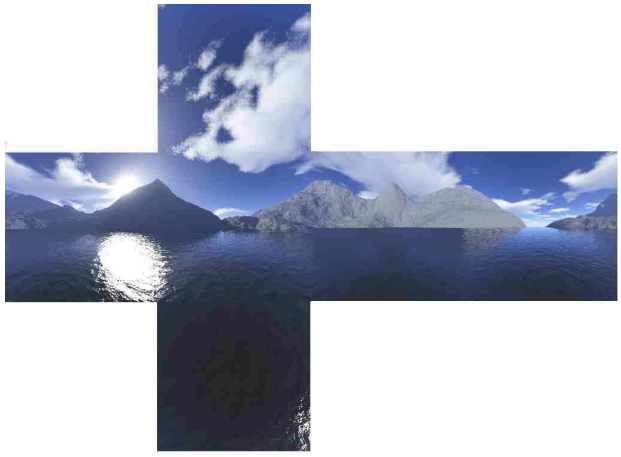
Figur nr. 24: Bilden visar en skydome, ritad med linjer, sedd utifrån [18]

För att sedan ge betraktaren en realistisk upplevelse av att han/hon tittar upp mot en himmel, finns det olika lösningar för detta. Man kan lägga en textur i form av en himmelsbild på insidan av halvsfären. Bilden brukar kallas skybox, i dessa sammanhang. Man tänker sig här att man vikt ut den tänkta bilden, i sex sidor. Dessa sex sidor motsvarar insidan av en kub, och projiceras sedan på halvsfären insida.

Figur nr. 25a visar hur de sex olika delarna av bilden mappas till halvsfärens yta, och figur nr. 25b visar ett exempel på en skyboxbild. Figurerna är tagna från en artikel om hur man skapar skyboxar [5].



Figur nr. 25a: Utvik av en skybox [19]



Figur nr. 25b: Exempel på en skyboxbild

Fördelen med denna teknik är att den kan bli väldig realistisk. Nackdelen är att det är svårt att få himmeln att skifta utseende under t.ex. en dag, från morgon till kväll. Om det dessutom förekommer moln på himmelsbilden, måste dessa befinna sig på ett relativt långt avstånd från kameran. Annars finns det risk för att en betraktare uppfattar molnen på himmeln alltför orörliga då han/hon vandrar runt i landskapet.

En annan teknik för att ge betraktaren en känsla av en realistisk himmel, är att färgsätta insidan av halvsfären. Här brukar man blända från en färg för horisonten till en annan för zenit, d.v.s. halvsfärens högsta punkt. T.ex. brukar en klarblå himmel skifta mellan ljus cyan färg till en lite rikare medelblå färg. Under gryning och skymning förekommer en annan typ av färgsättning, där man rör sig med varma färger som breder ut sig i öster och väster upp till zenit. Genom att här välja en uppsättning attraktiva färger, och sedan interpolera mellan dessa under en tid, kan man på så sätt ge betraktaren en uppfattning av att t.ex. en dag har gått.

3.4.2 Solblänk

Solblänk är en effekt som är vanligt förekommande i en spelmiljö där man vill ge betraktaren en känsla av att han ser upp mot en stark ljuskälla, t.ex. solen. Effekten har hämtats från den reflektion som sker i en kamera med multipla linser. Hur detta fenomen uppstår rent fysiskt brukar man inte bry sig om, då det inte är nödvändigt att känna till för att få en trovärdig effekt. Vi skall nu helt kort ge en förklaring till hur man kan skapa solblänkseffekten i en spelmotor. Det man först behöver är ljusets position i skärmkoordinater (lx,ly) och skärmens centrumkoordinater (cx,cy). Man beräknar sedan riktningsvektorn från centrum till ljusets position och normaliserar den, så att den senare kan användas för skalning.

Ett exempel på hur de bilder man använder sig av vi visualiseringen av effekten visas i figur nr. 26, bilderna är hämtade från Crystal Space.

Figur nr. 26: Exempel på bilder som används vi skapandet av solblänkseffekten.

För att sedan få centrumkoordinaten för den första stjärnbilden, skalar man den normaliserade riktningsvektorn med avståndet mellan (cx,cy) och (lx,ly). Med hjälp av denna centrumkoordinat kan man sedan beräkna de övriga bildernas positioner. Här får man experimentera sig fram med olika avstånd mellan bilderna, då det inte finns någon standard lösning. Figur nr. 27 visar ett exempel på hur solblänkseffekten kan se ut i en utomhusmiljö i Crystal Space.

Figur nr. 27: Exempel på hur solblänkseffekten kan se ut i Crystal Space

3.4.3 Dimma

Då det gäller dimma i en spelmotor, brukar det anges att motorn har stöd för volumetrisk dimma. Att använda ordet volumetrisk är på sitt sätt redundant, då dimma rent visuellt kan ses som partiklar upphängda i luften där dessa partiklar ockuperar en volym i rymden. Desto större täthet mellan partiklarna, ju mindre kan man se genom dimman. Partiklarna som är upphängda i luften blockerar ljuset, och färgen på dimman är ingenting annat än partiklarnas färg. Är partiklarna röda, ser man röd dimma. Med hjälp av detta resonemang kan man bygga upp en beräkningsmodell för dimma i en spelmotor, där dimmans visuella intensitet är en funktion av dimvolymens djup.

I en enkel lineär dimfunktion har man att dimmans intensitet är direkt proportionell mot djupet av dimvolymen.

```
I = täthet * (Zmax – Zmin)
```

Där I är dimmans intensitet i en viss punkt, täthet anger just dimmans täthet i denna punkt och Zmax resp. Zmin definierar djupet av dimvolymen. För att sedan använda dimfunktionen för att rendera en polygon, multipliceras I med dimmans färg för att sedan läggas till och divideras med polygonens ursprungsfärg.

Ovanstående var ett enkelt exempel på hur man kan implementera dimma i mjukvaran. Idag finns det dimfunktioner implementerade i hårdvaran, där man med instruktioner i bl.a. OpenGL kan ange dimvolymen, täthet och dimmans färg.

3.4.4 Vattenytor

Visualisering av realistiskt vatten i spelmiljö är en komplicerad uppgift. Följande punkter kan därvid komma i beaktande :

1. Vattenytans rörelser, som kan vara efterbildade naturliga vågrörelser orsakade av vind eller av föremåls rörelse i vattnet. Hit hör även speciella rörelser av typen plask, som uppkommer exv. då ett föremål faller i vattnet, samt skum på vågkammar och då bränningar slår mot rev. I litteraturen finner man i huvudsak tre metoder för modellering av nämnda rörelser hos vatten : Användning av partikelsystem, vattenytan betraktas som ett elastiskt membran och rörelser hos detta beräknas genom att lösa vågekvationen (en känd partiell differentialekvation för vibrationer hos sådana membran), vattenytans vågrörelser beräknas med hjälp av stokastiska fördelningsfunktioner baserade på experimentella mätningar av rörelser hos verkliga vattenytor exv. havs- och sjöytor [1], [13], [10].

2. Interaktionen mellan vattnet och föremål, som avbildas flytande på dess yta eller på ett visst djup. Enligt litteraturen förefaller detta enklast hanteras genom att betrakta vattenytan som ett elastiskt membran enligt ovan. Användande av partikelsystem är möjligt men kräver mer komplicerad fysikalisk teori [1], [10].

3. Hur ljus från vattnets omgivning inverkar på dess yta samt hur dess ev. botten ska synliggöras. Omgivningsprojektion kan vara en bra metod i dessa fall [10], [4].

En bra spelmotor bör innehålla stöd för visualisering av vatten med beaktande av samtliga punkter ovan, men detta är troligen inte koncentrerat enbart till dess terrängspecifika del, utan utgörs av sådant som även behövs i andra miljöer.

4. Översikt av fyra olika terrängmotorer

De terrängmotorer vi har valt att presentera och ge en översikt av, är utvalda under den förundersökning som gjordes i början av vårt examensarbete. Vi hade som målsättning att hitta den eller de terrängmotorer som visualiserade så naturtrogen terräng som möjligt. Vidare tittade vi på ett antal olika spel och har valt ut ett som vi finner vara representativt för den genren. Vi kommer även att ge en beskrivning av Crystal Space och dess möjligheter till att visualisera en naturtrogen terrängmiljö.

4.1 Urval och motivering

Vi har i vår rapport valt att presentera tre stycken terrängmotorer plus Crystal Space, som är den motor som vi själva skall bygga vidare på. Vi börjar med den terrängmotor som vi anser levererar den mest naturtrogna miljön, Codecreatures. Presentationen av denna terrängmotor blir något kort då vi tyvärr inte hittat så mycket materiell om denna. Vi kommer sedan in på ett svenskt system som heter BlueBerry 3D. BlueBerry uppvisar också en naturtrogen miljö, men den stora visuella skillnaden mellan BlueBerry och Codecreatures är att den sistnämnda uppvisar en till det närmaste fotorealistisk miljö. BlueBerry har däremot andra möjligheter som inte Codecreatures tillhandahåller. Att sedan valet föll på Serious Sam: The Second Encounter är för att det till skillnad från andra spel som vi tittade på, har en bra och informativ webbsida. Vidare har vi funnit att spelet är representativt för hur en terrängmiljö kan se ut i en spelmiljö.

4.1.1 Codecreatures

Codecreatures är en avancerad terrängmotor. Den är framtaget av ett tyskt företag som heter CodeCult för att fungera dels i spel men även i simuleringsmiljöer. Olika möjligheter som Codecreatures tillhandahåller är:

- **Snabb renderare**
  Multinivå - trådbaserad renderingspipeline för optimalt användande av existerande resurser. Avancerat visibilityhanteringssystem, för att undvika utritandet av skymda objekt.

- **Skalbarhet**
  Öka en scens komplexitet genom att oändligt variera uppförandet, och omfånget på scenens detaljnivå. Detta åstadkommes med hjälp av olika tekniker som polygonreduktion i realtid och uppdelandet av ytor.

- **Genetiska algoritmer**
  Organiska objekt som träd och plantor kan genereras under körning med godtycklig komplexitet.

- **Parametriska landskap och omgivningar**
  Skogar, ängar och fält kan organiseras parametriskt och därigenom genereras med hänsyn till systemets hastighet. Träd, plantor och stenar såväl som djur kan genereras dynamiskt för att på så sätt kunna skapa flera olika typer av scener.

- **Vertex- och Pixel-shaders**
  Systemet använder sig mycket intensivt av vertex- och pixelshaders. Pixelshaders låter programmeraren att lägga på ett antal olika skuggeffekter nere på pixelnivå. Några exempel på detta är ljussättning per pixel, reflektioner och bumpmapping. Vertexshaders fungerar på motsvarande sätt fast på vertexnivå. En vertex kan ses som den punkt som håller ihop en geometrisk figur, exempelvis hörnpunkterna i en polygon. Dessa punkter kan tilldelas en mängd olika effekter på motsvarande sätt som pixelshaders.

- **Antal polygoner per bildruta**
  Naturscenerna är mycket detaljerade med stort antal polygoner i siktvolymen (view frustum). Mellan 250 000 och 1 000 000 polygoner renderas för varje bildruta. Hela landskapet kan bestå av bortemot 1 000 000 polygoner, och ett träd utgörs av cirka 15 000 polygoner.

- **Animationer**
  Gräset animeras av många täta blad.
  Animerade vattenytor med realtids dynamiska scen reflektioner och defraktioner.

- **Simulering**
  Vattnets fasförändringar simuleras på ett fysiskt korrekt sätt.
  Himmelsfären simuleras med hjälp av raytracing, färgbländning som ger en realistisk atmosfär och tids beroende ljussättning.

4.1.2 Blueberry3D

Blueberry är namnet på ett svenskt program för visualisering och simulering av terräng. Utmärkande för Blueberry är att det bl.a. använder sig av procedurell geometri, i likhet med Codecreatures. Detta ger, enligt skaparna av Blueberry, fördelarna att man får tillgång till obegränsad detaljnivå, ingen repetition av geometrin och optimalt användande av systemresurserna.

Följande är ett utdrag från en artikel av Mattias Widmark [14]:

”Procedurell geometri kan definieras som att geometriska figurer skapas genom ett program. Som exempel kan man ta en sfärritningsrutin. Man matar in centrumpunkten och sfärens tänkta radie, rutinen beräknar sedan en mängd trianglar som beskriver sfärens yta. De här trianglarna kan ses som en instans av procedurell geometri. Då det gäller Blueberry används begreppet procedurell geometri endast i realtid, d.v.s. det görs inga förberäkningar av någon geometri utan allt görs under körning.

Fördelar:

- **Obegränsad upplösning**
  Detaljer ner till en centimeterskala och under kan skapas genom att man använder fraktaler vid beräknandet av en procedurell geometri.

- **Rikedom**
  I praktiken kommer det inte att förekomma någon upprepning, inte ens för stora områden. Detta genom att använda en kontrollerad slumpmässighet i proceduren.

- **Naturlig terräng**
  Den obegränsade upplösningen, fraktaler och slumpmässiga rikedomen gör att det blir möjligt att skapa naturliga områden, som verkligen blir levande, på ett helt annat sätt än traditionella tekniker.

- **Minskad tid för design**
  Om man skulle göra allt för hand, blir de små detaljerna i en terrängmodell de dyrbaraste att designa. När en procedurell geometri används, kan stora delar av detaljerna abstraheras genom att små procedurer använder sig av kontrollerad slumpmässighet.

- **Kompakt databas**
  Storleken på databasen kan hållas till ett minimum genom att det mesta av dess innehåll lagras implicit i programmet självt.

- **Optimalt användande av systemet**
  En terrängmotor som använder procedurell geometri kan enkelt justera detaljrikedomen med avseende till hårdvaran, då det inte finns någon gräns för mängden av detaljer som finns tillgängliga.”

Forts. på citat från föregående sida.

”Nackdelar:

- **CPU-intensivt**
  Procedurell geometri använder sig av datorns CPU intensivt. Men hur som helst, är ändå dagens standarddatorer kraftfulla nog att skapa en variations- och vegetationsrik terräng med hjälp av procedurell geometri.

- **Systemminne**
  Procedurell geometri kräver en hel del systemminne. Detta beror till största delen på det faktum att fler detaljer än förut kan skapas. Men också på att de genererade figurerna behöver tillgång till extra tillståndsvariabler, som används vid genereringen. Dessa variabler hade inte behövts om figurerna hade blivit förskapade. Men i praktiken är 256 MB minne tillräckligt.

- **Detaljnivå**
  Procedurell geometri kräver en avancerad detaljnivåsbehandling. All geometri i en terräng modell kan inte genereras och användas på en gång. Det skulle belasta vilket hårdvarusystem som helst, för mycket. Istället, behöver man ett schema som talar om var en hög detaljnivå behövs och var den inte behövs. Ett relaterat problem till detta är diskontinuitet då man skiftar från en hög detaljnivå till en lägre, med så lite visuell störning som möjligt. Detaljnivåsbehandlingsproblemet kan beskrivas och diskuteras på många olika sätt, men det finns ännu så länge ingen perfekt lösning.”

Några av de geometrigenereringsfunktioner som BlueBerry tillhandahåller är:

- **Höjdkarta**
  En geometri för en höjdkarta skapas genom en variation av tvådimensionell *”Fractal Brownian Motion”*. Grovleken på terrängytan bestäms per terrängklass.

- **Markstruktur**
  För att åstadkomma en realistisk markstruktur, används flera olika lager. Normalt projiceras ett marklager i Blueberry3D, till ett riktigt lager, som jord, sten eller sand. Lagret tillhandahåller även en mängd parametrar som beskriver egenskaper som erosion, grovhet och bördighet. Varje terrängklass kan ha sin egen uppsättning av marklager.

  Vid generering av markstruktursgeometrin tas alla de lokala marklagren och dess parametrar med i beräkningen. Många naturliga effekter som att stenar sticker upp ur jorden runt omkring dem och att havsvågor sveper iväg gräs och smuts, lämnande kvar endast sand, kan fås genom att noggrant definiera marklagren.

- **Distribuering av vegetationen**
  Distribueringen av vegetationsföremål utförs av en annan variation av *”Fractal Browninan Motion”*. Effekten av ett naturligt utseende hos blad och grenverk på buskar och träd, skapas automatiskt. Markens bördighet tas även med i beräkningen.

- **Vegetationsföremål**
  Vegetations föremål, som individuella träd och buskar, modelleras genom användandet av *”Iterated Function Systems”*. En specifik art skapas med hjälp av Bluberry3D:s interaktiva trädeditor, och unika instanser av arten genereras sedan med hjälp av en motsvarande geometrigenerator och en kontrollerad slumpmässighet.

- **Krökta ytor**
  Bluberry3D använder sig av polygonytor för att modellera vägar, vattendrag, vägar, stigar, shackt, hus, fundament etc. Den huvudsakliga idén med detta är att användaren tillhandahåller en minimal mängd information för att sedan låta systemet ta hand om alla detaljer för själva modelleringen. Som ett exempel på detta, kan en väg modelleras som en mängd punkter i terrängen och en profil. Generatorn för krökta ytor interpolerar punkterna, och anpassar terrängen runt omkring så att vägen smälter in, för att sedan till slut placera vägen.

  Krökta ytor, liksom all annan procedurell geometri, beräknas i realtid. Resultatet av detta blir att det inte har någon betydelse på hur nära håll betraktaren observerar ytan, den kommer ändå att se bra ut.

4.1.3 Serious Sam: The Second Encounter

Serious Sam: The Second Encounter är namnet på ett datorspel i genren actionspel. Spelet går kortfattat ut på att man med diverse vapen till sin hjälp, skall försöka utrota så många varelser som möjligt. Men spelet uppvisar en mycket god detaljrikedom när det gäller utomhusscenerna. Här kan man t.ex. se träd som vajar i vinden, gräsbuskar, och annan vegetation.

Men om man skall jämföra detta spels terrängmotor med BlueBerry3D:s terrängmotor så finns det stora skillnader. En av de största skillnaderna gäller träden. I BlueBerry3D byggs dessa upp av en stomme av polygoner för att sedan byggas vidare ner på detaljnivå med hjälp av fraktaltekniker. Om man studerar träden noggrant i Serious Sam ser man att även här byggs träden upp av en polygonstomme, men man har valt att sätta ihop grenar och trädkronan med hjälp av raka skivor med pålagd bladtextur, billboards.

Vidare kan man i Serious Sam se att trädens grenar rör sig. Detta ser bra ut på lite avstånd, men om man betraktar dem en stund ser man att själva grenarna inte har någon fysisk förbindelse med trädstammen, utan att de bara är påklistrade på trädet. Det yttersta av grenen är en animation av hur en riktig trädgren skulle röra sig om det blåste.

Då det gäller BlueBerry3D:s träd vajar dessa faktiskt i vinden på riktigt, d.v.s. där använder man sig av en fysisk modell som genererar beteendet istället för en förskapad animation. Härmed inte sagt att det ena skulle vara bättre eller sämre än det andra. Då det gäller Serious Sam kan man säga att huvudsyftet med detta spel inte är att ge spelaren upplevelsen av en så naturtrogen terräng som möjligt, utan mer att ge spelaren en häftig spelupplevelse. BlueBerry3D är kan då mer ses som en ren terrängmotor där man specialiserat sig just på det visuella och inte så mycket på saker runt omkring, som fiender, artificiell intelligens m.m. Serious Sam ger enligt vårt tycke en god ram, för vad man kan åstadkomma med hjälp av en modern spelmotor, då det gäller visualisering av terräng.

Nedan följer ett utdrag av vad spelmotorn Serious Sam tillhandahåller för möjligheter:

- **Komplexa arkitekturer**
  Serious Sam motorn kan rendera stora områden och komplexa arkitekturer. Stora områden kan vara en terrängmiljö men även en inomhusmiljö, med olika rum. Som exempel på en komplex arkitektur, kan man ta rekonstruktionen av ett gammalt egyptiskt tempel som förekommer i spelet.

- **Realistisk rendering**
  Serious Sam motorn har ett kraftfullt ljussättningssystem och avancerade texturerings möjligheter. Vilket ger en realistisk bild, till spelaren. Motorn klarar av att rendera ljusa miljöer såväl som mörka med en dovhet i atmosfären.

- **Volumetrisk dimma**
  En av möjligheterna med motorn är att lägga på olika lager med dimma i ett rum eller i en dalsänka, som morgondis. Dimman kan dessutom förändras, flyttas, roteras, ändra densitet eller ändra färg, i realtid.

- **Dis**
  Dis är ett naturfenomen som orsakas av partiklar som reflekteras i atmosfären. Användandet av dis i Serious Sam bidrar till att öka känslan av realism, i framförallt utomhus miljöerna.

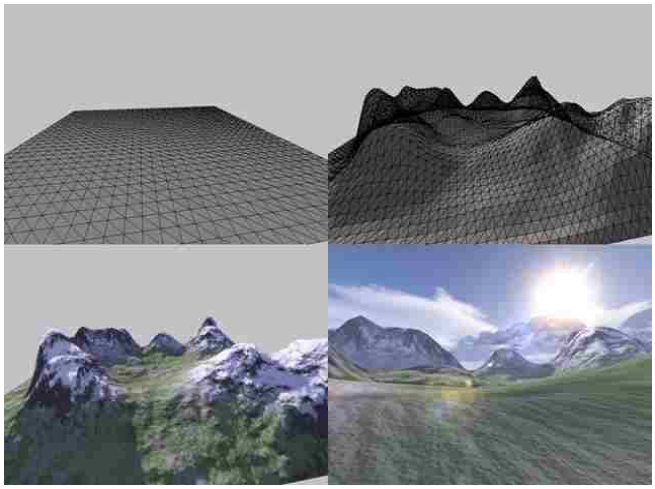
- **Objekt LOD**
  Serious Sam motorn har en avancerad LOD-algoritm, som klarar av att hantera upp till 32 detaljnivåer per objekt. Detta gör att man säger sig kunna hantera hundratals eller till och med tusentals objekt, och ändå tillhandahålla en godtagbar bildfrekvens.

- **Terränggeneration**
  Terräng är en av primitiverna som kan användas för att bygga upp en värld i Serious Sam. Terräng kan byggas upp manuellt eller bli uppbyggd med hjälp av en höjdkarta. När den sedan blir texturerad med en lämplig texturmap så får man en trolig och vacker utomhusmiljö.

- **Godtyckliga bakgrunder**
  Det finns stöd för att lägga in bakgrunder i form av en skydome.

- **Linsreflektioner**
  Detta är egentligen en effekt som orsakas av filmkameror eller fotokameror som har multipla linser. Effekten gör så att betraktaren får uppfattningen att han/hon tittar mot en stor och stark ljuskälla. Effekten framträder vanligtvis som en serie av cirklar och ringar i spektralfärger som framkommer från en stark ljuskälla, och ger betraktaren en känsla av att titta mot solen.

- **Dag-natt skiftningar**
  Genom att modifiera ljusfärger i realtid, låta solen vandra över himmelen och ändra bakgrunden, kan Serious Sam ge en övertygande känsla av att natten faller på. Dagsljusets atmosfär kan mjukt övergå till en gyllene solnedgång för att till slut övergå i månljus.



4.1.4 Crystal Space 094002

Olika möjligheter som Crystal Space tillhandahåller för utomhusmiljöer är:

- Möjlighet att använda transparenta och semitransparenta texturer, vilket tillåter att man kan se ut genom ett fönster eller genom en vattenyta.

- För att återge en realistisk och snygg himmel finns möjligheten att använda en dynamisk gouraudskuggad skydome. Med lite programmering finns möjligheten att ha en sol som rör sig och som faktiskt påverkar färgen på himmelsfären i realtid.

- Möjlighet att dela in himmelsfären i olika lager för animering av moln och andra atmosfäriska effekter.

- Statisk färgad ljussättning med riktiga skuggor. Ljusen och skuggorna beräknas före det att världen visas.

- Dynamisk färgad ljussättning.

- Stöd för 3D-sprites med möjlighet till bildanimation.

- Stöd för 2D-sprites och partikelsystem som använder dessa.

- Djupkorrekt färgad volumetrisk dimma i sektorer. För detta finns stöd i både mjukvarurenderaren och hårdvarurenderaren.

- Det finns möjlighet att lägga en ljusring kring ljusen för att skapa atmosfäriska effekter. Här finns även möjlighet att skapa solblänk, linsreflektioner.

Vidare finns det en landskapsmotor som tillhandahåller en mängd olika möjligheter. Den har stöd för fyra olika statiska LOD-nivåer och en riktad dynamisk ljuskälla. I framtiden skall det även finnas stöd för dynamiska skuggor, men detta är i version 0.94 av Crystal Space fortfarande ett att-göra moment. Den grundläggande idén bakom landskapsmotorn är att hela terrängen delas in i ett antal block. Vanligtvis är detta åtta gånger åtta block, men användaren ges möjligheten att själv ange detta. Indelningen av terräng till block är viktig av tre olika anledningar. Den första anledningen är att valet av LOD-nivå sker på blockvis, d.v.s. varje block har sin egen LOD-nivå oberoende av vad dess grannar har för nivå. Den andra anledningen är att visualiseringsklippningen sker även den blockvis. Den tredje och sista anledningen är att varje block kan ha en egen textur, vilket är viktigt för att hårdvaran ofta har en begränsning för hur stor en textur får vara. Men genom att blocken kan ha separata texturer kan man dela upp en väldig stor textur över hela landskapet.

Varje block är vidare indelat i ett rutnät. Till exempel med ett rutnät bestående av 16 x 16 rutor kommer varje block bestå av 16 x 16 kvadrater eller 16 x 16 x 2 trianglar på den högsta detalj nivån. Kombinationen av block och rutnät kontrollerar hur många trianglar hela terrängen kommer att bestå av. Landskapsmotorn har stöd för fyra LOD-nivåer, vilka kan användas oberoende av varandra. Nivån noll är mest detaljerad. Det finns stöd för användaren

att ange på vilket avstånd motorn skall byta LOD-nivå. Användaren kan även ställa in kvalitén på varje LOD-nivå.

Landskapsmotorn har en visualiseringsalgoritm som är baserad på quad-tree. Algoritmen klipper bort geometri på blocknivå, vilket innebär att antingen syns hela blocket eller inte.

För att bygga upp landskapet anger man en höjdkarta, en gråskalebitmap, motorn tolkar sedan denna med hjälp av en höjdfunktion. Motorn kan även beräkna landskapets normaler automatiskt, men det finns även stöd för att själv ange en normalfunktion för landskapet.

Crystal Space har också en textureringsrutin som kan generera realistiska terrängtexturer givet ett antal bastexturer och en höjdkarta.

4.2 Sammanfattning

Om man ser till de rent tekniska detaljerna hos de beskrivna terrängmotorerna är Code Creatures den terrängmotor som är mest tekniskt avancerad. Detta på grund av att de använt sig av det relativt nya stöd, som finns i dagens grafikkort, för vertex- och pixelshaders. CodeCreatures uppvisar även andra avancerade tekniker i form av bl.a. snabba renderingstekniker, som delvis är skrivna i ren maskinkod. När det gäller Blueberry använder den sig inte av vertex- och pixelshaders. En väsentlig teknisk detalj hos Blueberry är däremot det stöd som finns för procedurell geometri. Detta gör att Blueberry blir väldigt skalbart och anpassningsbart för visualisering av olika miljöer. Vidare finns det i Blueberry implementerat ett stöd för att visa billboardsbilder i bakgrunden och polygonobjekt i förgrunden.

Motorn, som ligger till grund för Serious Sam, uppvisar kanske inte så tekniskt avancerade möjligheter för visualiserandet av en utomhusmiljö. Men då den är mer avsedd för att vara flexibel och klara olika typer av tekniker, är den ändå bra på det den gör. Till sist kan man väl säga att Crystal Space innehåller de flesta grundläggande teknikerna för att visa både inomhusmiljöer och utomhusmiljöer, men den är inte riktigt så tekniskt avancerad, som de övriga. Den är däremot väldigt flexibel i den mening att den kan anpassas till i stort sett vilken plattform som helst. Den ställer heller inte så höga krav på hårdvaran då den bland annat innehåller en mjukvarurenderare.

5. Implementation: Design och Strategi

Den implementation vi valt att göra i vårt arbete, har utmynnat i ett plugin för spelmotorn Crystal Space. Vi skall i detta kapitel ge en överblick över den implementation vi har gjort och i det nästkommande kapitlet jämföra de olika resultat vi har kommit fram till.

5.1 Motivering och ambitionsnivå

I Crystal Space har det sedan några versioner tillbaka funnits stöd för att generera och visa en terrängyta, beskriven i kapitel 4.1.4. För att rita upp och visa ytan har man till sin hjälp ett plugin som i Crystal Space heter terrfunc. Då vi med vårt arbete ville skapa och undersöka olika tekniker för att visa vegetation, upptäckte vi att det var svårt att placera ut objekt på terrängytan på ett bra sätt. Detta ansåg vi vara en brist, och därför utvecklade vi en metod för att lättare kunna placera ut en större mängd objekt på en terrängyta i Crystal Space.

Det stöd som finns för visning och utplacering av ett objekt på en terrängyta, är att man i en skriptfil får ange en beskrivning av det objekt man ville placera ut, se kapitel 2.1. Vill man placera ut flera objekt av samma typ får man i skriptfilen ange dessa ett och ett, som ett separat meshobject. Ett exempel på hur detta kan se ut i en skriptfil är följande:

```
MESHOBJ 'treeObj_1' (
      PLUGIN ('spr3d')
      PARAMS (
        FACTORY ('treeFact')
        ACTION('default')
        MIXMODE (KEYCOLOR(0,0,0))
        BASECOLOR (1,1,1)
      )
      MOVE (V (0,74,50))
      ZTEST ()
      PRIORITY ('tree')
)
MESHOBJ 'treeObj_2' (
      PLUGIN ('spr3d')
      PARAMS (
        FACTORY ('treeFact')
        ACTION('default')
        MIXMODE (KEYCOLOR(0,0,0))
        BASECOLOR (1,1,1)
      )
      MOVE (V (10,75,50))
      ZTEST ()
      PRIORITY ('tree')
)
...
o.s.v.
```

För varje objekt får man tala om var det skall vara och hur det skall vara placerat. Detta blir ganska omständligt om man som i vårt fall ville placera ut en större mängd växter för skapandet av en naturmiljö. Vi bestämde oss för att behålla den redan befintliga beskrivningsrepresentationen av en meshfactory (se kapitel 2.1). Men istället för att skriva in denna i en skriptfil implementera stöd för att kunna ange den som en enskild fil till vårt plugin istället. Ett exempel på detta är följande metod:

```
void csVegFuncSystem::Load_objects( csVector2* upper_left_vertex,
     csVector2* lower_right_vertex, int nmb_of_species,
     float min_dist, char* name, char* path );
```

Detta skulle innebära att metoden vi ville implementera skulle kunna ladda in en godtycklig beskrivningsfil av ett objekt, tolka denna, och placera ut objektet i världen. I metoden i exemplet ovan finnas stöd för användaren att ange hur många objekt som skall placeras ut, inom vilket område de skall placeras ut och minsta avstånd emellan dem. Här skulle inte användaren heller behöva tänka på y-koordinaten, utan endast ange hörnpunkterna till området i form av x- och z-koordinater. Vad som mer var önskvärt, var att låta det finnas en slumpmässighet i utplaceringen liksom i hur objekten var vända.

Siktsträckan i en utomhusmiljö är ofta mycket lång, vilket gör att betraktare som rör sig i miljön har möjlighet att se objekt på ett långt avstånd ifrån dem. Detta gör att det hela tiden är mycket information som behöver bearbetas och renderas i en utomhusmiljö. För att dra ner något på denna informationsmängd kan man låta terrängmotorn representera de objekt långt ifrån betraktaren med enkla billboardsbilder, istället för ett detaljerat objekt. Detta blir en form av LOD-teknik, som kraftigt kan reducera informationsmängden och öka bildfrekvensen.

Ovanstående resonemang och det faktum att vi märkte en sjunkande bildfrekvens, vid utplacerandet av endast ett polygonobjekt i världen, gjorde att vi bestämde oss att bidra med en teknik för bländning mellan polygonobjekt och billboardsbilder. Crystal Space skulle på så sätt utökas med ytterligare en viktig funktion för visualisering av en utomhusmiljö.

Vi valde att i vår implementation av bländningen låta användaren ange både billboardsbilden och polygonmodellen av objektet. Detta gör att det kan vara svårt att få en helt perfekt övergång där inga störningar syns. Men är man bara noggrann vid skapandet av billboardsbilden och inställningen då övergångarna skall ske märker man inte mycket av själva bländningsförfarandet. Det man helst hade sett var att man vid övergången från polygonobjekt till billboardsrepresentationen, hade låtit Crystal Space ta en bild av polygonobjektet och sedan använda denna för bländningsförfarandet. Vi började med att undersöka om detta skulle vara möjligt, men kom efter en tid fram till att det skulle vara svårt att lokalisera enbart polygonobjektet i scenen som skulle renderas. Vilket skulle vara nödvändigt då kameran kanske är i rörelse under själva övergången och en billboardsbild innehållande delar av bakgrunden, berg, moln m.m. inte skulle bli naturlig att övergå till. Den metod vi implementerat kan säkerligen finslipas ytterligare, men det är en enkel och effektiv metod, med en stor fördel jämfört med att växla mellan olika representationer.

Ambitionsnivån var att vår implementation av pluginet skulle kunna visualisera och placera ut vegetation av minst lika god kvalitet som i Serious Sam: The Second Encounter, men helst i nivå med Codecreatures och BlueBerry3D. Våra ambitioner fick dock anpassas till de ramar som Crystal Space ställde i form av bristande dokumentation och därav följande svåröverskådlighet.

Med tanke på den ambitionsnivå vi strävade efter fanns det andra saker man skulle kunna tänka sig att implementera som tillägg till Crystal Space. Med hänsyn tagen till Bluberry och Codecreatures vore en hantering av kollisioner med vegetationen önskvärd. Med detta menas att om betraktaren gick in i en buske eller ett träd, skulle denna ge en respons tillbaka till betraktaren på något sätt. Busken skulle t.ex. kunna animeras så att det såg ut som den delade på sig, medan trädet gjorde så att betraktaren studsade tillbaka eller helt enkelt stannade upp. Men för att införa en hantering av beskrivna kollisioner, skulle det behövas ett tillägg till det redan befintliga plugin för kollisionshantering.

Vidare skulle ett system för hantering av hur ett träd fysiskt rör sig i vinden eller hur en gräsmatta fysiskt rör sig då man låter en boll rulla över den, kunna ses som ett bra tillägg till Crystal Space. Liknande de system som finns i Blueberry och Codecreatures, men på motsvarande sätt skulle även denna hantering kräva ett tillägg till ett befintligt plugin.

Man kan dock med vårt plugin placera ut ett träd, en buske eller en gräsmatta. Då det finns stöd för animering av objekt i Crystal Space (se kapitel 4.1.4), kan man få det att se ut som trädet, busken eller gräsmattan rör sig i vinden. Vid våra försök att använda pluginet för kollisonshantering har detta visats sig vara dåligt integrerat med terrfunc-pluginet.

Vi anser till sist att möjligheterna att visa en mer komplett utomhusmiljö med växter och träd, och att göra detta med en god detaljrikedom och bildfrekvens har förbättrats avsevärt i Crystal Space genom vårt plugin.

5.2 Funktionalitet hos de implementerade teknikerna

Vi skall i detta kapitel ge en översikt av funktionerna hos vårt plugin i en typisk terrängapplikation. Vi kommer sedan i underkapitlen gå in lite mer specifikt på hur dessa tekniskt implementerats.

Vi har implementerat vårt plugin med hänsyn tagen till att pluginet för terrängytan redan är laddat. Då pluginet för terrängytan är laddat, höjdkartan är beräknad o.s.v., kan vårt plugin användas för att placera ut terrängobjekten och kontrollera dessa. Vi förutsätter att det finns en kamera, ett laddningsplugin för inladdning av grafiska objekt o.s.v.

Terrängobjekten kan ha två olika beteenden påkopplade. Det första beteendet är en inställning om användaren vill att ett objekt skall följa med betraktarens rörelse runt objektet. Användaren ges här möjlighet att ange inom vilket avstånd kameran behöver befinna sig för att objektet skall börja rotera. Anges inget avstånd, finns det ett default avstånd som används. Detta är tänkt i första hand för en billboard i form av en enkel bildskiva, men kan användas för ett godtyckligt objekt.

Det andra beteendet är en inställning gällande övergången mellan två objekt. Här finns det möjlighet att ange inom vilket avstånd kameran behöver befinna sig för att påverka objektet. Men här anger man inte bara på vilket avstånd själva övergången skall ske när man närmar sig objektet utan även på vilket avstånd tillbakagången skall ske, då kameran förflyttar sig från objektet. Även här finns det ett default avstånd som automatiskt används om användaren avstår från att ange några egna.

Beträffande själva utplaceringen av terrängobjekt på landskapsytan så bygger denna på förutsättningarna :

Att man har laddat vårt plugin med de objekttyper som skall placeras ut, angett inom vilket område på landskapsytan detta ska göras, satt minsta inbördes avstånd mellan dem och antal att placera ut, samt angett vilket beteende de skall uppvisa.

Utplaceringen görs sedan i samband med att den första bildrutan renderas, och därefter är objekten en del av själva terrängapplikationen. Positioneringen vid utplaceringen inom det angivna området görs så att alla objekt, som ska placeras ut, slumpmässigt tilldelas en plats inom detta med minst det angivna minimiavståndet mellan sig. Syftet med den slumpvisa positioneringen är att den utplacerade mängden ska likna en naturlig förekommande sådan.

5.3 Ett plugin för visualisering av vegetation i Crystal Space

Ett plugin är ett instickspogram till ett annat program. Detta innebär att pluginet är en del av ett, ofta lite större program. Fördelen med detta är att man kan programmera pluginet för sig och sedan lägga det till huvudprogrammet, när det är färdigt. En förutsättning för att pluginet skall kunna prata med sitt huvudprogram är att de använder sig av ett specifikt protokoll. I Crystal Space heter detta system: SCF – Shared Class Facility. Ett annat exempel på ett liknande system är Microsofts COM teknik. Tidigare versioner av Crystal Space använde sig av COM teknik, men då det uppstod allt för mycket buggar, valde man att implementera ett eget system, SCF.

Det som SCF-systemet huvudsakligen bygger på är gränssnitt (eng. interface). Man definierar ett gränssnitt genom ett antal abstrakta metoder som man vill få tillgång till inom ett objekt. I praktiken utgörs SCF-gränssnitt av C++ strukturer eller klasser. Ett exempel på hur ett sådant abstrakt gränssnitt kan se ut är följande, som är hämtat från vår egen implementation:

```
struct iVegFuncSystem : public iBase
{
virtual void Contact() = 0;
virtual void Init_VegFunc(iLoader* loader, iGraphics3D* g3d,
                          iView *view) = 0;
virtual void Place_Objects() = 0;
virtual void Blending() = 0;
virtual void Billboard_Rotation () = 0;
……
};
```

Ovanstående exempel är hämtat från en fil, vars namn inklusive sökväg är ”ivaria/vegfunc.h”. Själva implementationen av ovanstående gränssnitt görs i en annan fil som vi valt att döpa till ”csvegfunc.h” och ”csvegfunc.cpp”. I både csvegfunc.h och det applikationsprogram som skall använda pluginet, inkluderas det abstrakta gränssnittet med raden:

```
include# ”ivaria/vegfunc.h”
```

I applikationsprogrammets headerfil skapar man en struktur och en pekare till det plugin som man vill ladda in till programmet. Detta ser ut som följer:

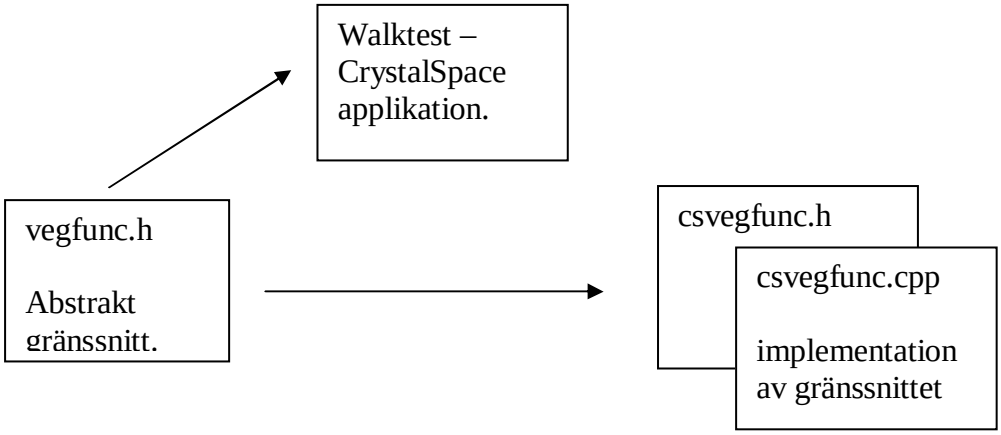
```
struct iVegFuncSystem;

iVegFuncSystem* vegfunc_system;
```

För att sedan ladda pluginet för användning i applikationsprogrammet, kan följande rader användas:

```
p = cfg->GetStr("Walktest.Settings.VegFuncPlugin",
                "crystalspace.vegetationplanter.vegfunc");

vegfunc_system = CS_LOAD_PLUGIN (plugin_mgr, p, iVegFuncSystem);

if (!vegfunc_system)
{
    Report (CS_REPORTER_SEVERITY_ERROR, "Sorry, No Vegfunc plugin
            found!");
         return false;
}
else
{
    Report (CS_REPORTER_SEVERITY_ERROR, "Yes, a Vegfunc plugin
            found!");
    vegfunc_system->Contact();
    vegfunc_system->Init_VegFunc(LevelLoader,myG3D, view);
}
```

Ovanstående är bara ett av flera sätt att ladda in ett plugin till en applikation i Crystal Space. Det är också det sätt som vi har använt oss av.

Då SCF-systemet är komplext och innehåller flera olika funktioner, bl.a. kan man låta olika gränssnitt ärva varandra, anser vi oss inte ha möjlighet att inom ramen för vårt arbete redogöra i detalj för dessa funktioner, då vi inte själva använder oss av några av dem. Till slut exemplifierar vi SCF strukturen med en bild över hur vårt plugin kan användas tillsammans med en applikation för Crystal Space, figur nr. 28.



Figur nr. 28: Schematisk bild över strukturen hos ett SCF-baserat plugin

5.3.1 Utplaceringsteknik

Här beskrivs den teknik vi utvecklat för att slumpmässigt placera ut ett givet antal vegetationsobjekt, med ett viss givet minimumavstånd inbördes, inom ett visst givet rektangulärt område på en landskapsyta.

Varje utplacerat objekt ges även en slumpmässigt vald vridningsvinkel relativt kamerans position. Den metod, som beräknar utplaceringspositioner i planet och vridningsvinklar, tar som indata:

- Områdets bortre vänstra hörns koordinater i XZ-planet = (xbv, zbv)
- Områdets främre högra hörns koordinater i XZ-planet = (xfh, zfh)
- Det givna antalet objekt att placera ut = nmbofs stycken
- Minimiavståndet inbördes mellan de utplacerade objekten = md

Först i metoden görs en kontroll av att nmbofs inte överskrider maximala antalet objekt = maxobj, som går att placera ut inom det givna områdets yta förutsatt det givna minimiavståndet md mellan dem.

Om nmbofs > maxobj så sättes nmbofs = maxobj innebärande att endast maxobj stycken objekt placeras ut.

Metoden returnerar som resultat en vektor vegdata innehållande nmbofs stycken structer (C++-terminologi), vilka var och en innehåller X- resp. Z-koordinaten för en utplacerings-

punkt samt en vridningsvinkel = v. Dessa taltripplar beräknas i en följd med hjälp av slumptalsgeneratorn rand( ) (i C++-standardbibliotek). Som frö till slumptalsgeneratorn ges inför beräknandet av trippelföljden värdet unsigned int curr = (unsigned) time (NULL) (time( ) är C++´s realtidsklockfunktion). Den nämnda taltrippel-structen är definierad :

```
struct randomdata {float x; float z; float v;};
```

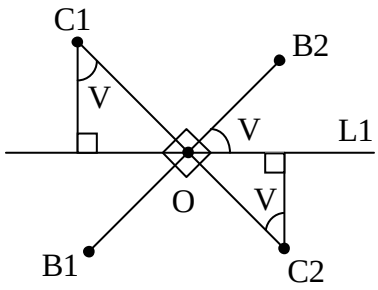
Värdena x, z och v för de nmbofs structelementen i vektorn vegdata beräknas med följande med C++-kod :

```
float x0 = xbv; float z0 = zbv;  float fmax = (float) RAND_MAX;
float xd = xfh - xbv; float zd = zfh - zbv;
float xc = 0.0f; float zc = 0.0f;
// RAND_MAX är ett implementationsdefinierat stort positivt heltal

vegdata[0].x = x0 + ( ( (float) rand( ) / f_max )*xd ) ;
vegdata[0].z = z0 + ( ( (float) rand( ) / f_max )*zd ) ;
vegdata[0].v = ( ( (float) rand( ) / f_max )*2*PI ) ;

for ( int i = 1 ; i < nmbofs ; i ++ ) {

      xc = x0 + ( ( (float) rand( ) / f_max )*xd ) ;
      zc = z0 + ( ( (float) rand( ) / f_max )*zd ) ;

      for ( int j = 0 ; j < i ; j ++ ) {
      // kontroll av att avståndet från punkten (xc , zc) till alla
      // tidigare beräknade placeringspunkter är >= md

        if ( abs( vegdata[ j ].x - xc ) < md && abs( vegdata[ j ].z
             - zc ) < md ) {
          while ( abs( vegdata[ j ].x - xc ) < md && abs(
                  vegdata[j].z - zc ) < md ) {

                  xc = x0 + ( ( (float) rand( ) / f_max )*xd ) ;
                  zc = z0 + ( ( (float) rand( ) / f_max )*zd ) ;

          } // slut på while-satsen

        j = -1;   // avståndskontroll igen som ovan för den i while-
                  // satsen nyberäknade punkten ( xc , zc )

        } // slut på if-sats

      } // slut på inre for-sats

     vegdata[ i ].x = xc ;  vegdata[ i ].z = zc ;
     vegdata[ i ].v = ( ( (float) rand( ) / f_max )*2*PI ) ;

} slut på yttre for-sats

return vegdata ;   //  Härmed avslutas metoden.
```

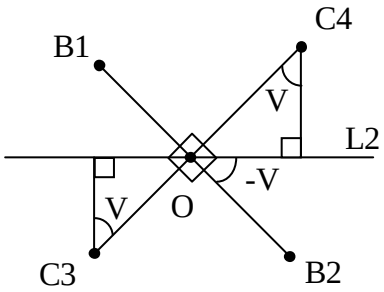
Själva utplaceringen av objekten på landskapsytan hanteras av en annan metod varvid höjdvärdena för placeringspositionerna i planet beräknas.

5.3.2 Rotationsteknik

Tekniken vi utvecklat för rotation av en billboard i form av en enkel bildskiva förklaras med hjälp av figurerna nr. 29a och 29b nedan.



Figur nr. 29a: Kamera- och billboardspositioner



Figur nr. 29b: Kamera- och billboardspositioner

Linjerna L1 och L2 är parallella med X-axeln i koordinatsystemet som innehåller figurerna 29a och 29b. Systemet har en Z-axel i samma plan som X-axeln och vinkelrät mot denna samt en Y-axel vinkelrät mot XZ-planet. Linjen B1, B2 med mittpunkten O är ovankanten på en billboard, i form av en enkel bildskiva, sedd uppfrån längs en linje parallell med systemets Y-axel. Punkten O har koordinaterna (ox,oz) i XZ-planet. Punkterna C1, C2, C3 och C4 med koordinaterna (cx1,cz1), (cx2,cz2), (cx3,cz3) och (cx4,cz4) är fyra kamerapositioner.

Bildskivan ska, enligt vår implementation, vrida sig kring en axel, som går genom mittpunkten O och är parallell med Y-axeln, så att B1, B2 alltid är vinkelrät mot linjen som går genom O och kamerans position i XZ-planet. P.g.a. koordinatsystemets orientering är skivan vriden positivt v radianer i figur nr. 29a och negativt -v radianer i figur nr. 29b. Av likformighetsskäl inses direkt att vinklarna markerade med v i figur nr. 29a är lika samt att vinklarna markerade med v resp. -v i figur nr. 29b är lika till sina absolutbelopp. Bildskivans vridningsvinkel beräknas med följande C++-kod i vår rotationsimplementation :

```
float delta_x = camera_origin.x - billboard_origin.x ;
float delta_z = camera_origin.z - billboard_origin.z ;
float alpha = 0 ;
if ( delta_z != 0 ) alpha = atan2f(delta_x, delta_z) ;
else if ( delta_x > 0 ) alpha = - PI / 2 ;
else if ( delta_x < 0 ) alpha =  PI / 2 ;
```

Vridningsvinkeln blir = - alpha, vilken tas som parameter i en annan funktion, som sedan anropas för att ställa bildskivan i denna vinkel. Variablerna camera_origin.x och camera_origin.z är kamerapositionens X- resp. Z-koordinater. Variablerna billboard_origin.x och billboard_origin.z är punkten O´s ovan X- resp. Z-koordinater.

Funktionen atan2f(float d_x, float d_z) = arctan(d_x / d_z).

I figuren 29a är vridningsvinkeln v radianer lika med:

```
- arctan( (cx1 - ox) / (cz1 - oz ) ) = - arctan( (cx2 - ox) / (cz2 - oz ) )
```

för kamerapositionerna C1 och C2.

I figuren 29b är vridningsvinkeln -v radianer lika med:

```
- arctan( (cx3 - ox) / (cz3 - oz ) ) = - arctan( (cx4 - ox) / (cz4 - oz ) )
```

för kamerapositionerna C3 och C4. I fallet (delta_x > 0) befinner sig kameran till höger om O och i fallet (delta_x < 0) till vänster om O.

5.3.3 Beskrivning av bländningsförfarandet

Här skall vi beskriva den metod vi implementerat för att gradvis övergå från en billboard, i form av en enkel bildskiva, till ett polygonobjekt. Vi kommer här hela tiden utgå från att övergången sker mellan en sådan billboardsrepresentation av ett terrängobjekt och dess polygonrepresentation.

Vi börjar med är att kontrollera avståndet till objektet i fråga. Är avståndet mindre än det minimiavstånd, som ställts in, börjar den gradvisa övergången ske genom att vi placerar ut en osynlig representation av polygonobjektet, på exakt samma ställe och med samma vridningsvinkel som bildskivan. Med osynlig menas att vi använder alfa-kanalen för att göra polygonobjektet genomskinligt. Sedan sker den gradvisa övergången genom att vi för varje bild gör polygonobjektet mindre genomskinligt och billboardsobjektet mer genomskinligt. Detta innebär att när övergången är färdig har billboarden försvunnit och polygonobjeket visas i dess ställe.

Vi har en motsvarande kontroll ifall kameran rör sig ifrån ett polygonobjekt. Då sker också en gradvis övergång, men i detta fall kommer övergången att ske mellan polygonobjektet och billboarden. Avståndet då denna övergång sker är ej detsamma som när övergången mellan billboarden och polygonobjektet sker, utan kan anges oberoende av minimiavståndet.

De olika variablerna för att kontrollera avstånden är lokala för varje art. Med detta menas att vi har byggt upp en datastruktur som dels håller reda på i vilket läge ett visst objekt befinner sig i sin gradvisa övergång, och dels på vilket avstånd övergångarna skall ske. Man kan på detta sätt ange att t.ex. vid visandet av en björk skall övergången ske på ett längre avstånd än vid visandet av en ek.

Att vi valt att kunna ange två olika avstånd för övergång och tillbakagång har sin förklaring i att värdet av hela bländningsförfarandet ligger i att man på långt håll endast har en billboardsrepresentation av ett objekt. Men även då man inte ser objektet ifråga. Med detta menas att om man tänker sig att man närmar sig ett träd, övergången sker på ett förhållandevis långt avstånd, och man sedan vänder sig om och går ifrån det igen, då kan övergången tillbaka till billboardsrepresentation ske på ett kortare avstånd från objektet då detta ändå inte kommer att synas.

6. Tester och jämförelser

I detta avsnitt jämför vi representationerna billboards och polygonmodell samt testar dem med avseende på visuell kvalitet och bildfrekvens. Vi testar även vår teknik för en gradvis övergång mellan dem. De resultat vi har kommit fram till redovisas i de följande underavsnitten och diskuteras avslutningsvis i det sista av dem.

6.1 Val av tekniker och begränsningar i testförfarandet

När det gäller att visualisera växtlighet i form av billboards, finns det, som nämndes i tidigare kapitel, ett antal olika sätt att göra detta på. De sätt som vi har valt att koncentrera oss på är för det första att bara ha en enkel bildskiva, som vrider sig med framsidan vänd mot betraktaren då denne rör sig. De övriga teknikerna kan ses som en och samma, men skiljer sig ändå från varandra på ett markant sätt. Den teknik de bygger på är att man först placerar två bildskivor i ett kors, sedan fyra och även sex bildskivor. Varför vi har valt att koncentrera oss på billboardstekniker är för att dessa spelar en viktig roll i dagens spel- och simuleringsmiljöer.

Vi har även valt att testa en ren polygonteknik i form av polygonträd. Detta för att dels visa på den visuella skillnaden mellan polygonmodeller och billboards, men även för att ge en uppfattning om hur stor skillnaden är mellan dem med avseende på bildfrekvens. Polygonobjekten är i detta fall uppåt begränsade till ett antal av sjutusen trianglar. Denna begränsning har vi funnit rimlig med tanke på bildfrekvensen. Vidare satte vi en begränsning på hur många objekt, som visades samtidigt, till nio stycken, då detta var ett lagom antal att rymmas inom siktvolymen.

Den sista tekniken vi har valt att testa är hur övergången mellan billboards och polygonmodeller ter sig med avseende på visuell märkbarhet och bildfrekvens. Övergången testades under samma förutsättningar som i stycket ovan. Här sattes den visuella begränsningen av bristande kvalitet på billboardsrepresentationen.

Det har avslutningsvis varit svårt att hitta bra bilder och 3d-modeller för användning i våra tester. Detta har inneburit en viss begränsning för testförfarandet, men de modeller och texturer vi använt oss av anser vi vara tillräckliga för att åstadkomma representativa resultat.

6.2 Förväntade testresultat

Beträffande utfallet av våra tester nedan hade vi vissa hypoteser enligt följande :

- Angående jämförelse av olika billboardstyper :

  Endast två bildskivor i kors är inte tillräckligt för att ge ett visuellt naturtroget intryck. Vid rörelse av kameran runt en sådan representation syns skivorna alltför tydligt.

- Angående jämförelse av billboards- med polygonrepresentation :

  På ett visst längre avstånd från kameran syns det inte om ett objekt är representerat med en billboard eller med en polygonmodell av detta.

  En billboardsrepresentation, bestående av ett fåtal trianglar, av ett objekt resulterar i betydligt högre bildfrekvens jämfört med en polygonrepresentation av detta bestående av tusentals trianglar. Storleksordningen på frekvensskillnaden återstår att se vid testningen.

- Angående övergången från billboard till polygonmodell :

  På ett visst längre avstånd från kameran blir övergången knappast visuellt märkbar.

  Bildfrekvensen kommer att avta under övergången från ett högsta värde vid dess början till ett betydligt lägre vid dess slut, men om den avtar monotont, d.v.s. att alla frekvensvärden är lägre än eller lika med sitt närmast föregående, återstår att se vid testningen. Viktigast är dock att bildfrekvensen inte sjunker för mycket under övergången. Den bör inte mer än obetydligt, 1-2 frames, underskrida värdet för polygonrepresentationens vid övergångens slut.

6.3 Utvärdering av billboardstekniker

Utvärderingen bestod av jämförelse av de visuella resultaten vid representation av två olika trädobjekt med:

- En enkel skiva som roterar så att dess framsida alltid är vänd mot kameran
- Två bildskivor i kors
- Fyra bildskivor i kors
- Sex bildskivor i kors

Ett lummigt, se figur nr. 30, och ett glest träd, se figur nr. 31, representerades båda på ovan angivna fyra sätt. De placerades ut i rad på landskapsytan, med den enkla bildskivan till vänster följd av de övriga i tur och ordning. Representationerna jämfördes sedan med avseende på visuellt resultat.

Figur nr. 30: Lummigt träd

Figur nr. 31: Glest träd

Resultat :

Fyrskivsbillboard och enkel roterande bildskiva ger klart bäst resultat för både det lummiga och det glesa trädet jämfört med sex och tvåskivor i kors. Sex skivor ökar lummigheten hos träden för mycket så att de ter sig förändrade. Två skivor räcker inte till för att dölja representationens tvådimensionalitet, ty betraktaren ser lätt skivorna vid rörelse runt träden. Beträffande det glesa trädet ger en enkel roterande skiva bättre resultat än fyrskivsbillboarden, ty dess skivor uppfattas lättare vid rörelse runt trädet än vad som är fallet beträffande det lummiga trädet. Det glesa trädet ter sig dessutom något mer förändrat av fyrskivsrepresentationen jämfört med det lummiga trädet. Slutsatsen blir att en enkel roterande bildskiva alltid är att föredra.

6.4 Jämförelse av polygonrepresentation med billboards

Här redovisas resultaten av jämförelse, med avseende på bildfrekvens och märkbar visuell skillnad, mellan ett objekt visualiserat med en billboard och med en polygonmodell. Jämförelsen utformades enligt följande:

Två tillgängliga träd, en körsbärsbjörk och en ek, valdes som jämförelseobjekt. De jämfördes visualiserade som en enkel roterande bildskiva, fyra bildskivor i kors och polygonobjekt. Beträffande båda träden placerades nio exemplar av varje representationstyp ut på samma slumpmässigt valda platser på landskapsytan och den av Crystal Space angivna bildfrekvensen för vyn sedd både bakifrån och framifrån noterades. Jämförelsen var avsedd att ske i en tänkbar spelmiljö.

Terrängen där träden placerades ut innehöll därför en skydome, solblänk och animation av moln. Båda trädens billboardsrepresentationer hade samma antal trianglar, men ekens polygonmodell hade betydligt färre trianglar än körsbärsbjörkens.

Resultaten av jämförelserna presenteras i tabellen nedan:

Trädtyp	**Representation**	**Antal trianglar**	**Bildfrekvens för vyn sedd framifrån**	**Bildfrekvens för vyn sedd bakifrån**
Körsbärsbjörk	En roterande skiva	2	14.00	10.00
Körsbärsbjörk	Fyra skivor i kors	8	14.00	10.00
Körsbärsbjörk	Polygonmodell	6737	5.70	5.00
Ek	En roterande skiva	2	14.00	11.67
Ek	Fyra skivor i kors	8	14.00	11.67
Ek	Polygonmodell	3892	7.78	7.80

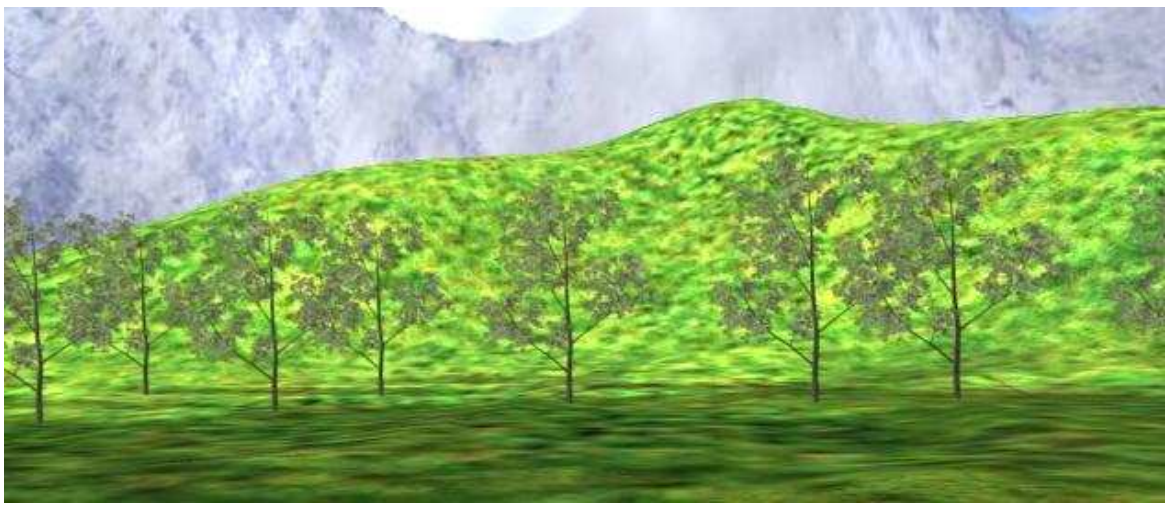
Skillnaderna i bildfrekvens mellan vyn sedd framifrån och bakifrån för samma modelltyp kan förklaras av skillnad i formen på de landskapsytavsnitt, som är synliga i respektive vyperspektiv.

På nära håll syns klart skillnad mellan polygonrepresentationen och samtliga billboardstyper, men på längre håll från kameran märks ingen skillnad. Inom avstånd där skillnad syns märks denna minst beträffande den enkla roterande bildskivan för båda trädtyperna. Vid användandet av polygonmodell istället för billbordsrepresentation sjönk bildfrekvensen med ca. 50-60 %.

6.5 Resultat vid övergång mellan billboard och polygonmodell

Här redovisas undersökningsresultaten av hur bildfrekvensen varierar under övergången mellan en enkel roterande bildskiva och polygonmodell samt även beträffande övergångens synbarhet. Undersökningen gjordes enligt följande:

Samma trädtyper och antal som i avsnitt 6.4 användes. Körsbärsbjörkarna placerades ut som enkla roterande skivor på slumpmässigt valda platser på landskapsytan, se figur nr 32. Därefter åstadkoms en övergång till polygonmodeller genom närmande av kameran till skivorna. Under övergången noterades hur den av Crystal Space angivna bildfrekvensen varierade samt dess högsta och lägsta värde. Samma sak gjordes beträffande eken på samma platser som björkarna. Dessutom företogs nämnda övergångar på kortare och längre avstånd från kameran för att studera deras synbarhet.



Figur nr. 32: Körsbärsbjörkar under övergång

Resultat beträffande körsbärsbjörkarna:

- Vid övergången från billboards till polygonmodeller avtog bildfrekvensen monotont från sitt högsta värde **11.67** till sitt lägsta **4.20,** för polygonrepresentationen vid övergångens slut.

- Nära kameran märks övergången men på längre avstånd blir den knappast märkbar.

Resultat beträffande ekarna :

- Vid övergången från billboards till polygonmodeller avtog bildfrekvensen monotont från sitt högsta värde **11.67** till sitt lägsta **5.83**, för polygonrepresentationen vid övergångens slut.

- Nära kameran märks övergången men på längre avstånd blir den knappast märkbar.

Under övergången visade sig således bildfrekvensen ej understiga värdet för polygonrepresentationens vid dess slut.

6.6 Sammanfattning och diskussion av tester och jämförelser

Utvärderingen av representation med billboards resulterade i att glesa träd bäst representeras med en enkel roterande bildskiva. För lummiga träd går det lika bra att använda fyrskivsalternativet. Enbart två skivor eller fler än fyra ger klart sämre visuella resultat i båda fallen, vilket, beträffande tvåskivsalternativet, bekräftar förväntat resultat.

Jämförelsen av billboards med polygonrepresentation fick, som förväntat, till resultat betydligt högre bildfrekvens vid förstnämnda jämfört med sistnämnda. Skillnaden var mellan 100 - 150 %. Detta bekräftades vid testningen av övergång mellan representationssätten. På långt avstånd från kameran kan, som antagits, träd visualiseras lika bra med billboards som med polygonmodeller och med avsevärt högre bildfrekvens.

Testningen av vår teknik för övergången mellan billboards och polygonmodell visade, som antagits, att denna knappast märks om den sker på längre avstånd från kameran. Den svaga diskontinuitet, som kunde märkas, om man tittade efter extra noga, beror på skillnader i utseendet mellan träden visualiserade som billboard och polygonmodell, vilka uppkom vid skapandet av våra billboards i programmet Xfrog. Under övergången avtog, som antagits, bildfrekvensen från ett högsta värde vid dess början till ett betydligt mindre och dessutom lägsta värde vid dess slut. Vidare visade sig avtagandet vara monotont. Övergången visade sig således kunna fortskrida med acceptabelt sjunkande bildfrekvens, ty därvid underskreds ju aldrig dess slutvärde.

Vi anser att den vegetation vi visualiserat i våra tester har minst lika god visuell kvalitet som den i Serious Sam: The second encounter. Däremot har vi inte riktigt lyckats komma i nivå med vad Blueberry3D och Codecreatures kan prestera, men detta anser vi beror på de begränsningar som Crystal Space sätter.

6.7 Förbättringar

Beträffande utplaceringen av objekt skulle det innebära en förbättring av vår implementation om man med den kunnat placera ut flera än en objekttyp inom ett angivet område. För varje typ skulle kunna anges antal individer att placera ut och ett minimumavstånd mellan dem inbördes. Dessutom behövs angivande av ett minsta avstånd mellan alla utplacerade individer av olika typ i området. För att åstadkomma detta måste bl. a. metoden, som beräknar utplaceringspositioner ändras så att samtliga de positioner, som den returnerar vid ett antal på varandra följande anrop, har ett angivet minsta avstånd till varandra. Detta skulle kunna göras på följande sätt:

Som minimumavstånd mellan samtliga beräknade utplaceringspositioner, här kallat min_total, kan väljas det minsta bland minimumavstånden för alla de objekttyper, som ska placeras ut inom utplaceringsområdet. Innan beräkningen av placeringspositioner börjar måste två kontroller göras av antalen objekt att utplacera :

- För varje objekttyp kontrolleras att det angivna placeringsantalet, med tillhörande minimumavstånd, får plats inom utplaceringsområdet. Om inte minskas antalet så att det får plats.

- Efter kontroll och ev. antalsnedsättning, enligt punkten ovan, summeras samtliga objektantal att utplacera. Sedan undersöks om den beräknade totalsumman, sum_total, objekt får plats inom området förutsatt minsta avståndet min_total mellan dem. Är inte så fallet beräknas maximala antalet objekt, som får plats med min_total mellan sig. Detta antal, g_sum, fördelas sedan på objekttyperna, som ska placeras ut, i proportion till vars och ens andel av sum_total.

Kontrollerna och beräkningarna i de två punkterna ovan tas omhand av en ny särskild metod, vilken anropas en gång som förberedelse till beräkningarna av samtliga utplaceringspositioner. Dessa beräkningar kan för varje objekttyp göras så här:

En viss metod anropas en gång och returnerar en vektor, veg_data, med samtliga utplaceringspositioner för objekttypen. Beträffande den objekttyp, vars utplaceringspositioner blir de första att beräkna, fungerar metoden som den ovan beskrivna avsnitt 5.2.1 med tillägget att de returnerade utplaceringspositionerna lagras i en vektor, g_data, med kapacitet att rymma g_sum objekt. Meningen med lagringen i g_data är att de beräknade placeringspositionerna vid kommande anrop för ytterligare objekttyper ska kontrolleras så att de håller avståndet min_total till de beräknade vid tidigare anrop. Vid anropen för de övriga objekttyperna utförs i metoden nämnda kontroll samtidigt som då också kontrolleras att minimumavståndet för objekten inbördes av den aktuella typen upprätthålles. Alla positioner, som klarar båda kontrollerna, lagras fortlöpande i både g_data och vektorn veg_data.

7. Demoapplikation

I en demoapplikation har vi sammanställt exempel på skillnaderna mellan de olika tekniker för visualiserandet av växtlighet som vi testat. Demon syftar till att ge en uppfattning om vad de tekniker vi valt att studera och implementera kan ha för användningsområde i en terrängmiljö. Teknikerna demonstreras var och en för sig, men även hur de kan användas tillsammans för att skapa en skog av en eller flera trädarter. I samband därmed visas effekter som en skydome, solblänk, moln och regn.

För att åskådliggöra billboardstekniker har vi placerat ut två olika växter (träd), det ena lite mer tätbevuxet, och det andra lite glesare. Vidare är varje träd representerat av fyra olika billboardstekniker. Dessa tekniker är en enkel roterande bildskiva, två, fyra och sex bildskivor satta i kors (se kapitel 6.3, figur 30, 31).

Vi visar även i vår demoapplikation enskilda objekt representerade som polygonmodell och billboardsmodell. Detta för att ge betraktaren en möjlighet att jämföra de olika teknikerna och bilda sig en uppfattning om de visuella skillnaderna mellan dem.

Vidare demonstreras tekniken vi implementerat för övergång mellan polygon- och billbordsmodell av ett trädobjekt. Detta visas med hjälp av två olika träd. Träden har skapats med hjälp av ett speciellt program för modellering av växter. Då träden modellerats färdigt togs en skärmdump, träden klipptes ut och sparades som bilder, för att sedan fungera som texturer för billboardrepresentation. För att övergången ska bli så lite märkbar som möjligt roteras både billboard och polygonobjektet i förhållande till betraktaren. Detta gör att vi endast behöver en billboardbild och att det blir betraktaren som *planterar* polygonmodellen, då övergången är färdig.

För att sedan kunna ge betraktaren en möjlighet att se värdet med att ha en bländning mellan polygon- och billboardsmodell, har vi även placerat ut ett träd som på ett långt avstånd är en billboardsmodell. Men då betraktaren kommer närmare skiftar vi direkt över till polygonmodellen.

På en sluttning finns det vidare en skog av billboardträd representerade med en enkel roterande bildskiva. Skogen är skapad med hjälp av den utplaceringsteknik som finns implementerad i vårt plugin. Träden är slumpvis utplacerade med hänsyn till minsta inbördes avstånd och är även roterade slumpvis i förhållande till varandra.

Figur nr. 33: Skog av billboardträd

Sist i vår demo visas en blandning av de olika teknikerna i form av ytterligare en grupp träd med inslag av lägre växtlighet, t.ex. en buske eller en grästuva representerad med en enkel roterande bildskiva. Gruppens träd är på långt avstånd enkla roterande bildskivor, men blandas över till polygonmodeller då betraktaren närmar sig dem.

Referenser

Litteratur

[1] Mark DeLoura, Game Programming Gems, ISBN 1584500492, Charles River Media, August 2000.

[2] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, Steven Worley, Texturing and Modeling, ISBN 0122287304, AP professional, 2:a upplagan, 1998.

[3] Przemyslaw Prusinkiewics, Aristid Lindenmayer, The Algorithmic Beauty Of Plants, ISBN 3540972978, Springer Verlag, 1990.

[4] Alan Watt, 3D Computer Graphics, ISBN 0 201 39855 9, Adison-Wesley, 3:e upplagan, 2000.

Internetmateriel

[5] Gavin Bell, Creating Backgrounds, http://www.gamasutra.com/features/visual_arts/19981023/skybox_02.htm, 2002-05-03.

[6] Willem H. de Boer, Fast Terrain Rendering Using Geometrical MipMapping, http://www.flipcode.com/tutorials/geomipmaps.pdf, 2002-02-20.

[7] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, Mark B. Mineev-Weinstein, ROAMing Terrain: Real-time Optimally Adapting Meshes, http://www.llnl.gov/graphics/ROAM/roam.pdf, 2002-02-22.

[8] Hugo Elias, Perlin Noise, http://freespace.virgin.net/hugo.elias/models/m_perlin.htm, 2002-03-24.

[9] António Ramires Fernandes, Terrain Tutorial, http://www.lighthouse3d.com/opengl/terrain/index.php3?introduction, 2002-03-27.

[10] Nick Foster, Ronald Fedkiw, Practical Animation of Liquids, http://graphics.stanford.edu/~fedkiw/papers/stanford2001-02.pdf, 2002-06-03.

[11] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust, Gregory A. Turner, Real-Time Continuous Level of Detail Rendering of Height Fields, http://www.gvu.gatech.edu/people/peter.lindstrom/papers/siggraph96/siggraph96.pdf, 2002-02-23.

[12] Bernd Linterman, Oliver Deussen, Interactive Modeling of Plants, http://www.inf.tu-dresden.de/ST2/cg/mitarbeiter/sub_frames/mitarbeiter-Dateien/deussen/cga_plants.pdf, 2002-04-29.

[13] Simon Premože, Michael Ashikhmin, Rendering Natural Waters, http://www.cs.utah.edu/vissim/papers/water/waterColorPG.pdf, 2002-06-03.

[14] Mattias Widmark, Procedural Geometry for Real-time Terrain Visualisation in Blue-Berry3D, http://www.nada.kth.se/~kai/SIGRAD/SIGRAD2001/Widmark.pdf, 2002-02-20.

[15] Andreas Ögren, Continuous Level of Detail In Real-Time Terrain Rendering, Master's Thesis, http://www.cs.umu.se/~tdv94aog/ROAM.pdf, 2002-03-19.

Bildmateriel

[16] http://www.lighthouse3d.com/opengl/terrain/index.php3?fault

[17] http://zeus.fri.unilj.si/~aleks/slicing-and-blending/

[18] http://www.flipcode.com/tutorials/tut_skydomes.shtml

[19] http://www.gamasutra.com/features/visual_arts/19981023/skybox_02.htm

Bilaga A: Pseudokod med kommentarer

Följande är ett utdrag från boken Texturering & Modeling [2]:

```
      double HybridMultifractal(Vector point, double H, double lacunarity,
                                double octaves, double offset)
      {
            double frequency, result, signal, weight, remainder;
            double Noise3(Vector v);
            int i;
            static int first = TRUE;
            static double *exponent_array;

      /* precompute and store spectral spectral weights */

      if (first) {
            exponent_array = ( double * ) malloc ( ( octaves + 1 ) * sizeof
                             ( double ) );
            frequency = 1.0 ;

            /* compute weight for each  frequency */
            for ( i = 0 ; i < octaves ; i++ ) {
                  exponent_array[ i ] = pow( frequency, -H );
                  frequency *= lacunarity;
            }

            first = FALSE ;
      }

      /* get first octave of function */

      result = ( Noise3( point ) + offset ) * exponent_array[0];
      weight = result ;

      /* increase frequency */

      point . x *= lacunarity ;
      point . z *= lacunarity ;
      point . y *= lacunarity ;

      /* spectral construction inner loop, where the fractal is built */

(I loopen dämpas högre frekvenser kraftigt i områden med elevation nära
noll, vilket bevarar dem jämna. Inom områden med högre elevation avtar
dämpningen och de växer till i skrovlighet med ökande antal iterationer.)

      for ( i = 1 ; i < octaves ; i++ ) {

            /* prevent divergence */
            if ( weight > 1.0 )
                  weight = 1.0;

            /* get next higher frequency */
            signal = ( Noise3( point ) + offset ) * exponent_array[ i ] ;
            result += weight * signal ;

            /* update the monotonically decreasing weighting value */
            weight *= signal ;
```

```
        /* increase frequency */
        point . x *= lacunarity ;
        point . z *= lacunarity ;
        point . y *= lacunarity ;

    } /* for */

    /* take care of remainder in " octaves " */
    remainder = octaves - ( int ) octaves ;
    if (remainder)
        /* " i " and spatial frequency are preset in loop above */
        result += remainder * Noise3( point ) * exponent_array[ i ] ;

    return  result ;

    } /* HybridMultifractal( ) */
```

Procedurens parametrar :

- point är punkten vars höjdvärde = result beräknas.

- H avgör den fraktala dimensionen för de skrovligast genererade områdena i kartan, den fyller motsvarande funktion som variabeln r i formeln (avsnitt 3.1.2):

$$d(i) = d(i-1) * 2^{-r} \ ,$$

  H = 0.25 är ett lämpligt värde för att åstadkomma naturtrogna landskap i stil med de som beskrivs i avsnitt 3.1.5.

- lacunarity fyller motsvarande funktion som 2:an i formeln strax ovan och sättes lämpligen = 2.

- octaves anger antalet använda frekvenser i kodens beräkningar och beräknas lämpligen i sin tur så här octaves = $\log_2$(skärmens upplösning) – 2, vilket torde ge ett värde i intervallet 6 - 10.

- offset bestämmer resultatets heterogenitet i ovan nämnd mening. Värden nära noll ger hög heterogenitet, som med växande värden avtar till monofraktalitet, som, till sist vid c:a 100, slutar med en plan yta. Lämpligt värde är c:a 0.7 [2].

Funktionen Noise3 (Vector v) är slumptalsgeneratorn för perlinbrus.